Урок 8 > Повышение отзывчивости

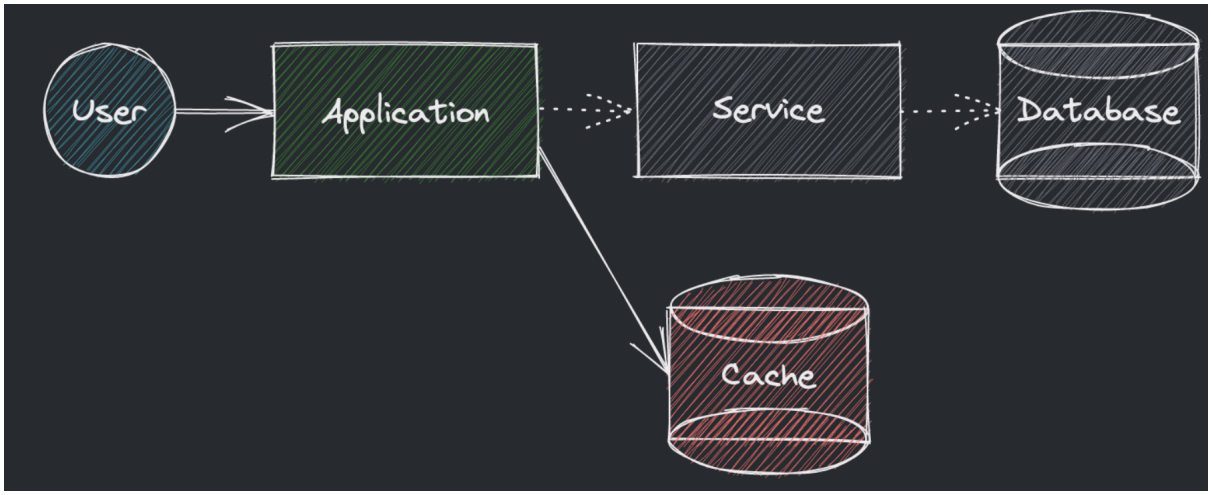
> Оглавление

> Кэширование данных

Самый лучший способ ускорить вычисления – не совершать их вообще. Для этого есть кэш. Он также позволяет ускорить обращение к данным, используя более быстрые хранилища.

Эффективность кэширования базируется на предположениях, что:

- Более вероятно, что новые обращения будут происходить к недавним данным
- Большая часть нагрузки приходится на малую часть запросов (принцип Парето 80 — 20)

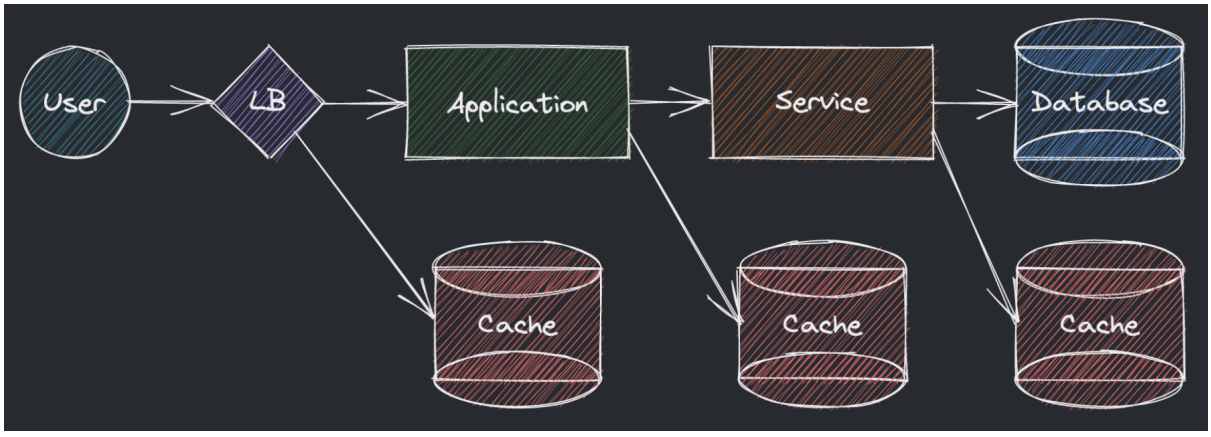


Вместо обращения к сервису можно получать быстрый ответ из кэша

Кэш может располагаться на практически любом из уровней:

- На запрос к фронтенду отдаем уже собранную страницу
- Для запроса на вычисления предоставляем сразу готовый ответ
- Для данных используем RAM вместо SSD, SSD вместо HDD и HDD вместо сети

Экономим большую часть железа ценой малого количества более дорогих машин. А отклик лучше!



