Distributed systems

Репликация

Репликация — это процесс, под которым понимается копирование данных из одного источника на другой (или на множество других) и наоборот.

Цели репликации:

- Дать возможность системе продолжать работать в условиях, когда некоторые части системы недоступны (**Availability**)
- Расположить данные ближе к клиентам, т. е. уменьшить время получения данных (**Latency**)
- Увеличить количество серверов которые могут обрабатывать запросы на чтения, т. е. повысить пропускную способность системы (**Read throughput**)

Виды репликация

по механизму передачи данных:

- Синхронная
- Асинхронная
- Полусинхронная

по топологии системы:

- Single leader
- Multi leader
- Leaderless

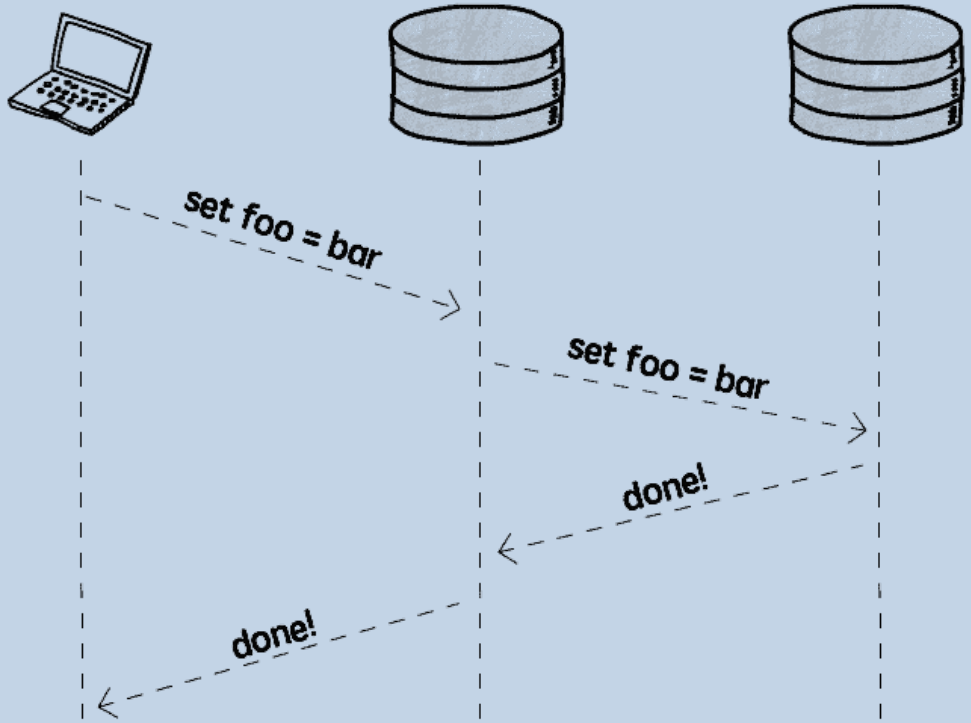
Синхронная репликация

Плюсы:

- Согласованность данных

Минусы:

- При отказе одной из нод, потеря доступности



Асинхронная репликация

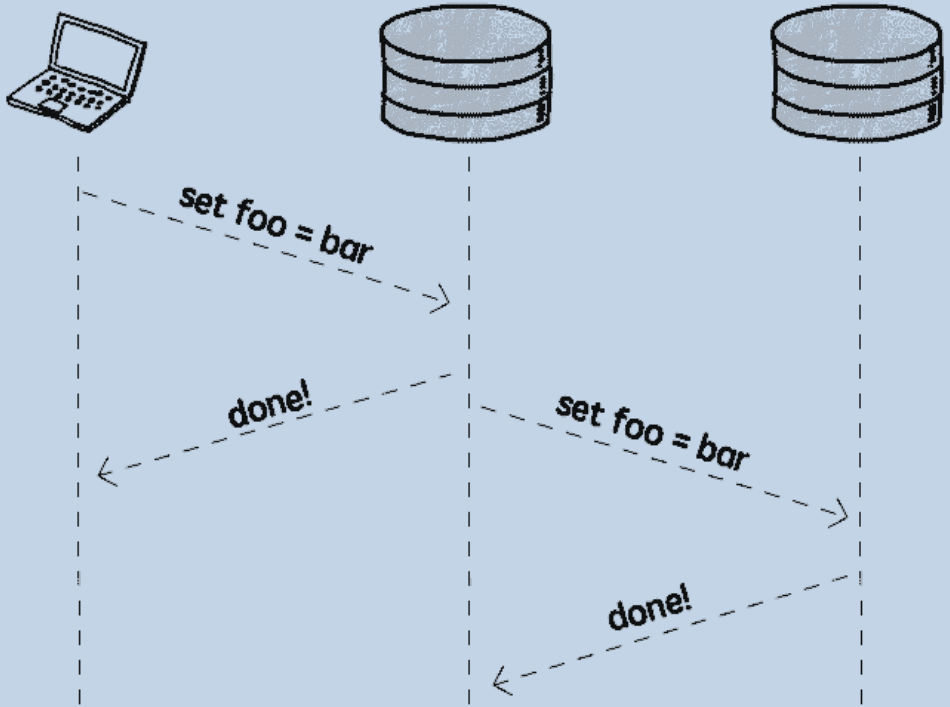
Плюсы:

- Система продолжает работать при отказе одной из нод.

Минусы:

- При отказах возможна потеря согласованности данных

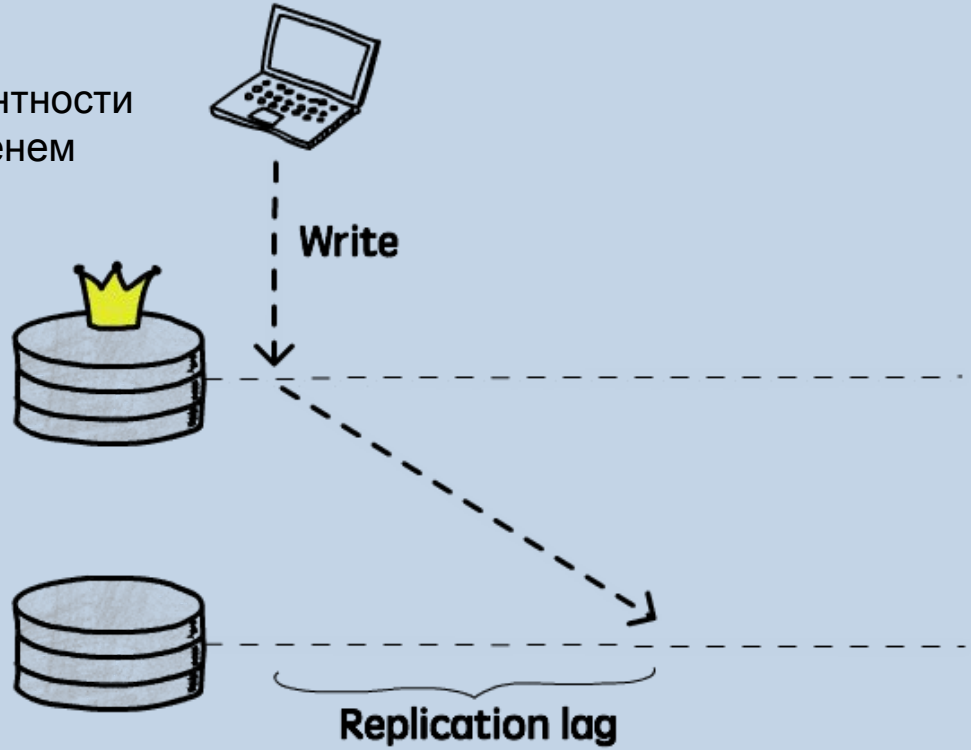
* Часто используется смешанный подход (полусинхронная репликация)



