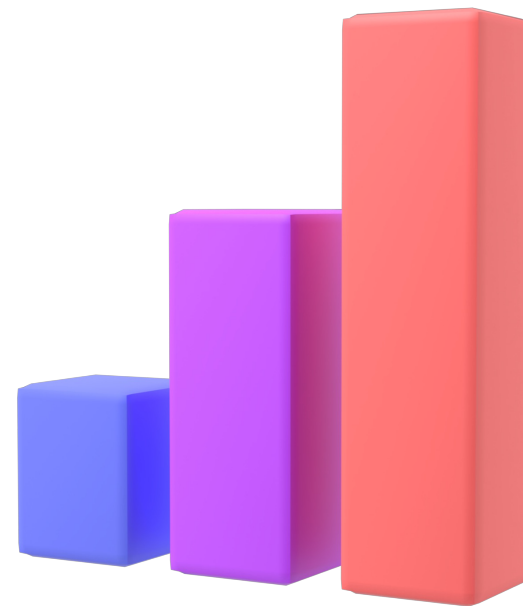


# System Design

Распределенное хранение данных

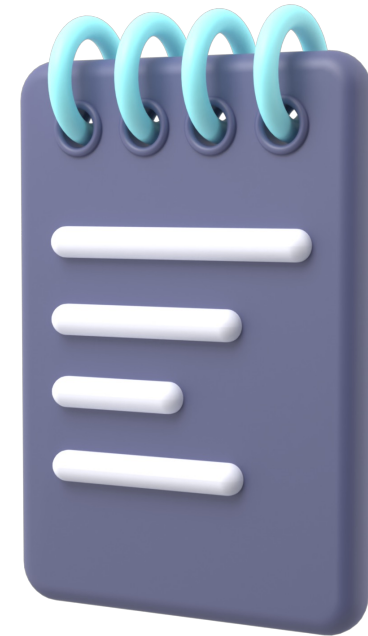


**Проверь запись**



# Маршрут занятия

- Репликация
- CAP теорема
- Партиционирование
- Шардирование
- Дополнительное



# Репликация

# Репликация и бэкапы

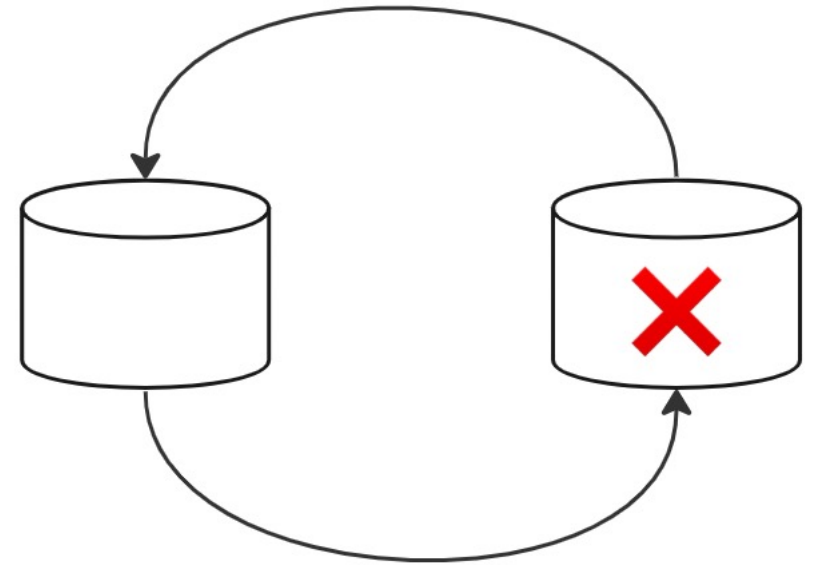
Бэкап – это **резервное копирование** содержимого диска с целью последующего восстановления.

**Репликация** – это создание клона базы данных для быстрого подхвата функций поврежденной системы



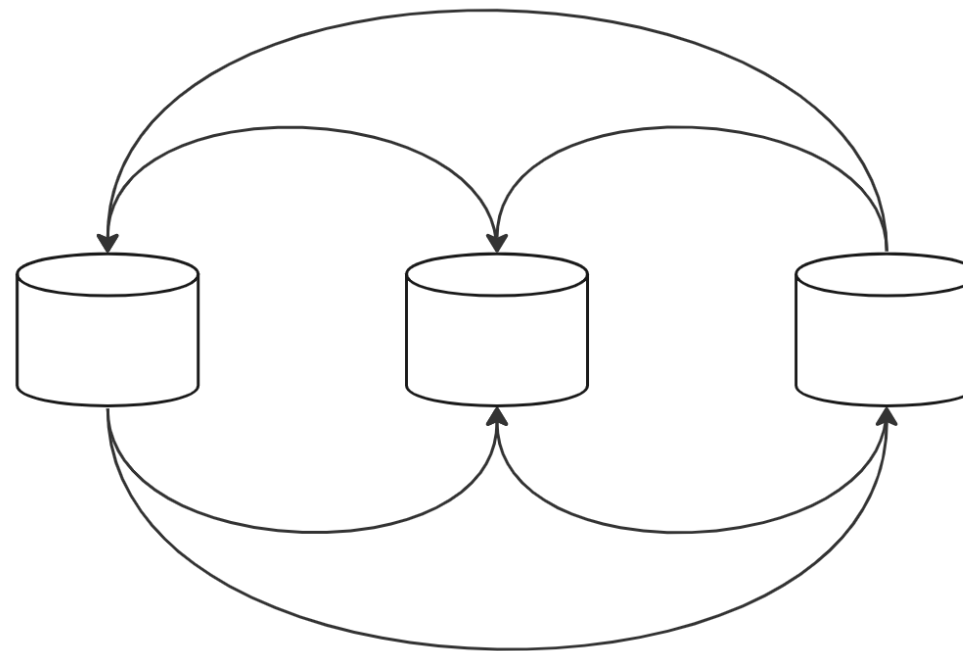
# Надежность

Поддержка резервной базы данных на случай потери основной



# Масштабирование чтения

Снижение нагрузки на чтение за счет переноса части запросов на реплики



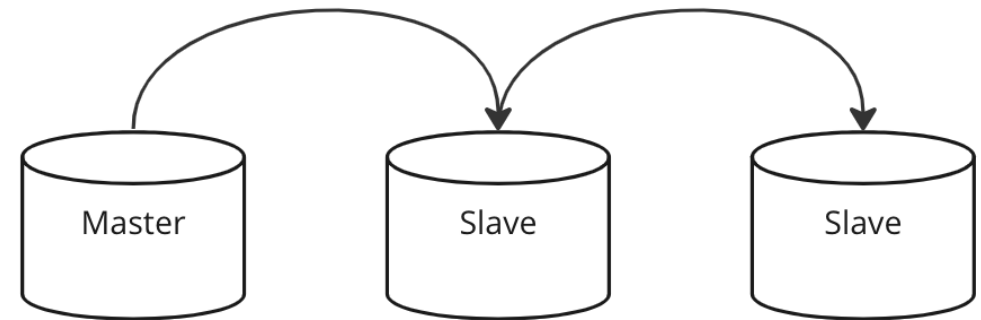
# **Виды ролей в репликации**

В какие базы данных пишем и из каких читаем?

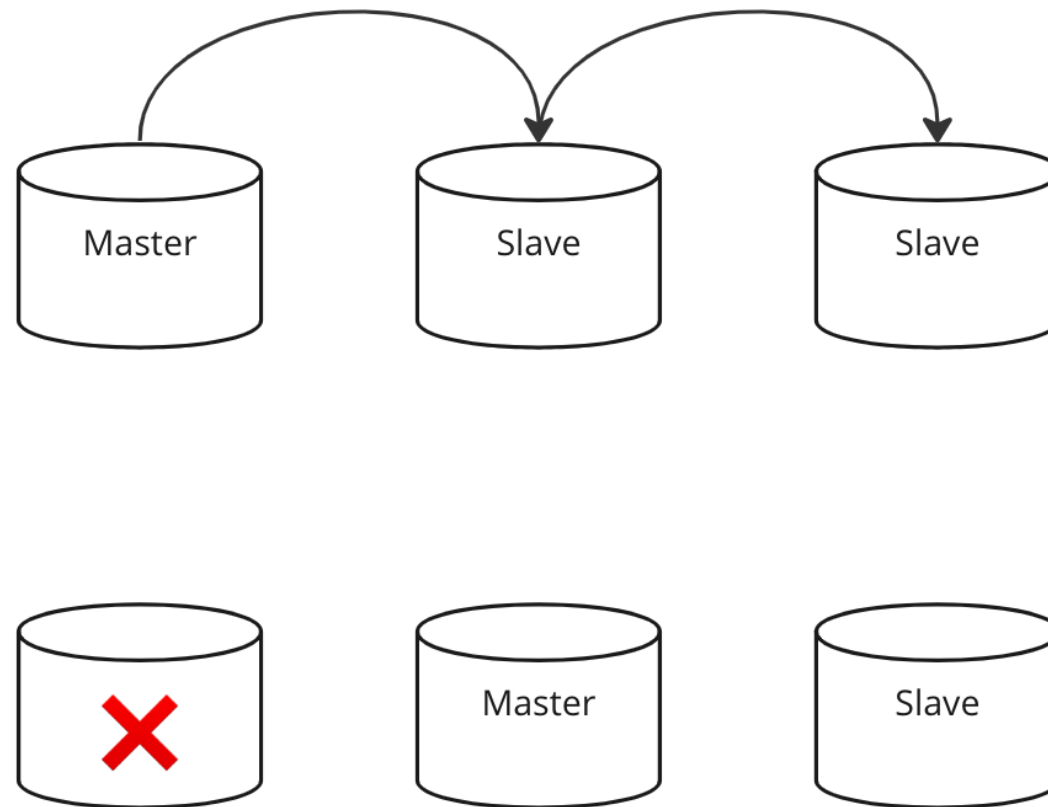


# Master - slave

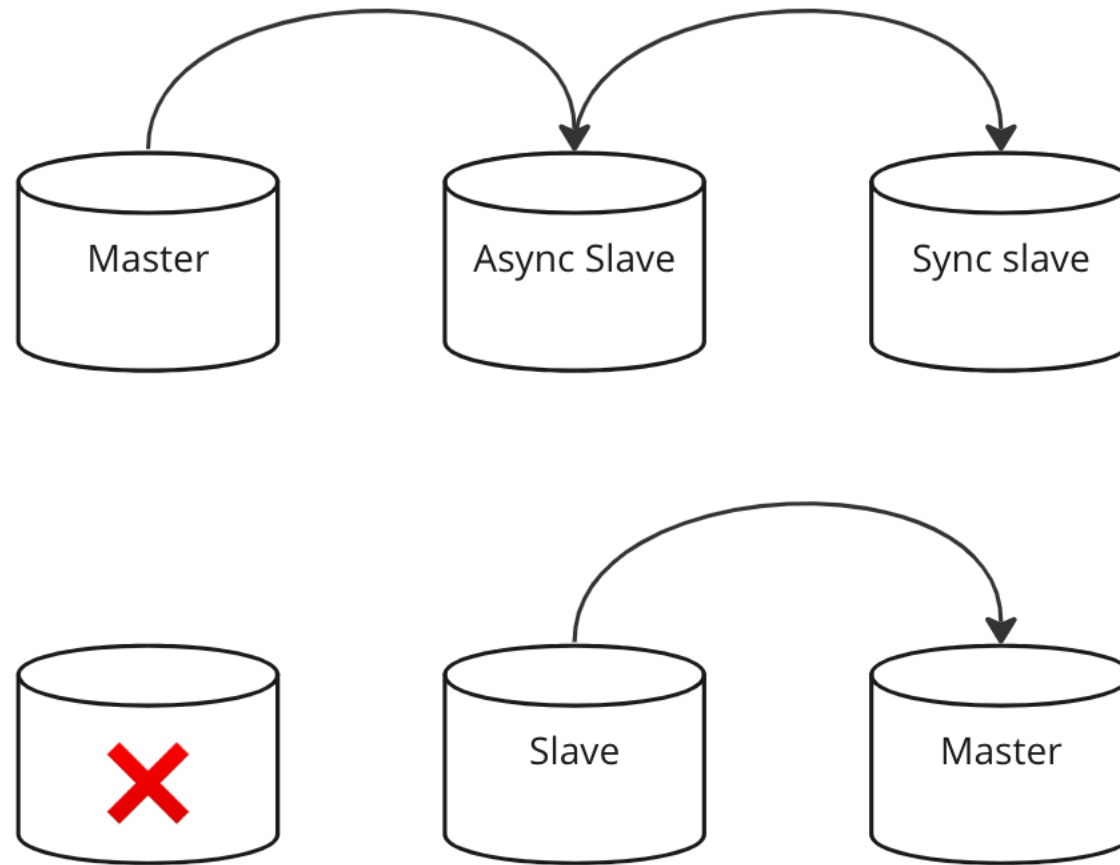
- Пишем в мастер
- Читаем из слейвов или из мастера
- В случае падения мастера получаем downtime на запись