Использование кэша на разных уровнях: балансировщика, приложения, сервиса

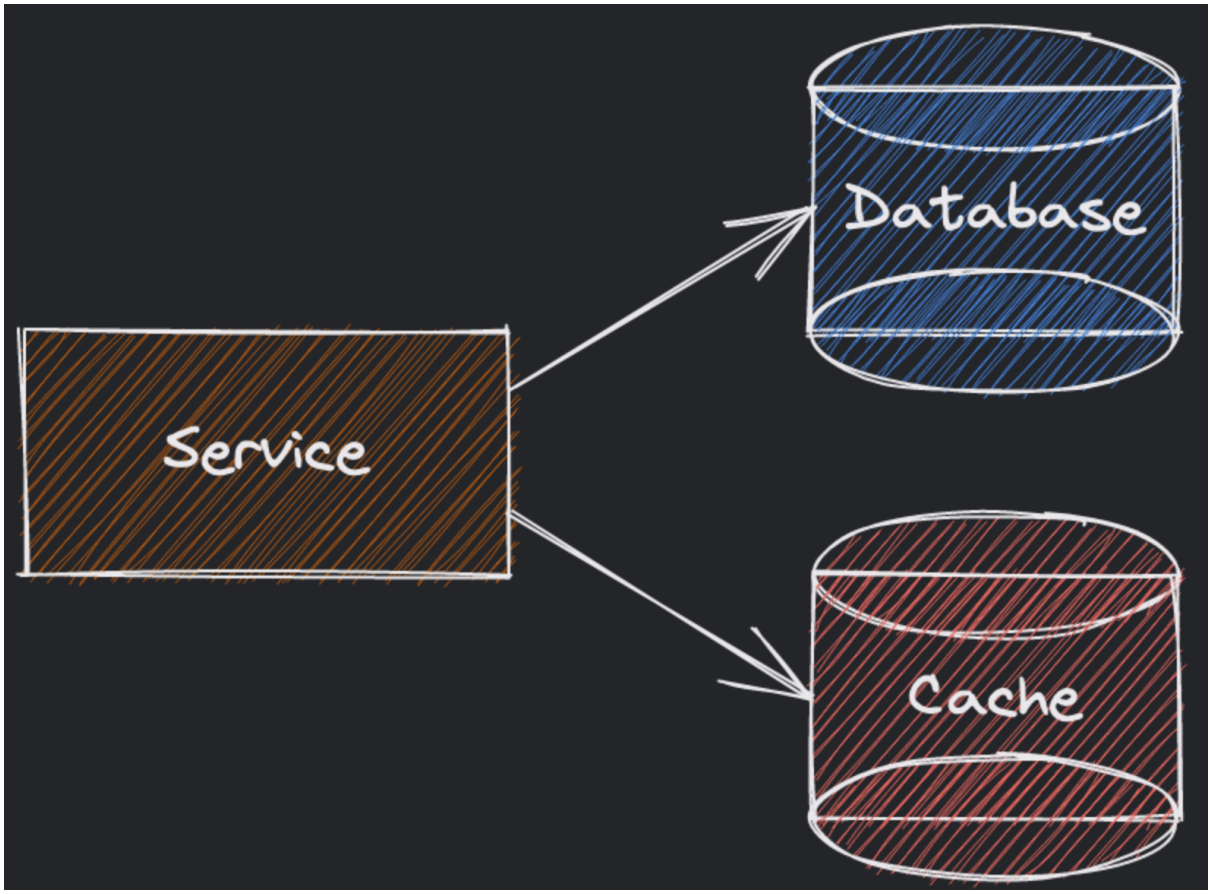
Инвалидация кэша

> There are only two hard things in Computer Science: cache invalidation and naming things – Phil Karlton

Быстрое получение данных это замечательно, но что с обновлением?

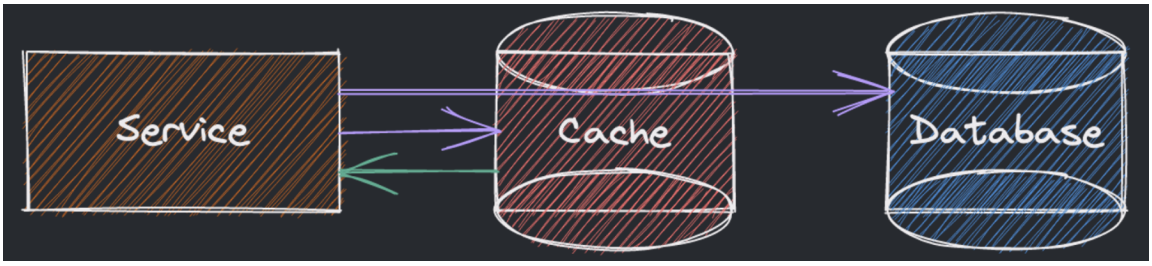
Рано или поздно данные в исходной БД изменятся и кэшированные данные придут в негодность.

В таком случае необходимо произвести инвалидацию кэша.

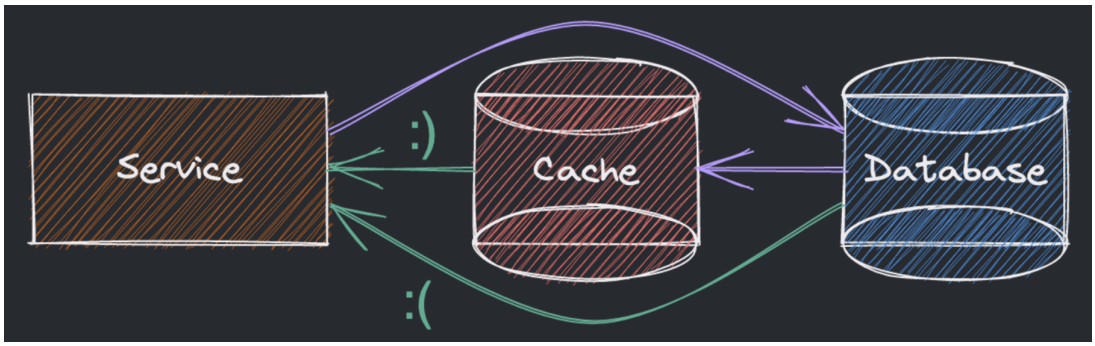


Способы инвалидации кэша:

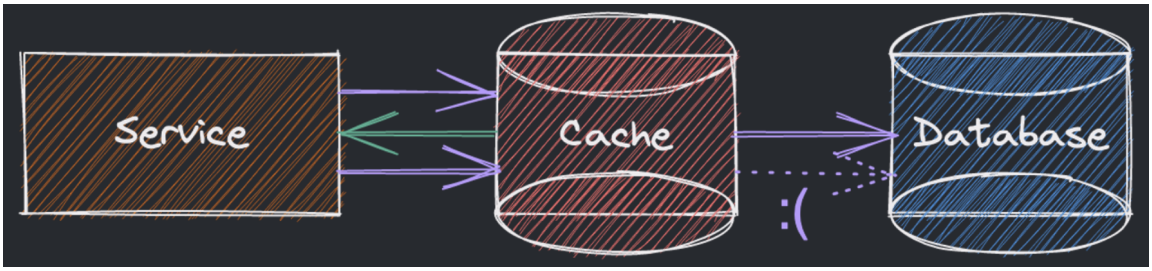
1. Сквозная запись: данные сразу записываются и в источник, и в кэш
    a. Плюсы: гарантированная консистентность, минимизация потерь
    b. Минусы: запись становится даже медленнее из-за двух операций



2. Запись в обход: данные записываются напрямую в источник
    a. Плюсы: не нагружаем кэш записью невостребованных данных
    b. Минусы: для недавних данных считывать нечего, идем в источник



3. Реверсивная запись: данные сначала записываются в кэш
    a. Плюсы: низкие задержки и высокая пропускная способность на запись
    b. Минусы: можно потерять недавние данные, не продублированные в источник



Вытеснение данных

По правилу 80-20 мы храним 20% данных в кэше, но как вытеснять место под актуальные данные?

