УДК 004.91

ПОДХОД К СОЗДАНИЮ СЕРВИСА ДИНАМИЧЕСКОГО ПОИСКА ДЛЯ ВЫСОКОНАГРУЖЕННЫХ СИСТЕМ

[1,2]Александров Д.В., [1,3]Калинин А.И.

[1]Национальный исследовательский университет «Высшая школа экономики» (НИУ ВШЭ), 109028, Москва, Покровский бульвар, д. 11, e-mail: aleksandrov_d@mirea.ru, kalinin_ai@hse.ru

[2]МИРЭА – Российский технологический университет», 119454, Москва, проспект Вернадского, 78

[3]Публичное акционерное общество «Сбербанк России», 117312, Москва, ул. Вавилова, д. 19

Существует множество высоконагруженных систем для поиска различной информации в Интернете. Скорость поиска и качество результатов (релевантность) поиска могут быть невысокими или поисковая система может иметь недостатки в части функциональности, например, по извлечению или фильтрации данных. Цель представленной работы – предложить решение по преодолению этих недостатков на основе результатов исследования существующих подходов и сравнения современных фреймворков и библиотек, активно применяемых ИТ-сообществом. Основными подходами, кратко рассмотренными в статье, являются кэширование данных, поиск пакетов информации в базе данных, расстановка приоритетов поисковых запросов, быстрый поиск похожей информации, а также интеграция предлагаемого программного решения с облачными платформами. В качестве результата представлен сервис, который можно интегрировать в типовые поисковые системы, где для поиска информации в базе данных требуется быстрое отказоустойчивое приложение. Также в статье представлены результаты тестирования предлагаемого сервиса в клиентской системе одной из компаний для демонстрации эффективности предлагаемого решения для быстрого динамического поиска информации.

Ключевые слова: сервис; динамический поиск, база данных, кэширование, маскировка, служба теневого копирования томов, ТМОВ.

AN APPROACH TO CREATING A DYNAMIC SEARCH SERVICE FOR HIGHLOAD SYSTEMS

[1,2]Alexandrov D.V., [1,3]Kalinin A.I.

[1]National Research University Higher School of Economics (HSE University), 11 Pokrovsky Bulvar, Moscow 109028, , e-mail: aleksandrov_d@mirea.ru, kalinin_ai@hse.ru

[2]Federal State Budget Educational Institution of Higher Education «MIREA – Russian Technological University» 78 Vernadsky Avenue, Moscow 119454

[3]Public Joint Stock Company "Sberbank of Russia", 117312, Moscow, st. Vavilova, 19

There are a lot of high-load systems for searching various information using the Internet. However, such systems havedis advantages and various trade-offs. For example, the search speed may drop, the quality of the search result itself may not satisfy the user, or the search engine may have poor functionality for extracting or filtering data. This paper aims to solve this problem by analyzing existing solutions, considering various approaches and comparing modern frameworks and libraries. The main approaches that will be considered are the possibilities of data caching, searching for batches of information in the database, prioritization of requests, quickly finding similar information, and integrating the proposed solution with cloud platforms. As a result, it will be the service that is not so difficult to add into typical, ready-made systems, where a fast and fault-tolerant algorithm is required to search for information in the database. The service will also be tested in the company customer search system, to show the efficiency and effectiveness of the quick search solution.

Keywords: service, driving search, database, caching, masking, volume shadow copy service, TMOV

1. Введение

В век информационных технологий люди ежедневно заходят в Интернет и часто пользуются поисковыми сервисами, такими как Яндекс, или ищут своих друзей в социальных сетях, находят различные товары в интернет-магазинах и ищут информацию в информационных хранилищах. Кроме того, поисковые сервисы необходимы сотрудникам на работе, например, для поиска клиентов компании или различных контрактов в

локальных системах. Существенными особенностями таких поисковых сервисов являются высокая скорость поиска информации, наличие различных видов ее фильтрации и способность выдерживать огромные нагрузки в любое время суток. В связи с этим сервисы постоянно обновляют, расширяя их функционал, чтобы многие люди могли пользоваться ими без лишних временных затрат.

Много исследований посвящено созданию быстрых и эффективных поисковых сервисов. Яндекс, Google, Netflix, Apache и многие другие IT-гиганты постоянно участвуют в подобной деятельности, регулярно открывая новые подходы к ускорению и улучшению поиска данных. Например, система YouTube имеет множество прокси-серверов, разделенных на зоны для различных стран, куда направляется основной трафик пользователей; такая прокси-инфраструктура называется сетью доставки контента (Content Delivery Network, CDN). Кроме того, Google создала свою систему для внутреннего хранения частых поисковых запросов и быстрого ответа на запросы клиентов без обращения к центральному серверу. Такая система также может объединять идентичные запросы в один, что снижает нагрузку на ее. В [1] более подробно описаны стратегии, которые YouTube использует для сокращения времени ожидания клиентов.

Другим примером является серверная инфраструктура Netflix, где используется трехуровневая CDN [2]. В такой системе на первом уровне службы хранят содержимое в кэше, который может быстро возвращать данные в ответ на запросы. На втором уровне хранится географически релевантный контент, который является наиболее используемым в определенном регионе. И на третьем уровне хранится независимый от региона наиболее часто используемый контент. На рис. 1 схематично показано распределение трафика по всему миру от исходного сервера, что обусловлено необходимостью работы в различных регионах планеты. Netflix пытается хранить как можно больше данных ближе к пользователю в части географического местоположения, а также сохранять контент в быстрой памяти, не пытаясь повторно считывать его из базы данных. Таким образом, компания старается максимально кэшировать данные, чтобы нагрузка на ядро системы была минимальной, а пользователи получали запрошенные данные как можно быстрее.

