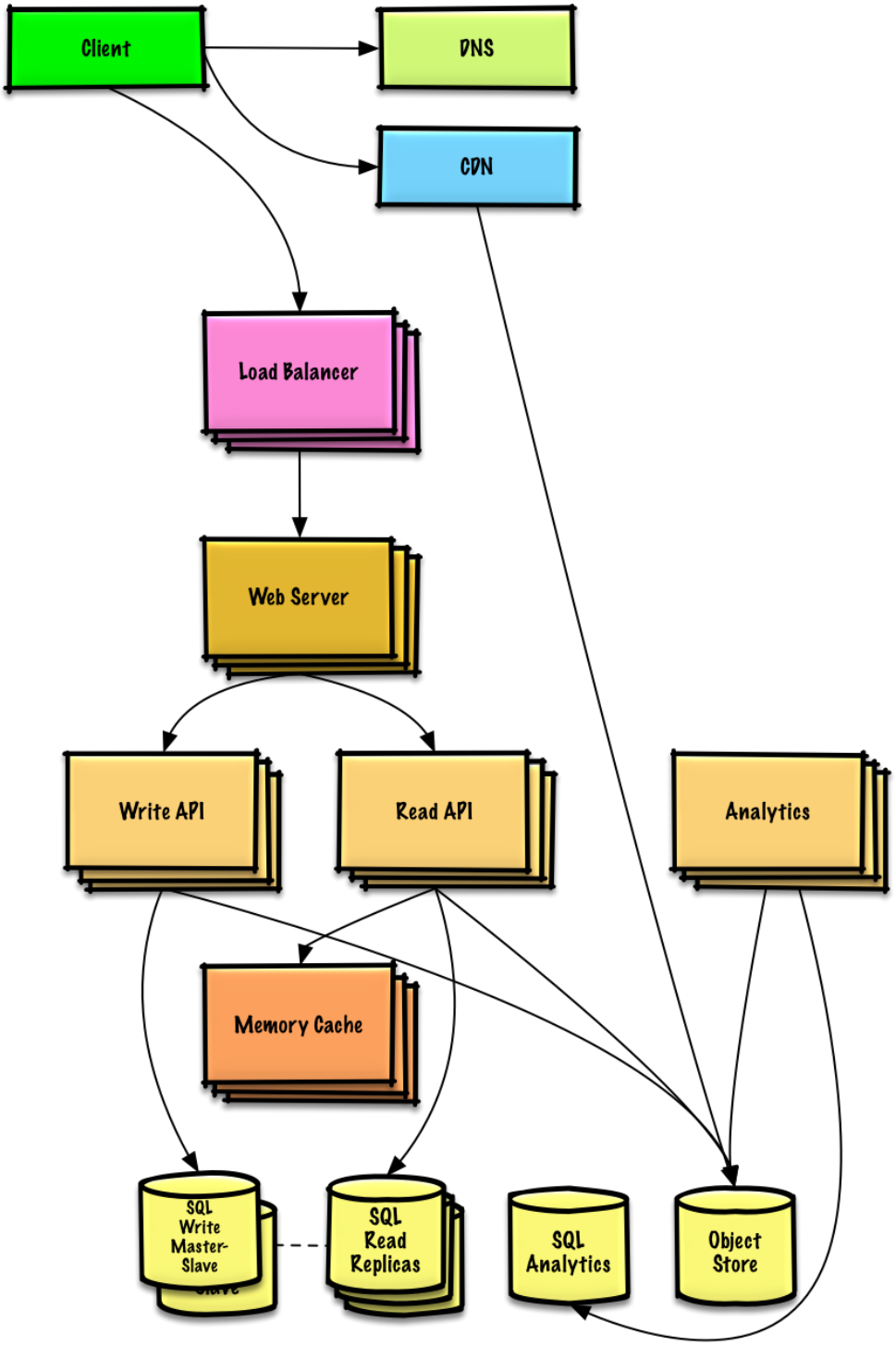
Букварь по дизайну систем (Ольховой Д.Р.)



Словарь

Node – нода - узел с каким-либо ресурсом
Content – контент - данные
Traffic – трафик - запрос/ответ, данные, которые передаются от сервера клиенту и наоборот
Hardware – железо - аппаратная часть
Instance – инстанс - созданный объект какой-либо сущности. Например *инстанс сервера API*
Headers – хедеры - заголовки (как правило TCP пакета, но может быть и HTTP запроса)

Производительность (Performance) vs масштабируемости (Scalability)

Сервис является масштабируемым, если его производительность увеличивается пропорционально добавляемым ресурсам. В общей сложности, увеличение производительности увеличивает объем данных, которые может обработать сервис.

- Если у вас есть проблемы с производительностью, то ваша система является медленной для одного пользователя.
- Если у вас есть проблемы с масштабируемостью, то ваша система работает производительно для одного пользователя, но имеет проблемы при многопользовательском доступе.

Дополнительный материал

- A word on scalability
- Scalability, availability, stability, patterns

Задержка (Latency) vs Пропускная способность (Throughput)

Латентность – это время, необходимое для выполнения какого-либо действия или для получения какого-либо результата.

Пропускная способность – это количество операций или результатов в единицу времени.

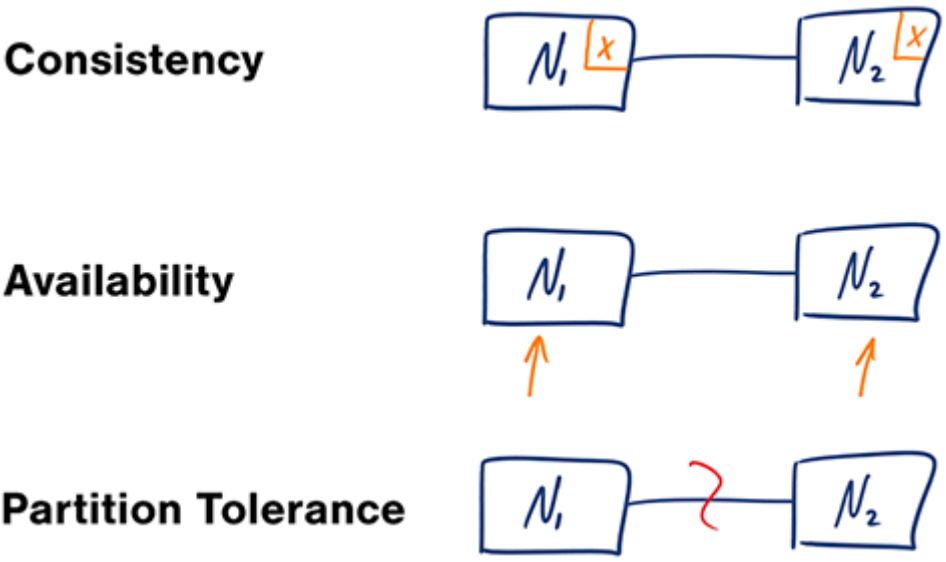
В общем случае следует стремиться достигнуть максимальной пропускной способности при допустимой латентности.

Дополнительный материал

- Understanding latency vs throughput

Доступность (Availability) vs (Консистентность) Consistency

CAP теорема



В распределённых системах вы сможете гарантировать только две вещи из трёх:

- Консистентность - Каждое чтение возвращает самые свежие данные или ошибку.
- Доступность - Каждый запрос возвращает результат. Без гарантий, что результат содержит последние данные.
- Устойчивость к разделению - Система должна функционировать несмотря на потери ресурсов, узлов(node) или каких либо других ошибок коммуникации между сервисами.

Сеть не является надежной, соответственно, вам необходимо поддерживать устойчивость к разделению. Следует делать компромиссы между консистентностью данных и доступностью данных.

CP - (consistency and partition tolerance) - Консистентность + устойчивость к разделению

В такой системе в результате ожидания ответа от ноды можно получить timeout (истечение времени ожидания). Такой подход отлично подходит для систем, в которых

требуется соблюдение атомарности операций. Например, транзакций в банке. Нельзя купить что-то, не успев пополнить счет.

AP - (availability and partition tolerance) - Доступность + устойчивость к разделению

Ответы возвращают наиболее свежие данные, доступные ноде, но они могут быть не самыми свежими в целом. Запись данных в распределенных системах занимает некоторое время (необходимо сделать множество копий между несколькими базами данных), поэтому данные могут появиться не сразу.

Этот подход подходит системам, бизнес-логика которых допускает частичную консистенцию. Также этот подход хорош для систем, которые должны продолжать функционировать вне зависимости от внешних ошибок.

Дополнительный материал

- CAP theorem revisited
- A plain english introduction to CAP theorem
- CAP FAQ
- eventual consistency

Паттерны консистенции

Поддерживая множество копий тех же данных, мы сталкиваемся с проблемой их синхронизации. Клиент всегда должен получать только самые свежие данные. Вспоминая CAP теорему - каждое чтение должно возвращать самые свежие данные или ошибку.

Слабая консистенция

После записи, читатель может увидеть записанное, а может и не увидеть. Здесь применяется метод best effort (A best effort approach is taken)

Данный подход можно наблюдать в таких системах как **memcached**. Слабая консистенция отлично подходит для использования в real-time, например VoIP, видео чатах и мультиплеерных играх. Например, если вы разговариваете по телефону и потеряли связь на некоторое время, то вы не услышите то, что было сказано в этот момент времени.

Частичная консистенция

После записи, чтение вполне себе вернет последнии данные, но, возможно не сразу (обычно в течение нескольких миллисекунд). Данные реплицируются асинхронно.

Этот подход часто используется в системах DNS и Email. Частичная консистенция отлично подходит для системы с высокой доступностью.

Сильная консистенция

Сразу после записи, будет возможно считать самые последнии данные. Но запись, разумеется происходит синхронно и это может влиять на доступность системы. Данные реплицируются по всей системе синхронно.

Данный подход используется в таких системах как RDBMSes. Сильная консистенция отлично подходит для систем, в которых требуются транзакции.

Дополнительный материал

- Transactions across data centers

Паттерны доступности

Существует два главных паттерна для поддержки высокой доступности: отказоустойчивость и реплицирование.

Отказоустойчивость (Fail-over)

Активный-пассивный (Active-passive)

С таким подходом происходит отправка пульса (heartbeats) между активным и пассивным серверами. Если пульс прекращается, то пассивный сервер берет на себя обязанности активного сервера переключая IP адреса на себя. Время простоя (downtime) зависит от того, как быстро пассивный сервер готов начать работу в качестве активного. Пассивный сервер может ожидать в горячем (hot) или холодном (cold) состоянии.

Пассивный сервер может убить активный сервер путем использования команды - shoot the other node in the head (STONITH), это метафорическая команда и может быть реализовано как угодно. Делается это для того, чтобы в системе не функционировало одновременно два активных сервера, так как это может привести к ошибкам.

Активный-пассивный отказоустойчивый подход может так же называться как master-slave - мастер-раб.

Активный-активный

С активный-активный подходом, оба сервера обрабатывают трафик, распределяя нагрузку между собой.

Если доступ к серверу выставлен наружу, то DNS должен знать IP адреса обоих серверов. Если сервера выставлены для внутреннего использования, то логика на уровне приложения должна знать об обоих серверах.

Активный-активный, может так же называться как - master-master - мастер-мастер.

Недостатки

- Отказоустойчивость добавляет сложности и тербует большего количества железа.

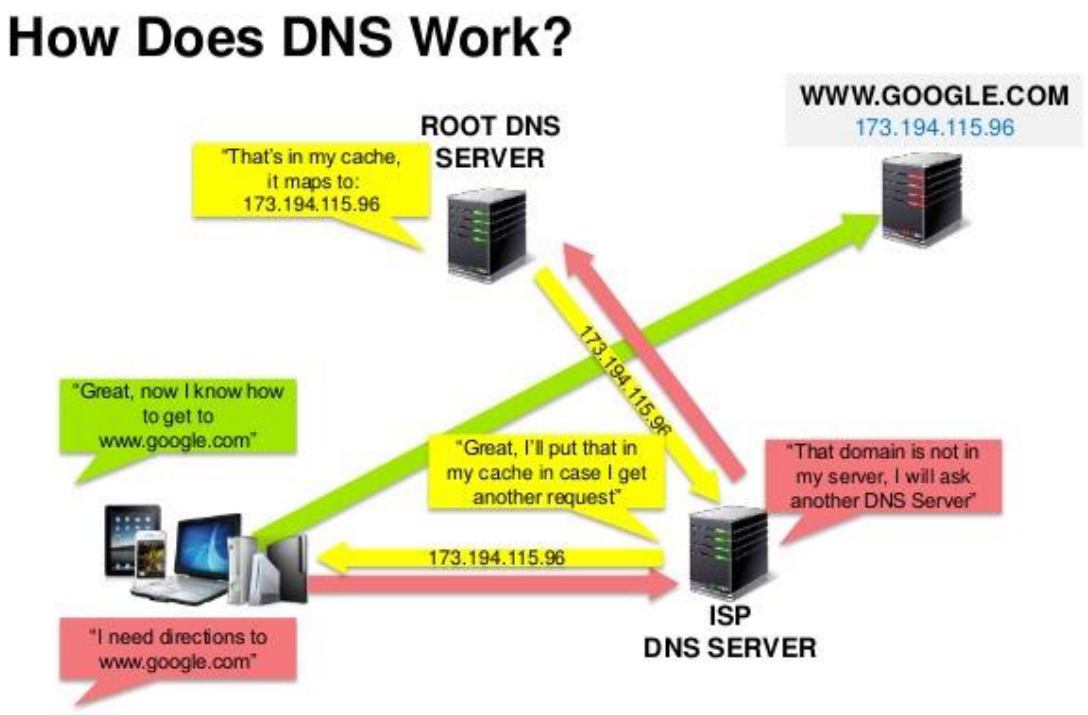
- Существует потенциальная возможность потери данных, если данные не успели реплицироваться с активного на пассивный сервер.

