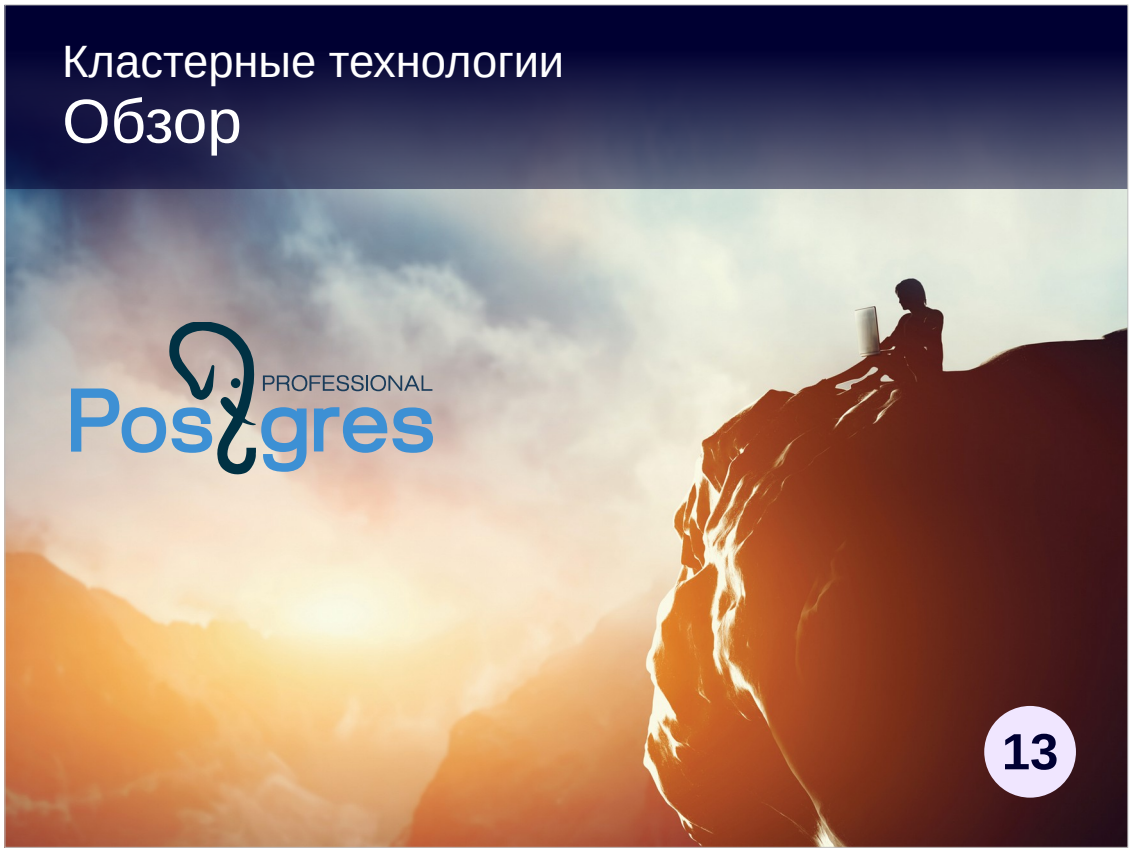


Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

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

Темы

Ожидания от кластера

Средства реализации

Решения с реализацией внутри PostgreSQL

Решения с внешними системами управления

Ожидания от кластера

Высокая доступность системы

- в том числе отказоустойчивость

Масштабируемость

Согласованность данных

Высокая доступность

Обеспечение минимального времени простоя системы

вызванного как сбоями, так и плановыми работами

Различные характеристики

отношение времени работы системы к общему времени → 100 %
время наработки на отказ → ∞
RTO — целевое время восстановления → 0
RPO — целевая точка восстановления → 0
и другие

Доступность системы — отношение времени, в течении которого система сохраняет доступность, к общему времени эксплуатации.

Понятие доступности более широкое, чем отказоустойчивость, поскольку простои могут быть вызваны не только отказами, но и плановыми работами.

Часто говорят о доступности

- «одна девятка» 90 % — 36,5 дней простоя в год,

- «две девятки» 99 % — 3,65 дней простоя в год,

- «три девятки» 99,9 % — 8,76 часов простоя в год,

- «четыре девятки» 99,99 % — 52,56 минут простоя в год,

- «пять девяток» 99,999 % — 5,26 минут простоя в год, и т. д.

Высокая доступность предполагает цифры, близкие к 100 %.

Для примера: облачный сервис для реляционных СУБД Amazon RDS обещает (https://aws.amazon.com/rds/sla/) доступность узлов на уровне 99,95 %.

Другими характеристиками доступности могут быть:

- время наработки на отказ;

- целевое время восстановления (Recovery Time Objective) — время, за которое система должна восстановиться;

- целевая точка восстановления (Recovery Point Objective) — период, за который можно потерять данные при сбое) и др.

Масштабируемость

Масштабируемость

изменение характеристики системы при одновременном увеличении объема данных, нагрузки и числа узлов в *N* раз

идеальная *масштабируемость времени отклика* — время отклика не должно увеличиваться

идеальная *масштабируемость пропускной способности* — число обрабатываемых запросов должно расти линейно

Ускорение

отношение времени отклика на одном узле ко времени отклика в кластере из *N* узлов

Важным понятием для кластера является *масштабируемость:* это свойство показывает, как меняется какая-либо характеристика системы с увеличением числа узлов, объема данных и нагрузки.

Масштабируемость времени отклика показывает, как меняется время отклика (в контексте СУБД — время выполнения запроса) с ростом объема данных и числа запросов при добавлении узлов кластера. В идеале время отклика не должно увеличиваться.

Можно говорить не только о масштабируемости времени отклика, но и об *ускорении,* которое показывает, как при той же нагрузке и том же объеме данных время отклика меняется с увеличением числа узлов. Эта характеристика особенно важна для OLTP-систем.

Масштабируемость пропускной способности показывает, как меняется число обрабатываемых запросов с ростом объема данных при добавлении узлов кластера. В идеале число обрабатываемых запросов должно увеличиваться линейно с ростом числа узлов.

