БАЛАНСИРОВКА НАГРУЗКИ НА ПРИЛОЖЕНИЕ - ОТ ИНФРАСТРУКТУРЫ ДО БАЗЫ ДАННЫХ

Шевцов А.С.

Шевцов Алексей Сергеевич - ведущий инженер компании,
Material Bank,
Boca Raton, Florida, USA

Аннотация: *балансировка нагрузки не только предотвращает перегрузку серверов, но и улучшает и повышает доступность, обеспечивая стабильный бесшовный опыт для пользователей. В этом руководстве мы рассмотрим различные стратегии балансировки нагрузки на различных уровнях - от инфраструктуры до логики приложения.*

Бесперебойная работа приложения во многом зависит от того, насколько хорошо оно управляет нагрузкой. У вас может быть всего один экземпляр приложения с ограниченными ресурсами и огромной нагрузкой, которую необходимо сбалансировать. Или у вас может быть кластер с репликами приложения, или многосерверная конфигурация, и необходимо эффективно распределить трафик и рабочую нагрузку по ним.

Ключевые слова: *архитектура, высоконагруженные информационные системы, балансировка нагрузки, шаблоны проектирования информационных систем.*

Уровень сетевой и инфраструктурный

Эффективная балансировка нагрузки на этом уровне может значительно повлиять на производительность и масштабируемость приложения в целом.

DNS-Балансировка

Балансировка нагрузки DNS (Domain Name System) является одной из самых ранних и простых форм балансировки. В сущности, она работает на уровне инфраструктуры и переводит доменные имена в IP-адреса.

Сервер DNS может возвращать разные IP-адреса в циклическом порядке для каждого запроса, который он получает. В контексте микросервисов, каждый микросервис может иметь несколько экземпляров, работающих на разных серверах, каждый с уникальным IP-адресом. Балансировщик DNS может возвращать IP-адрес другого сервера для каждого нового запроса, гарантируя, что запросы распределяются между всеми доступными экземплярами.

Несколько характеристик DNS LB:

1. **Географическое распределение:** Балансировка нагрузки DNS может направлять трафик к ближайшему географически экземпляру приложения, минимизируя задержку и улучшая общую производительность.
2. **Обработка отказов:** Балансировка нагрузки DNS также может помочь при сбое. Если сервер выходит из строя, DNS может перестать направлять трафик на IP-адрес этого сервера и перераспределить его между оставшимися серверами.
3. **Масштабируемость:** По мере масштабирования системы и добавления новых экземпляров приложения, балансировщик DNS автоматически включает их в пул ресурсов для распределения трафика.

Безусловно, важно понимать и ограничения:

1. **Проблемы с кэшированием:** Ответы DNS могут кэшироваться в разных местах (например, на локальных машинах или промежуточных серверах ISP), что означает, что клиент может продолжать отправлять запросы на сервер, который не работает.
2. **Отсутствие осведомленности о нагрузке:** Балансировка нагрузки DNS не знает о нагрузке или емкости сервера; она просто перебирает IP-адреса в циклическом порядке.
3. **Несогласованность в сохранении сессии:** Если ваше приложение требует, чтобы клиент придерживался одного и того же сервера в течение сессии, балансировка нагрузки DNS может быть не лучшим выбором, поскольку она изначально не поддерживает сохранение сессии.

Балансировка нагрузки на уровне TCP

Балансировка нагрузки TCP (Transmission Control Protocol) работает на транспортном (4-м) уровне сетевой модели OSI. Она распределяет клиентские запросы на основе TCP-сессий, а не отдельных IP-пакетов, что делает ее более эффективной и надежной, чем другие типы балансировки нагрузки.

Вот как балансировка нагрузки TCP может помочь управлять рабочими нагрузками в среде микросервисов:

1. **Персистентность:** TCP Балансировщики отслеживают состояние TCP-соединений. Они могут гарантировать, что все пакеты сессии между клиентом и сервером отправляются на тот же сервер, даже если с IP-адресом назначения связано более одного сервера.

2. **Контроль состояния:** TCP Балансировщики могут периодически проверять состояние серверов и прекращать отправку трафика на любой сервер, который не проходит эти проверки. Это обеспечивает высокую доступность и надежность системы.

3. **Масштабируемость:** TCP Балансировщик позволяет легко масштабировать ваши сервисы. По мере добавления новых экземпляров сервиса, балансировщик нагрузки может автоматически распределять трафик на эти новые экземпляры.

И несколько ограничений:

1. **Отсутствие контекста приложения:** Поскольку балансировка нагрузки TCP работает на транспортном уровне, она не имеет никаких сведений о запросах и ответах HTTP. Это значит, что она не может принимать решения на основе содержимого сообщений HTTP.