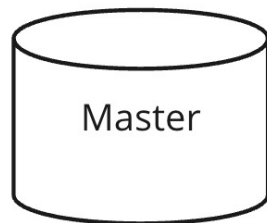
# Failover



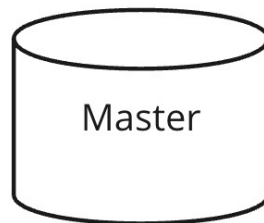
# Hot Standby



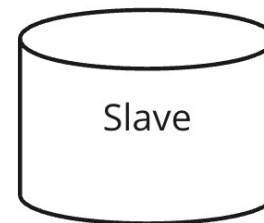
# Split Brain



Master



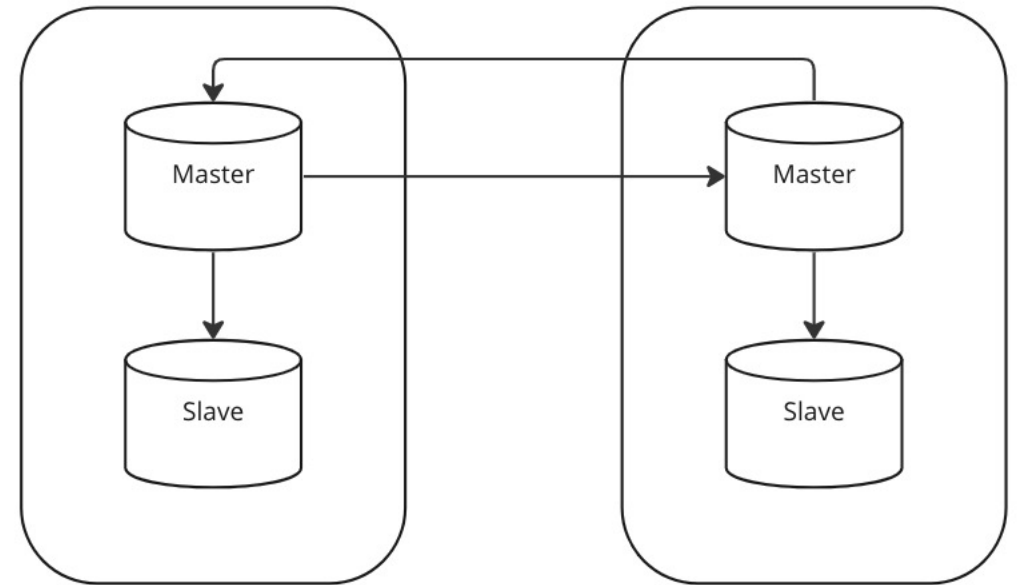
Master



Slave

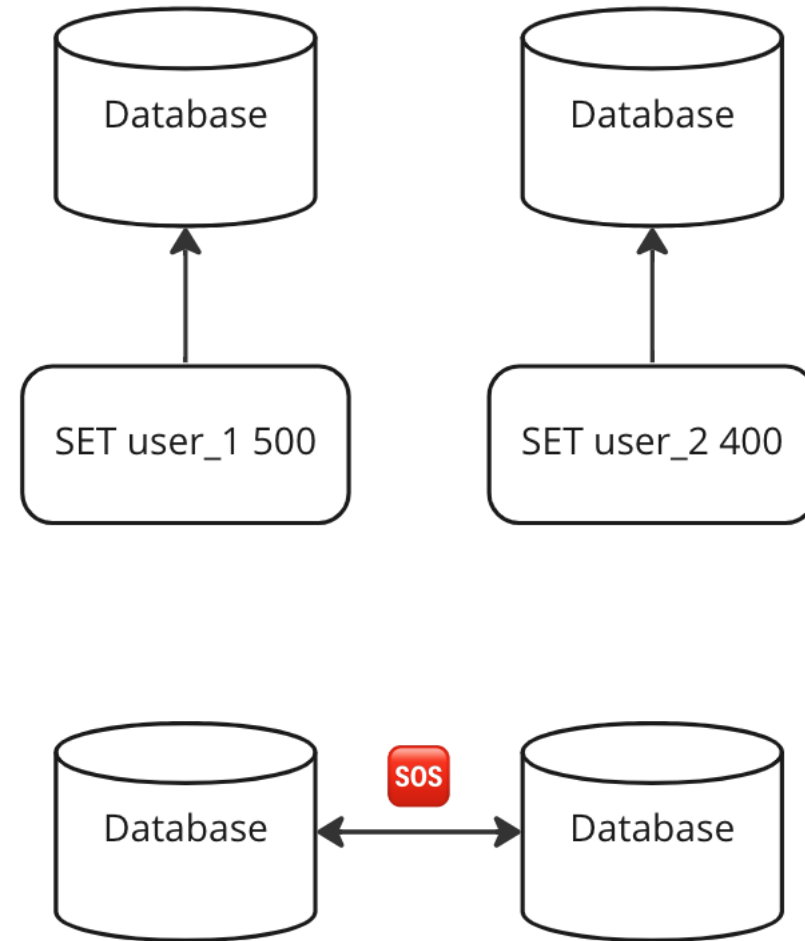
# Master - master

- Появляются конфликты
- Пишем в разные мастера
- Читаем из слейвов или мастеров
- В случае падения мастера нет никакого downtime на запись



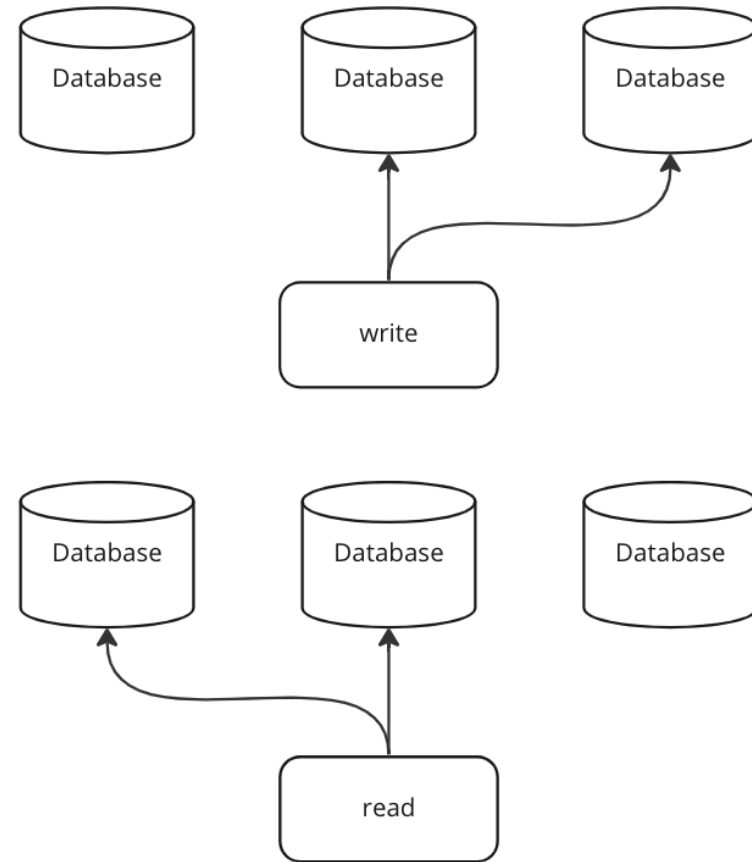
# Конфликты

- LWW
- Ранг реплик
- Решение конфликтов на клиенте
- Conflict-replicated data type (CRDT)

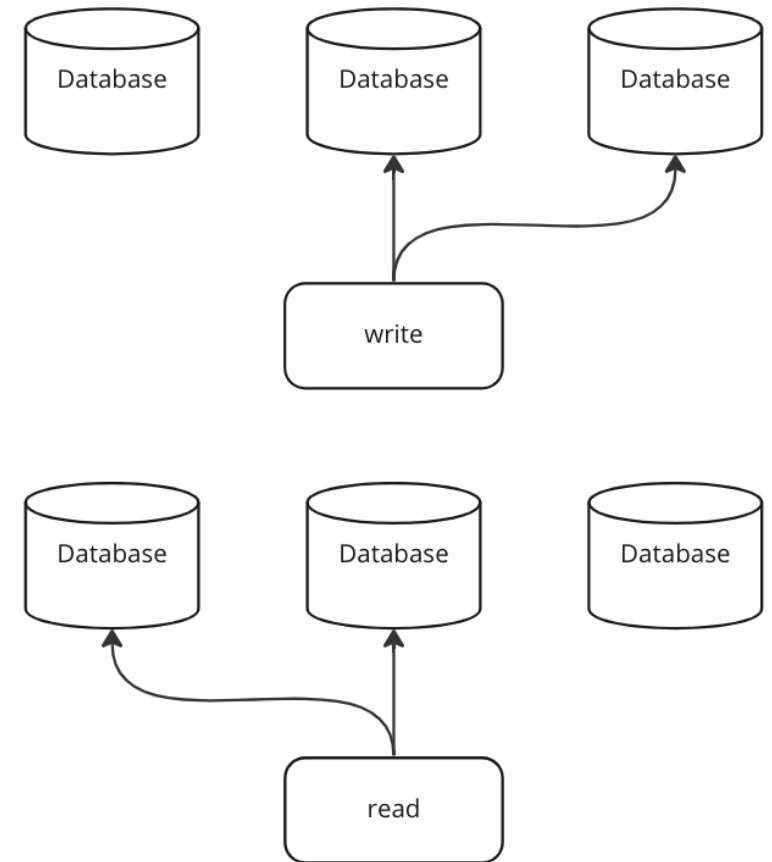


# Master - less

- Пишем в определенные ноды
- Читаем с определенных нод



- **$W + R > N$**  – гарантируется строгая согласованность
- **$W + R \leq N$**  – не гарантируется строгая согласованность
- **$R = 1$  и  $W = N$**  – система оптимизирована для быстрого чтения
- **$W = 1$  и  $R = N$**  – система оптимизирована для быстрой записи



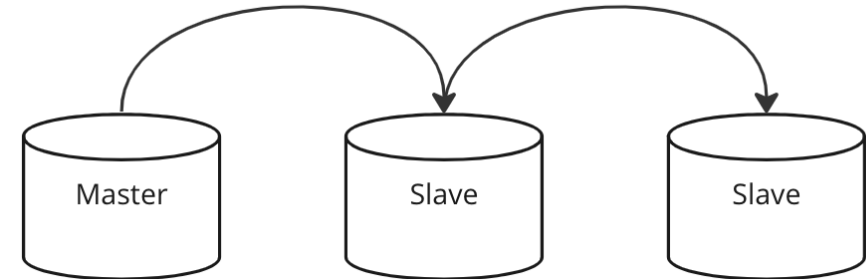


# Типы репликации

Когда передавать данные?

