в зимних условиях. Томск: Издательство ТГАСУ, 2014. -412с.
6. В.Д. Копылов. Устройство монолитных бетонных конструкций при отрицательных температурах среды М.: Изд-во АСВ, 2014. - 184с.

УДК 62

Филисов Д.А.
Руководитель команды разработки Grid Dynamics
Белград, Сербия

ВЫСВОБОЖДЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ: ГЛУБОКОЕ ПОГРУЖЕНИЕ В ПРИЛОЖЕНИЯ С ВЫСОКОЙ НАГРУЗКОЙ

Аннотация

В этой статье рассматривается область параллелизма в серверных архитектурах и его оптимизация для сценариев с высоким трафиком. Цель состоит в том, чтобы предоставить разработчикам всестороннее представление о стратегиях и методах, которые можно использовать для эффективной обработки больших объемов трафика, обеспечивая при этом оптимальную производительность, оперативность реагирования и масштабируемость.

Статья начинается с содержательного введения, в котором подчеркивается важность серверной архитектуры, способной управлять интенсивным трафиком от одновременных пользователей. Это подчеркивает необходимость того, чтобы разработчики оптимизировали свою серверную инфраструктуру для удовлетворения требований высокой активности пользователей.

Далее в статье исследуется концепция параллельной обработки, управляемой событиями, демонстрируются практические примеры кода на JavaScript и Node.js. Это демонстрирует, как разработчики могут использовать функциональные блоки и возможности цикла обработки событий для эффективного управления выполнением кода.

В статье далее рассматривается дедупликация запросов как метод оптимизации для уменьшения избыточных запросов и улучшения воспринимаемой задержки. В нем освещаются сценарии, в которых выполняется несколько идентичных запросов, и преимущества дедупликации запросов при обработке "горячих" данных.

Кроме того, в статье обсуждается использование пакетной записи в качестве еще одной эффективной стратегии параллелизма на стороне сервера. В нем рассматриваются преимущества объединения нескольких операций записи в пакеты, снижающие нагрузку как на приложение, так и на базу данных.

В заключение статьи подчеркиваются преимущества разработки кода, эффективно использующего параллелизм. Это подчеркивает важность выбора бессерверных вычислительных решений для использования параллелизма на стороне приложения, позволяющего внедрять эффективные шаблоны, которые полностью используют параллелизм.

Понимая и внедряя концепции и стратегии, представленные в этой статье, разработчики могут раскрыть неиспользованный потенциал своих приложений, добиваясь повышения производительности, масштабируемости и отзывчивости в сценариях с высоким трафиком.

Ключевые слова:
параллелизм, архитектура сервера, высокий трафик, оптимизация, обработка на основе событий, дедупликация запросов, пакетная запись, производительность приложений, масштабируемость, отзывчивость.

PERFORMANCE UNLEASHED: A DEEP DIVE INTO HIGH LOAD APPLICATIONS

Abstract

This article delves into the realm of parallelism in server architectures and its optimization for high-traffic scenarios. The goal is to provide developers with a comprehensive understanding of the strategies and techniques that can be employed to efficiently handle large volumes of traffic while ensuring optimal performance, responsiveness, and scalability.

The article begins with an insightful introduction that emphasizes the importance of server-side architecture capable of managing intensive traffic from concurrent users. It highlights the need for developers to optimize their server infrastructure to meet the demands of high user activity.

Subsequently, the article explores the concept of parallel event-driven processing, showcasing practical code examples in JavaScript and Node.js. It demonstrates how developers can leverage functional blocks and exploit the event loop's capabilities to efficiently manage code execution.

The text further investigates query deduplication as an optimization technique to reduce redundant queries and improve perceived latency. It highlights scenarios where multiple identical requests occur and the benefits of deduplicating queries in handling hot data.

Additionally, the article discusses the utilization of batched writes as another effective strategy for server-side parallelism. It explores the advantages of aggregating multiple write operations into batches, reducing the load on both the application and the database.

Finally, the article concludes by emphasizing the rewards of developing code that effectively utilizes parallelism. It underscores the importance of choosing serverless compute solutions to leverage application-side parallelism, allowing the adoption of efficient patterns that fully harness parallelism.

By understanding and implementing the concepts and strategies presented in this article, developers can unlock the untapped potential of their applications, achieving enhanced performance, scalability, and responsiveness in high-traffic scenarios.

Keywords:
parallelism, server architecture, high-traffic, optimization, event-driven processing, query deduplication, batched writes, application performance, scalability, responsiveness.