- First In First Out (FIFO): удаляем самые давние записи несмотря на их популярность

- Last In First Out (LIFO): удаляем самые свежие записи несмотря на их популярность
- Least Recently Used (LRU): удаляем записи, не использованные дольше всего
- Most Recently Used (MRU): удаляем самые недавние из использованных записей
- Least Frequently Used (LFU): ведем счет обращений, удаляем самые непопулярные
- Random Replacement (RR): удаляем случайно выбранные записи.

Одним из оптимальных вариантов видится комбинация из принципов в основе LRU и LFU.

В некоторых системах вместо 20-80/80-20 может быть и 1-20-79/50-40-10 (пользователи-звезды).

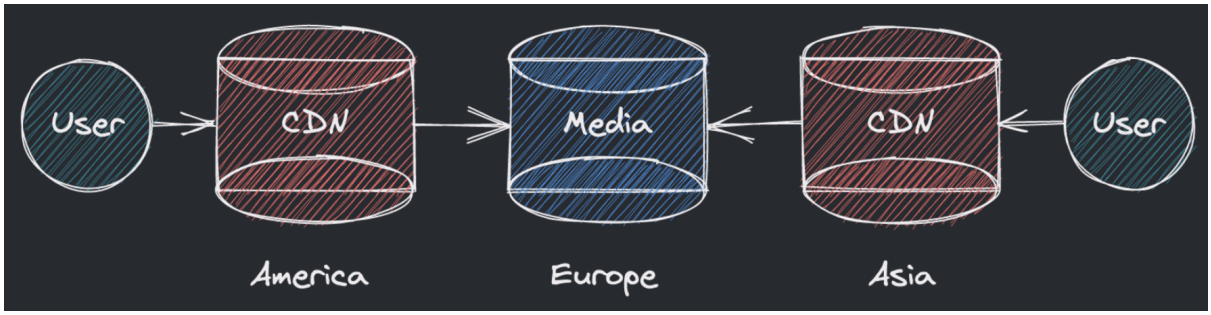
В таких случая стоит выделить пользовательские кластеры с еще более глубоким кешированием.

Content Delivery Network

Особым видом кэша можно считать Content Delivery Network (CDN) – это распределенные хранилища данных, которые используются в крупных системах, покрывающих различные регионы и даже весь мир.

Инстансы CDN распределены по местам присутствия сервиса и особо полезны в случаях использования часто запрашиваемых медиа файлов вроде изображений, звука или видео.

При запросе файла CDN отдаст его напрямую при наличии, иначе же обратится к центральным серверам.



Netflix Open Connect

Ранее мы разбирали пример с Netflix, где выяснили, что за трафик он бы не расплатился.

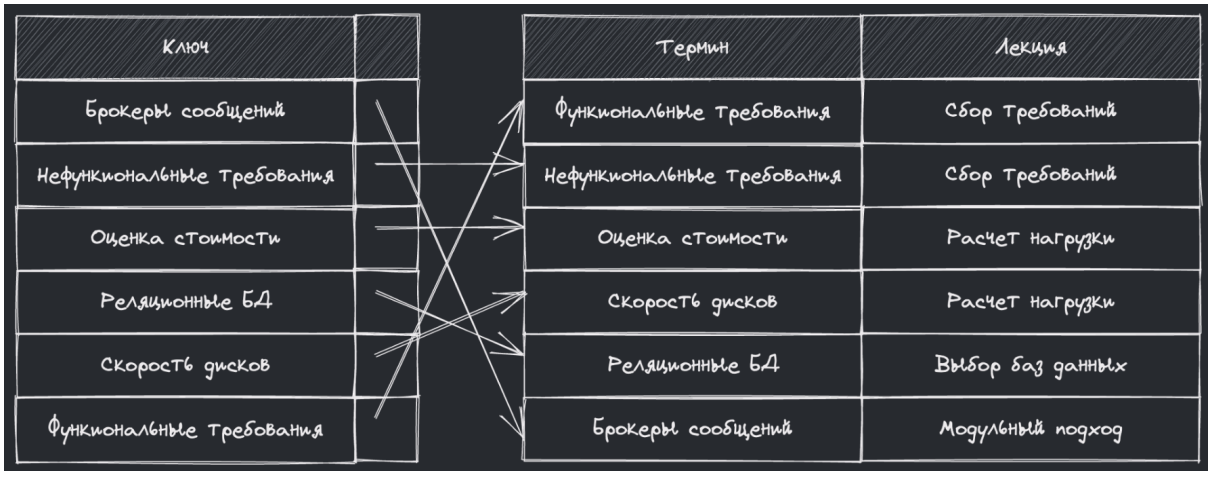
Но он за него и не платит! Netflix очень популярный сервис стриминга видео, и при передаче контента от Netflix к пользователю есть еще и посредники в виде провайдеров, которым это все передавать.

Чтобы Netflix не обанкротился, провайдеры не сошли с ума от трафика, а зрители получили лучший опыт, была предложена инициатива Open Connect – Netflix просто предоставляет свои сервера с данными провайдерам, те их устанавливают в точках обмена или присутствия, и все остаются в выигрыше.

> Индексирование баз данных

В случае неизбежного обращения к исходным данным в БД рано или поздно и это хочется ускорить.

На помощь приходит индексирование БД, которое ускоряет поиск нужных записей в таблице.



Получение отсортированного ключа и маппинг в место, где он реально находится

Индексирование осуществляется по одному или нескольким полям одной таблицы.

В результате повышается скорость поиска нужных записей на основе значений ключа в индексе.

Особенно полезно в случае большого количества маленьких записей или распределенной БД.