Идеальная масштабируемость (увы, не достижимая на практике) позволяет решать проблемы роста простым добавлением новых узлов в кластер.

Согласованность

Локальные транзакции

- практически никаких гарантий при переключении между узлами
- согласованность обеспечивает приложение

Глобальные (распределенные) транзакции

- ACID-согласованность
- проблема атомарности транзакций

Чем более строгие гарантии согласованности данных предоставляет кластер, тем больше он похож на «единой целое», тем проще с ним работать, тем меньше надо учитывать особенности в прикладном коде. Но тем дороже это обходится в смысле производительности.

Если кластер основан на схеме «мастер-реплика» с одним пишущим узлом, и транзакции выполняются локально на одном из узлов, гарантии могут оказаться очень слабыми. Например, один узел мог успеть применить запись о фиксации транзакции и ее результат уже доступен клиентам, а другой узел — еще нет. В таком случае у клиента, в частности, нет гарантии, что он не прочитает старое значение (на одном узле) уже после того, как прочитал новое (на другом узле). Обработка таких ситуаций ложится на приложение, которое должно быть рассчитано на работу в этой конфигурации.

Если кластер предоставляет *глобальные (распределенные) транзакции,* то можно говорить о согласованности в обычном смысле ACID: глобальная транзакция должна переводить базу данных из одного согласованного состояния в другое согласованное (C), при условии, что транзакция полностью выполняется на всех узлах (A) при отсутствии помех со стороны других конкурентных транзакций (I).

В этом случае требуется обеспечить *атомарность* транзакций: все узлы кластера должны выполнить одинаковое действие — либо применить, либо отменить транзакцию. PostgreSQL предоставляет для этого относительно простой протокол двухфазной фиксации (2PC), который, однако, не устойчив к сбоям.

Средства реализации

Обеспечение высокой доступности

Виды сбоев и их обнаружение

Распределенные протоколы консенсуса

Обеспечение масштабируемости

Высокая доступность

Плановые работы без прерывания обслуживания

- средствами PostgreSQL
- временный вывод узла из кластера

Дублирование компонентов, исключение точек отказа

- серверы PostgreSQL (и другие необходимые системы)
- сетевое оборудование
- электропитание

Обнаружение сбоев и управление восстановлением

Время простоя может быть вызвано не только возникающими неполадками, но и плановыми работами, которые невозможно выполнить без прерывания обслуживания. С каждой версией PostgreSQL уменьшается число действий, требующий перезагрузки сервера. Кроме того, кластер должен уметь отрабатывать плановый останов одного из серверов с минимальными проблемами для пользователей.

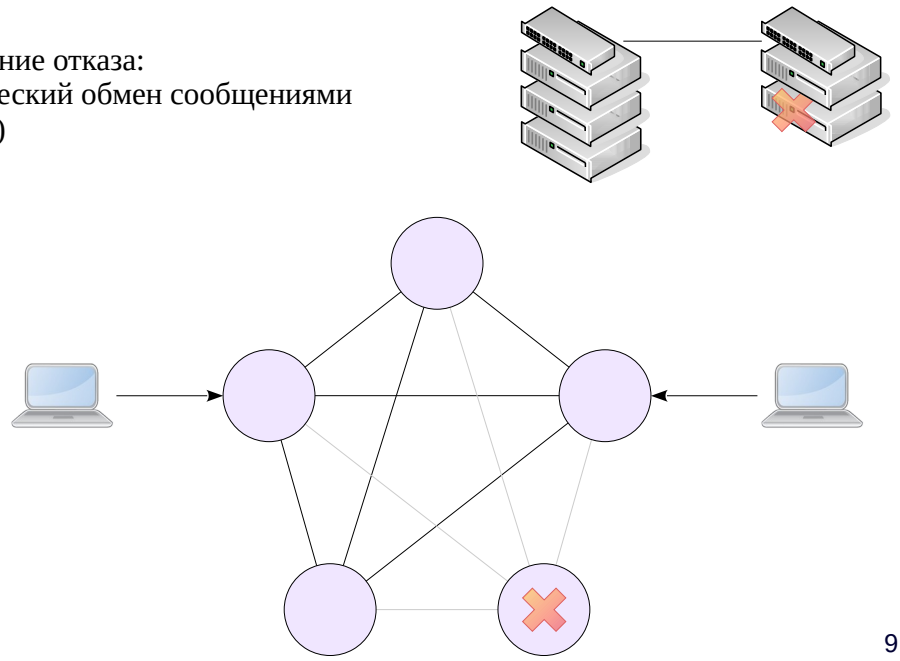
Основа обеспечения отказоустойчивости — дублирование всех узлов системы. Речь не только о серверах баз данных, но и о сетевом оборудовании, обеспечении электропитания и пр.

Разумеется, недостаточно просто дублировать компоненты системы. Надо обеспечить обнаружение сбоев в работе системы и их обработку — корректное переключение на резервные узлы и возвращение узлов в строй после устранения неисправности.

Большая часть необходимых средств отсутствует в стандартном PostgreSQL или вообще относится к другим компонентам. В целом, чем более доступной должна быть система, тем сложнее она будет устроена.

Отказ узла

обнаружение отказа:
периодический обмен сообщениями
(heartbeat)



Чтобы говорить об обнаружении и управлении сбоями, нужно определиться, что считать сбоем. С точки зрения программного обеспечения кластера обычно рассматривают два вида сбоев: *отказ узла* и *разделение сети* (network partitioning).

При отказе узла этот узел перестает функционировать как часть кластера. Отказ может быть вызван остановом или сбоем сервера СУБД, сбоем операционной системы, выключением сервера и т. п.

