Оптимизация WEB приложений

Лекции

Науменко С.С.

Цели и задачи оптимизации

С каждым годом Интернет растет вширь и вглубь. Увеличивается пропускная способность каналов, пользователи переходят с коммутируемого доступа на безлимитный. Сайты становятся больше по размеру, больше по наполнению и сложнее во взаимодействии. Размеры загружаемых файлов при этом увеличиваются многократно, а время ожидания пользователей не уменьшается.

За последние 5 лет средний размер веб-страниц вырос втрое (по данным исследования Akamai), а за последний год — в полтора раза (по данным webo.in). При этом каждая страница использует в среднем по 60 объектов, что крайне негативно сказывается на общем времени загрузки. Только порядка 5-10% от общего времени загрузки приходится на серверную часть. Все остальное составляет именно клиентская архитектура.

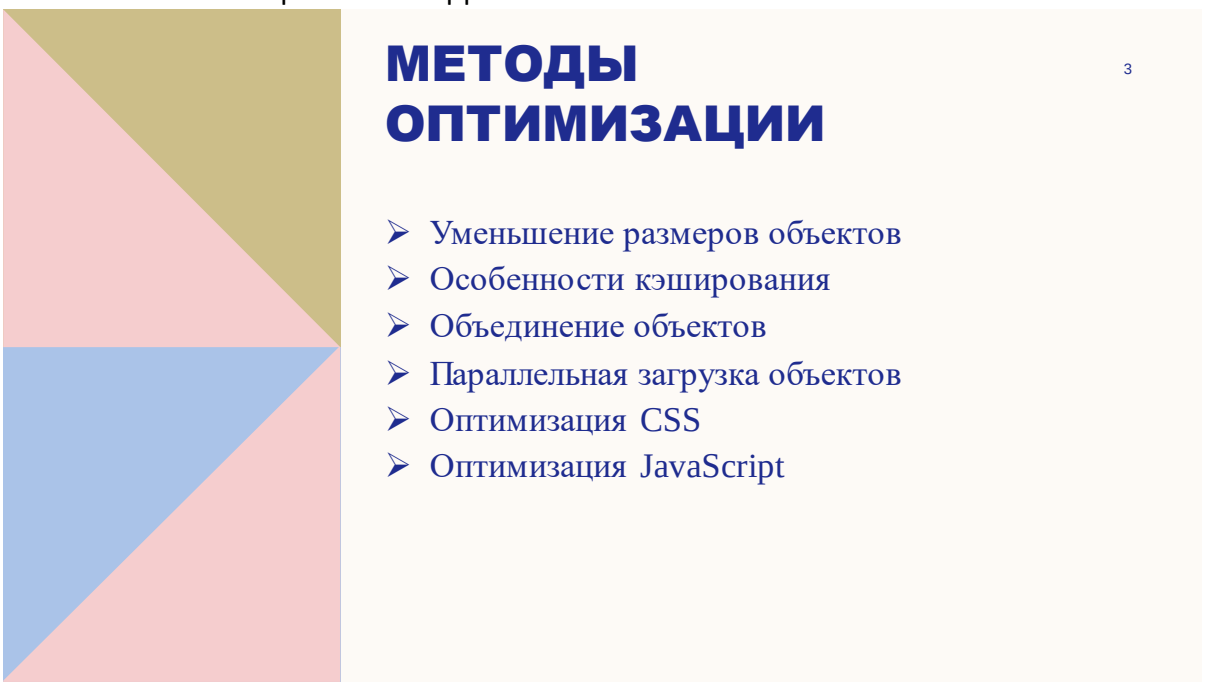
В настоящее время на каждой странице вызывается несколько десятков внешних объектов, а размер исходного файла составляет не более 5% от общего размера. Если учитывать расслоение пользователей по скоростям доступа, то стремление уменьшить число пользователей, превышающих допустимый порог ожидания на 1-5%, заставляет применять все более сложные и передовые технологии.

Естественно, что технологии эти не ограничиваются сжатием текстовых (HTML, CSS, JavaScript) файлов на стороне сервера. Как несложно понять, основную часть внешних объектов на странице составляют изображения или мультимедийные файлы. И для них тоже есть свои методы оптимизации.

Основные задачи оптимизации

Если говорить кратко, то можно выделить 3 основных задачи клиентской оптимизации:

1. Оптимизация размера файлов.
2. Оптимизация задержек при загрузке.
3. Оптимизация взаимодействия с пользователем.



Краткий обзор технологий

Все основные методы оптимизации можно разбить на 6 групп (каждая из которых позволяет решить одну из заявленных задач).

- Уменьшение размеров объектов. Здесь фигурируют сжатие и методы оптимизации изображений.
- Особенности кэширования, которые способны кардинально уменьшить число запросов при повторных посещениях.

- Объединение объектов. Основными технологиями являются слияние текстовых файлов, применение CSS Sprites или data:URI для изображений.
- Параллельная загрузка объектов, влияющая на эффективное время ожидания каждого файла. В пятой главе, помимо этого, приведены примеры балансировки запросов со стороны клиентского приложения.
- Оптимизация CSS-производительности, что проявляется в скорости появления первоначальной картинки в браузере пользователя и скорости ее дальнейшего изменения.
- Оптимизация JavaScript. Есть достаточно много проблемных мест в JavaScript, о которых необходимо знать при проектировании сложных веб-приложений.

Психологические аспекты производительности

Согласно многочисленным исследованиям, пользовательское раздражение сильно возрастает, если скорость загрузки страницы превышает 8-10 секунд безо всякого уведомления пользователя о процессе загрузки. Последние работы в этой области показали, что пользователи с широкополосным доступом еще менее терпимы к задержкам при загрузке веб-страниц по сравнению с пользователями с более узким каналом.

Терпимое время ожидания

При проведенных исследованиях было установлено, что терпимое время ожидания для неработающих ссылок (без обратной связи) находилось между 5 и 8 секундами. С добавлением уведомления пользователя о процессе загрузки (обратной связи), например, индикатора загрузки, такое время ожидания увеличилось до 38 секунд. Распределение времени для повторных попыток зайти на неработающие ссылки имело максимум в районе 23 секунд (без наличия каких-либо индикаторов, показывающих, что страница загружается или в данный момент недоступна).

Таким образом, можно заключить, что для 95% пользователей время ожидания ответа от неработающего сайта составит не более 8 секунд. Если учесть стремление пользователя посетить сайт повторно, то исследования продемонстрировали крайне малое (почти равное нулю) число пользователей, ждущих более 10 секунд.

Как время ответа сайта влияет на пользовательскую психологию

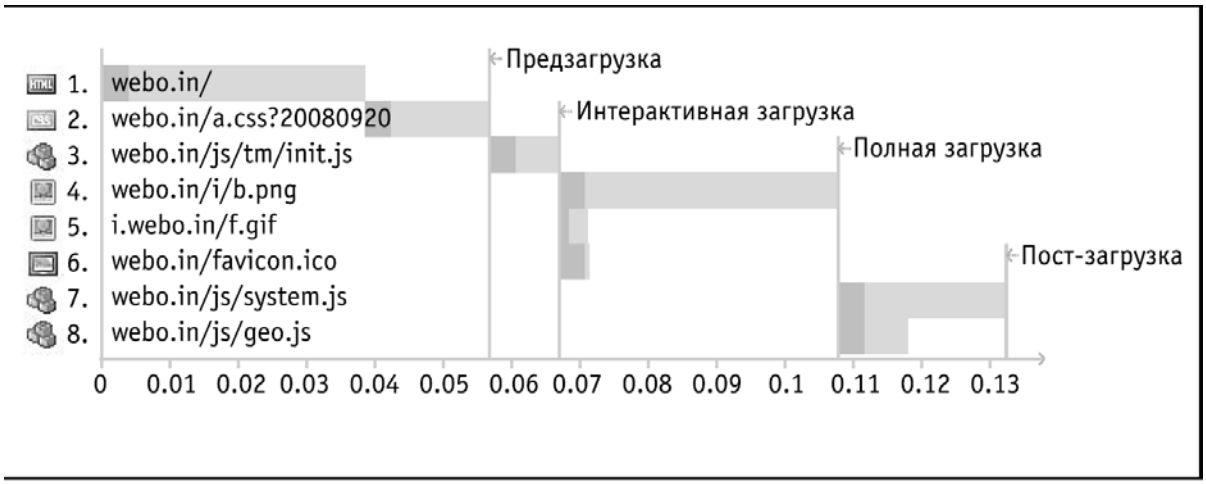
Пользователи ощущают, что сайты, которые загружаются медленно, менее надежны и менее качественны. В случае, если удерживать время загрузки в пределах «терпимого», пользователи будут ощущать гораздо меньше неудовлетворенности от посещения сайта, среднее число просмотров

страниц возрастет, увеличится конверсия посетителей и снизится число отказов. Сайты, которые быстро загружаются, также кажутся пользователям более интересными и привлекательными.