Введение

В области разработки программного обеспечения построение клиент-серверного приложения требует создания серверной архитектуры, способной беспрепятственно управлять интенсивным трафиком, исходящим от значительного числа одновременных пользователей. Разработчики активно стремятся оптимизировать свою серверную инфраструктуру, чтобы обеспечить надежную производительность даже в условиях интенсивной активности пользователей.

В настоящей статье предпринята попытка глубже проникнуть в многогранную область параллелизма, предлагая наглядные примеры для объяснения сложных стратегий, которые можно использовать для эффективного обслуживания больших объемов трафика. Эти тщательно разработанные стратегии позволяют разработчикам максимально эффективно использовать параллелизм при развертывании приложения на таких платформах, как AWS App Runner, AWS Fargate или любая другая вычислительная платформа, поддерживающая одновременную обработку множества запросов единым прикладным процессом.

Параллелизм приобретает первостепенное значение при проектировании и оптимизации приложений, подвергающихся значительным рабочим нагрузкам. Применяя разумные стратегии параллелизма, разработчики могут гарантировать, что их приложения демонстрируют оптимальную производительность, отзывчивость и масштабируемость. Эта статья посвящена всестороннему исследованию различных подходов, которые раскрывают истинный потенциал приложений, работающих в сценариях с высоким трафиком, тем самым проясняя оптимальное использование параллелизма.

Обзор литературы

Концепция параллелизма и ее применение в серверных архитектурах является широко изучаемой темой в области компьютерных наук. Способность эффективно управлять обширным трафиком от одновременных пользователей, особенно в сценариях с высоким трафиком, является критически важным аспектом оптимизации серверной инфраструктуры.

Как обсуждалось Херлихи и Шавитом (2008) [1], параллелизм играет фундаментальную роль в вычислительной технике, где задачи разбиваются на части и выполняются одновременно для повышения производительности системы. Эта концепция приобретает все большее значение в серверной среде в связи с распространением веб-приложений и сервисов, требующих высокой скорости реагирования и масштабируемости.

Параллельная обработка, управляемая событиями.

Технология параллельной обработки, управляемой событиями, стала ключевой стратегией при работе со сценариями с высоким трафиком. Как пояснили Сурьянараянан и др. (2012) [2], архитектура, управляемая событиями, позволяет одновременно обрабатывать события в приложении, что приводит к повышению быстродействия и производительности.

В области JavaScript и Node.js, был проведен ряд работ. Крокфорд (2008) [3] иллюстрирует использование функциональных блоков и цикла обработки событий для эффективного управления выполнением кода. Уникальная управляемая событиями неблокирующая модель ввода-вывода в Node.js был популярным выбором для создания масштабируемых сетевых приложений, как отметили Хьюз-Краучер и Уилсон (2012) [4].

Дедупликация запросов

Дедупликация запросов - еще один важный подход к улучшению параллелизма на стороне сервера. В работах Колдера и др. (1997) [5] и Вейкума и Воссена (2001) [6] подробно рассказывается о том, как устранение избыточных запросов может значительно снизить нагрузку на базу данных и повысить производительность системы. Необходимость дедупликации запросов, особенно в сценариях с высоким трафиком, также подчеркивается Бернсом и др. (2010) [7], где они обсуждают стратегии эффективной обработки "горячих" данных.

Пакетные записи

Пакетная запись — это еще одна важная стратегия, выявленная в области параллелизма на стороне сервера. Она включает в себя группировку нескольких операций записи в один пакет, тем самым снижая нагрузку на приложение и базу данных (Gray et al., 1996) [8]. Более того, Чанг и др. (2008) [9] объясняют основополагающие принципы распределенной системы хранения структурированных данных, где они подчеркивают преимущества пакетной обработки.

Бессерверные вычислительные решения и параллелизм на стороне приложения

В последние годы большое внимание также уделяется бессерверным вычислениям и параллелизму на стороне приложений. Бессерверная архитектура, описанная Робертсом (Roberts, 2016) [10], позволяет разработчикам создавать и запускать приложения, не думая о серверах, что обеспечивает возможность автоматического масштабирования и взимания платы только за затраченное вычислительное время. Джонас и др. (2019) [11] далее исследуют, как можно эффективно использовать параллелизм на стороне приложения в бессерверных средах, подчеркивая преимущества внедрения эффективных шаблонов для полного использования параллелизма.

В заключение, этот обзор литературы иллюстрирует различные подходы и методы, доступные разработчикам для обработки сценариев с высоким трафиком за счет эффективного параллелизма на стороне сервера. В последующих разделах этой статьи мы более подробно рассмотрим практическую реализацию этих стратегий с целью дать разработчикам возможность лучше оптимизировать свою серверную инфраструктуру и раскрыть неиспользованный потенциал своих приложений.