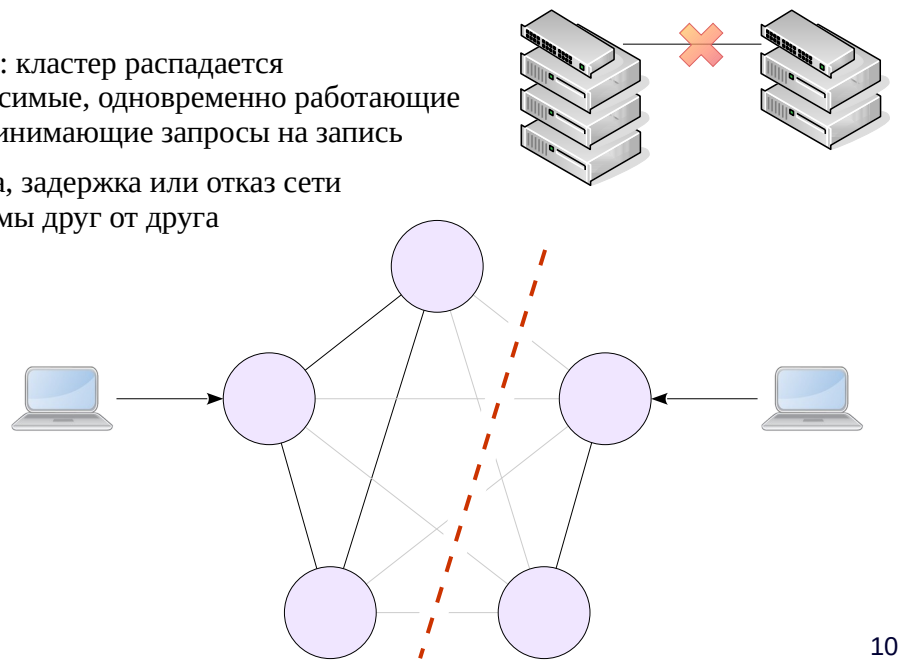
Для обнаружения сбоев узлы кластера периодически обмениваются короткими сообщениями (heartbeat). Если какой-либо из узлов не отвечает в течении определенного времени, фиксируется и отрабатывается сбой этого узла.

Процедура отработки отказа зависит от архитектуры кластера. Например, в кластере, построенном на схеме «мастер-реплика», при сбое мастера какая-то из реплик должна занять его место (должно произойти переключение ролей).

Разделение сети

split-brain: кластер распадается
на независимые, одновременно работающие
части, принимающие запросы на запись

отказ узла, задержка или отказ сети
неотличимы друг от друга



Разделение сети происходит при сбое сетевого оборудования. При этом все узлы могут сохранить работоспособность, но одна группа теряет связь с другой группой узлов кластера.

Самая неприятная ситуация возникает в случае, когда обе группы сохраняют связь с клиентами. Это может привести к проблеме split-brain: кластер распадается на две части, каждая из которых начинает работать автономно и при этом принимать запросы на запись. Возникает рассогласование данных, и при восстановлении данные одной из групп будут вынуждено потеряны.

Заметим, что сеть может выйти из строя, а может временно начать терять пакеты или доставлять их с задержкой (более того, в некоторых ОС обычная загруженность процессора тоже может приводить к сетевым задержкам). Все эти случаи неотличимы от сбоя узла: если от узла вовремя не поступит подтверждение, кластер должен считать такую ситуацию сбоем. Узел может сохранять работоспособность или нет, но узнать это невозможно, если с ним нет связи.

Многие системы будут нестабильно работать в условиях ненадежного сетевого подключения. Чтобы уменьшить вероятность ложных срабатываний, можно увеличить пороговое значение, но тогда увеличится и время обнаружения настоящего сбоя, а это приведет к снижению доступности. Для примера: etcd предлагает устанавливать порог в 10 раз выше, чем время нормального отклика узла.

https://github.com/coreos/etcd

Проблема консенсуса

Узлы кластера должны уметь приходить к общему мнению

- общее представление о текущем состоянии всех узлов кластера
- атомарная фиксация глобальных транзакций

Консенсус через обмен сообщениями

- нет общей памяти (shared nothing)
- используются специальные распределенные протоколы, учитывающие возможность сбоев на каждом этапе переговоров
- построение, доказательство корректности и тестирование реализации — сложная задача!

Кластер, составленный из независимых узлов, должен работать — в некотором смысле — как единое целое. Чтобы управлять сбоями, узлы должны разделять общее представление о состоянии всех узлов кластера. Чтобы принять решение о фиксации глобальной транзакции, все узлы кластера должны согласиться, что это возможно, и в итоге выполнить фиксацию атомарно.

Поскольку никакого общего устройства хранения у узлов нет (мы рассматриваем системы класса shared nothing), то единственный способ прийти к общему мнению — договориться, используя какой-либо *протокол распределенного консенсуса.* Протокол определяет, какими сообщениями и в каком порядке должны обменяться узлы, чтобы прийти к соглашению.

Важный момент: распределенные алгоритмы *очень сложны,* поскольку учитывают множество граничных случаев. Например, они должны корректно работать в ситуациях, когда на любом этапе переговоров происходит какой-либо сбой. Доказательство корректности таких алгоритмов и тестирование их реализаций — также очень сложная задача. Поэтому не стоит пытаться создать собственный алгоритм или доверять неизвестному.

Протоколы 2PC, 3PC

Двухфазная фиксация

prepare: координатор проводит голосование;
commit: если все «за», координатор сообщает о фиксации.

поддерживается PostgreSQL (PREPARE TRANSACTION + COMMIT)

Трехфазная фиксация

prepare: координатор проводит голосование;
precommit: если все «за», координатор сообщает всем о намерении;
commit: если все успешно, координатор сообщает о фиксации.

Свойства

возможны ситуации (например, разделение сети),
в которых невозможно разрешить транзакцию до устранения сбоя

Простой протокол для фиксации глобальных транзакций, поддержка которого встроена в PostgreSQL, — двухфазная фиксация (2-Phase Commit). Узел, выполняющий транзакцию, считается координатором. На подготовительной фазе (prepare) координатор сообщает остальным узлам о фиксации транзакции и дожидается от них подтверждения, что они готовы выполнить фиксацию (если хотя бы один узел не готов — транзакция обрывается). На фазе фиксации (commit) координатор сообщает всем узлам о решении зафиксировать транзакцию. При получении от всех подтверждения, координатор тоже фиксирует транзакцию.

Этот протокол позволяет обойтись минимумом сообщений, но не устойчив к сбоям. Например, при отказе координатора после фазы prepare, остальные узлы не имеют информации о том, должна ли транзакция быть зафиксирована или отменена. Им придется ждать устранения сбоя.

