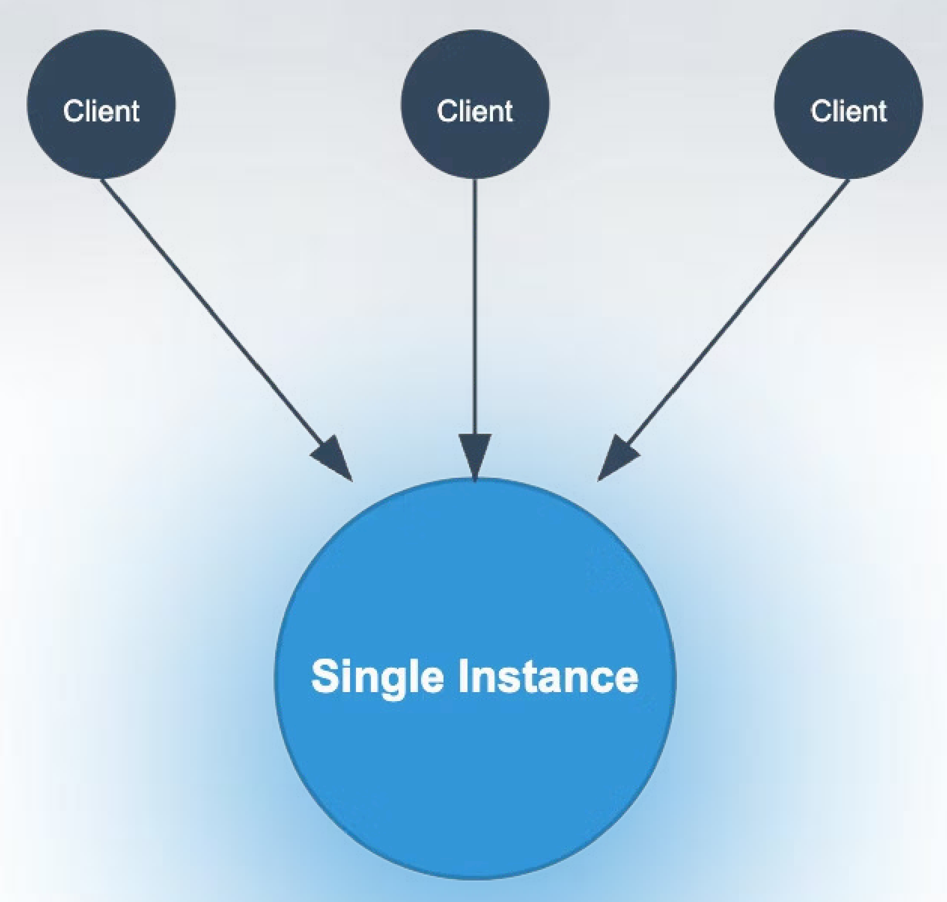
Шардинг против репликации: ментальная модель, которую вы действительно поймете

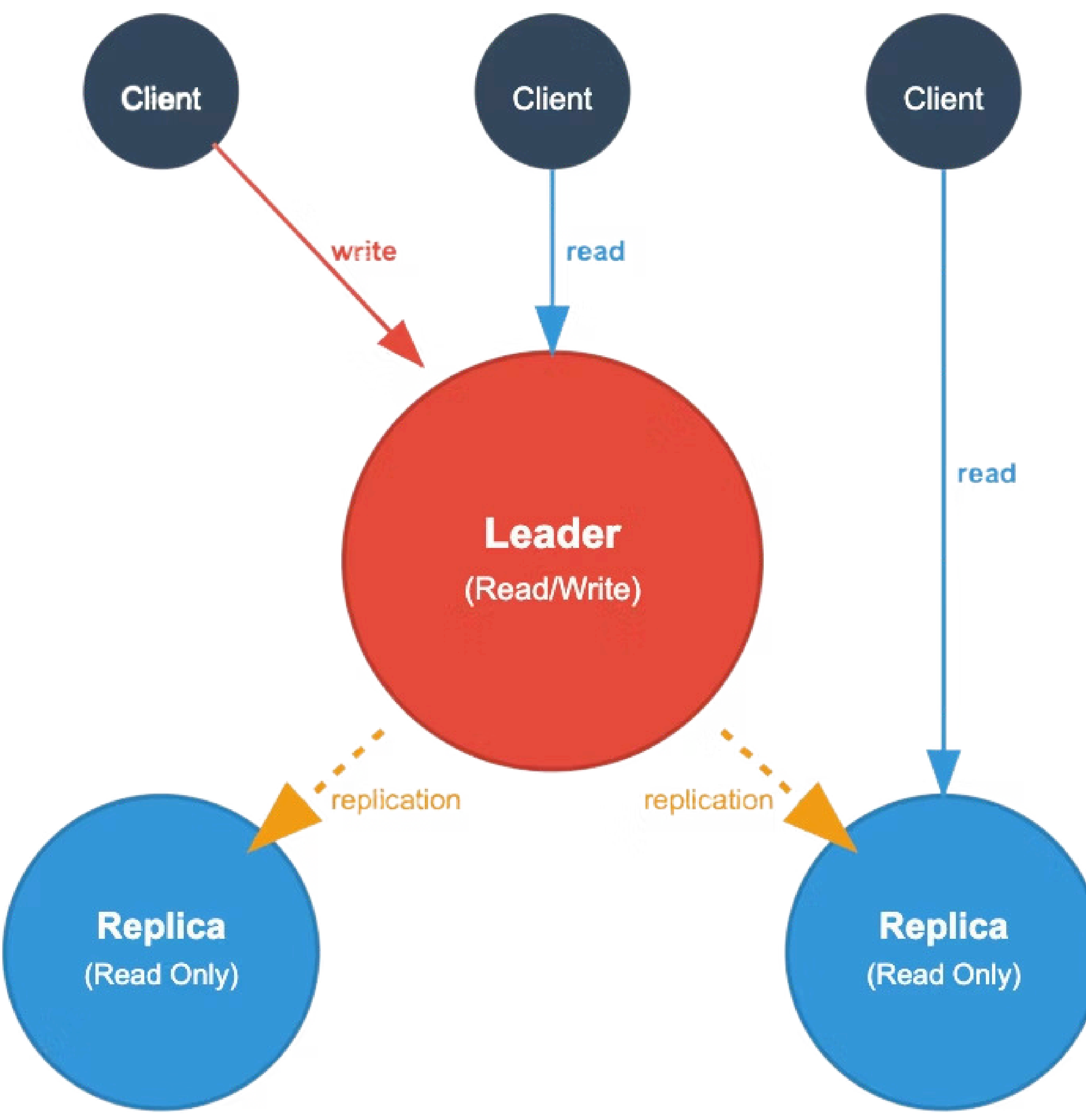
Шардинг и репликация — два фундаментальных подхода к построению распределенных систем, но разница между ними и их компромиссы не всегда очевидны на практике.