2. **Смещение сессий:** Если сервер выходит из строя и снова начинает работать, сессии, которые были связаны с ним, могут переключиться на другие серверы, что приводит к неравномерному распределению нагрузки.

CDN

Сеть доставки контента (CDN) - еще один мощный инструмент для балансировки нагрузки путем предоставления статического контента пользователям от ближайшей географической точки присутствия (PoP), что сокращает задержку и снимает нагрузку с исходного сервера.

Уровень платформы и архитектуры (прикладной)

Платформа оркестрации сервисов

В **Kubernetes** балансировка нагрузки может выполняться на двух уровнях - среди подов (внутри) и снаружи - для доступа клиентов к сервисам. Сервисы внутри Kubernetes могут автоматически распределять запросы среди подходящих подов с использованием cluster-IP (внутреннего адреса), а Ingress-контроллеры или балансировщики нагрузки облачного провайдера могут обрабатывать внешний трафик.

Вот простой пример горизонтального масштабирования в Kubernetes. Предположим, у вас есть Deployment для вашего API-приложения:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-api
  template:
    metadata:
      labels:
        app: my-api
    spec:
      containers:
      - name: my-api
        image: my-api:1.0.0
```

Рис. 1. Пример Deployment-а приложения с тремя репликами.

Можно автоматически масштабировать это приложение при помощи HorizontalPodAutoscaler в Kubernetes:

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-api
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-api
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 80
```

Рис. 2. Пример Deployment-а с использованием HorizontalPodAutoscaler.

В этом примере кол-во подов будет расти до 10шт, стараясь поддерживать среднюю загрузку ЦПУ на уровне 80%.

Микросервисная архитектура

Микросервисная архитектура - это шаблон проектирования, в котором общее приложение разбивается на атомарные, слабо связанные службы. Каждый сервис отвечает за определенную функциональность и может разрабатываться, разворачиваться и масштабироваться независимо от других.

Например, различный функционал маркетплейса - аутентификация пользователей, обработка платежей и управление заказами - могут быть разделены на отдельные компоненты. Каждый из этих сервисов затем можно отдельно масштабировать в зависимости от нагрузки на него.

API Gateway

API Gateway служит единой точкой входа для клиентов, что делает его идеальным местом для балансировки нагрузки. Он может перенаправлять запросы с клиента к различным службам на бекенде на основе заранее определенных правил, эффективно распределяя рабочую нагрузку. Этот паттерн может помочь не только с балансировкой нагрузки, но и с балансировкой сложности в целом, например:

- **Разделение обязанностей** - запросы клиентских приложений будут отделены от особенностей реализации серверных.
- **Упрощение поддержки и изменения API** - клиентское приложение будет меньше знать о структуре ваших API, что сделает его более устойчивым к изменениям в этих API.
- **Повышенная безопасность** - конфиденциальная информация может быть скрыта, а ненужные данные на фронтенд могут быть опущены при отправке ответа на фронтенд. Абстракция затруднит атаку на приложение.

Service Meshes

Service Mesh — это настраиваемый инфраструктурный слой с низкой задержкой, разработанный для обработки большого объема запросов между программными интерфейсами приложения (API). Service Mesh гарантирует быстрое и надежное взаимодействие между контейнеризированными или эфемерными службами инфраструктуры приложений. Service Mesh предлагает функции, включающие обнаружение сервисов, балансировку нагрузки, шифрование, трассировку, аутентификацию и авторизацию, а также поддерживает паттерн автоматического отключения (circuit breaker).

Service Mesh обычно реализуется путем предоставления каждому экземпляру сервиса экземпляра прокси, который называется *Sidecar. Sidecar* обрабатывают коммуникации между сервисами, производят мониторинг и устраняют проблемы безопасности, то есть все, что может быть абстрагировано от отдельных сервисов. Таким образом, разработчики могут писать, поддерживать и обслуживать код приложения в сервисах, а системные администраторы могут работать с Service Mesh и запускать приложение. Примеры распространенных решений - Istio, Linkerd, Consul.

