В whoosh, юрент или яндекс go API обеспечивает:

- корректное общение клиента и сервера,
- надёжную работу микросервисов,
- безопасность и масштабируемость всей системы.

Хорошо спроектированный API — это как идеальный мост:
незаметный, но жизненно важный.

ЗАКЛЮЧЕНИЕ (2–3 мин)

Проектирование систем — это искусство баланса.
Ты постоянно выбираешь: скорость или надёжность, простота или гибкость.

Главное — понимать, для кого и зачем ты строишь систему.
Начинай с требований, продумывай архитектуру, закладывай масштабирование и делай API понятным.

И тогда твой сервис, как и *whoosh, юрент или яндекс go*, будет стабильно работать, даже когда им пользуются тысячи людей.

Глава 3. Продвинутое проектирование систем

Спикер: Тимур

Вступление к 3 главе

Добро пожаловать в третью главу нашего курса. Меня зовут Тимур, я Senior разработчик и ментор. В предыдущей главе мы построили фундамент — научились проектировать базовую архитектуру систем. Теперь пришло время углубиться в те вопросы, знание которых поможет вам глубже погрузиться в system design, расширить свой кругозор и начать применять эти знания в работе.

Сегодня мы познакомимся с понятием согласованности данных в распределенной системе и поймем как ее можно обеспечивать, также узнаем о разных вариантах кэширования для снижения latency нашей системы. Эти знания вы сможете применять

как и в работе, так и на system design интервью, когда будете углубляться в конкретные решения в вашей системе."

Согласованность

Вспомним термины из CAP теоремы

- **Доступность** означает, что система отвечает на запросы, при этом не обязательно являются ли ответы валидными.

- **Согласованность** (консистентность) означает, что пользователи не видят противоречащих друг другу данных.

И ещё одно важное понятие, которое часто звучит «страшно», но я объясню просто —**линеаризуемость**:

Линеаризуемость — это свойство системы при котором она ведёт себя так, будто все операции исполняются по очереди в одном месте. Если запись подтвердилась, то все последующие чтения увидят именно эту запись.

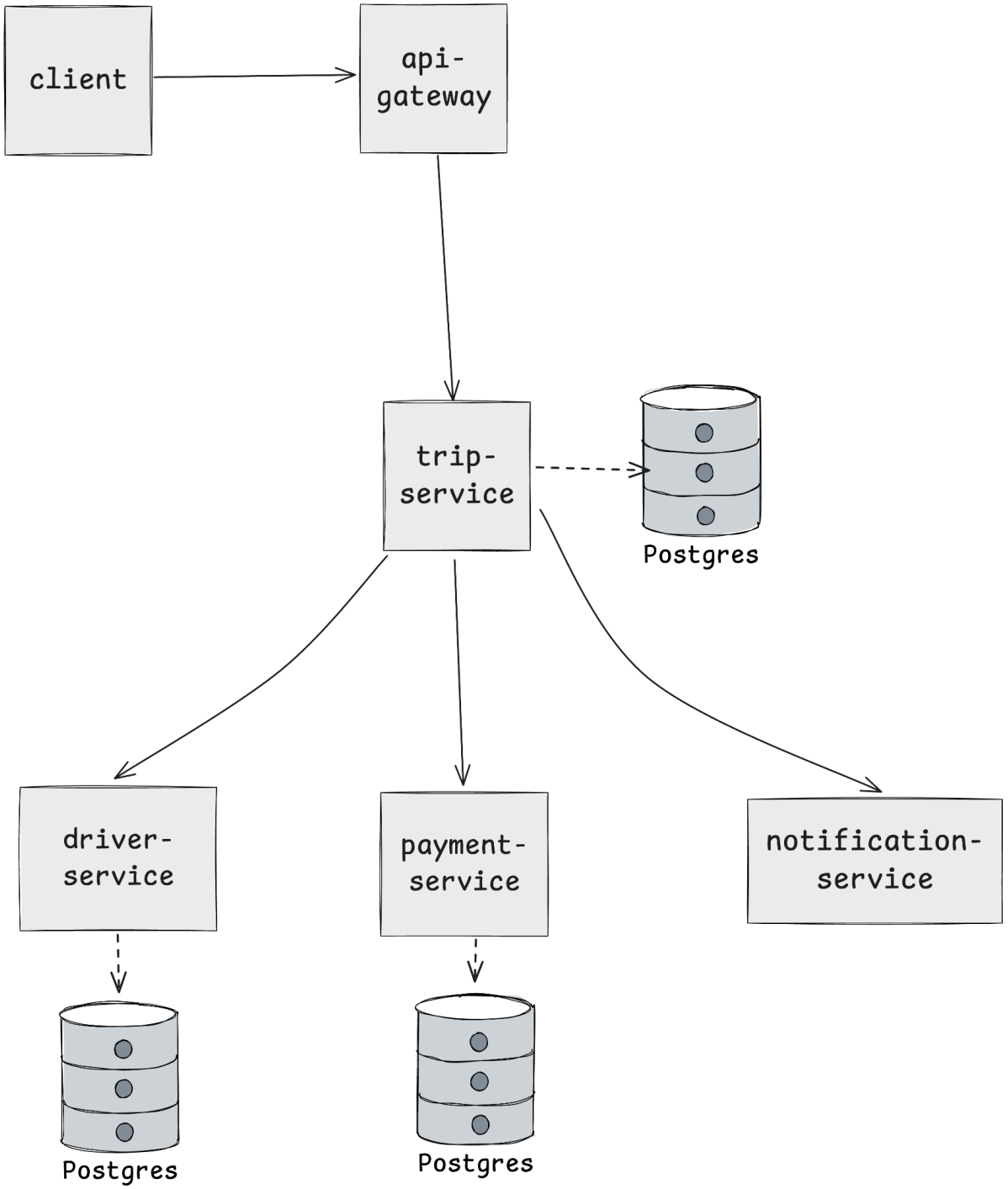
Это самый строгий режим. Представь интернет-магазин с последней парой кроссовок. Двое — Алиса и Боб — пытаются купить их почти одновременно. **Линеаризуемость** гарантирует, что итог будет таким, как будто покупки обработали по одной в некотором порядке, уважая реальное время совершения операции: если покупка Алисы успела завершиться до начала покупки Боба, Боб не сможет купить товар.

Реальные системы чаще живут в компромиссах. Есть модели слабее:

- **Причинная(causal)** модель выглядит так — события, связанные причинно, видны в правильном порядке. Независимые события могут «обгонять» друг друга. В такси отзыв не появится раньше самой поездки.В социальной сети пользователь видит свой комментарий сразу, даже если его не видят другие пользователи.

- **Конечная (eventual) модель она гласит**— если новых записей нет, все копии данных «сойдутся» со временем. В промежутках разные пользователи могут видеть разные значения.

Теперь давайте рассмотрим согласованность на примере реальной системы. В системе такси: заказ поездки затрагивает минимум 4 сервиса: сервис поездок, сервис поиска водителей, сервис платежей и уведомлений. У сервиса поездок и сервиса платежей отдельные базы данных. Как нам обеспечить согласованность нашей системы?



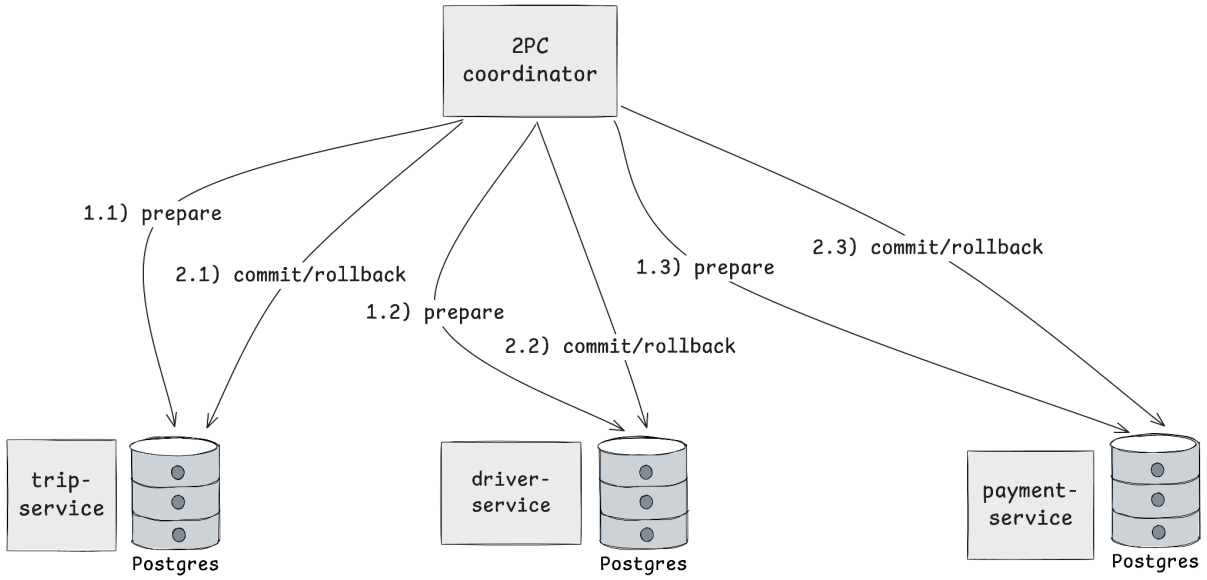
Явно распишем все инварианты нашей системы(сноска в видеоряде - Инвариант — это правило/утверждение про состояние системы, которое **всегда должно быть истинно во всех допустимых состояниях. Мы запускаем систему так, чтобы инвариант был верен в начале, и проектируем каждую операцию так, чтобы она не могла его нарушить.). Это правила, которые система не имеет права нарушать ни при каких сбоях:

- **У водителя не может быть двух активных заказов одновременно.**
- **Платёж либо проведен(зарезервирован), либо нет — промежуточного состояния не бывает.**
- **Нельзя начать поездку, если у пользователя на счету недостаточно средств.**(тут для простоты картины мы считаем, что пользователь имеет внутренний счет, которым мы можем легко управлять)
- **Статусы переходят только вперёд по валидным шагам, например статус поездки может обновляться строго по порядку: поездка создана, ожидание водителя, ожидание оплаты, подтверждена.**

Все остальные участки — допускают короткую рассинхронизацию.