Отстование репликации (Replication lag)

Из-за задержки при репликации
может возникать временная потеря консистентности
данных, которая восстанавливается со временем
так называемая (***eventual consistency***)

В теории время за которое все ноды
обновятся до состояния лидера
не лимитировано, однако на практике
ожидается обновление
в течении миллисекунд

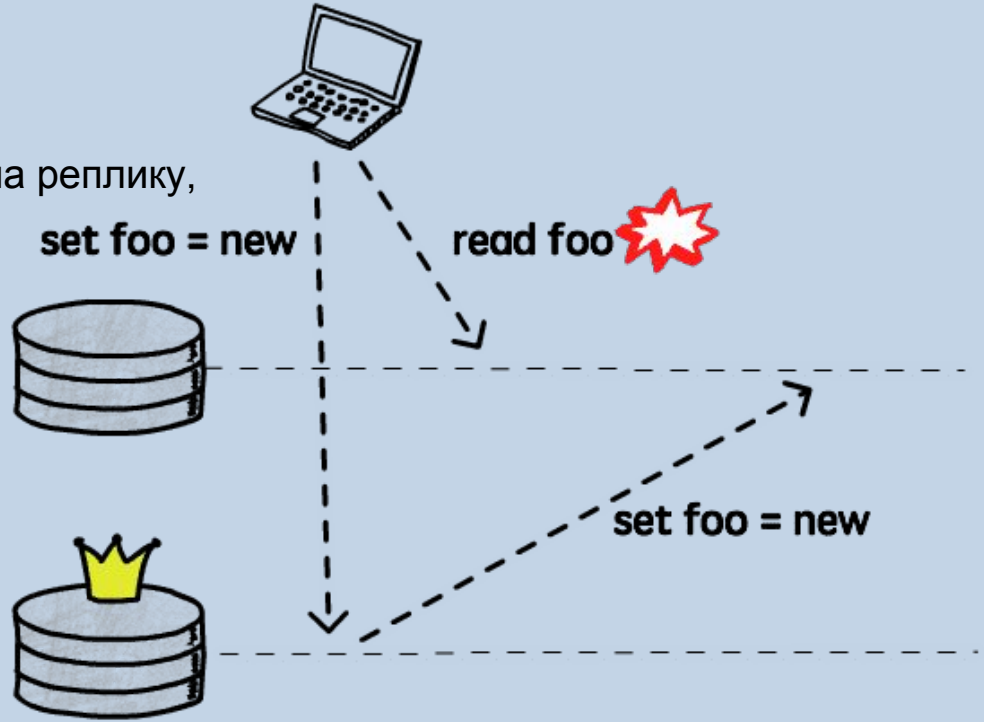


Read-your-writes consistency

При асинхронной репликации,
если запрос на изменение данных
приходит на мастер-ноду,
а последующий запрос на чтение попадает на реплику,
клиент может увидеть устаревшие данные,
хотя он сам же их обновил.

Допустимость такого поведения
определяется бизнес логикой приложения.

Гарантия при которой клиент
сразу видит свои изменения
называется *read-your-writes consistency*

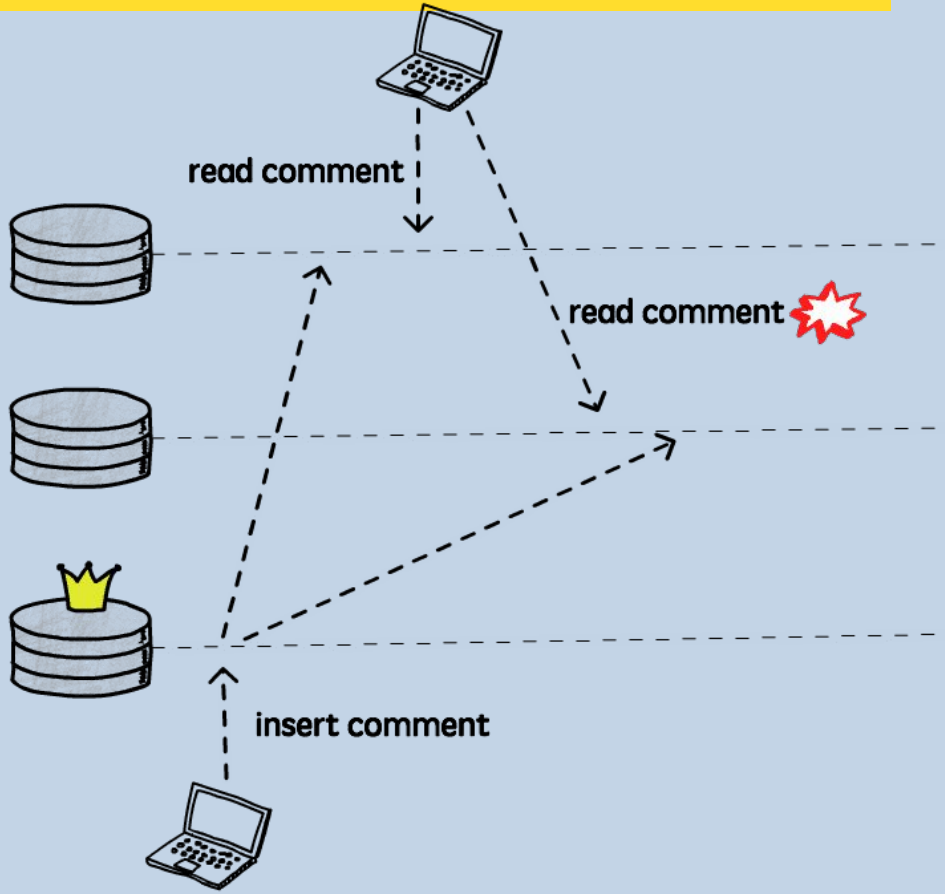


Monotonic Reads consistency

Также при асинхронной репликации,
может возникнуть ситуация,
когда клиент после повторного запроса,
увидит предыдущее состояние.
(как будто последние изменения
не пропали, потом снова появились).

