Moscow 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

“Инфраструктура распределённой трассировки для ClickHouse”

Выполнил студент группы 166, 4 курса,

Кожихов Александр Олегович

Руководитель ВКР:

Приглашенный преподаватель,

Руководитель группы разработки ClickHouse,

Миловидов Алексей Николаевич

Куратор:

< степень>, <звание>, <ФИО полностью>

Аннотация

В современных распределенных системах очень важно иметь удобные и информативные инструменты для наблюдения за состоянием системы, как предоставляющие необходимые данные в режиме реального времени, так и спустя время. Отсутствие таких инструментов сильно осложняет поиск узких мест, ошибок и ограничивает общее понимание происходящего в системе. В данной работе рассматривается один из таких инструментов - распределенная трассировка запросов - для системы управления базами данных ClickHouse. Данный метод позволяет понимать логику выполнения запроса, в том числе распределенного и в том числе неограниченного ClickHouse. В работе реализованы и опробованы генеричные методы, дающие возможность внедрять трассировку запроса в желаемые компоненты ClickHouse. Для агрегации данных и визуализации трассировки используется система Jaeger Tracing. В работе будет продемонстрирована распределенная трассировка на основе SELECT запроса, в которой будет возможность наблюдать за его логикой с выделенными фазами запроса, а также с более мелкогранулярными действиями, такими как операции обращения к диску.

Abstract

In modern distributed systems, it is very important to have convenient and informative tools for monitoring the state of the system, both providing the necessary data in real time and after a while. The lack of such tools greatly complicates the search for bottlenecks, errors, and limits the overall understanding of what is happening in the system. In this paper, we consider one of these tools - distributed query tracing - for the ClickHouse database management system. This method allows you to understand the logic of query execution, including distributed queries and ones, that are not bound within ClickHouse. We have implemented and tested generic methods that make it possible to implement query tracing in any desired ClickHouse components. The Jaeger Tracing system is used for data aggregation and trace visualization. This work will demonstrate a trace based on a SELECT query, which will allow you to observe query logic with the highlight of query phases, as well as more granular actions, such as disk access operations.

Оглавление

1. Список ключевых слов

ClickHouse, DBMS, distributed tracing, Jaeger Tracing, OpenTracing API

2. Введение

ClickHouse [1] - это столбцовая система управления базами данных, применяемая для OLAP вычислений, т.е. для интерактивной аналитической обработки данных. Система ClickHouse нацелена на скорость обработки запросов (в том числе и распределенных) и подразумевает отказоустойчивую работу на кластерах из десятков и сотен машин.

Для таких систем задачи профилирования, обнаружения узких мест или некорректного/неожиданного поведения при выполнении распределенного запроса могут оказаться крайне нетривиальными - необходимо учитывать взаимодействие разных компонентов системы, расположенных на разных физических машинах кластера. Поэтому одним из важных условий, которые нужно учитывать при разработке такой системы, как ClickHouse, является наличие инструментов, которые позволили бы эффективно решать описанные выше проблемы.

Стандартным подходом для обнаружения некорректного поведения является использование дебаг-логов (debug logs), однако для распределенных систем, и особенно для систем с микросервисной архитектурой (microservice architecture), бывает нелегко понять судьбу выполнения конкретного запроса.

Для нахождения узких мест можно использовать тестирование производительности (performance testing), за которым разработчики ClickHouse активно следят - можно найти сравнения с другими системами как на сайте [2], так и в независимых источниках. Тем не менее, зачастую такое тестирование позволяет судить лишь о некоторой единой величине как общее время выполнения запроса, но не позволяет получить более детальное представление о запросе.

