Москва 2020

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет

«Высшая школа экономики»

Факультет компьютерных наук

Основная образовательная программа
«Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему
Интеграция в ClickHouse функциональности обработки HTTP User Agent

Выполнил студент группы 166, 4 курса,

Филитов Михаил Егорович

Руководитель ВКР:

Руководитель группы разработки СУБД ClickHouse,
Миловидов Алексей Николаевич

Содержание

Аннотация

Заголовок User-Agent HTTP-запроса передается при каждом запросе от клиента к серверу. Эта строка описывает детали конфигурации клиента и помогает серверу убедиться, что возвращаемый контент подходит для запроса хоста. Анализ этого заголовка позволяет компаниям распознавать целевую аудиторию обслуживания и предоставлять лучший сервис, используя эту информацию. Более того, User-Agent помогает описывать специфичные для платформы проблемы. Именно поэтому этот заголовок широко используется в аналитических отделах компаний, предоставляющих онлайн-услуги. Из-за большого количества запросов компаниям приходится хранить все данные запросов в специализированной базе данных. ClickHouse - одна из баз данных, сочетающая в себе эффективную систему хранения и аналитические возможности. В этой статье я описываю свое исследование парсинга UserAgent, представляю план внедрения такого парсера в аналитическую базу данных ClickHouse и описываю реализацию.

Работа содержит 27 страниц, 4 главы, 10 рисунков, 14 источников.

Ключевые слова — Кликхаус, Бэкенд, Юзер-агент Заголовки, база данных, аналитика

Abstract

User-Agent header of HTTP request is transmitted during each request from client to server. This string describes client configuration details and helps server to ensure that returned content is appropriate for requesting host. Analysis of this header allows companies to recognize the target audience of service and provide better service using this information. Moreover, User-Agent helps to describe platform-specific problems. That is why this header is widely used in analytics departments of companies which provide online services. Due to large number of requests this companies have to store all the requests data in specialized database. One of databases combining effective storage system and analytical capabilities is ClickHouse. In this paper I describe my study of UserAgent parsing, represent the plan of implementing such parser in analytical database ClickHouse and describe the details of realization.

The work contains 27 pages, 4 chapters, 10 drawings, 14 sources.

Keywords - Clickhouse, Backend, User-Agent, Headers, database, analytics

Используемые термины

1. Регулярное выражение (regular expression) - формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. Для поиска используется строка-образец, состоящая из символов и метасимволов и задающая правило поиска.
2. База данных (database) - это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных (СУБД). Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных, или, для краткости, просто базой данных.
3. Кэш (cache) - промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Доступ к данным в кэше осуществляется быстрее, чем выборка исходных данных из более медленной памяти или удалённого источника, однако её объём существенно ограничен по сравнению с хранилищем исходных данных.
4. Маппинг (mapping) - определение соответствия данных между потенциально различными семантиками одного объекта или разных объектов.
5. Юнит тестирование (unit testing) - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии*, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.
6. Компилятор (compiler) – программа, принимающая в себя код, написанный на высокоуровневом языке программирования, и переводит его в машинный код.
7. Open source – программное обеспечение с открытым исходным кодом.

Введение

Разработчики приложений и контента сталкиваются со значительными проблемами, связанными с различными конфигурациями клиентских систем. Конфигурации могут быть заданы браузером, используемым клиентом, операционной системой, оборудованием. Правильное определение User-Agent имеет решающее значение для предоставления правильно отформатированного контента и обеспечения наилучшего пользовательского опыта[1]. Поставщики контента обычно используют сложные регрессионные тесты, чтобы убедиться, что контент ведет себя корректно на всем разнообразии платформ. Существует общий механизм, который помогает распознавать характеристики хоста клиента: обработка строки UserAgent, передаваемой в виде заголовка с запросом на сервер. Многие интернет-приложения используют User-Agent в основном для согласования контента. Примерами таких приложений являются: веб-сканеры, боты, мобильные приложения, браузеры. Количество различных UserAgents огромно: каждая возможная комбинация модели телефона, браузера, поколения процессора, установленной операционной системы может формировать корректную строку UA. Более того, несмотря на строгие форматы этого заголовка, описанные в специализированных документах RFC (Request for Comments) RFC 1945 [2] и еще раз здесь RFC7231 [3], UA может отличаться в деталях.