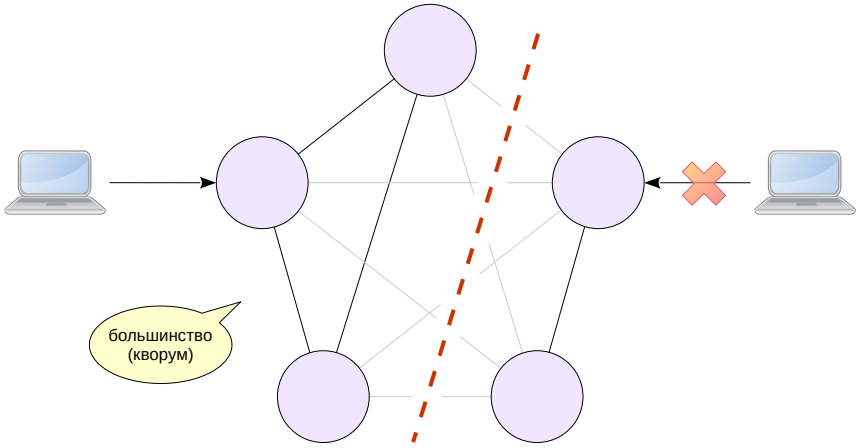
Это ограничение преодолевает более сложный протокол трехфазной фиксации (3-Phase Commit). В нем вводится дополнительная фаза precommit, на которой координатор сообщает всем узлам о принятом решении фиксировать транзакцию. После этого, даже если произойдет сбой координатора, оставшиеся узлы будут иметь право зафиксировать транзакцию самостоятельно (а до фазы precommit — самостоятельно отменить ее).

Однако оба алгоритма не справляются с ситуациями разделения сети.

Кворумные протоколы

Paxos, Raft, ZAB, E3PC и другие

при отказе до *N*/2–1 узлов связное большинство продолжает работу
меньшинство блокируется до устранения сбоя



13

Алгоритмы, основанные на *кворуме* (большинстве голосов), позволяют в условиях разделения сети не блокировать узлы, сохранившие связь друг с другом и образующие кворум. Таким образом (если голоса всех узлов равны), из *N* узлов *N*/2+1 смогут продолжить работу.

Узлы, оставшиеся в меньшинстве, вынуждены ожидать исправления сбоя, чтобы принять решение о фиксации или откате открытых транзакций. Такие узлы могут полностью не обслуживать клиентов или как минимум не принимать запросы на запись, чтобы не допустить ситуацию split-brain.

Существует ряд алгоритмов, доказанных математически и имеющих протестированные и проверенные временем реализации.

- Paxos опубликован в 1998 году, а фактически появился в конце 80-х. Есть целое семейство этих алгоритмов для различных ситуаций.

- Raft создавался как алгоритм, эквивалентный Paxos, но более простой для понимания. В этом алгоритме один из узлов выбирается в качестве лидера, который может изменять данные; остальные узлы читающие. Этот протокол лежит в основе таких систем, как etcd и Consul (предоставляющих распределенное хранилище «ключ-значение»).

- ZAB используется в системе ZooKeeper.

- E3PC - Enhanced 3PC улучшает алгоритм трехфазной фиксации, разрешая фиксировать транзакцию при получении большинства (а не всех) голосов.

Существуют иные подходы, позволяющие решать проблему консенсуса, например, Virtual synchrony (используется в системе Corosync).

Масштабируемость

Пропускная способность по чтению

- распределение нагрузки по узлам кластера

Пропускная способность по записи

- не достигается при полной репликации данных на все узлы
- шардинг — распределение строк таблиц по разным узлам, сильно зависит от природы хранимых данных

Время отклика

- параллельные вычисления на нескольких узлах

Проще всего достичь масштабирования пропускной способности по отношению к читающим транзакциям. Для этого достаточно каким-то образом распределять читающие транзакции более или менее равномерно по узлам кластера.

Достичь масштабирования пишущей нагрузки обычно сложнее: в конфигурациях типа «мастер-мастер» узлы вынуждены обмениваться друг с другом сообщениями и данными, так что общая нагрузка на систему с увеличением числа узлов только возрастает. Однако это возможно, если данные распределятся между узлами, а не реплицируются на каждый узел полностью. Обычно для этого применяется *шардинг:* распределение происходит на уровне строк таблиц. Можно провести аналогию с секционированием, в котором секции расположены на разных узлах кластера.

Заметим, что возможные схемы шардирования очень сильно связаны с логикой данных; какое-то универсальное решение тут невозможно.

Использование шардинга позволяет достичь и масштабирования времени отклика. Для этого кластер должен уметь распараллеливать запросы, выполняя их одновременно на нескольких узлах, имеющих необходимые данные. Это похоже на параллельное выполнение запросов в стандартном PostgreSQL, но параллельные процессы выполняются не внутри одного сервера, а на разных узлах. Конечно, при этом возрастает стоимость и пересылки данных (например, результатов частичной агрегации) между процессами.

Распределение нагрузки

Средства, специфичные для PostgreSQL

- библиотека libpq
- PgBouncer — менеджер пула
- Pgpool-II — балансировщик и менеджер пула
- …

Другие средства

- виртуальный IP-адрес
- HAProxy — балансировщик TCP/HTTP
- round-robin DNS
- …

Для распределения нагрузки уже имеется достаточно много готовых решений; обычно такие средства не входят в состав самого кластера.

- Библиотека libpq позволяет указать в строке подключения несколько узлов (перебираются по порядку) и требования у узлу (должен ли принимать пишущие транзакции).

- PgBouncer — удобный менеджер пула соединений для PostgreSQL.

- Pgpool-II — менеджер пула и балансировщик, анализирующий SQL-запросы и направляющий пишущую нагрузку на мастер, а читающую на реплики.

Из общих средств, не связанных с PostgreSQL, отметим:

- *Виртуальный IP-адрес* позволяет перенаправлять сетевые пакеты на один из реальных IP-адресов узлов кластера, например, к текущему мастеру.

- HAProxy — высокопроизводительный балансировщик и прокси-сервер, работающий на уровне TCP и HTTP. Для периодического обновления списка узлов можно использовать средство управления конфигурациями confd. Другой вариант — использовать возможность самого HAProxy выполнять периодические проверки (health check) к узлам кластера для исключения неподходящих из списка балансировки.

