**System Design (Telegram)**

Функциональные требования – писание требуемого поведения системы в определенных условиях. Функциональные требования определяют, каким должно быть поведение продукта в тех или иных условиях. Они определяют, что разработчики должны создать, чтобы пользователи смогли выполнить свои задачи (пользовательские требования) в рамках бизнес-требований:

1. Сообщения только из текста
2. Авторизация не нужна
3. Поддерживает устройства: мобильные телефоны и web-браузеры
4. Чаты и личные сообщения будут
5. Удаления сообщений не будет
6. Редактирований не будет
7. Реакций пользователей не будет
8. Тегирование пользователей (перессылка на сообщения)
9. Кросс-девайсная синхронизация (телефон, компьютер)
10. Проверка прочитанности сообщений

Нефункциональные требования – описывают, как должен работать программный продукт и какими свойствами или характеристиками обладать, чтобы доставить ту ценность, которую несёт система, с учетом условий ее существования. Такие требования вносят вклад в инфраструктуру, а не в поведение системы:

1. Ограничение в 50.000.000 пользователей
2. Ограничение пользователей в сети 10.000.000 DAU(сколько пользователей сидит в оннлайн)
3. 500 символов макс длина сообщения
4. 1.5 секунды время на доставку на сообщения
5. Шифрование сообщений не будет
6. История не удаляется
7. Ограничение в 100 пользователей в чате
8. Регионы: только Россия
9. В среднем происходит 20 записей и 20 чтений операций на пользователя в день (RPS)

Чат от диалога отличается кол-вом человек.

На собеседовании:

1. Задать вопрос: сразу же начинать считать нагрузку, ли сначала написать api или проектировать схемы базы данных?
2. Нарисовать схему (разбить логику на микросервисы):
3. Выделить сущности (сообщения, пользователь, чат)

Выбрать базу данных исходя:

1. Какими БД команда умеет пользоваться, чтобы не нужно было тратить время на переучивание
2. Какой контент будет храниться(файлы, картинки)
3. Какая будет нагрузка
4. В реляционных БД проще писать запросы, так что они в приоритете
5. ACID БД (какой контент будет храниться, как они будут согласованы, нагрузка)

5.1) Atomicity(атомарность) — гарантирует, что каждая транзакция будет выполнена полностью или не будет выполнена совсем. Не допускаются промежуточные состояния.

5.2) Consistency(согласованность) — Транзакция, достигающая своего нормального завершения (EOT — end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты

5.3) Isolation(изолированность) — Во время выполнения транзакции параллельные транзакции не должны оказывать влияния на её результат.

5.4) Durability(надежность) — Во время выполнения транзакции параллельные транзакции не должны оказывать влияния на её результат.

1. Шардирование БД(вертикально или горизонтально разрезаем на разные куски для балансировки БД).

Варианты:

1. Админы переливают ночами вручную с одной БД в другую (VKGroup).
2. С помощью микросервиса смартшардирование, который переливает трафик с нагруженного в менее нагруженного БД(OZON).
3. Балансировка БД после шардирования
4. Репликация в случае, если одна БД отказала, варианты:
5. Синхронная
6. Асинхронная

Сущности в БД

Больше всего нагрузку будет составлять message, поэтому это главная сущность.

В кэш хранить записи не стоит, т.к. много памяти занимает.

User:

1. Id
2. Name
3. Escription

Channel:

1. Id
2. Name
3. Description

User\_per\_channel:

1. User\_id
2. Channel\_id

Message:

1. id
2. user\_id
3. channel\_id
4. timestamp
5. text
6. mentions array

last\_message

1. channel\_id
2. message\_id

last\_seen:

1. user\_id
2. channel\_id
3. message\_id

mention:

1. channel\_id
2. user\_id

База данных База данных База данных

User Chanel Message

WEB Client

Расчет нагрузки

RPS(read) = 10.000.000 \* 20(кол-во сообщений) = 200.000.000 (операций в день) / (24(часы)\*60(минуты)\*60(секунды)) = 2314 (операций в секунду)

RPS(write) = 10.000.000 \* 20= 200.000.000 / (24\*60\*60) = 2314

Data per day = 10.000.000 \*20(кол-во сообщений) \* 500(символов, 1 символ=байт) = 100 GB (вес информации в день)/ (24\*60\*60) = 1MB (в секунду).

1MB - это копейки для компьютера.

Data per year = 100 GB \* 365(кол-во дней в году) = 36 TB.

*Литература:*

System Design (Алекс Сюй) – простая книга

Высоко-нагруженные приложения (Мартин Клеппман) – читать, когда есть какой-то опыт

Микросервисы (Крис Ричардсон) – читать, когда есть опыт

Распределенные системы (Э. Тененбаум, М. Ван Стеен) – сложная книга

Видео:

Проектирование систем(Олег Бунин)

Распределенные системы(Олег Сухорослов)

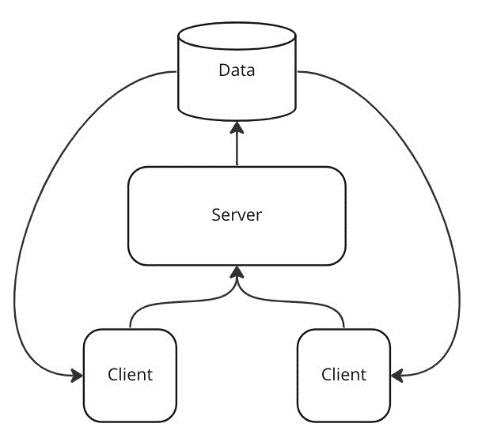
Ссылки:

<https://github.com/donnemartin/system-design-primer>

[https://github.com/G33kzD3n/Catalogue/blob/master/System Design Interview An Insider’s Guide by Alex Xu](https://vk.com/away.php?to=https%3A%2F%2Fgithub.com%2FG33kzD3n%2FCatalogue%2Fblob%2Fmaster%2FSystem%20Design%20Interview%20An%20Insider%E2%80%99s%20Guide%20by%20Alex%20Xu%20&utf=1)([z-lib.org](https://vk.com/away.php?to=http%3A%2F%2Fz-lib.org&utf=1)).pdf

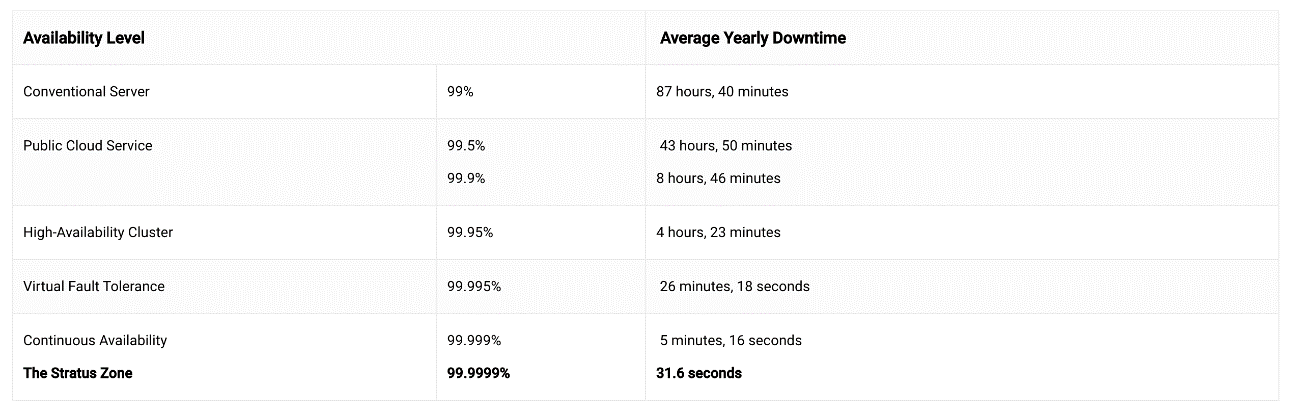
**Урок 1**

**Клиент-серверная архитектура**



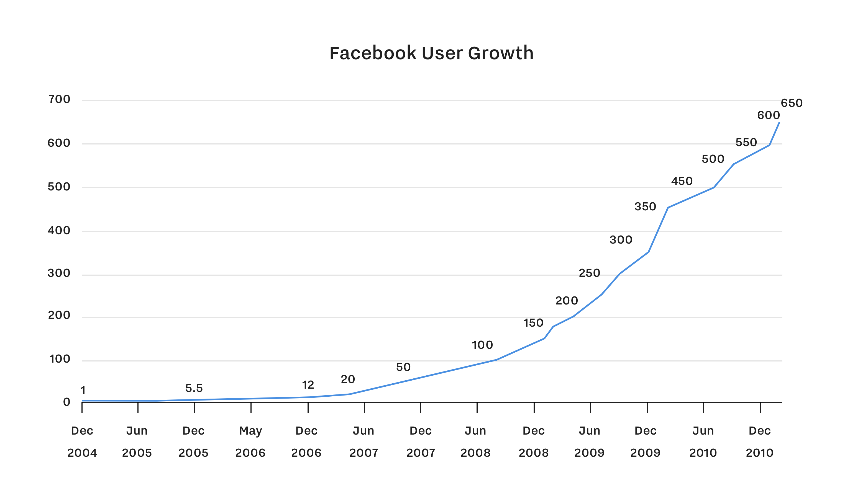
Критерии:

1. Надежность



* 1. Uptime – время работы
  2. Downtime – нерабочее время

1. Масштабируемость – система должна справляться с возросшей нагрузкой



2.1) Вертикальное – заменяем менее мощный сервер, более мощным сервером. Стоимость увеличения ресурсов не линейно, относительно их увеличения мощности (возможен downtime)

2.1.1) Большая стоимость

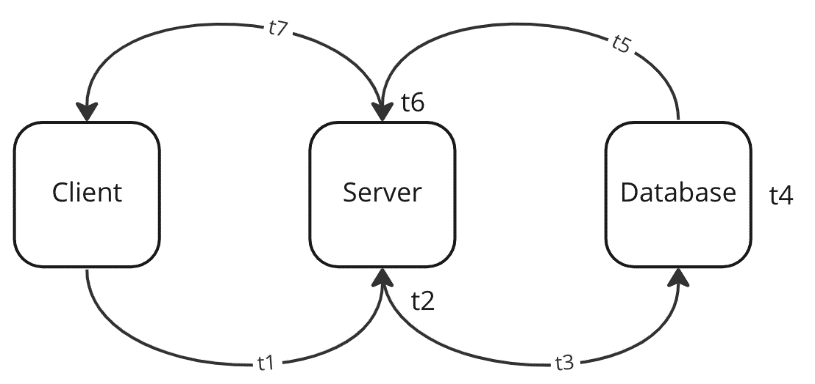
2.2) Горизонтальное – добавляем к серверу еще один сервер

2.2.1) Stateless – сервер, который не хранит никакого состояния внутри себя: нет кэшей, на жесткий диск ничего не сохраняет.