Такой эффект может возникать,
если запросы клиента на чтение
распределяются по разным репликам

Гарантия при которой клиент
не увидит состояние в прошлом
monotonic read consistency



Single leader replication

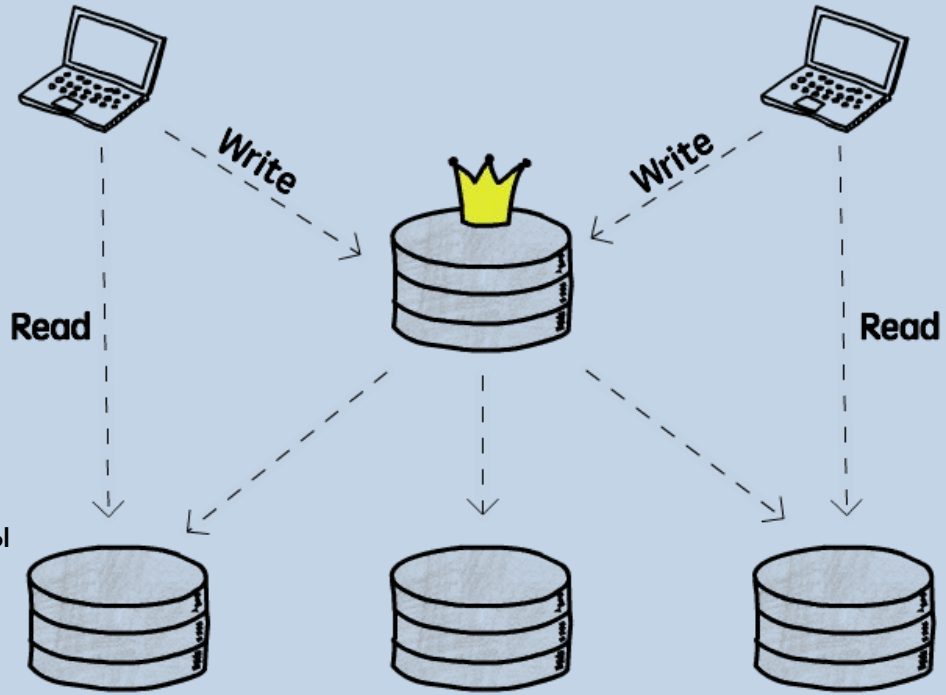
Наиболее распространенная топология, все запросы на изменение данных проходят через одну ноду (**Leader**)

Плюсы:

- Не нужно обрабатывать конфликты при конкурирующих запросах на изменение данных

Минусы:

- Single point of failure
- Не масштабируется
- Нужен способ перевыбора ведущей ноды



Multi leader replication

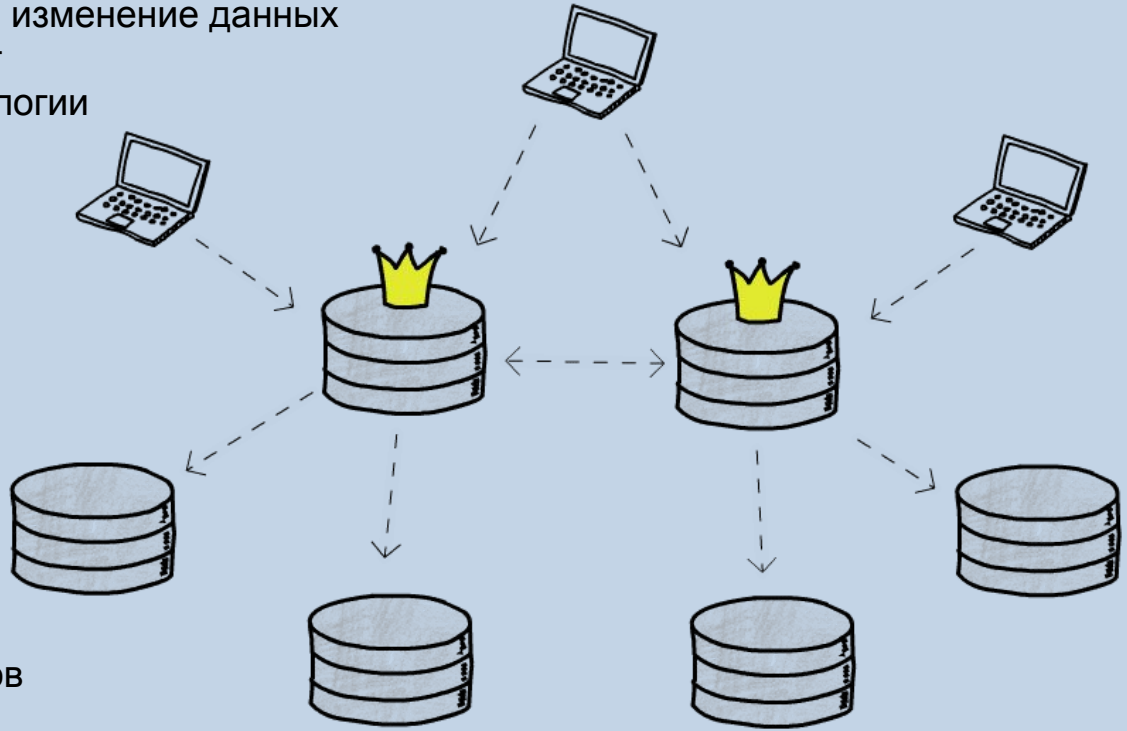
Топология при которой запросы на изменение данных могут принимать несколько нод. Решает проблемы, которые возникают при топологии single leader

Плюсы:

- можно разместить сервера ближе к клиентам
- позволяет масштабировать систему с ростом нагрузок
- Оффлайновые клиенты

Минусы:

- Нужен механизм обработки конкурирующих запросов

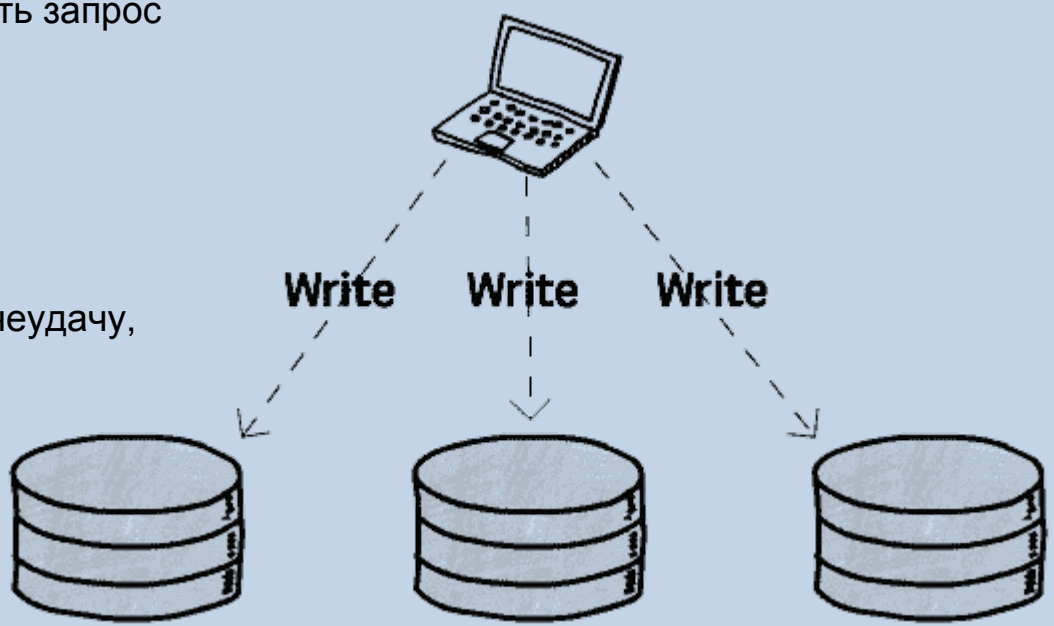


Leaderless replication

Топология взятая из **Amazon’s DynamoDB**.
Нет лидера, любая нода может обрабатывать запрос

Плюсы:
- Хорошо масштабируется
- Нет единой точки отказа

Минусы:
- Если запись на одну из нод потерпел неудачу, нужен механизм досинхронизации ноды до актуального состояния



Leaderless Replication

Возможные способы досинхронизации:

- *Read repair* - отправлять запросы на все реплики, сравнивать ответы и на отстающие реплики записывать актуальное состояние.
- *Anti-entropy process* - фоновый процесс, который постоянно сканирует реплики на поиск устаревших данных и обновлять их до актуального состояния

Quorum

Сколько нод должны участвовать в кворуме, чтобы мы всегда получали актуальные данные?