Система с одним экземпляром

Это самая простая конфигурация, где все происходит внутри одного узла. Хотя она не предлагает горизонтальной масштабируемости, она обеспечивает сильные гарантии согласованности и простое рассуждение о состоянии системы. Примеры включают PostgreSQL с одним экземпляром, Redis без кластеризации или автономные серверы приложений. Единственный способ справиться с возросшей нагрузкой — это вертикальное масштабирование (добавление большего количества CPU, памяти или хранилища к той же машине), которое имеет физические и экономические ограничения. В конечном итоге вы столкнетесь с аппаратными ограничениями, которые сделают дальнейшее масштабирование невозможным или экономически нецелесообразным.



Репликация с одним лидером



Самая простая форма введения масштабирования в такую систему — добавление большего количества фолловер-реплик. Эти реплики могут быть либо read-репликами для обслуживания клиентских запросов на чтение, либо просто находиться в горячем резерве, чтобы заменить лидера (который также называется мастером или write-репликой) в случае, если он выйдет из строя или сломается. Все реплики имеют доступ к полному набору данных (в базах данных обычно данные полностью реплицируются), поэтому нет необходимости вводить маршрутизацию трафика или стратегии шардинга (кроме маршрутизации записей всегда к лидеру).

В обоих случаях лидер должен реплицировать свои данные на эти реплики, и есть различные подходы к этому (например, в базах данных — отправка журнала опережающей записи, **WAL**, или логических команд). Но у нас все еще есть только один лидер для обработки запросов, которые изменяют данные или состояние.

Поэтому клиенты должны подключаться только к лидеру для выполнения write-запросов, но read-запросы могут обслуживаться любой read-репликой. Это смягчает сценарии использования с высоким соотношением чтения к записи, но не очень помогает, когда у нас много записей. На самом деле, для некоторых сценариев с высокой интенсивностью записи это может быть не лучшей конфигурацией, потому что:

> ⓘ **Что такое WAL-журнал и зачем он нужен**
>
> Во многих системах хранения данных важно не потерять изменения при сбоях — например, при внезапном выключении сервера. Чтобы обеспечить надёжность, используется **WAL-журнал** (Write-Ahead Logging).
>
> Принцип работы простой: **перед тем как изменить данные “на месте”, система сначала записывает информацию об этих изменениях в специальный журнал**. Если что-то пойдёт не так, система сможет восстановить данные, «проиграв» записанные в журнал операции.

- если мы выбираем синхронный режим репликации, то наша система может стать медленной или даже недоступной в случае, если она не может достичь некоторых read-реплик
- если мы выбираем асинхронный режим репликации, то читатели могут видеть устаревшие данные, если они обращаются к read-репликам, что может быть неприемлемо (например, подумайте об использовании устаревших/несогласованных данных банковского счета)

Еще один вопрос — что происходит, если лидер умирает. Обычно есть два подхода:

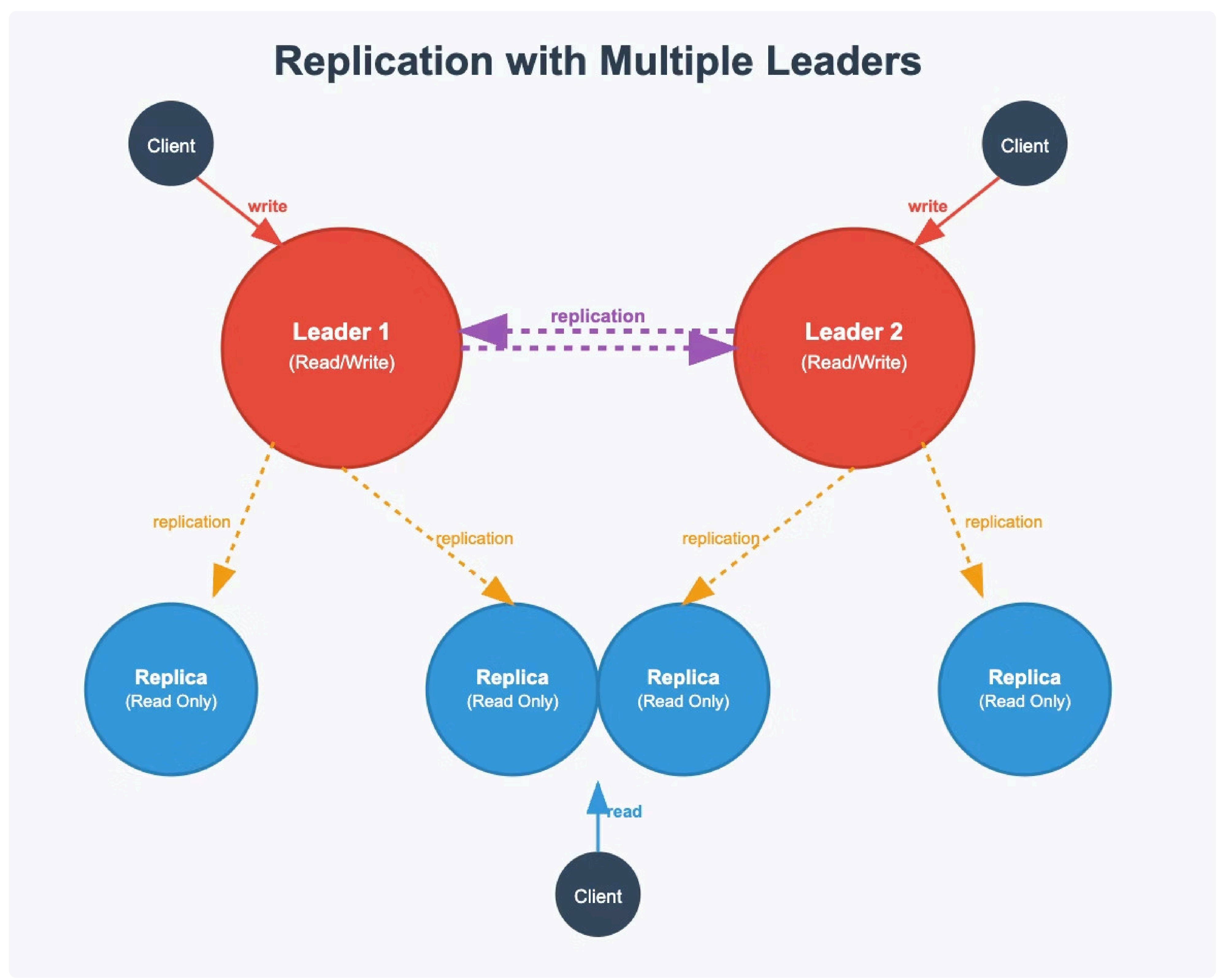
- ручное переключение — администратор должен прийти и установить одну из read-реплик в качестве нового лидера. Некоторые традиционные настройки баз данных все еще используют этот подход для критических систем, где администраторы хотят полного контроля над решениями о переключении. Система становится недоступной до вмешательства администратора, но преимущество в том, что вы избегаете сценариев split-brain и обеспечиваете, чтобы наиболее квалифицированная реплика стала новым лидером на основе человеческого суждения.
- автоматическое назначение лидера — системы используют алгоритм распределенного консенсуса, такой как **Paxos** или **Raft** или другие, для выбора нового лидера. Преимущество — более быстрое восстановление и уменьшенное человеческое вмешательство, но недостаток — сложность и потенциал для сценариев split-brain, если алгоритм консенсуса терпит неудачу или происходят сетевые разделения.

> ⓘ **Split-brain** — это ситуация, когда из-за сетевого разделения или сбоя связи несколько частей кластера **считают себя основными** и продолжают принимать изменения независимо друг от друга.

Один яркий пример, поддерживающий этот подход — PostgreSQL с потоковой репликацией. Вы можете настроить hot standby реплики и read-реплики, выбирая между синхронными и асинхронными режимами репликации. Другие примеры включают MySQL с master-slave репликацией, MongoDB replica sets (до шардинга) и Redis Sentinel для высокой доступности.

Несмотря на некоторые проблемы с масштабированием, у такого подхода есть важное преимущество: он автоматически обеспечивает линейную согласованность данных. Поскольку все операции проходят через одного лидера, глобальный порядок изменений задаётся естественным образом. Благодаря этому проще гарантировать корректность данных и проще анализировать поведение системы.

Репликация с несколькими лидерами



Некоторые ситуации, такие как гео-распределенные развертывания, совместное редактирование (ваше мобильное устройство, которое может быть оффлайн в течение дней, и сервер — тоже распределенная система!) или требования высокой доступности, требуют настройки нескольких реплик и назначения более одной из них в качестве write-реплик (то есть лидеров). Другими словами, у нас есть несколько экземпляров, которые могут принимать записи одновременно.

В этой конфигурации каждый лидер действует как фолловер для других лидеров. Также каждый лидер обычно управляет своим собственным набором фолловеров.

В системах с несколькими лидерами важно, как они обмениваются изменениями между собой. Существует несколько распространённых топологий:

1. **Все-ко-всем (full mesh)**
   - Каждый лидер напрямую реплицирует изменения всем другим лидерам.
   - Минимальная задержка распространения данных
   - Плохо масштабируется: количество связей резко растёт с числом узлов
   - Используется в небольших кластерах
2. **Кольцевая (ring)**
   - Каждый лидер отправляет изменения только одному соседу по кольцу, и данные «протекают» по цепочке.
   - Меньше сетевых соединений → выше пропускная способность
   - Больше задержка на доставку изменений до всех узлов
3. **Звездообразная (star)**
   - Один центральный лидер принимает и распределяет изменения остальным лидерам.
   - Простая конфигурация и управление
   - Центральный лидер становится узким местом и точкой отказа

> ⓘ Кто такие лидеры и фолловеры в распределённых системах
>
> В кластерах, где данные хранятся на нескольких узлах, важно точно определить, какой из них отвечает за приём и распространение новых изменений. Для этого используют роли **лидера** и **фолловеров**.
>
> - **Лидер** — узел, который принимает записи, координирует порядок операций и распространяет изменения другим узлам.
> - **Фолловеры** — узлы-реплики, которые получают данные от лидера и поддерживают копию его состояния.

В отличие от систем с одним лидером, конфигурации с несколькими лидерами жертвуют естественным упорядочиванием записей, делая разрешение конфликтов необходимым. Однако они наследуют преимущества масштабирования чтения от систем с одним лидером, добавляя при этом возможности масштабирования записи. Компромисс — увеличенная сложность в поддержании согласованности между несколькими write-узлами.