- Некоторые системы (Consul), предоставляют *round-robin DNS*: выбор узлов и балансировка может происходить на этапе разрешения имени (но кеширование сводит на нет преимущества такого решения).

Есть и другие средства. Многие можно использовать совместно.

«Простые» решения

Расширение обычного PostgreSQL

- не требуется высокая доступность или согласованность

Примеры: Postgres-BDR, Citus

Компоненты

Обычный PostgreSQL с установленным расширением

Точка входа

способ направить клиентов на нужный узел кластера

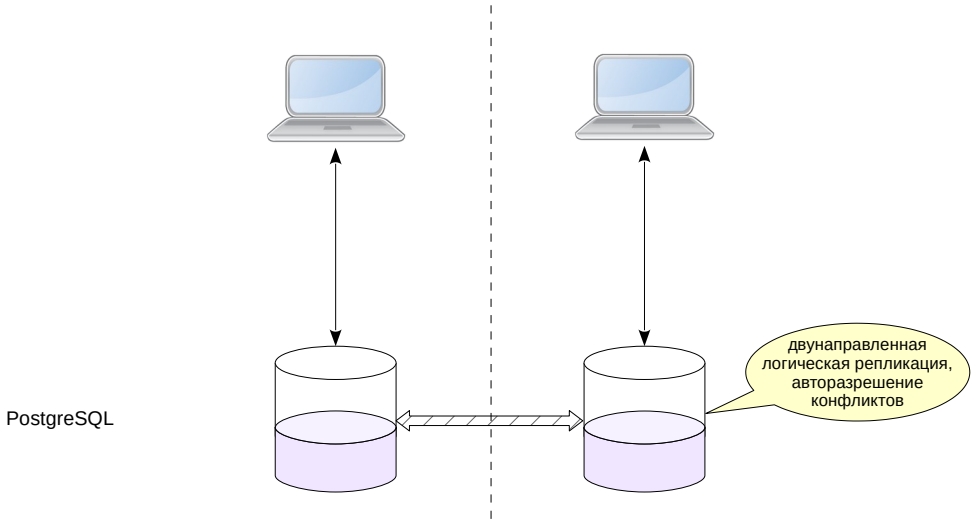
В случаях, когда от кластера не требуется высокой доступности и можно пренебречь вероятностью сбоя, или нет серьезных требований к согласованности данных, можно ограничиться «простым» кластером, составленным из **обычных экземпляров PostgreSQL.** Если в таком кластере и реализованы глобальные транзакции, то используется стандартный протокол 2PC, неустойчивый к сбоям.

Слово «простой» взято в кавычки: такие системы могут, например, не обеспечивая отказоустойчивость, в то же время предоставлять богатые возможности по масштабированию и т. п.

Если подключение допускают несколько узлов кластера, то для клиентов обычно требуется единая **точка входа,** которая направит их на нужный узел.

Преимущество такого подхода — в простоте и дешевизне решения.

Пример: Postgres-BDR



Расширение Postgres-BDR (EDB) добавляет возможность двусторонней логической репликации с автоматическим разрешением конфликтов (используя один из готовых алгоритмов или свой собственный).

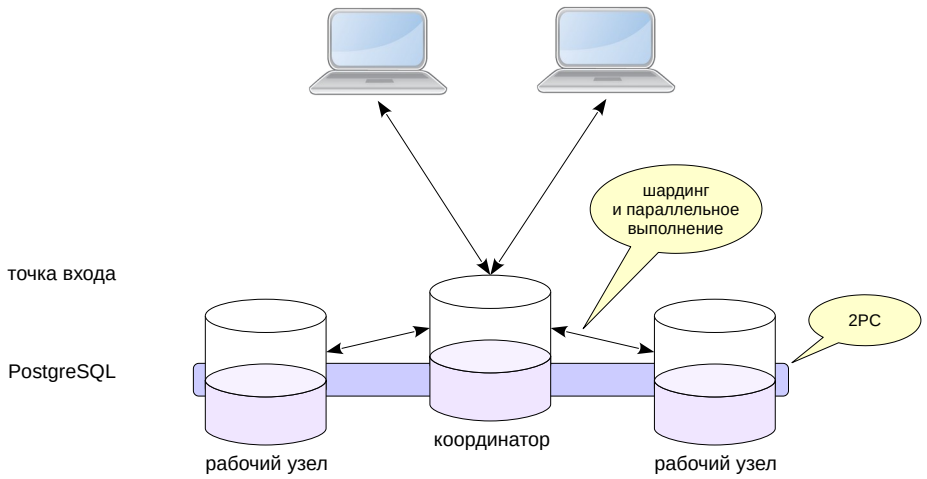
https://www.enterprisedb.com/products/bidirectional-replication-bdr-postgresql-database

В такой схеме получаем два или более *полностью независимых* друг от друга узла. Из-за возможности конфликтов нет гарантии, что все узлы кластера будут содержать одинаковую информацию. Даже если конфликты отсутствуют, для приложения нет гарантий согласованности данных при чтении с разных узлов.

Зато — полностью отказавшись от согласованности — кластер оказывается устойчивым к любым сбоям. Например, при отказе сети узлы кластера продолжают функционировать отдельно друг от друга, а при появлении связи обмениваются накопившимися данными (разрешая возникающие конфликты — возможно, разным образом на разных узлах). Это, в частности, позволяет строить георасределенные кластеры, не неся дополнительных затрат на фиксацию глобальных транзакций.

(Альтернативный режим Eager Replication обеспечивает согласованность, задействуя механизм консенсуса).

Пример: Citus



Citus (Citus Data, https://www.citusdata.com/) — расширение PostgreSQL для распределения данных и запросов по узлам кластера. Имеется Community-версия с открытым исходным кодом.

Citus — пример системы, в которой данные не реплицируются полностью, а шардируются на рабочие узлы. Один узел выделяется в качестве координатора — он содержит метаинформацию о расположении данных.

В состав расширения входит специальный оптимизатор запросов, генерирующий параллельные планы выполнения, а исполнитель выполняет запросы, распределяя их на соответствующие узлы. За счет этого кластер Citus позволяет достичь масштабируемости по времени отклика и масштабируемости пишущей нагрузки.