Обычно количество нод (n) выбирают нечетным, а нод на чтение и на запись: $r = w = (n + 1) / 2$

The quorum condition, w + r > n, allows the system to tolerate unavailable nodes as follows:

- *If w < n, we can still process writes if a node is unavailable*
- *If r < n, we can still process reads if a node is unavailable*
- *With n = 3, w = 2, r = 2 we can tolerate one unavailable node*
- *With n = 5, w = 3, r = 3 we can tolerate two unavailable nodes*
- *Normally, reads and writes are always sent to all n replicas in parallel. The parameters w and r determine how many nodes we wait for—i.e., how many of the n nodes need to report success before we consider the read or write to be successful.*

Sloppy Quorums and Hinted Handoff

Идея - если в какой-то момент времени ноды, которые должны обработать запрос с кворумом недоступны, то дать возможность принять решение о записи доступным нодам (sloppy quorums)
Когда работа системы будет восстановлена, ноды временно принявшие решение о записи должны передать данные на ноды отвечающие за данные (hinted handoff)

Репликация под капотом?

Так как репликация это процесс передачи данных между серверами, существуют разные способы как передавать данные:

- *Statement-based replication*
- *Write-ahead log (WAL) shipping*
- *Logical (row-based) log replication*

Statement-based replication

В данном способе мастер отправляет на реплики тот же запрос, который он обработал сам, в случае реляционных баз данных - INSERT, UPDATE , DELETE. Реплика обрабатывает запрос как будто его отправил клиент.

С одной стороны подход очень прост и эффективен, но имеет ограничения:

- Запрос может содержать недетерминированные функции, такие как now() или rand()
- Если используются авто-инкрементальные столбцы, то их нужно поддерживать в согласованном состоянии
- Запросы использующие триггеры, хранимые процедуры или кастомные функции могут приводить к неприятным побочным эффектам

Возможный воркраунд - вместо всех недетерминированных вызовов подставлять значения полученные на мастере.

Write-ahead log (WAL) shipping

Движок хранения базы данных записывает любое изменение в *WAL*.
WAL - это доступный только на запись журнал изменений в виде потока байт.
Мы можем не только записывать в этот журнал, но рассылать изменения на реплики, которые будут копировать записи к себе в журнал и применять их,
таким образом мы получим точную копию физических данных мастера.

Главный недостаток метода - работа с данными на очень низком уровне, *WAL* содержит информацию какие байты находятся в каких сегментах на дисках. Это делает данный вид репликации сильно зависящим от внутреннего устройства базы данных. Могут возникнуть проблемы с обновлением на новые версии.

Logical (row-based) log replication

Улучшенной версией предыдущего метода является логическая репликация,
идея отделить внутреннее устройства от формата журнала.

Для реляционной базы данных этот журнал представляет собой последовательность записей описывающих изменения на уровне строк:

- Для добавленной строки - запись со значениями каждого столбца
- Для удаленной информация, которая однозначно определяет строке (первичный ключ)
- Для измененной строки - ключ и новые значения столбцов

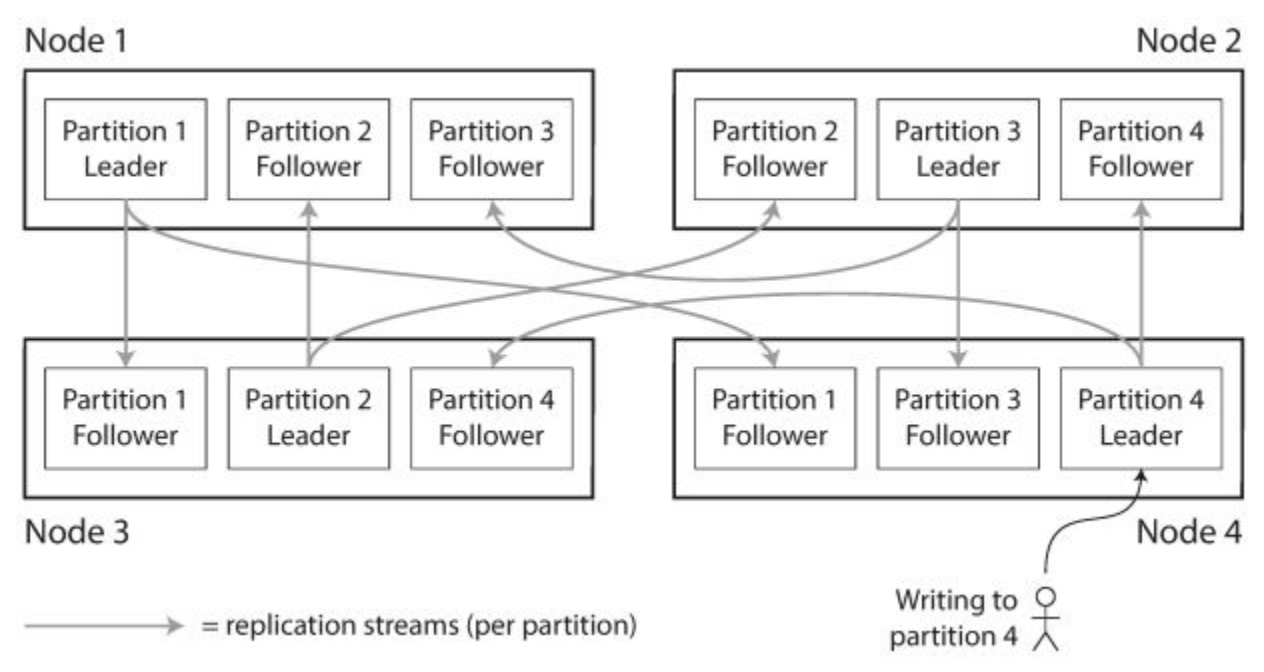
Данный способ избавляет от проблемы с сильной зависимостью от внутреннего устройства базы данных, также это более удобный формат для обработки внешними системами