Теперь рассмотрим сценарий начала поездки. Пользователь отправляет запрос на создание поездки, в это время наша система определяет стоимость поездки, резервирует деньги на счету, закрепляет водителя, подтверждает поездку и затем отправляет уведомление водителю и пассажиру с информацией о заказе. Все шаги, кроме отправки уведомления необходимо сделать атомарно, ведь исходя из наших инвариантов мы не можем позволить поездке начаться, если пользователь не способен за нее заплатить. Но данные находятся в разных сервисов. Тут проблему атомарности нам помогут решить распределенные транзакции. Есть два популярных способа осуществить подобную транзакцию: двухфазный коммит и сага.

Принцип работы 2PC



Двухфазный коммит работает напрямую с храналищами, в нашем случае мы могли бы его реализовать с помощью оркестратора, имеющего доступ к базам данных всех нужных сервисов.
Как следует из названия, протокол выполняется в две фазы

Фаза подготовки (prepare phase, фаза голосования): Координатор рассылает всем участникам запрос на подготовку к коммиту. Каждый участник локально выполняет свою часть работы транзакции (например, делает необходимые изменения в своей БД) но не фиксирует их, а помечает как “готовые к коммиту” (в базах данных это обычно означает записать изменения в журнал и заблокировать ресурсы). После этого участник отвечает координатору либо “готов” (Yes), если его этап прошёл успешно и он готов зафиксировать, либо “не могу” (No), если произошла ошибка и он не сможет закоммитить. Все участники по сути голосуют “за” или “против” общей фиксации.

Фаза фиксации (commit phase): Координатор собирает ответы. Если все участники ответили "готов/Yes", то координатор посылает команду Commit всем участникам. Каждый участник получает эту команду и выполняет фиксацию своих изменений (окончательно применяет их в своей системе). Если хотя бы один участник ответил отказом ("No"), либо не ответил из-за сбоя, координатор посылает команду Rollback всем тем, кто был готов. В результате все участники отменят свои изменения (или не будут фиксировать). Таким образом, достигается атомарность: либо все commit, либо все rollback.

После завершения второй фазы координатор может сообщить инициатору (например, приложению, начавшему транзакцию), что транзакция успешно выполнена или откатилась.

Для реализации 2PC необходима поддержка со стороны всех участников. Каждый ресурс должен обладать интерфейсом приготовления/фиксации. В мире реляционных БД это стандарт XA; в NoSQL-хранилищах или пользовательских сервисах такой поддержки часто нет. Поэтому в чистых микросервисах, где сервисы используют разные хранилища, самостоятельно реализовывать 2PC сложно – нужно либо писать слой-адаптер для каждого (чтобы сервисы умели принимать команды prepare/commit), либо ограничиться теми ресурсами, что уже поддерживают XA.

Ограничения и недостатки двухфазного коммита
Хотя 2PC гарантирует строгую согласованность, в распределённых системах он имеет известные недостатки:

Блокировка ресурсов и производительность: На фазе подготовки каждый участник обычно блокирует изменяемые ресурсы (например, строки в базе) до получения команды commit или rollback. Если участников много, и они ждут друг друга, это может затормозить систему. В случае задержек или сбоев ожидание может быть длительным, и заблокированные ресурсы недоступны другим транзакциям – снижая параллелизм. 2PC поэтому плохо масштабируется с ростом числа участников: задержка одного узла удерживает всех.

Точка отказа – координатор: Координатор транзакции – центральный “командир”. Если он выйдет из строя в неподходящий момент, участники останутся в подвешенном состоянии. Например, участники подготовились и ответили “Yes”, ждут команду, а координатор умер или потерял связь – они не знают, коммитить им или откатывать. Такая ситуация называется блокировкой (blocking) в 2PC. В базах данных участники в ожидании решения могут держать транзакцию открытой очень долго. Требуется администратор или специальный механизм восстановления координатора, чтобы решить судьбу транзакции. Существуют расширения (протокол трехфазного коммита, 3PC) для смягчения этой проблемы, но полностью без координатора 2PC не работает.

Оверхед на коммуникацию: 2PC требует два раунда сетевого взаимодействия (prepare, затем commit/rollback). В условиях высокой нагрузки или сетевых задержек это добавляет значительный оверхед к каждой транзакции. Saga же, напротив, чаще работает асинхронно, и шаги могут выполняться без центральной синхронизации.

Слабая приспособленность к отказам связи: Если один из участников недоступен, координатор не получит от него “готов”, и вся транзакция должна откатиться (ради консистентности жертвуем доступностью). В микросервисах, где отказ отдельного сервиса не редкость, 2PC приводит к более частым отказам всей операции по сравнению с Saga, которая могла бы дождаться восстановления сервиса и продолжить (или выполнить компенсации).

Сложность внедрения в микросервисах: Применение 2PC между независимыми сервисами идёт вразрез с философией слабой связанности. Все участники должны быть “подчинены” общему координатору и протоколу. Это усложняет архитектуру – фактически получается распределённый монолит на уровне транзакций. Поэтому многие избегают 2PC в микросервисной архитектуре, предпочитая eventual consistency подходы (Saga, TCC и пр.).

Следует отметить, что 2PC остаётся полезным в ограниченных сценариях, где небольшое число участников и жёсткое требование атомарности. К примеру, финансовые операции между двумя банковскими системами могут использовать двухфазный коммит, если наличие центрального координатора приемлемо. Но во внутренней архитектуре современных веб-сервисов 2PC – редкий гость; чаще его заменяют на паттерны, обеспечивающие “логическую” атомарность, как Saga.

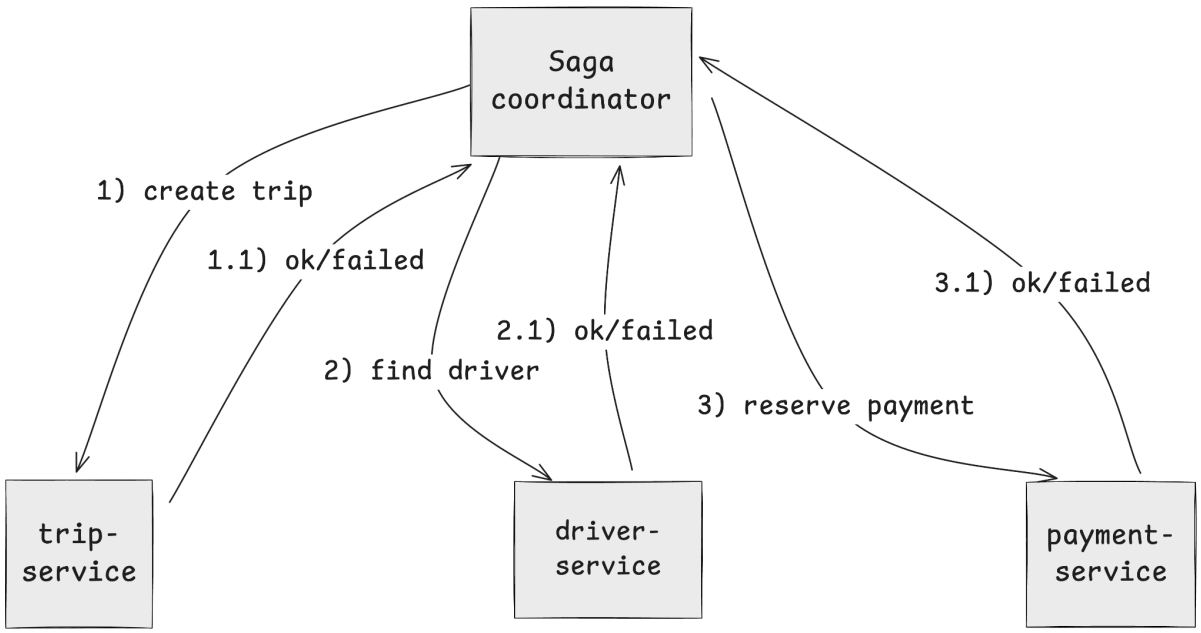
Saga – это паттерн, позволяющий достичь целостности данных в распределённых системах без использования глобальных ACID-транзакций. Идея состоит в том, чтобы разбить большую бизнес-транзакцию, затрагивающую несколько сервисов, на последовательность локальных транзакций в каждом из сервисов. Каждый шаг выполняется в рамках своего сервиса атомарно. Если все шаги прошли успешно – вся сага считается успешной. Если же на каком-то этапе произошла ошибка, паттерн Saga

предусматривает выполнение компенсирующих транзакций для отмены уже выполненных действий и возврата системы в согласованное состояние.