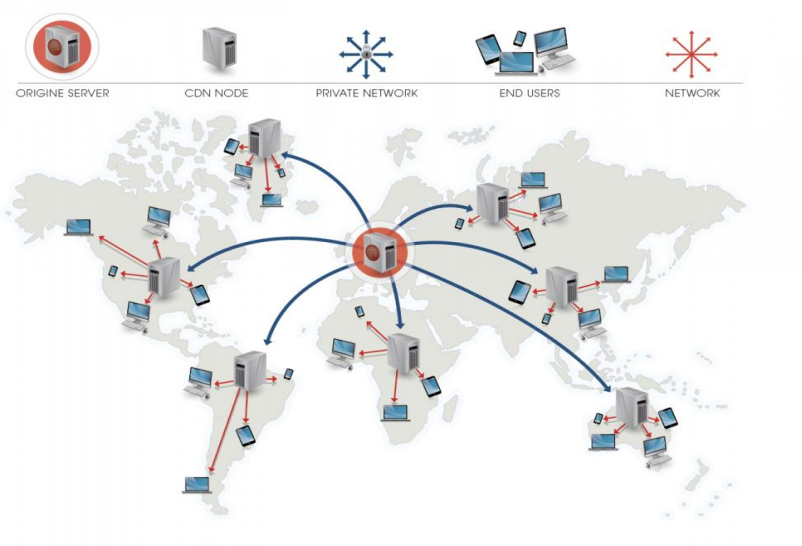

Рис. 1. Географическое распределение трафика

Вторым интересным достижением Netflix в серверной части является то, что компания одной из первых использовала архитектуру микросервисов и начала работать с программным обеспечением для оркестровки контейнеризированных приложений в высоконагруженных системах [3]. Netflix решила создать свою собственную систему микросервисов, в которой их собственный оркестратор приложений выбирает, куда отправлять трафик, как будет осуществляться балансировка микросервисов, а также увеличиваться или уменьшаться выделенные системные ресурсы в процессе работы. Такой подход позволяет им правильно распределять ресурсы системы и предотвращать ее перегрузку.

Еще одним решением Netflix является относительно новая библиотека под названием Spring Cloud [4]. Эта библиотека относится только к Java-фреймворку Spring Boot, что позволяет использовать облачные принципы в процессе написания программного кода. Более того, в производственной системе может оказаться возможным исключить из использования программное обеспечение для оркестровки контейнерных приложений и ускорить взаимодействие системных служб. Поскольку непосредственно код отвечает за взаимодействие системных

служб, дополнительное программное обеспечение, управляющее трафиком без видимости самих служб, не требуется.

Таким образом, изучая решения различных ИТ-гигантов, направленные на ускорение работы их сервисов, можно отметить две основные идеи в части проектирования высоконагруженных систем:

1) кэширование часто используемых данных в быстро доступной памяти,

2) балансировка, основанная на загрузке сервисов в оркестраторах.

В этой статье эти идеи будут рассмотрены более подробно.

Целью проекта, о котором идет речь в статье, является решение задачи быстрого поиска данных с использованием готовых программных продуктов и облачных приложений, которые могут работать в современных динамичных, масштабируемых, слабо связанных, отказоустойчивых, управляемых и наблюдаемых средах. Кроме того, рассматривается применение различных методов разработки облачных приложений, которые включают надежную автоматизацию в сочетании с частыми и предсказуемыми изменениями: автоматизация, непрерывная доставка и DevOps (акроним от англ. development & operations) – методология автоматизации технологических процессов сборки, настройки и развертывания программного обеспечения.

2. Постановка задачи на создание поискового сервиса

Учитывая, что IT-компании стараются «идти в ногу» с основными тенденциями и использовать лучшие современные решения, тем не менее ими часто создаются мало функциональные и неэффективные поисковые сервисы. Такие проблемы часто связаны с созданием поисковых сервисов, что является трудоемким и нерентабельным для большинства компаний, которые не могут позволить себе нанять множество высококвалифицированных специалистов для решения этой задачи. Кроме того, многие программисты надеются, что готовая библиотека или программное обеспечение сделают все за них. Например, с тех пор как стали доступны облачные технологии, разработчики размещают все свои сервисы в облаке без заранее подготовленной инфраструктуры и ожидают от такой системы отличных результатов. В качестве альтернативы они создают базу данных без каких-либо индексов или хорошо выстроенных связей, что является одной из причин, по которой производительность системы существенно снижается.

Проект авторов «Служба динамического поиска для высоконагруженных систем» направлен на то, чтобы показать, что он может гибко решать описанные выше проблемы поисковых сервисов, что позволит интегрировать такое решение в большинство систем без каких-либо сложностей с взаимодействием служб центральной системы. Кроме того, основная цель заключается в том, чтобы опробовать несколько подходов к разработке поискового сервиса, найти наилучший вариант с точки зрения скорости и качества поиска, а также изучить различные способы разработки отказоустойчивых и непрерывно работающих сервисов при больших нагрузках. Проект поможет понять, как подходы к построению баз данных влияют на работу сервиса при различных нагрузках, и как стоит использовать память для уже отправленных запросов. Кроме того, сравнение различных подходов поможет понять, имеет ли смысл использовать микросервисную архитектуру при разработке поисковых сервисов. В то же время данный проект позволит рассмотреть так называемый «нативный» подход к созданию интеграционных сервисов. Он позволяет безболезненно добавлять дополнительные функциональные возможности в уже функционирующие сложные системы, такие как поисковые сервисы.

Кроме того, в конечном итоге сравнение скорости работы поможет найти наилучший способ представления поисковой строки, понять, что это такое, и определить, когда следует анализировать данные, поступающие от пользователя. Поскольку этот фактор немаловажен, анализ данных также занимает некоторое время. Кроме того, система должна быстро понимать, какие данные хочет получить пользователь, и по каким критериям их следует фильтровать, используя данные клиентского запроса.

3. Методологические аспекты создания высоконагруженных систем

3.1 Технологии разработки поисковых сервисов

3.1.1 Клиент-серверный фреймворк

Прежде чем определить подходы, которые будут использоваться при реализации поискового сервиса, стоит определиться со стеком технологий, поскольку правильный выбор поможет с минимальными усилиями упростить разработку сервиса и даже ускорить его работу.

Выше уже упоминался написанный на языке программирования Java фреймворк Spring Cloud, однако следует заметить, что использование языка, которым пользуются многие разработчики, не гарантирует, что это будет правильный выбор в конкретной ситуации. Причины применения Java детально описаны в [5], что дает представление о том, какими преимуществами обладает данный язык. Кроме того, имеет смысл сравнить фреймворки для написания серверных сервисов. При сравнении одним из важнейших критериев является скорость работы клиент-серверной среды, а также и скорость взаимодействия сервиса с базой данных. Java имеет три основные клиент-серверные среды: Jetty, Netty и Tomcat.

Jetty – это контейнер сервлетов с открытым исходным кодом, который также действует как веб-сервер. То есть это контейнер, в котором присутствуют готовые реализации и протоколы для создания подключений клиентов к серверу. Работа Tomcat и функционирование Jetty очень похожи, поэтому следующие описания Tomcat применимы и к Jetty.