Основы параллелизма: Изучение циклов обработки событий и параллельного кода

В основе оптимизации при высоком трафике лежит фундаментальная концепция, согласно которой единый серверный прикладной процесс может эффективно обслуживать множество клиентов одновременно. Чтобы понять это явление, крайне важно вникнуть в работу серверного прикладного процесса. Как правило, серверные приложения получают HTTP-запросы, выполняют определенные операции в ответ на эти запросы и впоследствии предоставляют соответствующий ответ клиенту (рис. 1).

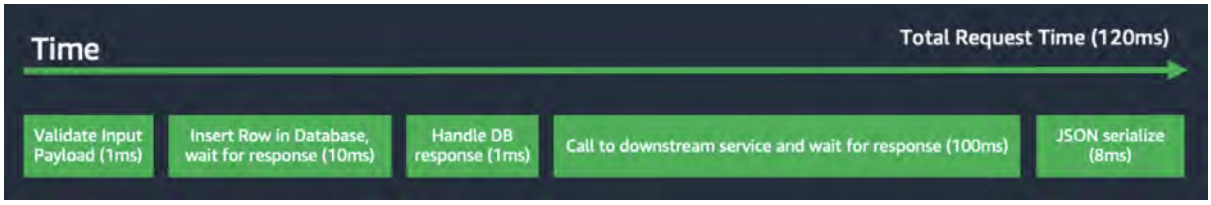

Рисунок 1 - Жизненный цикл одного HTTP-запроса

Figure 1 - The lifecycle of a single HTTP request

Жизненный цикл одного HTTP-запроса можно представить следующим образом: при получении HTTP POST-запроса серверный процесс проверяет входящую полезную нагрузку JSON. Впоследствии серверный процесс инициирует SQL-запрос для вставки строки данных в базу данных, тем самым сохраняя полученную полезную нагрузку. На следующем этапе серверный процесс ожидает ответа от базы данных, чтобы убедиться в успешном выполнении операции вставки. После этого серверный процесс выполняет вызов другой нисходящей службы, потенциально информируя ее о вновь сохраненном объекте в базе данных. Затем серверный процесс ожидает результата этого вызова, гарантируя его успех. Наконец, серверный процесс отвечает на исходный HTTP-запрос, выдавая код состояния 200 OK.

В целом весь этот процесс обычно занимает около 120 миллисекунд. Однако при ближайшем рассмотрении становится очевидным, что подавляющее большинство этого времени не отводится на выполнение инструкций процессора. Вместо этого он в основном тратится на ожидание сетевых операций ввода-вывода (рис. 2).

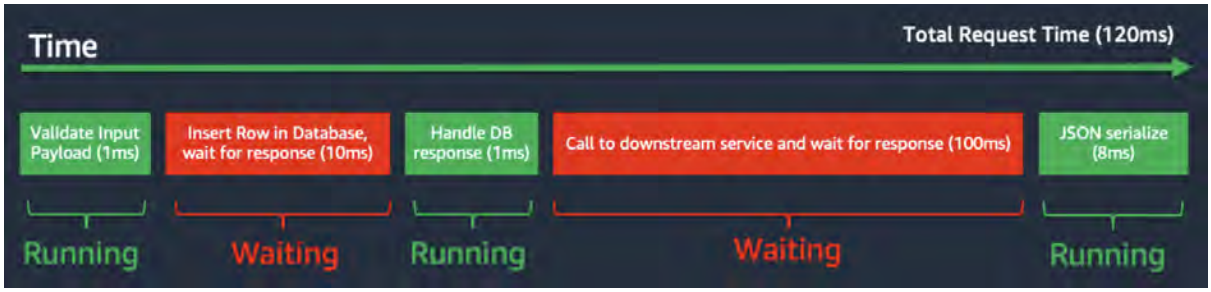

Рисунок 2 - Время ожидание сетевых операций ввода-вывода

Figure 2 - Waiting time for network I/O operations

Из общих 120 миллисекунд всего 10 миллисекунд фактически отводятся на выполнение кода приложения, в то время как оставшееся время тратится на холостой ход. Такая неэффективность вызывает озабоченность.

Для этого предлагается альтернативный подход, продемонстрированный на рисунке 3.