Количество веб-разработчиков, которые используют этот заголовок для распознавания конфигурации хоста клиента для аналитики или для предоставления подходящего ответа большое, необходимость обработки заголовка возникает практически в любом современном приложении. Вот почему они требуют общего решения для своих нужд. Это может быть синтаксический анализ библиотеки или инфраструктуры, поддерживаемой членами сообщества, может быть специализированным отдельным сервером, может быть базой данных.

Clickhouse - это база данных, разработанная для интернет-аналитики. Она очень эффективна обрабатывает большие объемы данных, и является проектом с

открытым исходным кодом [4], что означает, что каждый разработчик может улучшить его или клонировать и изменить код для своих запросов. Также этот проект поддерживается разработчиками Яндекса, и пользователи могут сообщить о проблеме с запросом функции или с объявлением ошибки. Одним из таких запросов является добавление поддержки разбора строки User-Agent на части: версии процессора, операционной системы и т.д. Кроме этого, для сохранения ее в виде отдельных объектов. Цель моего проекта - удовлетворить этот запрос и предоставить самое быстрое решение для поддержания высокой скорости обработки.

В рамках данной выпускной квалификационной работы были поставлены 3 цели:

1. Исследовать существующие библиотечные решения для обработки заголовка User-Agent.
2. На основе проведенного анализа выбрать наиболее подходящее решение и разработать план разработки решения.
3. В соответствие с планом реализовать выбранное решение.

Для достижения первой цели необходимо выполнить следующие задачи:

1. Изучить аналоги: существующие библиотечные решения, существующие подходы к решению задачи.
2. Выявить сильные и слабые стороны реализаций, при необходимости написать тесты, проверяющие производительность или корректность работы.
3. На основы выявленных плюсов и минусов реализаций разработать требования к финальной реализации.

Для достижения второй цели необходимо выполнить следующие задачи:

1. Изучить существующий программный код в Clickhouse.
2. Составить план реализации функционала, основываясь на структуре существующего кода.

Для достижения третьей цели необходимо выполнить следующие задачи:

1. Построить минимальную работающую версию решения.
2. Улучшить решение, используя найденные плюсы и минусы у существующих реализаций.
3. Написать тесты решения.

Работа структурирована следующим образом:

В первой главе проведен анализ существующих решений, описание реализаций и выводы об их сильных сторонах, а также выводы о требованиях к реализуемому подходу.

Во второй главе фиксируется техническое задание и описывается план реализации от простого базового функционала к более продвинутому.

В третьей главе описываются детали реализации, расписывается предложенное решение, с учетом стиля и структуры исходного кода проекта.

В четвертой главе показывается отчет о тестировании, показываются и объясняются написанные тесты, основанные на разработанных требованиях к реализации, и фиксируются результаты прохождения этих тестов.

1 Анализ существующих решений для обработки заголовка User-Agent

В данной главе будет проведен анализ существующих библиотечных решений для работы с заголовком User-Agent. Основной идеей обработки данного заголовка является сопоставление заголовка заранее заданным регулярным выражениям, позволяющее понять, из каких частей состоит заголовок. Важным аспектом является то, как подбираются регулярные выражения, поскольку соответствие регулярного выражения строке – достаточно дорогая операция.

1.1 Существующие библиотечные решение и их сравнение

1.1.1 Faisalman/ua-parser-js [5]

Это достаточно популярная библиотека для обработки заголовков User-Agent, имеющая больше, чем 4600 звезд на github.com.

Данная библиотека не поддерживает добавление новых данных с регулярными выражениями, все данные находятся непосредственно в программном коде (как можно увидеть на Рис. 1.1.1.1), что означает, что при появлении новых браузеров или процессоров нужно будет или руками менять код библиотеки, что является плохой практикой или ждать, пока авторы библиотеки добавят новые выражения, что может занять неопределенное время. Эти регулярные выражение записаны в самом исходном коде и не загружаются ниоткуда, что является неподходящим методом реализации, поскольку добавление нового правила обработки заголовка существенно осложнено. Оно требует изменения кода.