Плюсы:

- Увеличение скорости поиска записей по ключу в индексе (O(N) -> O(logN))
- Ускоренный выбор нужного шарда в случае распределенной БД

Минусы:

- При обновлении данных нужно обновить не только их, но и записи в индексе
- Как итог, уменьшается скорость записи, обновления и удаления записей в таблице

Когда применяем:

- Поиск по многочисленным или распределенным записям в БД становится бутылочным горлышком

Когда не применяем:

- Если операций на запись значительно больше операций на чтение, то индексирование только во вред

Лишние индексы не строим, а более ненужные не забываем удалять.

Создание ID записей

Еще один вопрос, который необходимо разрешить в случае распределенной системы, это генерация ID. В случае разделения таблиц по нескольким машинам простого автоинкремента на 1 уже не хватит. Посмотрим, какие есть варианты генерации и как это может быть связано с отзывчивостью.

Попробуем сгенерировать:

- Уникальные целочисленные (int64) ID
- Не обязательно увеличивающиеся с шагом 1

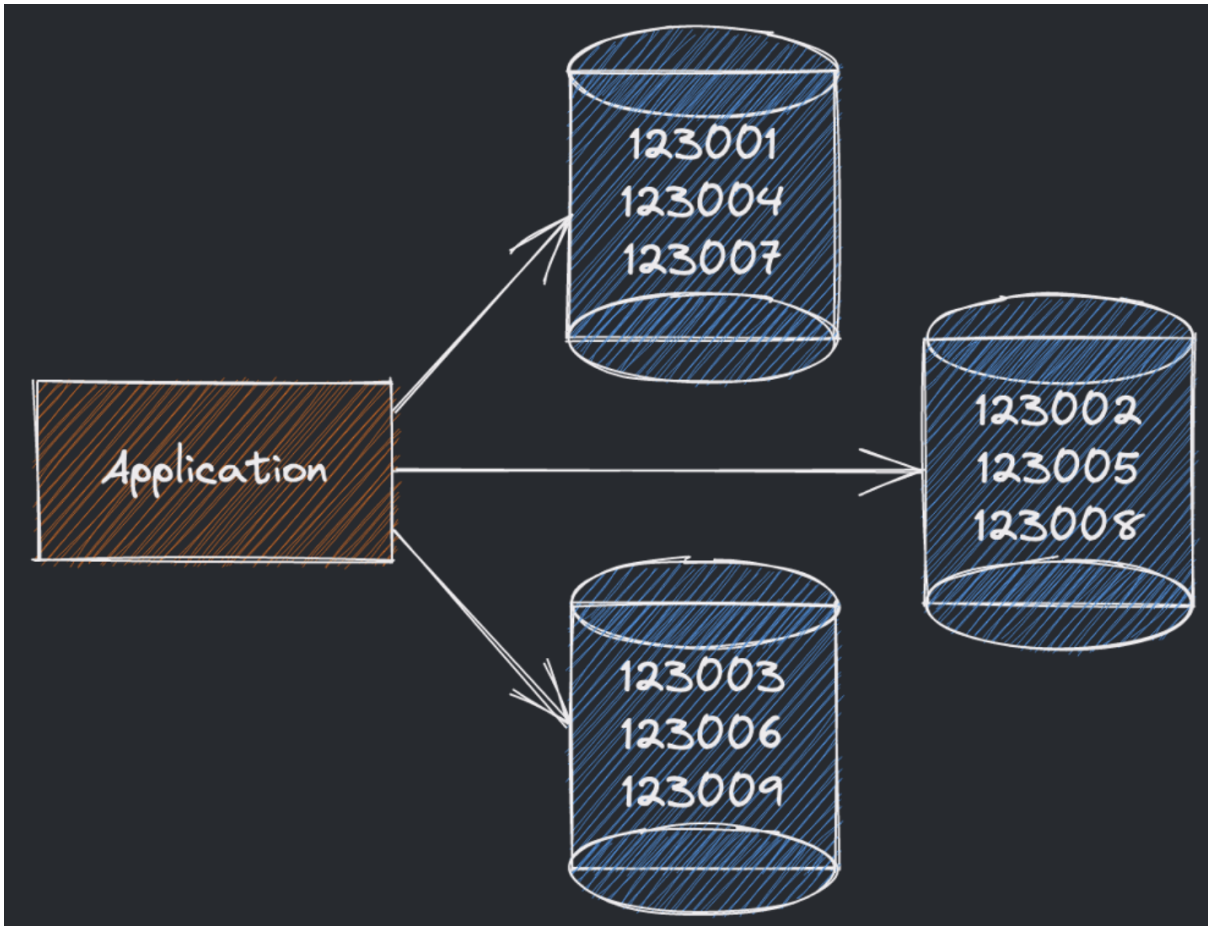
- В целом (по часам) идущие в порядке генерации
- Которых будет хватать на десятки тысяч генераций в секунду

Пример с ID фотографий

Самостоятельная генерация инстансом

Один их способов это создание ключей каждым основным инстансом БД самостоятельно. Мы все также инкрементируем идентификаторы, но с

заданным шагом больше 1. Плюс такого подхода в том, что он масштабируется под нужное количество серверов.



Из минусов этого подхода отметим:

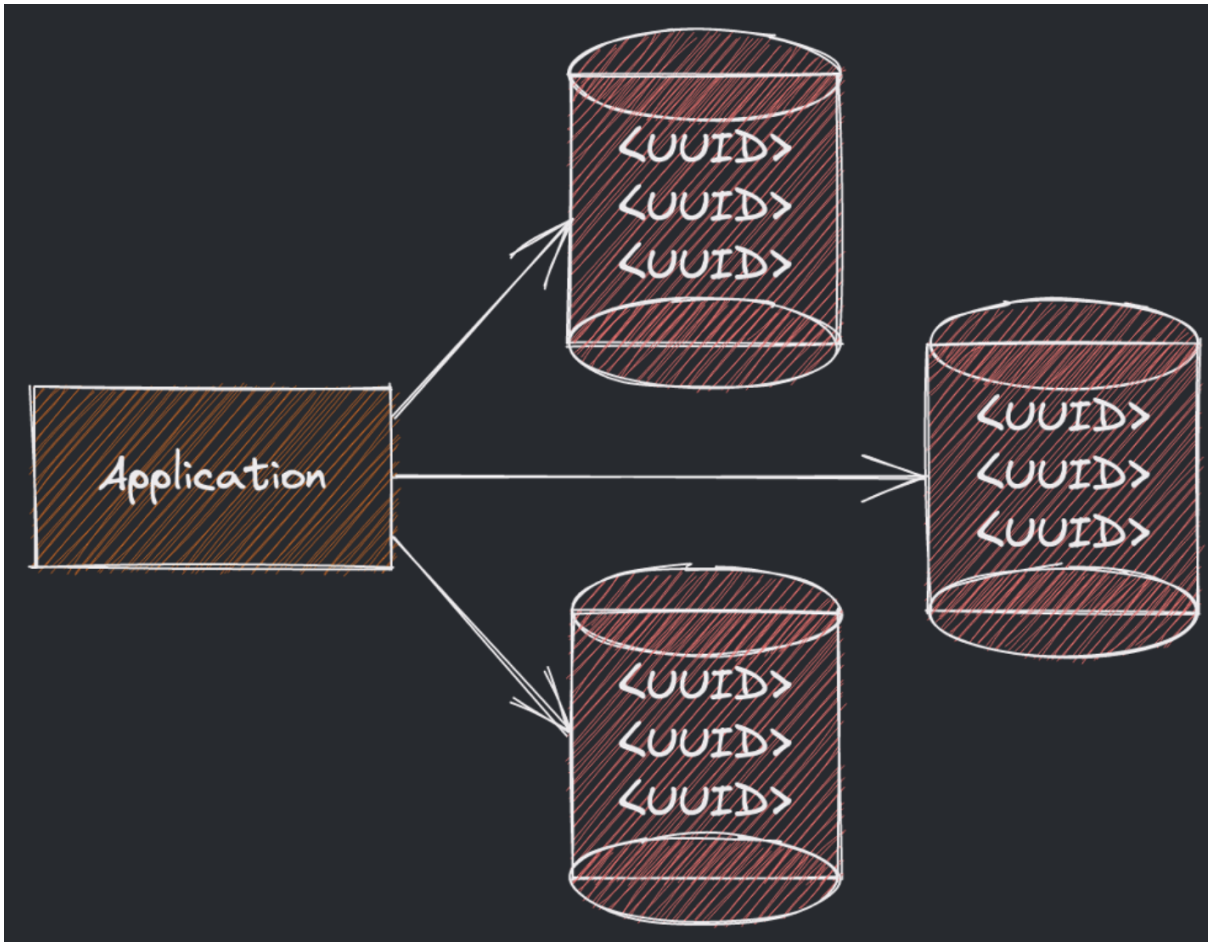
- Идентификаторы могут «разбежаться» со временем и стать не сравнимыми между серверами
- Масштабирование подхода статично и сложно построиться под изменение количества машин

UUID

Один из способов генерации ID, который не требует синхронизации и легко масштабируется это UUID.

Universally unique identified (UUID) — это универсальный 128-битный идентификатор сущностей, который по стандарту содержит в себе в числе прочего временную отметку и идентификатор устройства.

Пример UUID: 123e4567-e89b-12d3-a456-426614174000. Вероятность коллизий крайне мала!



Плюсы:

- Идентификаторы создаются независимо основными репликами базы данных
- Легко масштабировать, т.к. нет необходимости в координации между ними

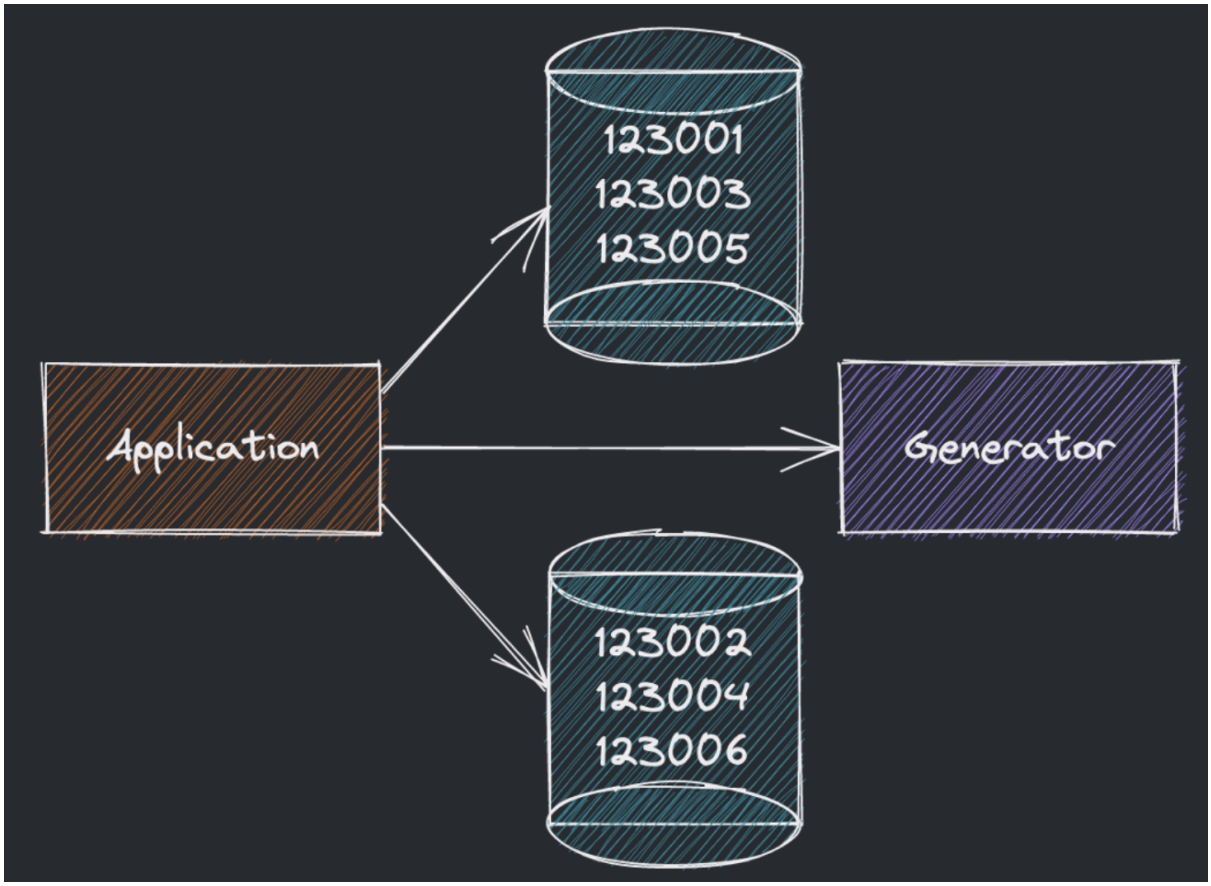
Минусы:

- 128 бит по стандарту больше 64-х бит
- Идентификаторы не линейны по времени
- Идентификаторы могут быть не целочисленными

Сервис-генератор

Еще одни подходом является использование отдельного сервиса-генератора ID с инкрементом.

При создании записей в БД приложение обращается к нему за получением нового идентификатора.



Плюсы:

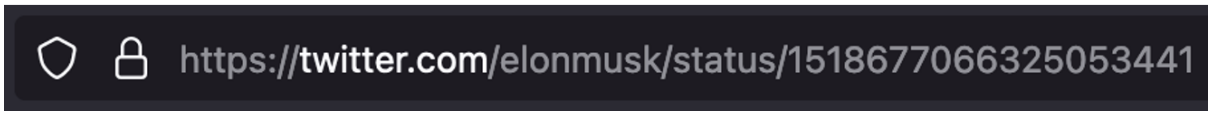
- Получаем числовые упорядоченные линейно упорядоченные по времени ID
- Подход легко масштабируется для систем не сильно большого объема

Минусы:

- Генератор является единой точкой отказа в системе, без него ничего не записать в систему
- Для добавления надежности нужно будет отдельно думать про избыточность и синхронизацию

Twitter

Пока что все рассмотренные подходы имели те или иные недостатки по модулю наших требований. Еще один пример генерации ID был предложен компанией Twitter:



В качестве идентификатора твита выступает 64-битное число, которое содержит в себе:

- (1) Зарезервированный бит
- (41) временную метку с момента начала работы Твиттера
- (5) идентификатор датацентра
- (5) идентификатор машины в датацентре
- (12) последовательный номер в рамках миллисекунды по модулю ДЦ/машины

Плюсы:

- Помещаемся в 64 бита
- Все ID генерируются независимо
- Все ID упорядочены с точностью до секунд (зависит от NTP сервера)
- Каждая машина в теории может генерировать несколько миллионов ID в секунду
- Упрощается маршрутизация при доступе к данным, тем самым в целом повышается отзывчивость

> Соединения и протоколы

Мы уже понижали вычислительную и сетевую нагрузку путем масштабирования системы.

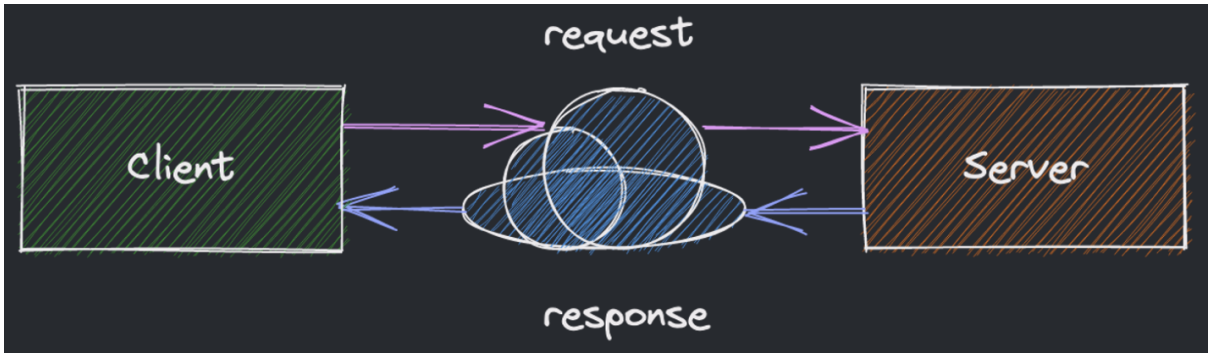
Также изучили другие способы повысить отзывчивость системы при доступе к данным.

Еще одним вопросом, которым мы задавались в рамках оценки нагрузки были соединения.

А как они вообще влияют на нагрузку и отзывчивость системы? Можно ли что-нибудь «подкрутить» и там?

Типичное взаимодействие между клиентским приложением и сервером по HTTP выглядит как:

- Клиент открывает соединение с сервером
- Клиент отправляет запрос на сервер
- Сервер подготавливает ответ
- Сервер отправляет ответ по открытому соединению

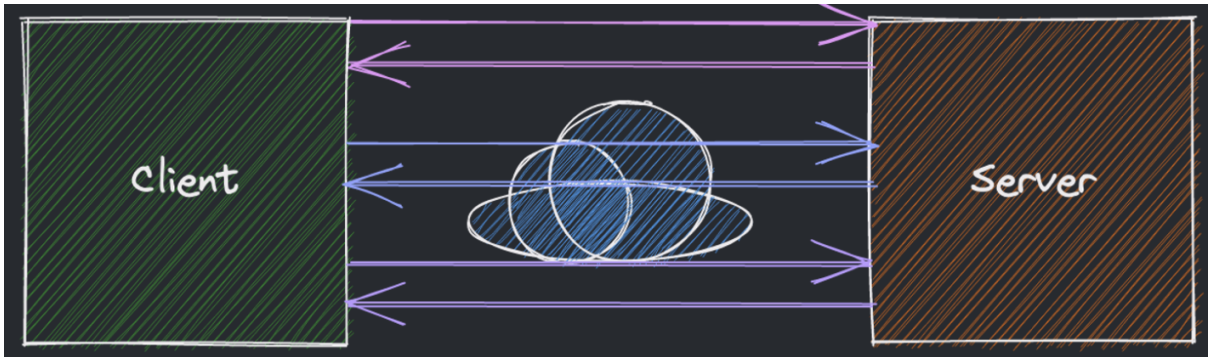


AJAX Polling

При необходимости получить дополнительные данные на странице можно воспользоваться AJAX.