Взвешенный роутинг

Этот подход полезен для постепенной раскатки новой версии приложения. Можно направить небольшую часть трафика в новую версию, понаблюдать за поведением и метриками, и затем увеличивать его долю, если не возникает проблем. Ниже пример конфигурации Istio в Kubernetes:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-service
spec:
  hosts:
  - my-service
  http:
  - route:
    - destination:
        host: my-service
        subset: v1
      weight: 90
    - destination:
        host: my-service
        subset: v2
      weight: 10
```

Рис. 3. Пример конфигурации Istio в Kubernetes.

Здесь 90% трафика направляется в первую версию приложения и 10% во вторую.

Circuit Breaking (автоматическое отключение)

Это паттерн проектирования, используемый в современной разработке ПО для увеличения отказоустойчивости системы. В распределенной системе службы постоянно взаимодействуют друг с другом. Если вызываемый микросервис не работает должным образом или отвечает с задержкой, это может негативно сказаться на работе остальных служб. Механизм позволяет избежать подобной проблемы. Например, в Istio можно установить конфигурацию, которая прекратит отправку запросов к службе в случае её многократного сбоя:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-service
spec:
  host: my-service
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

Рис. 4. Пример конфигурации Istio с реакцией на сбои в приложении.

В данном примере, если приложение не ответит успешно один раз (consecutiveErrors: 1), оно будет исключено из пула балансировки на 3 минуты (baseEjectionTime: 3m).

Очереди сообщений

Очереди сообщений также могут помочь сбалансировать нагрузку в приложении. Они позволяют различным частям системы обмениваться информацией и обрабатывать операции асинхронно. Очередь сообщений предоставляет временное хранилище сообщений, когда целевая программа занята или не подключена.

В контексте балансировки нагрузки очереди сообщений могут помочь следующим образом:

1. **Обработка всплесков трафика:** В случае внезапных всплесков трафика, вместо перегрузки серверов, вы можете поставить запросы в очередь и обрабатывать их в удобном темпе.
2. **Разделение сервисов:** Очереди сообщений могут разделять ваши сервисы так, чтобы высокая нагрузка на один не влияла на другие. Если один микросервис перегружен, это не повлияет на другие, так как они общаются через очередь.
3. **Обеспечение сохранности данных:** Даже если сервис выходит из строя, данные не теряются, потому что они хранятся в очереди. Как только сервис восстановится, он может продолжить обработку запросов из очереди.

Например, рассмотрим платформу электронной коммерции, где пользователи делают заказы. Когда заказ оформлен, он может проходить через несколько этапов, таких как проверка наличия товара на складе, обработка платежа, подтверждение заказа и др.

Каждый из этих этапов может обрабатываться отдельными сервисами. Когда заказ оформлен, он помещается в очередь. Сервис инвентаризации выбирает заказ из очереди, проверяет наличие на складе, а затем возвращает его обратно в очередь. Затем сервис обработки платежей забирает заказ из очереди, обрабатывает платеж и так далее.

Таким образом, даже при внезапном росте числа заказов, система сможет справиться. Заказы просто будут ждать в очереди, пока они не будут обработаны, вместо того, чтобы перегружать систему.

Вот простой пример на Python с использованием RabbitMQ в качестве брокера сообщений:

```
import pika

# устанавливаем соединение с сервером RabbitMQ
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost')
)
channel = connection.channel()

# убеждаемся, что очередь существует
channel.queue_declare(queue='task_queue', durable=True)

# функция, которая будет обрабатывать сообщения из очереди
def callback(ch, method, properties, body):
    # имитируем тяжелую задачу, "засыпая" на 5 секунд
    time.sleep(5)
    print(f" [x] Получено {body}")

channel.basic_consume(queue='task_queue', on_message_callback=callback)

print(' [*] Ждем сообщения. Для выхода нажмите CTRL+C')
channel.start_consuming()
```

Рис. 5. Пример использования приложении RabbitMQ в качестве брокера сообщений.

В этом примере устанавливается соединение с сервером RabbitMQ, объявляется очередь под названием 'task_queue' и задается функция обратного вызова для обработки сообщений из очереди. Затем скрипт ожидает прихода сообщений в очередь и обрабатывает их по одному.

В случае высокой нагрузки, сообщения будут ждать в очереди, пока работник не будет готов их обработать, тем самым эффективно балансируя нагрузку.

Уровень приложения

Логика работы приложения также может сильно способствовать правильному распределению нагрузки на него или на взаимодействующие с ним сервисы.

Очередь входящих запросов

Стратегия в том, чтобы все входящие запросы сначала добавлять в очередь внутри приложения. И затем выполнять их с приемлемой для себя скоростью.

Простой пример с использованием массива для хранения очереди:

```
let queue: string[] = [];

queue.push('request1');
queue.push('request2');

let request = queue.shift();
console.log(`Processing: ${request}`);
```

Рис. 6. Пример использования массива для обработки очереди запросов.

Несколько процессов или потоков приложения

Можно распределять обработку запросов между несколькими потоками или процессами. Например, в Node.js можно использовать модуль ***cluster*** или модуль ***worker_threads*** для создания рабочих процессов/потоков.

В этом примере, главный процесс создает рабочий процесс для каждого ядра CPU. Каждый рабочий процесс затем самостоятельно обрабатывает запросы, эффективно распределяя нагрузку.