2.2.2) Statefull – база данных, которая хранит данные

Stateless в приоритете, чем Statefull.

1. Производительность

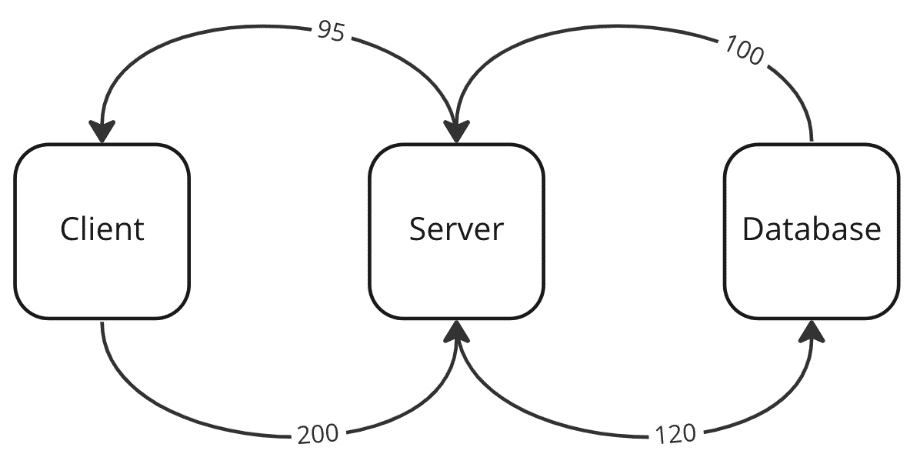


3.1) Response Time (время от отправки запроса клиентом до получения ответа) = t1(время до сервер) + t2(время обработки сервером) + t3(время до бд) + t4(время обработки бд) + t5 (время ответа бд) + t6 (время обработки сервером) + t7(ответ сервера)

3.2) Latency (время от отправки запроса клиентом до начала обработки сервером) = t1 + t2

3.3) Low-latency – минимальное время отклика сервером, запросы сразу обрабатываются сервером (Тинькофф инвестиции)

4) Пропуская способность – кол-во запросов во временном интервале (сек, мин). Например, 300 МБ/с



5) Удобство сопровождения – баланс между сложными решениями и дешевыми/быстрыми

5.1) Observability

5.2) Улучшение процессов – автоматический deploy (сборка, тесты)

5.3) Дополнительный инструментарий – дополнительные фреймворки, программы для быстродействия

6) Безопасность

1. Доступность
2. Нагрузка(rps)

SLA/SLO/SLI

SLA (service level agreement) – соглашение между поставщиком со своими клиентами: сколько будет ваш продукт работать(uptime) и сколько лежать(downtime). В случае невыполнения условий, поставщик должен будет выплатить компенсацию клиентам.

SLO (service level objectives) – цели, которые команда должна решить, чтобы выполнить соглашение, какой будет uptime и какой будет downtime

SLI (service level indicators) – показатели системы, метрика

Highload

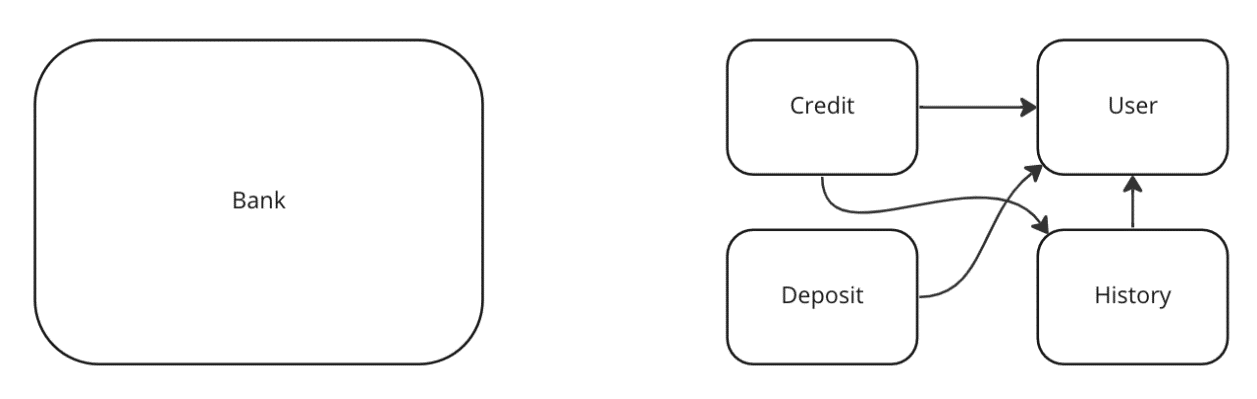
Data-intesive

1. Нужно сохранять большие данные
2. Нужно запоминать результаты ресурсоемких операций
3. Нужно предоставлять пользователям возможность искать или фильтровать данные

Compute-intensive

1. Нужно отправлять сообщения другим процессам
2. Нужно перемалывать большие объемы данных

Архитектуры: Монолит VS Микросервисы



1. Микросервисы – сервисы четко отделяются от друг друга
   1. Микро необязательно про размер, но про зону ответственности
   2. Самодостаточны, идеальны для горизонтального масштабирования
   3. Разные технологии для разных задач
   4. Распределенная кодовая база (разные языки программирования)

Плюсы:

* 1. Независимые релизы и разработка
  2. Независимая масштабируемость
  3. Независимая деградация
  4. Возможность пробовать новые технологии

Минусы:

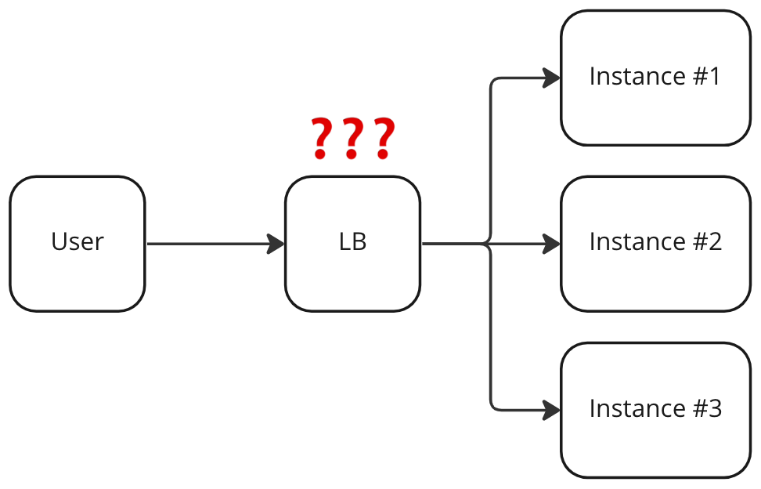
* 1. Зоопарк технологий
  2. Сетевой вызов отвалится вероятнее, чем внутрений
  3. Распределенность и тразакционность
  4. Удаленные вызовы дороже локального испольнения
  5. Понимание всего контекста запроса
  6. Сложность тестирования всей системы

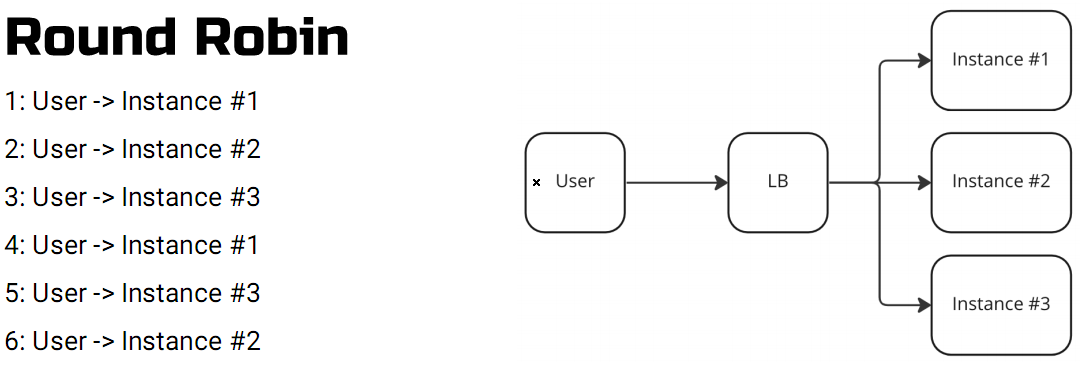
1. Монолит – большая системы

Мандат Безоса – для каждого микросервиса своя бд.