Разрешение конфликтов

В распределённых системах одни и те же данные могут изменяться одновременно на разных узлах. Например, две реплики получают запрос на изменение одной и той же записи почти в одно и то же время. Каждая применяет изменение локально и пытается распространить его на другие узлы. В итоге система сталкивается с конфликтом: существует несколько разных версий одного и того же объекта, и нужно выбрать или сформировать единую правильную версию.

Чтобы разрешать такие конфликты, используются разные подходы:

LWW (Last Write Wins) — «последняя запись побеждает»

Система выбирает вариант данных с наиболее «новым» временем изменения. Подход проще всего реализовать, но у него есть значительные ограничения:

- точность полностью зависит от синхронизации часов на узлах
- часы могут расходиться даже при использовании NTP
- риск потерять корректные обновления, если время определено неверно (запись может быть признана «старой», хотя фактически она более актуальна)

Этот метод чаще используется в системах, где согласованность не критична, но важна высокая доступность.

Версионные векторы (Vector Clocks)

Метод основан на отслеживании причинно-следственных связей между обновлениями. Вместо одного timestamp каждое обновление сопровождается вектором версий по каждому узлу.

Это позволяет:

- определить, какое обновление логически новее другого
- выявить конкурирующие изменения, которые нельзя просто упорядочить
- не полагаться на синхронизацию физических часов

Однако векторные версии сложнее реализовать и требуют дополнительной логики на клиенте или сервере.

Конфликтное разрешение на уровне приложения

Система не пытается сама решить конфликт. Она передаёт обе версии данных приложению, и уже прикладная логика принимает решение:

Примеры:

- объединение списков вместо выбора одного
- выбор значения по бизнес-правилам (например, сумма заказов)
- показ пользователю двух вариантов для ручного выбора

Этот метод даёт максимальную гибкость, но требует вовлечения разработчиков.

CRDT (Conflict-Free Replicated Data Types)

Это специальные структуры данных, которые гарантируют, что их можно объединять из любых реплик без конфликтов и без потерь информации. Объединение выполняется автоматически и всегда приводит к согласованному состоянию.

Используются там, где:

- обновления происходят часто и независимо
- важно отсутствие конфликтов
- требуется высокая доступность и офлайн-режим (например, совместное редактирование документов)

Примеры CRDT: наборы с автоматическим объединением, счётчики, текстовые структуры.

Вывод глобального порядка

Системы с несколькими лидерами обычно не могут надежно обеспечить линеаризуемость из-за фундаментального вызова установления глобального упорядочивания между одновременными записями к разным лидерам. Даже с точно синхронизированными часами сетевые задержки и дрейф часов делают практически невозможным гарантировать линеаризуемое упорядочивание. Некоторые системы, такие как Google's Spanner, пытаются это сделать, используя атомные часы и GPS для точной временной синхронизации, но это требует специализированного оборудования и сопровождается значительной операционной сложностью.

Линеаризуемость — это строгая модель согласованности данных, которая гарантирует, что все операции над данными в распределённой системе выглядят так, **как будто они выполняются последовательно — по одной, в едином порядке — и мгновенно становятся видимыми для всех узлов**.

Уровень согласованности зависит от выбранной схемы репликации между write-репликами и между write- и read-репликами. Варианты могут быть разными: от максимально быстрой и доступной конфигурации с полностью асинхронной репликацией — до строгой и медленной, где каждый запрос ждёт подтверждения от всех реплик. Также возможны промежуточные режимы, балансирующие между производительностью и согласованностью.

Когда использовать несколько лидеров

Системы с несколькими лидерами сохраняют преимущества масштабирования чтения: фолловеры всё так же могут обслуживать read-запросы. Но появляются дополнительные сложности:

- **Конфликты записей** — лидеры могут изменять одни и те же данные, и система должна уметь корректно их разрешать.
- **Потеря глобального порядка операций** — записи, выполненные разными лидерами, не имеют естественной общей последовательности.
- **Задержки репликации между лидерами** — данные успевают расходиться, и пользователи могут сталкиваться с временной несогласованностью.

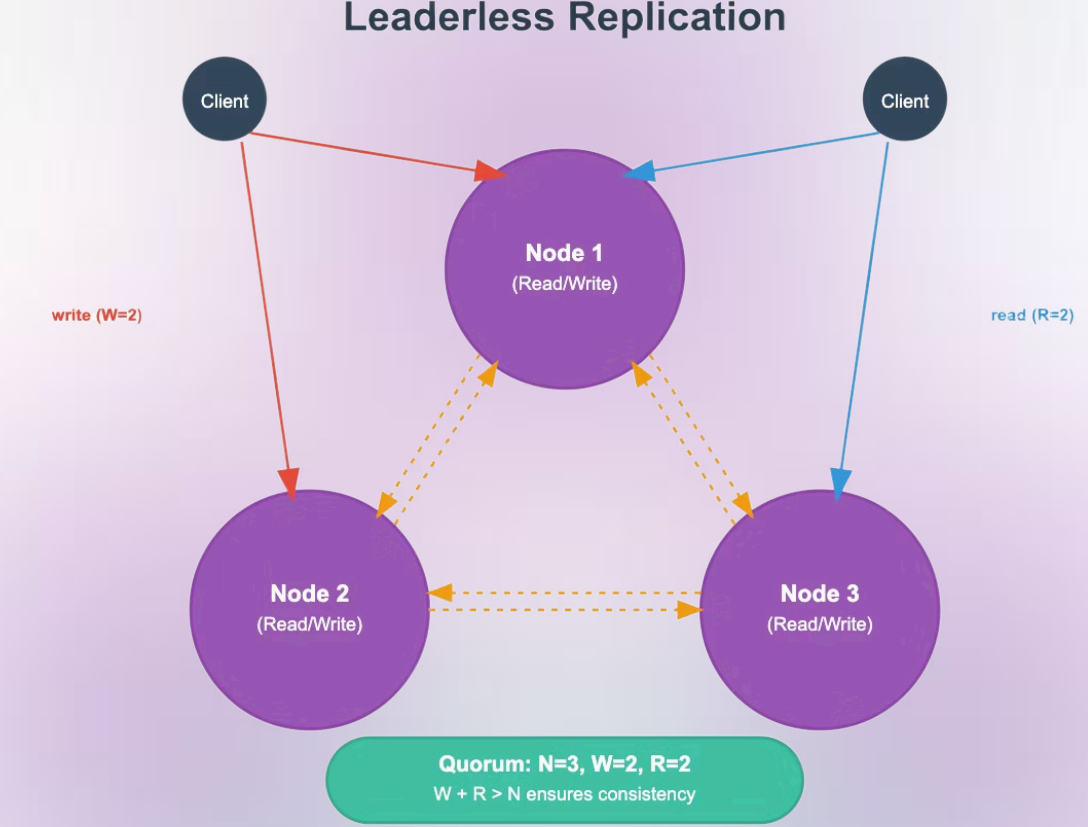
Использование мультилидерной репликации оправдано в ситуациях, когда:

- требуется **географическое распределение**: каждый регион имеет «своего» лидера, что снижает задержки для локальных операций;
- данные имеют **естественное разделение**: разные лидеры обслуживают разные категории или сегменты данных, сводя конфликты к минимуму;
- нужно **более предсказуемое и управляемое разрешение конфликтов**, чем обычно возможно в других системах

Безлидерная репликация

Безлидерная репликация — это конфигурация, где каждая реплика может обслуживать как read, так и write запросы, так что это фактически репликация с несколькими лидерами, доведенная до крайности (каждая реплика — лидер).

Беслидерные системы устраняют единую точку отказа, которую представляют лидеры, но они требуют более сложной клиентской логики для обработки кворум-чтений и записей. Они наследуют вызовы разрешения конфликтов от систем с несколькими лидерами, но обрабатывают их по-другому через техники, такие как read repair и анти-энтропийные процессы. В отличие от систем на основе лидеров, нет специальной роли для любого узла, делая систему более симметричной и потенциально более простой в эксплуатации.



Кворум-согласованность

Один важный факт в том, что безлидерная репликация, в отличие от репликации с несколькими лидерами, берет на себя ответственность за согласованность, вводя кворум-согласованность. Например, если у нас есть 3 реплики, мы можем ждать ответа только от 2 реплик при чтении и от 2 реплик при записи, и это даст нам гарантии согласованности (потому что по крайней мере одна read-реплика, которую мы запрашиваем, имеет последнее значение).

Чтобы поддерживать согласованность в условиях задержек и отказов, беслидерные системы применяют дополнительные механизмы:

Read repair

это техника для обнаружения устаревших значений (которые возможны с кворумами) и их обновления при чтении с нескольких узлов.

Анти-энтропия

это фоновый процесс, который сравнивает реплики и исправляет различия.

Sloppy quorum

позволяет записям продолжаться даже когда некоторые целевые узлы недоступны, временно записывая на разные доступные узлы

Hinted handoff

обеспечивает, что когда оригинальные узлы восстанавливаются, они получают записи, которые временно хранились в другом месте, поддерживая согласованность без блокировки операций во время отказов.

Разрешение конфликтов

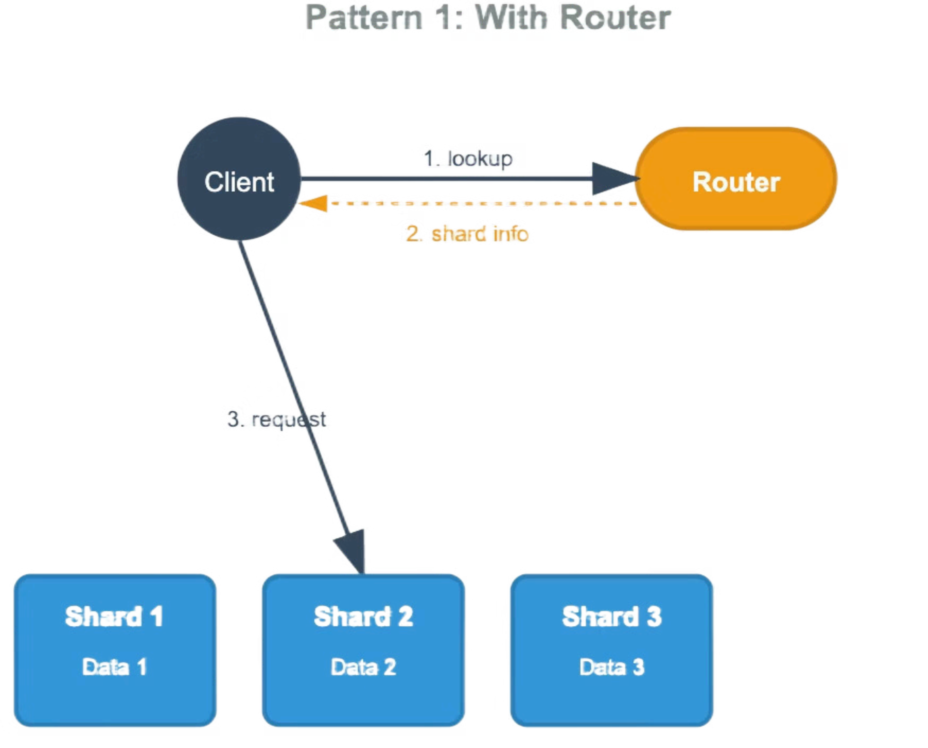
В безлидерной архитектуре нет единого узла, который определяет «правильную» версию данных. Все реплики равноправны и могут принимать записи независимо друг от друга. Это означает, что **конфликты неизбежны**, особенно если несколько клиентов обновляют один и тот же объект одновременно, но система должна уметь их корректно разрешать.