Netty – это неблокирующая клиент-серверная платформа для разработки сетевых приложений Java, таких как серверы протоколов. Его преимущество перед более старым Tomcat заключается в том, что Tomcat создает огромное количество блокирующих виртуальных (параллельных) потоков для обработки каждого клиентского запроса при значительной нагрузке. Эти виртуальные потоки могут полностью засорить очередь задач в естественных (последовательных) потоках и начать блокировать запросы других людей, что приводит к простою службы, что, в свою очередь, уже может привести к полному отключению службы и, возможно, системы в целом. Netty лишен данного недостатка. Эта среда отслеживает естественные потоки и правильно расставляет приоритеты для различных задач, то есть запросов пользователей, что обеспечивает высокую стабильность работы сервиса. Тем не менее, реальная разница видна только при огромных нагрузках на удаленный сервер, которые требуются для высоконагруженных поисковых сервисов.

Производительность Netty по сравнению с Jetty и Tomcat превосходит все ожидания, что подтверждено результатами тестировании, представленными в [6].

Рис. 2 и 3 демонстрируют, что Netty намного эффективнее Tomcat с точки зрения управления нагрузкой, что очевидно, поскольку каждый раз Tomcat использует очень дорогостоящую операцию – создание нового виртуального потока, который сильно загружает центральный процессор. Проверки скорости загрузки и нагрузки на другие компоненты устройства также произведены в [6]. В результате Netty определенно выигрывает.

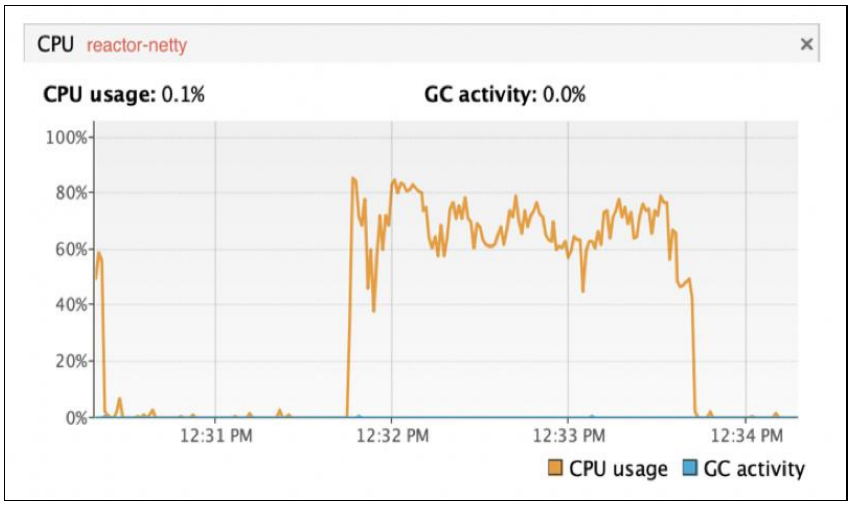

Рис. 2. Результаты оценки производительности Netty [6]

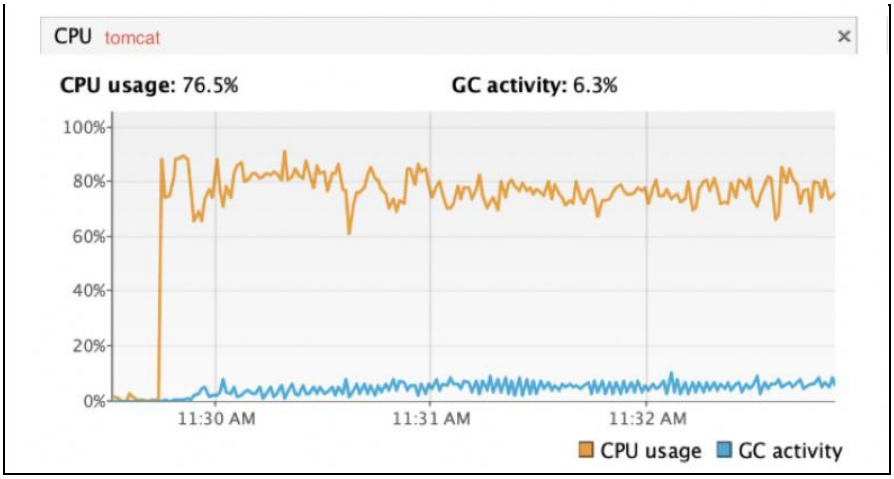

Рис. 3. Результаты оценки производительности Tomcat [6]

Netty также известен своей высокой производительностью на фоне различных фреймворков, написанных на

разных языках программирования [7]. На рис. 4 показано, что Netty занимает вторую позицию по скорости, уступая только C# framework, однако в рамках данной работы авторами выбран именно Netty. Таким образом, для написания поискового сервиса использовались Java с фреймворком Spring Boot и веб-клиент Netty.

3.1.2 Инфраструктура: сервер баз данных

Прежде чем подключать базу данных к сервису, программисты должны сразу же подумать о способе управления различными версиями базы данных. Если этот момент не будет принят во внимание, то при различных модификациях кода и базы данных возникнут трудности с их соответствующей синхронизацией. Поэтому необходимо следить за версией базы данных. Для Java-платформы существует два основных фреймворка: liquibase [8] и flyway [9].

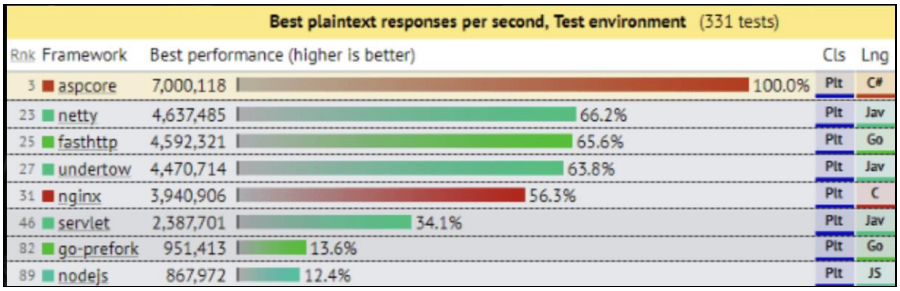

Рис. 4. Число ответов в секунду для различных фреймворков [7]

Поисковый сервис использует liquibase, его преимущества перед flyway описаны в [10]. Следует отметить, что самые значительные недостатки flyway – это то, что он не является бесплатным и не имеет открытого исходного кода. Функции, которые есть в бесплатной liquibase, отсутствуют в бесплатной версии flyway.

В случае Netty обычный подход подключения к базе данных по сеансу не подходит, поскольку создаются блокирующие виртуальные потоки, которые могут помешать работе Netty.

Следовательно, единственным решением в данном случае является r2dbc [11]. Этот фреймворк позволяет подключаться к реляционной базе данных по неблокирующим потокам, поскольку каждый запрос будет запрашивать не всю таблицу целиком, а только пакеты данных.