Существуют другой менее стандартный подход, который называется распределенной трассировкой запроса (distributed request tracing). Его идея заключается в том, чтобы связать отдельные части запроса специальными идентификаторами, обладающими иерархической семантикой. А затем post factum проагрегировать и визуализировать все части запроса в удобном представлении, позволяющем оценить, что происходило с запросом, на каких компонентах системы и в каких ее логических сущностях. Важно понимать, что распределенная трассировка - лишь метод работы с запросом, а не готовая система, работающая из коробки. Это значит, что кодовая база должна быть готова к трассировке запроса - иметь необходимые внутренние инструменты. В совокупности с тем, что описываемый метод имеет широкие возможности для расширения, мы получаем, что задача трассировки - это большая интегральная задача, которую нельзя один раз решить и больше никогда к ней не возвращаться, ведь система растет и меняется, а значит в коде появляется все больше логики, которую потенциально может быть полезно трассировать. Поэтому тема данной работы имеет в названии словосочетание “инфраструктура распределенной трассировки”, а не просто

“распределенная трассировка”. Таким образом, смысл работы именно в том, чтобы предоставить и опробовать генеричные методы, позволяющие легко внедрять трассировку по всей кодовой базе ClickHouse.

Более четко сформулируем цели и задачи данной работы, которые модифицировались и детализировались в течение выполнения работы.

1. Исследовать существующие подходы к распределенной трассировке, посмотреть на опыт других компаний, команд. Выбрать подход, подходящий для системы ClickHouse.
2. Реализовать механизм для передачи данных о распределенном запросе по сетевому протоколу (верхнеуровневый взгляд на трассировку запроса), поддержать трассировку для запросов, выполняющихся многопоточно.
3. Добавить возможность передавать в ClickHouse данные о запросах, трассировка которых началась вне, используя специальный HTTP заголовок.
4. Детализировать трассировку с помощью класса CurrentMetrics и добавления к спанам тэгов (tags) и логов (logs).
5. Научиться передавать собранные данные в систему Jaeger для дальнейшей агрегации и визуализации трейса распределенного запроса.
6. Детализировать трассировку, позволив отражать логические компоненты выполнения запроса, т.н. процессоры (реализации виртуального класса IProcessor).

3. Описание OpenTracing API, Jaeger и схожих подходов

Существует широко используемый программный интерфейс (далее - API) для распределенной трассировки, который называется OpenTracing API [3], было решено ориентироваться именно на него. Перед тем как приступить к описанию данного API, расскажем о существующих подходах.

Одной из первых появилась система Zipkin [4], разработанная в компании Twitter, которая предоставляет как собственное API для сбора трейса, так и систему, позволяющую с этими данными работать. Выбранное в этой работе API не является частью какой-то одной системы трассировки, однако существует его популярная реализация, которая называется Jaeger Tracing [5] - это система, изначально разрабатываемая в компании Uber, а затем вместе с OpenTracing API получившая поддержку организации Cloud Native Computing Foundation. Jaeger написан на языке Go [6] , а клиентская часть системы в настоящий момент поддержана на 6 языках программирования, что делает Jaeger удобным для интеграции во множество проектов на разных языках программирования. Часто наряду с OpenTracing рассматривается система OpenCensus, прародитель которой - система Census - возникла в компании Google. Стоит отметить, что в настоящий момент ведется работа по слиянию двух систем OpenTracing и OpenCensus в одну новую - OpenTelemetry [7]. Разработчики обещают поддерживать обратную совместимость в течение двух лет и ожидают, что пользователи за это время

переедут на единую OpenTelemetry. Однако сейчас система все еще находится в разработке, например в репозитории проекта на GitHub клиента для языка C++ еще не существует. Помимо того, что OpenTracing API довольно популярно, оно было выбрано в частности по следующим двум причинам: уже существовали известные примеры использования OpenTracing API технологиями, которые интегрированы с ClickHouse, а единое API позволило бы бесшовно выполнять трассировку запросов, выполняемых на стыке нескольких технологий. Вторая причина в том, что в GitHub-репозитории ClickHouse уже существовал довольно востребованный пользовательский запрос (issue) [8] на добавление данной функциональности.

Введем основные определения OpenTracing API.