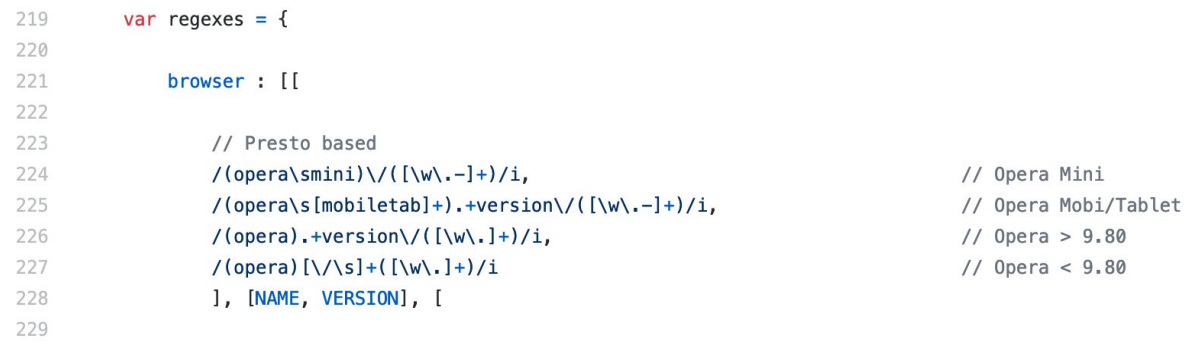

Рис. 1.1.1.1 Фрагмент кода библиотеки, содержащий регулярные выражения [6]

Язык, используемый в библиотеке, – JavaScript. Поскольку Clickhouse написан с использованием C++, данное решение не подходит в существующем виде, поскольку будет требоваться существенная переработка.

Данная реализация не использует кэш обработанных данных, таким образом для обработки типовых заголовков работа будет проделываться каждый раз заново. Кроме того, реализация выполнена таким образом, что для получения составных частей заголовка необходимо обрабатывать его заново для каждой части, что тоже является недопустимым при работе с большими объемами данных.

Реализованный алгоритм – последовательный проход по всем заданным регулярным выражениям. Такой подход является не самым эффективным, поэтому тоже не подходит. Как можно увидеть на Рис. 1.1.1.2 для строки проверяется соответствие каждого регулярного выражения по очереди.

```
// loop through all regexes maps
while (i < arrays.length && !matches) {

    var regex = arrays[i],       // even sequence (0,2,4,..)
        props = arrays[i + 1];   // odd sequence (1,3,5,..)
    j = k = 0;

    // try matching uastring with regexes
    while (j < regex.length && !matches) {

        matches = regex[j++].exec(ua);
```

Рис. 1.1.1.2 Фрагмент кода библиотеки, описывающий проход по всем регулярным выражениям [7].

Покрытие тестами недостаточное, есть тесты на функционал получения нужного поля из строки, но нет тестов производительности, нет unit тестов, проверяющих работу отдельных компонентов библиотеки.

1.1.2 Matomo-org/device-detector [8]

Менее популярная библиотека, имеющая 1700 звезд на github.com, однако более подходящая нам, чем предыдущая.

Формат данных – частично yml, частично запись в исходном коде, аналогично предыдущей библиотеке. Например, поисковые боты обрабатываются на основе регулярных выражений из файла bots.yml, в то время как браузеры обрабатываются по данным из файла browsers.yml и записанным в коде маппингам браузеров на сокращения, что усложняет дополнение или изменение существующих правил обработки. Как можно видеть на Рис. 1.1.2.1 соответствия названий и внутренних сокращений прописано в коде, что при необходимости добавления новых браузеров требует изменения программного кода библиотеки. Такое решение не подходит.

```
29      protected static $availableBrowsers = array(
30          '1B' => '115 Browser',
31          '2B' => '2345 Browser',
32          '36' => '360 Phone Browser',
33          '3B' => '360 Browser',
34          'AA' => 'Avant Browser',
35          'AB' => 'ABrowse',
36          'AF' => 'ANT Fresco',
37          'AG' => 'ANTGalio',
```

Рис. 1.1.2.1 используемые в библиотеке сокращения для названий. [9]

Язык, используемый в библиотеке, - PHP, аналогично предыдущей библиотеке он не подходит для использования библиотеки.

Кэш есть, однако, присутствует проблема из предыдущей библиотеки – для получения каждого поля из строки заголовка необходимо обрабатывать строку заново. Что негативно сказывается на эффективности работы алгоритма.

Реализованный алгоритм – последовательное применение известных регулярных выражений для заголовка. Аналогично предыдущему решению нам не подходит.