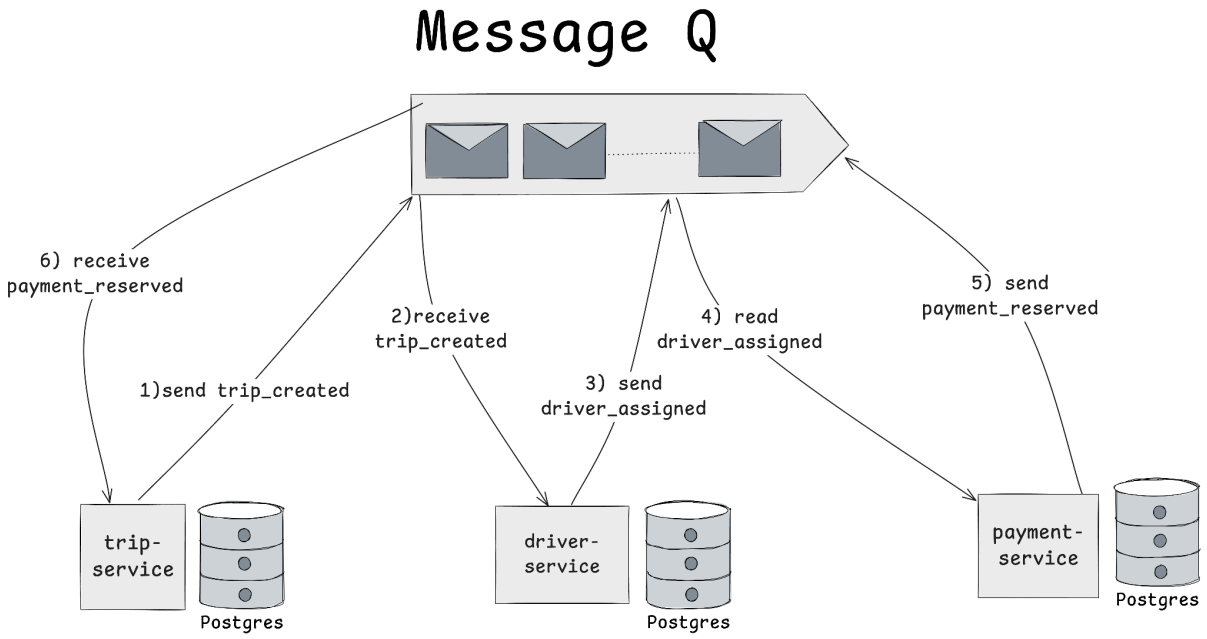
Оркестрация vs хореография саги
Существуют два подхода к координации шагов саги: оркестрация и хореография. Разница в том, где заложена логика последовательности действий:

Оркестрацией управляет специальный центральный компонент – оркестратор саги. Оркестратор знает сценарий целиком и по шагам рассылает команды сервисам: что делать дальше. Он вызывает Service A, затем Service B, и т.д., и в случае ошибки также инициирует компенсирующие действия. Оркестратор упрощает понимание последовательности (вся логика сосредоточена в одном месте), но вводит дополнительный компонент и точку контроля. В нашем сценарии оркестратором может выступать сервис поездок, т.к он может знать все о сценарии создания поездки. Например сервис поездок создает при получении запроса на создание поездки создает оркестратор Саги, который начинает распределенную транзакцию:

1) Создает поездку в собственной базе данных со статусом “created или pending”
2) Отправляет запрос на поиск водителя в driver-service.
3) Дожидается ответа и решает продолжать транзакцию или же откатить ее. В случае если водитель не был найден, в базе сервиса trip поездка помечается как canceled
4) Если водитель был найден, обновляет в записи поездки информацию о водителе(например его id)
5) Отправляет запрос в сервис платежей для того, чтобы зарезервировать средства клиента
6) Если получилось зарезервировать, то помечает заказ как подтвержденный, одновременно с этим асинхронно отправляя сообщение в notification service. (не забыл о паттерне transactional outbox). Если зарезервировать средства не получилось, то все предыдущие шаги откатываются. Водитель освобождается, заказ помечается как отмененный



При Хореографии нет центрального руководителя – каждый сервис сам реагирует на события и решает, что делать дальше. То есть взаимодействие основано на обмене сообщениями: выполнение шага одним сервисом генерирует событие, которое подхватывают другие сервисы, запускающие свои шаги. Например:
Все сервисы общаются через шину данных:



Хореография исключает центральный координатор – система распределённая и более гибкая, но последовательность действий “размазана” по сервисам, что усложняет сопровождение и отладку.

Оба подхода широко используются. В простых системах часто хватает хореографии (реактивный подход), но для сложных процессов с множеством условий может быть удобнее оркестрация (явное прописывание сценария).

Преимущества паттерна Saga
Паттерн Saga стал популярным решением для согласованности данных между микросервисами благодаря нескольким сильным сторонам:

Отсутствие глобальных блокировок и распределённых ACID: Cага позволяет обойтись без двухфазного коммита и других тяжёлых механизмов. О двухфазном коммите поговорим позже. Каждый сервис работает в пределах своей транзакции, что упрощает масштабирование. Данные между сервисами синхронизируются через события/команды, а не через общий транзакционный менеджер.

Устойчивость к частичным сбоям: Если один из сервисов в процессе недоступен или операция не удалась, Saga может компенсировать уже выполненные действия. Система возвращается в консистентное состояние без ручного вмешательства. Это повышает отказоустойчивость (каждый локальный шаг откатывается независимо).

Гибкость бизнес-логики: Компенсационные действия не обязательно должны быть точной обратной операцией – это могут быть любые меры, удовлетворяющие бизнес-требованиям для восстановления целостности. Например, если отменить платёж нельзя автоматически, можно пометить заказ как требующий ручного вмешательства. Такая бизнес-ориентированная обработка ошибок невозможна при механическом ACID-откате.

Масштабируемость и производительность: Сервисы взаимодействуют асинхронно (особенно при хореографии), что позволяет им работать параллельно и не ждать друг друга, если логика позволяет. Нет удержания глобальных блокировок – повышается пропускная способность системы. В больших распределённых системах Saga масштабируется лучше, чем централизованный транзакционный координатор.

Сложности и подводные камни Saga
Несмотря на плюсы, реализация Saga приводит к ряду сложностей, о которых архитекторам и разработчикам нужно помнить:

Сложность программирования и компенсирующая логика: Разработчики должны явно продумывать и кодировать компенсирующие транзакции для каждого шага. Это добавляет работы и требует глубоко понять бизнес-процесс: что делать, если шаг X уже выполнен, а шаг Y провалился? Проектирование таких компенсаций порой непросто (например, «отменить платеж» – нетривиальная задача, если деньги уже списаны).

Идемпотентность операций: Критически важно, чтобы и основные шаги, и компенсации были идемпотентны, то есть повторный запуск операции не изменял состояние сверх первого эффекта. Причина – в распределённой среде неизбежны ретраи и дублирующиеся сообщения. Например, сервис может случайно дважды получить событие или повторно выполнить компенсацию после сбоя. Если компенсационное действие (например, отмена заказа) вызвать дважды, на выходе система должна остаться в корректном состоянии (второй вызов не должен неожиданно что-то испортить). Для этого часто приходится проверять текущее состояние перед действием: если операция уже выполнена ранее, повторно не делать. Идемпотентность усложняет код, но без неё Saga ненадёжна.

Промежуточная неполная согласованность: Пока сага не закончена (или не откатилась при ошибке), система может быть в состоянии, когда часть сервисов уже применили изменения, а другие – ещё нет. Это состояние Eventually Consistent – «в конечном счёте согласованное». Нужно учитывать в бизнес-логике, что между шагами саги данные временно несогласованы. Например, поездка может быть создана, но оплата деньги еще не зарезервированы – другим сервисам нельзя считать его окончательно оформленным. Обычно эту проблему решают через статусы (pending, in progress, etc.) и сокрытие «полупроведённых» операций от конечных пользователей, пока сага не завершится.

Отладка и мониторинг: Разобраться, что происходит в распределённой саге, сложнее, чем в локальной транзакции. Логика разбросана по сервисам (особенно при

хореографии), последовательность действий не видна в одном месте. Необходимо внедрять корреляцию (например, уникальный идентификатор саги, передаваемый во все сообщения), вести детализированные логи, использовать распределённые трейсинг-системы. Только так можно отследить цепочку действий и выявить, где произошла ошибка.

Долговечность состояния саги: Если используется оркестратор, важно хранить прогресс саги в надёжном хранилище. В случае сбоя оркестратора он должен восстановиться и продолжить сагу (или откатить её). Это требует дополнительной работы – например, сохранять статус каждого шага в БД оркестрации. При хореографии долговечность обеспечивается брокером сообщений (сообщения не теряются) и идемпотентностью обработчиков.

Не подходит для коротких строго консистентных операций: Saga хороша для бизнес-процессов, где допустима некоторое время отложенная консистентность. Если же нужна мгновенная атомарность и недопустима промежуточная несогласованность, то Saga может быть слишком рискованным выбором.

Мы познакомились с паттернами, помогающими нам обеспечивать согласованность данных в распределенной системе на примере нашего сервиса такси. Далее рассмотрим теорию кэширования данных.

КЭШИРОВАНИЕ

В распределенных высоконагруженных системах часто применяют кэширование. Зачастую на собеседованиях по систему дизайну от вас ожидают грамотного использования кэша.
Сейчас мы разберем основные аспекты кэширования:

1. Узнаем что такое кэширование и как оно помогает сервисам держать нагрузку?
2. Поговорим о видах кэширования
3. Разберем основные алгоритмы взаимодействия с кэшом на уровне сервиса.
4. Поговорим об алгоритмах вытеснения данных из кэша
5. Разберем способы инвалидации данных в кэше
6. Также я расскажу зачем нужно версионировать и тегировать кэш
7. Поговорим о многомерном кэшировании

Для чего нужно кэширование?
- Кэширование помогает снизить latency сервисов на запись и чтение в зависимости от выбранного подхода.

- Благодаря кэшированию можно снизить нагрузку на сторонние сервисы
- Переиспользование вычислением - например вы реализуете рекомендательную систему, котоая по ночам создает наборы рекомендаций для всех пользователей сервиса. С помощью кэшей можно заранее рассчитать нужные пользователю данные и отдавать их почти мгновенно.

- Правильно настроенный подход к кэширование помогает системе жить при краткосрочных отказах. Актуально

Прежде чем погружаться в алгоритмы кэширования, определим ключевые термины.

Cache Hit — это когда запрошенные данные нашлись в кэше. Это наша цель.
Cache Miss — противоположность, данных нет и приходится обращаться к базе данных.

Hit Ratio — процент успешных обращений к кэшу. Если из 100 запросов 85 нашлись в кэше, hit ratio составляет 85%. Это главная метрика эффективности кэширования.