1. Спан (span) - логическая сущность, соответствующая определенному компоненту (операции) запроса. Спан обладает следующими обязательными параметрами: время начала, время окончания операции, имя операции, информация о других спанах, породивших данный. Примерами спанов могут быть операции любой гранулярности - от одного чтения с диска, до операции обработки запроса на шарде (shard) таблицы.
2. Трейс (trace) - множество всех спанов, соответствующих одному запросу. Другими словами, трейс полностью определяет трассировку одного запроса.
3. Трейсер (tracer) - объект в коде, через который осуществляется создание новых спанов, а также сериализация и десериализация при передачи информации про трейс через сетевые протоколы.

4. Спан-контекст (Span context) - объект, который уникально определяет спан. Как правило, это пара идентификаторов текущего трейса и текущего спана. Каждый спан знает о порождающих его спанах, как раз храня их спан-контекст. Из этого также следует, что именно спан-контекст достаточно передавать по сети при выполнении распределенного запроса.

Эти определения составляют основу модели OpenTracing API. Существуют дополняющие эту модель детали, которые будут затронуты далее. Информация о распределенном запросе - набор спанов, которые можно выстроить в иерархическую структуру, снабдив ее информацией о времени исполнения каждого спана. Эта структура на самом деле является направленным ациклическим графом, а ее ребра соответствуют причинно-следственной связи между парой спанов. На такую структуру удобно смотреть, чтобы понимать, какие подзапросы выполнялись какую часть времени, как это изображено ниже.

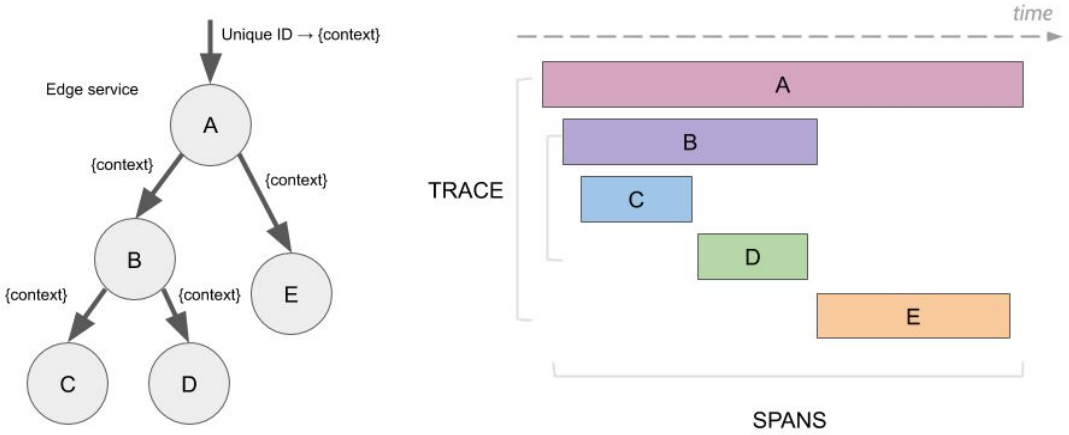

Рис. 3.1. Два представления трейса. Источник: https://www.jaegertracing.io/docs/1.18/architecture/

Для агрегации и визуализации трейсов используется система Jaeger, которая состоит из нескольких компонент.