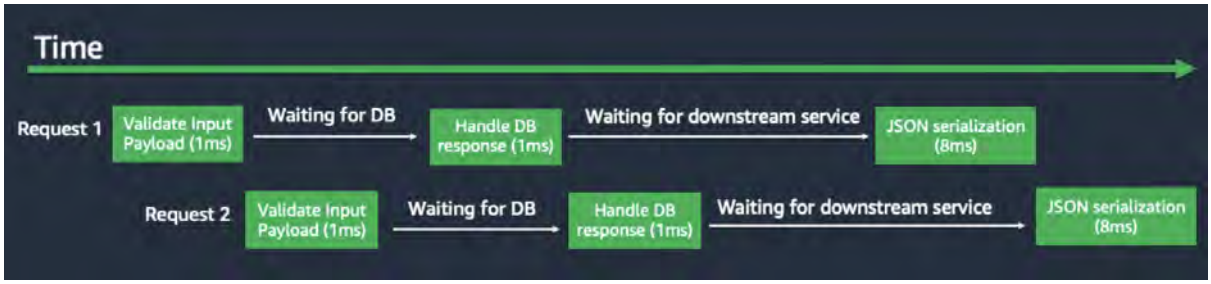

Рисунок 3 - Альтернативный подход операций

Figure 3 - Alternative approach of operations

В альтернативном подходе, когда приложение сталкивается с моментом ожидания операции сетевого ввода-вывода, оно может воспользоваться возможностью обработать другой входящий HTTP-запрос. За счет поэтапного выполнения инструкций кода процесс подачи заявки получил бы возможность обрабатывать несколько входящих запросов одновременно.

Большинство современных языков прикладного программирования включают встроенный цикл обработки событий, способный облегчить этот точный рабочий процесс. Цикл обработки событий работает как вечный цикл, непрерывно отслеживая события и реагируя на них. Эти события могут проявляться в виде HTTP-запроса, отправленного на сервер, получения ответа из базы данных или завершения операции сохранения файла. Если событие происходит во время выполнения другого кода, оно ставится в очередь.

Всякий раз, когда блок кода завершает свое выполнение, цикл обработки событий извлекает следующее событие из очереди и начинает выполнение соответствующего блока кода. Это позволяет циклу обработки событий поддерживать производительность процесса за счет постоянного приема событий от нескольких клиентов и выполнения связанного с ними кода.

Цикл обработки событий можно представить как планировщик задач в процессе выполнения, управляющий многочисленными небольшими фрагментами кода, для выполнения каждого из которых требуется всего лишь доля миллисекунды. Имея в распоряжении тысячу миллисекунд в секунду, планировщик цикла обработки событий эффективно использует это время для обработки входящих событий (рис. 4).

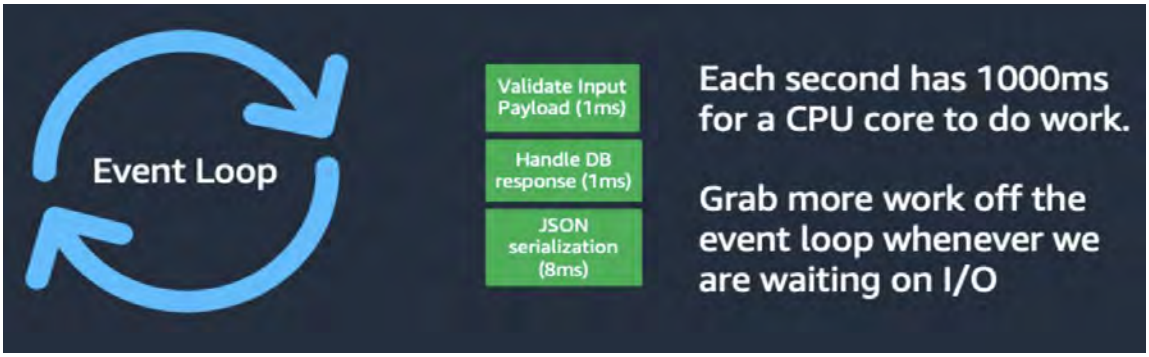

Рисунок 4 - Цикл обработки событий

Figure 4 - Event processing cycle

Эффективно используя возможности циклов обработки событий и параллельного выполнения кода, разработчики могут раскрыть потенциал для существенного повышения производительности и повышения быстродействия в сценариях с высоким трафиком. В последующих разделах мы рассмотрим передовые стратегии и рекомендации по оптимизации параллелизма в серверных архитектурах, позволяющие разработчикам раскрыть истинные возможности своих приложений, когда они сталкиваются со сложными рабочими нагрузками.