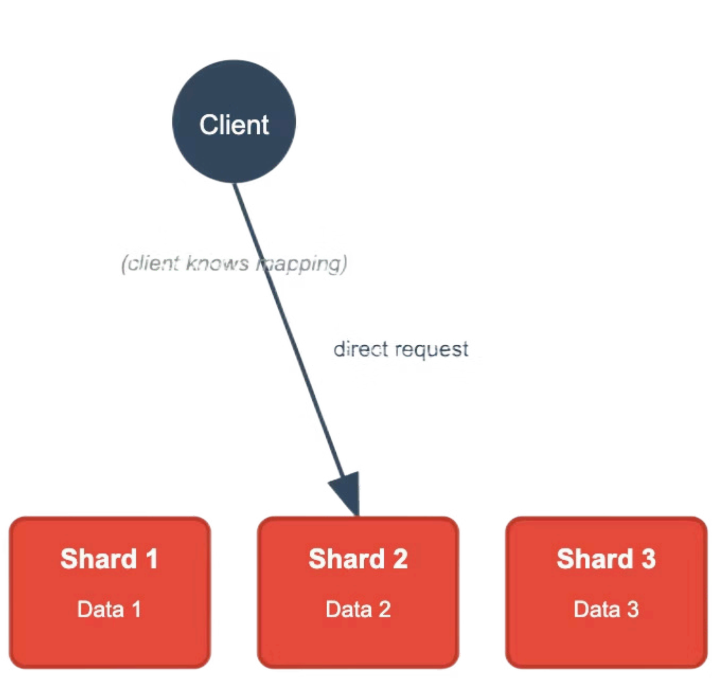
Используются те же механизмы, что и в мультилейдерных системах:

- **LWW (Last Write Wins)** — выбираем запись с самым новым временным штампом.
- **Версионные векторы** — отслеживаем, какие обновления происходили и как они связаны, чтобы определить, существует ли конфликт или одно значение является потомком другого.
- **CRDT (Conflict-free Replicated Data Types)** — такие структуры данных спроектированы так, что разные изменения могут быть безопасно объединены без потери информации.

> **Чем безлидерный подход отличается от мультилейдерного**
>
> Главная разница: в безлидерной репликации **конфликты обычно обнаруживаются и разрешаются при чтении, а не при записи.**
>
> - Записи обрабатываются быстро и независимо — без ожидания координации.
> - Но при чтении клиент или система может получить несколько разных версий данных и должен решить, какая из них правильная, или объединить их.
>
> Из-за этого:
>
> - Read-операции становятся более «умными» и потенциально тяжелее, поскольку могут включать логику сравнения и слияния.
> - Временная несогласованность становится нормой, и система работает над тем, чтобы со временем её устранить.

Шардинг





Если данных или запросов становится слишком много для одной машины, систему нужно масштабировать горизонтально — добавлять больше серверов. Для этого используется шардинг: мы делим данные и/или входящие запросы на части и распределяем их между несколькими экземплярами приложения. Каждый экземпляр обслуживает только свою часть нагрузки, за счёт чего система в целом выдерживает больший объём работы.

> **Шардинг ортогонален репликации** — у вас могут быть шардированные системы без репликации (каждый шард — это один экземпляр) или комбинировать шардинг с любой стратегией репликации (каждый шард может быть реплицирован с использованием подходов с одним лидером, несколькими лидерами или беслидерных подходов).

Вопросы шардинга

Но если мы делаем это, естественным образом возникают некоторые вопросы :

Разделение данных

Как мы разделяем данные и/или нагрузку и как добавляем/удаляем шарды?

Маршрутизация запросов

Как клиент узнает, какой шард должен обработать запрос (например, потому что он владеет данными)?

Балансировка нагрузки

Что если один шард получит непропорциональную долю нагрузки?

Отказоустойчивость

Что если шард становится дисфункциональным (например, отключается)

Давайте рассмотрим все эти вопросы.

Разделение данных и/или нагрузки

Есть несколько подходов к разделению нагрузки между экземплярами. Все они начинаются с того, что инженер выбирает ключ для сопоставления запросов с экземплярами, но алгоритмы фактического сопоставления варьируются:

Диапазоны ключей

Указание диапазонов ключей, которые сопоставляются с экземплярами (например, A-M идет к шарду 1, N-Z — к шарду 2). Это хорошо работает, когда ключи равномерно распределены, но может создавать горячие точки, если определенные диапазоны обращаются чаще. Примеры включают HBase и некоторые конфигурации MongoDB.

Хеширование

Вычисление хеша ключа и сопоставление его с экземплярами (например, hash(key) % num_shards). Это обеспечивает хорошее распределение, но делает невозможными диапазонные запросы, поскольку связанные ключи разбросаны по шардам. Добавление или удаление шардов требует значительного перемещения данных. Примеры включают Redis Cluster и некоторые развертывания Cassandra.

Каталог

Шардинг на основе каталога, где отдельная служба поиска сопоставляет ключи с шардами. Это обеспечивает гибкость в размещении данных и упрощает перебалансировку, но вводит дополнительный компонент, который может стать узким местом. Примеры включают некоторые распределенные файловые системы и ранние версии Google's Bigtable.

Согласованное хеширование

Согласованное хеширование — это распространённый способ распределения данных между серверами. Он работает так: каждому серверу назначается определённый диапазон хешей ключей, и именно данные с такими хешами этот сервер обслуживает. Есть улучшение — **виртуальные узлы (VNodes)**. Они позволяют при добавлении или удалении серверов не перегружать оставшиеся машины. Нагрузка распределяется более равномерно, и при изменении числа серверов нужно переносить только небольшую часть данных.

Минус метода — более сложный алгоритм хеширования.
Плюс — отличное балансирование нагрузки и минимальный объём перемещаемых данных при масштабировании.

Такой подход применяется в системах **Amazon DynamoDB, Apache Cassandra, Riak**.

Добавление или удаление узлов варьируется по стратегии шардинга:

- Шардинг на основе диапазонов требует разделения или объединения диапазонов и соответствующего перемещения данных.
- Шардинг на основе хеша обычно требует перехеширования и перемещения значительной части данных.
- Согласованное хеширование минимизирует перемещение данных, затрагивая только соседние узлы.
- Подходы на основе каталога могут обновлять сопоставления без немедленного перемещения данных, позволяя постепенную перебалансировку.