Репликация

Master-slave и master-master

Мы рассмотрим эти два подхода в разделе про базы данных.

Система доменных имен (Domain name system)



DNS - преобразует доменное имя в IP адрес. Например www.example.com в 127.0.0.1.

DNS иерархичен и имеет несколько высокоуровневых серверов. Ваш роутер или провайдер предоставляет информацию о том, какой из DNS серверов использовать. Низкоуровневые DNS серверы, кэшируют маппинг, который может устаревать из-за задержек обновлений. Так же DNS может быть закеширован браузером или операционной системой.

- NS-запись (name server) - определяет DNS сервер для вашего домена/субдомена.
- MX-запись (mail exchange) - определяет почтовый сервис для принятия почты.
- A-запись (address) - определяет IP адрес в виде доменного имени.
- CNAME-запись (canonical) - определяет другое имя (example.com как www.example.com) или A-запись.

Такие сервисы как CloudFlare и Route 53 предоставляют управляемые DNS сервисы. Некоторые DNS сервисы могут направлять трафик, используя разные методы:

- Взвешенное распределение трафика
    - Предотвращает попадание трафика на сервера, которые не способны принять запрос
    - Балансирует трафик между серверами
    - Позволяет проводить A/B тестирование
- Распределение по задержке
- Геораспределение

Недостатки DNS

- Запрос на DNS сервер добавляет некоторую задержку, но она вполне себе устраняется методами кеширования, описанными выше.
- DNS может быть сложен в настройке и управлении, но большинство DNS, управляются государствами и крупными компаниями.
- DNS серверы подвергаются атакам и такие атаки могут нести всемирный характер.

Дополнительный материал

- DNS architecture
- Wikipedia
- DNS articles

Сеть доставки контента (Content delivery network)

Content delivery network (CDN) - это глобальная и распределенная сеть прокси серверов, предоставляющая контент пользователям в различных локациях по всему миру. Как правило, CDN хранит статичные файлы такие как HTML, CSS, JS, но есть сервисы которые хранят и динамичный контент.

Использование CDN улучшает производительность двумя способами:

- Пользователи получают контент с ближайшего к ним CDN сервера
- Сервер с приложением не тратит усилия на обработку таких запросов

Push CDN

Push CDNs получают новый контент каждый раз когда оно обновляется на сервере. В данном случае, пользователь CDN (разработчик) отвечает за обновление CDN серверов актуальным контентом. Можно настроить время по которому истекает актуальность контента или обновлять его только тогда когда появится новый.

Push CDN отлично подходит для сайтов с маленьким трафиком или для сайтов контент которых меняется не часто. Лучше один раз загрузить контент на CDN, чем выкачивать его с сервера на постоянной основе.

Pull CDN

Pull CDN выкачивает новые данные при первом обращении пользователя. Такой первый запрос как правило медленней последующих, так как требует получения данных с сервера. Следующие запросы уже будут закешированы CDN-ном и буду происходить быстрей.

TTL - time-to-live, определяет как долго контент будет храниться в кеше. Pull CDN позволяет уменьшить объем данных, который хранится на CDN, но тем самым может вызвать увеличение ненужного трафика на сервер, если данные на CDN устаревают раньше, чем они действительно становятся не актуальными на сервере.

Pull CDN отлично подходит сайтам с большим трафиком, так как количество обращений к закешированным данным гораздо больше.

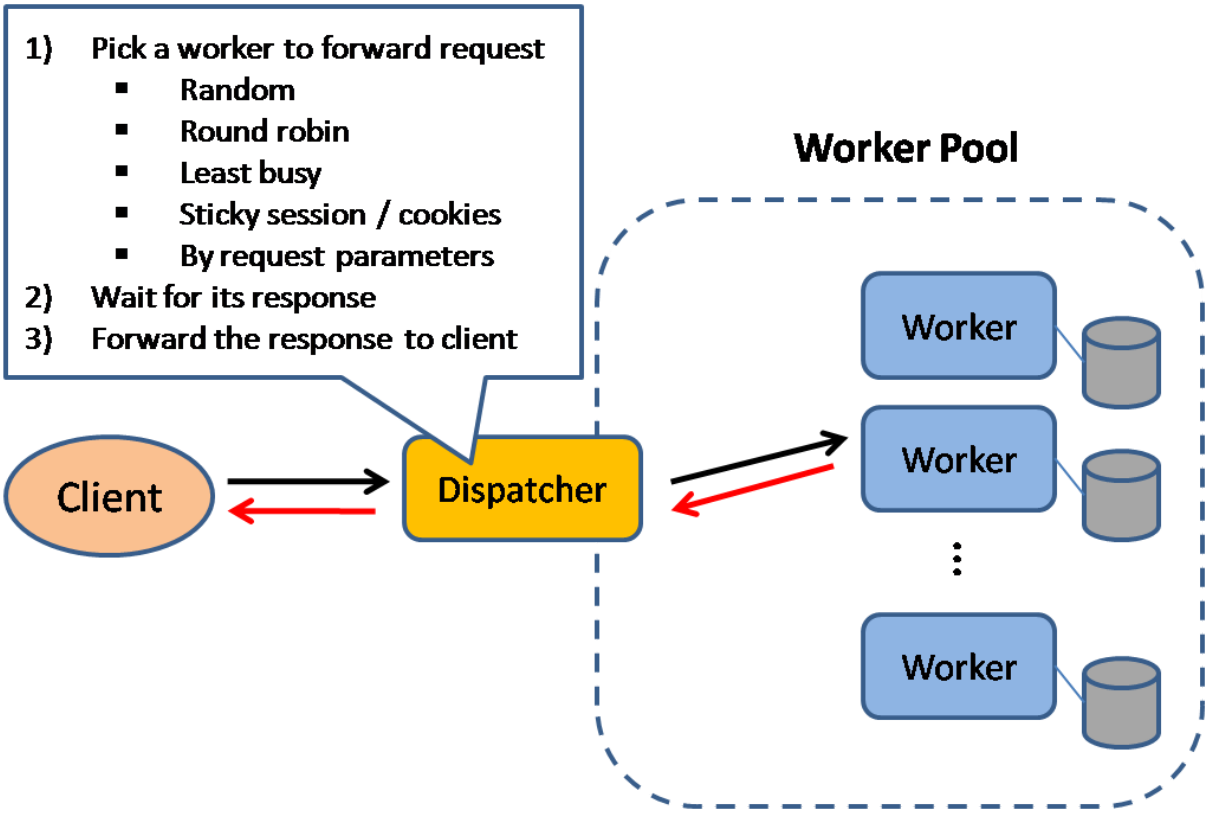
Недостатки: CDN.

- Контент на CDN может устаревать, если он обновляется раньше удаления кеша.
- Необходимость изменение ссылок на новый контент.

Дополнительный материал:

- Globally distributed content delivery
- The differences between push and pull CDNs
- Wikipedia

Балансировщик нагрузки (Load balancer)



Балансировщики нагрузки распределяют входящий трафик между сервисами (приложениями, базами данных), каждый раз возвращая ответ клиенту, который делал изначальный запрос.

Балансиры эффективны:

- Когда нужно предотвратить обработку запроса не работающим сервером;
- Когда нужно защитить ресурсы от чрезмерной нагрузки;
- Когда нужно избежать единой точки отказа;

Балансиры могут быть реализованы с помощью железа (дорого) или программно например с HAProxy.

Дополнительные плюсы использования:

- SSL терминация - расшифровка входящих запросов и шифровка ответов.
    - Сервера не будут тратить свои ресурсы.
    - Нет необходимости ставить сертификаты X.509 на каждый сервер
- Хранение сессий - выдает куки пользователями и пытается направлять одного и тоже юзера на тот же сервер. Убирает необходимость хранить

сессию на серверах, что в принципе и не должно происходить учитывая горизонтальное масштабирование, где инстансы одного и того же приложения запущены на множестве серверов.

Для защиты от ошибок, зачастую лучше иметь несколько инстансов балансеров в активный-активный или активный-пассивный режимах.

Балансиры могут направлять трафик опираясь на следующие метрики:

- Случайно
- Наименее загруженный
- Сессии/куки
- Round robin or weighted round robin
- По слою 4 модели OSI
- По слою 7 модели OSI

Layer 4 балансировка

Данная балансировка работает на транспортном уровне, определяя как распределить запрос. Обычно используется IP адрес и порт источника и получателя в заголовке пакетов, но не в их содержимом. Балансиры направляют трафик с помощью Network Address Translation (NAT)

Layer 7 балансировка

Данная балансировка работает на уровне приложения, для распределения трафика может быть задействованы и данные из хедеров, контента и кук. Балансир делает выбор на какой сервер отправить запрос и открывает соединение с выбранным сервером.

Балансер 4-го уровня имеет меньшую гибкость, но более высокую скорость, так как распределение происходит без чтения контента. Но на современном железе, это не сильно заметно.

Горизонтальное масштабирование (Horizontal scaling)

Балансиры могут так же помочь с горизонтальным масштабированием, улучшая производительность и доступность. Масштабирование используя простые сервера гораздо дешевле дорогих решений и дорогих мощных серверов (вертикальное масштабирование). Так же гораздо проще и дешевле

найти специалиста умеющего работать с простым сервером нежели с каким-либо проприетарным.

Недостатки: горизонтального масштабирования

- Представляет из себя непростую задачу
    - Сервера не должны хранить какое либо состояние, например сессии или изображения пользователей
    - Сессии и прочее должно храниться отдельно в базах данных (SQL, NoSQL) или в кеше (Redis, Memcached)
- Кэширующие сервера должны обрабатывать большинство запросов

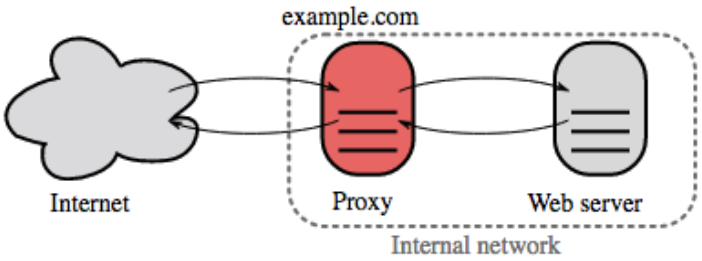
Недостатки:

- Балансиры могут быть узким местом системы, если сконфигурированы неверно или слишком слабые.
- Используя балансиры система уменьшает риски, но повышается сложность всей системы.
- Один балансер представляет из себя единственную точку отказа всей системы, конфигурирование нескольких балансиров только добавляет сложности всей системе.

Дополнительный материал

- NGINX architecture
- HAProxy architecture guide
- Scalability
- Wikipedia
- Layer 4 load balancing
- Layer 7 load balancing
- ELB listener config

Реверс прокси (веб-сервер) - Reverse proxy (web server



Реверс прокси это веб сервер который централизует внутренние сервисы и предоставляет унифицированный интерфейс наружу. Клиентские запросы пробрасываются к серверу который способен их обработать, а затем возвращает ответ.

Дополнительные плюсы:

- Повышенная безопасность - Прячет информацию о бекенд серверах, блокирует IP адреса, ограничивает количество соединений на клиента.
- Масштабируемость и гибкость - Клиенты видят только прокси сервер, а внутренняя конфигурация может быть любой.
- SSL терминация - Расшифровка входящих и шифровка исходящих пакетов в одном месте. Нет необходимости делать это на каждом сервере
- Сжатие - Сжатие ответов
- Кеширование - Возврат ответов на кэшированные запросы
- Статический контент - Отдает статический контент напрямую
    - HTML/CSS/JS
    - Фото
    - Видео
    - Прочее

Балансировщик vs реверс прокси

- Использование балансировщика полезно когда у вас несколько серверов. Чаще всего балансировщики используется для распределения нагрузки между серверами, которые выполняют одну и ту же задачу.
- Реверс прокси может быть полезным даже с одним сервером и предоставлять описанные выше полезные возможности.

- Такие решения как NGINX и HAProxy могут поддерживать layer 7 реверс прокси и балансировку нагрузки.