«Терпимое» время будет сильно зависеть от аудитории, но его можно достаточно надежно проверить: для этого нужно значительно (например, в 2-3 раза) увеличить (или уменьшить) время задержки при показе страницы и посмотреть на число отказов (число пользователей, закрывших страницу сразу после захода на сайт) и на число постоянных посетителей. Если при сильном увеличении (или уменьшении) задержки при загрузке сайта количество пользователей практически не изменилось, значит, страница уже загружается в допустимом диапазоне. Если же число пользователей претерпело видимые изменения, то, следовательно, со временем загрузки сайта нужно что-то делать.

Пользователи широкополосного доступа ожидают большей скорости загрузки, при этом пользователям с менее скоростным доступом приходится ждать гораздо дольше. По мере того, как высокоскоростной доступ проникает в массы, растет также и размер страницы. Пользователи испытывают психологические и физиологические проблемы при взаимодействии с «медленными» веб-страницами, чувствуют раздражение, если не могут завершить свои задачи, и воодушевление при работе с быстрыми сайтами.

Стадии загрузки страницы



В качестве основных проблемных мест при загрузке страницы любого веб-ресурса можно выделить четыре ключевых момента.

1. Предзагрузка — появление страницы в браузере пользователя. После некоторого времени ожидания загрузки при заходе на веб-ресурс у пользователя в браузере отображается нарисованная страница. В этот момент, вероятно, на странице отсутствуют рисунки и, скорее всего, не полностью функционирует JavaScript- логика.

2. Интерактивная загрузка — появление интерактивности (и анимации) у загруженной веб-страницы. Обычно вся клиентская логика

взаимодействия доступна сразу после первоначальной загрузки страницы (стадия 1), однако в некоторых случаях (о них речь пойдет чуть дальше) поддержка этой логики может (и должно, на самом деле) немного запаздывать по времени от появления основной картинки в браузере пользователя.

3. Полная загрузка страницы. Страница полностью появилась в браузере, на ней представлена вся заявленная информация, и она практически готова к дальнейшим действиям пользователя.

4. Пост-загрузка страницы. На данной стадии полностью загруженная страница может (в невидимом для пользователя режиме) осуществлять загрузку и кэширование некоторых ресурсов или компонентов. Они могут потребоваться пользователю как при переходе на другие страницы данного сайта, так и для отображения каких-либо анимационных эффектов или добавления функционала ради удобства использования.

Для большинства сайтов на данный момент стоит различать только предзагрузку (в которую по умолчанию включается интерактивная загрузка) и полную загрузку страницы. Пост-загрузка, к несчастью, сейчас используется крайне мало.

Оптимизация скорости загрузки веб-страницы сосредоточена на двух ключевых аспектах: ускорение предзагрузки и ускорение основной загрузки. Все основные методы сфокусированы именно на этом, потому что «загрузка» веб-страницы воспринимается пользователями как нечто находящееся посредине этих двух стадий.

Расставляем приоритеты

Оптимизация скорости загрузки веб-страницы сосредоточена на двух ключевых аспектах: ускорение предзагрузки и ускорение основной загрузки. Все основные методы сфокусированы именно на этом, потому что «загрузка» веб-страницы воспринимается пользователями как нечто находящееся посредине этих двух стадий.

В идеале загрузка страницы для пользователя должна заканчиваться сразу после предзагрузки, однако добиться этого весьма сложно, и оправдано это далеко не во всех случаях.

Узкие места

Первая и вторая стадии загрузки являются наиболее проблемными аспектами при анализе производительности. Это вполне понятно: загрузка первоначального HTML-файла, равно как и CSS JavaScript-файлов идет в один поток, — и на первое место выходит уменьшение числа запросов при загрузке.

Как только узкое место преодолено (в идеале, у нас должен быть один-единственный файл, который получает пользователь) и в браузере страница отобразилась, мы можем начать запрашивать с сервера все остальные ресурсы. Самое главное, что это можно делать с помощью десятков дополнительных соединений (как этого добиться, рассказывается в пятой

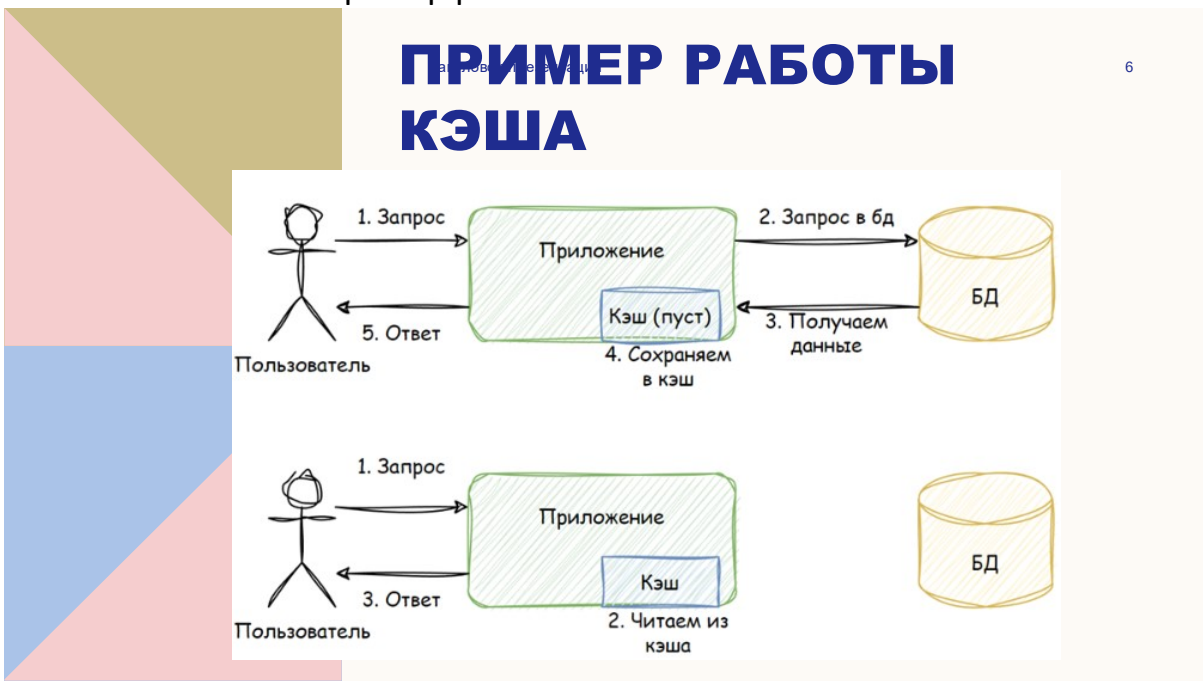
главе), ибо в браузере уже произошло событие готовности документа к дальнейшим действиям.

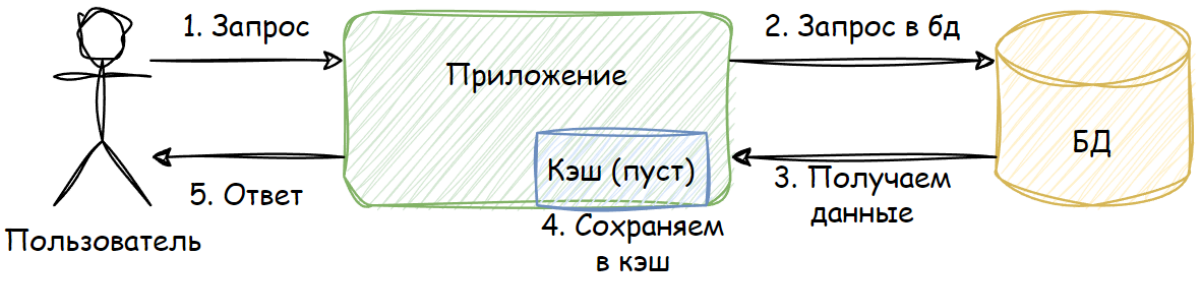
Мы можем настроить логику кэширования, последовательную загрузку JavaScript-модулей или даже пост-загрузку стилевых правил. Все это уже будет слабо отражаться на фактической скорости первоначальной загрузки: пользователь видит страницу в браузере, может с ней взаимодействовать (пусть даже сначала и не в полном объеме), для него она уже загрузилась (правда, только с психологической, а не с технической стороны).