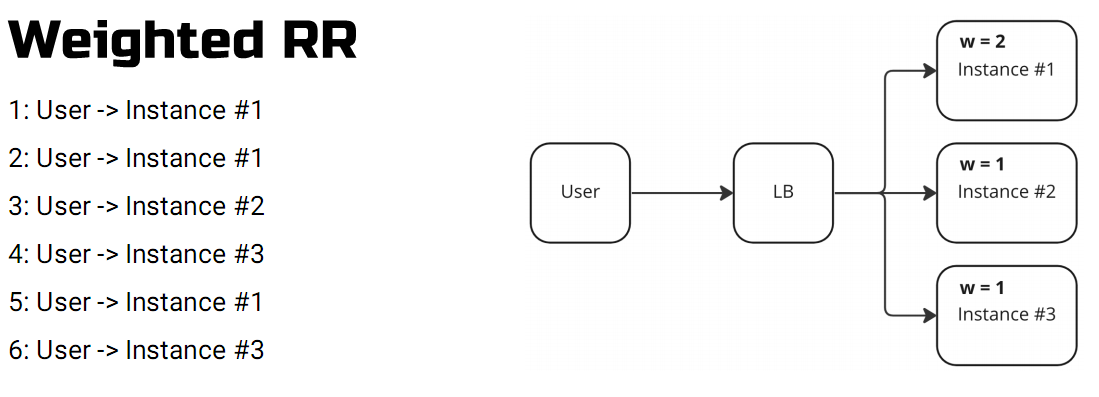
Балансировка нагрузки

Instance(микросервис)

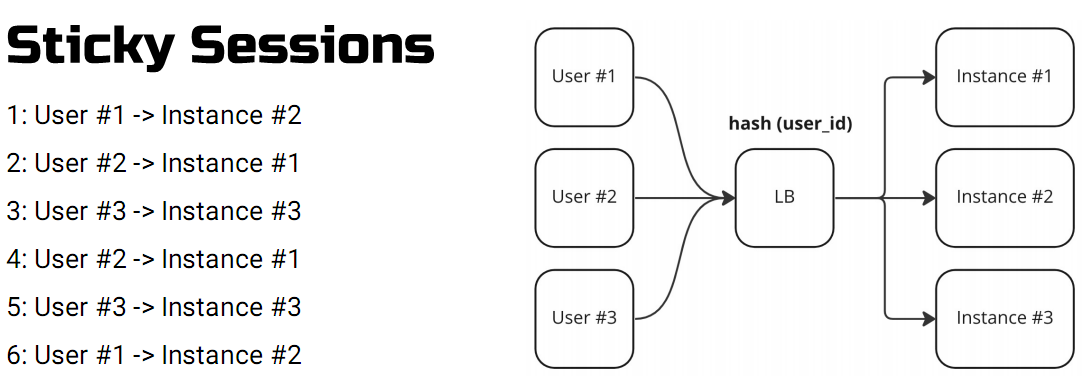




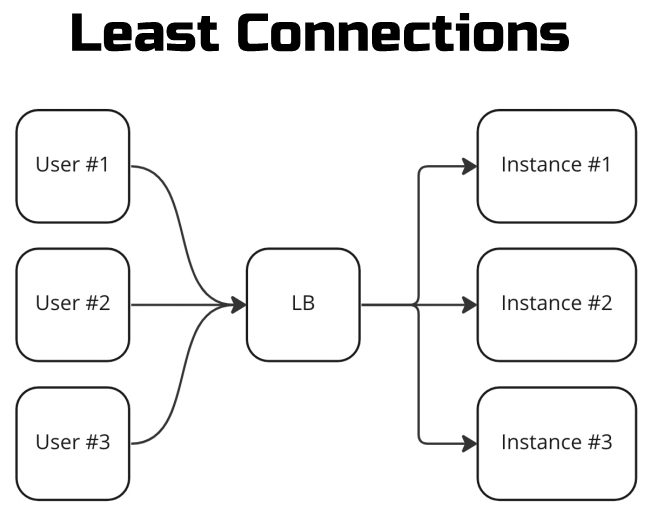
Round Robin - по кругу раскидываем всем запросы, неважна какая последовательность.



Weighted Round Robin - instances имеют разные мощности. Для instance добавляем вес.



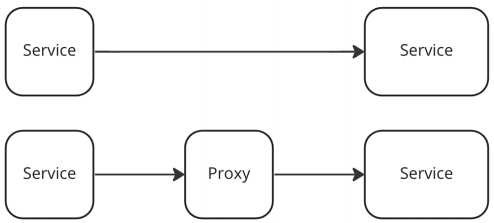
Sticky sessions – за каждым пользователем закрепляется свой instance. Нужна для этого map из hash, для поиска id пользователя в микросервисе.



Least connections – где меньше всего загруженность, в тот instance и пойдем.

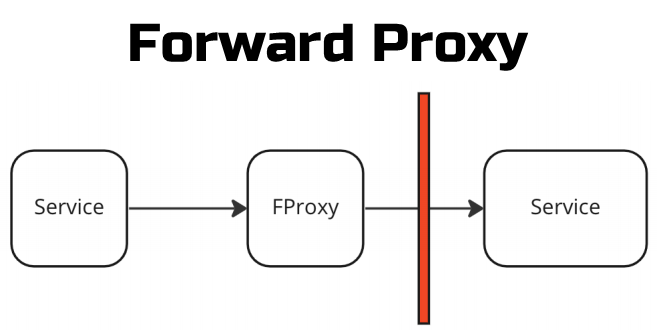
Nginx – популярный балансировщик (Weighted Round Robin). Он может выступать в качестве прокси.

Проксирование

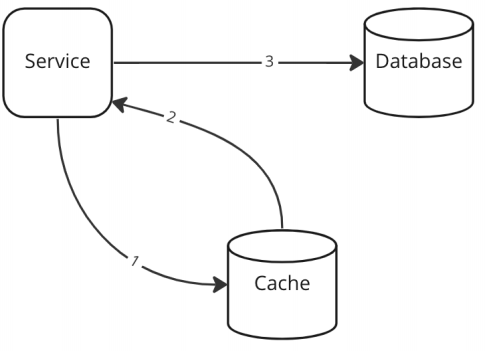
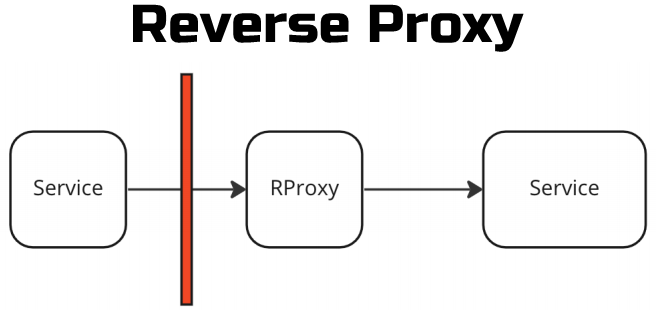


Проксирование - это посредник с помощью которого идет подключение

1. Взлом/защита
2. Кэширование данных
3. Ограничение трафика
4. Обход ограничений доступа
5. Анонимность пользователей
6. Сжатие и модификации данных



Forward proxy – клиент знает о прокси(vpn).

Reverse proxy – клиент не знает о прокси(google).

Кэширование

1. Сокращение response time сервисов
2. Снижение лишний нагрузки с поставщиком данных
3. Переиспользование ранее полученных и вычисленных данных
4. Стабилизация работы при кратковременных отказах систем – поставщиков данных

Термины:

1. Cache miss – промах кэша, запрошенный ключ не был найден в кэше
2. Cache hit – попадение в кэш, запрошенный ключ найден в кэше
3. Hit ratio – процент попаданий запросов в кэш, характеризует эффективность кэширования
4. Прогрев кэша – процесс наполнения кэша данными
5. Инвалидация – удаление кэшированных данных



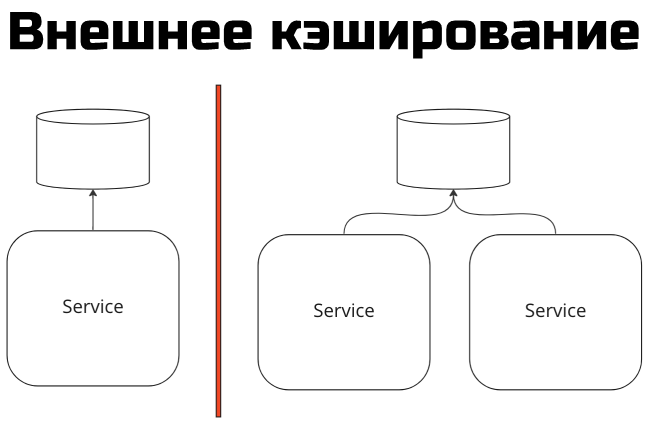
Внутреннее кэширование – хранение в памяти

Плюсы:

1. Высокая скорость
2. Отсутствие сетевых запросов
3. Нет расходов на Marshaling/Unmarshaling данных

Минусы:

1. Горизонтальное масштабирование
2. Прогрев кэша после падения сервиса



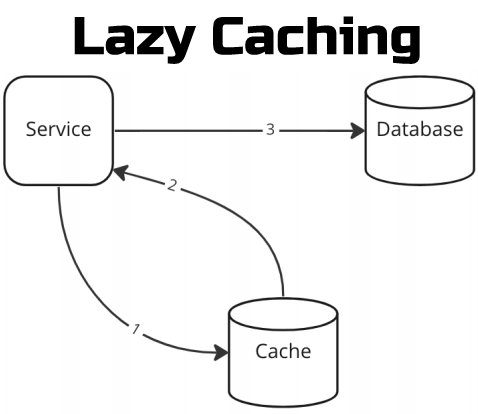
Внешнее кэширование – в качестве кэша есть бд.