Partitioning

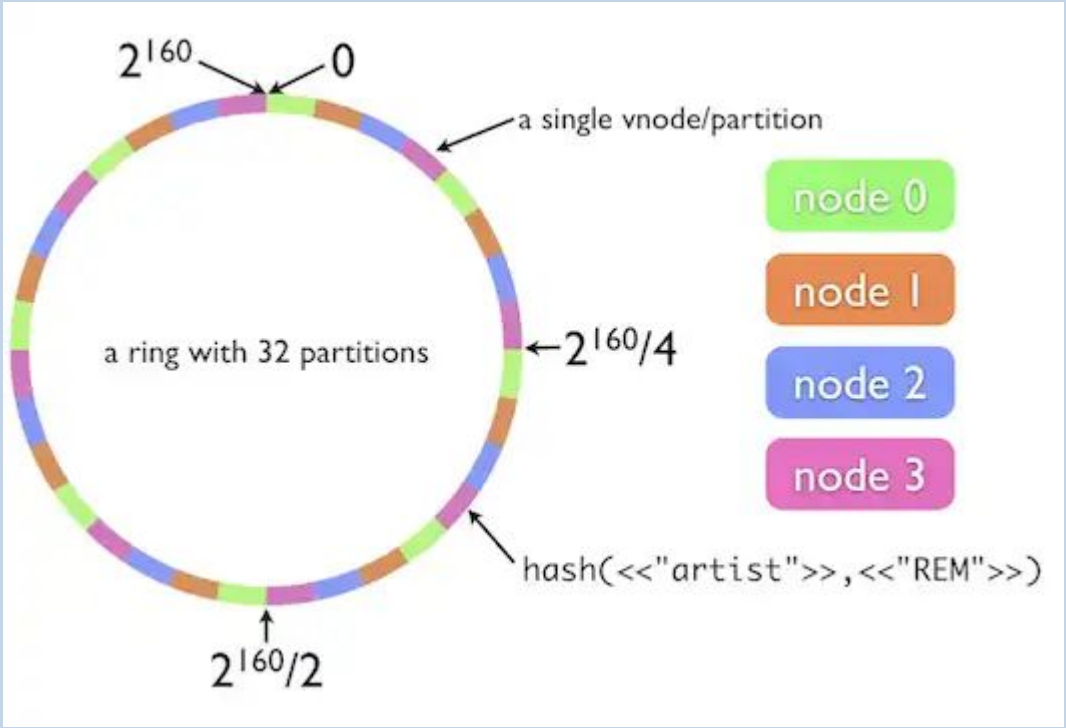
Партиционирование (иногда называют *шардингом*) - разделение хранимых объектов баз данных (таких как таблиц, индексов, материализованных представлений) на отдельные части с раздельными параметрами физического хранения. Используется в целях повышения управляемости, производительности и доступности для больших баз данных.

Возможные критерии разделения данных, используемые при секционировании — по предопределённым диапазонам значений, по спискам значений, при помощи значений хэш-функций; в некоторых случаях используются другие варианты. Под *композитными* (составными) критериями разделения понимают последовательно применённые критерии разных типов.

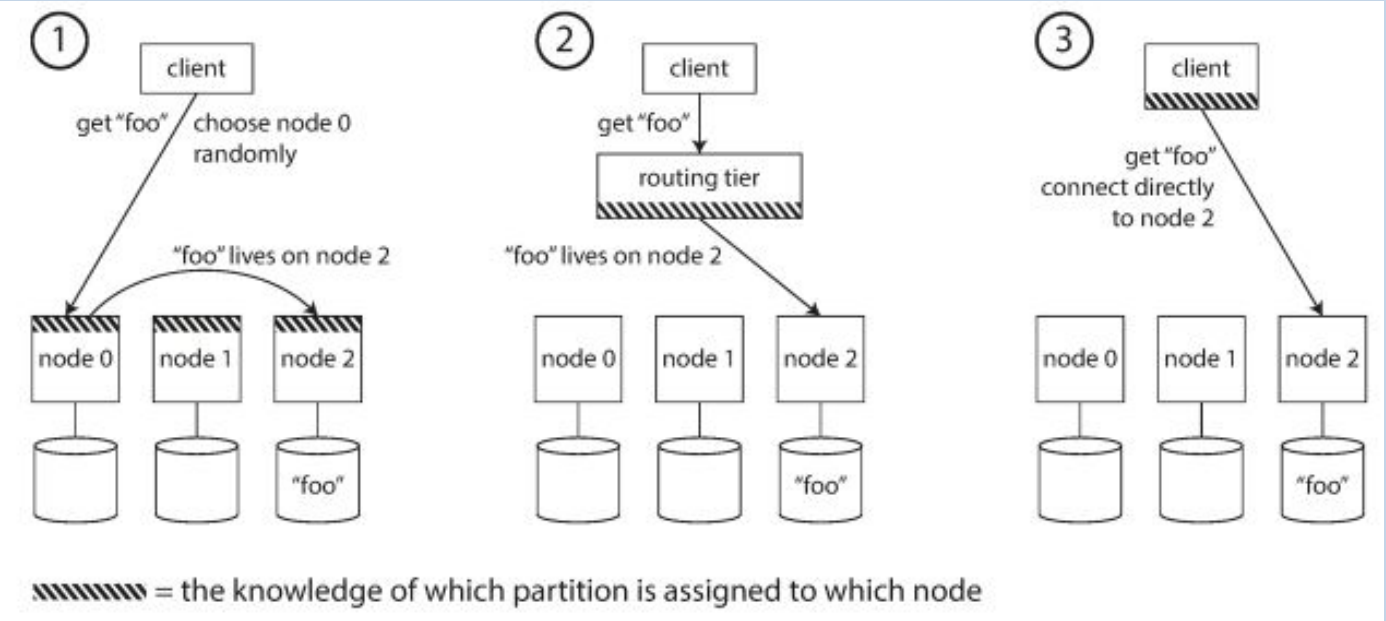
Combining partitioning and replication



Consistent hashing



Request routing



Транзакции

Транзакции не нужны?

Набор действий выполняемых на базой данных, которые либо одновременно фиксируются либо откатываются к предыдущему состоянию, как будто ничего не происходило.
Канонически транзакции описываются свойствами *ACID*

ACID свойства

Atomicity - Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной

Consistency - Транзакция достигающая своего нормального завершения и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных

Isolation - Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат

Durability - Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

Уровни изоляции транзакций

ANSI/ISO SQL стандарт определяет четыре уровня изоляции с различными возможными результатами для одного и того же сценария транзакции.
Эти уровни определяются в терминах трех явлений, которые либо разрешены либо нет на заданном уровне изоляции:

- Грязное чтение (***dirty read***) - допускает чтение еще не закомичинных данных (измененных другой транзакцией)
- Неповторяемое чтение (***non repeatable read***) - если данные были прочитаны в момент *T1* и перечитаны в момент *T2*, то при повторном чтении данные могут отличаться или отсутствовать
- Фантомное чтение (***phantom read***) - при повторном чтении могут появиться новые строки, которых не было при первом чтении

Уровни изоляции транзакций

Isolation level	Lost updates	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	don't occur	may occur	may occur	may occur
Read Committed	don't occur	don't occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Lock-based Concurrency Control

Первый подход в реализации изоляций при совместном доступе базируется на блокировках строк:

- *Read Uncommitted.* При записи устанавливается write lock на строку, которую изменяют и блокировка сразу отпускается после изменения. При чтении ставится read блокировка и после сразу отпускается
- *Read Committed.* При записи устанавливается блокировка и отпускается только когда транзакция фиксируется
- *Repeatable Reads.* В дополнение к блокировке на запись на все время транзакции происходит блокировка прочитанных данных

Multi-Versioned Concurrency Control (MVCC)

MVCC - подход предлагает вместо блокировок сохранять состояние на момент начала транзакции и в области видимости транзакции работать только с ним.