Практические примеры кода: JavaScript & Node.js

Чтобы проиллюстрировать реализацию кода параллельного цикла обработки событий в JavaScript и Node.js, рассмотрим следующие практические примеры (рис. 5, 6). Эти примеры демонстрируют использование функциональных блоков и демонстрируют способность цикла обработки событий эффективно управлять выполнением кода.

```
exports.handler = function (event, context, callback) {
  console.log('In the present');

  setTimeout(100, function() {
    console.log('Time travel to 100ms later');
    callback();
  });
}
```

Рисунок 5 - Использование времени простоя цикла обработки событий

Figure 5 - Using the idle time of the event processing cycle

```
exports.handler = async function (event, context, callback) {
  // Validate the event

  let dbResponse = await db.insert(); // Wait for IO from DB

  // Handle DB response

  let downstreamResponse = await downstreamService.put();

  // Serialize a JSON response
}
```

Рисунок 6 - Использование возможностей асинхронных функций

Figure 6 - Using the capabilities of asynchronous functions

Эти примеры подчеркивают способность цикла обработки событий эффективно управлять выполнением кода за счет разумного использования периодов простоя и предоставления контроля во время асинхронных операций. Используя такие методы, разработчики могут повысить общую отзывчивость и пропускную способность своих JavaScript и Node.js приложения, эффективно использующие возможности параллелизма.

Высвобождение возможностей дедупликации выборки

После успешного внедрения программного кода, способного одновременно выполнять несколько асинхронных задач, открывается доступ к множеству замечательных оптимизаций, которые используют этот новообретенный параллелизм.

Одним из особенно увлекательных методов является дедуплицирование выборки. Исследуем, как работает этот метод, рассмотрев следующий жизненный цикл запроса, который отражает поведение многих REST API (рис. 7).

Рисунок 7 - Поведение многих REST API

Figure 7 - Behavior of many REST APIs

1. Клиент инициирует запрос GET для получения сведений об учетной записи пользователя с идентификатором 1.

2. Серверный процесс отправляет SQL-запрос для получения соответствующих сведений об учетной записи из базы данных.

3. Примерно через 10 миллисекунд поступает ответ из базы данных.

4. Сервер приложений сериализует ответ в формате JSON и отправляет его обратно в виде HTTP-ответа.

Этот поток представляет собой обычное явление в клиентских/серверных приложениях. Однако также представим другой сценарий, в котором поступает второй запрос для того же пользовательского объекта, в то время как первый запрос все еще ожидает ответа на SQL-запрос (рис. 8).

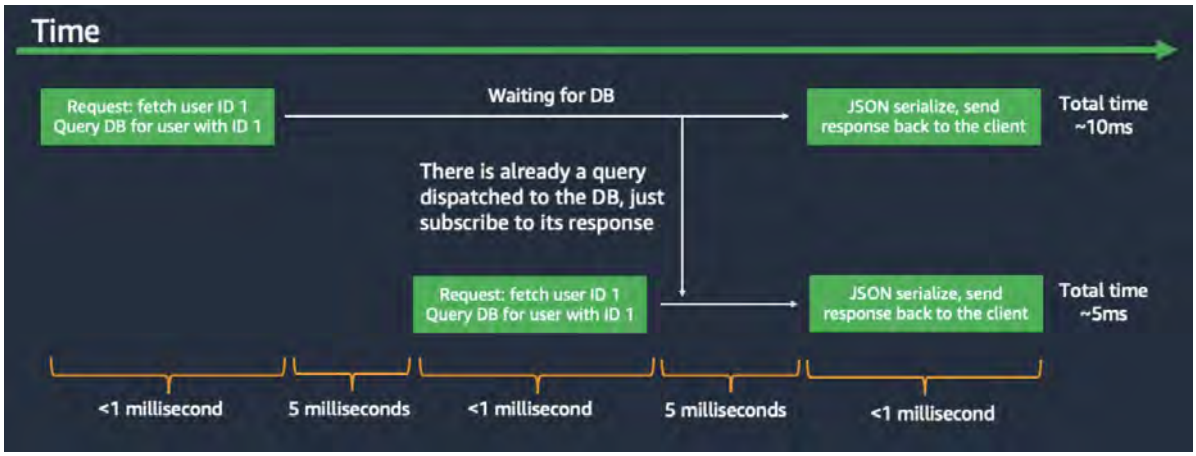

Рисунок 8 - Альтернативный сценарий событий

Figure 8 - Alternative scenario of events

1. Клиент отправляет запрос GET на получение сведений об учетной записи пользователя с идентификатором 1.

2. Серверный процесс отправляет SQL-запрос для получения сведений о пользователе из базы данных.

3. Примерно через 5 миллисекунд ответ на SQL-запрос еще не поступил, но поступает второй клиентский запрос для того же пользовательского объекта.

4. Вместо отправки избыточного SQL-запроса сервер идентифицирует текущий SQL-запрос и связывает второй запрос с ответом на первый запрос.

5. Оба HTTP-запроса выполняются с использованием одних и тех же данных, полученных из исходного SQL-запроса.

С точки зрения клиентов, инициирующих эти запросы, задержка первого запроса на стороне сервера составляет 10 миллисекунд, в то время как задержка второго запроса составляет всего 5 миллисекунд, поскольку он использует уже выполняющийся SQL-запрос в течение 5 миллисекунд.