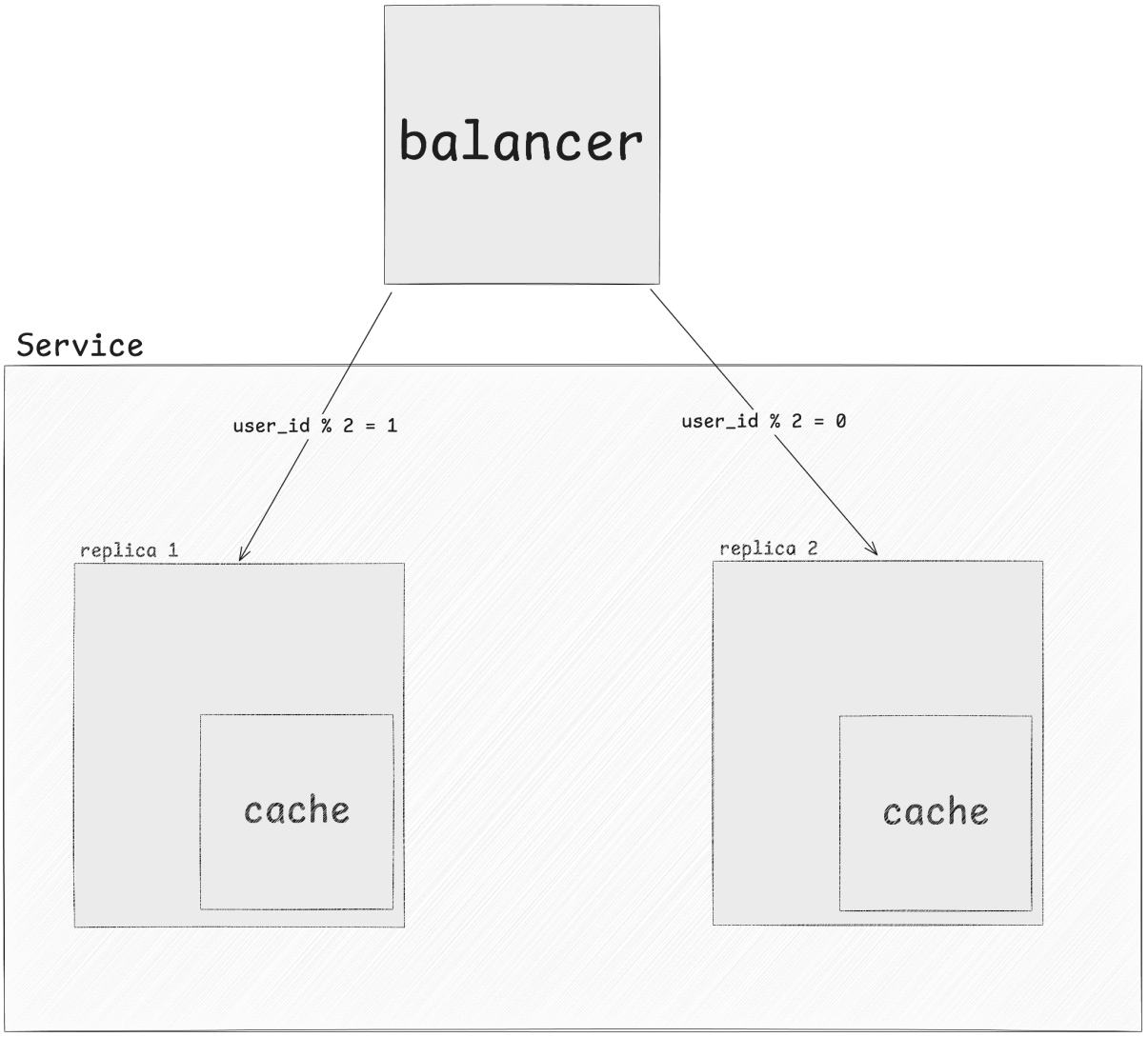
Hot key(Горячий ключ) - это ключ, на который приходится бОльшая часть запросов.

TTL- определяет, как долго данные живут в кэше. После истечения TTL данные считаются устаревшими и удаляются.

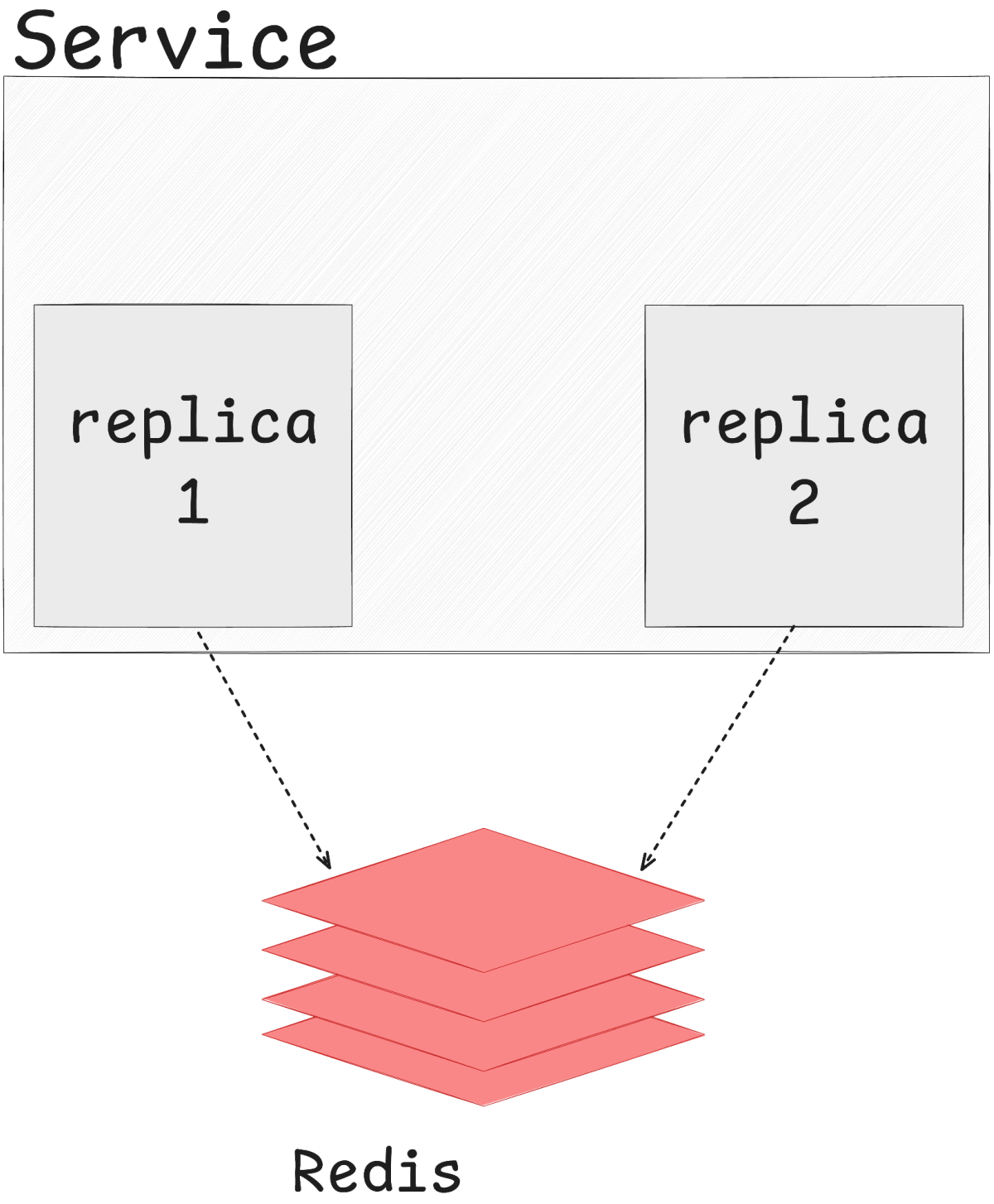
Прогрев кэша - процесс наполнения кэша данными

Инвалидация — процесс удаления данных из кэша, когда заканчивается место

Начнем с того, что разберемся с тем, какие есть виды кэширования на уровне приложения:
Внутреннее и внешнее кэширование:
Внутреннее кэширование использует структуры данных вашего языка программирования. Закешированные данные находятся в рамках процесса приложения. Таким образом мы получаем самый быстрый доступ к этим данным, нам не нужно делать никаких сетевых вызовов, соответственно мы избавляемся еще и от операций сериализации и десериализации данных. Однако при таких плюсах есть очевдиный минус - наше приложение становится stateful, то есть начинает хранить какое-то состояние. Если процесс завершится, то и все данные исчезнут. При реализации внутреннего кэширования придется решать проблему согласованной инвалидации данных во всех репликах, а также правильно настраивать балансировку трафика, чтобы в одну реплику всегда приходили запросы на данные, находящиеся в ее кэше.



Внешнее кэширование использует сторонние инструменты, такие как redis/valkey. Благодаря такому подходу наше приложение остается stateless, все реплики могут обращаться к одному внешнему кэшу. При таком подходе легко инвалидировать данные, ведь они находятся в одном месте. Расплачиваемся за такое удобство скоростью, сразу добавляется оверхед на подход по сети и сериализацию/десериализацию данных.