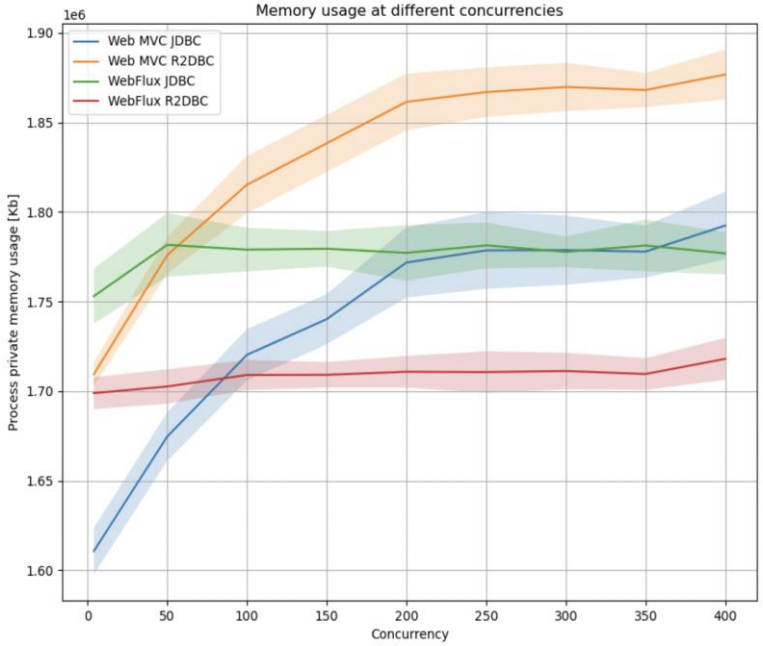

Рис. 5. Использование памяти в различных системах подключения к БД.

Преимущества r2dbc перед другими фреймворками также отмечены в [11], он явно превосходит фреймворки, представленные на рис. 5, по выдерживанию нагрузки.

Различие между фреймворками Spring-Boot для подключения к базам данных детально описано в [12].

3.2 Проектные решения

Далее необходимо углубиться в решения, которые используются в текущем поисковом сервисе.

3.2.1 Индексация таблиц в базе данных

Одно из главных в поисковом сервисе – правильно спроектированная база данных. Для того чтобы быстро фильтровать и при этом извлекать данные, идеально подходит реляционная база данных. Поиск данных с помощью запросов в такой базе данных прост, но не всегда быстр. Поэтому использование индексов в базе данных весьма актуально.

Индекс базы данных – это уникальная структура, используемая для ускорения поиска в таблицах. Индекс – это, по сути, крошечная таблица, которая сама по себе хэширует данные, как в хэше какой-либо таблицы, и использует этот хэш. Операция поиска в индексе выполняется быстро, что значительно увеличивает скорость поиска в базе данных. В большинстве систем управления базами данных (СУБД) данные кэшируются для быстрого возврата результатов.

3.2.2 Кэширование данных

Кэширование данных (Data Caching) также позволяет значительно ускорить использование сервисов. Более того, кэширование может быть настроено на разных уровнях обслуживания: рядом со службой или внутри службы.

Рядом со службой настраивается кэширование с использованием различных прокси-серверов. Например, это может быть сеть доставки контента, которую используют многие компании, или настраивают сервер Nginx. Этот прием позволит убрать поток запросов из самого сервиса, что его разгрузит, однако недостатком такого решения является то, что кэширование может быть некорректно настроено.

Кэширование внутри сервиса более эффективно при условии, что оно настроено на неблокирующем модуле. Запрос на уже кэшированные данные не будет мешать запросам к базе данных. Этот метод также позволяет точно настроить строку поиска и выполнить фильтрацию по кэшированному объекту.

Поисковая служба также использует кэширование, производимое на уровне кода. При запуске сервиса создается специальная область хранения в виде хэш-таблицы. Каждый раз, когда пользователь делает запрос, создается сумма строк параметров запроса, из нее создается хэш с помощью хэш-функции, и ответ сохраняется в таблице. И каждый раз, когда хэш запроса совпадает с тем, что есть в таблице, служба не будет обращаться к базе данных, но немедленно получит ответ из хранилища с более быстрым доступом.

Например, клиент отправляет строку запроса “имя = таблица и фильтр = месяц”. Алгоритм кэширования создаст строку «Tablemonth», затем уже алгоритм создаст хэш-значение с помощью хэш-функции, поместит это значение в таблицу как ключ и добавит ответ из базы данных как значение таблицы.

В данном алгоритме присутствует одна проблема: всякий раз, когда данные в базе данных обновляются, необходимо либо очищать кэш-хранилище, либо обновлять его новыми данными. Поисковый сервис реализует второй вариант. Поскольку сама операция обновления является очень трудоемкой, то вместе с ней обновляется таблица кэша. Программа просматривает операции удаления, обновления и добавления в таблице и запоминает идентификаторы значений, которые были затронуты. Алгоритм просматривает значения в кэше; если в хэш-таблице есть совпадающее указанное значение, то хэш-кэш удаляется.

3.2.3 Алгоритм маскировки информации

В данной статье предлагается свое решение для хранения данных в базе данных и потенциального ускорения поиска данных. Маскировка информации позволит более эффективно фильтровать данные из базы данных, поскольку потенциально может быть использовано несколько возможных параметров, усложняющих запрос данных. Однако, если все фильтры сведены к одному столбцу, достаточно выполнить поиск по этому столбцу в таблице, и будут возвращены необходимые данные. Этот подход очень похож на геокодирование, где координаты преобразуются в одну линию, и эту линию можно использовать, в частности, для определения расстояния.

Такой подход увеличивает скорость извлечения данных, как будет показано ниже. Но это значительно увеличивает размер данных, поскольку добавляется новый столбец. Главное преимущество подхода заключается в том, что не нужно перебирать разные столбцы, когда необходимо выполнить поиск данных в базе данных с использованием различных параметров, достаточно сформировать маску по запросу пользователя и использовать ее для поиска необходимых записей.

Например, в базе данных есть поля age, code и числовой идентификатор. Алгоритм сохраняет эти поля в шестнадцатеричном формате, что позволяет представлять данные в виде универсального уникального идентификатора (UUID – Universally Unique Identifier), но со скрытым значением. Замаскированное значение будет равно 94ed-5519e1f3-7973d5e7661e. Где 94ed – год, 5519e1f3 – код и т. д. Такое преобразование позволяет не перебирать все столбцы, а выгружать одно значение и сравнивать входные данные с ним. Когда пользователь вводит, например, год, происходит битовый сдвиг в биты года и используется обычное сравнение для

шестнадцатеричной системы, что позволяет избежать работы со строками, и сразу используются операции с числами, которые всегда выполняются быстрее. Например, пользователь вводит 2022, которое в шестнадцатеричном формате равно 07E6, и служба сравнивает 94ed из замаскированного значения с 07E6. Таким образом, сервис может вычислить числовое значение, используя замаскированное поле.