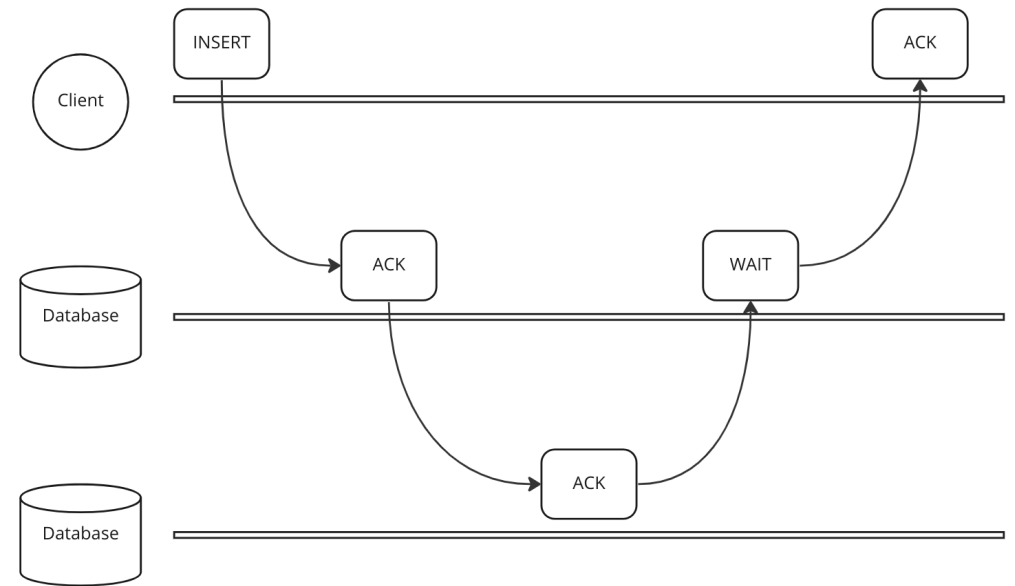
# Strong Consistency

Любая операция чтения из любого узла  
базы данных вернет последнюю операцию  
записи



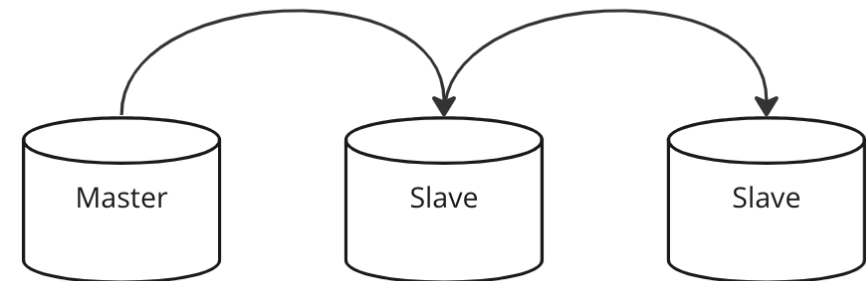
# Синхронная

1. Запись транзакции в журнал
2. Применение транзакции в движке
3. Отправка данных на все реплики
4. Получение подтверждения от всех реплик
5. Возвращение подтверждения клиенту



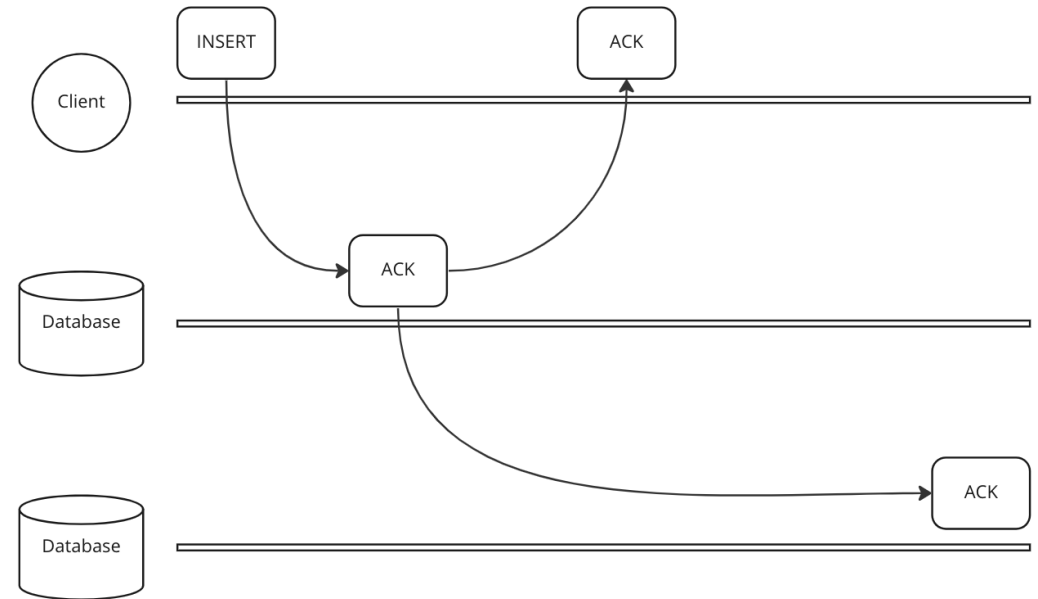
# Eventual Consistency

В отсутствии изменений данных, через какой-то промежуток времени после последнего обновления («в конечном счёте») все запросы будут возвращать последнее обновлённое значение

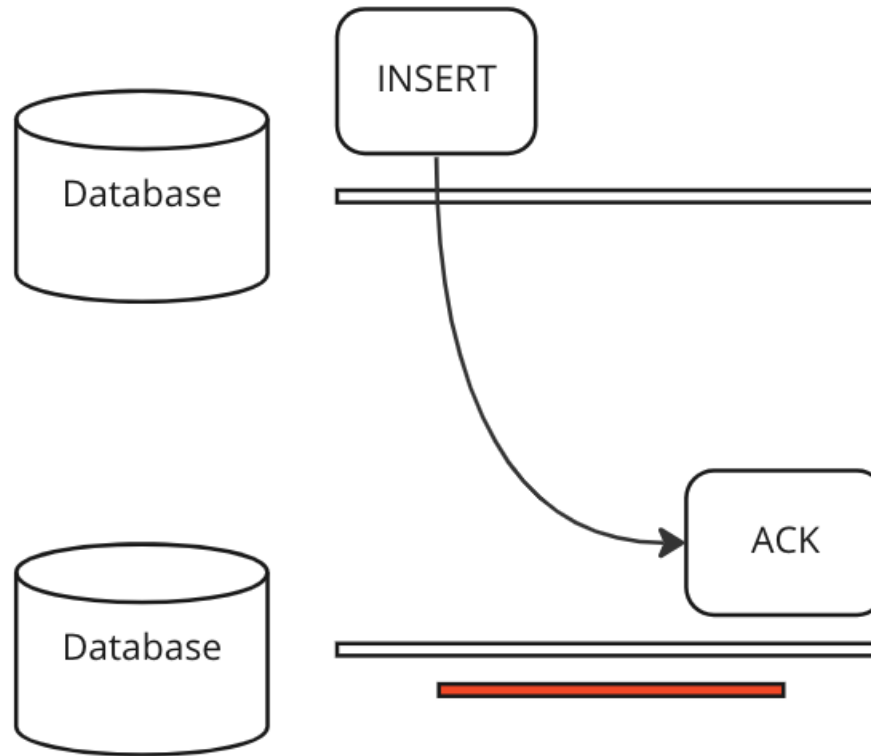


# Асинхронная

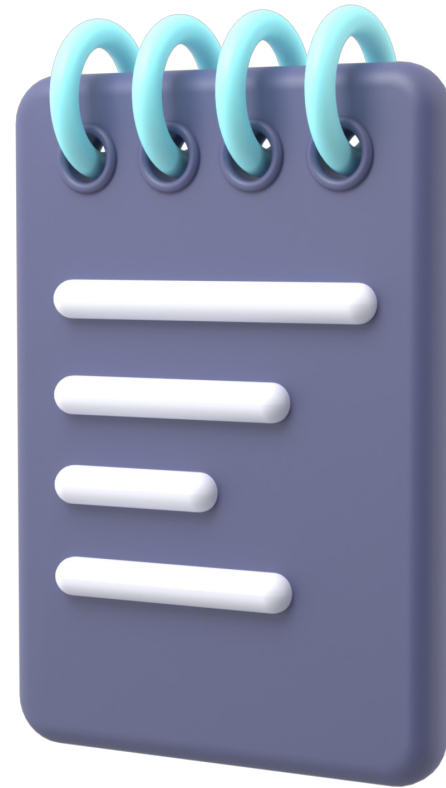
1. Запись транзакции в журнал
2. Применение транзакции в движке
3. Возвращение подтверждения клиенту
4. Отправка данных на реплики



# Replication Lag



Допустим, вы создаете клон Twitter.  
Пользователь может опубликовать твит со  
своего компьютера, который отправит  
запрос на запись в вашу распределенную  
базу данных.

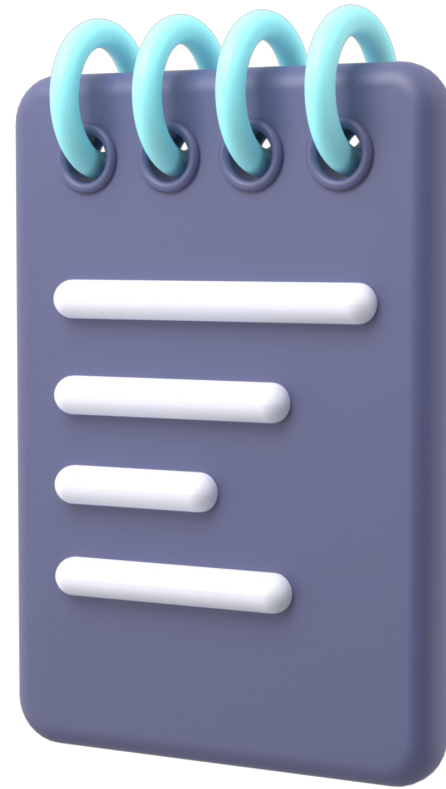


# **Чтение собственных записей**

Нужно отследить, когда пользователь в последний раз отправлял обновление. Если он отправил обновление в течение последней минуты, то запросы на чтение должен обрабатываться главным узлом



Возвращаясь к примеру с клоном Twitter, асинхронная репликация приведет к тому, что некоторые базы-фоловеры будут отставать от других узлов с точки зрения обновлений.

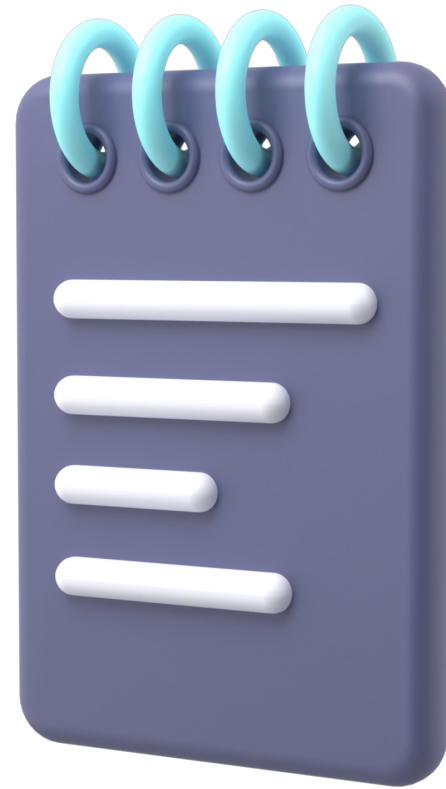


# Монотонное чтение

Обеспечить, чтобы каждый пользователь всегда читал из одного и того же узла-последователя (разные пользователи могут читать с разных реплик)

Допустим, у вас есть пользователь А и пользователь Б в вашем приложении-клонне Twitter. Пользователь А публикует в Твиттере фотографию своей собаки. Пользователь Б отвечает на это фото в твиттере комплиментом собаке.