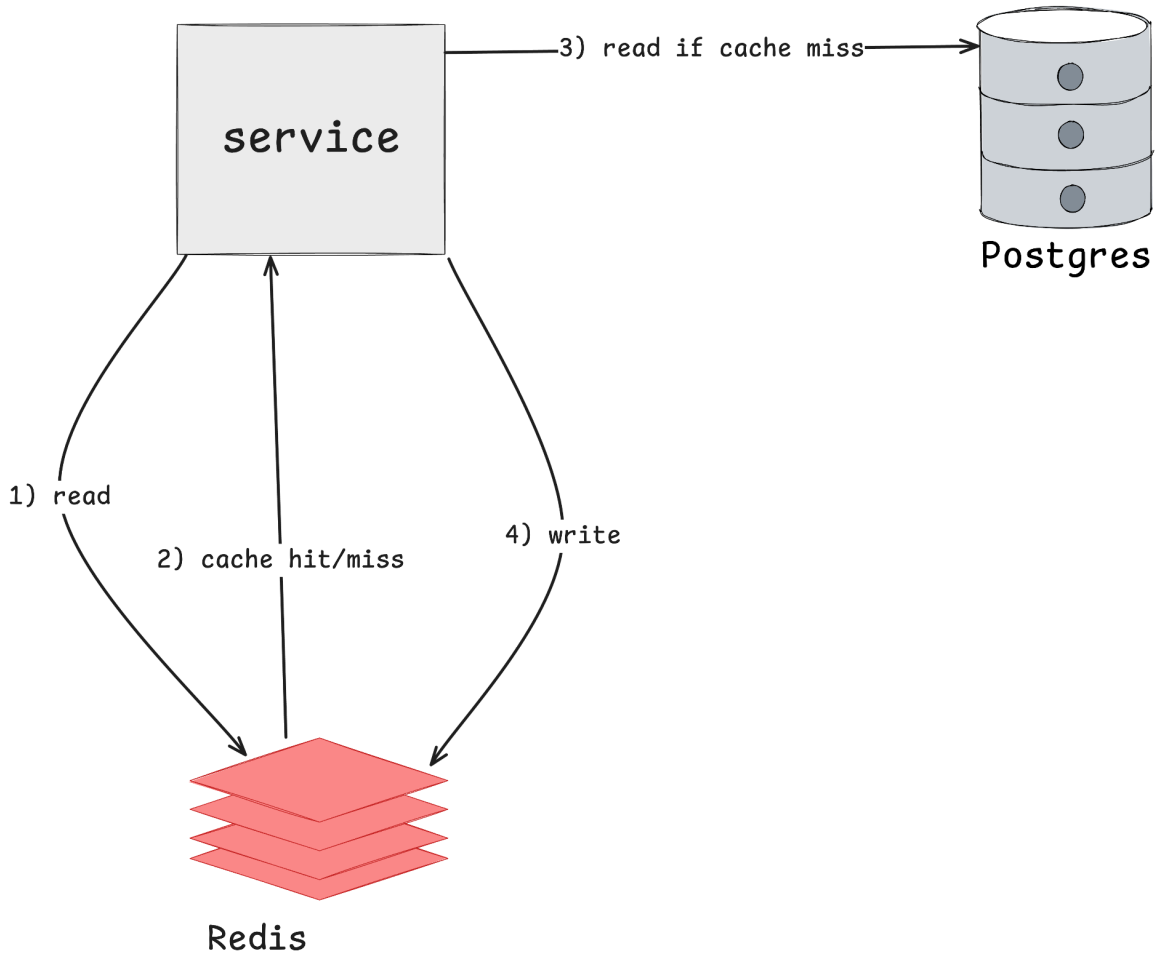


Давайте детально разберём каждый алгоритм уровня приложения, чтобы вы точно понимали, когда и что использовать."
Алгоритм Cache-Aside (Lazy Loading)

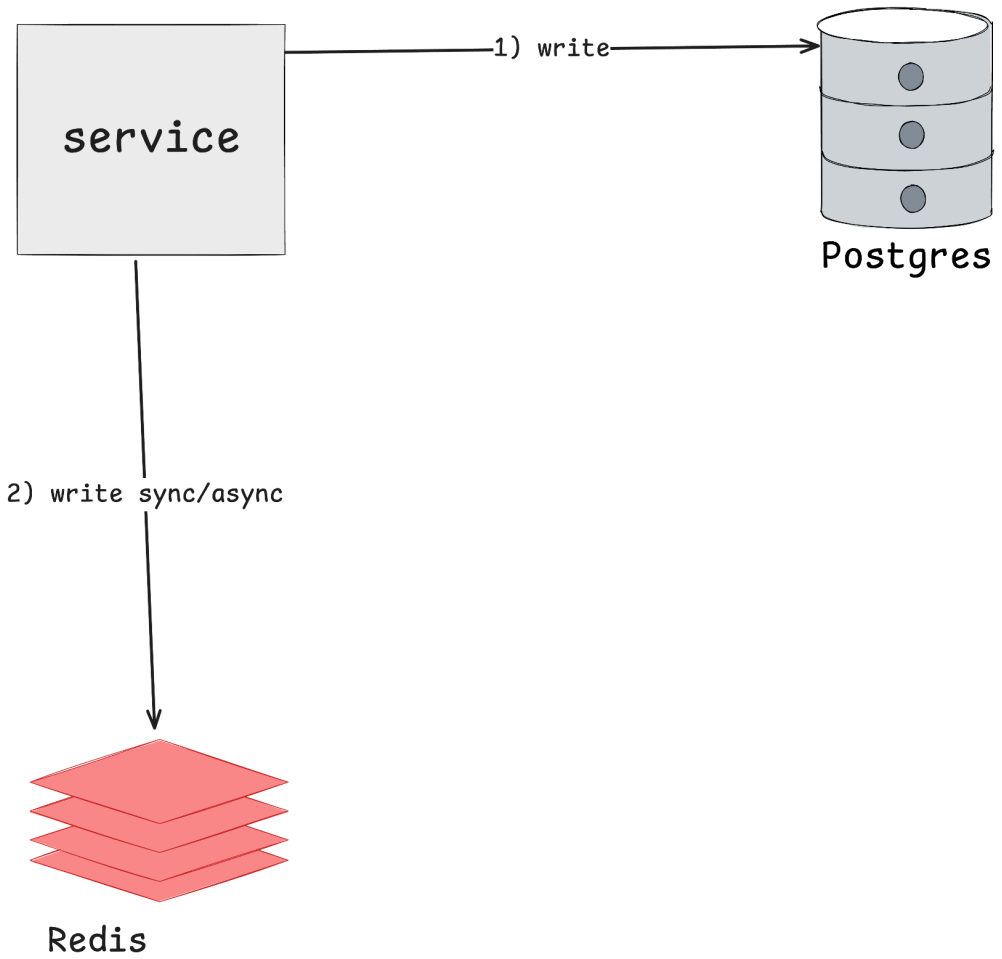
Cache-Aside — самый популярный паттерн. Приложение полностью контролирует процесс. При чтении сначала проверяем кэш. Если данных нет — идём в БД,

сохраняем результат в кэш и отдаём клиенту. При записи обновляем БД и инвалидируем кэш.

Read aside



Write aside



Преимущества этого подхода:
простота реализации, приложение решает что кэшировать.
простая инвалидация - при записях в бд, можно автоматически обновлять кэш
хорошо подходит для горячих ключей

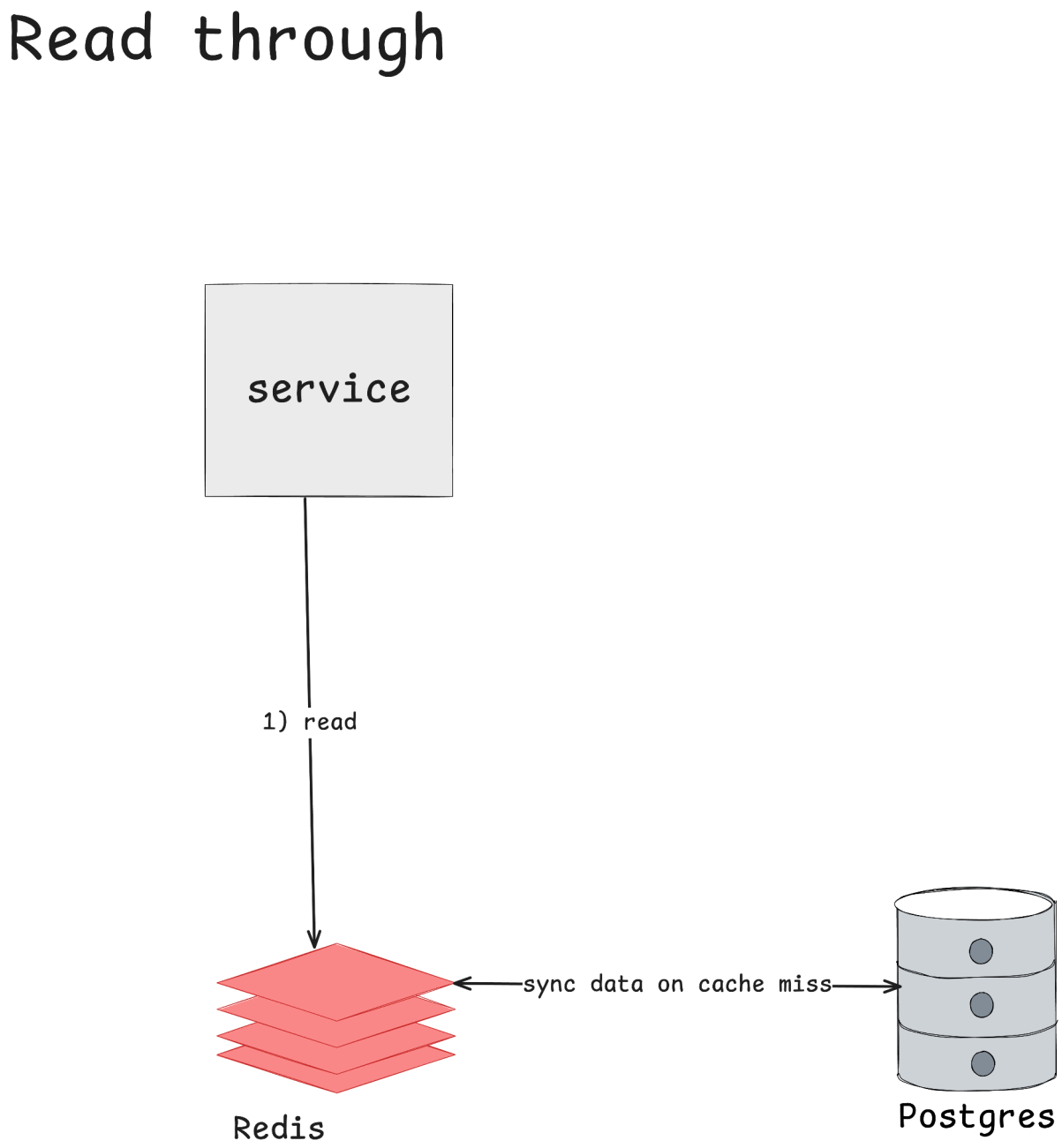
Недостатки:
дублирование логики кэширования
возможен **cache stampede** при одновременных запросах - это проблема, возникающая когда множество запросов одновременно пытаются получить данные, которые только что истекли в кэше.
холодный старт - первые запросы медленные, необходимо прогревать кэш
"

Когда применять:
При большой читающей нагрузке

Нужен контроль над тем, что кладется в кэш на уровне кода
Допустимо читать неактуальные данные

Далее обсудим Cache Through алгоритмы

Алгоритм Read-Through
Read-Through делает кэш прозрачным для приложения. Приложение обращается только к кэшу, а тот сам решает, когда и как загружать данные из БД. Это инкапсулирует логику кэширования и упрощает код приложения.