1.1.3 Ua-parser/uap-cpp [10]

Менее популярная библиотека, чем предыдущие две. На сайте github.com она имеет 26 звезд. Низкая популярность библиотеки может быть связана как с низким качеством реализации, так и с невысокой популярностью используемых технологий, поэтому данное решение будет рассматриваться в рамках анализа.

Используемый формат данных для хранения регулярных выражений – yaml. Данный формат активно используется в разработке и подходит для работы с регулярными выражениями, поскольку они не требуют сложной структуры для хранения данных. В отличие от предыдущих библиотек данные берутся только из дополнительных файлов и не находятся внутри кода, что делает изменение данных или их дополнение простым и не требующим внутренних изменений реализации библиотеки.

Библиотека написана с использованием языка программирования C++, что является тем же языком, что используется в Clickhouse, это делает возможным использование библиотеки по прямому назначению без переписывания.

Кэш не используется, однако для получения всех полей строка обрабатывается один раз и дальше работа происходит с объектом. Что положительно сказывается на времени работы, когда из одного заголовка нужно достать разные данные об устройстве, например, название браузера и операционную систему.

В библиотеки есть тесты, проверяющие производительность и работу отдельных функций через unit тестирование. Однако тесты производительности не сравнивают решение с существующими, а только пишут новую скорость обработки заданных строк, что является важным фактором при разработке библиотеки, дающим понимание о производительности при добавлении нового функционала, но не играют большой роли для пользователей библиотеки. Также

есть результаты проверки на разных процессорах и с использованием разных компиляторов, результаты можно увидеть на Рис. 1.1.3.1.

Performance benchmarks

Results of `make bench` on two different machines:

Processor	Compiler	Real time	User CPU time	System CPU time
Intel Core i7 2.2GHz	AppleClang 10.0.1	39.57	39.50	0.04
Intel N3700 1.6GHz	GCC 8.3	98.79	98.75	0.02

Рис. 1.1.3.1 результаты тестирования производительности

Поскольку в данном тесте были одновременно изменены процессоры и компиляторы, то однозначно сказать нельзя, что именно вызвало разные результаты теста, из-за данного факта польза данных результатов теста сводится к нулю.

Реализация – линейный проход по заданным регулярным выражениям, аналогичный предыдущим решениям. Однако, в данной библиотеке присутствует оптимизация, позволяющая сократить количество регулярных выражений, которые нужно пройти. Для работы этой оптимизации поддерживается индекс подстрок, которые должны присутствовать в регулярных выражениях, с помощью этого можно построить дерево с такими подстроками, которое позволит избежать необходимости проверять все регулярные выражения, а проверить только те, в которых содержатся необходимые подстроки. Данная оптимизация является фильтром и отбрасывает заведомо неподходящие выражения из обработки.

1.1.4 Highpower/uatraits [11]

Данная библиотека имеет 18 звезд на github.com и написана людьми, которые писали аналогичную библиотеку для внутреннего использования в компании Яндекс.

В качестве формата входных данных используется XML. Этот формат давно используется в разработке, однако сейчас становится менее популярным. Также этот формат используется внутри ClickHouse, что делает его удобным для работы. Данные с регулярными выражениями берутся из XML файла, что позволяет удобно изменять данные при необходимости.

Язык, используемый в библиотеке, - C++ - совпадает с языком, на котором разрабатывается ClickHouse.

Большое количество тестов, покрывающих как работу отдельных функций, так и обработки строк в целом.

Кэш не используется. Обработка одной строки заголовка осуществляется один раз, после чего используется объект.

Реализация – используется дерево, составленное из регулярных выражений. Такое решение позволяет на каждом этапе обработки строки отбрасывать много неподходящих регулярных выражений и существенно сократить время обработки одного заголовка.

Существует минус – данная библиотека в данный момент не поддерживается, последние правки были внесены в 2012 году. Также данная библиотека писалась давно из-за чего в ней присутствует реализация объектов, которые сейчас являются частью языка C++, например, shared_ptr.

1.1.5 Внутренняя библиотека Яндекса (uatraits-fast)

Данная библиотека является улучшенной версией библиотеки, описанной выше, сейчас она поддерживается и находится в разработке.

Аналогично предыдущей библиотеке используется XML и данные о регулярных выражениях берутся из файлов.