Существует причинно-следственная связь между двумя твитами, когда ответный твит пользователя Б не имеет никакого смысла, если вы не видите твит пользователя А.

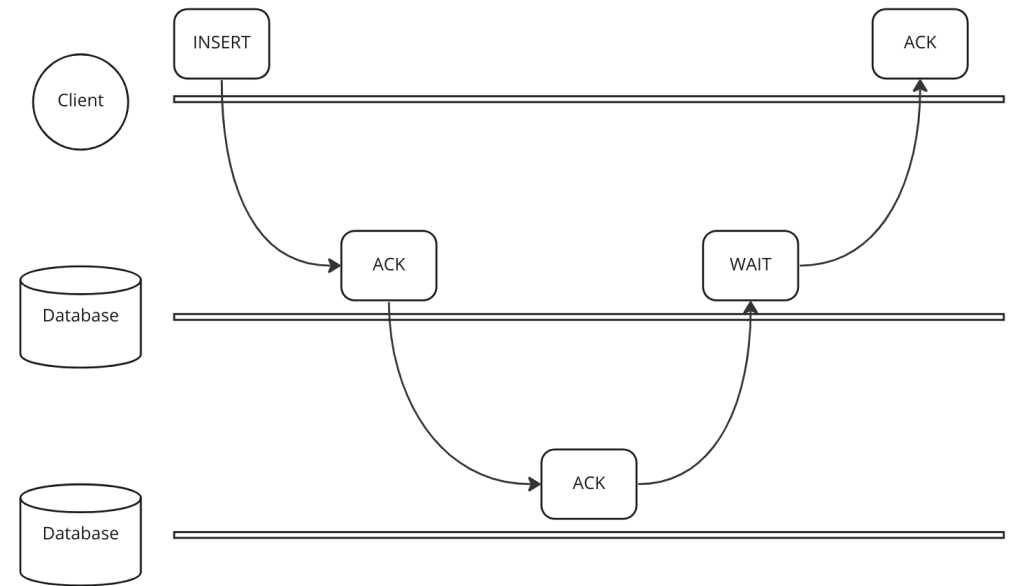


# **Согласованное префиксное чтение**

Гарантировать, что база данных всегда применяет операции в одном и том же порядке, а именно писать в одну и ту же секцию

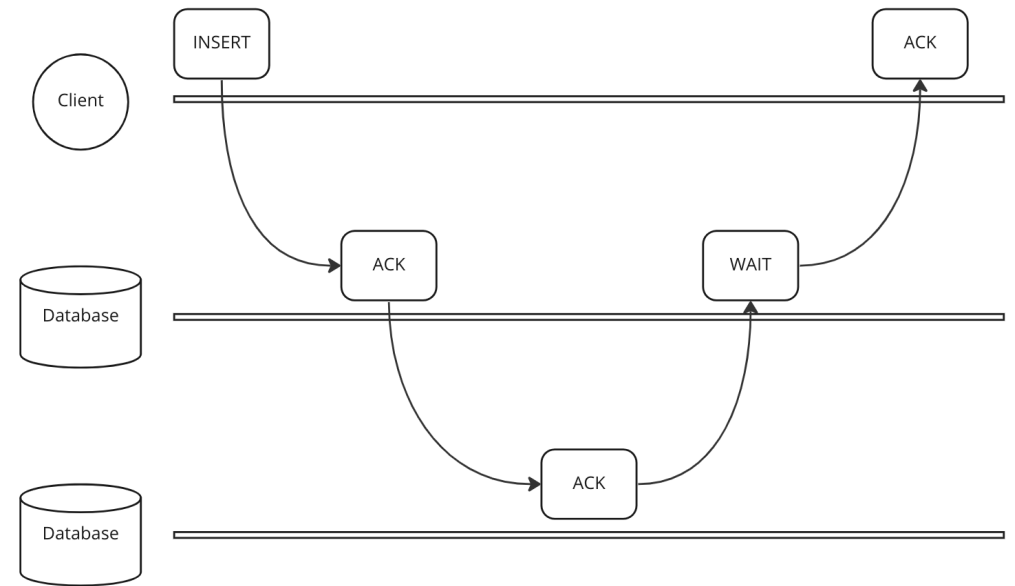
# Полусинхронная (semisync)

1. Запись транзакции в журнал
2. Применение транзакции в движке
3. Отправка данных на реплики.
4. Получение подтверждения от реплики о получении изменений (применены они будут «когда-то потом»)
5. Возвращение подтверждения клиенту



# Lose - less semisync

1. Запись транзакции в журнал
2. Отправка данных на реплики.
3. Получение подтверждения от реплики о получении изменений (применены они будут «когда-то потом»)
4. Применение транзакции в движке
5. Возвращение подтверждения клиенту



# **Форматы передачи данных**

Что будем и как будем передавать?

# Источник передачи данных

1. **push** – мастер рассылает данные репликам (PgSQL)
2. **pull** – реплики стягивают данные сами (MySQL)



# Statement base

1. Передаются запросы (но не сущности)
2. Каждый запрос считается на каждой ноде (random(),  
unix\_timestamp(), ...)

# Row based

1. Передаются измененные строчки в бинарном виде
2. Передаются сущности, даже если изменили одно поле  
(но можно настраивать full / minimal, ...)

# Mixed

База данных переключается из SBR в RBR или из RBR в SBR, в зависимости от той ли иной ситуации

# Логическая репликация

- Работает с кортежами (SBR, RBR)
- Не знает, как они хранятся на диске

# Физическая репликация

- Работает со страницами
- slave = master (байт в байт)

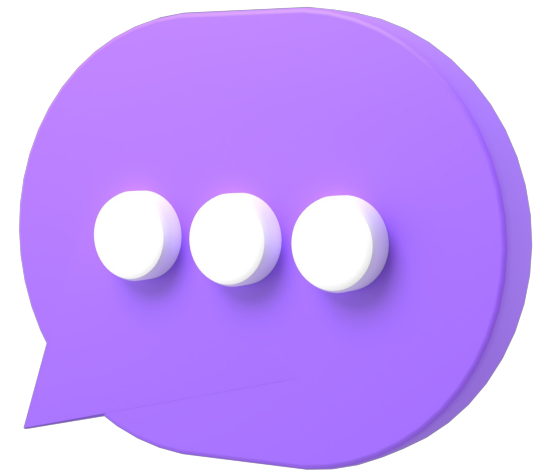
# Фильтрация репликаций

Можно реплицировать данные частично

# FAQ:

## Репликация

- Синхронная, асинхронная, полусинхронная
- master slave, master master, master less
- Способы передачи данных



# **CAP теорема**

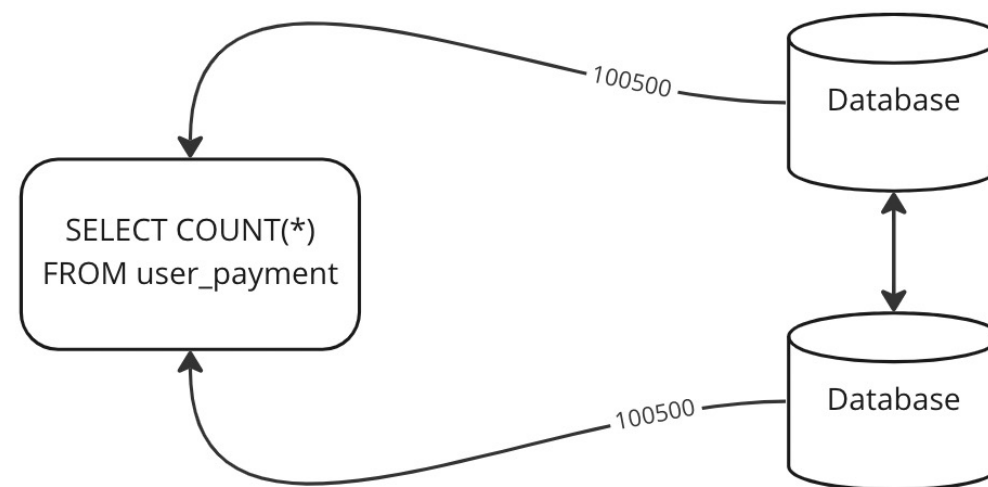


# CAP теорема

Утверждение о том, что в любой реализации распределенных вычислений возможно обеспечить не более двух из трёх следующих свойств (согласованность, доступность, устойчивость к разделению)

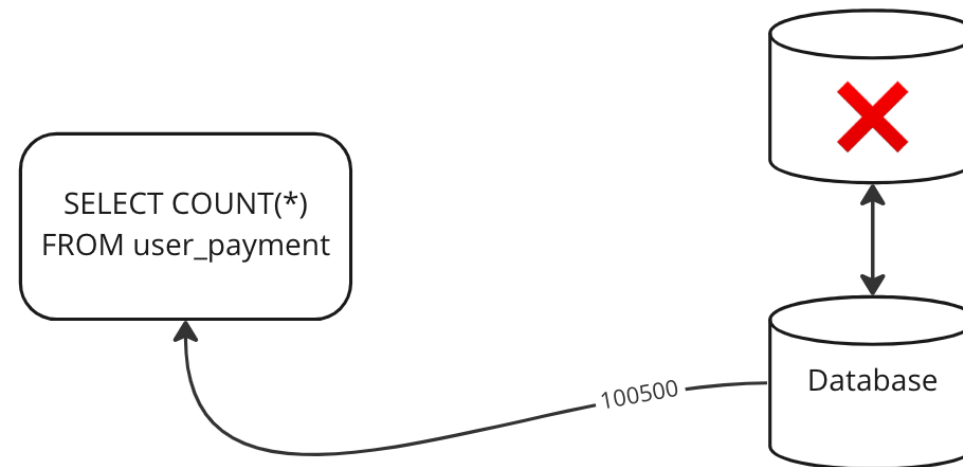
# Консистентность

Данные во всех нодах одинаковы



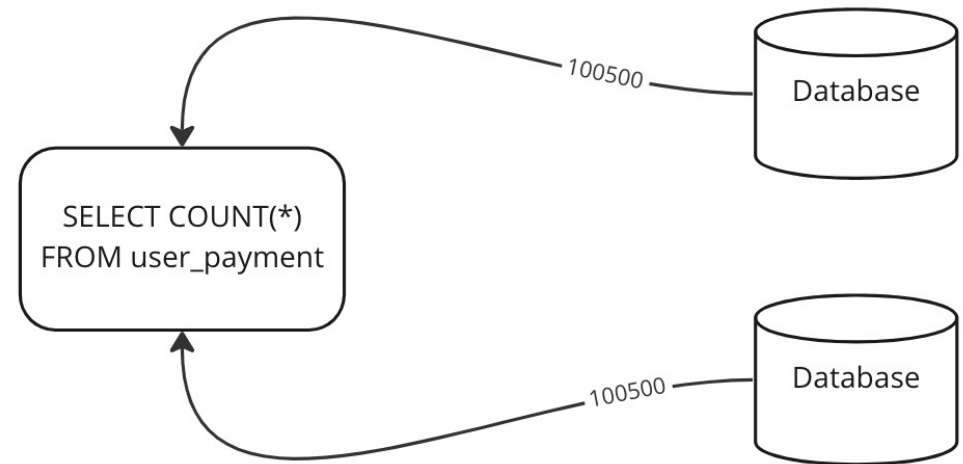
# Доступность

Если запрос пришел на живую ноду,  
запрос будет получен за конечное время



# Устойчивость к разделению

Продолжаем работать не смотря  
на отсутствие связи



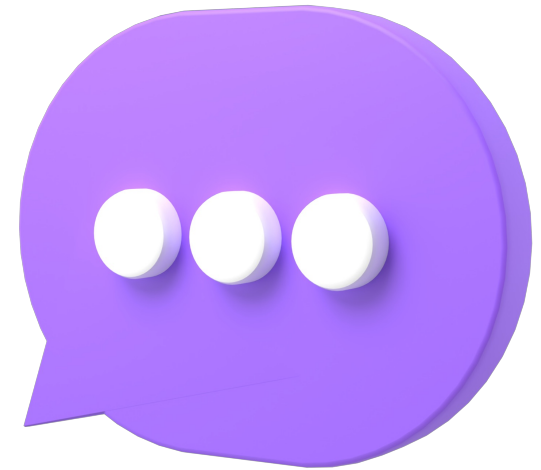
# Обрубаем связь

1. Наш мир не идеален, поэтому Р должна быть – **АС система**
2. Разрешаем из нод читать, но запрещаем читать – **СР система**
3. Разрешаем читать и писать – **АР система**