Плюсы:

1. Хранение большого объема данных
2. Простое горизонтальное масштабирование
3. После падения сервиса данные кэша не теряются

Минусы:

1. Скорость работы



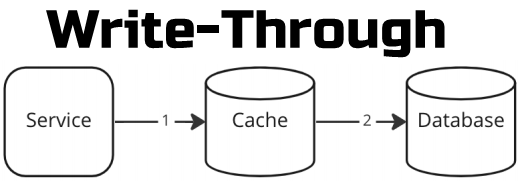
Lazy Caching – идем в кэш если элемент найден отдаем из кэша, если нет то получаем данные из бд и записываем данные в кэш

Плюсы:

1. Кэш содержит только запрашиваемые объекты
2. Новые объекты добавляются в кэш только по мере необходимости
3. Система устойчива к сбоям кэша

Минусы:

1. Каждый промах кэша требует совершить три операции(сервер->обновили данные->вернули пользователю)
2. Данные записываются напрямую в основное хранилище, что может привести к тому, что в кэше оказалось окажутся неактуальные данные



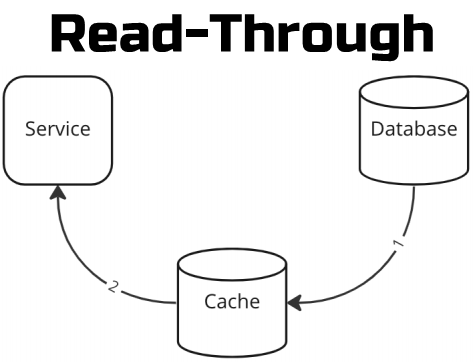
Write-through – данные всегда записываются в кэш, а потом сервис записывает в основное хранилище

Плюсы:

1. Кэш всегда актуален
2. Долгая запись при обновлении данных, но быстрое чтение

Минусы:

1. Кэш может быть заполнен ненужными объектами, к которым нет запросов, но потребляют лишняя память
2. Нужен дополнительный механизм позволяющий заново заполнять



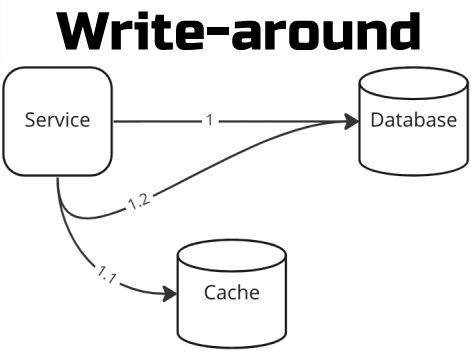
Read-through (сквозное чтение) – приложение всегда получает данные только из кэша, при этом логика наполнения кэша выносится в отдельный модуль.

Плюсы:

1. Логика приложения становится проще. Сервис ничего не знает о бд, она просто читает из кэша.

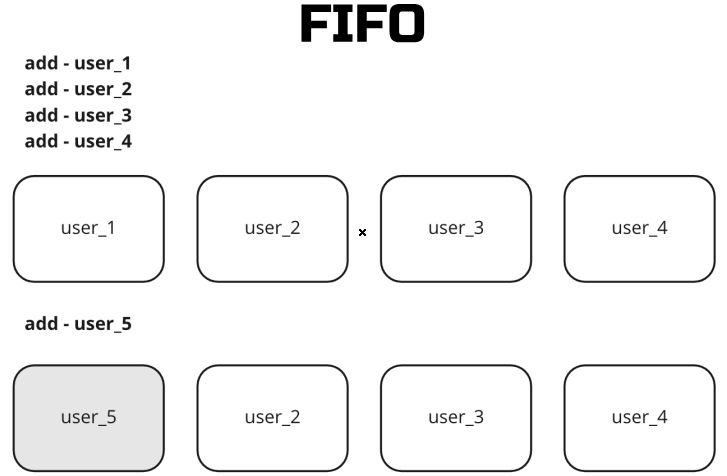
Минусы:

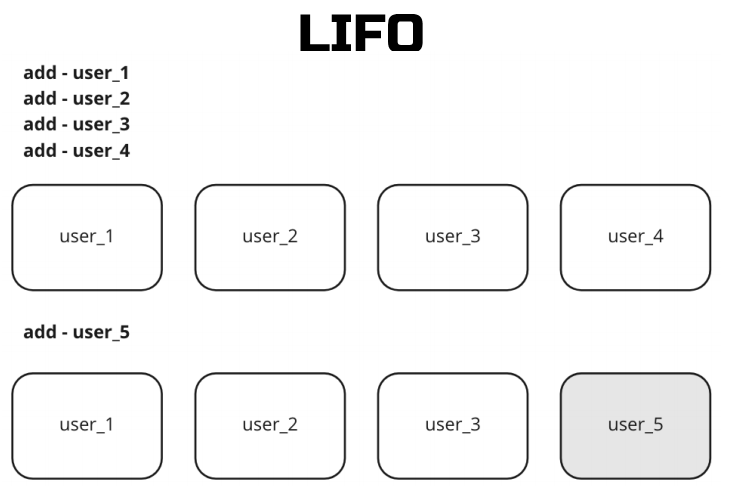
1. Система не устойчива к сбоям кэша
2. Сложность реализации

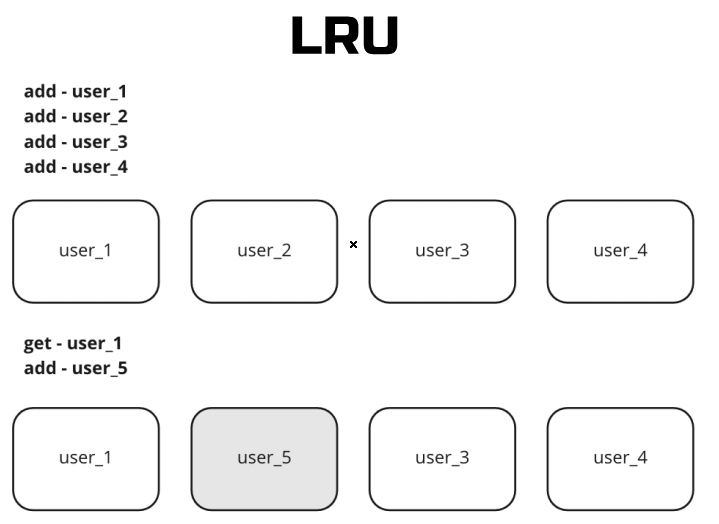


Write-around(запись в обход) – если элемент есть в кэше, то обновляем его в кэше и записываем в основное хранилище, если элемента нет, то сразу записываем в основное хранилище.

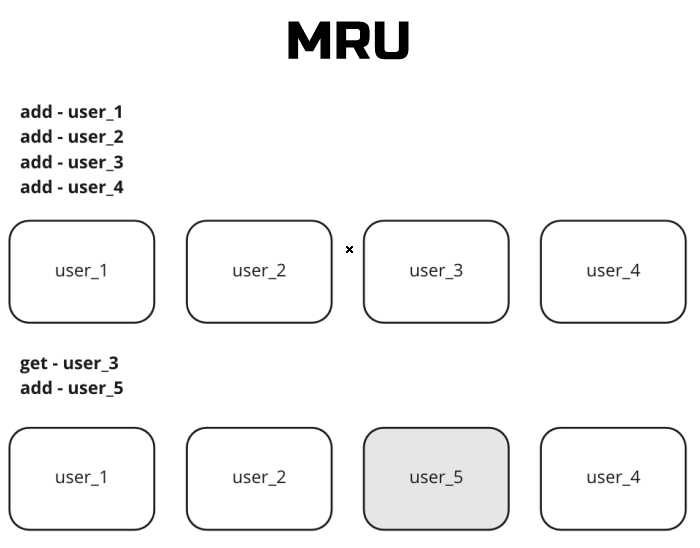
Алгоритмы замещения данных



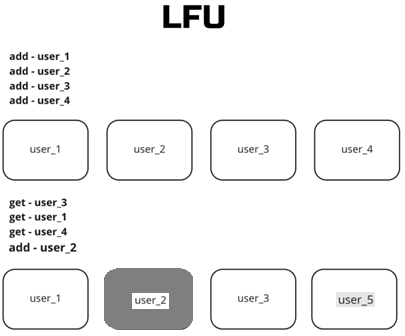




LRU (least recently used) – удаляем того, к кому было последнее обращение. После get -user\_1 было get\_user\_2. Если было обращение к user\_2, то удалится user\_1.



MRU (most recently used) – удаляем того, к кому только что было обращение.



LFU (least frequently used) – удаляем наименее используемый.

API

(Application Programming Interface)

CRUD – акроним, обозначающий четыре базовые функции: создание (create), чтение (read), модификация (update), удаление (delete).

Подходы:

1. REST (representational state transfer) – метод запроса (GET, POST, DELETE, PUT, PATCH). Применяется в HTTP.
   1. Существительное - URI (uniform resource identifier) запроса.
   2. Дополнительная информация – хедеры запроса.
   3. Содержимое – тело запроса и ответа в Json
   4. Результат – код ответа



1. SOAP (simple object access protocol) – устаревший подход, практически нигде не используется. Простой протокол доступа к объектам, обмен данными происходит в формате XML, который слишком многословен.



1. RPC (remote procedure call) – удаленный вызов процедур. Класс технологий, позволяющих программам вызывать функции или процедуры в другом адресном пространстве.

3.1) gRPC (google) – указывается структура сообщения, где есть типизация данных, есть четкий порядок. Пример, protobuf, где в качестве компилятора используется protoc. Protoc генерирует слева, справа появляется код, который может использоваться в программах.



Сначала идет номер (string name = 1), потом идет тип (string), потом идет длина строки.



Получается компактно и используется меньше трафика.

Термины:

1. Маршалинг(сериализация) - процесс преобразования информации (данных, двоичного представления объекта), хранящейся в оперативной памяти, в формат, пригодный для хранения или передачи.
2. Over fetching – перенасыщение информации, когда получаем от сервера не нужную информацию. Например, нам нужны только посты; а приходят посты, лайки и комментарии.
3. Under fetching – Недополучение данных. Например, нам нужны посты, лайки, и комментарии; а приходят только посты.
4. Решение проблемы under/over fetching – graphQL, мутаторы, подписки

graphQL



Мутаторы



Подписки



Выводы:

1. gRPC лучше использовать во внутреннем взаимодействии между сервисами.
2. REST лучше использовать во внешнем взаимодействии между клиентом и сервисами.
3. GrapghQL лучше для кастомизации запросов по REST.