Эта стратегия оказывается исключительно эффективной для приложений с "горячими" данными, где множество клиентов одновременно запрашивают одни и те же данные. Реализация дедупликации выборки не только снижает нагрузку на базу данных, но и значительно улучшает воспринимаемую задержку. Во многих случаях один запрос может обслуживать несколько идентичных запросов. Однако важно отметить, что эту стратегию не следует применять в сценариях, требующих строгой согласованности, таких как получение баланса банковского счета в режиме реального времени.

Для дальнейшей оптимизации дедупликации выборки вы можете использовать сеть доставки контента (CDN) перед вашим API, которая помогает дедуплицировать выборки на уровне HTTP-запроса. Однако все еще могут быть случаи, когда проскальзывает несколько идентичных запросов. Например,

если ваш CDN имеет несколько пограничных местоположений, каждое местоположение должно извлекать данные с исходного сервера приложений. Аналогичным образом, многочисленные параллельные запросы могут одновременно извлекать одни и те же базовые данные. В таких случаях дедупликация выборки служит дополнительным уровнем оптимизации, повышая скорость вашего сервиса и одновременно снижая нагрузку на базовый уровень данных.

Использование возможностей пакетной записи

Еще одна область, где можно эффективно использовать параллелизм на стороне сервера, - это операции на стороне записи. Давайте рассмотрим типичный жизненный цикл запроса на стороне сервера (рис. 9).

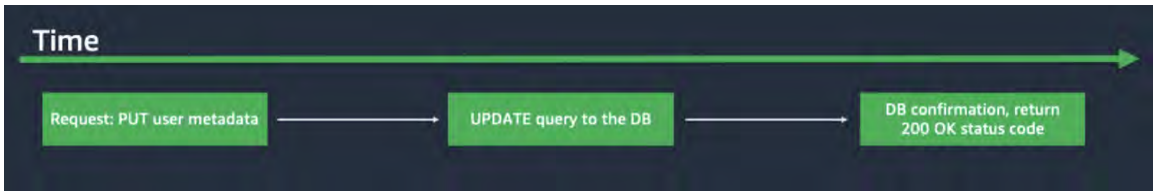

Рисунок 9 - Цикл запроса на стороне сервера
Figure 9 - Server-side request cycle

1. Клиент инициирует запрос POST или PUT с полезной нагрузкой JSON для сохранения данных.
2. Серверное приложение анализирует полезную нагрузку JSON и отправляет запрос SQL update для сохранения данных в базе данных.
3. База данных подтверждает завершение SQL-запроса.
4. Серверное приложение отвечает клиенту кодом состояния 200 OK, указывающим на успешный запрос.

Теперь рассмотрим альтернативный поток, который использует параллелизм на стороне сервера (рис. 10).

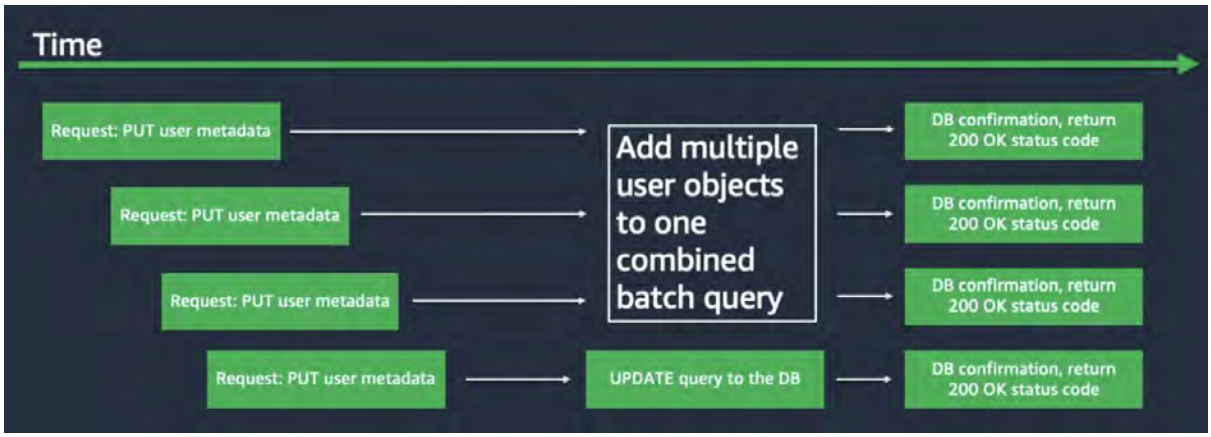

Рисунок 10 - Альтернативный поток, используемый параллелизм на стороне сервера
Figure 10 - Alternative flow used by server-side concurrency