Такой кластер нельзя назвать высокодоступным. Основной точкой отказа здесь является координатор: чтобы увеличить отказоустойчивость, документация предлагает использовать обычную физическую реплику. Отказ рабочих узлов также можно компенсировать физической репликацией или реплицировать шарды на несколько узлов средствами самого Citus.

Citus использует глобальные транзакции для того, чтобы гарантировать атомарность изменений на всех узлах кластера. Используется вариант протокола двухфазной фиксации 2PC.

Похожий функционал имеет система Postgres-XL (https://www.postgres-xl.org/)

«Внутренние» решения

Встраивание (почти всего) необходимого в PostgreSQL

- PostgreSQL с управлением глобальными транзакциями
- точка входа

Примеры: multimaster PostgresPro

Компоненты

PostgreSQL с изменениями в ядре

- управление глобальными транзакциями
- обнаружение сбоев

Точка входа

- способ направить клиентов на нужный узел кластера

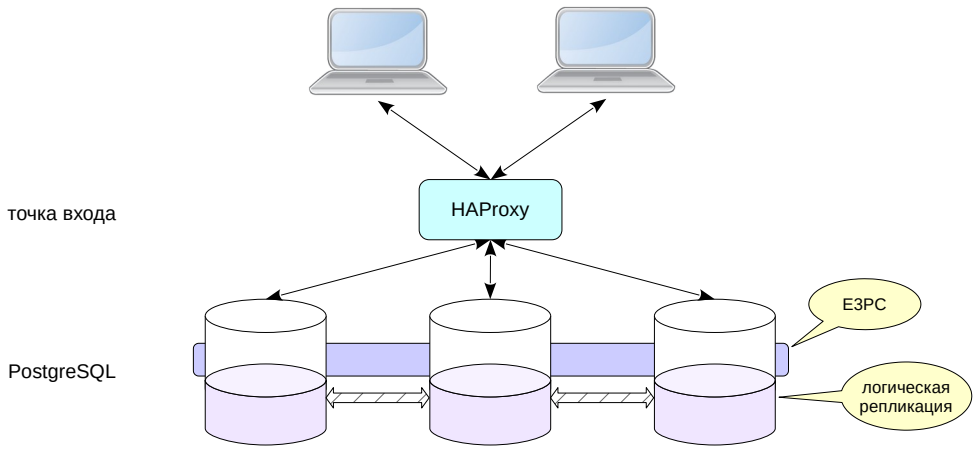
Идея подхода состоит в том, чтобы научить сами экземпляры PostgreSQL общаться, используя один из рассмотренных протоколов. Это непростой путь, поскольку приходится встраиваться в имеющуюся инфраструктуру. Из-за этого приходится идти на изменение ядра системы.

В качестве компонентов кластера выступают **экземпляры PostgreSQL**, в которые встроена поддержка глобальных транзакций и система обнаружения сбоев.

Иногда управление глобальными транзакциями (Global Transaction Manager) выделяют как отдельный сервер, тогда и он становится компонентом кластера. Проблема такого решения в том, что для отказоустойчивости необходимо иметь несколько экземпляров GTM, что приводит к увеличению сложности системы.

Для клиентов может потребоваться единая **точка входа,** которая направит их на нужный узел кластера.

Пример: multimaster



22

В расширении multimaster (Postgres Professional) в СУБД встраивается глобальный менеджер транзакций, который расширяет имеющийся двухфазный протокол 2PC до E3PC. Технически для этого на каждом узле запускается фоновый процесс-арбитр; эти процессы берут на себя необходимые дополнительные коммуникации. Используется алгоритм консенсуса Paxos.

https://postgrespro.ru/docs/enterprise/13/multimaster

Кроме того, узлы кластера обмениваются heartbeat-сообщениями, чтобы своевременно выводить из кластера сбойные узлы. Предусмотрена процедура автоматического возвращения узла в кластер после его восстановления от сбоя.

Репликация данных между узлами кластера основана на механизме логической репликации, DDL реплицируется собственным алгоритмом.

Любой из узлов кластера multimaster может принимать запросы как на чтение, так и на запись. Читающие транзакции выполняются локально. Пишущие транзакции выполняются глобально; узел, фиксирующий транзакцию, становится координатором.

Для распределения нагрузки имеет смысл балансировать соединения между узлами кластера. Например, можно указывать разные строки соединений для разных клиентов, а можно воспользоваться HAProxy.

Приложение, работающее на одном сервере, может быть запущено и на multimaster-кластере без дополнительных забот о согласованности данных. Поддерживаются уровни изоляции read committed и repeatable read. Однако, в силу текущих ограничений реализации, некоторые нюансы приложение должно учитывать.

«Внешние» решения

Внешние по отношению к PostgreSQL средства

- агент
- управляющий
- консенсус
- точка входа

Примеры: Stolon, Patroni, Corosync/Pacemaker

Компоненты

Агент

управляет одним экземпляром PostgreSQL

Управляющий

логика управления всем кластером

Консенсус

внешняя система под управлением одного из протоколов консенсуса

поддержка общего состояния для управляющих

Точка входа

способ направить клиентов на нужный узел кластера

В отличие от систем, модифицирующих PostgreSQL, в этом подходе используется стандартный PostgreSQL и набор внешних средств. На текущий момент это наиболее доступный и распространенный способ построения кластеров, но сложность существенно возрастает.

Поскольку PostgreSQL не может сам себя конфигурировать, перезапускать, останавливать и т. п., нужна программа, которая будет этим заниматься. Эту роль выполняет **агент.**

Логика управления кластера сосредоточена в **управляющих.** На этом уровне принимается решение о том, что такой-то узел следует считать вышедшим из строя, о том, какую из реплик следует назначить новым мастером при выходе старого из строя и т. п.

Чтобы разделять общую точку зрения, управляющие опираются на систему, работающую по одному из рассмотренных ранее распределенных протоколов **консенсуса.**

Наконец, для клиентов требуется единая **точка входа,** которая направит их на нужный узел (или узлы) кластера.