Язык, используемый в библиотеке, - C++ - подходит.

Кэш используется, что позволяет некоторые наиболее частые выражения не обрабатывать по нескольку раз, а использовать уже готовые результаты. Обработка одной строки заголовка осуществляется один раз, после чего используется объект.

Реализация – используется алгоритм Ахо-Корасик. Это позволяет проходить не все регулярные выражения для обработки строки, а поэтапно сокращать их количество.

Существует минус – данную библиотеку не предназначена для работы с open-source проектом и многие части необходимо будет переделать, к примеру зависимости. Кроме этого, в ней есть функционал, который не требуется в ClickHouse. Есть большое количество тестов, покрывающий как функционал отдельных частей (unit tests), так и функционал работы всего парсера.

1.2 Требования к финальной реализации

Решение должно быть написано на языке C++ для того, чтобы быть совместимым с уже существующим кодом в ClickHouse.

Реализация не должна проходить по всем регулярным выражениям подряд, должно быть эффективное отсеивание неподходящих выражений.

Данные для регулярных выражений должны браться из файла и загружаться в программу во время старта сервера. Для работы подходят как XML, так и YML форматы.

Использование кэша может присутствовать, а может отсутствовать.

Реализация должна содержать тесты на функционал и тесты на производительность.

1.3 Вывод

Таким образом результатом исследования существующих решений является выявление и установление ключевых особенностей, которые должны быть учтены в выбранном библиотечном решении или собственной реализации. Анализ решений показал, что нет библиотеки, которую можно использовать в проекте без дополнительных модификаций кода. В ходе сравнения работы различных реализаций обработки HTTP заголовка User-Agent были рассмотрены разные подходы к обработке строк, найдены сильные и слабые стороны решений, проанализированы возможные оптимизации, присутствующие в библиотечных решениях, некоторые из них возможно использовать в финальной реализации.

2 Выбор методов решения

В данной главе будет описано, выбранное решение. Будут приведены аргументы в пользу выбранного решения и написано подробное описание характеристик решения. Также будет приведен план реализации решения, начинающийся от более простого функционала к более сложному.

2.1 Выбор методов решения. Обоснование выбора

Для решения поставленной задачи будет использоваться в качестве основы последняя описанная библиотека uatraits-fast, поскольку она удовлетворяет описанным выше требованиям реализации, также она поддерживается и у нее гарантированно есть пользователи. Данная библиотека будет модифицирована: будет удален функционал, который не подходит для ClickHouse, внутренние зависимости будут переписаны, недостающие классы реализованы. В качестве движка для работы с регулярными выражениями будет использоваться не алгоритм Ахо-Корасик, а библиотека hyperscan [12]. Таким образом для обработки HTTP заголовка User-Agent в Clickhouse будет создано собственное решение, использующее сильные стороны описанных выше библиотек.

В качестве языка программирование выбран C++, поскольку это основной язык, на котором разрабатывается Clickhouse. Регулярные выражения для работы будут храниться в формате XML, поскольку этот формат используется при разработке ClickHouse. Будет использоваться кэш для частых заголовков. Данное решение оправданно, поскольку существуют заголовки, которые встречаются сильно чаще других[13]. Для тестов предлагается использовать проверки, аналогичные реализованным в перечисленных библиотеках, дополнив их недостающими тестами и более продвинутыми тестами производительности.

2.2 Почему Hyperscan

Hyperscan – библиотека, разработанная в компании Intel, предназначенная для быстрой обработки больших объемов регулярных выражений. Работа с выражениями делится на 2 этапа: компиляция и сканирование. Во время

компиляции функции принимают группу регулярных выражений вместе с идентификаторами и флагами параметров и компилируют их в неизменную базу данных, которая может использоваться API-интерфейсом сканирования Hyperscan. Этот процесс компиляции выполняет значительную работу по анализу и оптимизации для создания базы данных, которая будет эффективно соответствовать заданным выражениям. Скомпилированные базы данных могут быть сериализованы и перемещены [14], так что они могут быть сохранены на диск или перемещены между хостами. Они также могут быть собраны под определенные платформы (например, использование инструкций Intel® Advanced Vector Extensions 2 (Intel® AVX2)). Данная возможность позволяет сократить время старта приложения – не нужно будет каждый раз тратить время на компиляцию базы данных, можно будет ее просто загрузить, кроме этого при использовании нескольких инстансов сервисов можно будет выполнять компиляцию только один раз, что существенно уменьшит время старта.