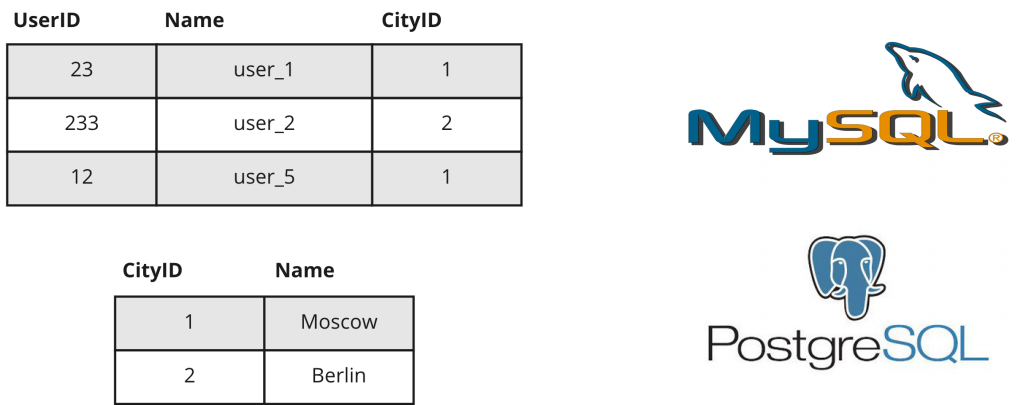
Базы данных

OLAP (online analytical processing) – большие данные храним, но редко читаем/записываем.

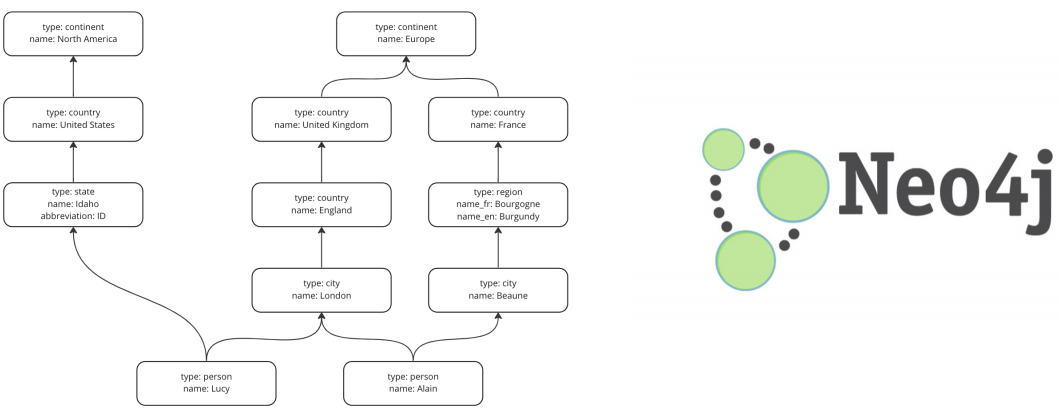
OLTP (online transaction processing) – большие данные храним, но быстро читаем/записываем.

Виды:

1. Реляционные – основаны на отношениях, связей. Хорошо работают на чтение. Есть привязка к типам данных. Поиск происходит по строчкам.



1. Документоориентированные (нереляционные) – нету связей (OLTP). Хорошо работают на запись. Есть привязка к типам данных. Например, анкета пользователей, резюме. 
2. Графовые – множественные связи (OLTP). Например, дороги, карты, соцсети.



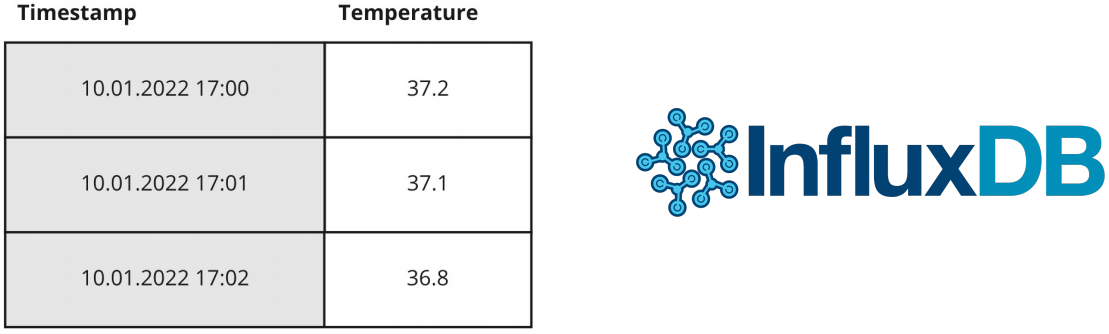
1. Key-value – map. Например, счетчик просмотра видео на youtube.



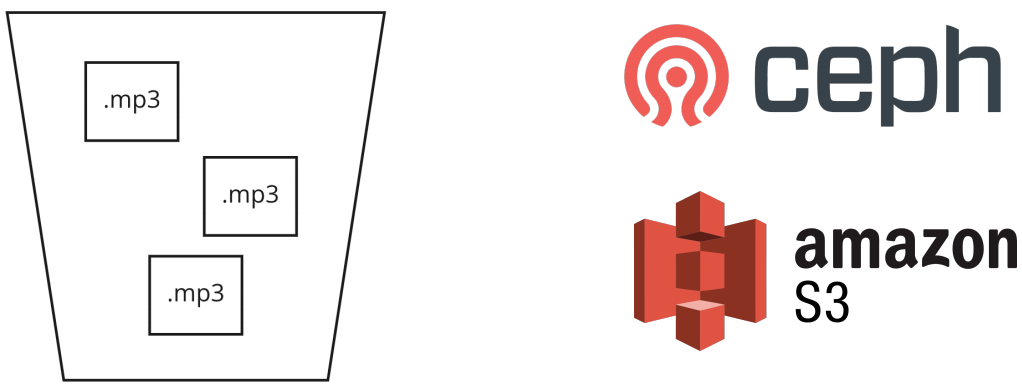
1. Колоночные – для аналитики (OLAP). Поиск происходит по столбцам.



1. Time series – привязка к временным событиям.



1. Blob store – хранение медиафайлы (музыка, картинки, видео, файлы). Используются бакеты. Поиск по id файла.



1. Embedded database – само приложение хранит бд. Лучше так не делать.
2. In-memory бд

9.1) можно периодически записывать на диск копии состояния бд

9.2) можно записывать на диск журналы изменений (что-то вроде WAL), но не сами данные

9.3) можно проводить репликацию состояния оперативной памяти на другие машины

Выбор базы данных

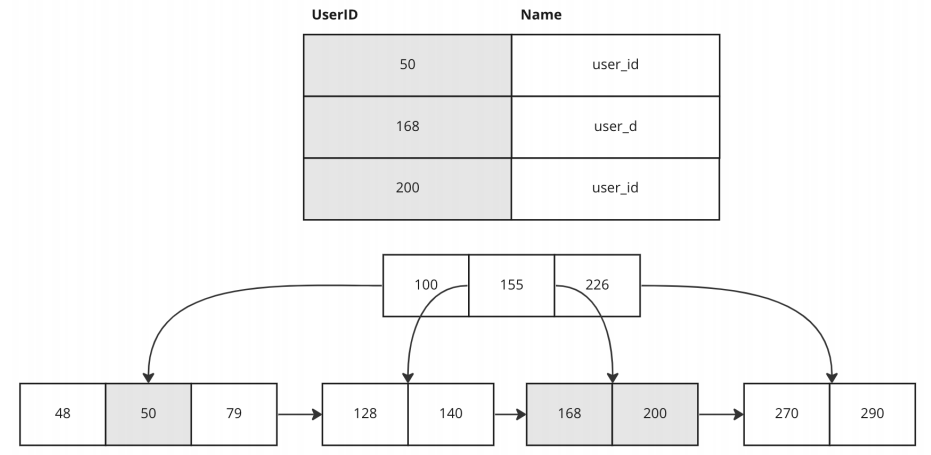
1. Транзакции (связи) – acid (реляционные), base (нереляционнные)
2. Формат данных
3. Навык работы с технологией
4. Характер обращений к данным
5. Сообщество и зрелость технологии
6. Частота изменяемости формата данных

Data retention – хранение данных определяет политики управления постоянными данными и записями для соблюдения юридических и бизнес-требований к архивированию данных. Например, через месяц какую-то часть данных нужно удалить.

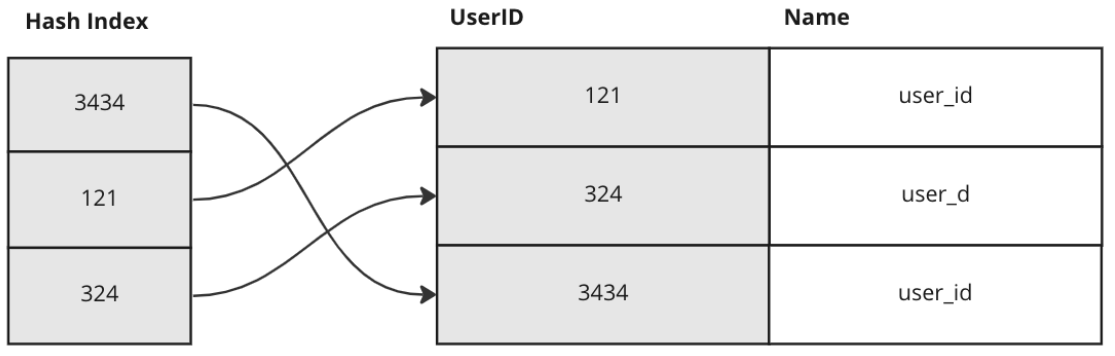
Индексы

Типы:

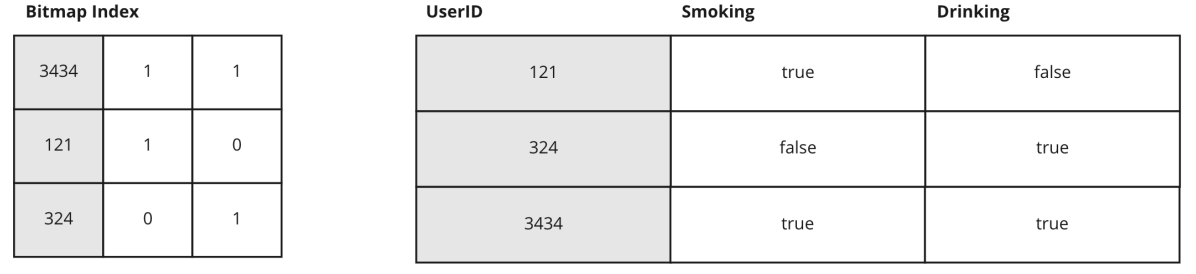
1. BTree – сбалансированное дерево поиска, считывает данные последовательно. Уровень ветвистости доходит до 200.