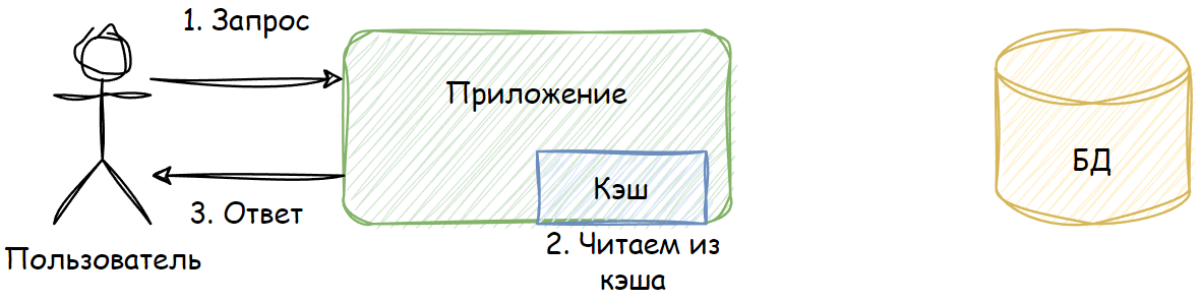
Как работает Кэширование

Логически кэш представляет из себя базу типа ключ-значение. Каждая запись в кэше имеет “время жизни”, по истечении которого она удаляется. Это время называют термином Time To Live или TTL. Размер кэша гораздо меньше, чем у основного хранилища, но этот недостаток компенсируется высокой скоростью доступа к данным. Это достигается за счет размещения кэша в быстродействующей памяти ОЗУ (RAM). Поэтому обычно кэш содержит самые “горячие” данные.

Вот самый базовый пример работы кэша:







Пример работы кэша

На этой схеме изображено первое обращение за данными:

1. Пользователь запрашивает некие данные
2. Кэш приложения ПУСТ, поэтому приложение обращается к базе данных (БД)
3. БД возвращает запрошенные данные приложению
4. Приложение сохраняет полученные данные в кэше
5. Пользователь получает данные

Вторая схема иллюстрирует последующие обращения за данными:

1. Пользователь запрашивает данные
2. Приложение уже имеет эти данные в кэше (ведь они были записаны туда при первом обращении) и поэтому НЕ ОБРАЩАЕТСЯ за ними к БД
3. Пользователь получает данные

Из этого простого примера видно, что только первый запрос данных приводит к обращению к БД. Все последующие запросы попадают в кэш до тех

пор, пока не истечет TTL. Как только это произойдет, новый запрос снова обращается к БД и заново кэширует данные. Так будет продолжаться все время, пока приложение работает.

В результате мы экономим время не только на обработке запросов в БД, но и на сетевом обмене с БД. Все это значительно ускоряет время получения ответа пользователем.

Метрики кэша

Работу кэша можно оценивать при помощи множества метрик разной степени полезности. Я опишу те, которые считаю базовыми и наиболее полезными.

- Объем памяти, выделенной под кэш. Это базовый показатель, по которому можно судить, сколько используется ресурсов

- RPS чтения/записи – количество операций чтения/записи за единицу времени. В обычной ситуации количество операций чтения должно быть в разы больше количества операций записи. Обратное соотношение свидетельствует о проблемах в работе кэша

- Количество элементов в кэше. Его полезно знать в дополнение к объему памяти, чтобы обнаруживать большие записи

- Hit rate – процент извлечения данных из кэша. Чем он ближе к 100%, тем лучше. Этот параметр буквально определяет то, насколько наш кэш полезен и эффективен

- Expired rate – процент удаления записей по истечении TTL. Этот показатель помогает обнаружить проблемы с производительностью, вызванные большим количеством записей с одновременно истекшим TTL

- Eviction rate – процент вытеснения записей из кэша при достижении лимита используемой памяти. Важный показатель при выборе стратегий вытеснения, о которых мы поговорим чуть позже

Что можно кэшировать?

Строго говоря, кэшировать можно что угодно, но не всегда это целесообразно. Все сильно зависит от данных и паттерна их использования.

Все данные можно условно разделить на 3 группы по частоте изменений:

Меняются часто. Такие данные изменяются в течение секунд или нескольких минут. Их кэширование чаще всего бессмысленно, хотя иногда может и пригодится.

Пример: ошибки (кэширование ошибок может быть настолько важным, что мы посвятили ему целую главу ближе к концу статьи)

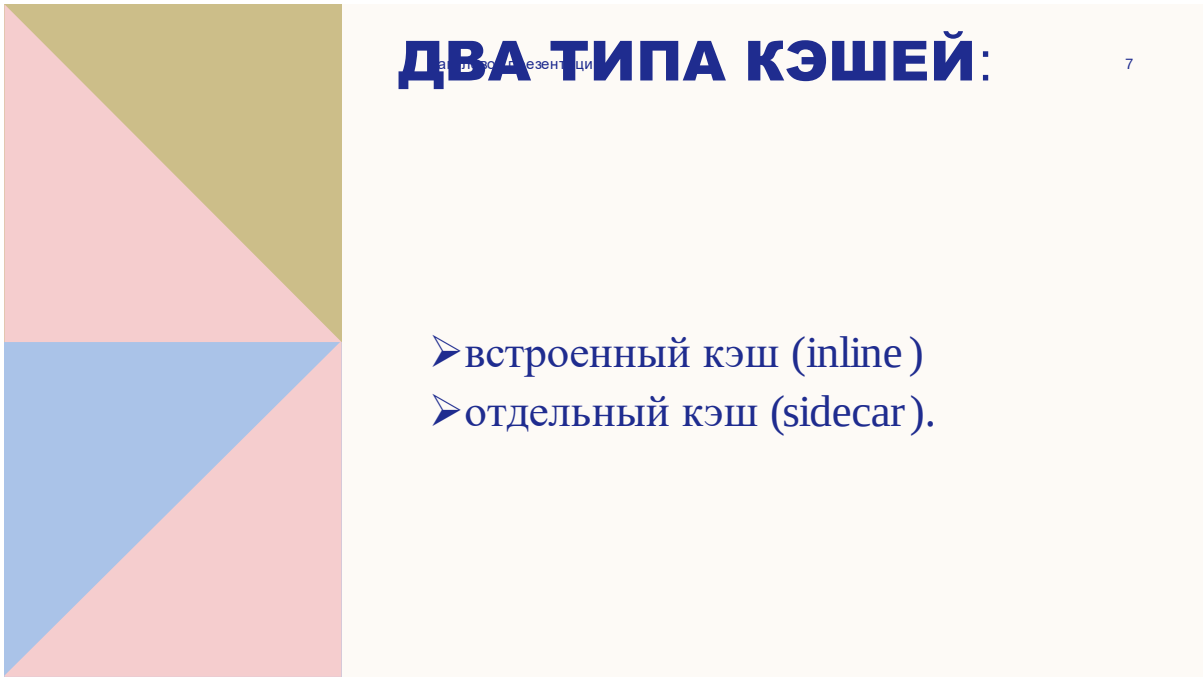
Меняются нечасто. Такие данные изменяются в течение минут, часов, дней. Именно в этом случае вы чаще всего задаетесь вопросом “Стоит ли мне кэшировать это?”

Примеры: списки товаров на сайте, описания товаров

Меняются крайне редко или не меняются никогда. Такие данные меняются в течение недель, месяцев и лет. В этом случае данные можно спокойно кэшировать. НО! Ни в коем случае нельзя усыплять бдительность верой в то, что какие-либо данные никогда не изменятся. Рано или поздно они изменятся, поэтому всегда выставляйте всем данным разумный TTL. ВСЕГДА!