В разных кластерных системах эти компоненты могут называться по-разному; некоторые могут объединяться вместе. Если какие-то компоненты отсутствуют, кластер надо дополнять другими решениями.

Ограждение (fencing)

Отключение узла в неизвестном состоянии от клиентов

- из-за разделения сети может возникнуть ситуация split-brain
- нужно не только оградить узел,
  но и не допустить в это время переход на реплику

Механизмы

- отключение питания
- отключение сетевого порта
- завершение и перенаправление клиентских соединений

Поскольку в этом подходе серверы PostgreSQL ничего не знают о том, что работают в кластере, важно уметь отгородить сбойный узел — возможно, работающий, но потерявший связь с кластером — от внешнего мира. Эта процедура называется fencing.

Если этого не сделать, то мастер, оказавшись в результате разделения сети в меньшинстве, может продолжить работу (ситуация split-brain). Полагаться на то, что управляющие, обнаружив сбой, сами остановят сервер PostgreSQL, нельзя: у управляющих может не оказаться связи с агентом, может произойти сбой в агенте и т. п.

Процедура может выполняться по-разному, но не должна рассчитывать на доступность узла по сети. Например:

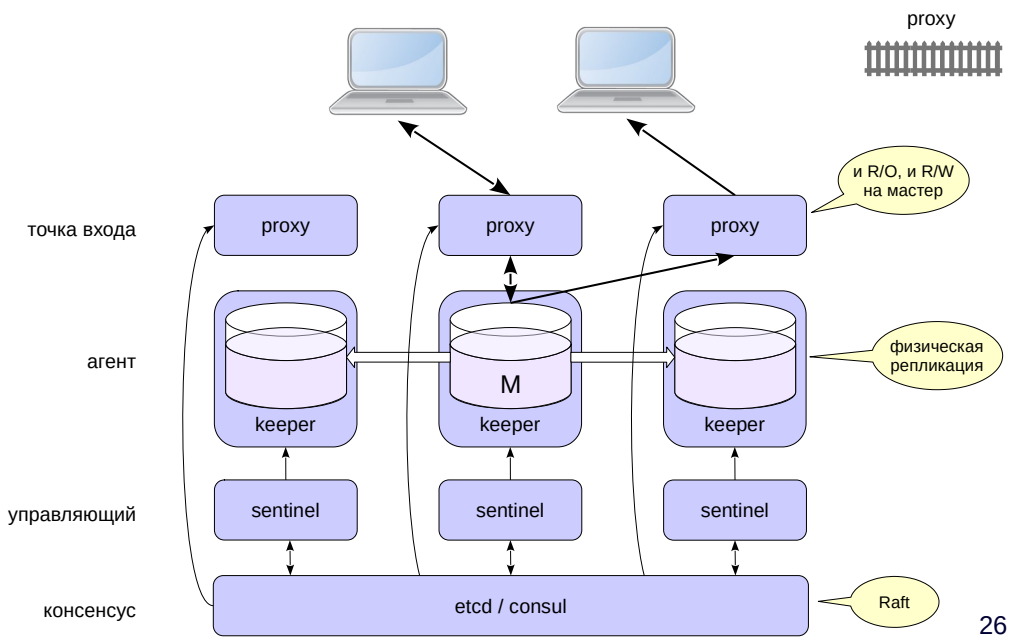
- программно отключать питание сервера (интерфейсы IPMI, PDU);

- программно отключать сетевой порт на коммутаторе;

- если соединение идет через специализированный компонент, то клиенты могут принудительно отключаться от сбойного узла на уровне этого компонента.

Надо заметить, что процедура ограждения уменьшает масштаб проблемы, но сама по себе не решает ее полностью. Нужно также гарантировать невозможность перехода на одну из реплик до тех пор, пока сбойный мастер не будет огражден. Ведь на принятие решения об ограждении требуется время, в течение которого узел может продолжать функционирование, и, следовательно, часть зафиксированных данных может потеряться при восстановлении (нулевое RPO не достигается).

Пример: Stolon



Stolon (SorintLab, https://github.com/sorintlab/stolon) — кластерное решение для высокой доступности с открытым исходным кодом, рассчитанное на облачную инфраструктуру.

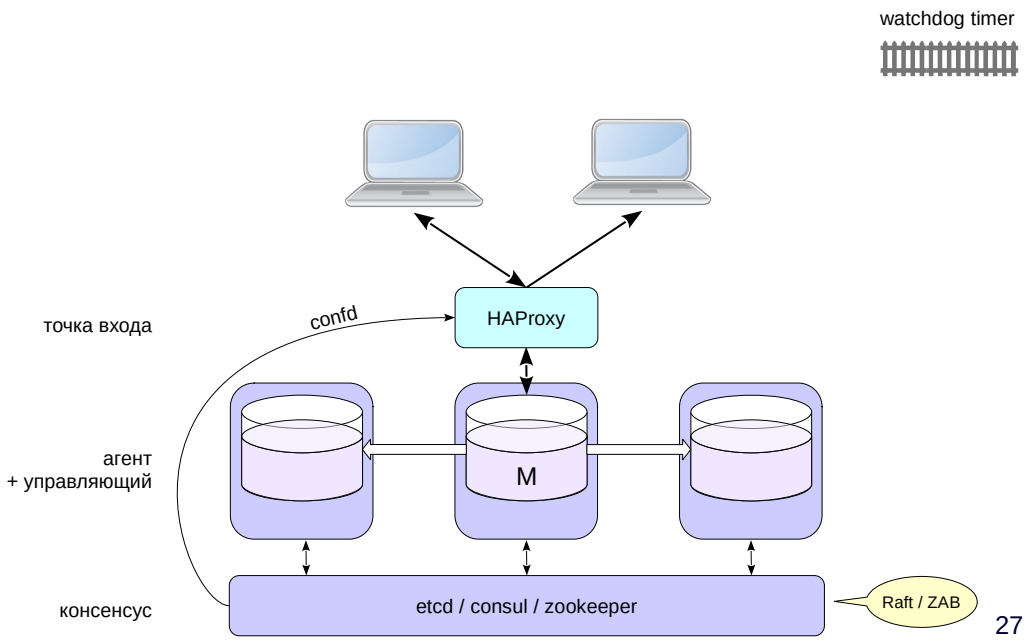
В нем реализованы все компоненты. Агенты в терминологии Stolon называются keeper, управляющие — sentinel, точки входа — proxy. Каждый из компонентов может быть представлен несколькими экземплярами для отказоустойчивости. Обычно (как на иллюстрации) каждый компонент устанавливается на каждый сервер PostgreSQL. Это не обязательно, но удобно и не вызывает дополнительных сложностей в случае разделения сети.