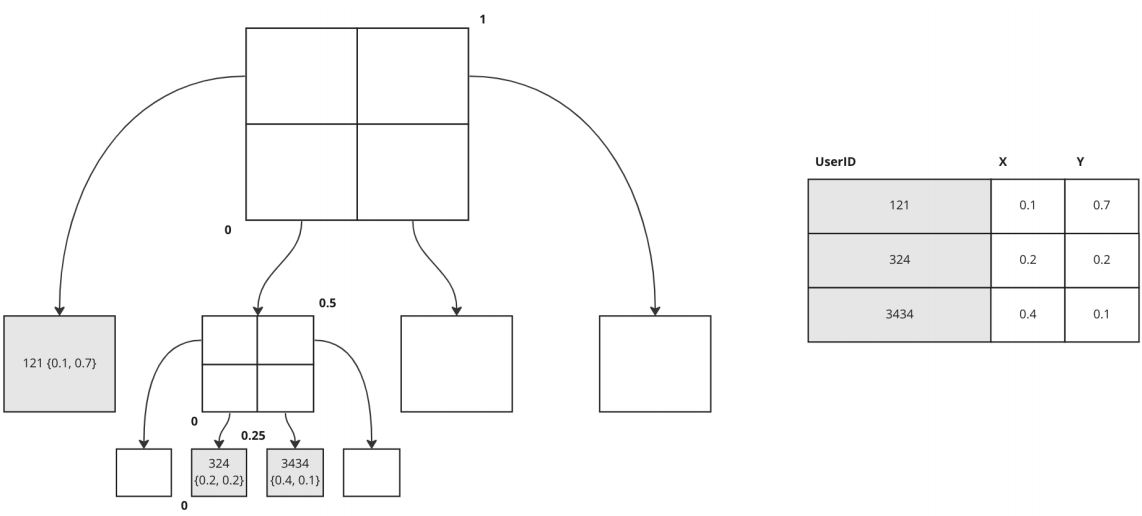
1. Hash



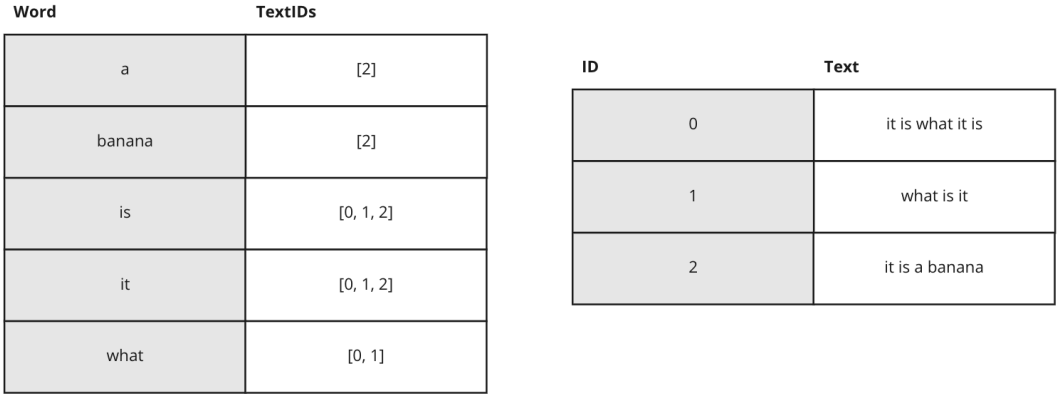
1. Bitmap – tinder (курит да/нет, женат да/нет)



1. Spatial – пространственные индексы. Есть система координат по x и y, разрезаем на 4 части и ищем по каждому квадратику.



1. Reversed – кол-во раз сколько то или иное слово встречается/храниться.



Плюсы:

1. Ускоряют чтение
2. Замедляют запись
3. Используют дополнительную память

Кластерные и некластерные

Кластерный индекс – только один индекс, благодаря которому таблица перестраивается и упорядочивается. При вставке элемента, таблица перестраивается. Например, primary key.

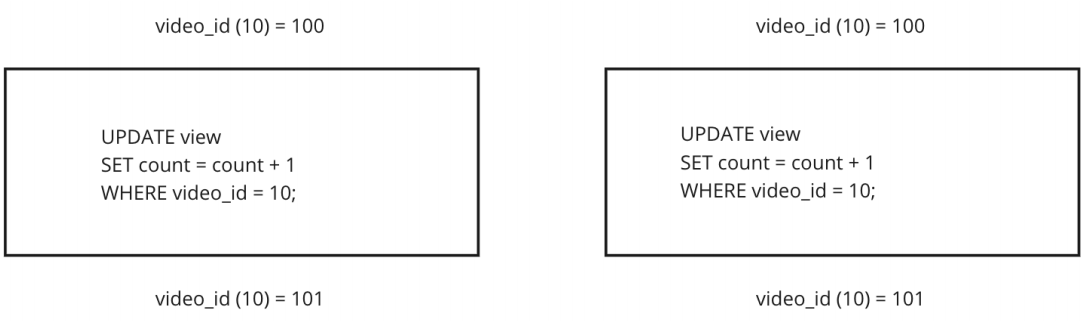
Некластерный индекс – может быть множество индексов, никак не влияет на внутреннее устройство таблицы.

Покрывающий индекс – покрывающие индексы называются некластеризованные индексы, которые разрешают один или несколько схожих результатов запроса напрямую, без доступа к базовой таблице и без уточняющих запросов.

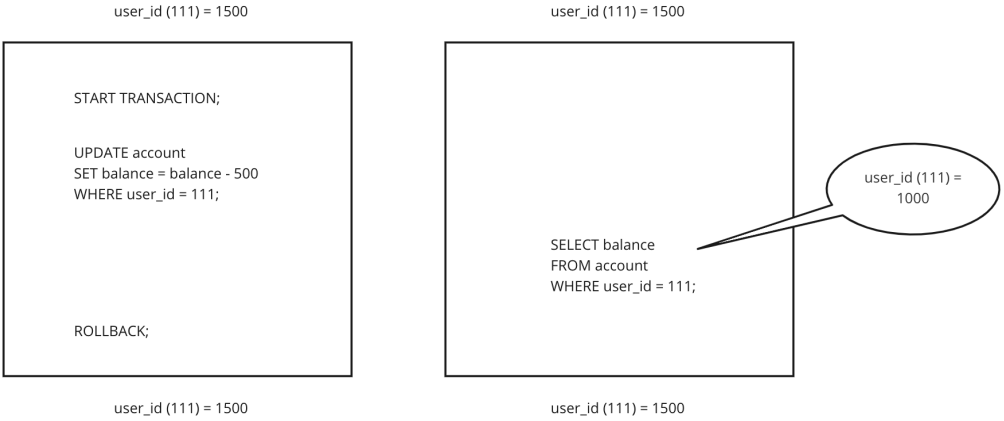
При наличии кластерного индекса строки таблицы упорядочены по значению ключа этого индекса. Если в таблице нет кластерного индекса, таблица называется кучей. Некластерный индекс, созданный для такой таблицы, содержит только указатели по записи таблицы.

Транзакции

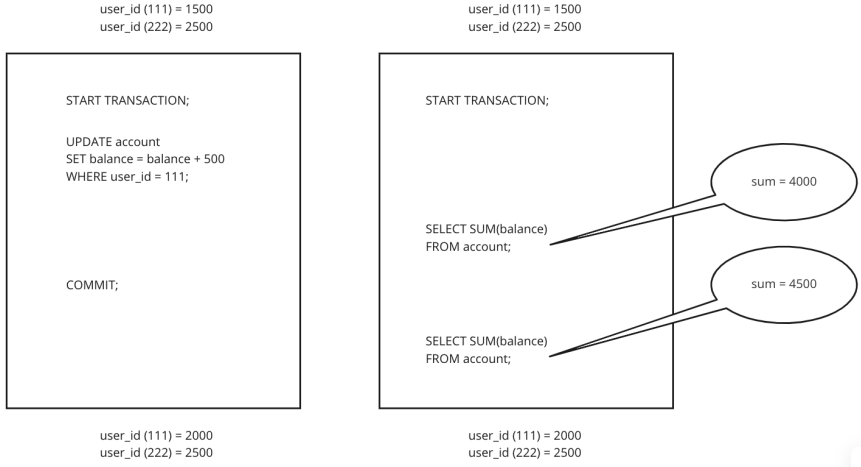
1. ACID (atomicity, consistency, isolation, durability)) – это стандарт того, какие гарантии должна давать бд, чтобы поддерживать транзакции. Каждая бд может по-своему этот стандарт реализовать.
   1. Атомарность – каждая транзакция бд является единым блоком, который использует подход «все или ничего» к выполнению. Если какой-либо оператора в транзакции терпит неудачу, вся транзакция откатывается. Если транзакция прошла успешно, то делаем commit.
   2. Изоляция транзакций – каждая транзакция происходит до и после каждой другой транзакции, и представление бд, которое транзакция видит в своем начале, изменяется только самой транзакцией до ее завершения. Ни одна транзакция не должна видеть промежуточный результат.
   3. Потерянное обновление – 2 одинаковые транзакции приходят одновременно в бд.