# FAQ:

## CAP теорема

- Консистентность
- Доступность
- Устойчивость к разделению



# **Партиционирование**

# Партиционирование

Метод разделения больших таблиц на много маленьких, и желательно, чтобы это происходило прозрачным для приложения способом (**секции находятся на одном и том же инстансе базы данных**)





# Вертикальное

ID	Name	Status	Description	Photo
10				
20				
30				
50				

ID	Name	Status
10		
20		
30		
50		

ID	Description	Photo
10		
20		
30		
50		

# Горизонтальное

ID	Name	Status	Description	Photo
10				
20				
30				
50				

ID	Name	Status	Description	Photo
10				
20				

ID	Name	Status	Description	Photo
30				
50				

# **FAQ:**

## **Партиционирование**

- Вертикальное
- Горизонтальное



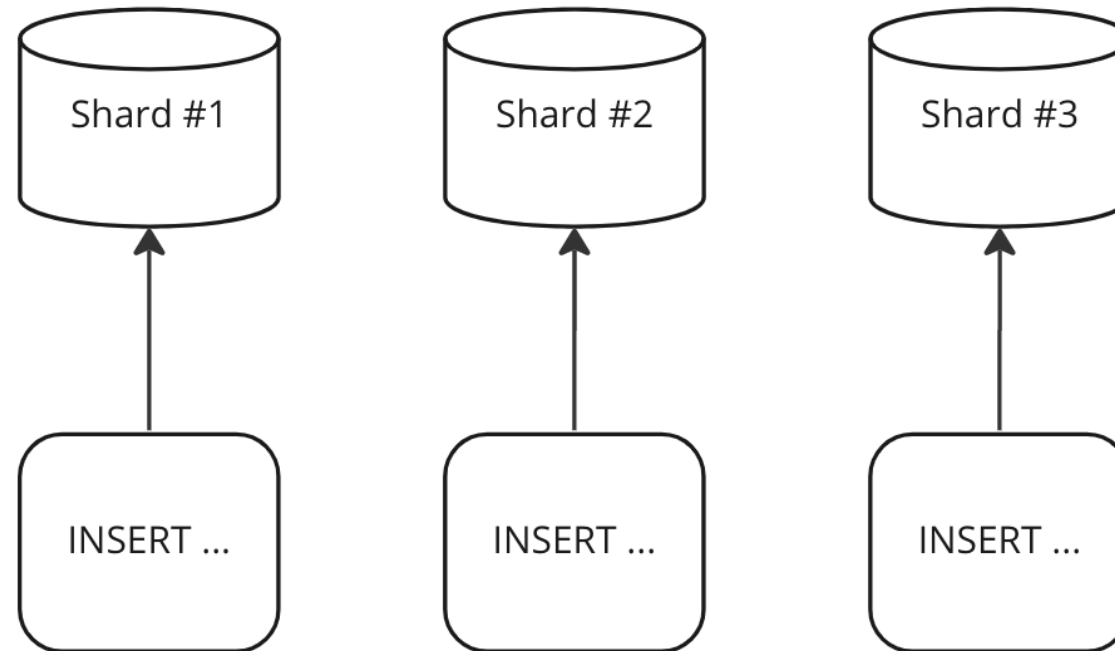
# Шардирование

# Шардирование

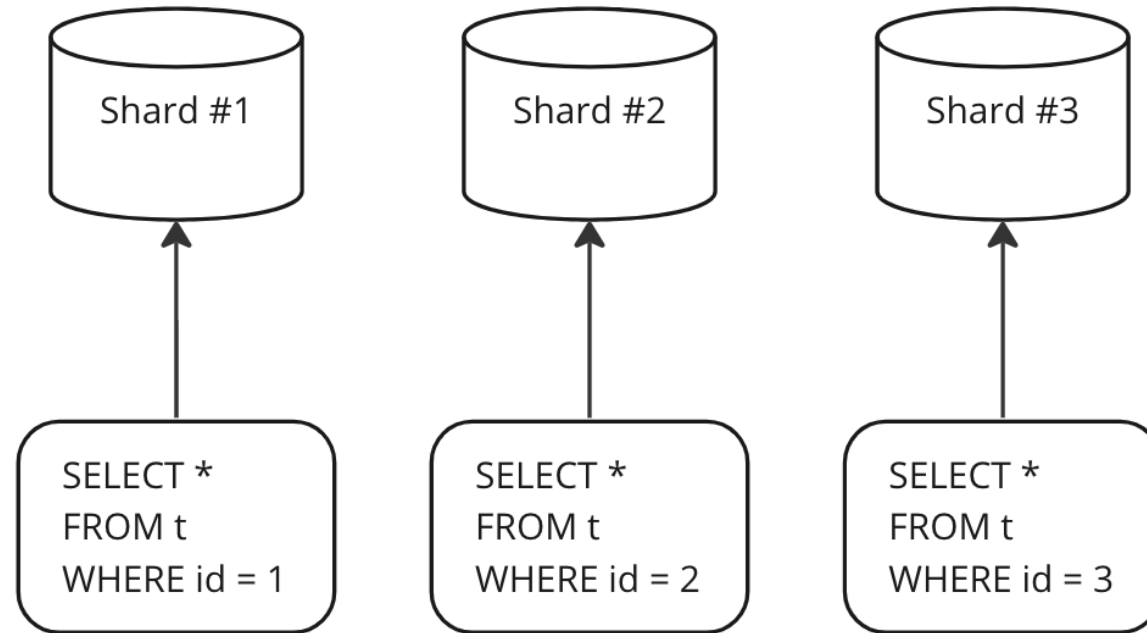
Подход, предполагающий разделение таблиц на независимые сегменты, каждый из которых управляется отдельным инстансом базы данных



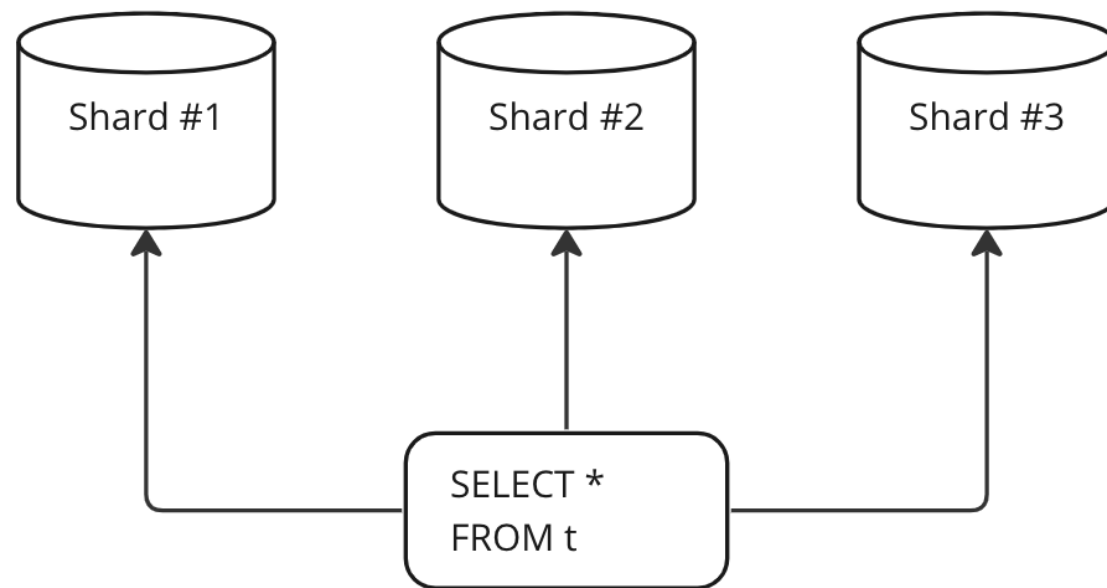
# Шардирование



# Шардирование

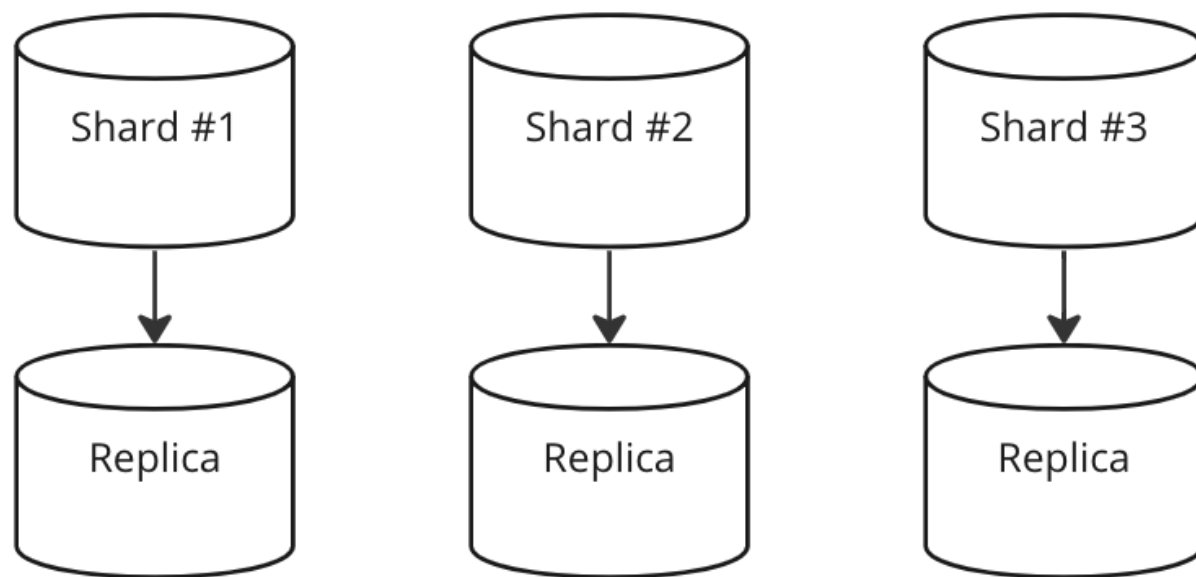


# Шардирование





# Шардирование



# **Способы шардирования**

Как распределять данные между шардами?