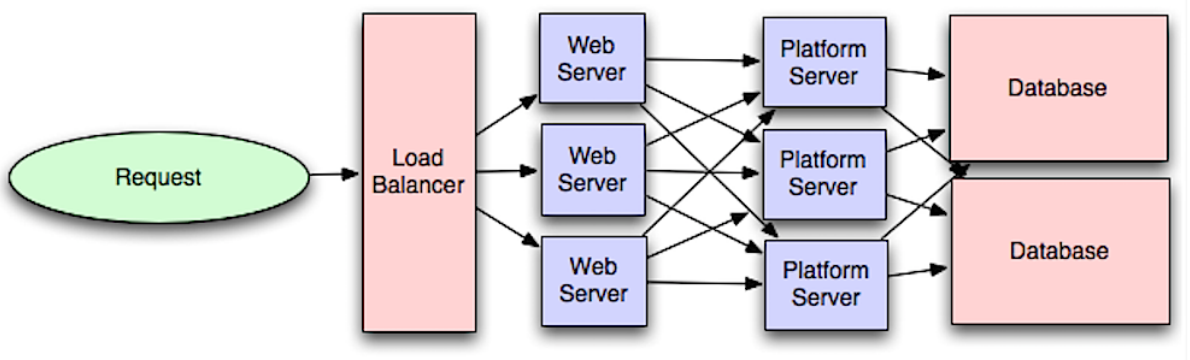
Недостатки:

- Использование реверс прокси увеличивает сложность системы.
- Использование одного реверс прокси сервера создает единую точку отказа. Решение этой проблемы увеличивает сложность системы.

Дополнительный материал

- Reverse proxy vs load balancer
- NGINX architecture
- HAProxy architecture guide
- Wikipedia

Слой/уровень приложения (Application layer)



Разделение веб уровня и приложения позволяет масштабировать и конфигурировать оба уровня отдельно друг от друга. Добавление нового API, не требует поднятия еще одного веб сервера.

Принцип единой ответственности позволяет писать маленькие автономные сервисы быстро и независимо друг от друга.

Микросервисы - Microservices

К данной дискуссии относятся микросервисы, которые могут быть описаны как набор независимых, маленьких и модульных сервисов выполняющих определенные задачи. Каждый сервис работает независимо и коммуницирует с другими по описанному API решая поставленные перед ним задачи. Pinterest, например, может иметь такие вот сервисы: пользовательский профиль, подписчики, лента, поиск, загрузка фото.

Термин «Microservice Architecture» получил распространение в последние несколько лет как описание способа дизайна приложений в виде набора независимо развертываемых сервисов. В то время как нет точного описания этого архитектурного стиля, существует некий общий набор характеристик: организация сервисов вокруг бизнес-потребностей, автоматическое развертывание, перенос логики от шины сообщений к приемникам (endpoints) и децентрализованный контроль над языками и данными.

«Микросервисы» — еще один новый термин на шумных улицах разработки ПО. И хотя мы обычно довольно настороженно относимся ко всем подобным новинкам, конкретно этот термин описывает стиль разработки ПО, который мы

находим все более и более привлекательным. За последние несколько лет мы видели множество проектов, использующих этот стиль, и результаты до сих пор были весьма позитивными. Настолько, что для большинства наших коллег этот стиль становится основным стилем разработки ПО. К сожалению, существует не так много информации, которая описывает, чем же являются микросервисы и как применять их.

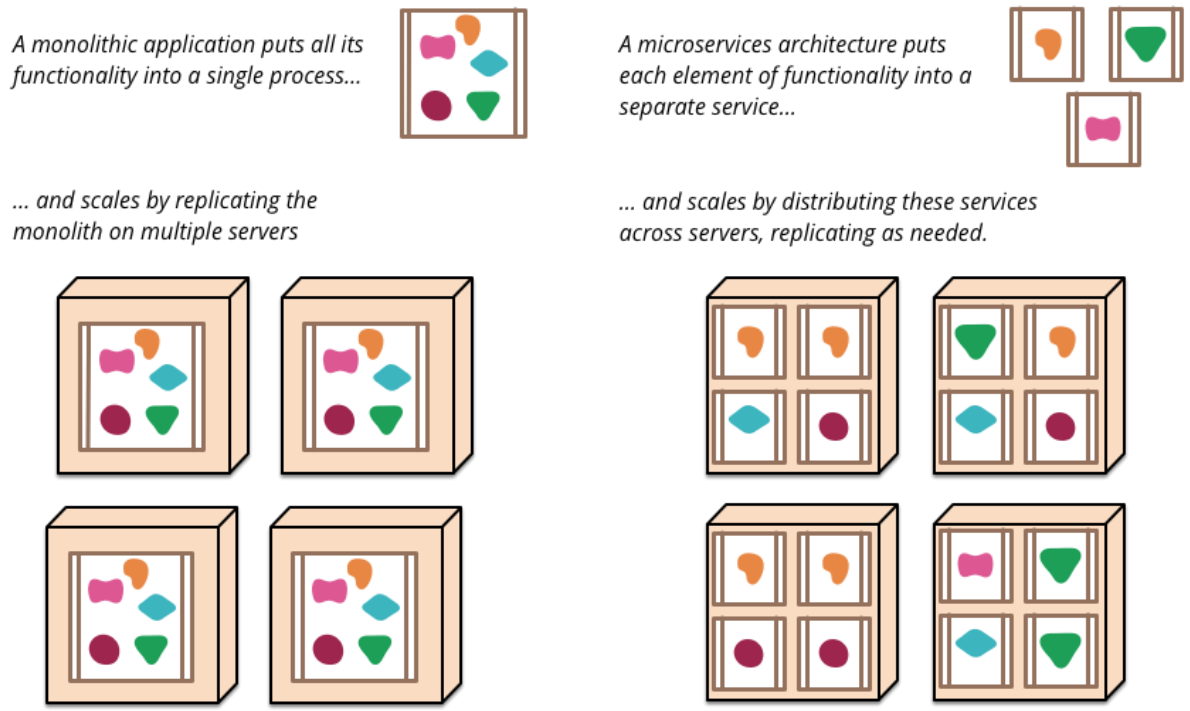
Если коротко, то архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо с использованием полностью автоматизированной среды. Существует абсолютный минимум централизованного управления этими сервисами. Сами по себе эти сервисы могут быть написаны на разных языках и использовать разные технологии хранения данных.

Для того, чтобы начать рассказ о стиле микросервисов, лучше всего сравнить его с монолитом (monolithic style): приложением, построенном как единое целое. Enterprise приложения часто включают три основные части: пользовательский интерфейс (состоящий как правило из HTML страниц и javascript-а), базу данных (как правило реляционной, со множеством таблиц) и сервер. Серверная часть обрабатывает HTTP запросы, выполняет доменную логику, запрашивает и обновляет данные в БД, заполняет HTML страницы, которые затем отправляются браузеру клиента. Любое изменение в системе приводит к пересборке и развертыванию новой версии серверной части приложения.

Монолитный сервер — довольно очевидный способ построения подобных систем. Вся логика по обработке запросов выполняется в единственном процессе, при этом вы можете использовать возможности вашего языка программирования для разделения приложения на классы, функции и namespace-ы. Вы можете запускать и тестировать приложение на машине разработчика и использовать стандартный процесс развертывания для проверки изменений перед выкладыванием их в продакшн. Вы можете масштабировать монолитное приложения горизонтально путем запуска нескольких физических серверов за балансировщиком нагрузки.

Монолитные приложения могут быть успешными, но все больше людей разочаровываются в них, особенно в свете того, что все больше приложений развертываются в облаке. Любые изменения, даже самые небольшие, требуют пересборки и развертывания всего монолита. С течением времени, становится труднее сохранять хорошую модульную структуру, изменения логики одного

модуля имеют тенденцию влиять на код других модулей. Масштабировать приходится все приложение целиком, даже если это требуется только для одного модуля этого приложения.



Эти неудобства привели к архитектурному стилю микросервисов: построению приложений в виде набора сервисов. В дополнение к возможности независимого развертывания и масштабирования каждый сервис также получает четкую физическую границу, которая позволяет разным сервисам быть написанными на разных языках программирования. Они также могут разрабатываться разными командами.

Мы не утверждаем, что стиль микросервисов это инновация. Его корни уходят далеко в прошлое, как минимум к принципам проектирования, использованным в Unix. Но мы тем не менее считаем, что недостаточно людей принимают во внимание этот стиль и что многие приложения получат преимущества если начнут применять этот стиль.

Свойства архитектуры микросервисов

Мы не можем сказать, что существует формальное определение стиля микросервисов, но мы можем попытаться описать то, что мы считаем общими

характеристиками приложений, использующих этот стиль. Не всегда они встречаются в одном приложении все сразу, но, как правило, каждое подобное приложение включает в себя большинство этих характеристик. Мы попробуем описать то, что мы видим в наших собственных разработках и в разработках известных нам команд.

Разбиение через сервисы

В течение всего срока нашего пребывания в индустрии мы видим желание строить системы путем соединения вместе различных компонент, во многом так же, как это происходит в реальном мире. За последние пару десятков лет мы видели большой рост набора библиотек, используемых в большистве языков программирования.

Говоря о компонентах, мы сталкиваемся с трудностями определения того, что такое компонент. Наше определение такого: компонент — это единица программного обеспечения, которая может быть независимо заменена или обновлена.

Архитектура микросервисов использует библиотеки, но их основной способ разбиения приложения — путем деления его на сервисы. Мы определяем библиотеки как компоненты, которые подключаются к программе и вызываются ею в том же процессе, в то время как сервисы — это компоненты, выполняемые в отдельном процессе и коммуницирующие между собой через веб-запросы или remote procedure call (RPS).

Главная причина использования сервисов вместо библиотек — это независимое развертывание. Если вы разрабатываете приложение, состоящее из нескольких библиотек, работающих в одном процессе, любое изменение в этих библиотеках приводит к переразвертыванию всего приложения. Но если ваше приложение разбито на несколько сервисов, то изменения, затрагивающие какой-либо из них, потребуют переразвертывания только изменившегося сервиса. Конечно, какие-то изменения будут затрагивать интерфейсы, что, в свою очередь, потребует некоторой координации между разными сервисами, но цель хорошей архитектуры микросервисов — минимизировать необходимость в такой координации путем установки правильных границ между микросервисами, а также механизма эволюции контрактов сервисов.

Другое следствие использования сервисов как компонент — более явный интерфейс между ними. Большинство языков программирования не имеют

хорошего механизма для объявления Published Interface. Часто только документация и дисциплина предотвращают нарушение инкапсуляции компонентов. Сервисы позволяют избежать этого через использование явного механизма удаленных вызовов.

Тем не менее, использование сервисов подобным образом имеет свои недостатки. Удаленные вызовы работают медленнее, чем вызовы в рамках процесса, и поэтому API должен быть менее детализированным (coarser-grained), что часто приводит к неудобству в использовании. Если вам нужно изменить набор ответственностей между компонентами, сделать это сложнее из-за того, что вам нужно пересекать границы процессов.

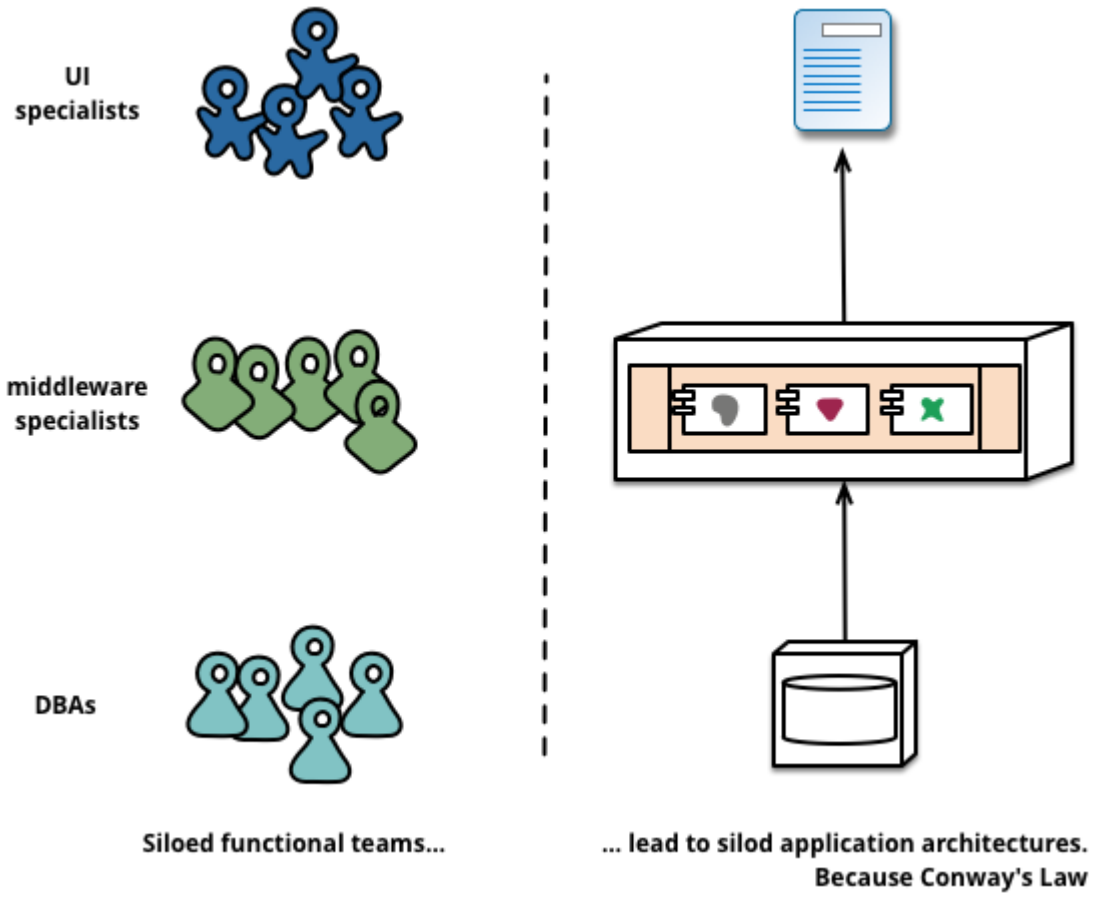
В первом приближении мы можем наблюдать, что сервисы соотносятся с процессами как один к одному. На самом деле сервис может содержать множество процессов, которые всегда будут разрабатываться и развертываться совместно. Например, процесс приложения и процесс базы данных, которую использует только это приложение.