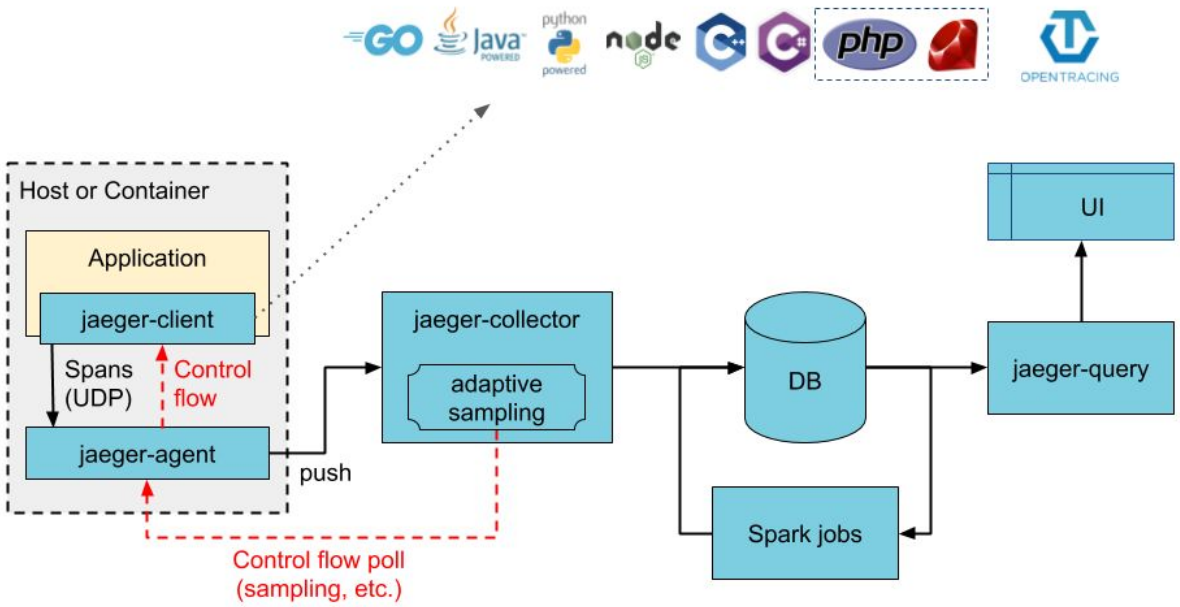

Рис. 3.2. Архитектура Jaeger Tracing. Источник: https://www.jaegertracing.io/docs/1.18/architecture/

1. Jaeger-client - это часть системы, которая реализует OpenTracing API на одном из множества языков программирования и напрямую интегрирована внутрь кодовой базы системы, запросы которой требуется трассировать.
2. Jaeger-agent - компонента, расположенная на каждой машине системы как отдельный небольшой сервис. Служит для сбора спанов с jaeger-client в батчи (batches) и отправки батчей в jaeger-collector.
3. Jaeger-collector - единая для системы точка сбора спанов со всех машин кластера, эти спаны затем отправляются в хранилище.

4. Jaeger-query - компонента, которая отображает в браузере интерфейс, через который можно получать информацию о трейсах из хранилища.