Для консенсуса используется хранилище «ключ-значение» etcd или хранилище, входящее в состав Consul (оба используют протокол Raft).

Дополнительное ограждение (fencing) не предусмотрено, так как proxy получает информацию о состоянии кластера непосредственно из слоя консенсуса и, в случае разделения сети, не пропустит клиентов к оставшимся в меньшинстве узлам.

Как клиентам выбрать из всех proxy тот, к которому подключаться? Для этого нужны дополнительные решения. Например, можно использовать виртуальный IP-адрес. Можно установить и балансировщик, но это не имеет смысла в данном случае, поскольку proxy перенаправляют всех клиентов на один узел (текущий мастер), в том числе и только читающую нагрузку.

Пример: Patroni



Patroni (Zalando, https://github.com/zalando/patroni) — еще одно кластерное решение с открытым исходным кодом, рассчитанное на использование как в облаках, так и на выделенных серверах.

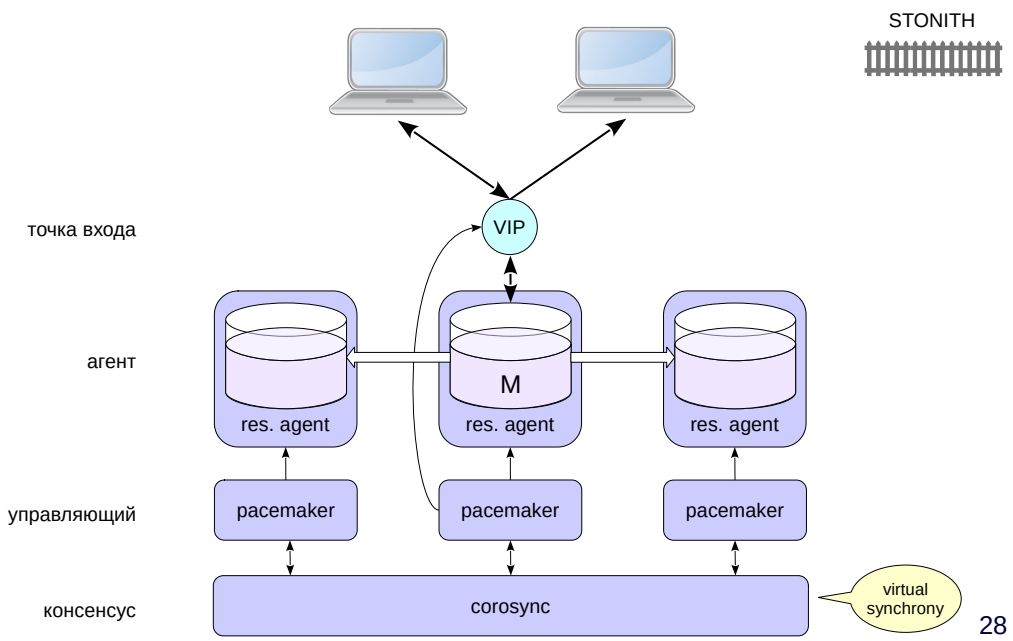
Здесь агент и управляющий объединены в одном компоненте. Для консенсуса используется хранилище «ключ-значение», предоставляемое одной из систем etcd (Raft), Consul (Raft) или ZooKeeper (ZAB).

Для точки входа требуется использовать отдельное решение. Обычно предлагается использовать HAProxy в связке с confd, который периодически обновляет конфигурацию HAProxy на основе информации из слоя консенсуса. Стандартная настройка предполагает перенаправление всех клиентов на узел, выполняющий роль мастера.

(Чтобы HAProxy сам не стал точкой отказа, необходимо иметь несколько таких серверов и, например, виртуальный IP-адрес для клиентов. Однако это приводит к тому, что конфигурации разных серверов HAProxy нужно обновлять синхронно, и мы снова сталкиваемся с вопросом построения кластера — уже на другом уровне.)

Для ограждения узла (fencing) может использоваться механизм сторожевого таймера (watchdog timer) ядра Linux. Если узел, будучи мастером, не может связаться со слоем консенсуса в течение определенного времени, таймер перезагружает операционную систему.

Пример: стек ClusterLabs



Компания ClusterLabs (https://clusterlabs.org/) предоставляет стек компонентов с открытым исходным кодом для построения кластеров из более или менее произвольных систем.

Агент (resource agent в терминологии этой системы) представляет конкретную систему как унифицированный «ресурс», реализуя операции останова, запуска, продвижения реплики и т. п. Этот компонент должен быть реализован отдельно для каждой системы.

Управляющий компонент — Pacemaker — взаимодействует с ресурсом через агента. Для общего состояния и обнаружения сбоев используется Corosync (Virtual synchrony).

Ограждению (fencing) узлов уделено большое внимание. Используется технология STONITH (Shoot The Other Node In The Head), позволяющая надежно изолировать сбойный узел от кластера и клиентов путем программного отключения питания или сетевого порта на коммутаторе.

Точкой входя для клиентов обычно является виртуальный IP-адрес, который управляющий поднимает на узле, являющимся в данный момент мастером.

Дополнительно при необходимости можно распределить читающую нагрузку по репликам с помощью HAProxy, PgBouncer и т. п.

Кластерное решение на основе стека Corosync/Pacemaker предлагает Postgres Professional; также имеется открытое решение PAF (http://clusterlabs.github.io/PAF/).

Итоги

Для создания кластера требуются средства,
не входящие в стандартный PostgreSQL

Огромное количество вариантов построения кластера

- разные цели: высокая доступность, масштабируемость
- решения с поддержкой глобальных транзакций
- решения с использованием внешних компонентов

Чем выше предъявляются требования,
тем сложнее система и тем дороже она обходится