Организация вокруг потребностей бизнеса

Когда большое приложение разбивается на части, часто менеджмент фокусируется на технологиях, что приводит к образованию UI команды, серверной команды и БД команды. Когда команды разбиты подобным образом, даже небольшые изменения отнимают много времени из-за необходимости кросс-командного взаимодействия. Это приводит к тому, что команды размещают любую логику на тех слоях, к которым имеют доступ. Закон Конвея (Conway's Law) в действии.

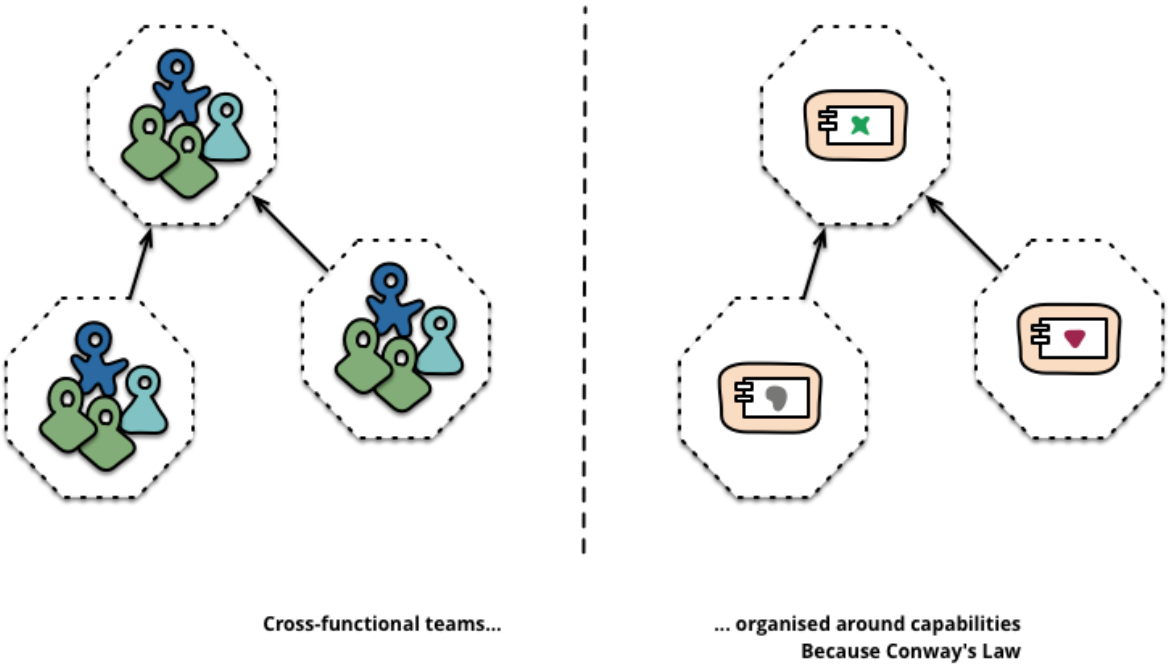
«Любая организация, которая проектирует какую-то систему (в широком смысле) получит дизайн, чья структура копирует структуру команд в этой организации»

— Melvyn Conway, 1967



Закон Конвея (Conway's Law) в действии

Микросервисный подход к разбиению подразумевает раазбиение на сервисы в соответствии с потребностями бизнеса. Такие сервисы включают в себя полный набор технологий, необходимых для этой бизнес-потребности, в том числе пользовательский интерфейс, хранилище данных и любые внешние взаимодействия. Это приводит к формированию кросс-функциональных команд, имеющих полный набор необходимых навыков: user-experience, базы данных и project management.



Сервисные границы, подкрепленные границами команд

Одна из компаний, организованных в этом стиле — www.comparethemarket.com. Кросс-фунциональные команды отвечают за построение и функционирование каждого продукта и каждый продукт разбит на несколько отдельных сервисов, общающихся между собой через шину сообщений.

Крупные монолитные приложения тоже могут быть разбиты на модули вокруг бизнес потребностей, хотя обычно этого не происходит. Безусловно, мы рекомендуем большим командам строить монолитные приложения именно таким образом. Основная проблема здесь в том, что такие приложения имеют тенденцию к организации вокруг слишком большого количества контекстов. Если монолит охватывает множество контекстов, отдельным членам команд становится слишком сложно работать с ними из-за их большого размера. Кроме того, соблюдение модульных границ в монолитном приложении требует существенной дисциплины. Явно очерченные границы компонент микросервисов упрощает поддержку этих границ.

Насколько большими должны быть микросервисы?

Хотя термин «Микросервис» стал популярным названием для этого архитектурного стиля, само имя приводит к чрезмерному фокусу на размере сервисов и спорам о том, что означает приставка «микро». В наших разговорах с теми, кто занимался разбиением ПО на микросервисы, мы видели разные размеры. Наибольший размер был у компаний, следовавших правилу «Команда двух пицц» (команда, которую можно накормить двумя пиццами), т.е.

не более 12 человек (*прим. перев.: следуя этому правилу, я в команде должен быть один*). В других компаниях мы видели команды, в которых шестеро человек поддерживали шесть сервисов.

Это приводит к вопросу о том, есть ли существенная разница в том, сколько человек должно работать на одном сервисе. На данный момент мы считаем, что оба этих подхода к построению команд (1 сервис на 12 человек и 1 сервис на 1 человека) подходят под описание микросервисной архитектуры, но возможно мы изменим свое мнение в будущем. (*прим. перев.: со времен статьи появилось множество других статей, развивающих эту тему; наиболее популярным сейчас считается мнение о том, что сервис должен быть настолько большим, чтобы он мог полностью «уместиться в голове разработчика», независимо от количества строк кода*).

Продукты, а не проекты

Большинство компаний по разработке ПО, которые мы видим, используют проектную модель, в которой целью является разработка некой части функциональности, которая после этого считается завершенной. После завершения эта часть передается команде поддержки и проектная команда распускается.

Сторонники микросервисов сторонятся этой модели, утверждая, что команда должна владеть продуктом на протяжении всего срока его жизни. Корни этого подхода уходят к Амазону, у компании есть правило "вы разработали, вам и поддерживать", при котором команда разработки берет полную ответственность за ПО в продакшне. Это приводит к тому, что разработчики регулярно наблюдают за тем, как их продукт ведет себя в продакшне, и больше контактируют с пользователями, т.к. им приходится брать на себя как минимум часть обязанностей по поддержке.

Мышление в терминах продукта устанавливает связь с потребностями бизнеса. Продукт — это не просто набор фич, которые необходимо реализовать. Это постоянные отношения, цель которых — помочь пользователям увеличить их бизнес-возможности.

Конечно, этого можно также достичь и в случае с монолитным приложением, но высокая гранулярность сервисов упрощает установку персональных отношений между разработчиками сервиса и его пользователями.

Умные приемники и глупые каналы передачи данных

(Smart endpoints and dumb pipes)

При выстраивании коммуникаций между процессами мы много раз были свидетелями того, как в механизмы передачи данных помещалась существенная часть логики. Хорошим примером здесь является Enterprise Service Bus (ESB). ESB-продукты часто включают в себя изощренные возможности по передаче, оркестровке и трансформации сообщений, а также применению бизнес-правил.

Комьюнити микросервисов предпочитает альтернативный подход: умные приемники сообщений и глупые каналы передачи. Приложения, построенные с использованием микросервисной архитектуры, стремятся быть настолько независимыми (decoupled) и сфокусировнными (cohesive), насколько возможно: они содержат собственную доменную логику и выступают больше в качестве фильтров в классическом Unix-овом смысле — получают запросы, применяют логику и отправляют ответ. Вместо сложных протоколов, таких как WS-* или BPEL, они используют простые REST-овые протоколы.

Два наиболее часто используемых протокола — это HTTP запросы через API ресурса и легковесный месседжинг. Лучшее выражение первому дал Ian Robinson: «Be of the web, not behind the web».

Команды, практикующие микросервисную архитектуру, используют те же принципы и протоколы, на которых построена всемирная паутина (и, по сути, Unix). Часто используемые ресурсы могут быть закешированы с очень небольшими усилиями со стороны разработчиков или IT-администраторов.

Второй часто используемый инструмент коммуникации — легковесная шина сообщений. Такая инфраструктура как правило не содержит доменной логики — простые реализации типа RabbitMQ или ZeroMQ не делают ничего кроме предоставления асинхронной фабрики. Логика при этом существует на концах этой шины — в сервисах, которые отправляют и принимают сообщения.

В монолитном приложении компоненты работают в одном процессе и коммуницируют между собой через вызов методов. Наибольшая проблема в смене монолита на микросервисы лежит в изменении шаблона коммуникации. Наивное портирование один к одному приводит к «болтливым» коммуникациям, которые работают не слишком хорошо. Вместо этого вы должны уменьшить количество коммуникаций между модулями.

Децентрализованное управление

Одним из следствий централизованного управления является тенденция к стандартизации используемых платформ. Опыт показывает, что такой подход слишком сильно ограничивает выбор — не всякая проблема является гвоздем и не всякое решение является молотком. Мы предпочитаем использовать правильный инструмент для каждой конкретной работы. И хотя монолитные приложения тоже в некоторых случаях могут быть написаны с использованием разных языков, это не является стандартной практикой.

Разбивая монолит на сервисы, мы имеем выбор, как построить каждый из них. Хотите использовать Node.js для простых страничек с отчетами? Пожалуйста. C++ для real-time приложений? Отлично. Хотите заменить БД на ту, которая лучше подходит для операций чтения вашего компонента? Ради бога.

Конечно, только потому что вы можете делать что-то, не значит что вы должны это делать. Но разбиение системы подобным образом дает вам возможность выбора.

Команды, разрабатывающие микросервисы, также предпочитают иной подход к стандартизации. Вместо того, чтобы использовать набор предопределенных стандартов, написанных кем-то, они предпочитают идею построения полезных инструментов, которые остальные девелоперы могут использовать для решения похожих проблем. Эти инструменты как правило вычленены из кода одного из проектов и расшарены между разными командами, иногда используя при этом модель внутреннего опен-сорса. Теперь, когда git и github стали де-факто стандартной системой контроля версий, опен-сорсные практики становятся все более и более популярными во внутренних проектах компаний.

Netflix — хороший пример организации, которая следует этой философии. Расшаривание полезного и, более того, протестированного на боевых серверах кода в виде библиотек побуждает остальных разработчиков решать схожие проблемы схожим путем, оставляя тем не менее возможность выбора другого подхода при необходимости. Общие библиотеки имеют тенденцию быть сфокусированными на общих проблемах, связанных с хранением данных, межпроцессорным взаимодействием и автоматизацией инфраструктуры.

Комьюнити микросервисов ценит сервисные контракты, но не любит оверхеды и поэтому использует различные пути управления этими контрактами. Такие шаблоны как Tolerant Readerи Consumer-Driven Contracts часто используются в микросервисах, что позволяет им эволюционировать независимо. Проверка Consumer-Driven контрактов как часть билда увеличивает уверенность в правильности функционирование сервисов. Мы знаем команду из Австралии,

которая использует этот подход для проверки контрактов. Это стало частью их процесса сборки: сервис собирается только до того момента, который удовлетворяет требованиям контракта — элегантный способ обойти диллему YAGNI.

Пожалуй наивысшая точка в практике децентрализованного управления — это метод, популизированный Амазоном. Команды отвечают за все аспекты ПО, которое они разрабатывают, включая поддержку его в режиме 24/7. Подобная деволюция уровня ответственности совершенно точно не является нормой, но мы видим все больше и больше компаний, передающий ответственность командам разработчиков. Netflix — еще одна компания, практикующая это. Пробуждение в 3 часа ночи — очень сильный стимул к тому, чтобы уделять большое внимание качеству написанного кода.

Микросервисы и SOA

Когда мы разговариваем о микросервисах, обычно возникает вопрос о том, не является ли это обычным Service Oriented Architecture (SOA), который мы видели десять лет назад. В этом вопросе есть здравое зерно, т.к. стиль микросервисов очень похож на то, что продвигают некоторые сторонники SOA. Проблема, тем не менее, в том, что термин SOA имеет сликом много разных значений и, как правило, то, что люди называют «SOA» существенно отличается от стиля, описанного здесь, обычно из-за чрезмерного фокуса на ESB, используемом для интеграции монолитных приложений.

В частности, мы видели так много неудачных реализаций SOA (начиная с тенденции прятать сложность за ESB, заканчивая провалившимися инициативами длительностью несколько лет, которые стоили миллионы долларов и не принесли никакой пользы), что порой слишком сложно абстрагироваться от этих проблем.