После создания базы данных Hyperscan ее можно использовать для сканирования данных в памяти. Hyperscan предоставляет несколько режимов сканирования, в зависимости от того, доступны ли данные для сканирования в виде единого непрерывного блока, распределены ли они по нескольким блокам в памяти одновременно, или же они должны сканироваться как последовательность блоков в поток.

Данная библиотека позволяет быстро обрабатывать регулярные выражения и позволяет уменьшить время компиляции выражений, поэтому ее использование вместе с библиотекой по обработке User-Agent является целесообразным.

2.3 План реализации

1. Простое добавление новой функции по получению браузера из строки User-Agent. На данном этапе не требуется, чтобы данные читались с диска, важно соблюсти правила добавления новых функций, научиться использовать hyperscan и получить работающий функционал. При

старте сервера должен создаваться класс с базой данных регулярных выражений и передаваться в функцию с использованием контекста.

2. Добавление uatraits-fast в работу функции. Этот шаг необходим для возможности реализации более сложных функций, например, получения всех полей из строки User-Agent.
3. Добавление дополнительных функций для работы с User-Agent: получение каждого поля из заголовка, получение всех полей сразу.
4. Написание тестов на функционал и на производительность функции (выбрать набор заголовков и регулярных выражений, проверить, как быстро получится их найти).
5. Добавление кэшированных результатов при необходимости, проверка улучшения скорости работы на написанных тестах производительности.

2.4 Вывод

В данной главе было описано выбранное решение и обоснованы критерии этого выбора. За основу решения выбрана внутренняя библиотека Яндекса, которую необходимо будет переделать, добавив необходимый функционал и реализовав недостающие зависимости. В качестве движка для поиска и обработки регулярных выражений выбрано библиотечное решение Hyperscan, разработанное компанией Intel. Также составлен план реализации необходимого функционала. План содержит в себе шаги, идущие в порядке по увеличению сложности шагов и увеличения итоговой кодовой базы, таким образом данный план поможет легче начать разработку в новом проекте с использованием простой реализации. Кроме этого, придерживаясь этого плана, остается возможность на ранних этапах реализации получать рекомендации относительно качества кода и возникающих ошибках без больших затрат со стороны руководителя, поскольку первоначальное решение не содержит в себе сложной логики и архитектурных решений.

3 Реализация

В этой главе описываются детали реализации. Классы, которые были добавлены с их описанием. А также принятые архитектурные решения. Под Функцией в данном контексте подразумевается класс, расположенный по пути Clickhouse/src/Functions, реализующий одну из функций, которую можно вызвать в клиентском приложении.

3.1 Основные классы

3.1.1 ExtractBrowserFromUserAgent

Для обеспечения работы функционала обработки необходимо добавить новую функцию, чтобы она работала как в clickhouse-server (сам сервер с базой данных), так и в clickhouse-client (клиентское приложение).

Поскольку для работы Функции нужно читать данные о регулярных выражениях и компилировать их, то необходимо проделать это один раз и дальше передавать эти данные в Функцию, поскольку на каждый запрос создается ее новый экземпляр и читать, и компилировать данные при каждом запросе нецелесообразно. Для этого необходимо прочитать данные при старте сервера и передавать их в Функцию.

Основная логика функции реализована в методе vector Рис. 3.1.1.1.

```
static void vector(const ColumnString::Chars & data,
                   const ColumnString::Offsets & offsets,
                   ColumnString::Chars & res_data,
                   ColumnString::Offsets & res_offsets, const Context & context)
```

Рис. 3.1.1.1 сигнатура метода vector

Данные в метод передаются непрерывным блоком из частей в поле data, offsets содержат отступы, позволяющие понимать, где граница между соседними частями. Также принимаются ссылки на аналогичные res_data и res_offsets, где после выполнения метода должны находиться итоговые измененные данные. Кроме этого, в метод передается context, в котором как раз и находится

заполненная база с регулярными выражениями, созданная при старте сервера. Также есть аналогичный метод vector_fixed, в котором принимается размер отступа и дальше работа выполняется в предположении, что все отступы одного заданного размера. Данный метод разделяет блок на части и для каждой части вычисляет, какой браузер может соответствовать переданной строке. После завершения вычислений записывает результат в соответствующую часть res_data и выставляет соответствующий offset.