Пример: картинки, DNS

Типы кэшей

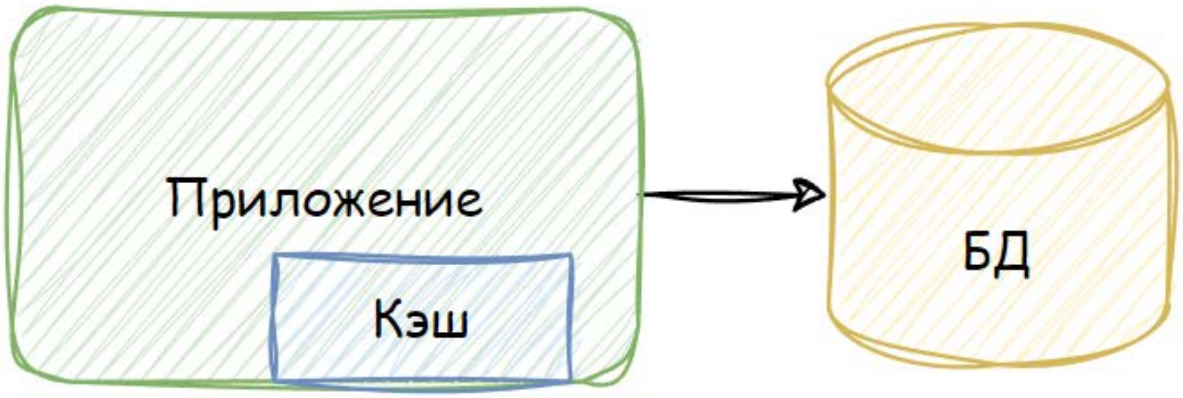


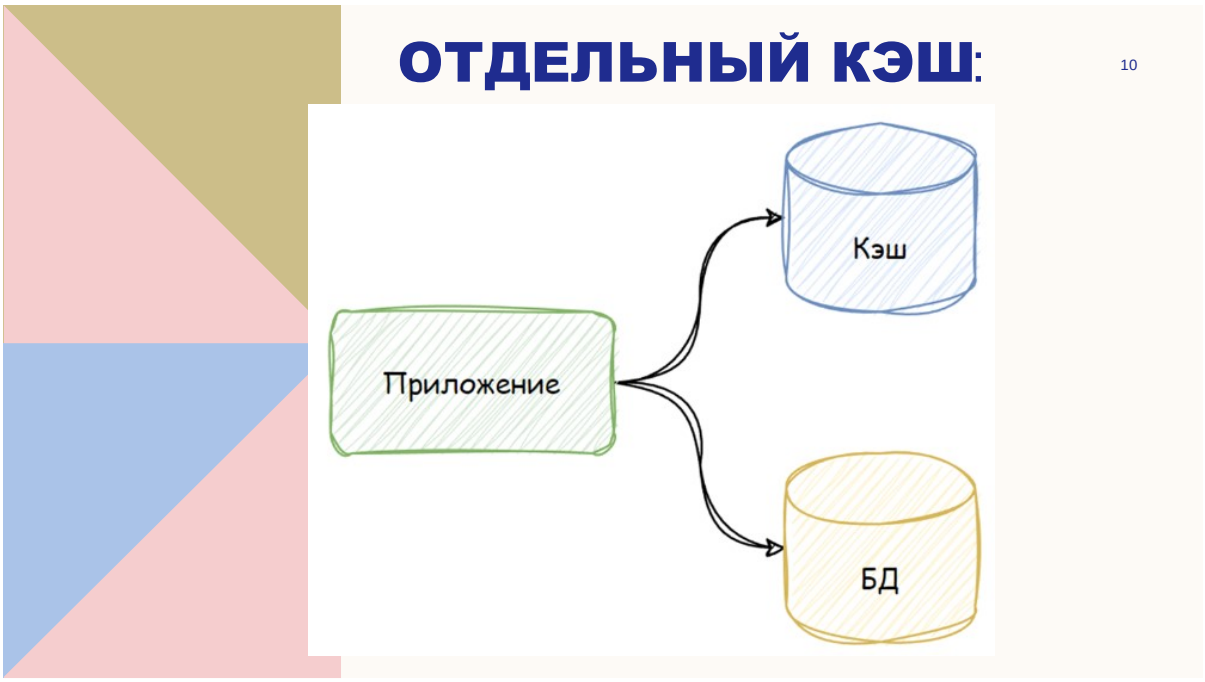
С точки зрения архитектуры, можно выделить два типа кэшей: встроенный кэш (inline) и отдельный кэш (sidecar).

Встроенный кэш – это кэш, который “живет” в том же процессе, что и основное приложение. Они делят один сегмент памяти, поэтому за данными можно обращаться напрямую. В качестве такого кэша может выступать map – обычный инструмент из арсенала go – или другие структуры данных

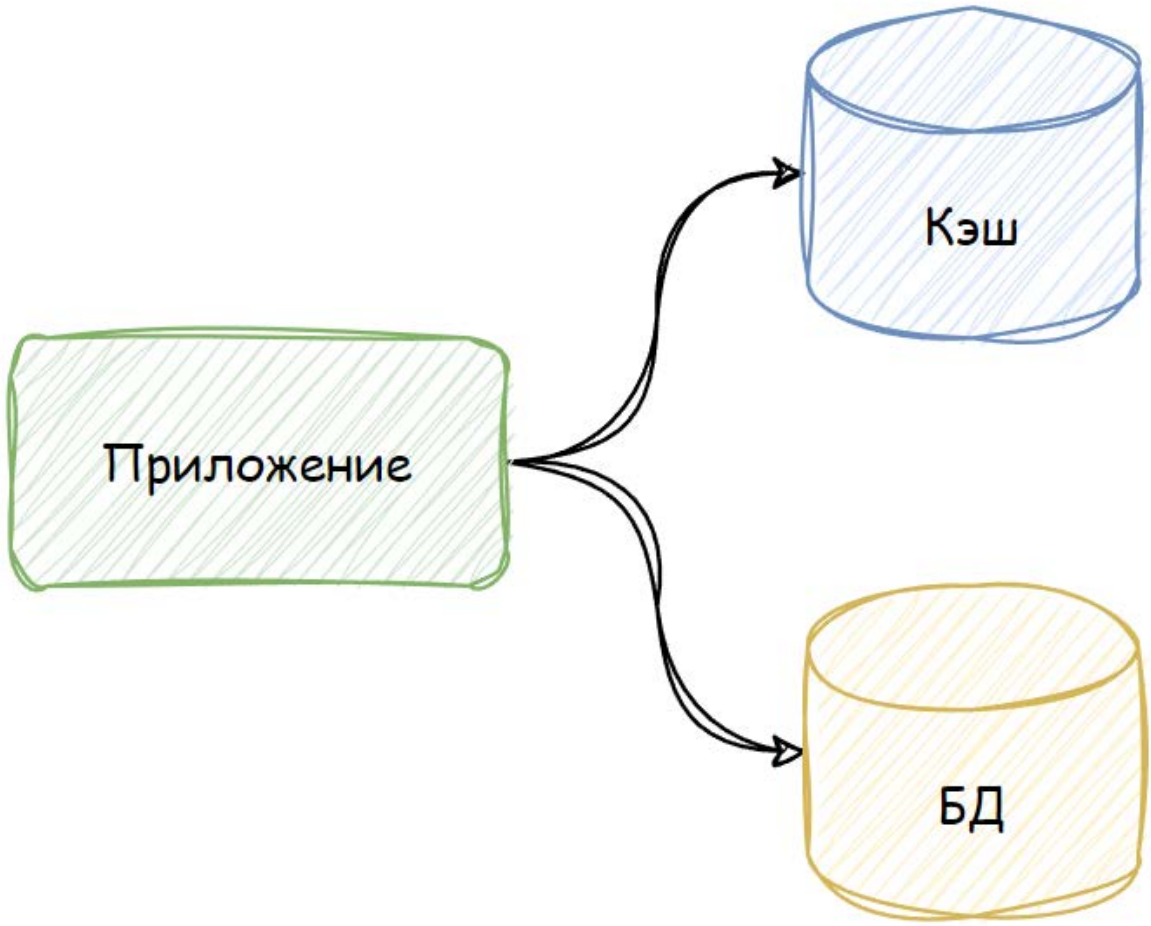
(подробнее об этом можно почитать здесь: https://github.com/hashicorp/golang-lru).