```
import cluster from 'cluster'
import os from 'os'

if (cluster.isMaster) {
  const numCPUs = os.cpus().length
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork()
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died`)
  });
} else {
  // Здесь вокрер-процесс может обрабатывать запросы
  // Например, брать из из очереди и выполнять
  console.log(`Worker ${process.pid} started`)
}
```

Рис. 7. Пример создания рабочих процессов для каждого ядра CPU при помощи модуля cluster в NodeJS.

Механизмы ограничения потока входящих запросов

Логику приложения можно организовать таким образом, что она будет обрабатывать лишь допустимый для себя поток запросов, а при достижении лимита начнет отвечать внешним системам специальным сигналом с запросом “сбавить частоту запросов”. Вот пример простого ограничителя частоты запросов:

```
class RateLimiter {
  private requests: number = 0;
  private lastReset: number = Date.now();

  increment() {
    this.requests += 1;
  }

  checkRateLimit() {
    const now = Date.now();
    // Сбрасываем счетчик запросов каждую минуту
    if (now - this.lastReset > 60000) {
      this.requests = 0;
      this.lastReset = now;
    }

    // Больше 1000 запросов за последнюю минуту - возвращаем true
    return this.requests > 1000;
  }
}

const rateLimiter = new RateLimiter();

// Для каждого входящего запроса
for (let i = 0; i < 2000; i++) {
  rateLimiter.increment();