# Range based

ID	Price
10	30
20	57
30	64
50	123

**Shard #1 (0...50)**

10	30
----	----

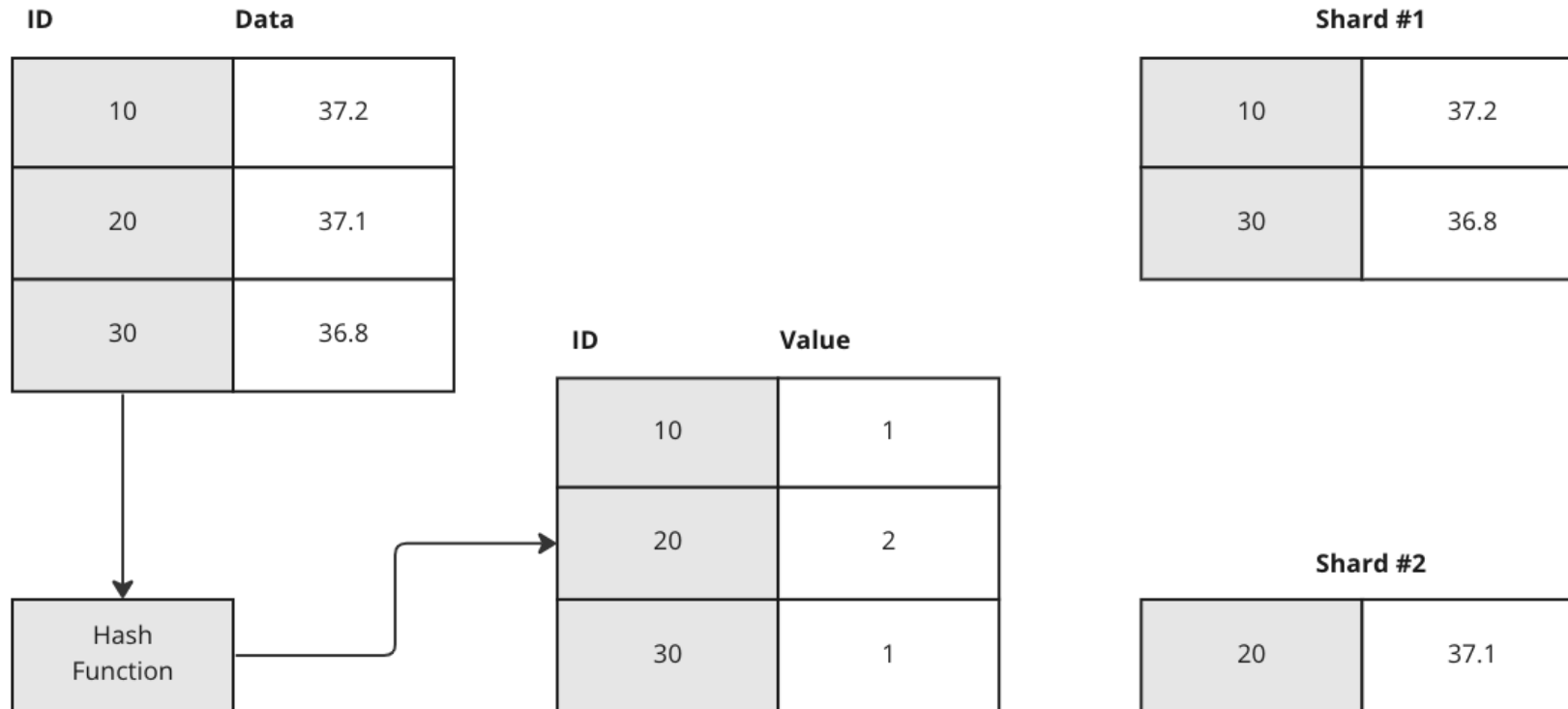
**Shard #2 (50...100)**

20	57
30	64

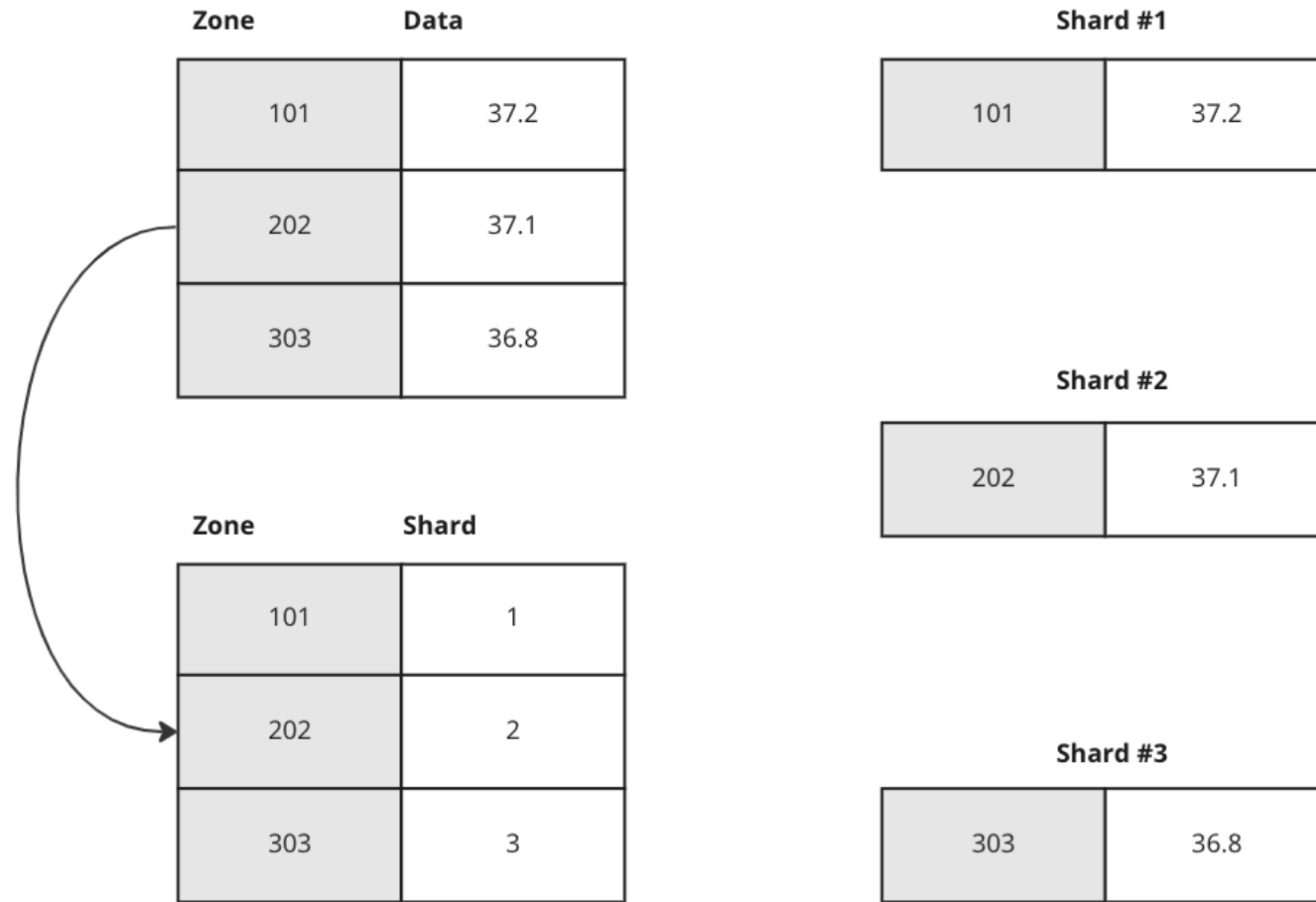
**Shard #3 (100+)**

50	123
----	-----

# Key based



# Directory based

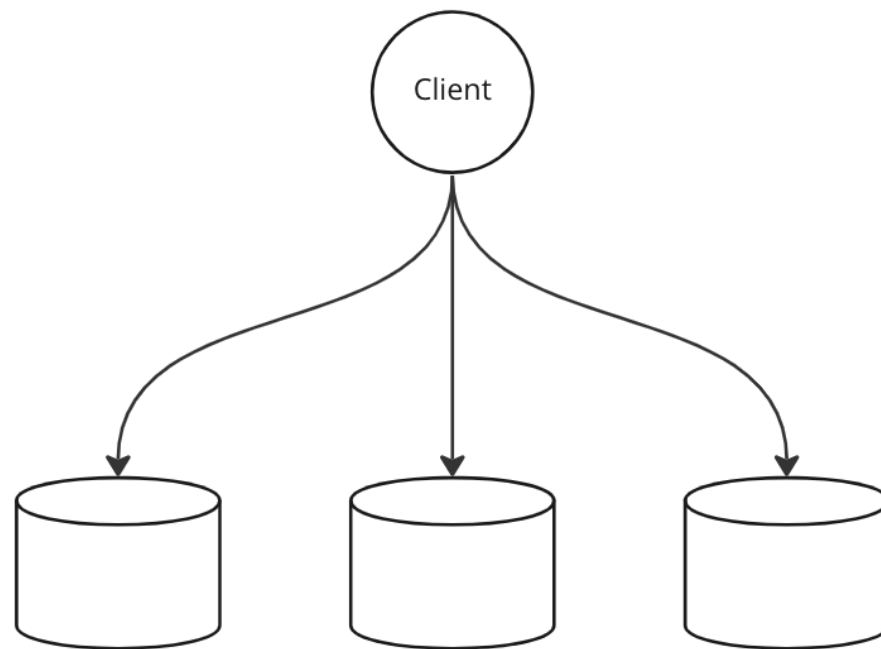


# Routing

Как понять куда идти?

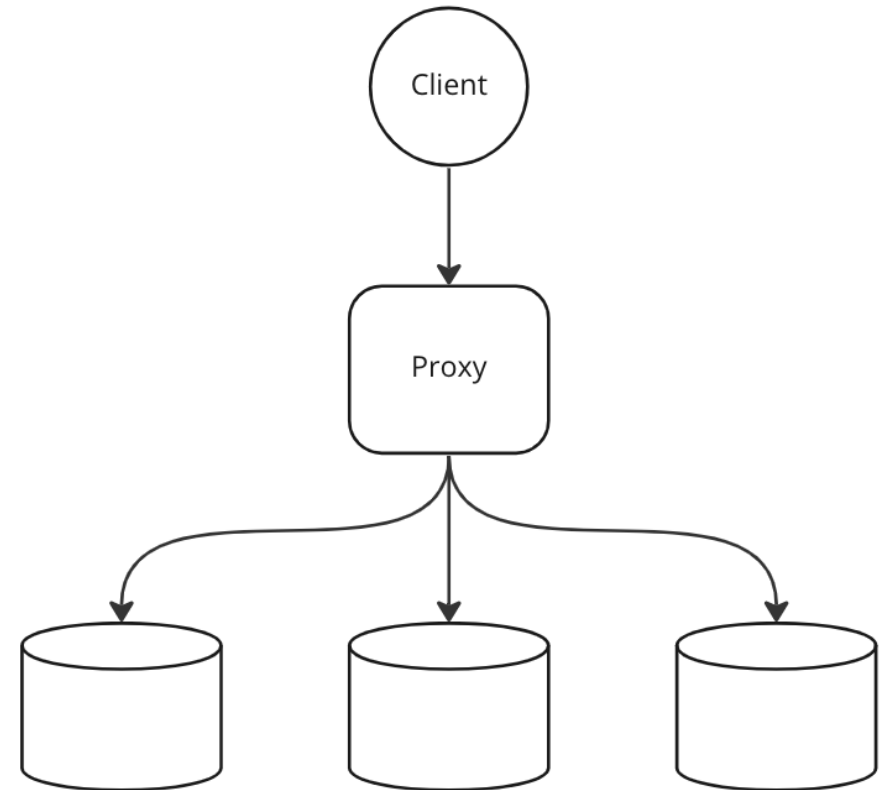
# Клиентский

- + нет лишних узлов
- дополнительная логика в клиенте
- сложности с обновлением хостов