В *MVCC* доступно два уровня изоляции:

- *Read Committed.* В данном варианте снапшот состояния делает на момент чтения данных, запросы на запись также требуют блокировки, но в данном подходе читатели не блокирует писателей и наоборот.

- *Repeatable Read/Serializable*. Снапшот делается на момент начала транзакции, на запись блокировка как в *read committed*.

Distributed systems

В распределенных базах данных гораздо сложнее обеспечить полноценные *ACID* транзакции. Зачастую под распределенной транзакцией имеют ввиду протокол двухфазного коммита (*XA transaction*), но этот протокол лишь описывает как скоординировать одновременное выполнение транзакций между серверами и никак не гарантирует *ACID* свойств на уровне всей системы.

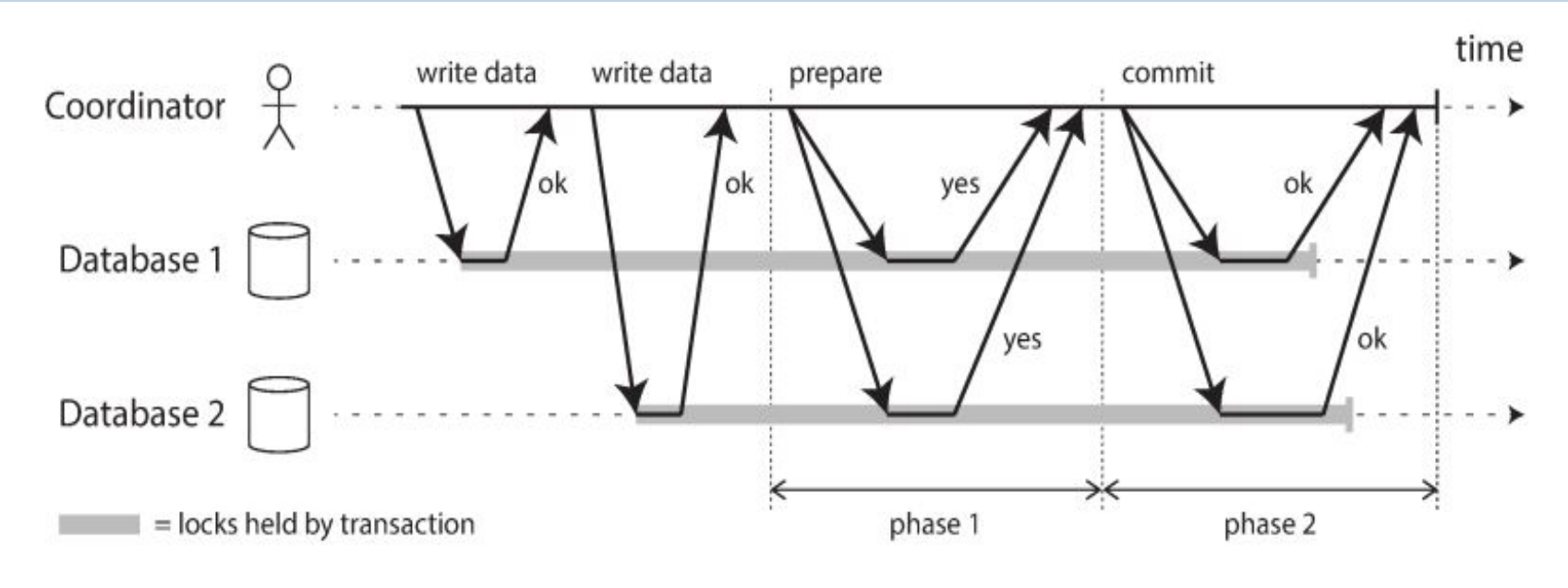
Двухфазный коммит (Two-phase commit)

Двухфазный коммит — это протокол, позволяющий выполнять транзакции на распределенной базе данных. На первой фазе *voting phase* менеджер транзакций координирует все транзакции будут ли они выполнены или отменены. На *commit phase* менеджер транзакции принимает решение о фиксации (commit) или отмене (abort) транзакции на основе ответа от каждого ресурса.

XA (eXtended Architecture) транзакция использует глобальный id транзакции (id) и локальный xid для каждого XA ресурса. Сначала менеджер проверяет что каждый ресурс готов к выполнению транзакции. Для каждого ресурса вызывается *prepare* метод возвращающий *OK* или *ABORT*. Если все ресурсы ответили *OK*, то начинается вызов *commit* для каждого xid.

У двухфазного коммита есть проблема с транзакцией, если происходит сбой на каком-то из ресурсов во время, второй фазы.

Two-phase commit

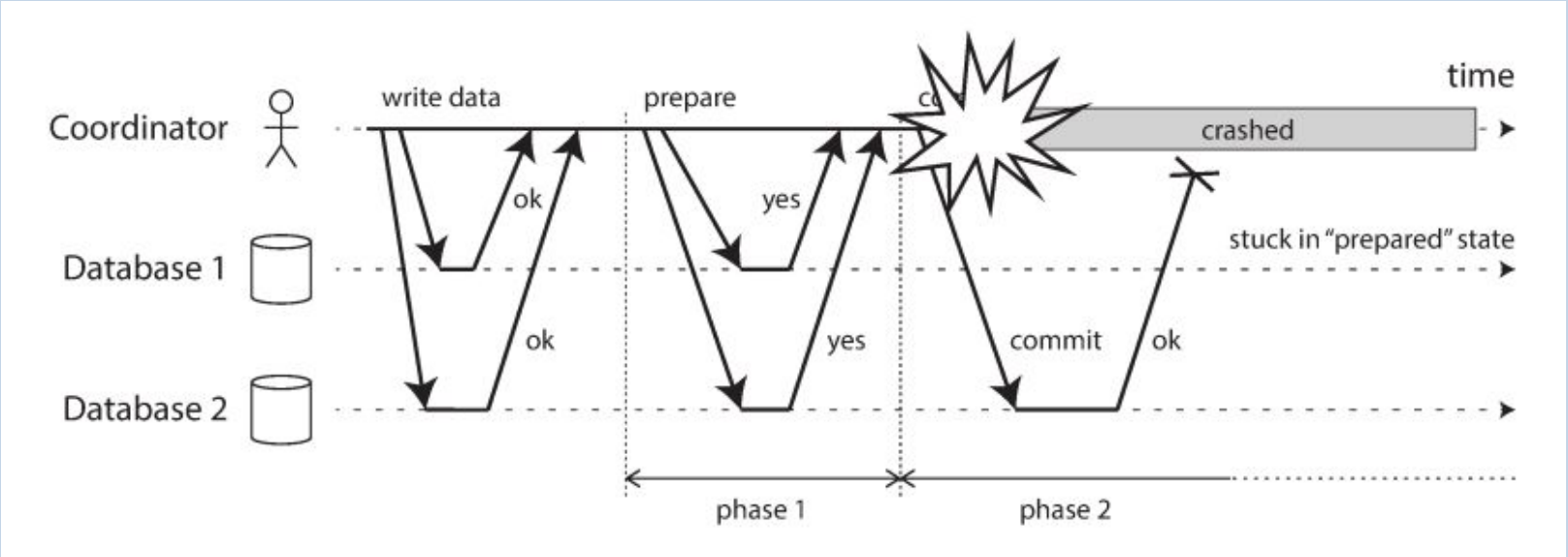


Two-phase commit

1. Получение глобально уникального id транзакции
2. Выполнение локальных транзакций с прикрепленным id
3. Когда приложение готово к коммиту координатор посылает *prepare* запрос каждой ноде по конкретной транзакции
4. Участники получив *prepare* запрос должны обеспечить выполнение комита при любых обстоятельствах
5. Когда координатор получил ответы от всех нод он делает решение коммита или аборта, свое решение он должен записать в лог транзакций, этот шаг - *commit point.*
6. Когда координатор принял решение, назад возврата нет и если он принял решение закомитить, он будет пытаться пока все ноды не закомитят свои изменения.