Преимущества:
Простота для клиента из-за наличия только одной точки входа

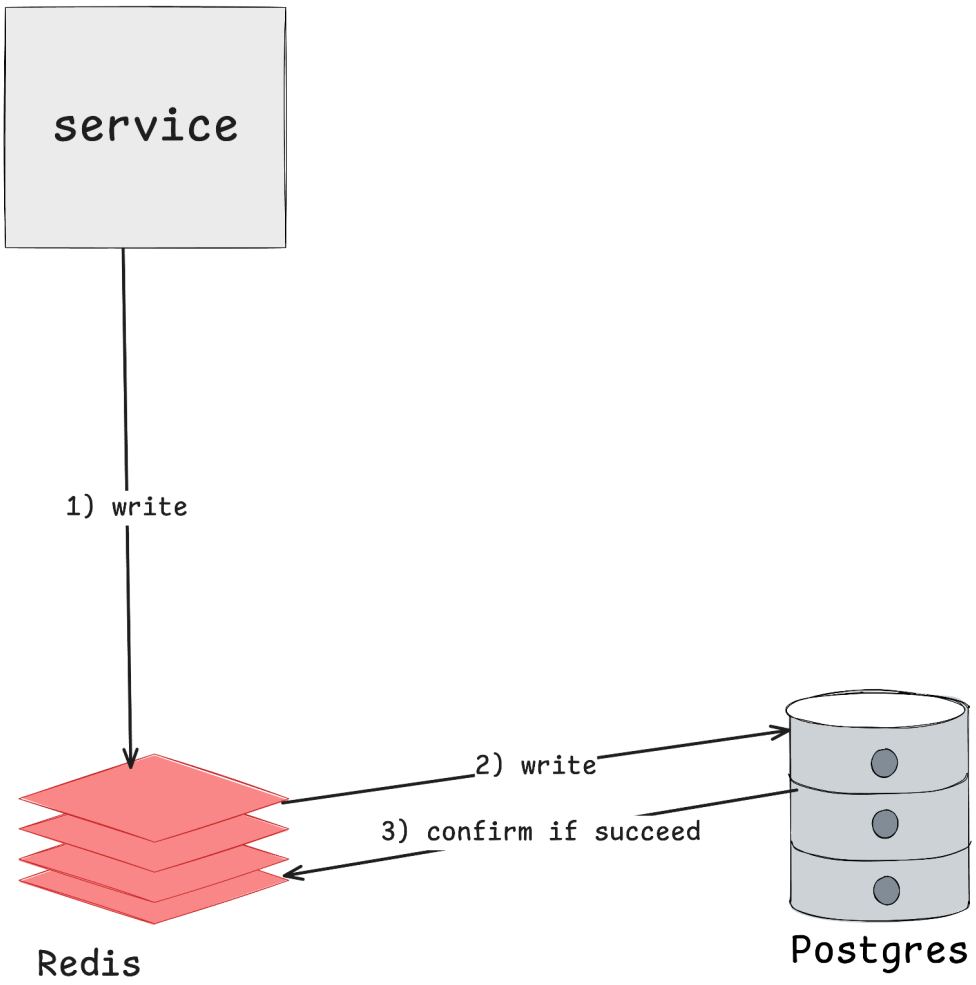
После прогрева кэша почти все запросы попадают в кэш

Недостатки:
Холодный старт - необходимо заполнять кэш перед стартом приложения
Конечная согласованность - без продуманной стратегии инвалидации в кэше могут оставаться старые или ошибочные данные

Когда использовать:
Чтения доминируют над записями
Есть повторяемость ключей, то есть одни и те же данные нужны разным пользователям
Допустима конечная согласованность

Алгоритм Write-Through
Write-Through обеспечивает строгую консистентность.
При записи приложение пишет в кэш, а кэш синхронно (в том же потоке) «проталкивает» изменение в базовый хранилище (origin) и подтверждает успех только если оба слоя записали данные.

Write through



Преимущества

кэш всегда согласован с основным хранилищем после успешной записи — меньше риска вернуть старые данные.

Недостатки:

Дополнительная задержка на запись

Чувствительность к отказам - при недоступности одного из узлов(бд или кэша) останавливается вся работа, поэтому нужно продумывать поведение системы при отказе одного из узлов.

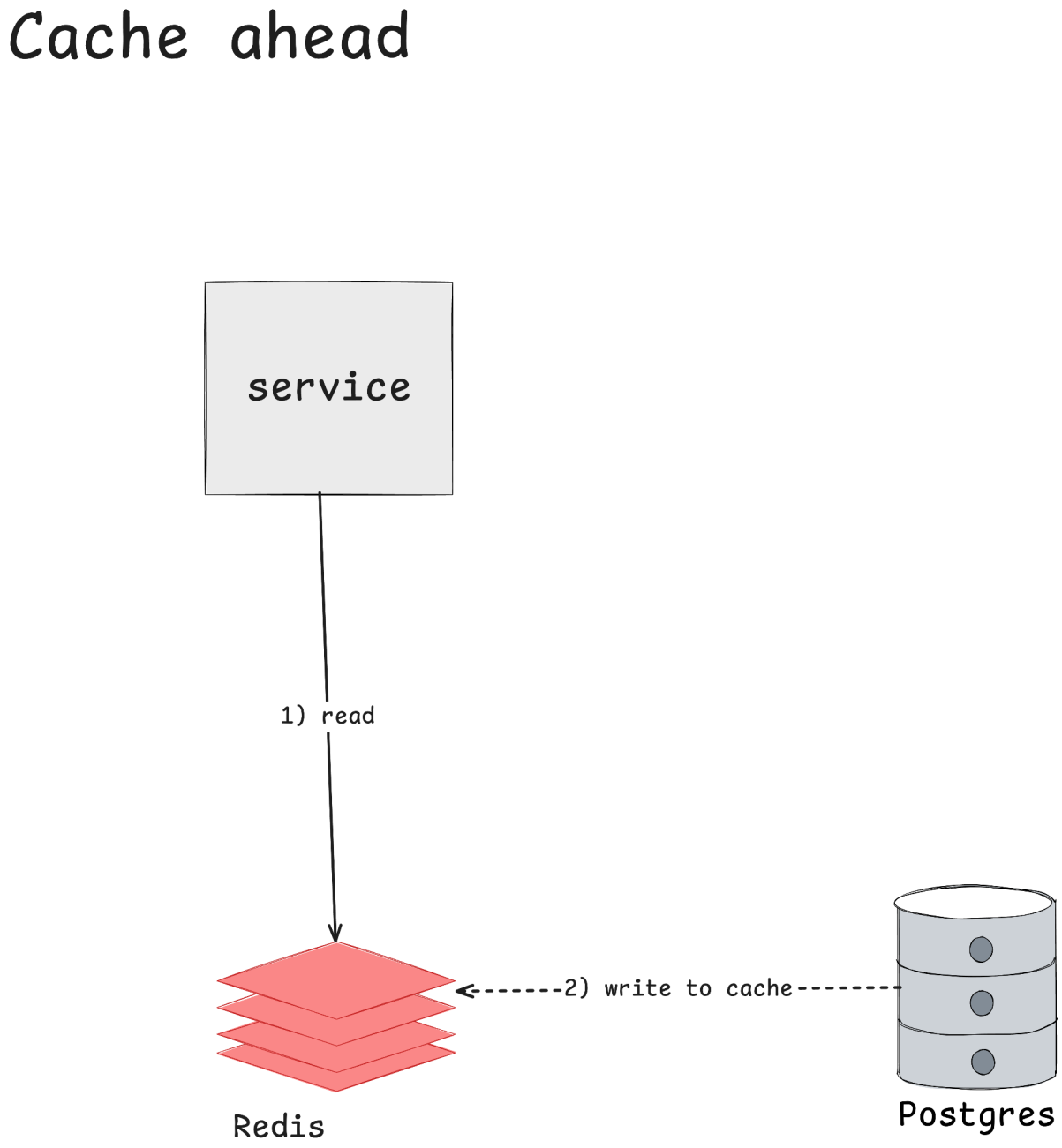
Когда применять:

При большой читающей нагрузке

Важна сильная согласованность на уровне кэша.

Критически важен порядок операций — сначала идем БД, потом кэш. Иначе при сбое БД в кэше окажутся несохранённые данные."

Алгоритм Cache-Ahead



Этот алгоритм стоит использовать в ситуациях, когда мы имеем предсказуемые паттерны доступа. Приложение ничего не знает о базе данных, работает только с кэшем, зачастую только на чтение. Например, мы знаем, что в нашей системе такси всегда нужно узнавать локацию водителей при заказе. Мы можем заранее наполнять кэш данными о водителях в определенном секторе города. Так мы всегда будем иметь актуальную информацию и читать данные почти без задержки.
Важно следить за тем, чтобы кэш был всегда прогрет и доступен, иначе наше приложение не сможет работать.