Безусловно, многие практики, используемые в микросервисах, пришли из опыта интеграции сервисов в крупных организациях. Шаблон Tolerant Reader — один из примеров. Другой пример — использование простых протоколов — возник как реакция на централизованные стандарты, сложность которых просто захватывает дух.

Эти проблемы SOA привели к тому, что некоторые сторонники микросервисов отказываются от термина «SOA», хотя другие при этом считают микросервисы одной из форм SOA, или, возможно, правильной реализацией SOA. В любом случае, тот факт, что SOA имеет разные значения, означает, что полезно иметь отдельный термин для обозначения этого архитектурного стиля.

Множество языков, множество возможностей

Рост платформы JVM — один из последних примеров смешивания языков в рамках единой платформы. Переход к более высокоуровневым языкам для получения преимуществ, связанных с использованием высокоуровневых абстракций, был распространенной практикой в течение десятилетий. Точно так же, как и переход «к железу» для написания высокопроизводительного кода.

Тем не менее, множество монолитных приложений не требуют такого уровня оптимизации производительности и высокоуровневых возможностей DSL-подобных языков. Вместо этого, монолиты как правило используют единый язык и склонны к ограничению количества используемых технологий.

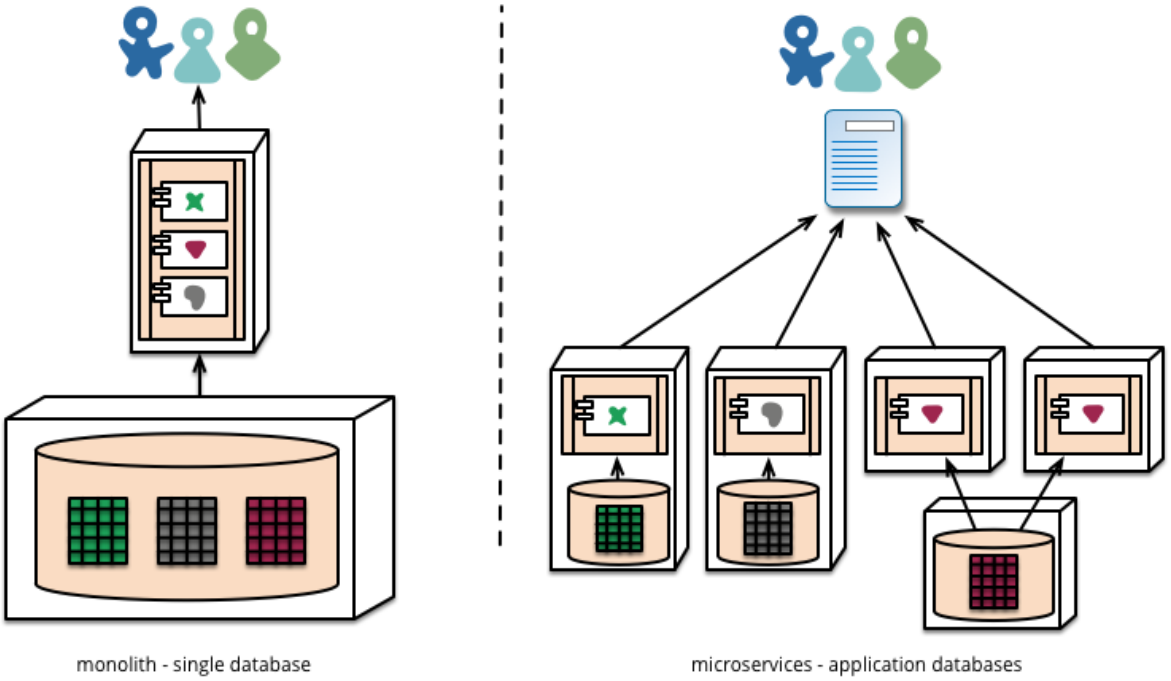
Децентрализованное управление данными

Децентрализованное управление данными предстает в различном виде. В наиболее абстрактном смысле это означает, что концептуальная модель мира у разных систем будет отличаться. Это обычная проблема, возникающая при интеграции разных частей больших enterprise-приложений: точка зрения на понятие «Клиент» у продажников будет отличаться от таковой у команды техподдержки. Некоторые атрибуты «Клиента» могут присутствовать в контексте продажников и отсутствовать в контексте техподдержки. Более того, атрибуты с одинаковым названием могут иметь разное значение.

Эта проблема встречается не только у разных приложений, но также и в рамках единого приложения, особенно в тех случаях когда это приложение разделено на отдельные компоненты. Эту проблему хорошо решает понятие Bounded Context из Domain-Driven Design (DDD). DDD предлагает делить сложную предметную область на несколько контекстов и мапить отношения между ними. Этот процесс полезен как для монолитной, так и для микросервисной архитектур, но между сервисами и контекстами существует естественная связь, которая помогает прояснять и поддерживать границы контекстов.

Кроме децентрализации принятия решений о моделировании предметной области, микросервисы также способствуют децентрализации способов хранения данных. В то время как монолитные приложения склонны к использованию единственной БД для хранения данных, компании часто предпочитают использовать единую БД для целого набора приложений. Такие решения, как правило, вызваны моделью лицензирования баз данных. Микросервисы предпочитают давать возможность каждому сервису управлять собственной базой данных: как создавать отдельные инстансы общей для компании СУБД, так и использовать нестандартные виды баз данных. Этот

подход называется Polyglot Persistence. Вы также можете применять Polyglot Persistence в монолитных приложениях, но в микросервисах такой подход встречается чаще.



Децентрализация ответственности за данные среди микросервисов оказывает влияние на то, как эти данные изменяются. Обычный подход к изменению данных заключается в использовании транзакций для гарантирования консистентности при изменении данных, находящихся на нескольких ресурсах. Такой подход часто используется в монолитных приложениях.

Подобное использование транзакций гарантирует консистентность, но приводит к существенной временной зависимости (temporal coupling), которая, в свою очередь, приводит к проблемамм при работе с множеством сервисов. Распределенные транзакции невероятно сложны в реализации и, как следствие, микросервисная архитектура придает особое значению координации между сервисами без использования транзакций с явным обозначением того, что консистентность может быть только итоговой (eventual consistency) и возникающие проблемы решаются операциями компенсации.

Управление несогласованностями подобным образом — новый вызов для многих команд разработки, но это часто соответствует практикам бизнеса. Часто компании стремятся как можно быстрее реагировать на действия

пользователя и имеют процессы, позвояющие отменить действия пользователей в случае ошибки. Компромисс стоит того до тех пор, пока стоимость исправления ошибки меньше стоимости потерь бизнеса при использовании сценариев, гарантирующих консистентность.

Стандартные, проверенные в бою, vs навязанные стандарты

Команды, использующие микросервисную архитектуру, склонны избегать жестких стандартов, установленных группами системных архитекторов. Они также склонны использовать и даже продвигать открытые стандарты типа HTTP и ATOM.

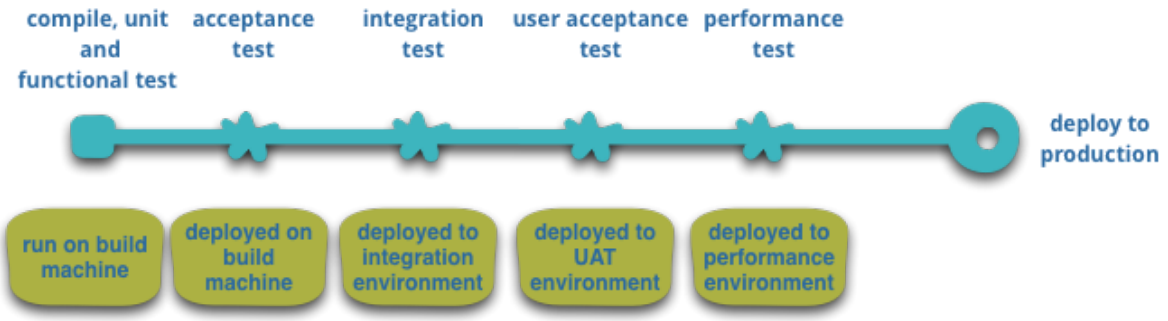
Ключевое отличие в том, как эти стандарты разрабатываются и как они проводятся в жизнь. Стандарты, управляемые группами вроде IETF, становятся стандартами только тогда, когда находятся несколько реализаций в успешных open-source проектах.

Это отличает их от стандартов в корпоративном мире, которые часто разрабатываются группами людей с небольшим опытом реальной разработки или имеют слишком сильное влияние, оказываемое вендорами.

Автоматизация инфраструктуры

Техники автоматизации инфраструктуры сильно эволюционировали за последние несколько лет. Эволюция облака в целом и AWS в частности уменьшила операционную сложность построения, разворачивания и функционирования микросервисов.

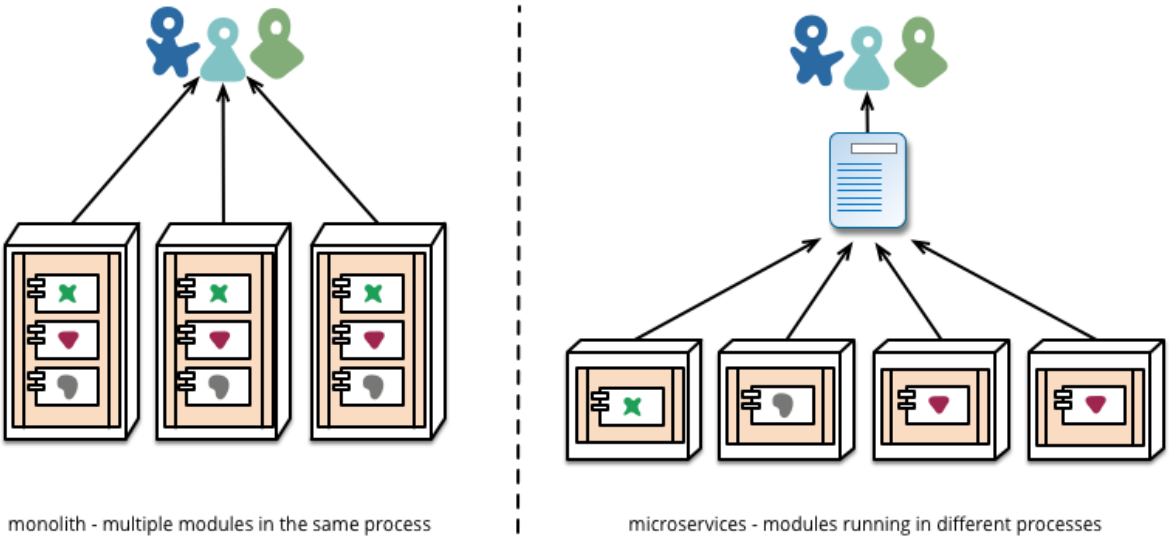
Множество продуктов и систем, использующих микросервисную архитектуру, были построены командами с обширным опытом в Continuous Delivery и Continuous Integration. Команды, строящие приложения подобнымм образом, интенсивно используют техники автоматизации инфраструктуры. Это проиллюстрировано на картинке ниже.



Так как эта статья не про Continuous Delivery, мы уделим внимание лишь паре его ключевых моментов. Мы хотим получать как можно больше уверенности в том, что наше приложение работает, поэтому мы запускаем множество автоматических тестов. Для выполнения каждого шага автоматического тестирования приложение разворачивается в отдельной среде, для чего используется автоматического развертывание (automated deployment).

После того как вы инвестировали время и деньги в автоматизацию процесса развертывания монолита, развертывание большего количества приложений (сервисов) уже не видится таким пугающим. Вспомните, что одна из целей Continuous Delivery — это сделать развертывание скучным, так что одно это приложение или три не имеет большого значения.

Другая область, где команды используют интенсивную автоматизацию инфраструктуры, — это управление микросервисами в продакшне. В отличие от процесса развертывания, который, как описано выше, у монолитных приложений не сильно отличается от такового у микросервисов, их способ фунционирования может существенно различаться.



Одним из побочных эффектов автоматизации процесса развертывания является создание удобных инструментов для помощи разработчикам и администраторам (operations folk). Инструменты для управления кодом, развертывания простых сервисов, мониторинга и логирования сейчас довольно распространены. Возможно наилучший пример, который можно найти в сети, — это набор open source инструментов от Netflix, но существуют и другие, к примеру Dropwizard, который мы довольно интенсивно используем.

Проектирование под отказ (Design for failure)

Следствием использования сервисов как компонентов является необходимость проектирования приложений так, чтобы они могли работать при отказе отдельных сервисов. Любое обращение к сервису может не сработать из-за его недоступности. Клиент должен реагировать на это настолько терпимо, насколько возможно. Это является недостатком микросервисов по сравнению с монолитом, т.к. это вносит дополнительную сложность в приложение. Как следствие, команды микросервисов постоянно думают на тем, как недоступность сервисов должна влиять на user experience. Simian Army от Netflix искуственно вызывает (симулирует) отказы сервисов и даже датацентров в течение рабочего дня для тестирования отказоустойчивости приложения и служб мониторинга.

Подобный вид автоматического тестирования в продакшне позволяет сэмулировать стресс, который ложится на администраторов и часто приводит к работе по выходным. Мы не хотим сказать, что для монолитных приложений не могут быть разработаны изощренные системы мониторинга, только то, что такое встречается реже.