1. Клиент отправляет запрос POST или PUT с полезной нагрузкой JSON для сохранения данных.
2. Вместо того чтобы немедленно отправить SQL-запрос, сервер вводит период ожидания, возможно, в течение 10 миллисекунд.
3. В течение этого периода в 10 мс на сервер поступают дополнительные запросы POST и PUT.
4. Сервер агрегирует эти запросы и отправляет один массовый запрос в базу данных либо по истечении периода времени в 10 мс, либо когда "пакет" заполняется достаточным количеством объектов.
5. После успешного завершения массового запроса все запросы получают 200 одобрительных ответов.

Эта стратегия может привести к небольшой задержке операций записи, но она значительно снижает нагрузку как на серверное приложение, так и на базу данных. Его эффективности способствуют несколько факторов, в том числе:

1. Оптимизация пакетов TCP: в большинстве конфигураций пакет TCP может содержать до 64 КБАЙТ. Отправка нескольких обновлений в рамках более крупного TCP-пакета более эффективна, чем передача нескольких пакетов меньшего размера.

2. Эффективность сериализации и десериализации: пакетная обработка данных более эффективна с точки зрения сериализации и десериализации, что приносит пользу как серверу, так и базе данных.

3. Эффективное сохранение: когда база данных сохраняет данные на носителе информации (например, на вращающемся жестком диске или твердотельном накопителе), выполнение однократной более масштабной очистки для долговременного хранения оказывается более эффективным.

Также преимущества этой стратегии еще больше проявляются по мере увеличения параллелизма серверного приложения. Для оптимизации его производительности можно настроить два основных параметра:

1. Размер пакета: определяет максимальное количество объектов, которое может накопиться перед запуском отправки пакетного запроса.

2. Задержка партии: указывает период ожидания поступления дополнительных товаров перед отправкой партии, даже если она еще не заполнена. Меньшая задержка отправки партии приводит к более быстрой отправке партии, в то время как более высокая задержка позволяет накапливать больше товаров.

По мере увеличения объема входящих запросов пакеты заполняются и отправляются быстрее, что приводит к снижению воспринимаемой задержки. Скорость, с которой выполняются вызовы к нижестоящей базе данных или службе, увеличивается ступенчато по мере увеличения параллелизма. Например, если размер пакета установлен равным 10, количество обращений к нижестоящей службе увеличивается со скоростью 1/10 по сравнению с общим объемом входящих запросов (рис. 11).

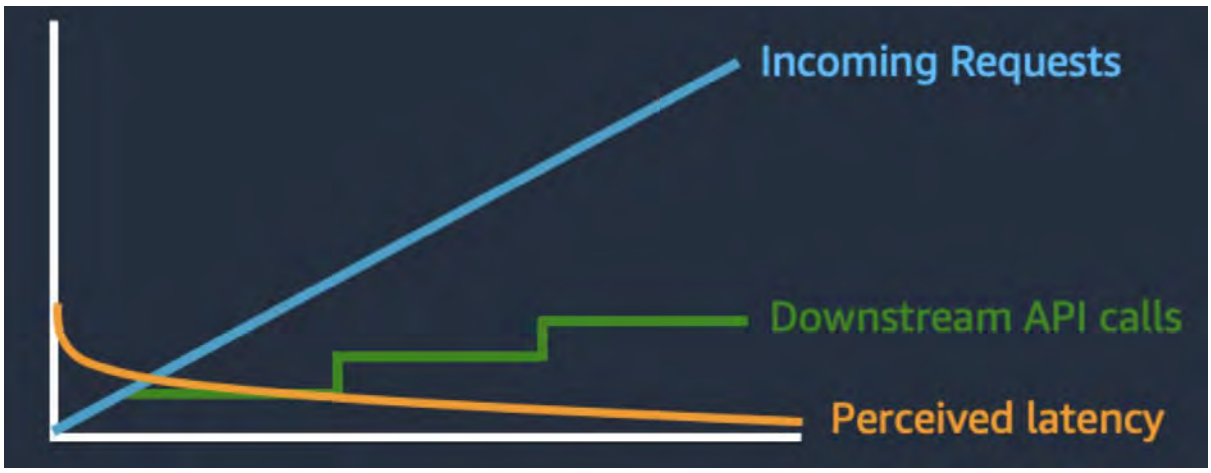

Рисунок 11 - Воспринимаемая задержка

Figure 11 - Perceived delay

Многие базы данных и последующие службы обеспечивают поддержку пакетных операций. Примечательно, что на платформе AWS основные сервисы, такие как Amazon DynamoDB (BatchWriteItem и BatchGetItem), Amazon Simple Queue Service (SendMessageBatch) и Amazon Simple Notification Service (PublishBatch), предлагают пакетные конечные точки, которые можно использовать для максимального использования преимуществ пакетной записи.