Таким образом, используя маскировку данных, можно обойти таблицу за один цикл и найти в базе необходимые данные. Далее данные будут возвращены конечному пользователю, что в дальнейшем позволит ускорить поиск, поскольку СУБД прекратит отправлять поисковые запросы, когда найдет и соберет необходимый пакет данных.

3.2.4 Алгоритм Bitap

Поскольку r2dbc отправляет запросы пакетами, фильтрация данных может быть перенесена за пределы системы управления базами данных. Потенциально это не должно привести к существенной потере производительности. Но, напротив, это потенциально могло бы ускорить процесс, поскольку нет необходимости просматривать всю базу данных целиком. Таким образом, поиск строк может выполняться в самом коде, а не в базе данных, что дает более широкий выбор алгоритмов поиска строк.

Как исследовано в [13, 14], существует множество алгоритмов поиска строк, а также подходов к поиску подстрок, но одним из лучших вариантов, которые можно применить в поисковой системе, является алгоритм bitap. Этот алгоритм позволяет сравнивать строки с учетом расстояния Левенштейна [15]. Также очень важно, чтобы поисковые службы обращали внимание на возможность наличия ошибок в строке поиска. Алгоритм bitap учитывает, насколько расстояние Левенштейна может рассказать о количестве ошибок, допущенных в тексте.

Конечно, не все ошибки могут быть исправлены, только следующие:

- отсутствие письма,
- смещение двух букв,
- перемещение двух букв,
- наличие одной лишней буквы.

Однако этих ошибок достаточно, чтобы точно понять, что ввел пользователь. Кроме того, подход к отбору показал, что наилучшее расстояние Левенштейна для поисковой системы равно 5. Поскольку при больших значениях поисковая система больше не отправляет ожидаемый результат, а при меньших значениях данных мало.

3.2.5 Взвешивание ответов

Благодаря тому, что алгоритм bitap может определять расстояние Левенштейна, можно построить приблизительную важность каждого значения в матрице. Учета только одного расстояния Левенштейна недостаточно для качественного поиска. Авторы статьи предлагают учитывать расстояния между словами и символами в строке поиска. Можно сформулировать алгоритм, посмотрев на расположение слов и букв.

Каждое расстояние Левенштейна преобразуется таким образом, что при большом значении возникает небольшое количество ошибок, а при значении веса, равном пяти, расстояние Левенштейна равно нулю. Также при добавлении нескольких слов определяется расстояние между словами: чем оно меньше, тем больший вес имеет ответ. Третий член – это размер самого слова, затем слово, которое находит bitap, сравнивается по размеру со словом, введенным клиентом, и при меньшей разнице в размере ответ также будет иметь большой вес.

Алгоритм можно описать формулами. Пусть *needle* – строка, которую вводит пользователь сервиса, а *haystack* – строка в базе данных, в которой алгоритм будет выполнять поиск. Здесь L = [dl0, dl1, ... , dli , ... ,dlN] – набор значений расстояния Левенштейна для слов в *needle,* и где *N* – количество слов в *needle*. Назовем *Nld* максимальным расстоянием Левенштейна, при котором пользователь может предположить, что слово похоже на желаемое.

$$W_l = \sum_{i=0}^{N} \frac{N_{ld} - d_{l_i}}{1} \quad . \qquad (1)$$

Таким образом, результирующий вес ответа по Левенштейну может быть рассчитан по формуле 1. Определим расстояние *dci* между символами:

$$d_{c_i} = |length(needle_i) - length(haystack_{p_i})| \qquad (2)$$

Формула 2 показывает, как вычислить расстояние между символами, где функция *length: word → N* вычисляет длину слова. *needlei* – это слово, где *i* – индекс слова в «игле». А *haystackpi* – это слово, где *pi* позиция слова из “стога сена”, которое сравнивалось с i-м словом в «игле». Это расстояние дает оценку разницы между словом и искомым словом из того, что хранится в базе данных.

$$W_{lc} = \sum_{i=0}^{N} \frac{N_{ld} - d_{l_i}}{d_{c_i} + 1} \quad (3)$$

Теперь формула может содержать *dci,* и формула может учитывать, как расстояние Левенштейна, так и расстояние между символами.

Формула 3 позволяет нам рассчитать вес с двумя расстояниями, может быть заметно, что чем больше расстояние между символами, тем меньше будет вес. Расстояние может быть равно нулю, поэтому формула содержит прибавление единицы. Сформулируем значение слова *distance weight Wd*. Для результирующего веса не хватает только слова *distance.*

$$W_d = \sum_{i=0}^{N-1} |p_i - p_{i+1}| \quad (4)$$

Формула 4 производит вычитание по модулю между позициями *pi* слов в *haystack*, которые были сопоставлены со словами из *needle*. Сумма этих различий может дать оценку того, насколько предложение отличается от того, что ищет пользователь, и от того, что хранится в базе данных.

Таким образом, после всех расчетов можно сформулировать результирующий вес W.

$$W = \frac{W_{lc}}{W_d + 1} \quad (5)$$

Формула 5 показывает расчет конечного веса. Все взвешенное расстояние Левенштейна *Wlc* делится на вес расстояний между словами *Wd. Таким* образом, можно оценить вес предложения на основе текста поиска.

Пример такого подхода: клиент отправляет строку «заказать еду», и системе необходимо найти записи в базе данных (при условии, что кэш пуст и нет сохраненного ответа на этот запрос). Пусть в базе данных будут записи с этими значениями:

1) Заказать еду для себя.

2) Заказывать еду для кого-то.

3) Заказать немного еды.

Рассмотрим слово “Зказ”. Для всех строк расстояние Левенштейна будет равно 1, потому что буквы “ка” поменяны местами, и алгоритм *bitap* ищет соответствующую подстроку. На данном этапе вес каждого ответа одинаков и равен 4. Но для строки 2 расстояние Левенштейна будет равно 4 в дополнение к замененному «Зк». Но полное слово в строке 2, в котором найдена подстрока на 3 символа длиннее остальных, поэтому вес второй строки уменьшается и становится 4 / 3, в то время как для строк 1 и 2 вес по-прежнему равен 4. И последний этап взвешивания – это расстояние между словами, где строка 1 явно выигрывает, поскольку расстояние просто равно нулю. Но чтобы не делить конечный вес на ноль, алгоритм взвешивания добавляет 1 к этому значению. Итак, строки 1 и 2 имеют вес 1, а строка 3 имеет вес 4. После этих вычислений промежуточный вес делится, и в результате получается:

1) Заказать еду для себя. Вес: 4

2) Заказывать еду для кого-то. Вес: 4 / 3

3) Заказать немного еды. Вес: 4 / 4 = 1

В результате клиент получит строки в порядке 1, затем 2 и 3. Таким образом, такой подход позволяет оценивать сходство слов не только по расположению букв, но и по их положению в тексте, поскольку слова могут располагаться в некоторых текстах в базе данных, но расстояние между ними будет огромным.