Для того, чтобы написанная Функция могла быть вызвана, она должна быть зарегистрирована, должна быть реализована функция, которая будет принимать фабрику Функций и добавлять туда еще одну: Рис. 3.1.1.2.

```
void registerFunctionExtractBrowserFromUserAgent(FunctionFactory & factory)
{
    factory.registerFunction<extractBrowserFromUserAgent>( case_sensitiveness: FunctionFactory::CaseInsensitive);
}
```

Рис. 3.1.1.2 регистрация функции.

Также для работы Функции, нужно знать, какое название для этого нужно задать имя функции строкой и определить метод getName() (Рис. 3.1.1.3), который будет возвращать это имя. В дальнейшем этот метод будет вызываться при поиске подходящей Функции.

```
static constexpr auto name = "extractBrowserFromUserAgent";
String getName() const override
{
    return name;
}
```

Рис. 3.1.1.3 Получение имени функции.

Остальные Функции для обработки User-Agent имеют схожие свойства, отличаются типы данных названия Функции и логика в методе vector. Это функции: extractOsFromUserAgent, extractBrowserVersionFromUserAgent.

3.1.2 UserAgent

Основная логика по получению необходимых полей из строки User-Agent реализована в усовершенствованной библиотеке Uatraits-fast, которая располагается в кодовой базе Clickhouse по пути

Clickhouse/src/Common/UatraitsFast. Данная библиотека хранит в себе скомпилированную базу регулярных выражений, для заданной строки находит все регулярные выражения, которые соответствуют ей и дальше строит дерево из них по заранее заданным правилам, на основе которого подбирает искомое выражение. User-Agent – класс, который используется снаружи библиотеки для обработки строк (Рис. 3.1.2.1).

```
class UserAgent : public boost::serialization::singleton<UserAgent>
{
friend class boost::serialization::singleton<UserAgent>;

public:
    void create(const Poco::Util::AbstractConfiguration & config);
    Agent detect(const std::string & user_agent) const;
    void reload();
```

Рис. 3.1.2.1 Описание класса UserAgent

Данный класс имеет 3 основных метода: create – принимает файл конфигурации и загружает в память данные по путям, указанным в конфигурации, reload – перечитывает данные, если они были изменены, detect – принимает в себя строку User-Agent производит обработку строки и возвращает класс Agent (Рис. 3.1.2.2) из которого уже можно получить требуемые данные.

```
class Agent
{
friend class UserAgent;
public:
    const OperatingSystem & getOperatingSystem() const;
    const Browser & getBrowser() const;
```

Рис. 3.1.2.2 Описание класса Agent

В данном классе основные методы – getBrowser и getOperationgSystem, первый возвращает структуру с описанием браузера, из которых можно взять более детальную информацию, например, название или версию, второй аналогичную информацию о операционной системе.

3.1.3 Uatraits

Внутренний класс библиотеки, используемый классом UserAgent при вызове метода detect, в котором изменен алгоритм поиска регулярных выражений. Как писалось ранее для матчинга регулярных выражений используется библиотека Hyperscan.

```
DB::MultiRegexps::ScratchPtr smart_scratch(scratch);
std::deque<std::pair<UInt8 , UInt64>> search_result;

auto on_match = []([[maybe_unused]] unsigned int id,
                   unsigned long long from, // NOLINT
                   unsigned long long /* to */, // NOLINT
                   unsigned int /* flags */,
                   void * ctx) -> int
{
    reinterpret_cast<std::deque<std::pair<UInt8, UInt64>>*>(ctx)->push_back(std::make_pair(from, id));
    return 0;
};
UInt64 length = user_agent_lower.size;
err = hs_scan(
    regexps_engine->getDB(),
    user_agent_lower.data,
    length,
    0,
    smart_scratch.get(),
    on_match,
    &search_result);
```

Рис. 3.1.3.1 Упрощенное использование библиотеки Hyperscan

Приведенный пример показывает упрощенный код по взаимодействию с библиотекой Hyperscan. Выделяется заранее место для работы библиотеки smart_scratch, чтобы избежать лишних аллокаций памяти и копирования, дальше добавляется лямбда функция on_match, которая передается в hs_scan и будет вызвана, как только будет найдено регулярное выражение, которое удовлетворяет строке. В эту функцию будет передан индекс регулярного выражения и позиция в строке, на которой оно сработало, далее эти данные сохраняются в очередь search_result, которая после завершения работы hs_scan будет содержать все совпадения регулярных выражений с заданной строкой. Далее эти данные отправляются на постобработку, где вычисляется, какие из сработавших регулярных выражений нужны и на основе этого возвращается найденная информация.