AJAX Polling — это подход, при котором клиент постоянно опрашивает сервер об изменениях.

При отсутствии изменений сервер возвращает пустой ответ.



Взаимодействие между клиентом и серверов при AJAX Polling устроено следующим образом:

- Клиент открывает соединение и запрашивает страницу по HTTP
- Уже со страницы затем поступают запросы с интервалом
- Сервер подготавливает ответ на каждый такой запрос

- Клиент повторяет данный сценарий взаимодействия с другой страницей

Минусы подхода AJAX Polling для обновления данных на клиенте:

- Необходимо постоянно опрашивать сервер вне зависимости от наличия обновлений
- Большая часть ответов приходит пустая, забивая при этом HTTP канал

HTTP Long-Polling

Альтернативой AJAX Polling является HTTP Long-Polling.

В отличие от AJAX Polling здесь уже сервер управляет отсылкой данных.

Клиент же создает «висящие» запросы, дожидается ответа и сразу же посылает новый запрос.

Взаимодействие между клиентом и серверов при HTTP Long-Polling устроено следующим образом:

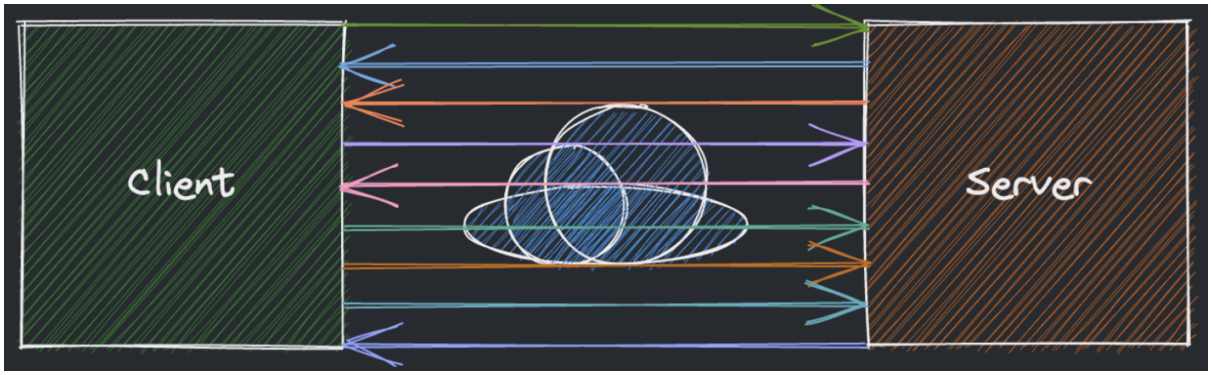
- Клиент открывает соединение и запрашивает страницу по HTTP
- Уже со страницы затем в свою очередь поступают запросы к серверу
- При наличии обновлений сервер посылает ответ на запрос клиенту
- Клиент затем почти сразу посылает новый такой же запрос
- У каждого установленного соединения есть таймаут, чтобы они не висели бесконечно

Веб-сокеты

Еще одним способом взаимодействия между клиентов и сервером является использования веб-сокетов:

- Соединение устанавливается клиентом через специальную процедуру (handshake)
- Веб-сокеты предоставляет долгоиграющее соединение между клиентом и сервером
- И клиент, и сервер могут обмениваться данными в произвольном порядке
- Коммуникация обеспечивается с минимальными издержками

Фактически веб-сокеты позволяют установить двусторонний диалог между клиентом и сервером



Server-Sent Events

Способ взаимодействия через Server-Sent Events (SSEs) похож в чем-то и на веб-сокеты, и на Long-Polling.

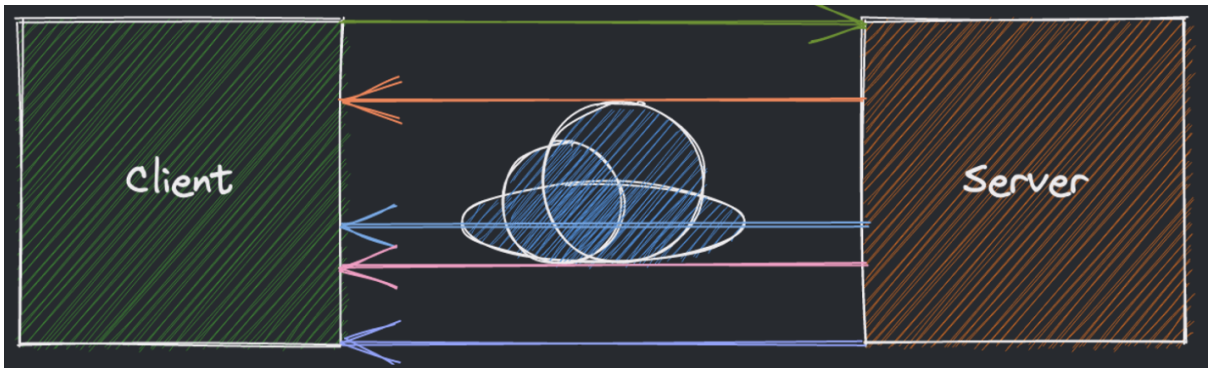
После первичного получения данных клиент подписывается на обновления с сервера.

Если же клиенту данные нужно наоборот передать, он может воспользоваться другим протоколом.