Кроме знаний об используемых фреймворках и алгоритмах необходимо иметь представление, какие облачные решения следует использовать.

3.3 Облачные решения для создания высоконагруженных систем

Netflix, Google и другие ИТ-компании используют свои сервисы в “облаках”, как упоминалось выше. Они делают это исключительно потому, что облачные решения помогают быстро и относительно легко запускать сервисы и использовать их для различных целей. Облачный подход приводит к словосочетанию Cloud Native [16]. Облачные нативные приложения – это высоко распределенные системы, которые находятся в облаке и устойчивы к изменениям. Системы состоят из множества служб, которые взаимодействуют по сети и развертываются в динамичной среде, где все постоянно меняется.

Прежде чем принять решение о том, где и как развернуть поисковый сервис, необходимо определить, как

приложение будет перенесено в облако, и в каком виде оно будет храниться и запускаться. Существует множество различных подходов и способов хранения и запуска кода, но выделяются два основных:

Создать и запустить код на определенном сервере (виртуальной машине) без дополнительных манипуляций,

Собрать образ приложения в универсальный архив (контейнер) и запустить не машинный код, а контейнеризированный образ самого поискового сервиса.

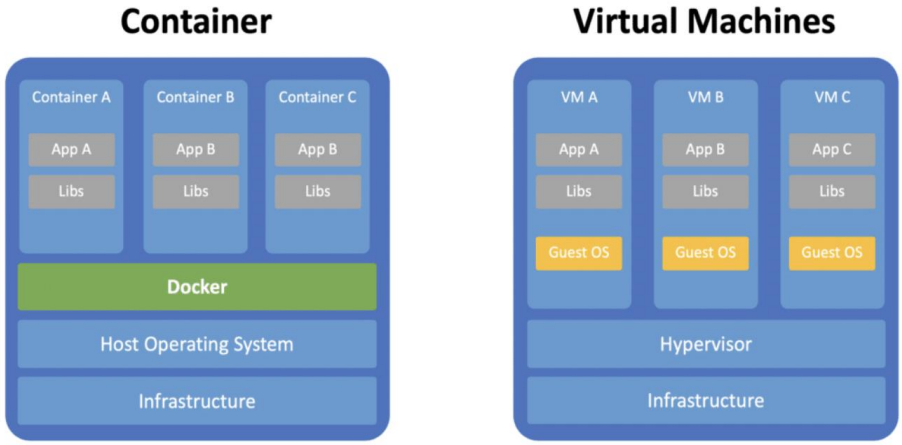

Рис. 6. Сравнение контейнера с виртуальными машинами

Большинство компаний за последние несколько лет использовали второй метод, и причины такого частого использования описаны в [17]. Среди решений для контейнеризации наиболее часто используется Docker [18]. Разницу между Docker и виртуальной машиной можно увидеть на рис. 6, причины ее использования представлены в [19]. Можно отметить, что docker позволяет создавать образы приложений, которые можно легко переносить. Преимущество docker, как видно из рис. 6, заключается в том, что контейнер не зависит от операционной системы, в которой он запущен, среда всегда будет одинаковой, что означает, что docker-образ при запуске на различных устройствах всегда будет работать одинаково.

Использование docker приводит к тому, что приложение может запускаться в оркестраторе сервисов (Service Orchestrator). Одним из наиболее популярных оркестраторов является Kubernetes [20, 21]. В данной работе используется именно он, поскольку является бесплатным и хорошо документированным. Выше было отмечено, что у IT-гигантов есть и свои собственные оркестраторы, которые настроены по-другому, например у Netflix, что описано в [3]. Таким образом, сервис, подготовленный в соответствии с принципами Cloud Native [16], может быть легко интегрирован в различные системы, которые даже не используют Docker или Kubernetes, поскольку это программное обеспечение может быть быстро запущено и установлено в любом облаке. Поисковый сервис заранее готовят к развертыванию в облаке, что способствует его легкой интеграции в различные системы. Так, представленный в виде микросервиса поисковый сервис, использующий библиотеку Spring Cloud, с большей вероятностью будет готов к такой интеграции.

4. Тестирование поискового сервиса

Для того чтобы понять реальную скорость работы сервиса, необходимо провести серию нагрузочных тестов. Сервисы протестированы с использованием приложения Postman [22]. Каждой службе отправлялось по 1000 запросов в секунду в течение 10 с. Сервис был развернут в контейнере docker на виртуальных машинах Linux с различной производительностью процессора и объемом оперативной памяти. Кроме того, сервис был подключен к базе данных, содержащей более 10 000 записей.

4.1 План тестирования поискового сервиса

1. Подключить сервис динамического поиска к базе данных.
2. Создать образ службы в docker.
3. Развернуть службу на разных компьютерах, настроенных по-разному, с точки зрения производительности и загруженных с уже запущенным демоном Docker [18].
4. Выполнить в Postman 1000 запросов в секунду в течение 10 с.

4.2 Результаты тестирования поискового сервиса

Сервис был протестирована на различных конфигурациях серверов (табл. 1).

Как видно из табл. 1, цифры практически одинаковы для любой конфигурации, дольше всего реагировала

виртуальная машина с одним гигабайтом оперативной памяти и одним процессорным ядром. На рис. 7 показана разница между аппаратными конфигурациями. Наилучшими вариантами использования памяти являются варианты с 1, 2, 4 и 6 гигабайтами соответственно. Поэтому именно эти конфигурации использовались для сравнительных тестов. В табл. 2 показано время отклика службы динамического поиска с алгоритмом поиска по умолчанию, где поиск в базе данных выполняется с помощью собственных SQL-запросов, и с использованием алгоритма маскирования.

В табл. 1 представлены результаты времени отклика для различных открытых поисковых источников. Из этих результатов видно, что максимальное время отклика значительно отличается от среднего. Это время было занято первыми запросами, которые запустили процесс кэширования, из-за чего время отклика немного увеличилось. Для более наглядного сравнения значений скорости отклика поисковых сервисов, построены графики (рис. 8(а), 8(б), 9).