Вот принципиальная схема работы inline cache:





Отдельные кэши (sidecar) – это обособленный процесс со своей выделенной памятью. Как правило, это хранилище типа ключ-значение, которое используется как кэш для основного. Примеры такого типа кэша – Redis, Memcached и другие хранилища типа ключ-значение. Вот принципиальная схема их работы:



Сравнение двух типов кэшей:

СРАВНЕНИЕ ДВУХ ТИПОВ КЭШЕЙ

11

Характеристика	Встроенный кэш (inline)	Отдельные кэши (sidecar)
Скорость ответа	Выше: обращаемся напрямую к памяти	Ниже: есть сетевые вызовы, а также оверхед на работу самого хранилища
Память	Кэш и приложение делят одну область памяти, поэтому ее тратится меньше	Под кэш выделена отдельная память, поэтому ее тратится больше
Согласованность данных	Плохая: каждая копия приложения содержит только свои данные	Хорошая: все копии приложения обращаются к единому хранилищу
Масштабирование	Кэш нельзя масштабировать отдельно от приложения	Кэш и приложение можно масштабировать независимо друг от друга
Ресурсы (память, ЦПУ и т.д.)	Общие, поскольку "живут" в одном процессе	Выделенные, поскольку "живут" в разных процессах
Простота сопровождения	Проще: кэш – просто структура данных, предоставляемая библиотекой	Сложнее: кэш – отдельно разворачиваемый компонент, требующий отдельного мониторинга и экспертизы
Изолированность	Низкая: проводить отдельно работы над кэшем крайне сложно	Высокая: мы можем проводить любые работы над кэшем независимо от приложения
Горячий старт	Сложнее: приложение обычно не общается с диском напрямую	Проще: например, redis может периодически сбрасывать данные на диск и восстанавливать состояние после рестарта

Характеристика	Встроенный кэш (inline)	Отдельные кэши (sidecar)
Скорость ответа	Выше: обращаемся напрямую к памяти	Ниже: есть сетевые вызовы, а также оверхед на работу самого хранилища
Память	Кэш и приложение делят одну область памяти, поэтому ее тратится меньше	Под кэш выделена отдельная память, поэтому ее тратится больше
Согласованность данных	Плохая: каждая копия приложения содержит только свои данные	Хорошая: все копии приложения обращаются к единому хранилищу
Масштабирование	Кэш нельзя масштабировать отдельно от приложения	Кэш и приложение можно масштабировать независимо друг от друга
Ресурсы (память, ЦПУ и т.д.)	Общие, поскольку “живут” в одном процессе	Выделенные, поскольку “живут” в разных процессах
Простота сопровождения	Проще: кэш – просто структура данных, предоставляемая библиотекой	Сложнее: кэш – отдельно разворачиваемый компонент, требующий отдельного мониторинга и экспертизы
Изолированность	Низкая: проводить отдельно работы над кэшем крайне сложно	Высокая: мы можем проводить любые работы над кэшем независимо от приложения
Горячий старт	Сложнее: приложение обычно не общается с диском напрямую	Проще: например, redis может периодически сбрасывать данные на диск и восстанавливать состояние после рестарта

Не стоит думать, что один тип кэша хуже другого. У каждого из них свои достоинства и недостатки и каждый из них нужно применять с умом. Более того, их можно комбинировать. Например, вы можете использовать встроенный кэш как L1, а отдельный кэш как L2 перед основным хранилищем. В результате, такая схема может в разы сократить время ответа и снизить нагрузку на основное хранилище.

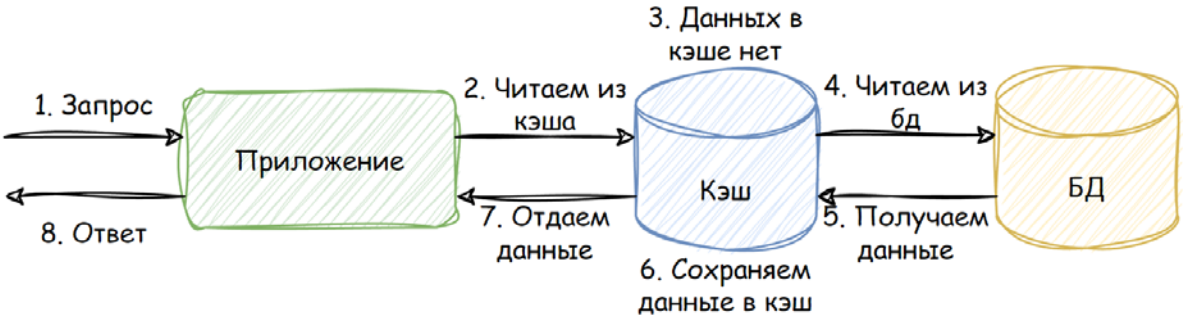
Стратегии работы с кэшем

Рассмотрим стратегии чтения и записи при работе с кэшем. В приведенных далее примерах кэш может быть как встроенным, так отдельным. Под “приложением” подразумевается некая бизнес-логика. Упор делается на описание взаимодействия компонентов друг с другом.

Cache through (Сквозное кэширование)

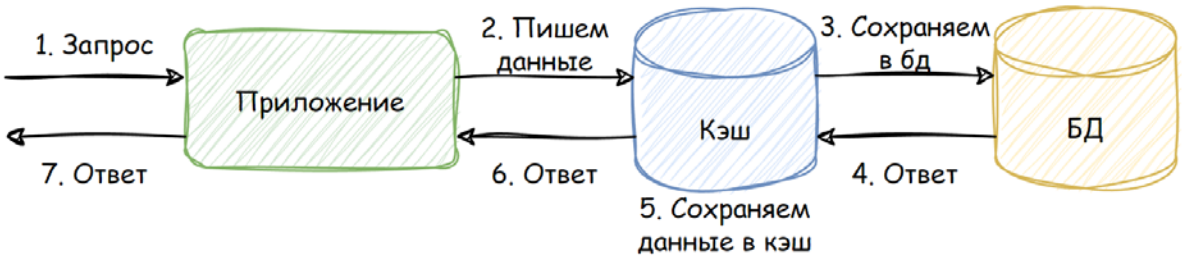
В рамках этой этой стратегии все запросы от приложения проходят через кэш. В коде это может выглядеть как связующее звено или “декоратор” над основным хранилищем. Таким образом, для приложения кэш и основная БД выглядят как один компонент хранилища.

Read through (Сквозное чтение)



1. Приходит запрос на получение данных
2. Пытаемся прочитать данные из кэша
3. В кэше нужных данных нет, происходит промах (miss)
4. Кэш перенаправляет запрос в БД. Это важный нюанс стратегии: к БД обращается именно кэш, а не приложение
5. Хранилище отдает данные
6. Сохраняем данные в кэш
7. Отдаем запрошенные данные приложению. Для приложения это выглядит так, как если бы хранилище просто вернуло ему данные, то есть шаги 3-6 скрыты от основной бизнес-логики
8. Возвращаем результат запроса

Write through (Сквозная запись)



1. Приходит запрос на вставку каких-либо данных
2. Отправляем запрос на запись через кэш. В этот момент кэш выступает только как прокси и сам по себе ничего не делает
3. Сохраняем данные в БД
4. БД возвращает результат запроса

5. Сохраняем данные в кэш. Делаем мы это специально после вставки, чтобы кэш и БД были консистентны. Если бы мы писали данные в кэш на шаге 2, а при этом на шаге 3 произошла бы ошибка, то кэш содержал бы данные, которых нет в БД, что может привести к печальным последствиям
6. Отдаем запрошенные данные приложению
7. Возвращаем результат запроса

Преимущества:

Очень простая схема работы с точки зрения организации кода

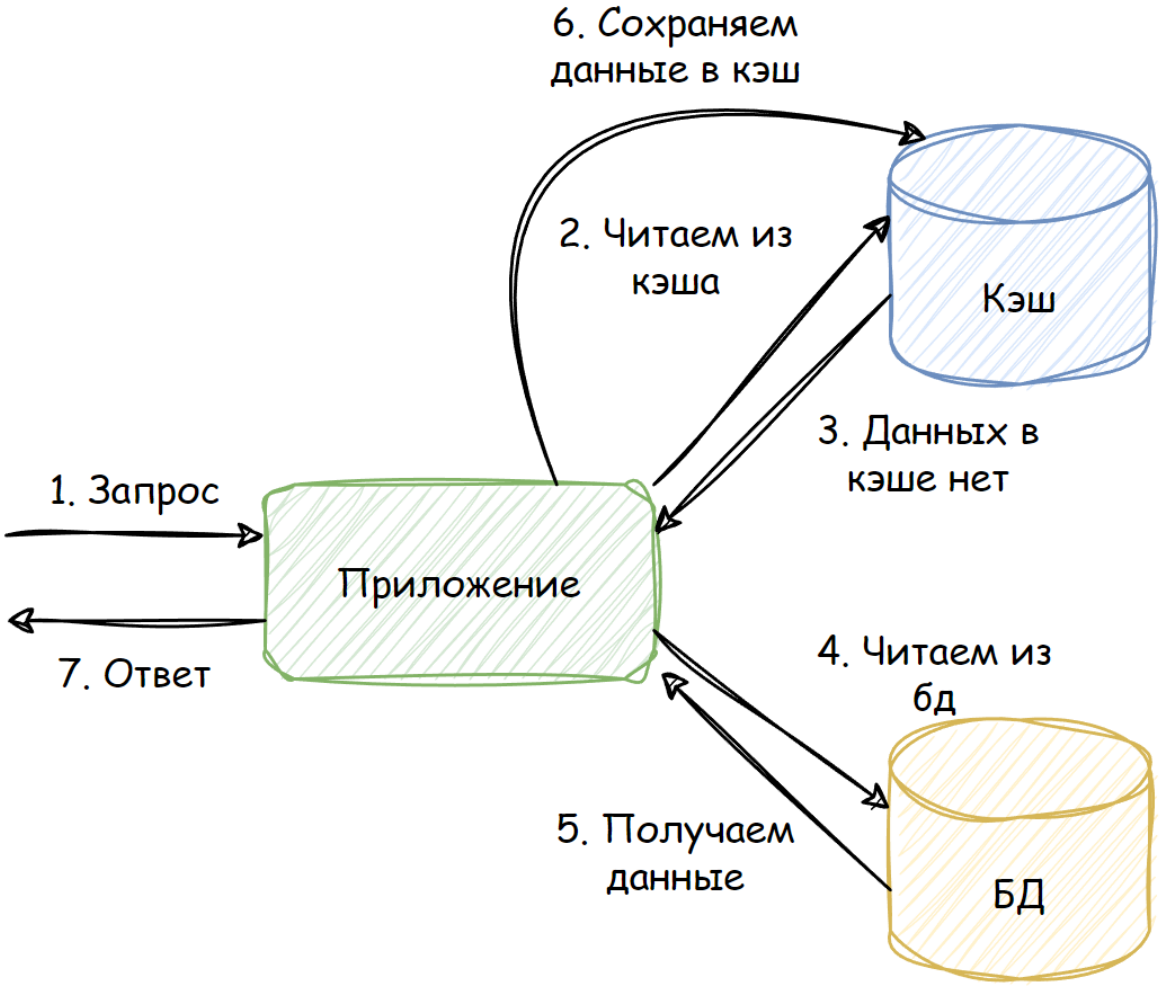
В коде легко добавить/убрать кэш, так как обычно это простая “обертка” над основным хранилищем

Недостатки:

Схема негибкая, поскольку записывать в кэш мы можем строго после выполнения основного запроса и в целом кэш скрыт от бизнес-логики

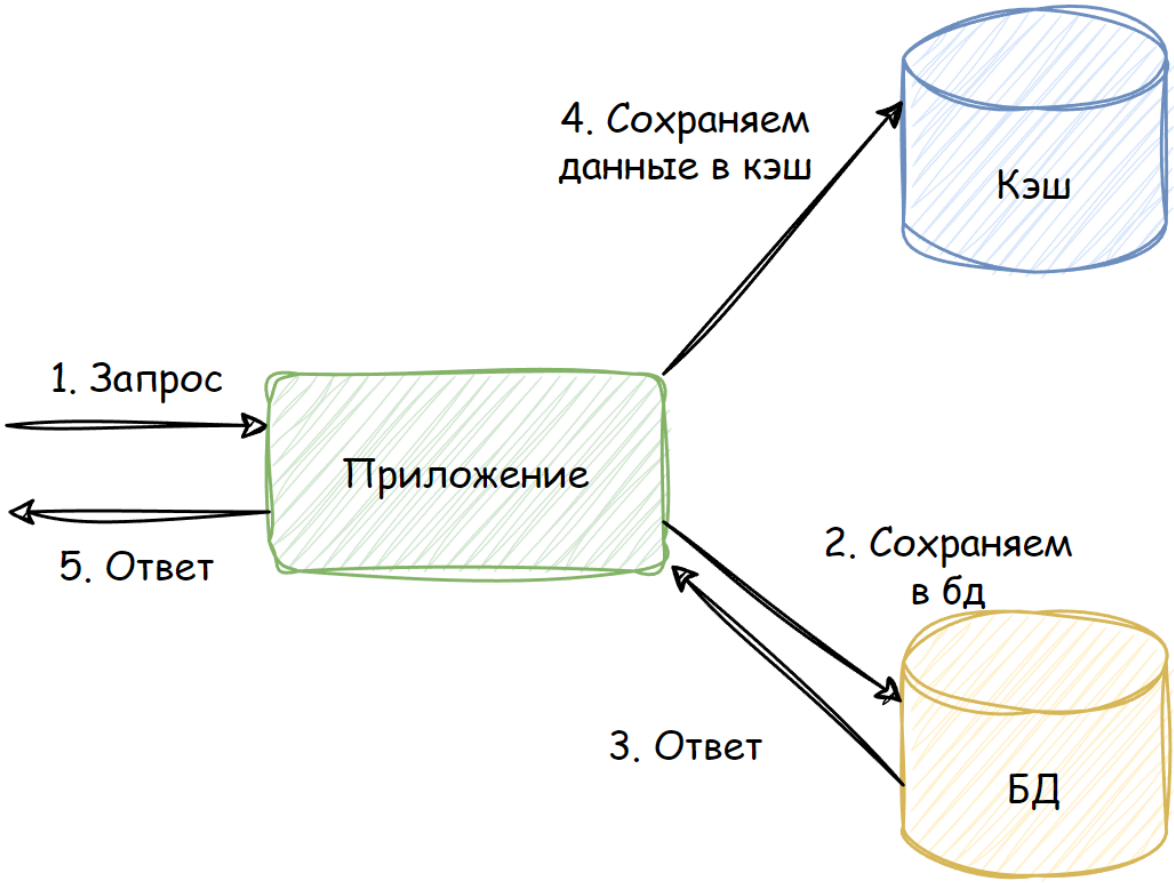
Cache aside (Кэширование на стороне)
В этой стратегии, приложение координирует запросы в кэш и БД и само решает, куда и в какой момент нужно обращаться. В коде это выглядит как два хранилища: одно постоянное, второе – временное.

Read aside (Чтение на стороне)



1. Приходит запрос на получение данных
2. Пытаемся читать из кэша
3. В кэше нужных данных нет, происходит промах (miss)
4. Приложение само обращается к хранилищу. В этом главное отличие от сквозного подхода: бизнес-логика в любой момент времени сама решает, куда обращаться – к кэшу или к БД
5. БД отдает данные
6. Сохраняем данные в кэш
7. Возвращаем результат запроса

Write aside (Запись на стороне)



1. Приходит запрос на вставку каких-либо данных
2. Сохраняем данные в БД
3. БД возвращает результат запроса
4. Сохраняем данные в кэш. Опять-таки, мы намеренно делаем это после вставки, чтобы кэш и БД были консистентны. Сохранение данных в кэше перед шагом 2 и ошибка на шаге 2 привели бы к появлению в кэше данных, которых нет в БД. Результат был бы все тот же – печальные последствия
5. Возвращаем результат запроса

Преимущества:

Гибкость. Управление полностью в руках приложения. Оно может сохранить данные в кэш сразу после обращения к БД, а может предварительно сделать еще ряд манипуляций с данными и только затем сохранить их в БД