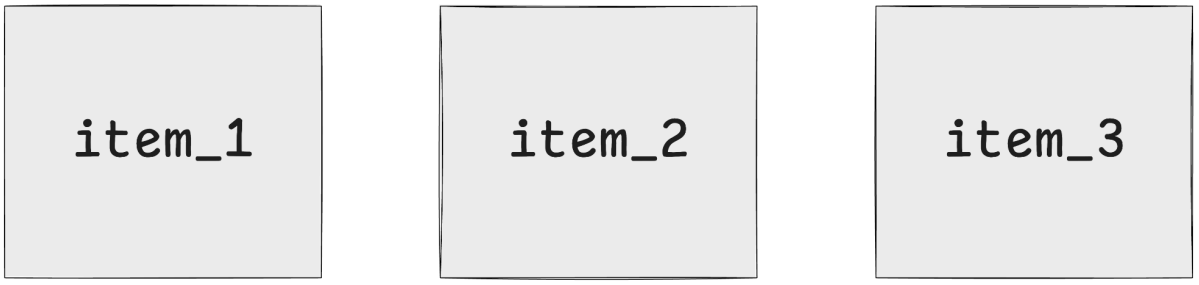
Вытеснение данных из кэша

Нам стоит помнить, что память не бесконечна. У кэша всегда должно быть ограничение по памяти. Поэтому существует процесс вытеснения данных из кэша. Рассмотрим базовые алгоритмы.

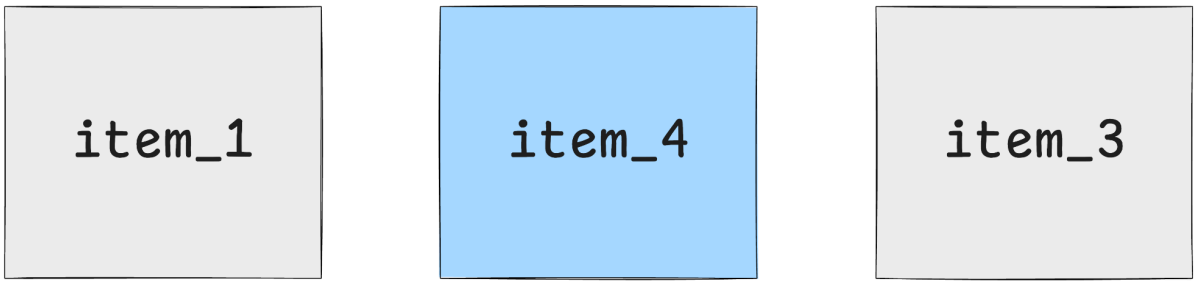
Random — случайный выбор. Просто выбираем случайный элемент для удаления. Работает хорошо при равномерном доступе к кэшу без горячих ключей.

Random

```
add: item_1
add: item_2
add: item_3
```



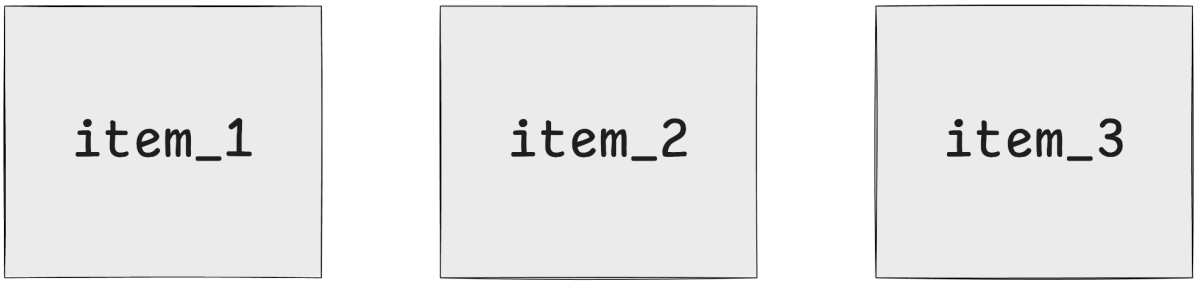

```
add: item_4
(remove random item)
```



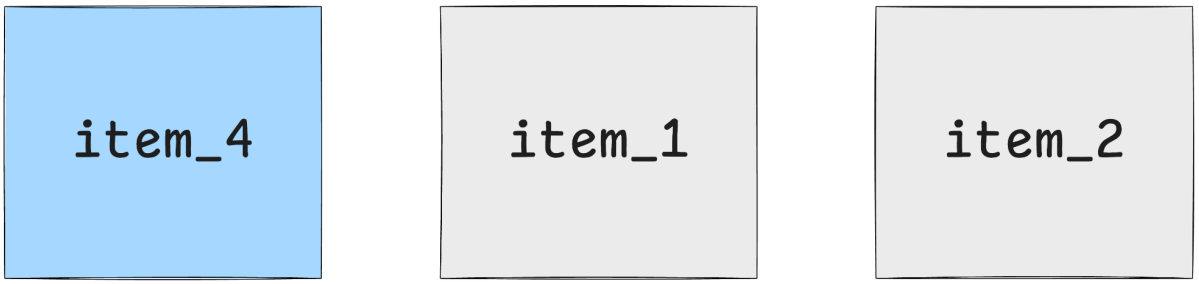
FIFO просто удаляет самые старые данные. Быстро, предсказуемо, но не учитывает паттерны доступа. Используется для кэша с TTL, где возраст важнее популярности.

FIFO(first in first out)

```
add: item_1
add: item_2
add: item_3
```



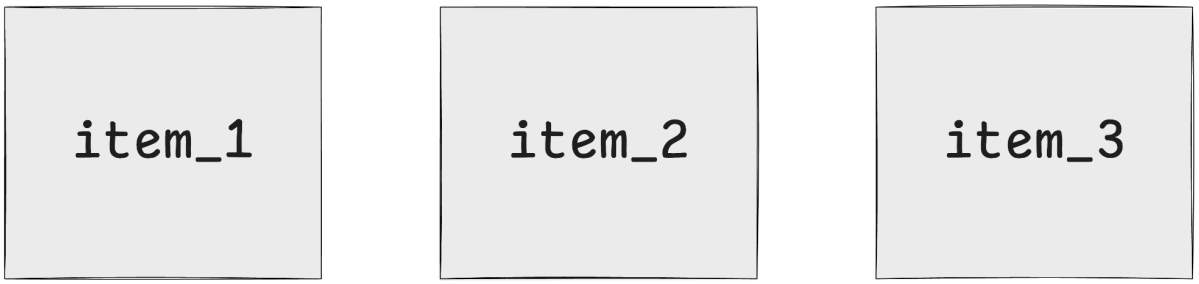

```
add: item_4
(remove item_1)
```



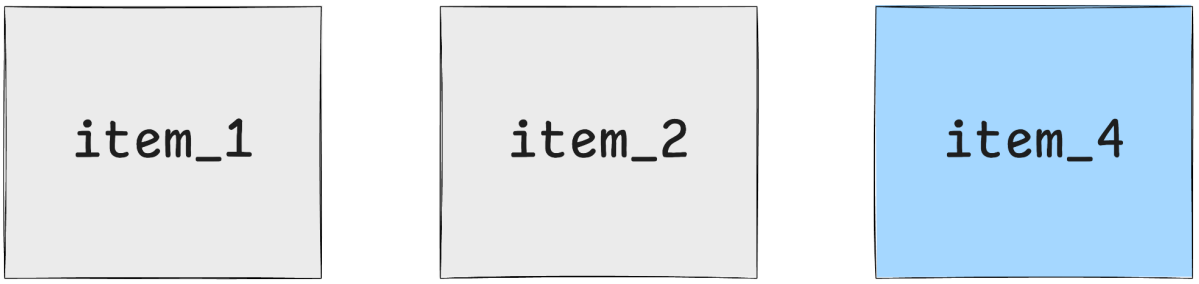
LIFO - по сути это стэк, удаляем самые новые данные.

LIFO(last in first out)

```
add: item_1
add: item_2
add: item_3
```



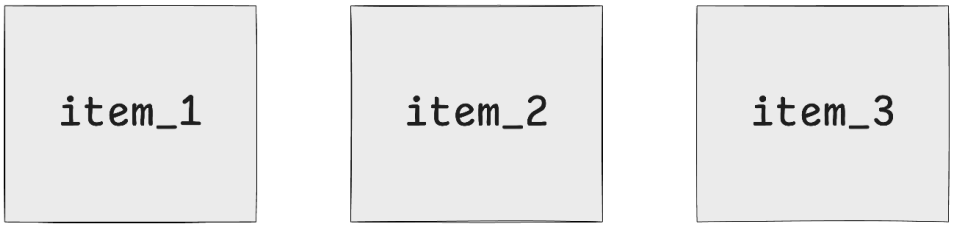

```
add: item_4
(remove item_3)
```



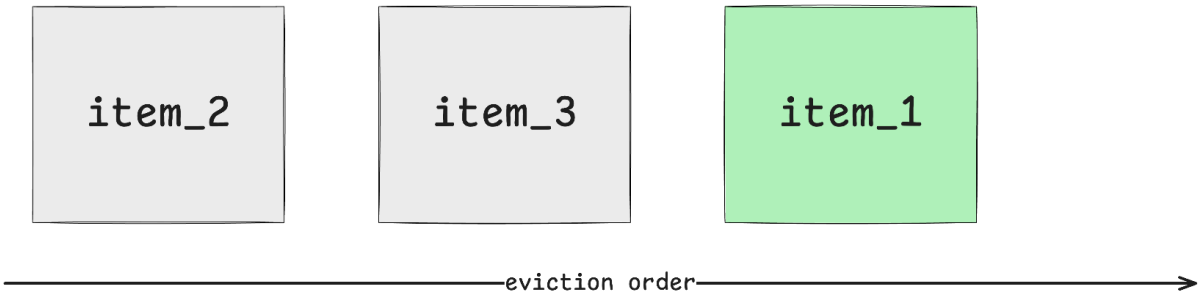
LRU — самый популярный алгоритм. Удаляются данные, к которым давно не обращались. Реализуется через двусвязный список и хэш-таблицу для O(1) операций. Идеально для кэша с равномерным доступом к данным.