Таблица 1. Время отклика в миллисекундах службы динамического поиска для различных конфигураций сервера

CPU Ядра / RAM память	1 GB	2 GB	4 GB	6 GB
1	395	-	-	-
2	-	387	381	385
4	-	373	383	379
6	-	370	372	368

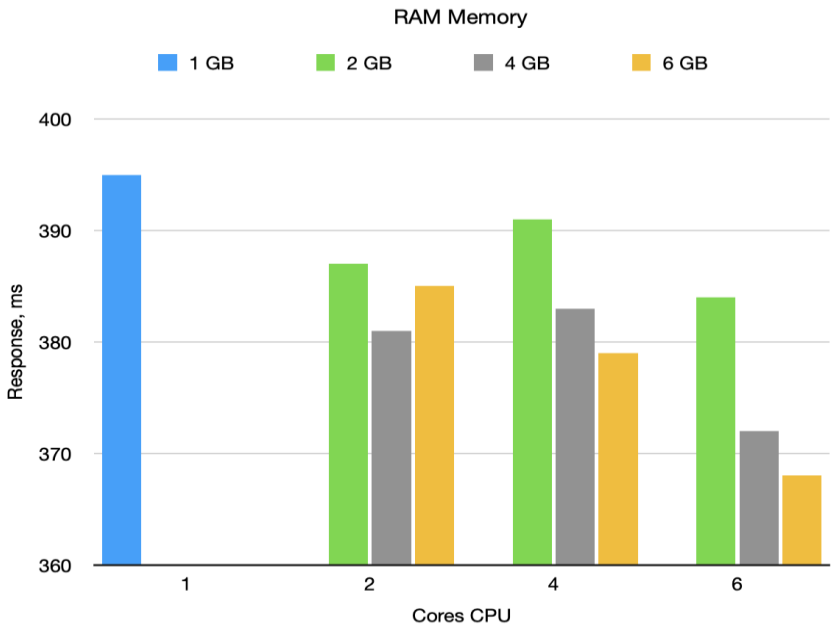

Рис. 7. Различные конфигурации виртуальной машины для службы динамического поиска

Как видно из рис. 8(а) и 8(б), аппаратное обеспечение, на котором запущена служба, не оказывает существенного влияния на время ее отклика. Похожие показатели видны уже на тестах с 4-мя ГБ оперативной памяти и 2-мя ядрами процессора. Из рис. 8(а) и 8(б) также можно сделать вывод, что скорость работы сервиса с алгоритмом маскирования выше, чем у сервиса с обычными SQL-запросами, что подтверждает предположение о том, что алгоритм маскировки может работать быстрее чем обычный поиск.

Таблица 2. Данные нагрузочного тестирования

Сервисы	CPU Ядра	RAM память, GB	Время ответа сервиса, мс		
			Минимальное	Среднее	Максимальное
Служба динамического поиска	1	1	282	395	987
	2	4	271	381	495
	4	6	285	379	498
	6	6	283	368	508
Служба динамического поиска (алгоритм маскирования)	1	1	226	392	954
	2	4	203	371	480
	4	6	214	367	386
	6	6	199	350	379
Служба A	-	-	247	366	526
Служба B	-	-	683	1094	2436
Служба C	-	-	780	945	2150

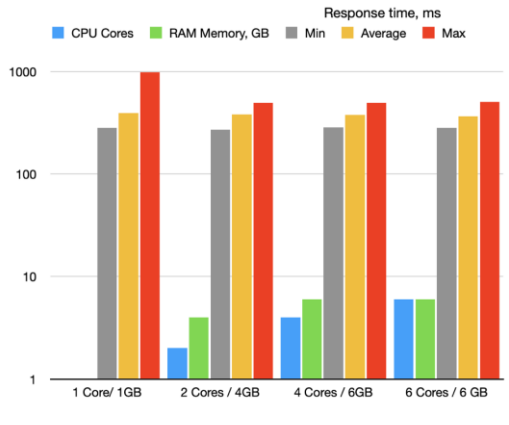

Рис. 8(а). Время отклика службы динамического поиска в зависимости от конфигурации виртуальной машины

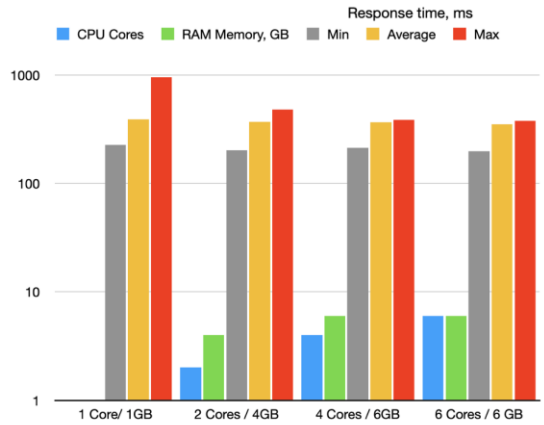

Рис. 8(б). Время отклика службы динамического поиска (алгоритм маскировки) в зависимости от конфигурации виртуальной машины

На рис. 9 показано, что сервис с алгоритмом маскировки оказался самым быстрым по сравнению с другими протестированными сервисами.

Также видно, что сервис с алгоритмом маскировки превосходит все другие сервисы, хотя по скорости он близок к поисковой службе A.

Имеются и сервисы, которые работают достаточно медленно – с максимальным временем отклика, превышающим 2 с.

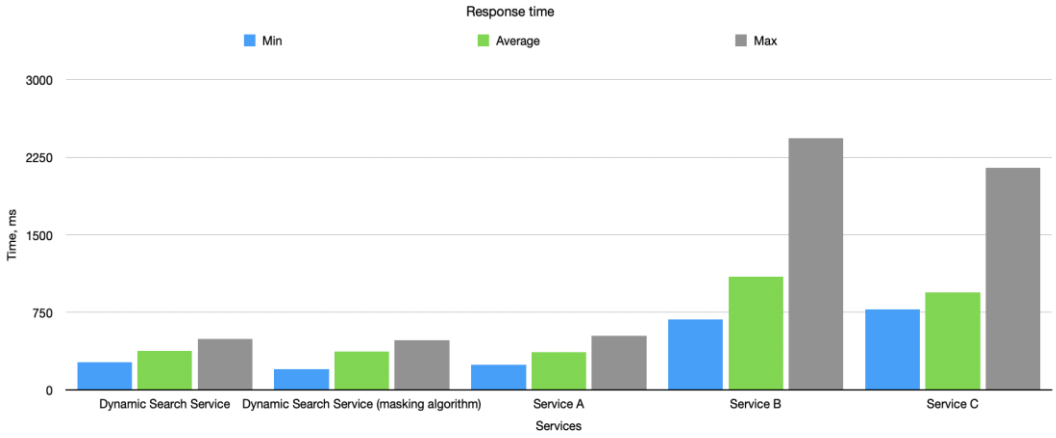

Рис. 9. Результаты сравнения значений времени отклика в различных реализациях поисковых сервисов

5. Заключение

Разработка поискового сервиса требует знаний об устройстве баз данных, способах взаимодействия с ними, типах сущностей, существующих в ней и т.д. Также разработчику необходимо уметь корректно применять индексирование баз данных. Кроме того, кэширование данных требует знания подходов и методов временного хранения данных. Например, для внешнего кэширования необходимо знать CDN-службы. Необходимо также понимать механизмы кэширования данных, которые могут ускорить процессы поиска.