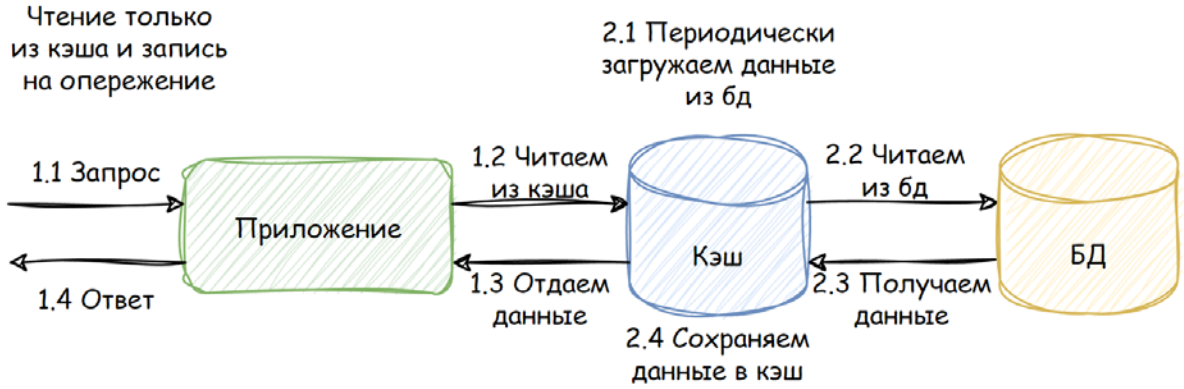
Недостатки:

Схема работы немного сложнее с точки зрения организации кода

В коде сложнее добавить/убрать кэш, поскольку это отдельный компонент, с которым взаимодействует бизнес-логика. Изменить этого взаимодействие может быть сложно

Cache ahead (Опережающие кэширование)

Эта стратегия предназначена только для запросов на чтение. Они всегда идут только в кэш, никогда не попадая в БД напрямую. По факту мы работаем со снимком состояния БД и обновляем его с некоторой периодичностью. Для приложения это выглядит просто как хранилище, как и в случае со сквозным кэшированием.



1. Входящие запросы на чтение:
2. Приходит запрос на получение данных
3. Читаем из кэша. Если в кэше нет нужных данных, то возвращаем ошибку. К БД в случае промаха не обращаемся
4. Если данные есть, то отдаем их приложению
5. Возвращаем результат запроса
6. Обновление кэша:
7. Периодически запускается фоновый процесс, который читает данные из БД.
8. Читаем актуальные данные из БД
9. БД отдает данные
10. Сохраняем данные в кэш

Преимущества:

Минимальная и полностью контролируемая нагрузка на БД. Клиентские запросы не могут повлиять на БД

В коде легко добавить/убрать кэш, поскольку можно просто заменить кэш на основное хранилище и обращаться уже к нему

Простота, так как не приходится иметь дело с двумя хранилищами

Недостатки:

Кэш отстает от основного хранилища на период между запусками обновления кэша. Нужно помнить, что на момент обращения свежие данные могут еще “не доехать” до кэша. Эта проблема может быть решена использованием сквозной записи или записи на стороне. Тогда, при обновлении данных в БД, данные будут обновляться и в кэше

Вы можете использовать описанные стратегии в любых комбинациях. Например, вы можете взять опережающие кэширование, добавить туда сквозную запись и чтение на стороне, чтобы добиться максимальный актуальности данных и избежать промахов по максимуму!

Стратегии инвалидации
Инвалидация – это процесс удаления данных из кэша или пометка их как недействительных. Делается это для того, чтобы гарантировать актуальность данных, с которыми работает приложение.

Инвалидация по TTL
TTL (Time To Live) – время жизни данных в кэше. При сохранении данных в кэш для них устанавливается TTL и данные будут обновляться с периодичностью не менее TTL.

Это самый простой способ инвалидации данных. Тем не менее, у этой стратегии есть свои подводные камни.

Самый главный из них – вопрос длительности TTL. Если TTL слишком короткий, то запись может “протухнуть” и стать недействительной раньше, чем обновление было бы необходимо, что приведет к отправке повторного запроса в источник данных. Если TTL слишком длинный, то запись может содержать устаревшие данные, что может привести к ошибкам или неправильной работе приложения. Обычно ответ на этот вопрос подбирается эмпирическим путем.

Есть, впрочем, и другой вариант. Источник данных может присылать TTL сам, тогда клиенту не придется выбирать TTL, а просто брать предлагаемый. Такой подход, например, можно использовать в HTTP.

Сложность иного рода возникает, если записи становятся недействительными одновременно в большом количестве. В таком случае возникает множество запросов в источник данных, что может привести к проблемам с производительностью, а то и вовсе “положить” его. Для избежания подобной ситуации, можно использовать jitter.

Jitter – это случайное значение, добавляемое к TTL. Если в обычном случае все записи имеют, например, TTL = 60 сек., то при использовании jitter с диапазоном от 0 до 10 сек. TTL будет принимать значение от 60 до 70 сек. Это позволит сгладить количество записей, переходящих в состояние недействительных одновременно.

Jitter позволил нам сгладить нагрузку на источник данных, когда “протухает” много записей сразу. Но что делать, если есть одна запись, которую интенсивно используют? Ее инвалидация приведет к тому, что все запросы, которые не нашли данных в кэше, одновременно обратятся к источнику. Тогда нам нужно схлопнуть все эти запросы в один. В go есть для этого отличная библиотека singleflight. Она определяет одинаковые запросы, возникающие одномоментно, выполняет лишь один запрос в источник, а затем отдает результат всем изначальным запросам. Таким образом, если у нас возникли десять запросов, библиотека выполнит только один из них, а результат вернет всем десяти. Стоит отметить, что эта библиотека работает только в рамках одного экземпляра приложения. Если у вас их несколько, то даже с использованием этой библиотеки в источник может уйти больше одного запроса.

Инвалидация по событию
При таком подходе данные инвалидируют при наступлении некоего события – обычно это обновление данных в источнике. На самом деле, мы уже рассмотрели этот способ, когда говорили про стратегии использования кэширования, а именно write through и write aside.

Также в качестве события для инвалидации данных может выступать время последней модификации данных. Такой способ используется в HTTP.

Стратегии вытеснения
Размер кэша ограничен, он гораздо меньше основного хранилища, а значит, мы не может разместить в нем все данные. Что делать, когда память, выделенная под кэш, полностью заполнена, а новые записи продолжают поступать?

Ничего не делать

Конечно, можно игнорировать все новые записи и ждать, пока существующие истекут по TTL. Однако в общем случае это крайне неэффективно, поскольку все запросы идут в источник данных и все преимущества кэша нивелируются. Поэтому обычно используются более эффективные стратегии вытеснения.

Random

Random – стратегия вытеснения, при которой удаляются случайные записи. Это самая простая стратегия: просто удаляем то, что первым попалось под руку. Но этот способ недалеко ушел от стратегии “ничего не делать”.

TTL

TTL – стратегия вытеснения, предусматривающая удаление той записи, которой осталось меньше всего “жить”, то есть у которой TTL истечет раньше всех. Как и random, эта стратегия немного лучше, чем “ничего не делать”, но все еще недостаточно эффективна.

LRU

LRU (Least Recently Used) – стратегия вытеснения, которая опирается на время последнего использования записи. Она удаляет записи, у которых время последнего использования старше остальных. Таким образом, в кэше остаются записи, которые использовались недавно. Эта стратегия опирается уже не на случай, а на паттерн использования данных, поэтому она гораздо эффективнее предыдущих.

Эта стратегия хорошо подходит, когда:

недавно использованные данные, скорее всего, будут использованы снова в ближайшем будущем

нет данных, которые используются чаще остальных

вы не знаете, что именно вам нужно

LFU

LFU (Least Frequently Used) – стратегия вытеснения, опирающаяся на частоту использования записи. Она удаляет записи, которые использовались реже всего. Так в кэше остаются данные, которые использовались чаще других. Эта стратегия тоже опирается не на случай, а на паттерн использования данных, поэтому она тоже эффективнее остальных и является альтернативой LRU.

Эта стратегия хорошо подходит, когда есть данные, которые используются значительно чаще остальных. Такие данные разумно не вытеснять из кэша, чтобы избежать лишних “походов” в источник.

Кэширование ошибок
Ранее я упомянул, что мы можем кэшировать ошибки. На первый взгляд это может показаться странным: зачем нам вообще кэшировать ошибки? На самом деле, это крайне полезная штука.

Представим себе, что клиент запрашивает данные, которых нет в источнике. Пусть это будем информация о товаре по id. Казалось бы, нет данных и ладно: просто сходим в источник, ничего не получим и сообщим клиенту. Но что, если таких запросов много? А что, если кто-то делает это специально?