Так как сервисы могут отказать в любое время, очень важно иметь возможность быстро обнаружить неполадки и, если возможно, автоматически восстановить работоспособность сервиса. Микросервисная архитектура делает большой акцент на мониторинге приложения в режиме реального времени, проверке как технических элементов (например, как много запросов в секунду получает база данных), так и бизнес-метрик (например, как много заказов в минуту получает приложение). Семантический мониторинг может предоставить систему раннего предупреждения проблемных ситуаций, позволяя команде разработке подключиться к исследованию проблемы на самых ранних стадиях.

Это особенно важно с случае с микросервисной архитектурой, т.к. разбиение на отдельные процессы и коммуникация через события приводит к неожиданному поведению. Мониторинг крайне важен для выявления нежелательных случаев такого поведения и быстрого их устранения.

Монолиты могут быть построены так же прозрачно, как и микросервисы. На самом деле, так они и должны строиться. Разница в том, что знать, когда сервисы, работающие в разных процессах, перестали корректно взаимодействовать между собой, намного более критично. В случае с библиотеками, расположенными в одном процессе, такой вид прозрачности скорее всего будет не так полезен.

Команды микросервисов, как правило, создают изощренные системы мониторинга и логирования для каждого индивидуального сервиса. Примером может служить консоль, показывающая статус (онлайн/офлайн) сервиса и различные технические и бизнес-метрики: текущая пропускная способность, время обработки запроса и т.п.

Синхронные вызовы считаются опасными

Каждый раз когда вы имеете набор синхронных вызовов между сервисами, вы сталкиваетесь с эффектом мультипликации времени простоя (downtime). Время простоя вашей системы становится произведением времени простоя индивидуальных компонент системы. Вы сталкиваетесь с выбором: либо сделать ваши вызовы асинхронными, либо мириться с простоями. К примеру, в www.guardian.co.uk разработчики ввели простое правило — один синхронный вызов на один запрос пользователя. В Netflix же вообще все API являются асинхронными.

Эволюционный дизайн

Те, кто практикует микросервисную архитектуру, обычно много работали с эволюционным дизайном и рассматривают декомпозицию сервисов как дальнейшую возможность дать разработчикам контроль над изменениями (рефакторингом) их приложения без замедления самого процесса разработки. Контроль над изменениями не обязательно означает уменьшение изменений: с правильным подходом и набором инструментов вы можно делать частые, быстрые, хорошо контролируемые изменения.

Каждый раз когда вы пытаетесь разбить приложение на компоненты, вы сталкиваетесь с необходимостью принять решение, как именно делить приложение. Есть ли какие-то принципы, указывающие, как наилучшим способом «нарезать» наше приложение? Ключевое свойство компонента — это независимость его замены или обновления, что подразумевает наличие ситуаций когда его можно переписать с нуля без затрагивания взаимодействующих с ним компонентов. Многие команды разработчиков идут еще дальше: они явным образом планируют, что множество сервисов в долгосрочной перспективе не будет эволюционировать, а будут просто выброшены на свалку.

Веб-сайт Guardian — хороший пример приложения, которое было спроектировано и построено как монолит, но затем эволюционировало в сторону микросервисов. Ядро сайта все еще остается монолитом, но новые фичи добавляются путем построения микросервисов, которые используют API монолита. Такой подход особенно полезен для функциональности, которая по сути своей является временной. Пример такой функциональности — специализированные страницы для освещения спортивных событий. Такие части сайта могут быть быстро собраны вместе с использованием быстрых языков программирования и удалены как только событие закончится. Мы видели похожий подход в финансовых системах, где новые сервисы добавлялись под открывшиеся рыночные возможности и удалялись через несколько месяцев или даже недель после создания.

Такой упор на заменяемости — частный случай более общего принципа модульного дизайна, который заключается в том, что модульность определяется скоростью изменения функционала. Вещи, которые изменяются вместе, должны храниться в одном модуле. Части системы, изменяемые редко, не должны находиться вместе с быстроэволюционирующими сервисами. Если вы регулярно меняете два сервиса вместе, задумайтесь над тем, что возможно их следует объединить.

Помещение компонент в сервисы добавляет возможность более точного (granular) планирования релиза. С монолитом любые изменения требуют пересборки и развертывания всего приложения. С микросервисами вам нужно

развернуть (redeploy) только те сервисы, что изменились. Это позволяет упростить и ускорить процесс релиза. Недостаток такого подхода в том, что вам приходится волноваться насчет того, что изменения в одном сервисе сломают сервисы, обращающиеся к нему. Традиционный подход к интеграции заключается в том, чтобы решать такие проблемы путем версионности, но микросервисы предпочитают использовать версионность только в случае крайней необходимости. Мы можем избежать версионности путем проектирования сервисов так, чтобы они были настолько толерантны к изменениям соседних сервисов, насколько возможно.

За микросервисами будущее?

Наша основная цель при написании этой статьи заключалась в том, чтобы объяснить основные идеи и принципы микросервисной архиктуры. Мы считаем, что микросервисный стиль — важная идея, стоящая рассмотрения для enterprise приложений. Не так давно мы разработали несколько систем используя этот стиль и знаем несколько других команд, которые используют этот подход.

Известные нам пионеры этого архитектурного стиля — это такие компании как Amazon, Netflix, The Guardian, the UK Government Digital Service, realestate.com.au, Forward и comparethemarket.com. Конференции 2013 года были полны примеров компаний, движущихся в направлении, которое можно классифицировать как микросервисы, например, Travis CI. К тому же, существует множество организаций, которые уже давно используют то, что мы называем микросервисами, но не используют это название. (Часто это называется SOA, хотя, как мы уже говорили, SOA может являться в самых разных и, зачастую, противоречивых формах.)

Несмотря на весь этот положительный опыт, мы не утверждаем, что микросервисы — это будущее проектирования ПО. И хотя наш опыт пока что весьма позитивен по сравнению с опытом использования монолитной архитектуры, мы подходим осознанно к тому факту, что прошло еще недостаточно времени для того, чтобы выносить такое суждение.

Часто настоящие последствия ваших архитектурных решений становятся видно только спустя несколько лет после того, как вы сделали их. Мы видели проекты, в которых хорошие команды с сильным стремлением к модульности разработали монолитные приложения, полностью «прогнившие» по прошествию нескольких лет. Многие считают, что такой результат менее вероятен в случае с микросервисами, т.к. границы между сервисами являются физическими и их сложно нарушить. Тем не менее, до тех пока мы не увидим достаточного количества проверенных временем систем, использующих этот подход, мы не можем с уверенностью утверждать, насколько микросервисная архитектура является зрелой.

Определенно существуют причины, по которым кто-то может считать микросервисную архитектуру недостаточно зрелой. Успех любых попыток построить компонентную систему зависит от того, насколько хорошо компоненты подходят приложению. Сложно понять где именно должны лежать границы компонентов. Эволюционный дизайн осознает сложности проведения правильных границ и важность легкого их изменения. Когда ваши компоненты являются сервисами, общающимися между собой удаленно, проводить рефакторинг намного сложнее, чем в случае с библиотеками, работающими в одном процессе. Перемещение кода между границами сервисов, изменение интерфейсов должны быть скоординированы между разными командами. Необходимо добавлять слои для поддержки обратной совместимости. Все это также усложняет процесс тестирования.

Еще одна проблема состоит в том, что если компоненты не подобраны достаточно чисто, происходит перенос сложности из компонент на связи между компонентами. Создается ложное ощущение простоты отдельных компонент, в то время как вся сложность находится в местах, которые труднее контролировать.

Также существует фактор уровня команды. Новые техники как правило принимаются более сильными командами, но техники, которые являются более эффективными для более сильных команд, необязательно являются таковыми для менее сильных групп разработчиков. Мы видели множество случаев, когда слабые команды разрабатывали запутанные, неудачные архитектуры монолитных приложений, но пройдет время прежде чем мы увидим чем это

закончится в случае с микросервисной архитектурой. Слабые команды всегда создают слабые системы, сложно сказать улучшат ли микросервисы эту ситуацию или ухудшат.

Один из разумных аргументов, которые мы слышали, состоит в том, что вам не следует начинать разработку с микросервисной архитектуры. Начните с монолита, сохраняйте его модульным и разбейте на микросервисы когда монолит станет проблемой. (И все же этот совет не является идеальным, т.к. хорошие интерфейсы для сообщения внутри процесса не являются таковыми в случае с межсервисным сообщением.)

Итого, мы пишем это с разумным оптимизмом. К этому моменту мы видели достаточно примеров микросервисного стиля чтобы осознавать, что он является стоящим путем развития. Нельзя сказать с уверенностью к чему это приведет, но одна из особенностей разработки ПО заключается в том, что нам приходится принимать решения на основе той, зачастую неполной, информации, к который мы имеет доступ в данный момент.

Обнаружение сервисов - Service Discovery

Такие системы как Consul, Etcd, Zookeeper, помогают сервисам находить друг друга (порты, адреса). Health checks (проверки здоровья), помогают убедиться, в работоспособности сервиса и зачастую делается с помощью простого HTTP запроса.

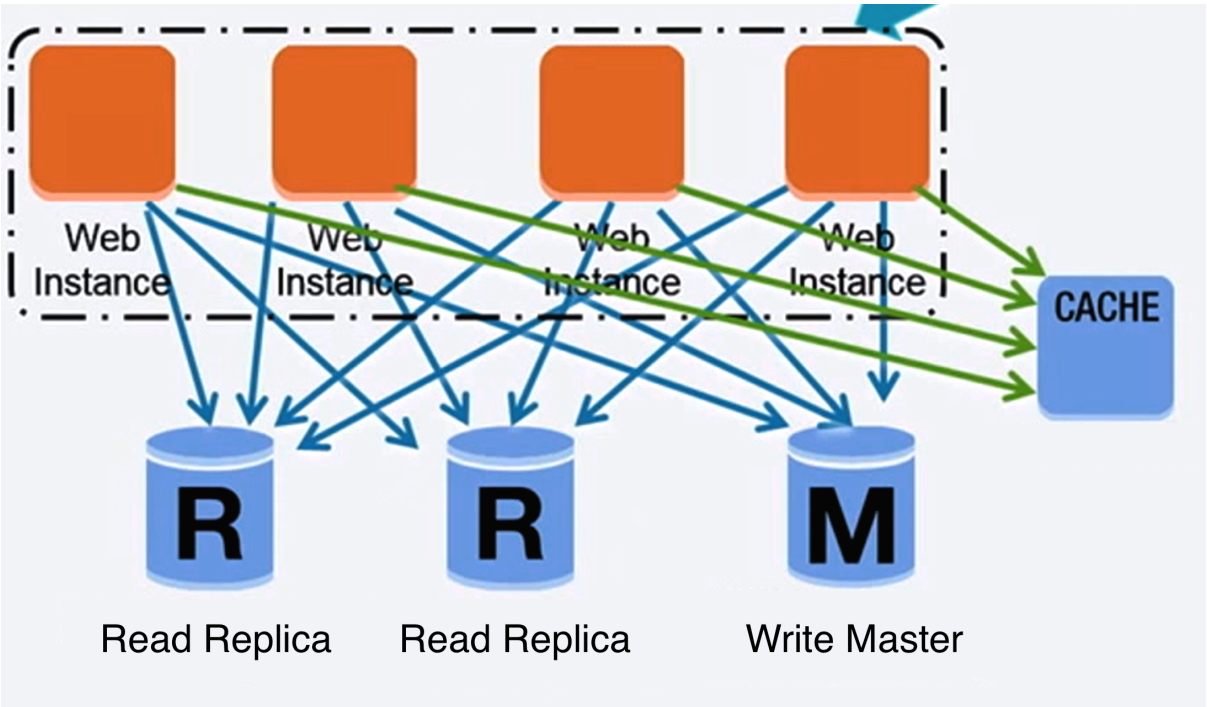
Недостатки: application layer

- Добавление уровня приложения с разделенными сервисами требует совершенного иного подхода в отличии от монолитного.
- Микросервисы добавляют сложности в деплоймент и развертку.