В данной схеме имеется 2 точки невозврата: когда участник сказал да и когда координатор принял решение о коммите.

Two-phase commit failure

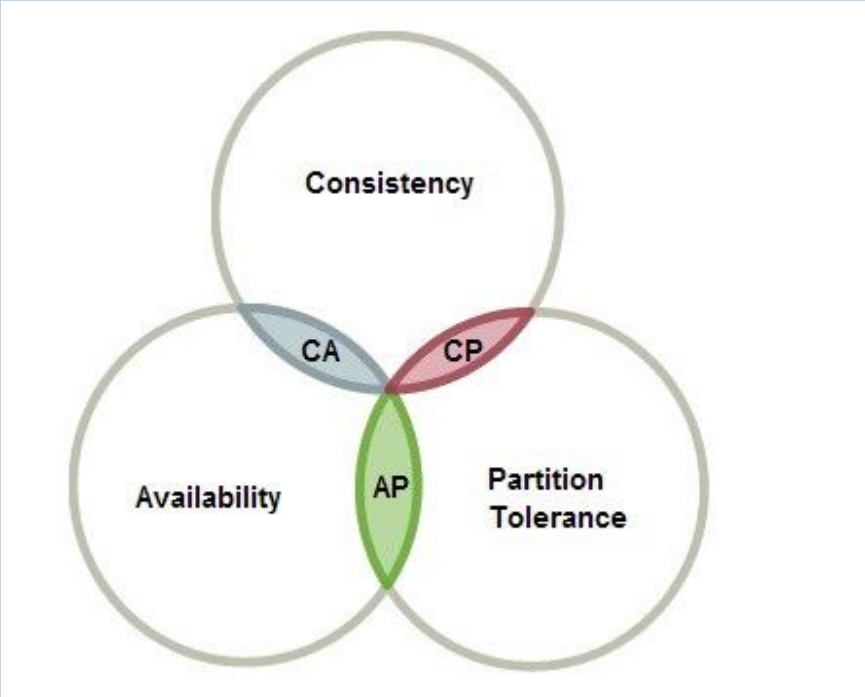


CAP

CAP теорема - в изначальной трактовке говорит о том, что нельзя одновременно обеспечить свойства консистентности, доступности и устойчивости при отказах нод.

На практике:

- Нельзя исключить *P*
  на самом деле выбор между *C* и *A*
- *C* и *A* не дискретны



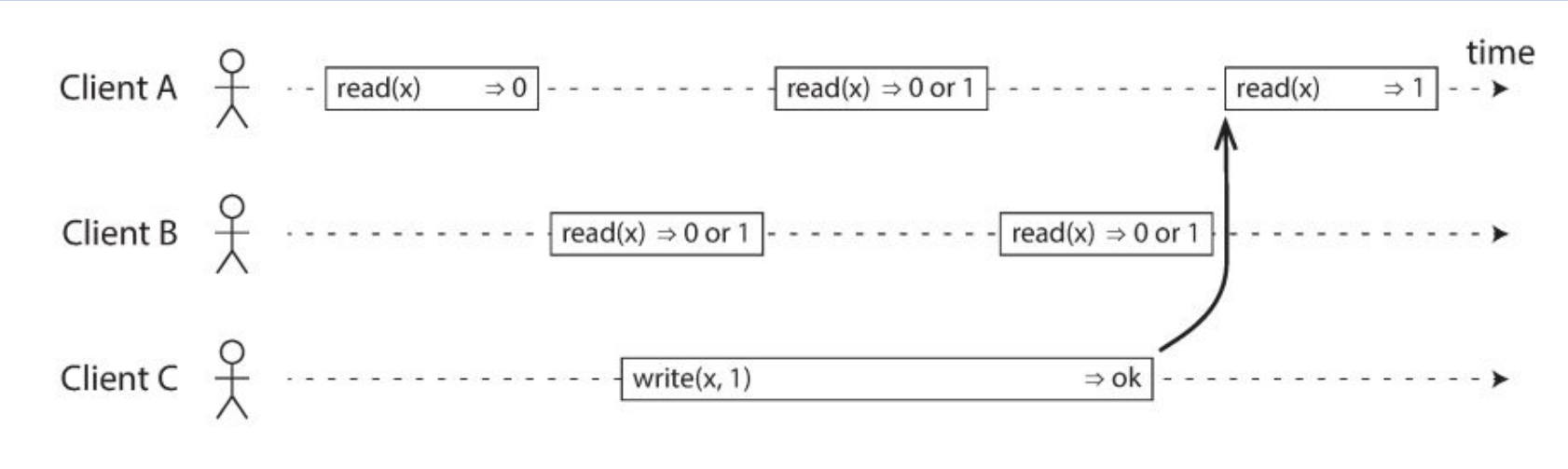
CAP vs ACID

- *CAP* - модель построена на операциях над *single write-read register* в отличии от *ACID,* где подразумеваются транзакционные операции, выполняющиеся атомарно над строками значений

- *C* - обозначают разную консистентность, в случае *ACID* подразумевается переход системы из одного согласованного состояние в другое согласованное, но на самом деле ответсетнность за согласованный переход лежит на уровне самого приложения, а не базы данных
  В случае *CAP*подразумевается Linearizable consistency