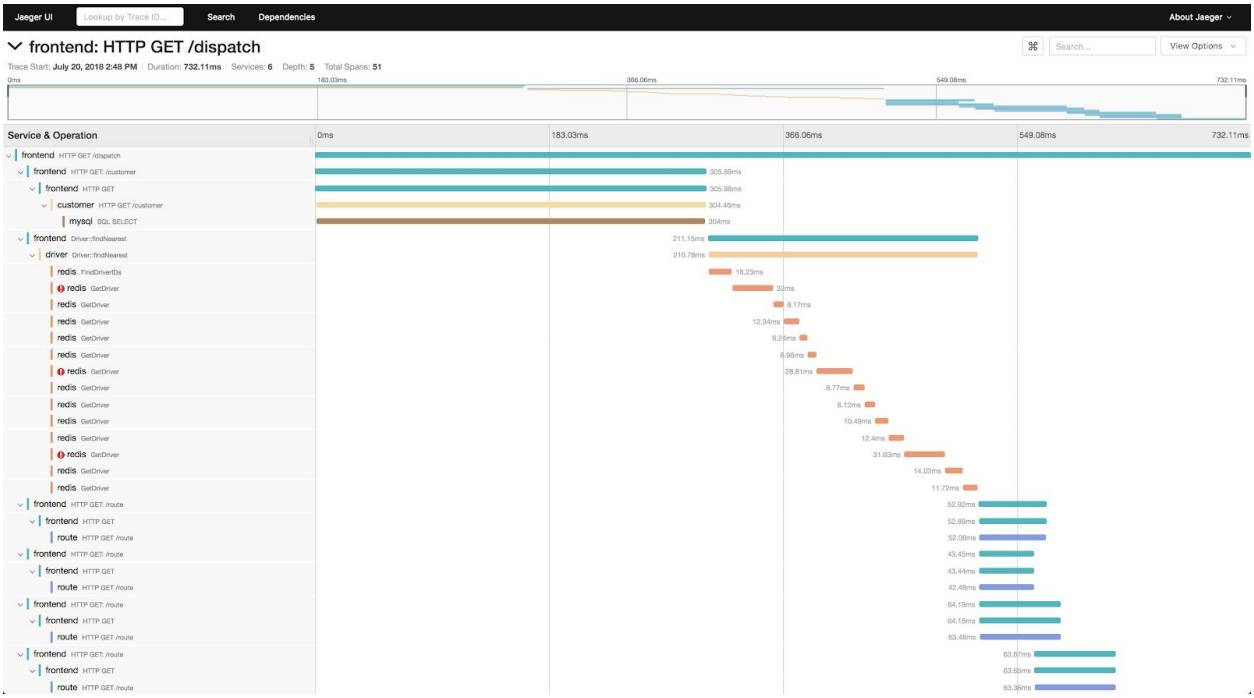

Рис. 3.3. Интерфейс Jaeger Query. Источник: https://www.jaegertracing.io/docs/1.18/

Примерно так выглядит основное окно работы с трейсом в Jaeger. Такой подробный отчет о выполнении запроса позволяет понять его логику, увидеть, какие подзапросы выполнялись слишком долго, возможно увидеть, что запрос выполнялся не так, как это ожидалось, например, происходили лишние походы по сети, или какие-то части запроса, которые должны были исполняться параллельно на самом деле исполнялись последовательно. Окончательную цель данной работы можно сформулировать так:

- Получить в интерфейсе Jaeger подобный трейс распределенного запроса к ClickHouse, который бы обладал

достаточной выразительностью, чтобы можно было увидеть выполнение различных логических компонент запроса.

4. Существующие в ClickHouse инструменты

Опишем некоторые инструменты для сбора статистик и метрик и для описания подзапросов, которые существовали в ClickHouse к началу выполнения данной работы. Некоторые из этих инструментов удалось удобным образом связать с распределенной трассировкой, о чем будет подробно рассказано в следующих главах.

4.1 QueryLog

Специальная таблица, которая позволяет хранить информацию о подзапросах. Здесь подзапрос - это часть запроса, которая исполняется на одной машине кластера в рамках, например, распределенного запроса. Эта таблица состоит из записей четырех типов: подзапрос начался или завершился и исключение до начала обработки запроса или во время обработки запроса. Каждая такая запись помещается в системную таблицу system.query_log. В ней довольно много полей: можно получить информацию о количестве обработанных строк и байт, исключениях, если такие возникали, разную системную информацию, а также о множестве счетчиков,

возникающих при профилировании запроса. Эти счетчики описаны в файле ProfileEvents, их довольно много разных - от счетчика времени на обработку запроса до счетчика на число вызовов операции lseek.

Чтобы использовать QueryLog, пользователь должен выставить в настройках флаг query_log перед выполнением самого запроса, а затем прочитать данные о запросе из вышеупомянутой системной таблицы. Этот подход может позволить получить полезную информацию о подзапросах. Разработчики могут ориентироваться на нее, чтобы искать нестыковки и ошибки в работе системы, а пользователи - чтобы понимать, какие подзапросы работали медленнее всего и иметь возможность соптимизировать свой запрос. Однако из-за обилия счетчиков может быть трудно найти что-то любопытное. Также информация в QueryLog может оказаться слишком общей, а отсутствие визуализации может затруднить интерпретируемость данных.