Дополнительный материал

- Intro to architecting systems for scale
- Crack the system design interview
- Service oriented architecture
- Introduction to Zookeeper
- Here's what you need to know about building microservices

База данных



Реляционная система управления базами данных (RDBMS)

Реляционная база данных использующая SQL является набором данных организованных в таблицы

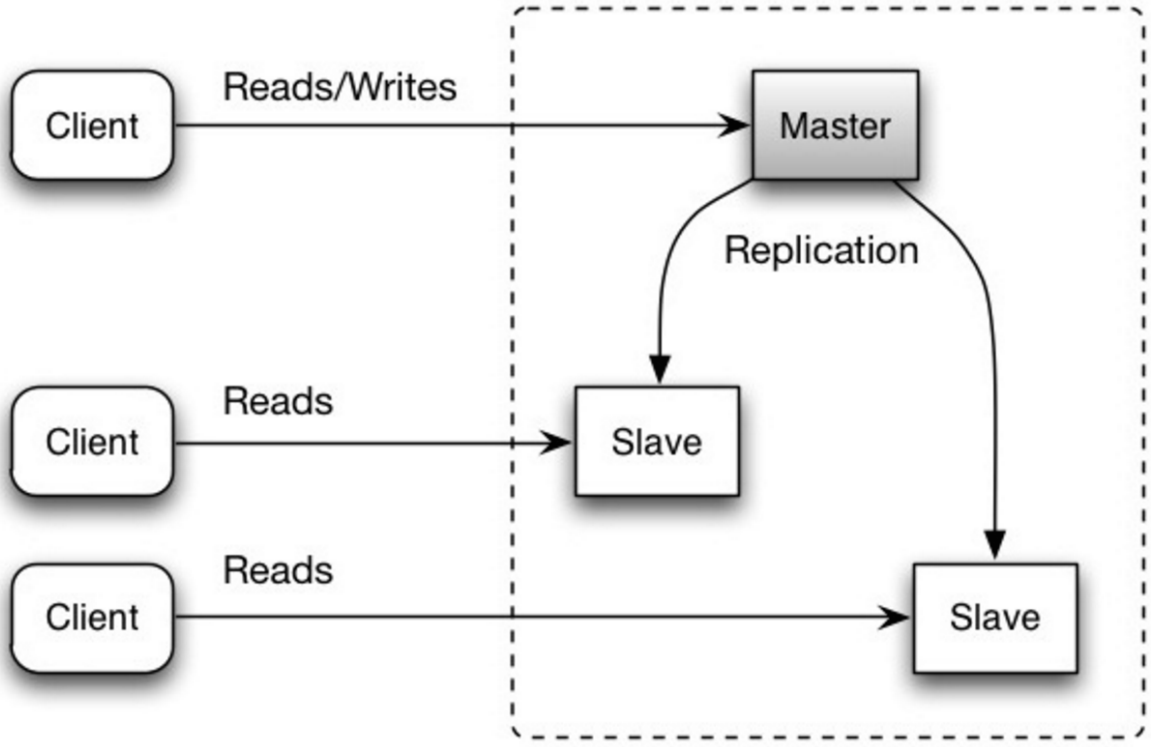
ACID - это набор свойств транзакций в базе данных

- Atomicity - Каждая транзакция выполняется целиком или не выполняется вовсе.
- Consistency - Каждая транзакция переводит базу данных из одного стабильного состояния к стабильному новому.
- Isolation - Выполнения транзакций поочередно или асинхронно должны приводить базу данных к одному и тому же состоянию.
- Durability - После выполнения транзакции, состояния базы данных не меняется.

Существует множество техник масштабирования реляционных баз данных. Мастре-раб репликация, мастер-мастер репликация, федерилизация, шардинг, денормализация, SQL тюнинг.

Мастер-раб репликация - Master-slave replication

Мастер сервера читают и пишут, реплицируя запись на одного или более рабов, с которых сервер считывает данные. Рабы также могут реплицировать данные на дополнительных рабов, в древовидной структуре. Если мастер упал, то система продолжает выполнение в только чтении и может повысить раба до мастера для поддержки записи.

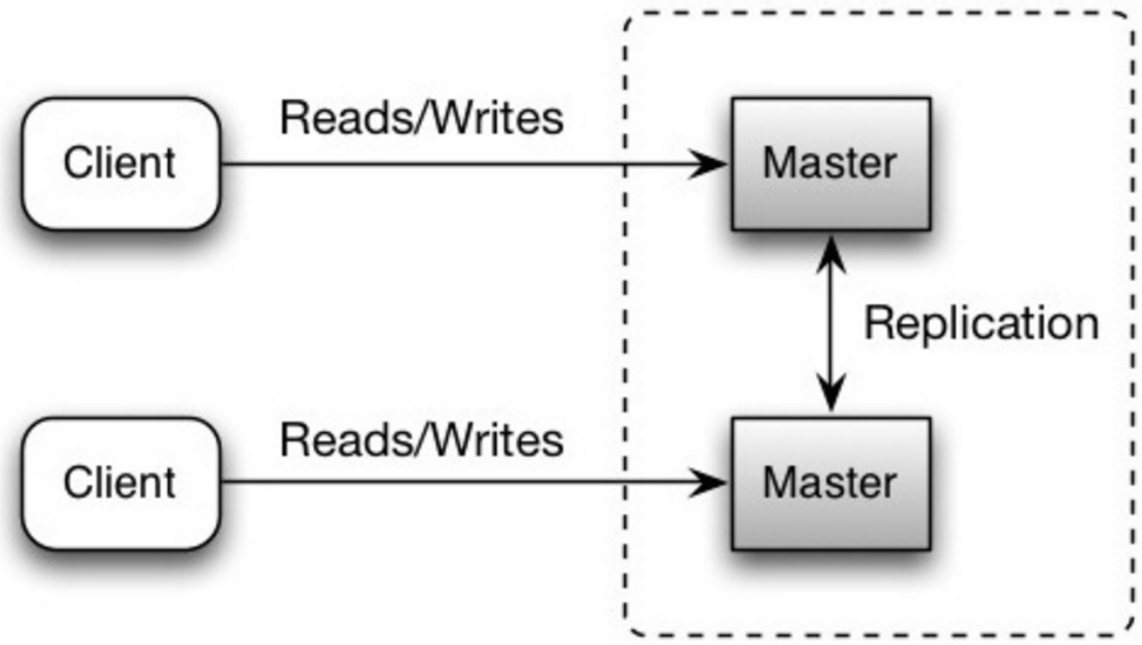


Недостатки: master-slave:

- Необходима дополнительная логика повышения раба в мастер.
- Сложность репликации

Мастер-мастер репликация - Master-master replication

Оба мастера читают и пишут координируя запись между собою. Если какой либо из мастеров упал, система продолжает работать с остальными мастерами.



Недостатки: master-master replication

- You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
- Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
- Conflict resolution comes more into play as more write nodes are added and as latency increases.
- See Disadvantage(s): replication for points related to both master-slave and master-master.

Недостатки: репликация

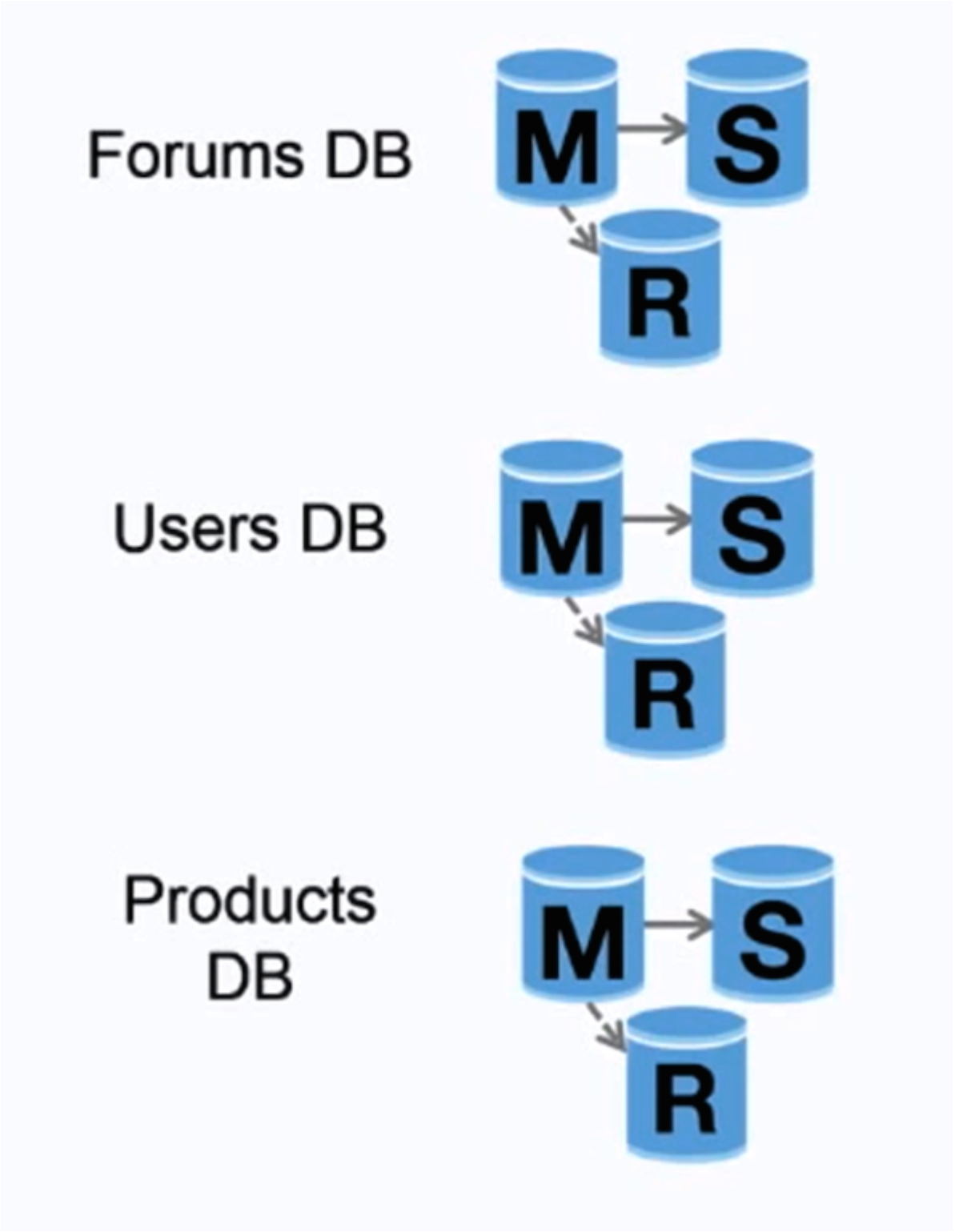
- Существует потенциальный риск потери данных если мастер упадет раньше записи новых данных на другие ноды.
- Записи повторяются на реплики с которых происходит чтение. Если операций много, то может нарушаться чтение.
- Чем больше рабов чтения, тем больше задержка репликации.

- Репликации требуют большего количества железа и добавляют сложности системе.

Дополнительный материал

- Scalability, availability, stability, patterns
- Multi-master replication

Федерализация - Federation



Федерализация это разбиение базы по ее функциями. Например вместо монолитной базы данных, иметь несколько баз данных под различные данные. Например: пользователи, продукты, покупки. Тем самым уменьшается трафик

чтения и записи, что уменьшает сложность репликаций. Меньшие базы данных также позволяют успешно настраивать системы кеширования. Без единого мастера, можно настраивать параллельную запись.

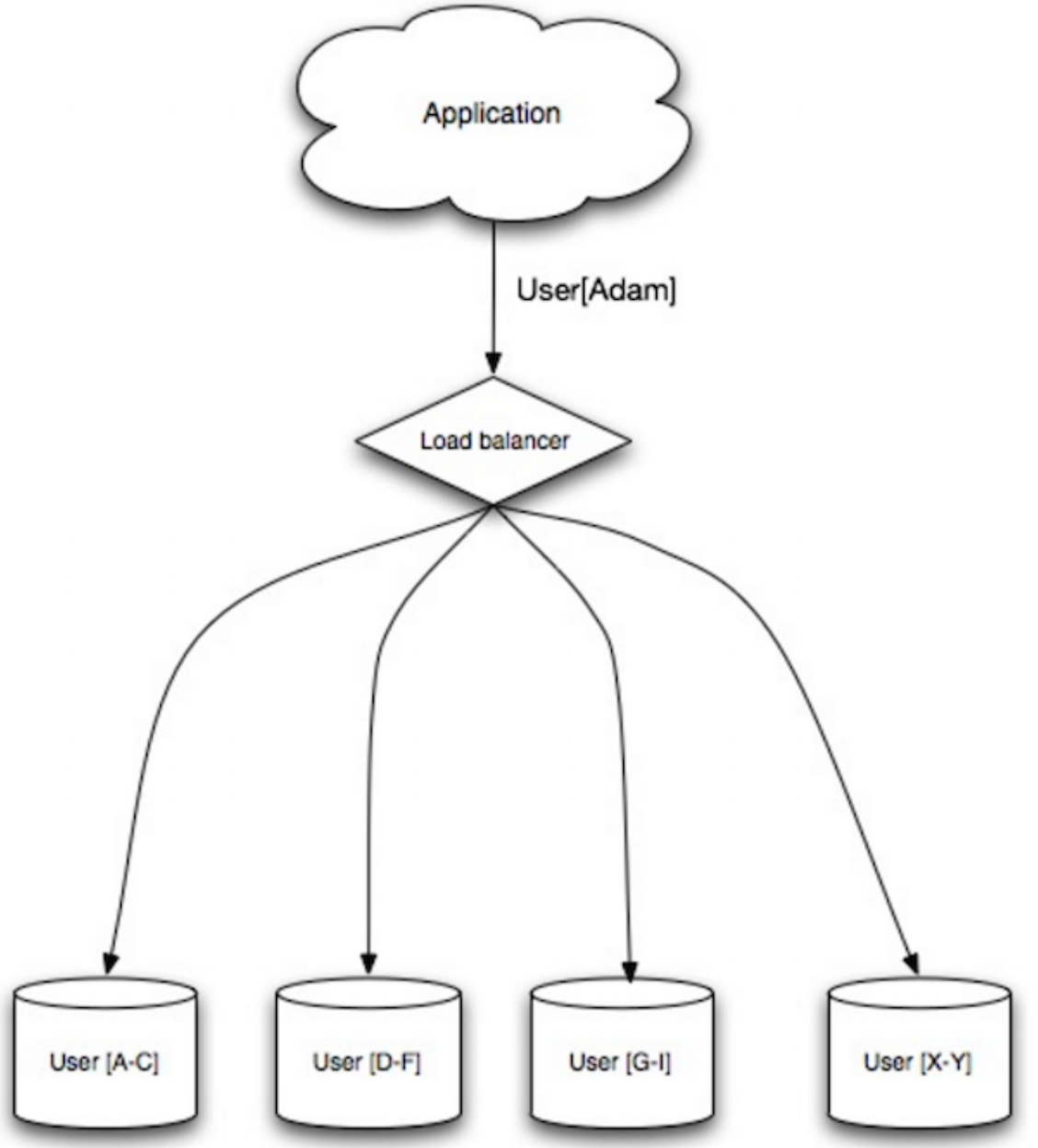
Минусы:

- Неэффективно если схема таблиц слишком большая.
- Необходимо обновлять логику приложение по которой идет запись в ту или иную базу данных.
- Работа с двумя и более базами данных намного сложней.
- Добавляет сложность и железо к системе

Дополнительный материал

- Scaling up to your first 10 million users

Шардирование/сегментация/шардинг - Sharding



Шардинг распределяет данные по различным базам данных. Чем больше данных тем больше шардов. Это похоже на федерализацию, и имеет те же плюсы и минусы, но тут данные распределяются скорее по количеству и схожим параметрам нежели по их доменной значимости. Например пользователей можно разбивать по географии или инициалам.