Заключение

Разработка кода, который может эффективно использовать параллелизм, может представлять

трудности, но награда существенна — высокоэффективное приложение, способное обрабатывать увеличенный трафик. Хотя писать код в вычислительной среде, подобной AWS Lambda, относительно просто, где каждый запрос может быть изолирован от других, недостаток заключается в невозможности эффективно использовать несколько параллельных запросов в рамках одной системы параллелизма. Однако выбор в пользу бессерверных вычислительных решений, таких как AWS App Runner или AWS Fargate, предоставляет возможность параллелизма на стороне приложения, позволяя внедрять эффективные шаблоны, которые полностью используют параллелизм.

Список использованной литературы:

1. Херлихи М., Шавит Н. Искусство многопроцессорного программирования. Морган Кауфманн. 2008.
2. Сурьянараянан Р., Джусак Дж., Верхоф М., Майклс П. Архитектура, управляемая событиями, для сервис-ориентированных архитектур: обзор литературы. Транзакции IEEE по разработке программного обеспечения, 2012. 38(2), 173-186.
3. Крокфорд Д. JavaScript: Хорошие стороны. O'Reilly Media, Inc., 2008.
4. Хьюз-Краучер Т., Уилсон М. Работают с Node.js . O'Reilly Media, Inc., 2012.
5. Колдер Б., Кринц С., Джон С., Остин Т. Размещение данных с учетом кэширования. В материалах восьмой международной конференции по архитектурной поддержке языков программирования и операционных систем. 1997. стр. 139-149.
6. Вейкум Г., Воссен Г. Транзакционные информационные системы: теория, алгоритмы и практика управления параллелизмом и восстановления. Морган Кауфманн. 2001.
7. Бернс Э., Адальберт У., Коннатсер С. Высокопроизводительные веб-сайты: необходимые знания для разработчиков интерфейсов. "О'Рейли Медиа, Инк.". 2010.
8. Грей Дж., Хелланд П., О'Нил П., Шаша Д. Опасности тиражирования и решение. В материалах международной конференции ACM SIGMOD по управлению данными 1996 года. 1996. стр. 173-182.
9. Чанг Ф., Дин Дж., Гемават С., Се В. С., Уоллах Д. А., Берроуз М., ... Грубер Р. Э. Bigtable: Распределенная система хранения структурированных данных. Транзакции ACM в компьютерных системах (TOCS), 2018. 26(2), стр. 1-26.
10.Робертс М. Бессерверные архитектуры. MartinFowler.com. [Электронный ресурс] - Режим доступа: https://martinfowler.com/articles/serverless.html . 2016 год
11.Джонас Э., Шлейер-Смит Дж., Сриканти В., Цай К. С., Ханделвал А., Пу К., ... Раган-Келли Дж. Упрощенное облачное программирование: взгляд Беркли на бессерверные вычисления. Препринт arXiv arXiv: 1902.03383. 2019.

References

1. Herlihy M., Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann. 2008.
2. Suryanarayanan R., Jusak J., Verhoef M., Michiels P. Event-driven architecture for service-oriented architectures: A literature review. IEEE Transactions on Software Engineering, 2012. 38(2), 173-186.
3. Crockford D. JavaScript: The Good Parts. O'Reilly Media, Inc. 2008.
4. Hughes-Croucher T., Wilson M. Up and Running with Node.js. O'Reilly Media, Inc. 2012.
5. Calder B., Krintz C., John S., Austin T. Cache-conscious data placement. In Proceedings of the eighth international conference on Architectural support for programming languages and operating systems. 1997. pp. 139-149.
6. Weikum G., Vossen G. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann. 2001.
7. Burns E., Adalbert W., Connatser C. High performance web sites: essential knowledge for frontend engineers. "O'Reilly Media, Inc.". 2010.
8. Gray J., Helland P., O'Neil P., Shasha D. The dangers of replication and a solution. In Proceedings of the 1996

ACM SIGMOD international conference on Management of data. 1996. pp. 173-182.
9. Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., ... Gruber R. E. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2018. 26(2), pp. 1-26.
10.Roberts M. Serverless Architectures. MartinFowler.com. [Electronic resource] - Access mode: https://martinfowler.com/articles/serverless.html. 2016
11.Jonas E., Schleier-Smith J., Sreekanti V., Tsai C. C., Khandelwal A., Pu Q., ... Ragan-Kelley J. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv preprint arXiv:1902.03383. 2019.