Это типичная схема так называемой атаки через промахи кэша (cache miss attack). Ее суть в запрашивании данных, которых заведомо не может быть в кэше, поскольку их нет в источнике. Вал таких запросов может привести к проблемам с производительностью источника и даже к его “падению”. Этого можно избежать, если кэшировать ошибку, тогда последующие запросы того же рода будут попадать в кэш и источник не пострадает.

Но тут тоже нужно быть осторожным. Если хранить в одном кэше и полезные данные, и ошибки, то в случае атаки полезные данные могут вытеснены из кэша. Поэтому я бы рекомендовал иметь выделенный кэш под ошибки. Он может быть меньшего объема, чем основной кэш.

Также кэширование ошибок полезно, если сервис, к которому вы обращаетесь, “почувствовал себя плохо”. Чтобы не забивать его запросами, которые, скорее всего, не будут выполнены, а лишь усугубят проблему, лучше кэшировать ошибки на несколько секунд. Таким образом, мы перестанем оказывать негативное воздействие на сервис и дадим ему возможность нормализоваться. Но с этой задачей лучше справляется паттерн Circuit Breaker, который мы не будем рассматривать в рамках этой статьи.

Заключение
На этом пока все. Я не ставил перед собой целью раскрыть тему кэширования исчерпывающе: наверняка многие вещи остались не рассмотренными в рамках

этой статьи. Я хотел лишь предоставить структурированную основу и хочу верить, что у меня это получилось.

Надеюсь, данный материал оказался вам полезен, вы открыли для себя что-то новое или структурировали уже известное.

1.1. Применение в разработке приложений

Пользователи обычно не знают, какие подходы применяются при разработке, как настроен сервер, какие клиентские и серверные средства разработки используются. Для них лишь важно, насколько сайт полезный, удобный и быстрый. Задача же веб-разработчиков заключается в том, чтобы не доставлять пользователю лишних неудобств, радовать его и тем самым стимулировать продажи, идущие через сайт, или число рекламных показов и кликов по ним.

Далее рассмотрим, как можно организовать создание веб-приложения, ориентируясь на самые важные аспекты клиентской оптимизации.

Этап 1: Доставка информации и оформления

На этом этапе разработчики должны сделать все возможное, чтобы не замедлить скорость загрузки страницы. Фактически идет речь об ускорении первой стадии загрузки. Наиболее важными методами здесь является сжатие (gzip) текстовых файлов и объединение файлов стилей (CSS). Для CSS- и JavaScript-файлов возможно применять статическое архивирование (без необходимости архивировать каждый раз эти файлы «на лету»).

При загрузке страницы браузер запросит все CSS-файлы, объявленные в head страницы, последовательно. Поэтому каждый файл добавляет задержку в загрузке, равную времени запроса к серверу. Для файлов скриптов рекомендуется применить либо также объединение, либо вообще вынести их в пост-загрузку.

Итог первого этапа — это доставленный и оформленный HTML. И издержки на доставку JavaScript сведены к минимуму (на этом этапе он только мешает, поскольку замедляет отображение основного содержимого страницы). Время от начала до завершения загрузки такой страницы при включённом и выключенном JavaScript (при вынесении его в пост-загрузку) фактически будет одинаковым.

Это и будет выигрышем в скорости загрузки!

Этап 2: Кэширование файлов оформления и параллельные запросы

На данном этапе разработчики должны обеспечить быструю загрузку других страниц сайта (если посетитель решит туда перейти). Этот этап должен проходить параллельно с первым. Настройка кэширующих заголовков

достаточно тривиальна. Несколько сложнее наладить процесс разработки для своевременного сброса кэша.

Одно или несколько дополнительных зеркал для выдачи статических ресурсов легко настраиваются в конфигурации, однако внедрить это в схему публикации изменений гораздо сложнее. Обычно это делают уже после разработки макетов страниц. Число дополнительных хостов следует напрямую из числа статических файлов (обычно картинок), поэтому надо определиться с ними на этапе автоматизации процесса публикации. О параллельных запросах рассказывается в пятой главе.

CSS Sprites достаточно трудоемки в автоматической «склейке», поэтому их внедряют обычно на первом этапе (при создании макета страниц). При использовании метода data:URI на первом этапе о них можно забыть, потому что автоматизированное решение просто в реализации и не требует от верстальщика отдельных технологических познаний.

Этап 3: Жизнь после загрузки страницы

Целью данного этапа является создание различных обработчиков событий, которые должны взаимодействовать с пользователем. Это могут быть и всплывающие подсказки, и подгрузка данных с сервера, и просто анимация. Все это можно назвать «оживлением» страницы.

Говорят, что иногда «грамм видимости важнее килограмма сути» — это как раз про JavaScript. Ведь именно на нем можно реализовать механизмы, упрощающие действия пользователя; можно сделать много различных визуальных эффектов, подчеркивающих оформление, удобство и полезность сайта (а фактически — усилить и сфокусировать всю работу, которую проделали разработчики на предыдущих этапах).

К этому моменту мы должны иметь оформленную HTML-страницу, на которой все ссылки и формы обязаны работать без JavaScript.

У нас должны быть готовы серверные интерфейсы для AJAX-запросов; структура страницы должна быть такой, чтобы для аналогичных кусков HTML-кода не приходилось реализовывать аналогичные, но не одинаковые куски JavaScript-кода. Скорее всего, должны быть созданы шаблоны страниц, где видно, как будет выглядеть страница после какого-то действия пользователя (обычно специалист по удобству использования создает макеты).

Чтобы не уменьшать скорость доставки контента и оформления, JavaScript-файлы (лучше всего, конечно, один JavaScript-файл; несколько файлов должны использоваться только при большой сложности клиентского интерфейса) должны быть подключены перед закрытием тега body (а в идеале — вынесены именно в пост-загрузку).

Задача по обеспечению взаимодействия пользователя с интерфейсом сайта сводится к выполнению следующих действий:

1. найти DOM-элементы, требующие «оживления» (далее — компоненты);
2. определить, что это за компонент;
3. обеспечить подключение необходимого кода JavaScipt;
4. следить за очередностью подключения файлов;
5. не позволять нескольких загрузок одного файла.

Все это напрямую следует из концепции «ненавязчивого» JavaScript, которая описана в седьмой главе.

Поиск необходимых DOM-элементов должен нам дать список названий JavaScript-компонентов. Названия компонентов должны однозначно соответствовать названиям файлов на сервере, в которых содержится код для них. Также нам может понадобиться загрузить некоторые дополнительные CSS-правила для найденных компонентов (в случае небольшого количества CSS-кода разумно будет включить его в основной файл) ради каких-то визуальных эффектов, которые можно пропустить на первом этапе загрузки. Например, все эффекты по смене изображения при наведении мыши обеспечиваются через CSS-правила и технику CSS Sprites.

Список названий компонент можно объединить в один запрос к серверу. В итоге на стадии пост-загрузки должны осуществляться запросы к файлам вида

static.site.net/jas/componentName1.css;componentName2.css

и static.site.net/jas/componentName1.js;componentName2.js.

У данного подхода есть два недостатка:

1. В папке /jas/ (которую мы, например, используем для кэширования наиболее частых вариантов подключения модулей) через некоторое время может оказаться очень много файлов, что теоретически может уменьшить время доступа к ним на сервере.
2. Иногда на странице может оказаться очень много компонент, причём так много, что длина имени запрашиваемого объединенного файла перевалит за возможности файловой системы (например, 255 символов у Ext3) — в этом случае потребуется разбить один запрос на несколько последовательных.

Этап 4: Предупреждаем действия пользователя

Если после посещения главной страницы большинство пользователей попадают внутрь сайта, то логично будет после полной загрузки главной страницы запрашивать стили и скрипты, применяемые на других страницах сайта. Для пользователя это выльется в небольшое увеличение трафика (при использовании сжатия текстовая информация составляет 10-20% от объема графики), однако во вполне заметное ускорение загрузки последующих страниц.

Аналогично можно рассмотреть и предзагрузку некоторых наиболее часто используемых картинок, которые отсутствуют на главной странице, и дополнительных JavaScript-модулей, которые применяются на текущей странице для дополнительного функционала и не запрашиваются при первой загрузке страницы (например, отвечают за первоначально скрытые блоки).

Естественно, что за балансировку третьей и четвертой стадий отвечает уже JavaScript-разработчик и фронтенд-архитектор — ведь именно в зоне ответственности последнего находится скорость загрузки страницы