  if (rateLimiter.checkRateLimit()) {
    console.log('Rate limit exceeded');
    // Здесь можно ответить 429 статусом, чтобы клиент замедлил поток запросов
    break;
  } else {
    // Обрабатываем запрос если лимит не превышен
  }
}
```

Рис. 8. Пример программного ограничителя потока запросов.

В этом примере мы ограничиваем пропускную способность до 1000 запросов в минуту. Если лимит превышен, мы выводим сообщение о том, что скоростной лимит превышен. В реальной ситуации ответим клиенту кодом HTTP 429 и заголовком Retry-After, указывающим, когда можно пытаться повторить запрос.

Выравнивание потока исходящих запросов

Можно “сглаживать” поток запросов из исходящей точки. Например, если вы знаете приложение создает неравномерные группы исходящих запросов, можно добавить ограничения вроде “выполняем максимум N запросов в одно время”. Для примера можно взглянуть на библиотеки вроде ***promise-parallel-throttle***.

В примере ниже пробуем распределить во времени отправку большого числа обновлений продуктов в небольшие пачки, а затем ограничиваем параллелизм запросов на обновление к другому сервису до максимум 5 пачек в одно время.

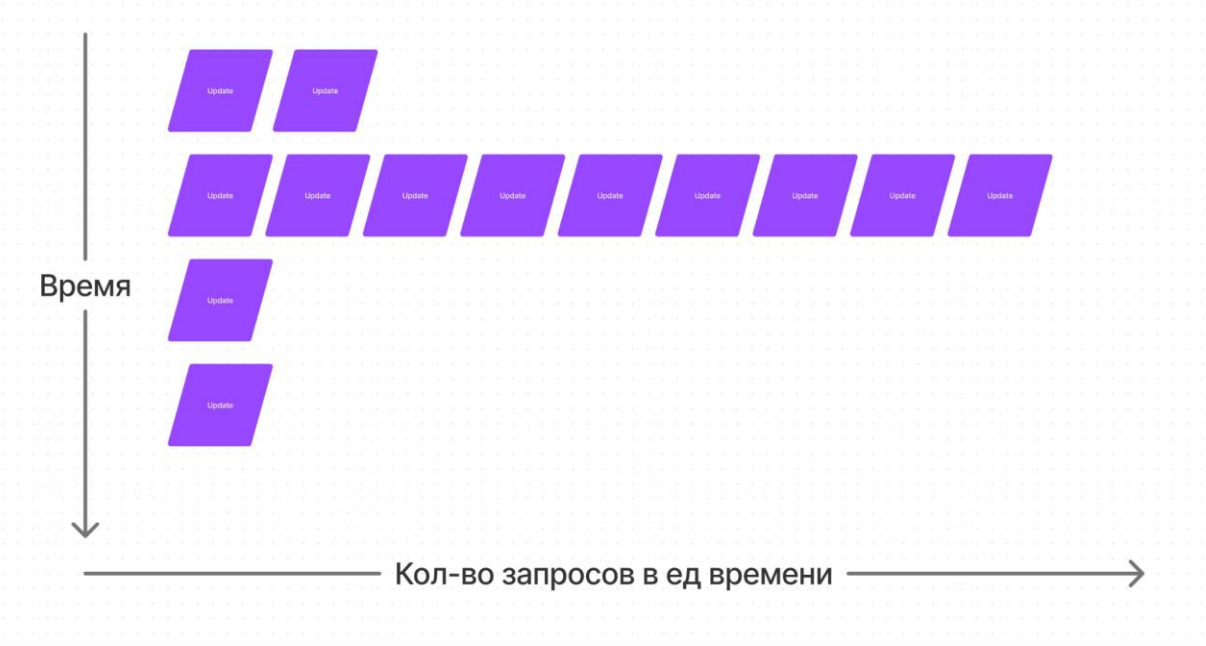

Рис.9. До выравнивания - могут быть спайки исходящих запросов, перегружающие другие приложения.

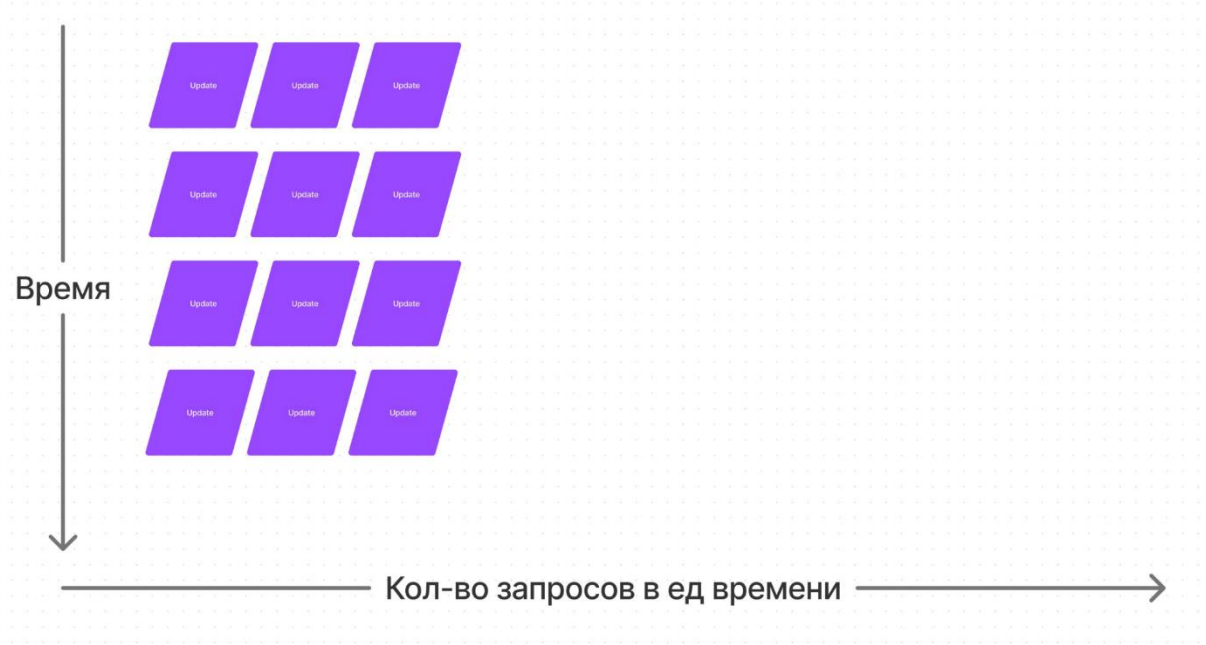

Рис. 10. После выравнивания - поток исходящих запросов сглаживается.