Маршрутизация запросов к их репликам

Чтобы запрос попадал именно на тот экземпляр, который отвечает за нужный сегмент данных, можно использовать **центральный компонент-маршрутизатор.** Клиент сперва обращается к нему, и тот определяет, на какой конкретный шард или реплику отправить запрос.

Примеры таких решений:

- **MongoDB mongos** — перенаправляет запросы на нужный шард
- **Citus (PostgreSQL)** — анализирует запросы и маршрутизирует их по узлам
- **Load balancers sticky sessions** — направляют запросы на сервер по ключу сессии
- **Kubernetes Services** — отправляют запросы в конкретные поды по правилам/атрибутам

Такой центральный компонент **не всегда нужен**, есть два варианта сопоставления запросов с экземплярами без него:

Умные клиенты

Клиенты могут вывести, к какому экземпляру они хотят подключиться на основе ключа. Это работает, когда алгоритм шардинга детерминирован и известен клиентам, например, с простым шардингом на основе хеша или известными разделениями диапазонов. Это предпочтительно, когда вы хотите минимизировать накладные расходы маршрутизации и иметь умных клиентов. Примеры включают клиенты Redis Cluster и некоторые клиентские библиотеки Cassandra, которые реализуют алгоритм согласованного хеширования локально.

Peer-to-peer координация

Наша система реализует умную peer-to-peer координацию, где клиенты могут связаться с любым узлом, и этот узел обрабатывает маршрутизацию внутренне. Клиенты поддерживают список всех IP-адресов узлов и могут связаться с любым случайным. Контактируемый узел либо обслуживает запрос (если он владеет данными), либо перенаправляет его к соответствующему шарду. Это хорошо работает для чтений и может работать для записей через техники, такие как hinted handoff, где узлы временно хранят записи для недоступных шардов. Примеры включают координирующие узлы Cassandra и координацию запросов Riak.

Непропорциональная нагрузка и дисфункциональные экземпляры

Непропорциональная нагрузка

Непропорциональная нагрузка возникает, когда определенные шарды получают значительно больше запросов, чем другие, часто из-за горячих точек в данных (знаменитые пользователи, популярный контент) или плохо выбранных ключей шардов. Стратегии смягчения включают выбор лучших ключей шардов, которые равномерно распределяют нагрузку, использование согласованного хеширования с виртуальными узлами для распределения горячих точек по нескольким физическим узлам, реализацию балансировки нагрузки на уровне приложения или создание дополнительных реплик для горячих шардов.

Дисфункциональные шарды

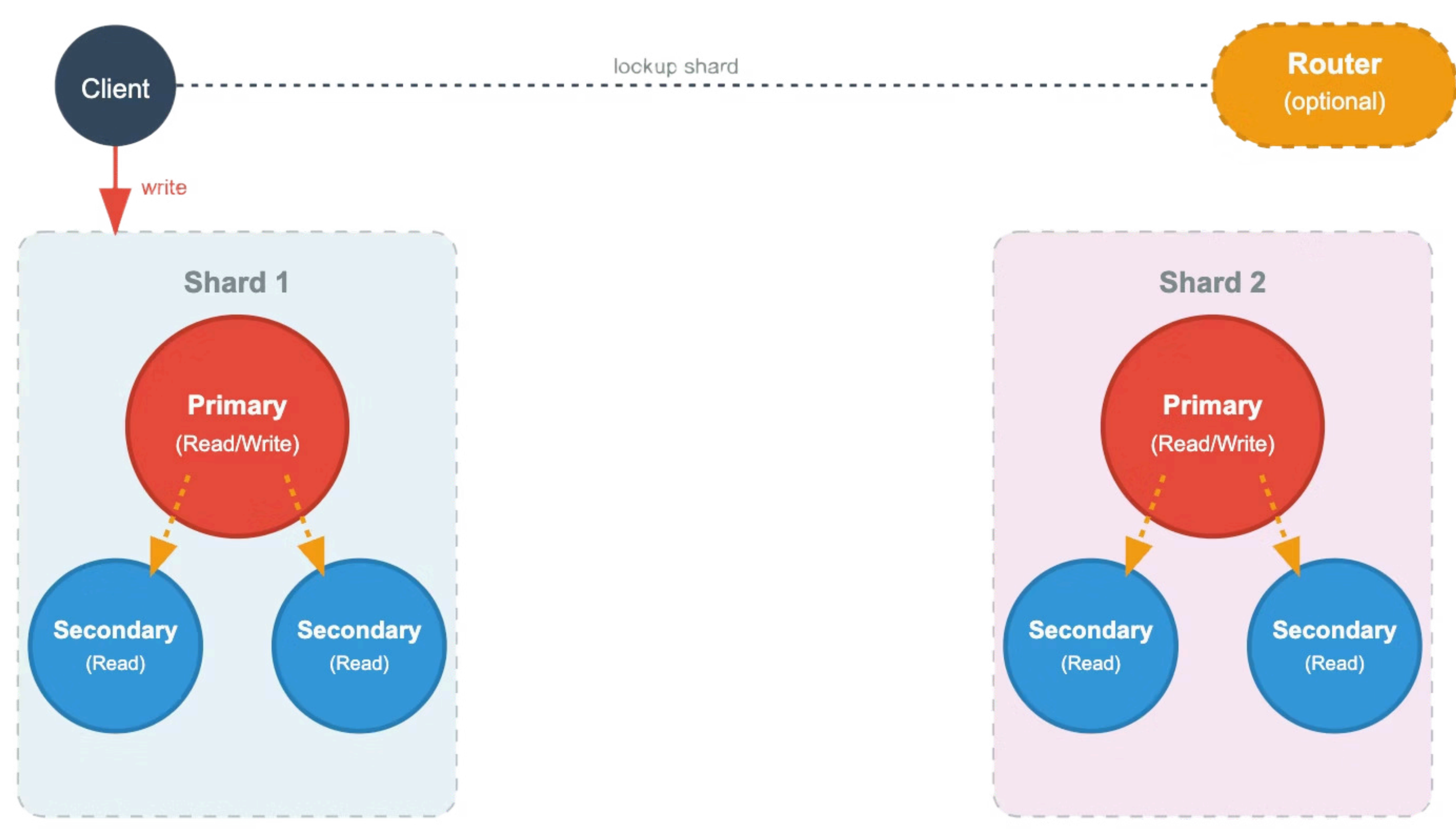
То, что произойдёт при сбое одного из экземпляров, зависит от того, используется ли **репликация данных**:

- Если репликация есть — другие реплики шарда могут временно заменить вышедший из строя экземпляр, и сервис продолжит работать.
- Если репликации нет — все данные этого шарда становятся недоступными, пока экземпляр не восстановят.

Чтобы минимизировать сбои, обычно применяют:

- **Проверки здоровья (health checks)** и автоматическое исключение неработающих узлов из маршрутизации
- **Автоматическое переключение (failover)** на реплику шарда
- **Несколько реплик для каждого шарда** для повышения отказоустойчивости
- **Circuit breakers** — быстрый механизм выявления проблемных узлов и перенаправления трафика в обход них

Репликация встречает шардинг



Репликация и шардинг — ортогональные концепции, но они могут дополнять друг друга. В этом разделе давайте обсудим, как их комбинировать для получения лучших качеств от обоих подходов:

Отказоустойчивость от репликации

обеспечивает доступность данных даже при отказе узлов

Распределение нагрузки от шардинга

обрабатывает наборы данных больше одной машины

Масштабирование чтения

через распределение реплик по шардам

Географическое распределение

где каждый регион имеет реплицированные шарды

Примеры комбинированных систем

MongoDB хорошо иллюстрирует эту комбинацию: она использует replica sets (обычно 3 узла с одним primary и двумя secondary) для каждого шарда, обеспечивая как отказоустойчивость, так и распределение нагрузки. Запросы распределяются по шардам на основе ключа шарда, в то время как чтения могут обслуживаться secondary репликами внутри каждого шарда.

Apache Kafka предоставляет еще один отличный пример: топики разделены на партиции (шардинг), где каждая партиция может иметь несколько реплик на разных брокерах (репликация). Одна реплика служит лидером, обрабатывающим чтения и записи, в то время как фолловеры обеспечивают отказоустойчивость. Этот дизайн позволяет Kafka масштабироваться горизонтально путем добавления большего количества партиций и поддерживать высокую доступность через репликацию.

Другие примеры включают кластеры Elasticsearch (шарды с репликами), Cassandra (согласованное хеширование с фактором репликации) и распределенные SQL базы данных, такие как CockroachDB.

Основные компромиссы комбинирования репликации и шардинга включают увеличенную операционную сложность (управление как распределением шардов, так и согласованностью реплик), более высокие требования к ресурсам (несколько копий данных по нескольким шардам) и более сложные сценарии отказов (вам нужно обрабатывать как отказы шардов, так и отказы реплик). Однако преимущества часто перевешивают эти затраты для больших систем, которым нужны как высокая доступность, так и горизонтальная масштабируемость. Ключ — выбор правильного фактора репликации и стратегии шардинга для вашего конкретного случая использования и операционных возможностей.

Выводы

Понимание распределенных систем фундаментально сводится к пониманию компромиссов между согласованностью, доступностью и устойчивостью к разделению — хотя мы можем расширить это, включив масштабируемость и операционную сложность. Каждый архитектурный паттерн, который мы исследовали, представляет разные точки на этих кривых компромиссов:

Системы с одним экземпляром

предлагают самые сильные гарантии согласованности и простейшие операции, но жертвуют масштабируемостью и доступностью. Они идеальны для небольших приложений или когда сильная согласованность первостепенна.

Репликация с одним лидером

обеспечивает золотую середину для нагрузок с интенсивным чтением, предлагая естественную линеаризуемость при добавлении масштабируемости чтения и базовой отказоустойчивости. Компромисс — ограниченная масштабируемость записи и потенциальные единые точки отказа.

Репликация с несколькими лидерами

обеспечивает географическое распределение и масштабируемость записи, но вводит сложность разрешения конфликтов и отказывается от сильных гарантий согласованности. Она идеальна, когда вам нужно обрабатывать записи в разных регионах или есть естественно разделенные паттерны записи.

Беслидерная репликация

максимизирует доступность и устраняет единые точки отказа через симметричную архитектуру, но требует сложной клиентской логики и тщательной настройки параметров кворума для балансировки согласованности и производительности.

Шардинг

становится необходимым, когда данные или пропускная способность превышают возможности одной машины, но вводит сложность распределения данных, маршрутизации и перебалансировки. Выбор стратегии шардинга (диапазон, хеш, согласованное хеширование или на основе каталога) зависит от ваших паттернов доступа и операционных требований.

Комбинированная репликация и шардинг

представляет архитектуру большинства крупномасштабных распределенных систем, предлагая как горизонтальную масштабируемость, так и отказоустойчивость за счет значительной операционной сложности.

> **Ключевое понимание** в том, что нет универсально «лучшего» подхода — правильный выбор зависит от ваших конкретных требований для согласованности, доступности, масштабируемости и операционной простоты. Современные распределенные системы часто используют разные паттерны для разных компонентов: строго согласованное хранилище метаданных может использовать репликацию с одним лидером, в то время как пользовательские данные могут быть шардированы с беслидерной репликацией для максимальной доступности.

Понимая эти фундаментальные паттерны и их компромиссы, вы можете принимать обоснованные архитектурные решения и лучше рассуждать о поведении сложных распределенных систем в продакшне.