Взаимодействие между клиентом и серверов через SSEs устроено следующим образом:

- Клиент открывает соединение и запрашивает страницу по HTTP
- Открытая страница открывает соединение с сервером
- Сервер посылает данные клиенту в случае наличия обновлений

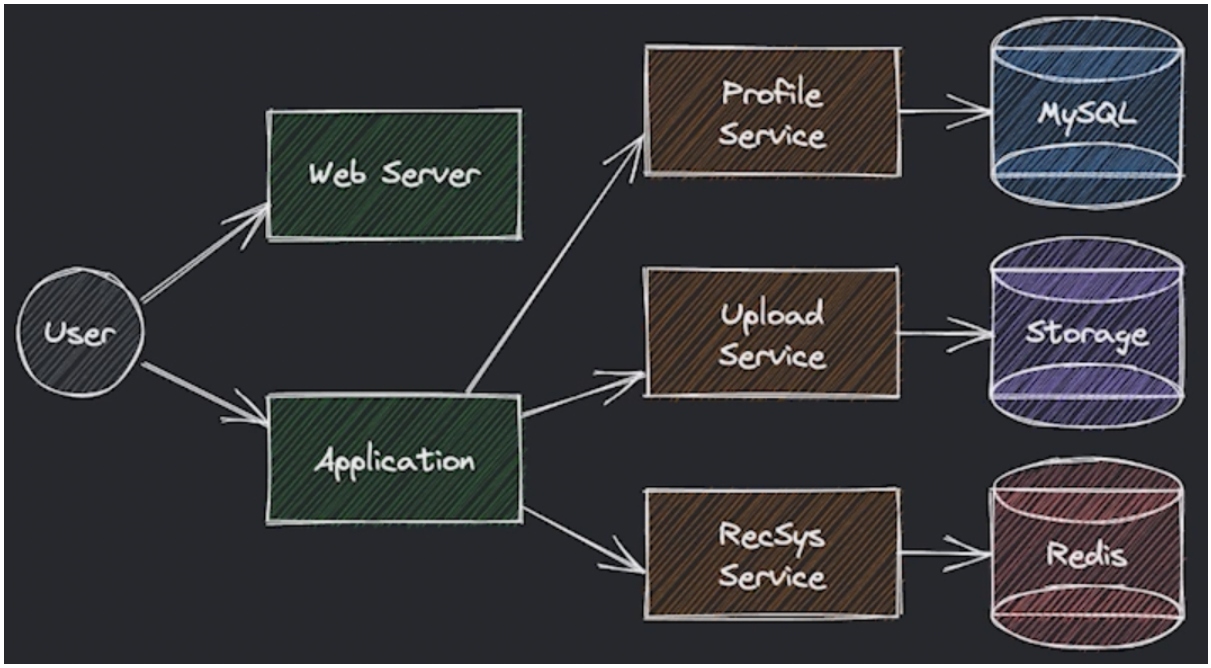
Этот способ подходит для систем, где сервер присылает клиенту обновления в реальном времени.



> Сферический сервис в вакууме

Представим, что у нас есть простой сервис, который нужно подготовить к масштабированию.

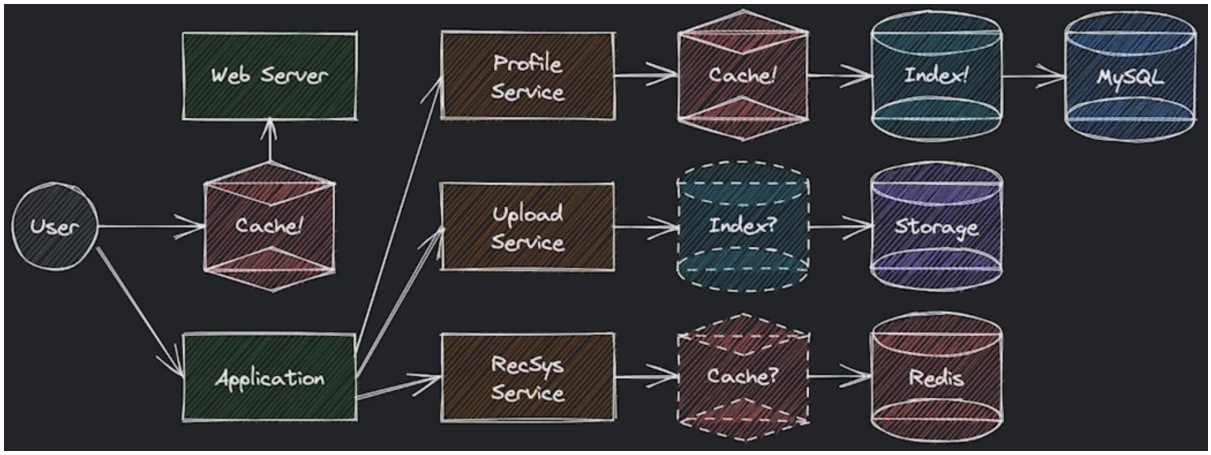
У сервиса есть пользователи, есть веб-сервер, у которого можно запрашивать веб-страницы, эти веб-страницы, как приложения, могут запрашивать данные пользователей, обновлять их и загружать контент. Также приложения взаимодействуют с сервисом рекомендаций, который общается с быстрым хранилищем



Внесем необходимые правки, чтобы сервис отвечал быстрее. Установим между пользователем и сервером кэш, чтобы каждый раз заново не запрашивать страницы. Когда запрашиваем профили, можем обращаться к кэшу, где, например, хранятся недавние профили. Если есть необходимость обратиться к исходной базе, можем ее проиндексировать.

У upload сервиса индекс стоит под знаком вопроса, так как, конечно, он может понять, в какой шард стоит загрузить файл. При этом он замедляет обращение к таблице с метаданными, так как там обычно чтение происходит чаще, чем запись.

У сервиса рекомендаций кэш также со знаком вопроса. Redis и так довольно быстрое хранилище, поэтому кэш не сильно поможет.



> Выводы

- Для повышения отзывчивости помимо масштабирования можно кэшировать запросы и данные
- Упростить доступ к данным в крупных гео-распределенных системах может применение CDN
- Неизбежные обращения к исходным данным в БД можно ускорить за счет построения индекса
- Балансировку и маршрутизацию запросов можно облегчить при правильном дизайне ID в БД
- В случае постоянного обмена между клиентом и сервером есть альтернативы частым запросам
- Чтобы оставаться на нужном уровне отзывчивости, можно применить ограничитель запросов.