# Proxy

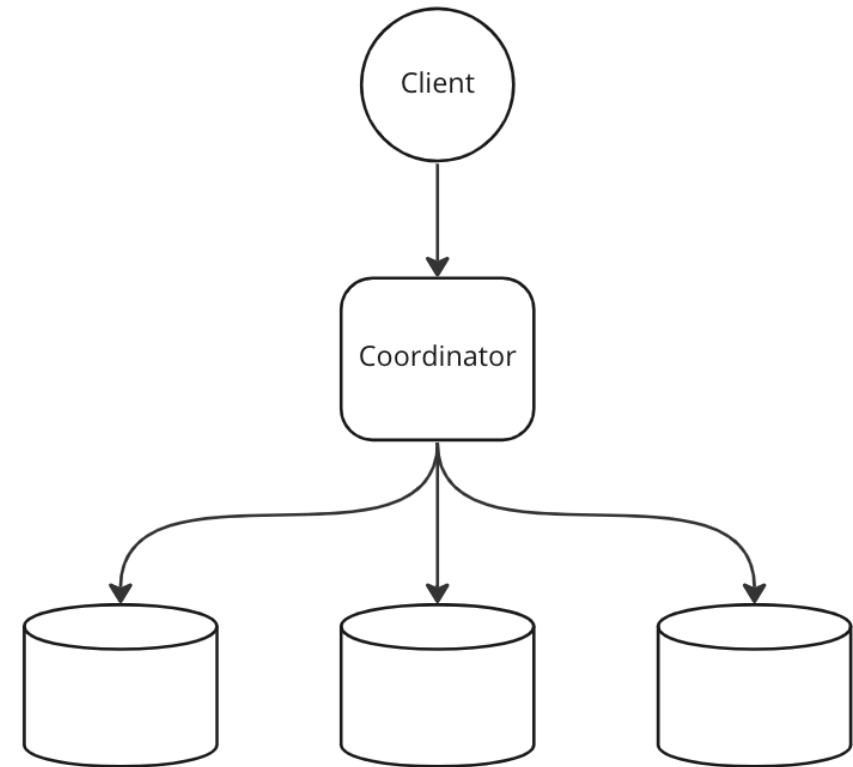
- + приложение не знает о шардинге
- дополнительный сетевой узел
- потеря функциональности
- единичная точка отказа





# Coordinator

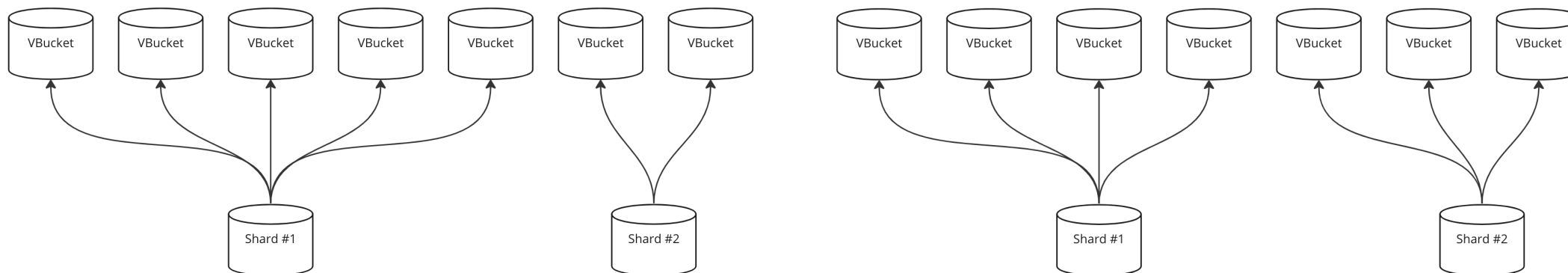
- + кэширование
- + приложение не знает о шардинге
- дополнительный сетевой узел
- инфраструктурная сложность
- единичная точка отказа
- нагрузка



# **Перебалансировка**

Как перенести данные из одного шарда на другой?

# Virtual buckets

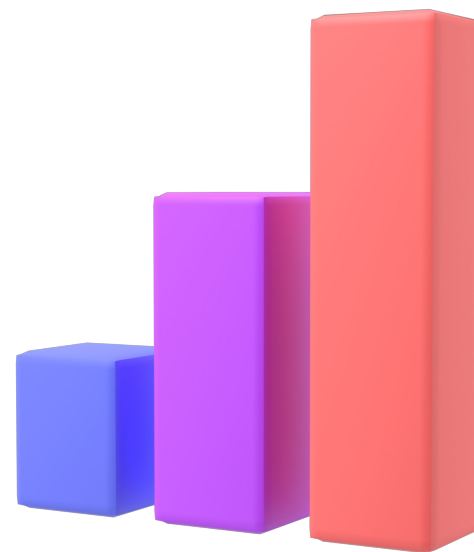


# Перебалансировка

1. Только чтение
2. Все данные неизменяемые (*пишем в tgt, читаем из src и tgt*)
3. Логическая репликация с src на tgt, после синхронизации переключаемся на tgt
4. Смешанный подход

# Resharding

- нужно добавить / удалить ноды
- исправление ошибок при выборе стратегии шардирования



# Hashing

$F(\text{key}) = \text{hash}(\text{key}) \% \text{shards\_number}$

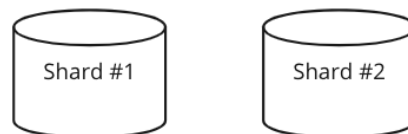


$$5 \% 3 = 2$$

$$6 \% 3 = 0$$

$$7 \% 3 = 1$$

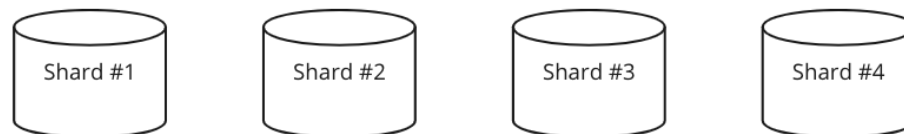
$F(\text{key}) = \text{hash}(\text{key}) \% \text{shards\_number}$