3.2 Вывод

В данной части были описаны детали реализации основных добавленных классов, которые используются при обработке User-Agent. Также было в общих чертах описан процесс добавления новых Функций и обозначен путь, который проходит запрос от Функции до ответа.

4 Заключение

В ходе работы были проанализированы различные подходы к обработке строки User-Agent в существующих библиотечных решениях. Было изучено 5 различных библиотек, внимательно проанализирован их исходный код и сделаны выводы о оптимальности работы алгоритма и существующих изъянах, на основе этого анализа выбрано итоговое решение для реализации.

Результатом работы над проектом является реализованный функционал обработки заголовка HTTP запроса User-Agent. Поддерживаются функции: extractBrowserFromUserAgent, extractBrowserVersionFromUserAgent, extractOSFromUserAgent, также реализована экосистема для добавления новых функций, таким образом данное решение может быть относительно просто расширено для обработки дополнительных полей заголовка User-Agent. После интеграции данной задачи у пользователей ClickHouse появляется возможность использования встроенных функций для обработки User-Agent, которые реализованы эффективно и позволяют получать необходимые поля из заголовка.

5 Библиография

x

1. Kline J. [и др.]. On the structure and characteristics of user agent string // Proceedings of the 2017 Internet Measurement Conference. 2017.

2. Fielding R., Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content // tools.ietf.org [Электронный ресурс]. URL: https://tools.ietf.org/html/rfc7231#section-5.5.3 (дата обращения: 20.05.2020).
3. Frystyk H., Berners-Lee T., Fielding R.T. Hypertext Transfer Protocol -- HTTP/1.0 // tools.ietf.org [Электронный ресурс]. URL: https://tools.ietf.org/html/rfc1945#section-10.15 (дата обращения: 20.05.2020).

4. Yandex, Clickhouse documentation // GitHub [Электронный ресурс]. URL: https://github.com/ClickHouse/ClickHouse (дата обращения: 20.05.2020).

5. Salman F. faisalman/ua-parser-js // GitHub [Электронный ресурс]. URL: https://github.com/faisalman/ua-parser-js (дата обращения: 20.05.2020).

6. Faisalman/ua-parser-js // GitHub [Электронный ресурс]. URL: https://github.com/faisalman/ua-parser-js/blob/master/src/ua-parser.js#L224 (дата обращения: 20.05.2020).

7. Faisalman/ua-parser-js/ // GitHub [Электронный ресурс]. URL: https://github.com/faisalman/ua-parser-js/blob/master/src/ua-parser.js#L224.

8. Matomo-org/device-detector // GitHub [Электронный ресурс]. URL: https://github.com/matomo-org/device-detector (дата обращения: 20.05.2020).

9. Matomo-org/device-detector // GitHub [Электронный ресурс]. URL: https://github.com/matomo-org/device-detector/blob/master/Parser/Client/Browser.php#L30 (дата обращения: 20.05.2020).

10. ua-parser/uap-cpp // GitHub [Электронный ресурс]. URL: https://github.com/ua-parser/uap-cpp (дата обращения: 20.05.2020).

11. Obolenskiy O. highpower/uatraits // GitHub [Электронный ресурс]. URL: https://github.com/highpower/uatraits (дата обращения: 20.05.2020).

12. Performance Considerations — Hyperscan 5.2.1 documentation // intel.github.io [Электронный ресурс]. URL: http://intel.github.io/hyperscan/dev-reference/performance.html (дата обращения: 20.05.2020).

13. Browse our database of 30.5 million User Agents // WhatIsMyBrowser.com [Электронный ресурс]. URL: https://developers.whatismybrowser.com/useragents/explore (дата обращения: 20.05.2020).

14. Introduction — Hyperscan 5.2.1 documentation // intel.github.io [Электронный ресурс]. URL: http://intel.github.io/hyperscan/dev-reference/intro.html#intro (дата обращения: 20.05.2020).