LRU(least recently used)

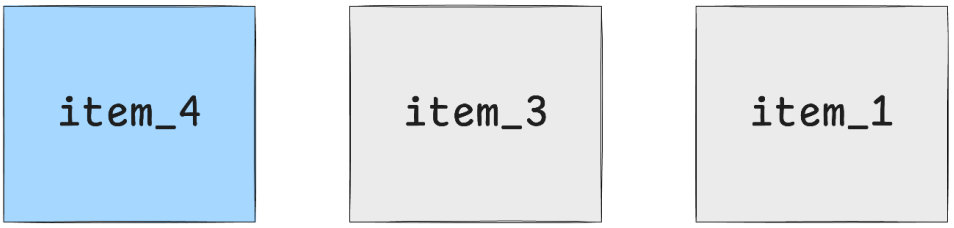
add: item_1
add: item_2
add: item_3



get: item_1



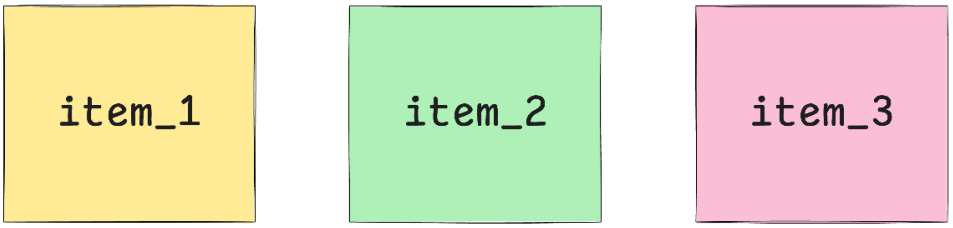
add: item_4
(remove item_2)



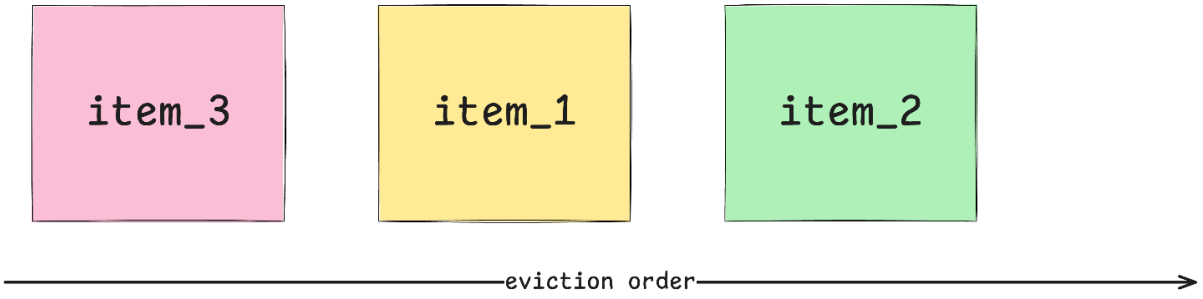
LFU отслеживает частоту обращений. Удаляются редко используемые данные. Проблема — популярные данные могут застрять в кэше навсегда.

LFU(least frequently used)

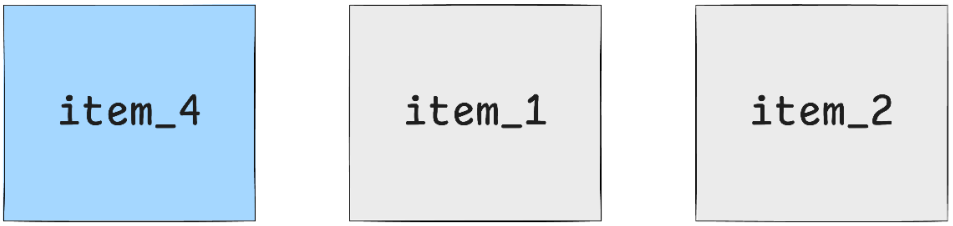
add: item_1
add: item_2
add: item_3



get: item_1 (10 times)
get: item_2 (100 times)
get: item_3 (5 times)



add: item_4
(remove item_3)



Инвалидация данных

Существует два основных подхода:
инвалидация по TTL и event-driven.

Во время инвалидации по TTL важно помнить, о паттернах заполнения кэша. Если вы одновременно заполняете кэш большим количеством данных с одинаковым TTL, то по истечению TTL вы столкнетесь с проблемой, при которой все запросы пойдут в базу данных, которая может не выдержать такой нагрузки. Это явление называется thundering herd problem. Для того, чтобы предотвратить эту проблему используют jitter вместе с ttl. По сути это просто добавление рандомного времени, для того, чтобы кэши не протухали одновременно.

Event driven инвалидация обычно реализуется через брокеры сообщений. Сервис слушает определенные события и удаляет данные из кэша, связанные с этим событием. Например мы хранили в кэше статус нашей саги, как только получили информацию о завершении последнего шага - удаляем данные из кэша.

Версионирование кэша

Все сервисы подвержены обновлению, но что происходит, когда при добавлении новой фичи мы обновляем модель данных, которую храним в кэше? Если ничего не делать и раскатить изменения, то пользователи не получат обновленные данные, т.к были закэшированы старые версии нашей модели. Можем представь наш сервис такси, представим, что вся информация о текущих тарифах хранится в кэше, соответственно сервис отдает данные из этого кэша. В новой фиче мы добавили новое поле в нашу модель, теперь мы добавили новое поле, отвечающее за наличие высокого спроса, фронтенд ожидает данные в новом формате, но приложение до сих пор отдает старую модель. Для того, чтобы решить эту проблему и актуализировать данные, мы должны добавить версии для нашего кэша. Обычно версию пишут как суффикс к ключу в кэше. Сервис с новой версией кэша, которая обычно выставляется в конфигурации сервиса попытается запросить данные, произойдет cache miss, соответсвенно ему придется обновить данные в кэше, взависимости от способа взаимодействия.

```
Cache versioning

add: item_1_v1
add: item_2_v1
get: item_1_v1 -> cache hit
get: item_2_v1 -> cache hit

New feature, version bumped to v2

get: item_1_v2 -> cache miss
```

Тегирование кэша

Бывают ситуации, когда в кэше хранятся связанные по смыслу данные, которые обязаны быть согласованными, но работаю с ними разные сервисы. Например у нас есть сервис каршеринга, в нем кэш в котором мы храним информацию о нашем автопарке. Сервис А хранит в кэше данные о марках автомобилей и их количестве, а сервис Б хранит всю информацию о нахождении конкретных автомобилей с разделением по городам/секторам. Внезапно мы решаем, что автомобиль марки aston martin не достоин находиться в нашем автопарке, и при новом обновлении выпиливаем его из сервиса А. Но видим такую ситуацию, информации на странице выбора авто больше нет в кэше, а вот позиции на карте автомобилей aston martin остались и отображаются в приложении. В случае, если мы закладывали такой сценарий заранее, нам стоило подумать о тегировании. Если на записях сервиса А и сервиса Б были навешаны одинаковые теги,эти данные были бы одновременно инвалидированы.

Многослойный кэш

Ранее мы рассматривали кэш уровня приложения, но на самом деле кэш выстраивается на разных уровнях, начиная от браузера и заканчивая кэшем базы данных. Расскажу о тех уровнях, которые стоит знать и всегда держать в уме при проектировании вашей системы:

1. Браузер кэширует статические файлы
2. Кэш на уровне прокси-сервера, самый яркий пример — это CDN
3. Кэш уровня приложения, который мы разбирали ранее
4. Кэш базы данных

Настроить согласованное поведение между всеми этими уровнями достаточно сложно, и мы не будем рассматривать, как это сделать в рамках текущего курса, однако вам будет полезно знать о существовании этих уровней. В высоконагруженных приложениях, особенно распределенных по разным регионам, применяются все возможные способы оптимизации работы приложения. Это нетривиальная задача, тут также нет серебряной пули, и при выборе подхода к работе с многоуровневым кэшем придется опираться на паттерны доступа к кэшу, которые предварительно необходимо выявить.

ЗАКЛЮЧЕНИЕ И ДОМАШНЕЕ ЗАДАНИЕ

В работе вы никогда не увидите идеальных систем, потому что их не бывает. Вам не нужно стремиться построить идеальную систему, гораздо важнее понимать: не нужно применять все паттерны сразу. Начните просто, усложняйте по мере роста.

Ключевое при проектировании систем - это заложить правильные абстракции, чтобы эволюция была возможна без полного переписывания системы."

ДОМАШНЕЕ ЗАДАНИЕ: ПРОЕКТИРОВАНИЕ СЕРВИСА ДОСТАВКИ ЕДЫ

Задача звучит так: у нас есть сервис доставки еды. Мы хотим спроектировать главный пользовательский путем: доставка заказа, которая включает в себя: проверку наличия блюда в конкретном ресторане, заполнение корзины, создание заказа, оплата заказа, назначение курьера. Какие гарантии согласованности стоить заложить заранее? Какие инварианты для данной системы вы выберете?

Определите все инварианты и распишите каким образом вы будете достигать выбранных гарантий согласованности.
Разбор домашки будет на бусти.

Удачи в выполнении домашнего задания! Напишите в комментария о том какая тема для вас была самой полезной во всей главе.

Глава 4. Системный дизайн в аналитике

Спикер: Артем

Введение

Давайте поговорим о том, как от роли, которую иногда называют «сборщиком требований», перейти к роли архитектора решений. И главное — зачем вам это вообще нужно?

Ключевой скачок для Middle-аналитика на пути к Senior-уровню — это переход от ответа на вопрос **«Что мы делаем?»** к способности самостоятельно отвечать на вопрос **«А как мы это будем делать?»**.

Это значит, что вы перестаёте быть просто *транслятором требований* — когда вы собрали информацию, формализовали её в User Story или Use Case, согласовали и передали на разработку. Вы становитесь тем, кто может эти требования технически обработать:

- составить архитектуру;
- спроектировать все интеграции;
- и выдать разработчикам такое техническое задание, по которому у них практически не останется вопросов. (когда говорю про выделенное желтым - надо показать картинку снизу)