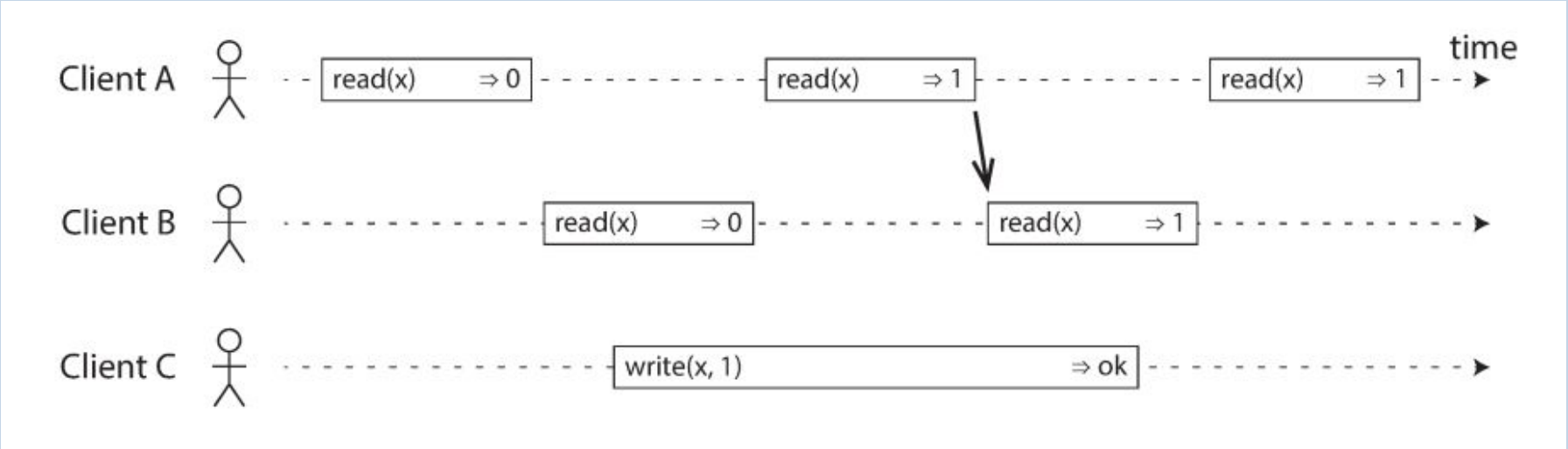
Linearizable consistency

Linearizable уровень консистентности при котором система ведет себя как будто существует одна копия данных над которой проводятся операции, то есть клиент не должен видеть никаких эффектов возникающих при репликации
Как быть в concurrent операциями?



Recency гарантия

Linearizable уровень консистентности при котором система ведет себя как будто существует одна копия данных над которой проводятся операции, то есть клиент не должен видеть никаких эффектов возникающих при репликации



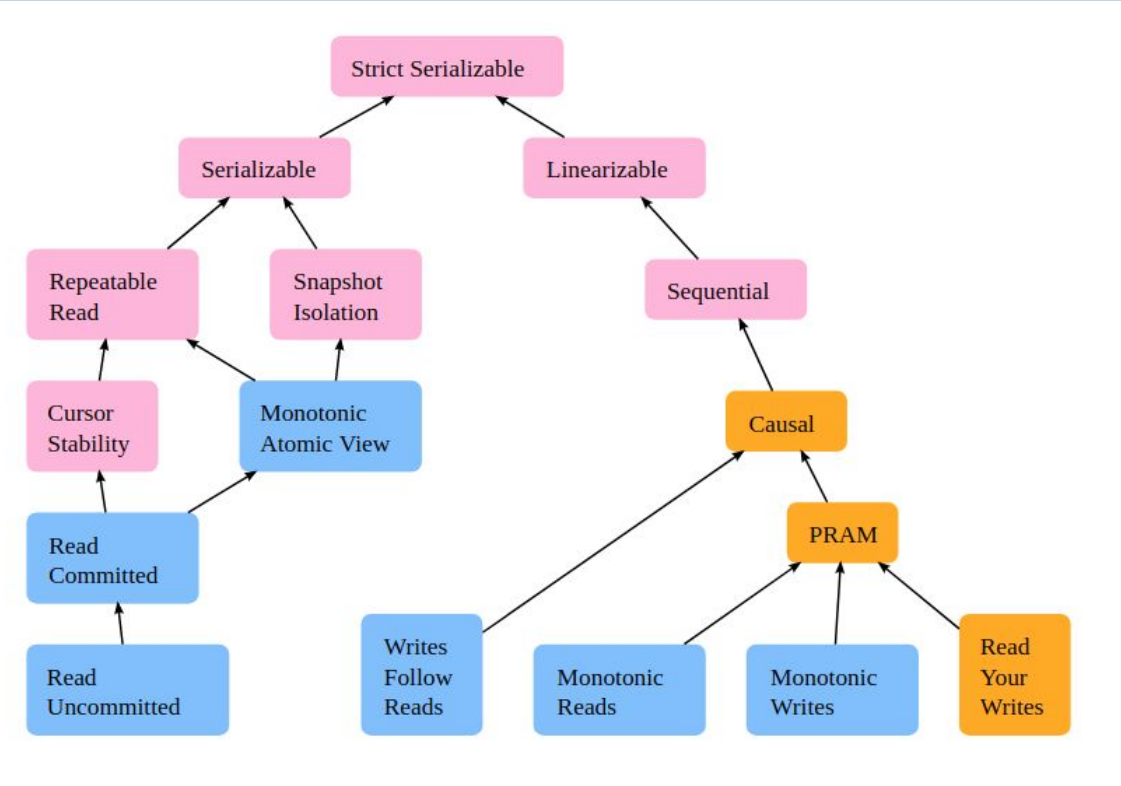
Где нужны гарантии Linearizable?

- *Locking and leader election*
- *Constraints and uniqueness guarantees*
- *Cross-channel timing dependencies*

Implementing Linearizable Systems?

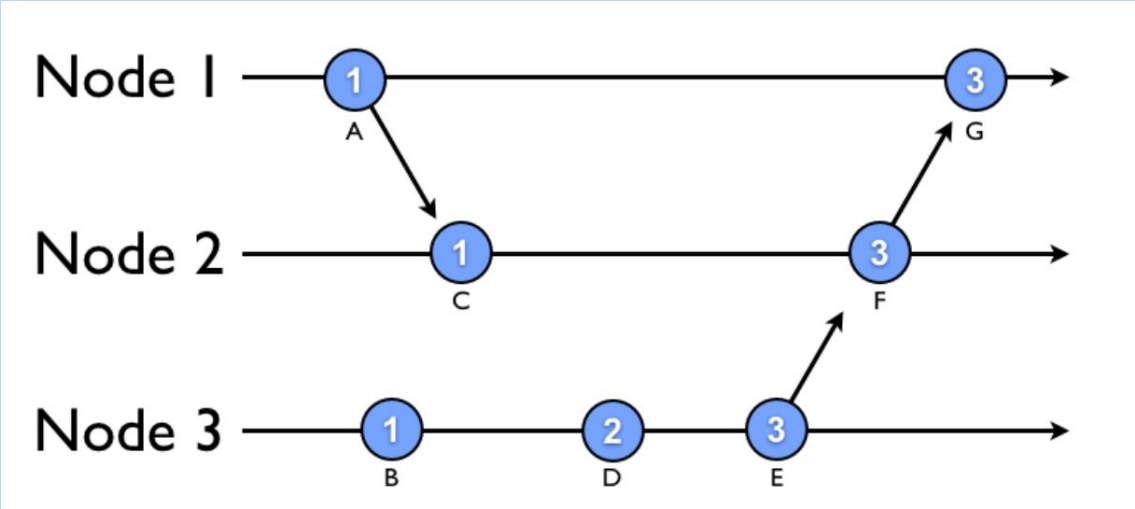
- *Single-leader replication (potentially linearizable)*
- *Consensus algorithms (linearizable)*
- *Multi-leader replication (not linearizable)*
- *Leaderless replication (probably not linearizable, cause LWW)*

Consistency models



Lamport timestamps

Отметки времени Лампорта - одна из попыток упорядочить события в распределенных системах.
Каждая нода имеет свой счетчик, который увеличивается на 1 при любом событии,
а также при обмене сообщениями счетчики синхронизируются к большему



Vector clock

Векторные часы - обобщение идеи временных отметок лампорта, позволяет ввести частичный порядок событий в системе