В статье представлен разработанный авторами поисковый сервис, который оказался достаточно эффективным даже без применения алгоритма маскировки, а при использовании последнего скорость поиска увеличивается примерно на 15 %. Отмечено, что конфигурация сервера, на котором может быть запущена служба динамического поиска, не имеет большого значения в части производительности.

Нагрузочное тестирование представленной реализации сервиса с алгоритмом маскировки даже при минимальной конфигурации аппаратуры показало весьма хорошие результаты, что позволяет его использовать даже на бюджетных серверах.

Данная статья является результатом исследовательского проекта, реализованного в рамках программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ).

Список литературы

1. Ruben Torres, Alessandro Finamore, Jin Ryong Kim, Marco Mellia, Maurizio M. Munaf, and Sanjay G. Rao. Dissecting video server selection strategies in the youtube cdn. In ICDCS. IEEE Comput- er Society, 2011 https://ieeexplore.ieee.org/document/5961681
2. How Netflix loads videos so fast? https://medium.com/deciphering-deadlocks/how-netflix-loads-videos-so-fast-2cef815e35d8
3. The Netflix Cosmos Platform https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad
4. Carnell, J., Spring Microservices in Action. [S.l.]: O'Reilly Media, 2020
5. Why Should You Use Java For Your Backend Infrastructure? https://www.devteam.space/blog/why-should-you-use-java-for-your-backend-infrastructure/
6. SpringBoot performance testing various Embedded Web Servers https://medium.com/@skhatri.dev/springboot-performance-testing-various-embedded-web-servers-7d460bbfdb1b
7. 7+ Million HTTP requests per second from a single server https://www.ageofascent.com/2019/02/04/asp-net-core-saturating-10gbe-at-7-million-requests-per-second/
8. Track, version, and deploy database changes https://www.liquibase.org
9. Version control for your database https://flywaydb.org
10. Database Migration tools: Flyway vs Liquibase https://dzone.com/articles/flyway-vs-liquibase
11. The Reactive Relational Database Connectivity https://r2dbc.io
12. Spring: Blocking vs non-blocking: R2DBC vs JDBC and WebFlux vs Web MVC https://clck.ru/38aSB4

13. Saqib Hakak, Amirrudin Kamsin, Palaiahnkote Shivakumara, Gulshan Amin Gilkar. Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. IEEE Comput- er Society, 2016 https://clck.ru/38aS8c
14. Gaston H. Gonnet, Ricardo Baeza-Yate. A new approach to text searching, 1988. https://clck.ru/38aSGA
15. Understanding the Levenshtein Distance Equation for Beginners https://clck.ru/38aSTf
16. Thomas, V. Cloud Native Spring in Action. Shelter Island: Manning, 2021
17. Benefits of containerization https://circleci.com/blog/benefits-of-containerization/
18. Docker overview https://docs.docker.com/get-started/overview/
19. Why is Docker so Popular https://www.section.io/engineering-education/why-is-docker-so-popular/
20. How to explain Kubernetes in plain English https://clck.ru/38aSNF
21. How Kubernetes used in industries Kubernetes used in industries. https://resosanchit.medium.com/research-on-kubernetes-8bd8b66d9948
22. Postman https://www.postman.com

References

1. Ruben Torres, Alessandro Finamore, Jin Ryong Kim, Marco Mellia, Maurizio M. Munaf, and Sanjay G. Rao. Dissecting video server selection strategies in the youtube cdn. In ICDCS. IEEE Comput- er Society, 2011 https://ieeexplore.ieee.org/document/5961681
2. How Netflix loads videos so fast? https://medium.com/deciphering-deadlocks/how-netflix-loads-videos-so-fast-2cef815e35d8
3. The Netflix Cosmos Platform https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad
4. Carnell, J., Spring Microservices in Action. [S.l.]: O'Reilly Media, 2020
5. Why Should You Use Java For Your Backend Infrastructure? https://www.devteam.space/blog/why-should-you-use-java-for-your-backend-infrastructure/
6. SpringBoot performance testing various Embedded Web Servers https://medium.com/@skhatri.dev/springboot-performance-testing-various-embedded-web-servers-7d460bbfdb1b
7. 7+ Million HTTP requests per second from a single server https://www.ageofascent.com/2019/02/04/asp-net-core-saturating-10gbe-at-7-million-requests-per-second/
8. Track, version, and deploy database changes https://www.liquibase.org
9. Version control for your database https://flywaydb.org
10. Database Migration tools: Flyway vs Liquibase https://dzone.com/articles/flyway-vs-liquibase
11. The Reactive Relational Database Connectivity https://r2dbc.io
12. Spring: Blocking vs non-blocking: R2DBC vs JDBC and WebFlux vs Web MVC https://clck.ru/38aSB4
13. Saqib Hakak, Amirrudin Kamsin, Palaiahnkote Shivakumara, Gulshan Amin Gilkar. Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. IEEE Comput- er Society, 2016 https://clck.ru/38aS8c
14. Gaston H. Gonnet, Ricardo Baeza-Yate. A new approach to text searching, 1988. https://clck.ru/38aSGA
15. Understanding the Levenshtein Distance Equation for Beginners https://clck.ru/38aSTf
16. Thomas, V. Cloud Native Spring in Action. Shelter Island: Manning, 2021
17. Benefits of containerization https://circleci.com/blog/benefits-of-containerization/
18. Docker overview https://docs.docker.com/get-started/overview/
19. Why is Docker so Popular https://www.section.io/engineering-education/why-is-docker-so-popular/
20. How to explain Kubernetes in plain English https://clck.ru/38aSNF
21. How Kubernetes used in industries. https://resosanchit.medium.com/research-on-kubernetes-8bd8b66d9948
22. Postman https://www.postman.com