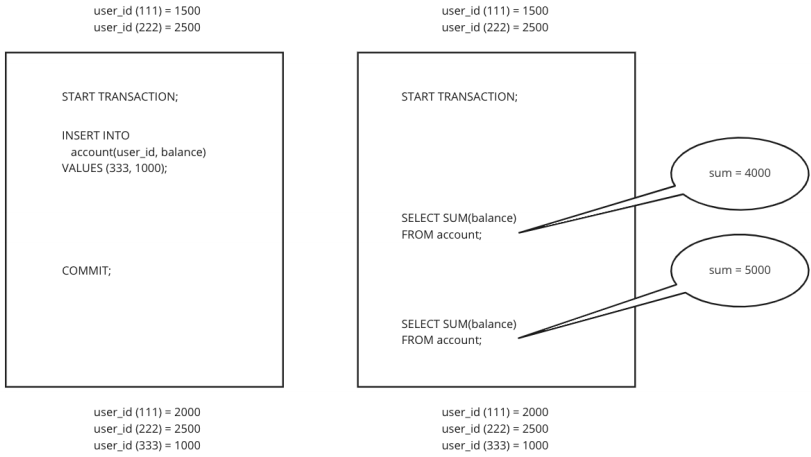
* 1. Грязное чтение – чтение данных транзакции из не commit транзакции.



* 1. Неповторяющееся чтение – во время чтения появляются новые данные после update после commit транзакции.



* 1. Чтение фантомов - во время чтения появляются новые данные после insert/delete после commit транзакции.



Уровни изоляции

READ\_UNCOMMITTED – могут происходить грязные чтения, неповторяющиеся чтения, фантомные чтения и потерянное обновление

READ\_COMMITTED – грязные чтения предотвращены, но могут возникать неповторяющиеся чтения, фантомные чтения и потерянные обновления

REPEATABLE\_READ – грязные чтения, неповторяющиеся чтения и потерянное обновление предотвращены, но могут возникать фантомные чтения

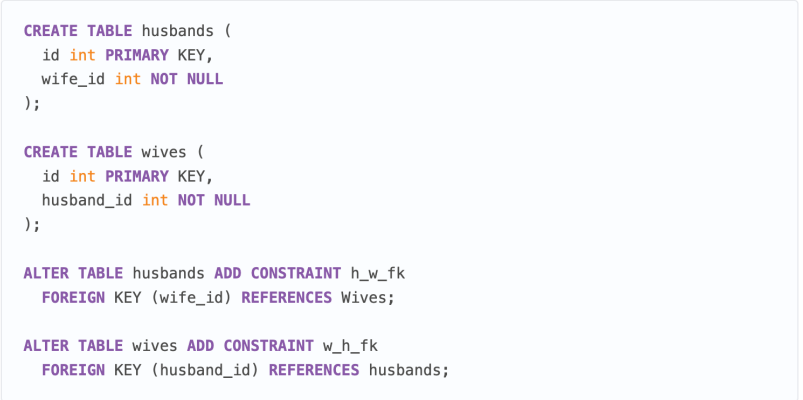
SERIALIZABLE – транзакции полностью изолированы – исключено влияние одной транзакции на другую в момент выполнения

У каждой бд свои уровни транзакции. При создании транзакции выбираем для нее уровень.

* 1. Согласованность – различные утверждения относительно данных (инварианты) должны всегда оставаться справедливыми



* 1. Отложенные транзакции – есть зависимые таблицы и без учета foreign key, вставка может не выполниться.



* 1. Устойчивость – гарантирует, что после фиксации транзакции в бд она постоянно сохраняется с помощью резервных копий и журналов транзакций. В случае сбоя эти механизмы могут использоваться для восстановления зафиксированных транзакций. Например, мы вставили новый элемент в кластер, таблица перестраивается и падает.
  2. WAL (write-ahead logging) – изменения в файлах с данными должны записываться только после того, как эти изменения были внесены в журнал. Записывать страницы данных на диск после подтверждения каждой транзакции нет необходимости, потому что мы знаем, что если случится сбой, то у нас будет возможность восстановить бд с помощью журнала.

1. BASE – как правило, NoSQL бд предоставляют ограниченные версии атомарности и изоляции, но им нужно продавать себя в красивой упаковке, поэтому они придумали свою аббревиатуру.

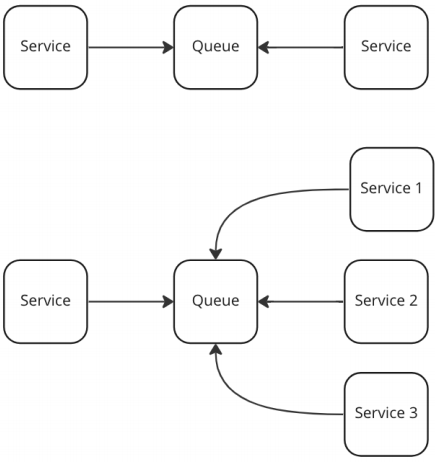
2.1) Basically Available – бд будут обеспечивать доступность данных, распространяя и реплицируя их по узлам кластера бд. В большинстве случаях в бд не будет WAL.

2.2) Soft State – из-за отсутствия строгой согласованности значения данные могут меняться со временем. Модель BASE разрывается с концепцией бд, которая обеспечивает собственную согласованность, делегируя эту ответственность разработчикам. Согласованность должны гарантировать разработчики бд.

2.3) Eventual Consistency – в конце концов увидите действительные данные, но есть вероятность, что ваша транзакция прочитает недействительные значения, т.е. временные, или частично обновленные, или устаревшие. Возможно, данные станут согласованными в дальнейшем.

Брокеры сообщений

Посредник для доставки сообщений. Например, адаптер.



Термины:

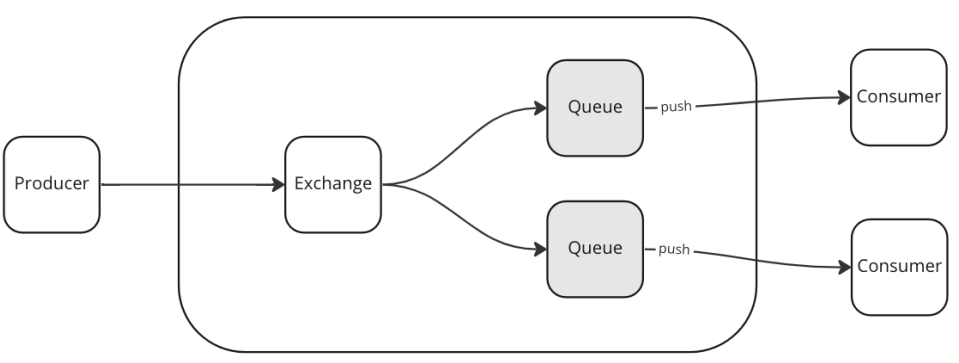
Consumer – сервис, который читает из очереди

Producer – сервис, который пишет в очередь

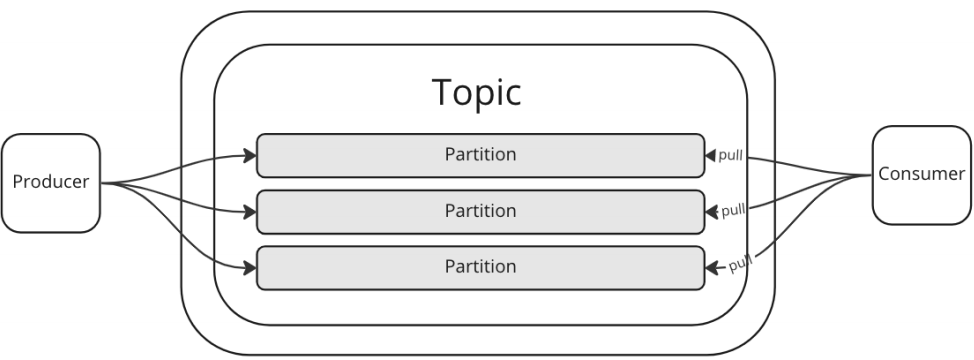
1. Буферизация – накопить несколько данных и потом обновить, а не по одному
2. Асинхронная связь – сервис отправил сообщение в очередь и забыл и не нужно дожидаться ответа
3. Слабое связывание – когда service не знает о service1, service2, service3, а они о нем.
4. Масштабируемость – если запись fail, то добавляем больше producer, если чтение fail, то добавляем больше consumer
5. Отказоустойчивость – если сервис упал, то он поднимется и перезапуститься
6. Понимание потоков данных – исходя из очереди можно понять кто в какой сервис идет



RabitMQ – есть роутер и очереди. Исходя из политики, роутер отправляет данные по очередям (сразу, по кругу). Работает по push модели – у него есть consumerы, которые подписаны на обновление очереди, когда пришло сообщение в очередь, RabitMQ сам отправит сообщение consumerу, когда придет подтверждение, что сообщение прочитано, RabitMQ удалит это сообщение из очереди.

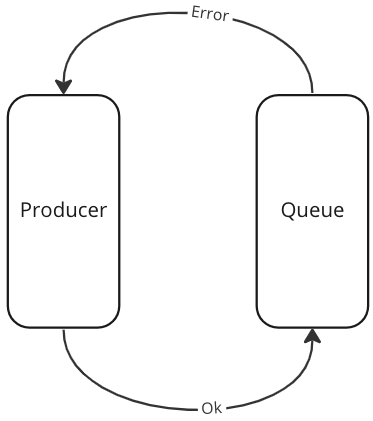


Kafka – есть топики, consumerы и prosuerы. Ключевое отличие от RabitMQ – когда приходит сообщение в очередь, то не Kafka будет отправлять сообщение в очередь, а consumer будет ходить за этим сообщением с каким-то интервалом. Когда прочитали сообщение из Kafka, Kafka это сообщение из очереди не удаляет, а хранит, чтобы еще какой-нибудь consumer может тоже это сообщение прочитать. Топики – это таблицы, они разделяются на partition. За каждым partition закреплен только consumer. Kafka удаляет исходя из настроек (через неделю, месяц). В Topic должен быть хотя бы один partition. Кол-во partition зависит от задач.