4.2. Другие системные логи (SystemLog)

QueryLog - это одна из реализаций интерфейса ISystemLog, смысл которой в том, чтобы складывать какие-то записи в системные таблицы. В случае QueryLog - это записи об исполнении подзапросов. Существуют также и другие системные логи: QueryThreadLog, TextLog, MetricLog, TraceLog, PartLog. Каждый из них позволяет фиксировать события определенного спектра, однако их рассмотрение выходит за рамки данной работы, поэтому ограничимся лишь их упоминанием. QueryLog же еще будет упомянут в следующих главах.

4.3. CurrentMetrics

Некоторый набор счетчиков, которые используются, чтобы регистрировать количество событий определенного типа, происходящих в настоящий момент. Примерами таких событий может служить количество файлов, открытых операционной системой на запись (OpenFileForRead), или число потоков, обрабатывающих запрос (QueryThread). Данный функционал реализован за счет использования в языке C++ того, что часто называют гардами (guards) - это специальные объекты, которые в простом случае влияют на некоторое наблюдаемое состояние дважды за время жизни - в конструкторе и в деструкторе. В случае CurrentMetrics в конструкторе объекта инкрементируется некоторый глобальный счетчик, а в деструкторе - декрементируется. Подобное решение получится использовать и в случае распределенной трассировки.

4.4. ProfileEvents

Эти счетчики уже упоминались в пункте 4.1., поэтому просто заметим, что отличие от CurrentMetrics в том, что ProfileEvents считают количество произошедших событий, а не количество происходящих в настоящий момент событий.

4.5. QueryProfiler

Профилирофщик, который с определенной частотой собирает текущий стек-трейс (stack trace), что затем можно использовать с утилитой flamegraph, чтобы узнать, какую часть времени в какой части кода проводила программа. Работает с использованием системного лога TraceLog.

4.6. Вывод

В системе ClickHouse существует довольно много специализированных инструментов, позволяющих получать ту или иную статистику о работе кластера. Такая информация может быть полезна как пользователям, так и разработчикам системы. Может возникнуть вопрос о целесообразности поддержки еще одного инструмента, поэтому заметим две вещи. Во-первых, информация, которую мы бы хотели получать при распределенной трассировке носит немного другой характер - хочется узнавать о логике исполнения запроса, не привязываясь к конкретным машинам кластера. А во-вторых, эта работа в частности является proof of concept для метода трассировки распределенных запросов в ClickHouse.

5. Наш вклад

Данная глава будет представлять собой описание того, что было сделано в рамках этой работы. При описании будем ориентироваться на

список целей и задач, который можно найти во Введении к данной работе. Тем более эта последовательность соответствует хронологическому порядку, в котором велась работа.

5.1. Передача информации про трейс по сетевому протоколу.

5.1.1. Реализация основных интерфейсов OpenTracing API.

Чтобы запустить простейший пример, не нужно реализовывать OpenTracing API полностью. Основные интерфейсы, которые требовалось реализовать сразу это:

- Структура SpanContext, которая служит идентификатором спана. Состоит из полей trace_id и span_id. Их значения обсудим позже.
- Класс Span, в котором есть поля
    - Название операции
    - Список спан-контекстов, которые соответствуют родительским спанам. В настоящий момент в API определены два вида причинно-следственных связей: ChildOf, FollowsFrom. Отличие в том, что для спанов, которые связаны отношением ChildOf, важно, что родительский спан дожидается завершения спана-потомка, а для FollowsFrom - нет. Примером, когда стоит использовать ChildOf служит, например, отправка запроса на чтение данных, которые требуются, чтобы завершить подсчет значения, вычисляемого в родительском спане. Примером FollowsFrom может служить отношение между producer и consumer в очереди сообщений.

    - Спан-контекст спана. Trace id получается из trace id предков, span id генерируется случайным образом.
    - Время начала жизни спана
    - Время конца жизни спана

  Также требовалась реализация следующего метода класса Span:

  - getSpanContext - позволяет получить спан-контекст спана, чтобы инициализировать новый спан.