$$5 \% 4 = 1$$

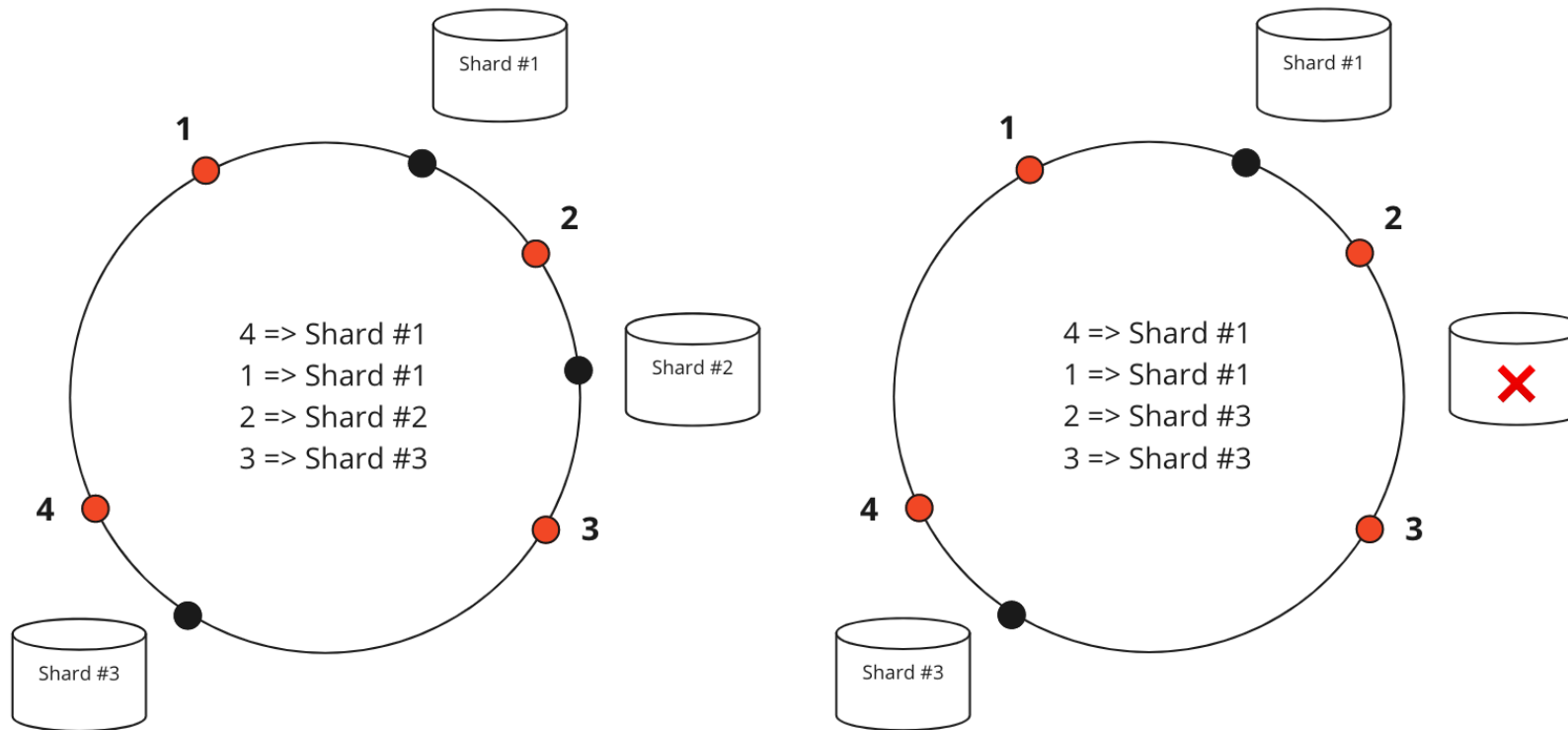
$$6 \% 4 = 2$$

$F(\text{key}) = \text{hash}(\text{key}) \% \text{shards\_number}$

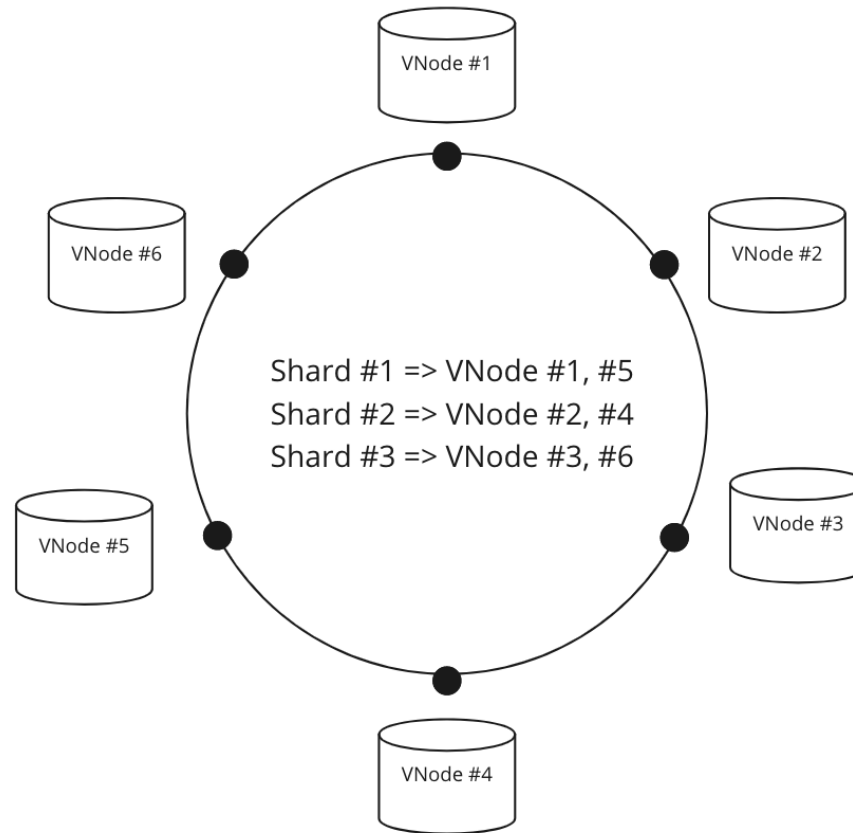


$$7 \% 4 = 3$$

# Consistent Hashing

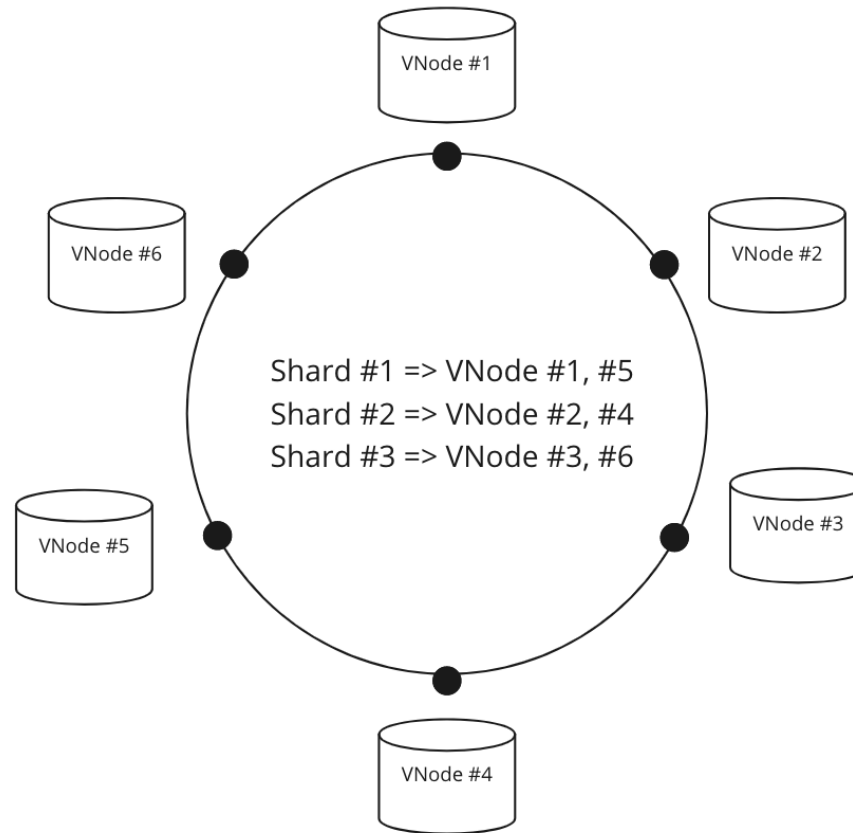


# Consistent Hashing





# Consistent Hashing



# Rendezvous Hashing

$F(123, 1) = \text{hash}(123, 1) = 345$

$F(123, 2) = \text{hash}(123, 2) = 456$

$F(123, 3) = \text{hash}(123, 3) = 121$



**123**

$F(242, 1) = \text{hash}(242, 1) = 233$

$F(242, 2) = \text{hash}(242, 2) = 124$

$F(242, 3) = \text{hash}(242, 3) = 434$



**123**

**542**

$F(123, 1) = \text{hash}(123, 1) = 345$

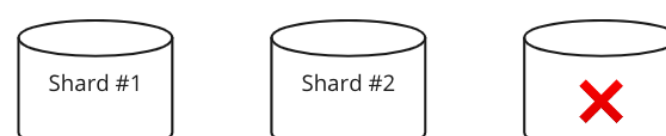
$F(123, 2) = \text{hash}(123, 2) = 456$



**123**

$F(242, 1) = \text{hash}(242, 1) = 233$

$F(242, 2) = \text{hash}(242, 2) = 124$



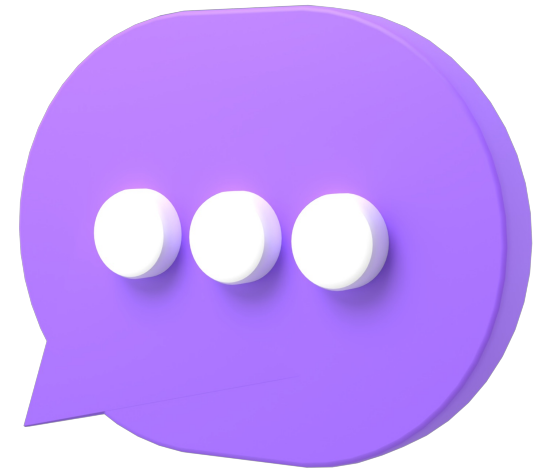
**542**

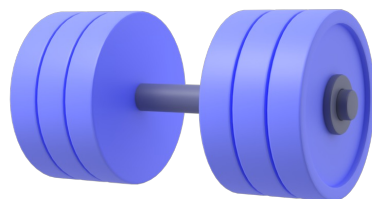
**123**

# FAQ:

## Шардирование

- Способ шардирования
- Routing
- Перевыброс
- Решардинг

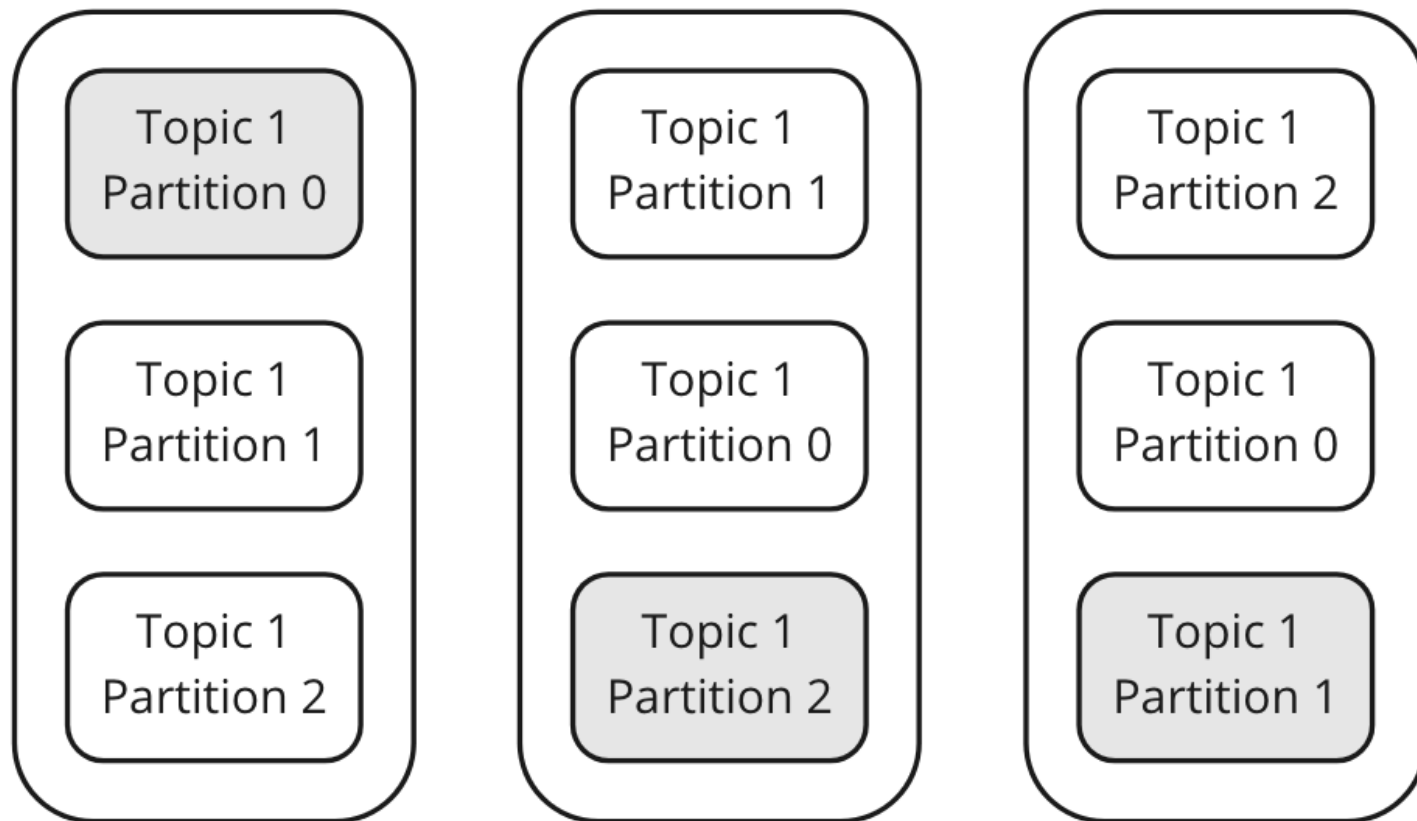




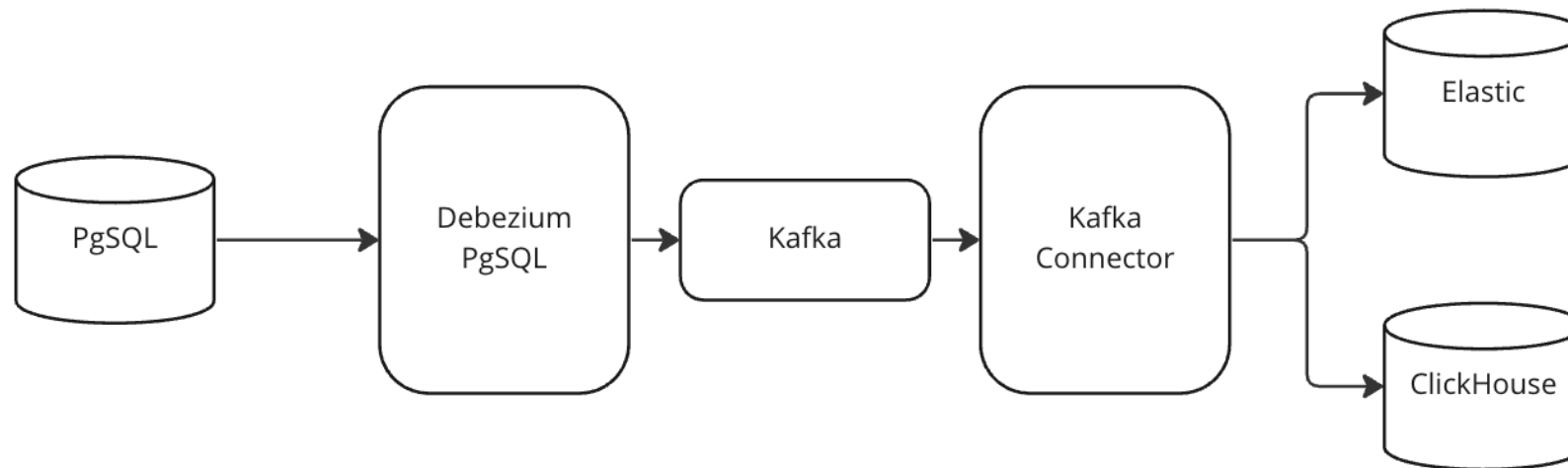
**Let's practise**

**Дополнительное**

## Kafka Cluster



# Capture Data Change (Debezium)



# libslave

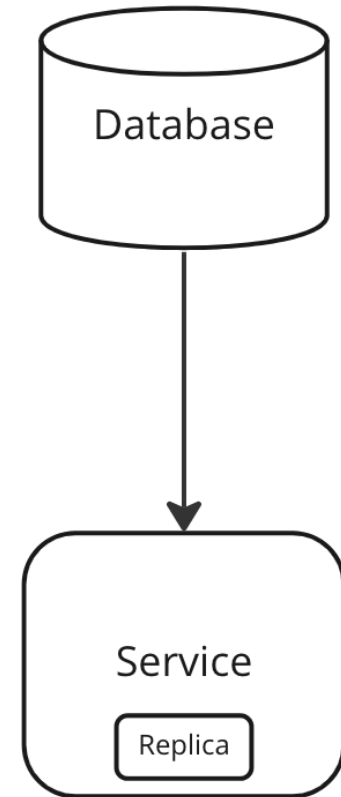
## ABOUT

---

This is a library that allows any arbitrary C++ application to connect to a Mysql replication master and read/parse the replication binary logs.

In effect, any application can now act like a Mysql replication slave, without having to compile or link with any Mysql server code.

One important use-case for this library is for receiving changes in the master database in real-time, without having to store the master's data on the client server.





# **FAQ:**

## **Дополнительное**

- Kafka cluster
- Debezium
- Libslave



# **Домашнее задание**