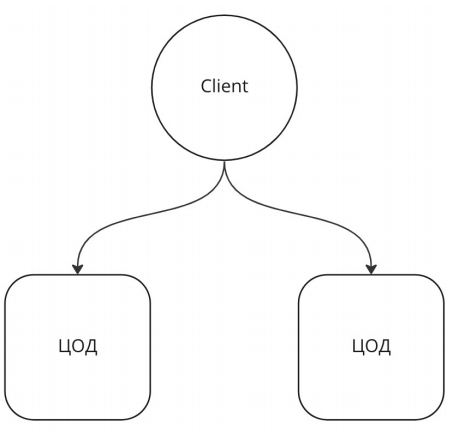


Гарантии доставки

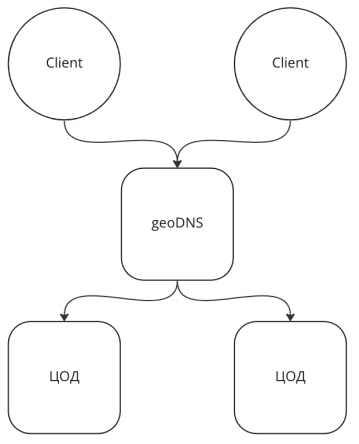
1. At least once – сообщение будет доставлено хотя бы один раз
2. At most once – сообщение будет доставлено не более одного раза
3. Exactly once – сообщение будет доставлено ровно один раз



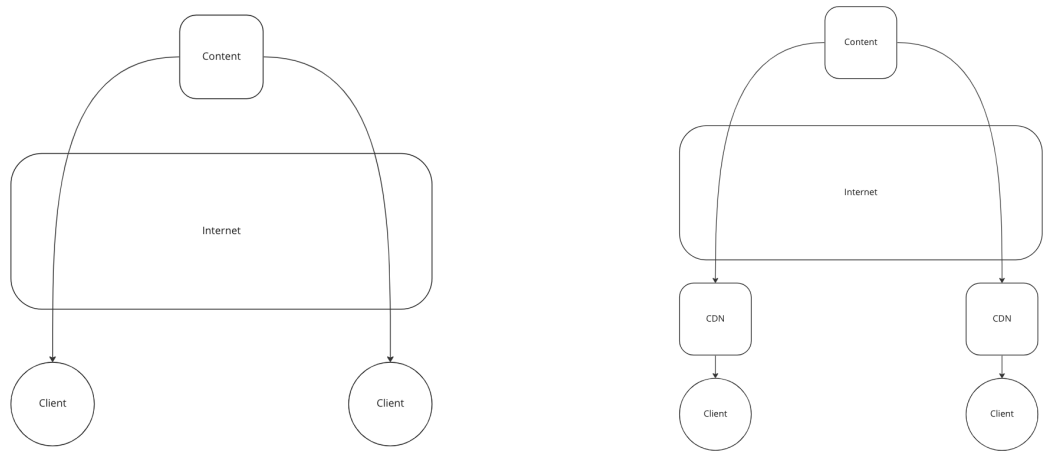
ЦОД - центр обработки данных



geoDNS (geographic domain name system) – можно привязать к одному доменному имени несколько ip-адресов. В зависимости от географического положения (определяется по ip-адресу, с которого пришел запрос) пользователь перенаправляется на ближайший сервер



CDN (content delivery network) – данные записываются в кэши и при повторном запросе данных, данные будут отдаваться с кэшей.



1. Исходный (origin) – на котором размещен запрашиваемый сайт, а также связанный с ним визуальный, музыкальный, видеоконтент
2. PoP (point of presence) – точка присутствия вспомогательных серверов. Их сеть размещается в различных регионах
3. Proxy-сервер – это промежуточное звено между пользователем и исходным сервером. Он отвечает за перенаправление, оптимизацию и преобразование передаваемого трафика

Observability

1. Мониторинг – нужно понимать, что с нагрузками, что с показателями системы. Обычно используется Prometheus, для определенных метрик. Grafana графически отображает метрики.

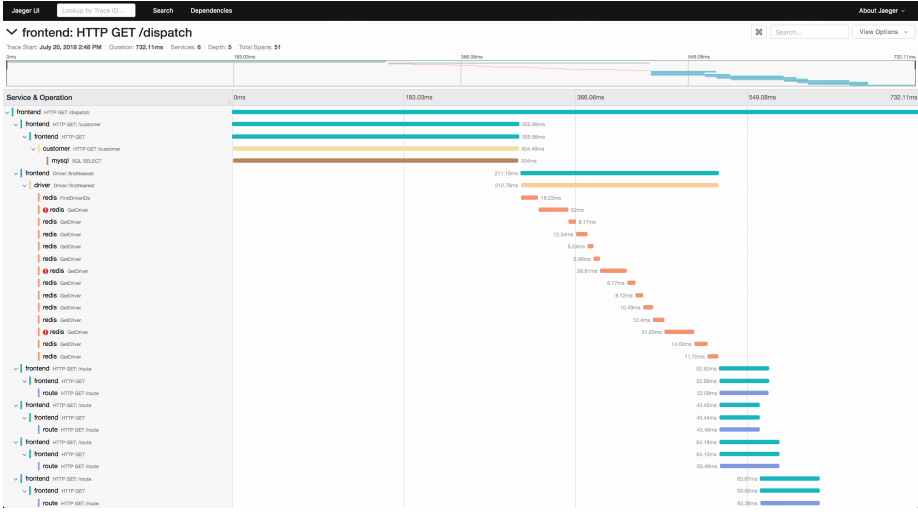
Основные метрики:

1. RPS (кол-во запросов в секунду), TRS (кол-во транзакций в секунду), QRS (кол-во запросов в секунду)
2. Response time
3. Errors rate (сколько ошибок приходит)
4. Traffic, CPU, RAM, HDD/SSD
5. Uptime/Downtime
6. Размеры очередей
7. Кол-во процессов/потоков

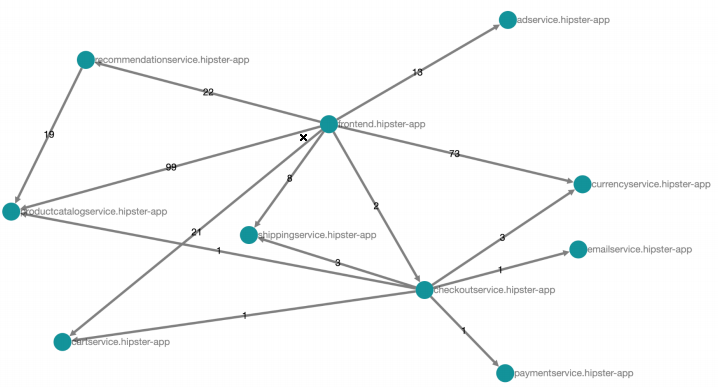
Трейсинг

Помогает понять, как произошла ошибка

1. Jaeger –



Исходя из трафика можно построить графы, как микросервисы ходят, с каким кол-во запросов

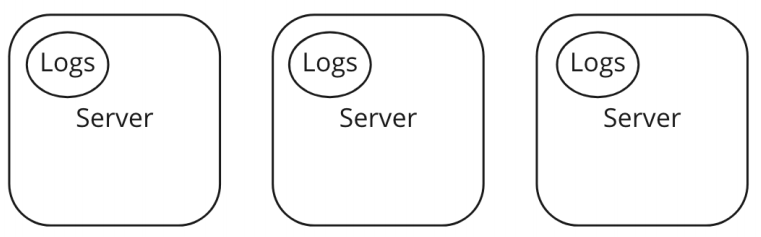


1. Zipkin -

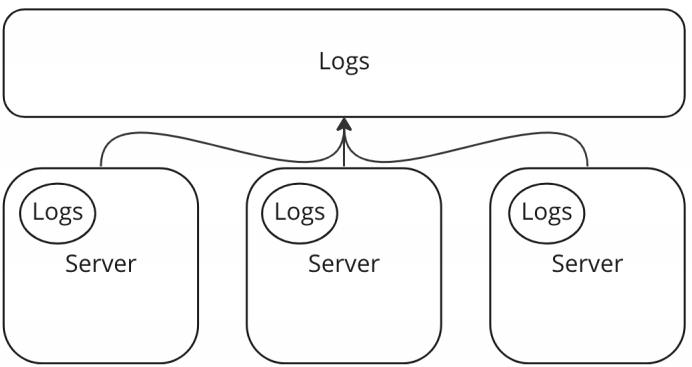
Логирование

Способы:

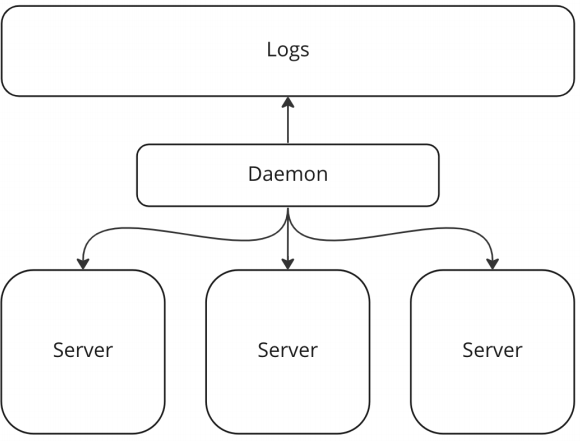
1. Логи можно хранить на каждом сервере и искать. Это плохо и медленно



1. Логирование – отдельная машина логов, через какой-то интервал приходят логи с каждого сервера (Тинькофф, Mail).



1. Daemon – система, которая сама забирает логи с каждого сервера и отправляет на отдельную машину логов





Непрерывное профилирование

Например, приложение упало ночью, с помощью pyroscope смотрит нагрузку системы в определенном промежутке времени. Можно смотреть разницу производительности debug и release.



Sentry – отправляет dump перед падением.