```
import * as Throttle from 'promise-parallel-throttle';

export const updateProducts = async (
  products: Product[],
) => {
  // Нарезаем длинный массив в пачки по 20 продуктов
  // Выполняем запросы на обновление этими пачками, максимум по 5 параллельно
  const requestPromises = sliceArray( products, 20 ).map(( productSlice ) => {
    return () => updateProductsRequest( productSlice )
  })

  await Throttle.all( requestPromises, { maxInProgress: 5 } )
}
```

Рис. 11. Пример программного сглаживания исходящего потока запросов.

Уровень баз данных

На уровне баз данных также существует ряд архитектурных приемов, которые помогут справляться с большим объемом запросов на чтение или запись данных.

Шардирование базы данных

Шардирование - это способ дробления большой базы данных на меньшие управляемые части, называемые "шардами". Каждый шард хранится на отдельном сервере для распределения нагрузки. Шарды обычно распределены по нескольким машинам, расположенным в разных физических местах.

Для приложения большого масштаба эффект от шардирования может быть значительным:

1. Растет производительность: Поскольку данные распределены по нескольким машинам, операции чтения/записи могут выполняться одновременно.

2. Растет доступность: Если данные правильно распределены, сбой в одном шарде не влияет на доступность других.

3. Растет масштабируемость: Шардирование позволяет осуществить горизонтальное масштабирование (добавление в сеть большего числа машин для управления увеличенной нагрузкой). По мере роста вашего приложения, вы можете добавлять больше шардов для обработки большего объема данных.

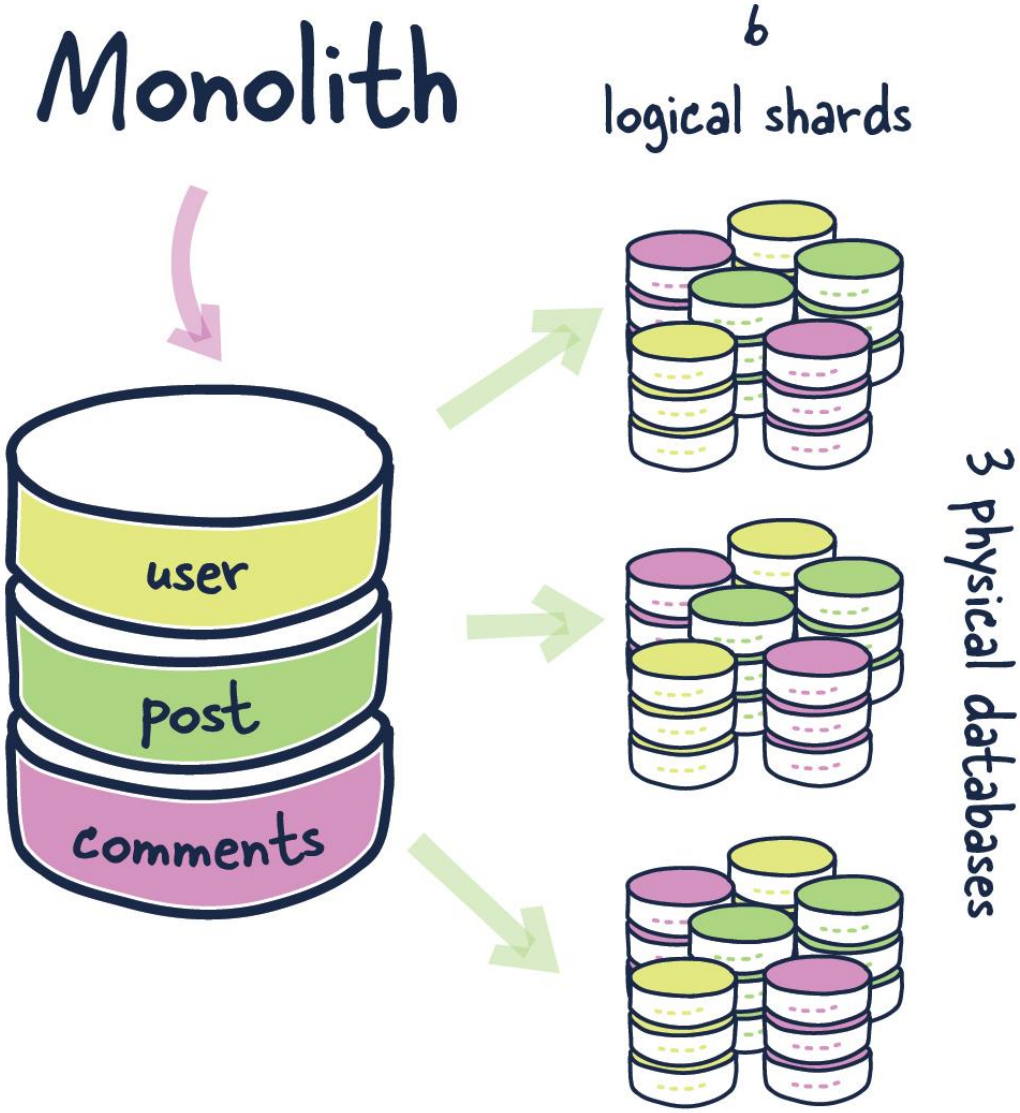

Рис. 12. Логическое разделение базы данных на шарды.

Вместе с шардированием возникнут и новые сложности, которым нужно будет уделить внимание. С шардированной базой данных мы сталкиваемся со сложностью управления множеством шардов, сложностью выполнения транзакций или агрегационных запросов между шардами, а также необходимостью хорошо продуманного ключа шардирования, чтобы избежать неравномерного распределения данных.

Практический пример: маркетплейс

Рассмотрим пример маркетплейса с миллионами пользователей, продуктов и транзакций. База данных может стать узким местом по производительности по мере роста платформы, замедляя операции.

Чтобы решить эту проблему, мы могли бы разделить базу данных на шарды на основе идентификаторов пользователей. Каждый шард может хранить данные для подмножества пользователей, включая информацию о их профиле, историю заказов, корзину и т.д. Например, у нас может быть:

- Шард 1: Пользователи с ID от 1 до 1 миллиона
- Шард 2: Пользователи с ID от 1 миллиона до 2 миллионов
- и так далее

Таким образом, когда пользователи взаимодействуют с платформой, их запросы направляются к соответствующему шарду. Это приводит к более быстрому выполнению запросов, поскольку каждый шард имеет меньше данных для обработки. Кроме того, по мере роста платформы можно добавлять больше шардов, что обеспечивает отличную масштабируемость.