Недостатки: sharding

- Более сложная логика и возможно более сложные SQL запросы.
- Данные могут распределяться не совсем равномерно в плане частоты их чтения. Например профили одних пользователей могут открываться чаще чем других, тем самым увеличивая нагрузку на один шард. Балансировка таких моментов добавляет сложности.
- Чтение из нескольких шардов сложней.

Дополнительный материал

- The coming of the shard
- Shard database architecture
- Consistent hashing

Денормализация

Денормализация это попытка улучшить чтение ценой производительности записи. Избыточные копии данных записываются в несколько таблиц, для уменьшение сложных объединений. Некоторые СУБД поддерживают такой функционал (PostgreSQL, Oracle)

После распределения данных подходом федерализации и шардинга, чтение связанных данных усложняется. Денормализация можешь облегчить этот момент.

В системах которых чтение происходит намного чаще записи (1000:1), объединение данных для чтения может существенно влиять на производительность.

Недостатки:

- Дупликации данных
- Ограничения могут помочь с излишними копиями, что в свою очередь увеличивать сложность системы.
- Данный подход при частых записях не оправдывает себя. Следует использовать в основном в системах с большим количеством операций чтения.

Дополнительный материал

- Denormalization

SQL тюнинг

SQL тюнинг это обширная тема и множество книг было написано на эту тему.

Очень важно тестировать свою базу данных и находить узкие места.

- Benchmark - Эмуляция высокой нагрузки и тестом производительности.
- Profile - Профилирование работы базы данных, для нахождения проблемных мест.

Хорошо продуманная схема

- MySQL запись на диск в непрерывном режиме.
- Использование `CHAR` вместо `VARCHAR` для полей с фиксированным размером.
    - `CHAR` обеспечивает быстрый рандомный доступ, в то время как с `VARCHAR`, необходимо найти конец строки, чтобы двигаться к следующей.
- Использование `TEXT` для больших блоков текста. `TEXT` так же позволяет делать булевый поиск. Использование `TEXT` поля позволяет хранить указатель на диске который определяет блок текста.
- Использование `INT` для больших чисел 2^32 или 4 млрд.
- Использование `DECIMAL` для денежных единиц.
- Не стоит хранить большие `BLOBS`, лучше хранить локацию к объекту.
- Установка `NOT NULL` ограничений для улучшения поиска.

Use good indices

- Запрос колонок (`SELECT`, `GROUP BY`, `ORDER BY`, `JOIN`) может быть быстрей с индексами.
- Индексы обычно представляют из себя самобалансирующиеся B-деревья которые хранят данные отсортированными и позволяют производить поиск, поочередный доступ, вставку и удаление логарифмической сложности.
- Индексы могут хранить данные в памяти при этом требуя больше места.
- Скорость записи может уменьшаться из за индексов, так как их необходимо обновлять.
- При загрузке больших данных, решение с отключением индексов на время загрузки и последующее обновление индексов может уменьшить время записи по сравнению с постоянной индексацией по мере записи.

Уменьшение больших объединений

- Денормализация

Партиции

- Разбивка таблицы и загрузка горячих данных в память для более быстрого доступа.

Кеш запросов

- В некоторых случая кеш в запросах может привести к проблемам производительности.

Дополнительный материал

- Tips for optimizing MySQL queries
- Is there a good reason i see VARCHAR(255) used so often?
- How do null values affect performance?
- Slow query log

NoSQL

NoSQL это коллекция сущностей представленных в виде ключ-значение хранилище, документов хранилище, широко колоночном хранилище. Данные денормализованы, объединения обычно делаются на уровне приложения. Большинство NoSQL баз не поддерживает ACID.

BASE часто используется для определения свойств NoSQL. В отличии от CAP теоремы BASE выбирает доступность, консистенции данных.

- Basically available - гарантирует доступность.
- Soft state - состояние системы может измениться даже без выполнения каких либо действий.
- Eventual consistency - система станет консистентной на некоторый период в течении которого она не будет получать какой либо ввод.

В дополнение, выбор между SQL и NoSQL, важно понимать, какой тип NoSQL баз данных подходит под конкретную задачу. Рассмотрим их.

Ключ-значение хранилище - Key-value store

Абстракция: hash table

Key-value store обычно позволяет производить операции сложностью O(1), . Хранилища могут поддерживать ключи в лексическом порядке, позволяя эффективно извлекать множество ключей. Также такие хранилища могут хранить различного рода мета-данные наравне вместе со значениями.

Предоставляют высокую производительность и часто используются для простых моделей данных или для быстро изменяющихся данных, например в in-memory кешах. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

Хранилища типа ключ-значение являются базисом для более сложных систем таких как файловая база данных и в некоторых случая графовые бд.

Дополнительный материал

- Key-value database
- Disadvantages of key-value stores
- Redis architecture
- Memcached architecture

Файловая/документная база данных - Document store

Абстракция: key-value store с документами сохраненными как значения

Файловые базы данных строются вокруг документов формата XML, JSON, binary и прочих. Документ хранит все данные объекта. Для работы с такими документами предоставляется API или язык запросов базирующийся на структуре документов.

По внутренней имплементации, документы могут быть организованы в коллекции, теги, метадату или директории. Однако документы сгруппированные вместе, могут иметь совершенно различные поля.

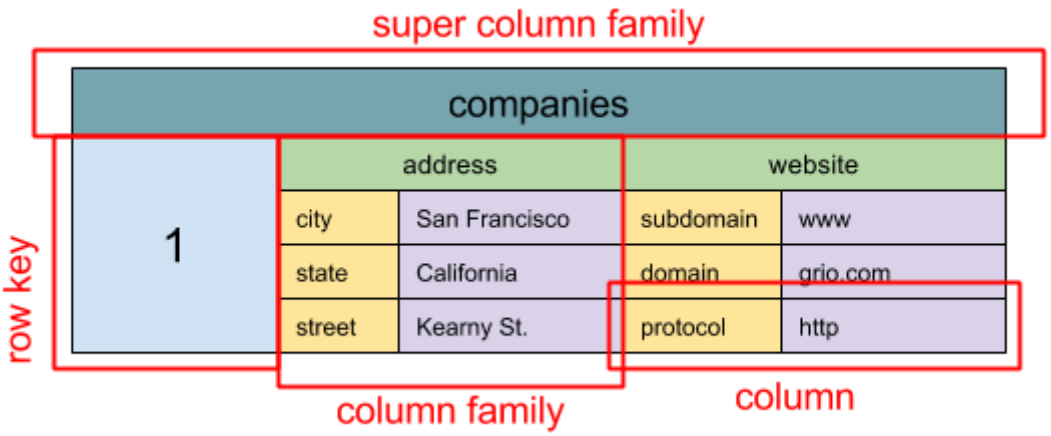
Такие базы данных как MongoDB или CouchDB так же предоставляют похожий на SQL язык запросов. DynamoDB поддерживает как и ключ-значение так и файловое хранение.

Файловые базы данных предоставляют высокую гибкость и часто используются в системах где данные часто меняются.

Дополнительный материал

- Document-oriented database
- MongoDB architecture
- CouchDB architecture
- Elasticsearch architecture

Колоночные базы данных - Wide column store



Абстракция: вложенный map `ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>`

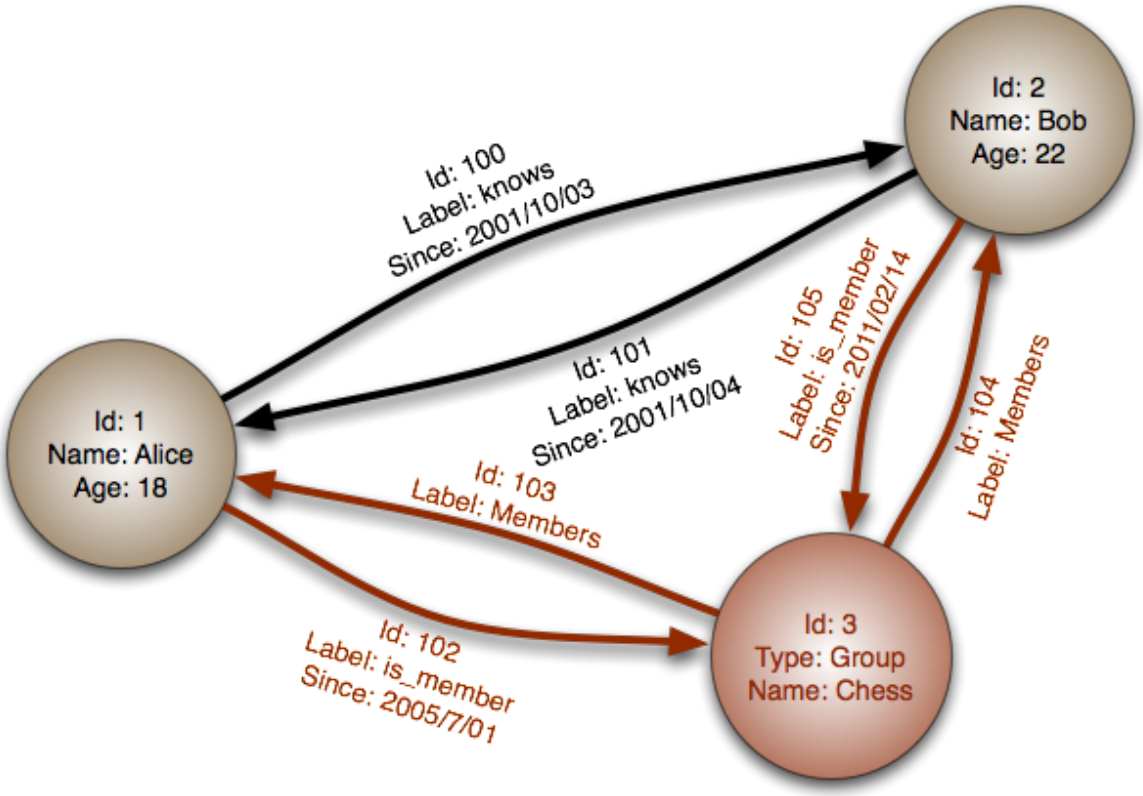
Колоночные базы данных используют в качестве базисной единицы данных - колонку. Колонка может быть сгруппирована в семейную колонку (аналогия таблиц в SQL).

Колоночные базы данных очень доступны и очень хорошо масштабируются. Они часто используются при обработке больших массивов данных

Дополнительный материал

- SQL & NoSQL, a brief history
- Bigtable architecture
- HBase architecture
- Cassandra architecture

Графовая бд - Graph database



Абстракция: graph

В графовых базах данных каждая нода это запись, а ребра - связи. Графовые бд, оптимизированы для представления комплексных связей между множеством ключей.

Графовые бд часто имеют высокую производительность работая с данными имеющие множество связей, например как соцсетях.

Дополнительный материал

- Graph database
- Neo4j
- FlockDB
- Explanation of base terminology
- NoSQL databases a survey and decision guidance
- Scalability
- Introduction to NoSQL
- NoSQL patterns
- From cache to in-memory data grid

- Scalable system design patterns
- Introduction to architecting systems for scale
- Scalability, availability, stability, patterns
- Scalability
- AWS ElastiCache strategies
- Wikipedia
- What is HTTP?
- Difference between HTTP and TCP
- Difference between PUT and PATCH
- Networking for game programming
- Key differences between TCP and UDP protocols
- Difference between TCP and UDP
- Transmission control protocol
- User datagram protocol
- Scaling memcache at Facebook
- Do you really know why you prefer REST over RPC
- When are RPC-ish approaches more appropriate than REST?
- REST vs JSON-RPC
- Debunking the myths of RPC and REST
- What are the drawbacks of using REST
- Crack the system design interview
- Thrift
- Why REST for internal use and not RPC
- Security guide for developers
- OWASP top ten