Разделение операций чтения и записи

CQRS (Command Query Responsibility Segregation - сегрегация ответственности за команды и запросы), - это архитектурный шаблон, при котором операции чтения и записи разделяются, часто на разные серверы или кластеры. Этот подход особенно полезен для приложений с большими нагрузками на чтение и запись, поскольку он позволяет оптимизировать каждый тип операций независимо.

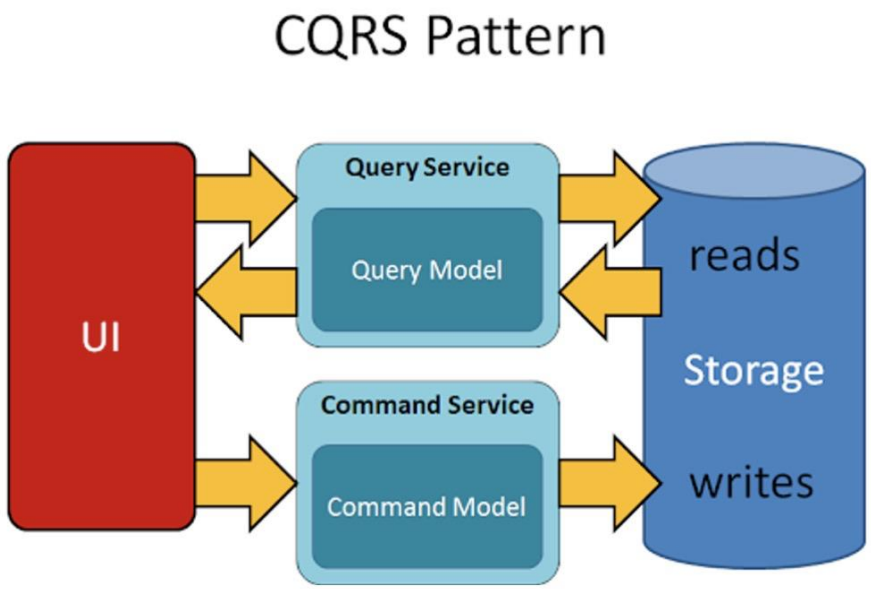

Рис. 13. Общая схема шаблона CQRS.

Преимущества здесь имеют сходные эффекты с техникой шардирования базы данных - легкое горизонтальное масштабирование и повышенная доступность. Кроме того, разные системы баз данных предназначены для более эффективной обработки операций чтения или записи. С разделением чтения и записи, вы можете выбрать разные системы для операций чтения и записи, что повышает общую производительность.

И, конечно, минусы - поддержание согласованности данных между серверами чтения и записи, и обработка задержек между обновлением сервера записи и отражением изменений на сервере чтения.

Практический пример: социальная медиа-платформа

Рассмотрим социальную медиа-платформу, где пользователи часто читают сообщения (например, прокручивают ленту новостей), но операции записи (например, создание нового сообщения) происходят реже. Операции чтения значительно превышают операции записи, но операции записи обычно требуют больше ресурсов.

Чтобы управлять этим, платформа может реализовать разделение чтения и записи. Операции записи, такие как создание нового сообщения или обновление профиля пользователя, могут направляться на специальный сервер записи. Этот сервер может быть оптимизирован для эффективной и надежной обработки операций записи.

В то же время операции чтения, такие как отображение ленты новостей пользователя или показ профиля пользователя, могут направляться на кластер серверов чтения. Эти серверы могут быть оптимизированы для быстрой обработки больших объемов операций чтения и могут масштабироваться для обработки высоких нагрузок.

Cache-Aside

Cache-aside - это шаблон кэширования, при котором приложение старается читать данные из кэша и записывать их в кэш при промахе. Этот подход позволяет простым образом контролировать, что должно кэшироваться и на какой срок, что делает его подходящим для многих приложений.

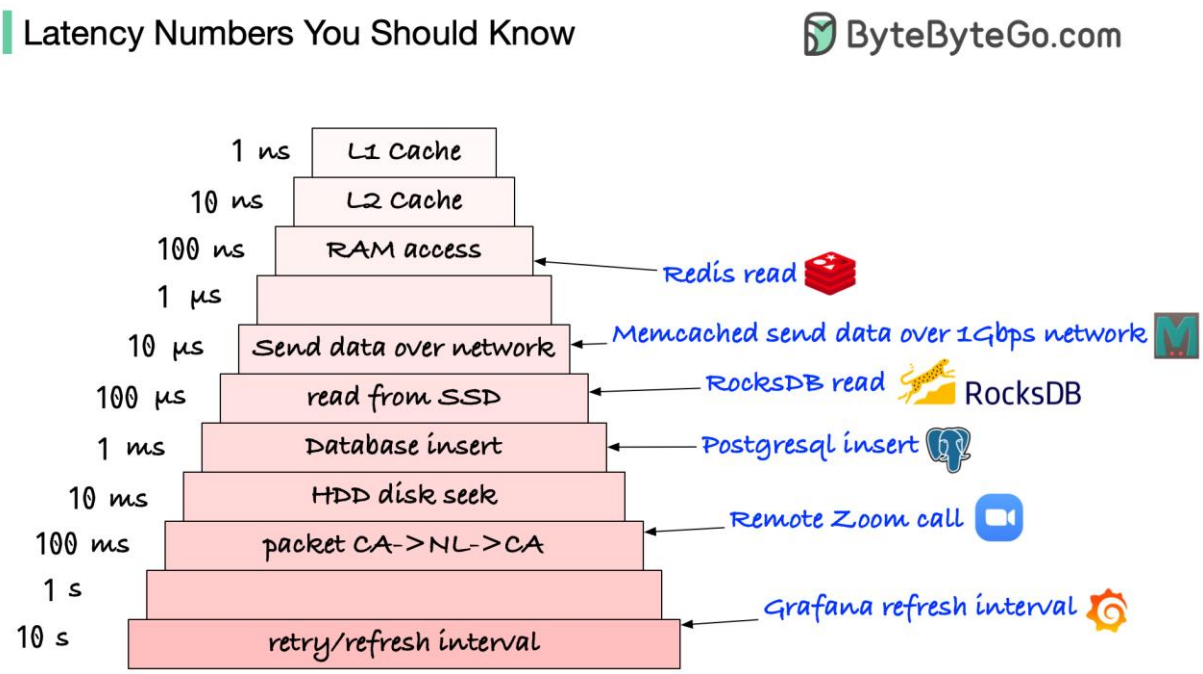

Рис. 14. Чтение данных из Redis (RAM) на порядок быстрее чтения из базы данных на HDD или SSD диске.

Основные этапы работы cache-aside:

1. Чтение данных: Когда приложению требуется прочитать данные, оно сначала пытается извлечь их из кэша. Если данные найдены (попадание в кэш), они немедленно возвращаются, что снижает нагрузку на базу данных и общее время ответа. Если данные не найдены в кэше, приложение извлекает их из базы данных, помещает их в кэш для будущих запросов, а затем возвращает.

2. Запись данных: Когда приложение записывает данные, оно записывает их напрямую в базу данных. Кроме того, чтобы поддерживать согласованность кэша, оно должно также сбросить любую кэшированную версию этих данных.

3. Политика вытеснения из кэша: Реализация эффективной политики вытеснения из кэша, такой как Least Recently Used (LRU) или Least Frequently Used (LFU), обеспечивает сохранение наиболее ценных данных в кэше, когда место ограничено.

При использовании техники cache-aside важно адекватно обрабатывать промахи кэша, чтобы избежать перегрузки базы данных запросами, и эффективно управлять сбросом кэша для обеспечения согласованности данных.

Практический пример: интернет-ритейлер

Рассмотрим крупного интернет-ритейлера. Некоторые товары более популярны и часто просматриваются клиентами, что приводит к большому числу операций чтения из базы данных. Стратегия cache-aside может помочь эффективно управлять этой нагрузкой.

Когда клиент запрашивает просмотр товара, приложение сначала проверяет кэш. Если товар находится в кэше, он немедленно возвращается клиенту. Если его нет в кэше, приложение извлекает товар из базы данных, сохраняет его в кэш для будущих запросов, а затем возвращает его клиенту.

В случае обновления информации о товаре приложение записывает новые данные в базу данных и сбрасывает кэшированный товар, обеспечивая тем самым, что будущие чтения будут извлекать обновленные данные.

Используя стратегию cache-aside, ритейлер может обеспечить быстрое время ответа для часто просматриваемых товаров и эффективно управлять нагрузкой на свою базу данных.

Заключение

Выбор правильной стратегии балансировки нагрузки может стать ключевым моментом для производительности, масштабируемости и удобства использования вашего приложения. Важно помнить, что нет единой верной для всех стратегии. Потребности вашего приложения, уже существующая архитектура и инфраструктура, требования к масштабируемости продукта - все эти факторы играют роль в определении нужной именно для вас стратегии. И понимая, как можно балансировать нагрузку на приложение на разных уровнях, вы сможете подобрать верное решение для вашего контекста.

Помните, что каждая система имеет свои уникальные проблемы и требования. Таким образом, ключ к успеху лежит не в принятии самой передовой стратегии, а в определении и реализации тех, которые наилучшим образом отвечают потребностям вашего приложения.

Список литературы

1. *Martin Kleppmann* Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems // O'Reilly Media, 2017.
2. *Alex Xu* System Design Interview – An insider's guide // Independently published, 2020.
3. *Joe Reis* Fundamentals of Data Engineering: Plan and Build Robust Data Systems // O'Reilly Media, 2022.