- Класс DistributedTracer, обладающий следующими методами:
    - CreateSpan - создает новый объект класса Span. В качестве параметров требует название операции и список спан-контекстов (вместе с типом причинно-следственной связи) родительских спанов. Чаще всего этот список состоит лишь из единственного элемента. Если список пустой, то новый спан считается корневым в трейсе.
    - InjectSpanContext и ExtractSpanContext. Эта пара методов используется для передачи спан-контекста по сети. Подробнее расскажем позже.

5.1.2. Передача спан-контекста по сети.

В ClickHouse существует несколько способов передавать и принимать данные по сети. Основные - HTTPHandler и TCPHandler. Внутри кластера работа идет с использованием TCPHandler, клиент может общаться с кластером обоими способами. При исполнении запроса каждое такое

общение внутри кластера подкрепляется данными о запросе и клиенте, которые хранятся в специальном классе ClientInfo. Поэтому было решено вызывать новые методы InjectSpanContext и ExtractSpanContext при сериализации и десериализации ClientInfo соответственно, чтобы добавлять trace_id и span_id в данные. Это однако меняет протокол, поэтому необходимо проверять версию протокола, а также продвинуть его ревизию.

5.2. Работа с трейсами в многопоточном окружении.

Имея инструменты для создания спанов и передачи спан-контекста можно уже организовать работу с трейсами.

5.2.1. Создание спана на подзапрос.

У каждого запроса в ClickHouse есть контекст, который и было решено сделать владельцем спана, соответствующего всему подзапросу к машине. Создается контекст в уже упомянутых TCPHandler и HTTPHandler при получении запроса. Для доступа к контексту используется специальный класс ThreadStatus и thread local переменная этого типа. Когда поток начинает работать с новым запросом, часть полей этой переменной, которая зависит от текущего исполнения, инициализируются заново. В частности инициализируется поле query_context, которое в том числе содержит и текущий для данного потока спан. Итак, данный спан живет, пока в этой thread local переменной живет текущий query_context, то есть пока исполняется запрос - этого и хотелось достичь. Заметим, что TCPHandler

одинаковым образом принимает как запросы от клиентов, так и подзапросы от других машин кластера, поэтому описанный подход позволил получать трейс, в котором гранулярность спанов была ограничена подзапросами к отдельным машинам.

5.2.2. Запрос, исполняемый в многопоточном окружении.

Дополним рассказ о работе с контекстом запроса деталями многопоточности. На самом деле контекст запроса един для всех потоков, исполняющих данный запрос. В ThreadStatus есть поле типа ThreadGroupStatus, общее для всех потоков исполнения запроса. У ThreadGroupStatus в частности есть поле query_context, которое на самом деле указывает на тот же самый объект, что и поле query_context в ThreadStatus. Это значит, что если мы захотим создать новый спан, то не сможем сделать этого в query_context, потому что в разных потоках могут выполняться различные логические компоненты запроса, каждую из которых может хотеться при трассировке представлять как отдельный спан.

Поэтому только корневой спан хранится в query_context, а любой новый - в thread local переменной ThreadStatus. Функция, позволяющая получить текущий спан, сначала проверяет, существует ли спан в thread local и возвращает его, если он существует, а иначе возвращает спан из query_context. Можно задать вопрос: для чего же тогда вообще использовать переменную query_context? Дело в том, что, потоки могут добавляться к исполнению какого-то запроса в совершенно разных местах в коде. Этот

процесс контролируется за счет ThreadGroupStatus, через которую любой поток может узнать о всех потоках исполняющих данный запрос, а также добавиться к исполнению запроса. Иногда, например если поток был добавлен к запросу, чтобы выполнить определенную рутину, логичным выглядит создание нового спана, соответствующего работе данного потока, и тут нет больших проблем. А иногда, если это поток общего назначения (с точки зрения исполнения запроса), то создавать новый спан не хочется, тем не менее нужно позволить новому потоку узнавать о родительском спане, который, вообще говоря, живет в другом потоке. Если где-то забыть это сделать или сделать некорректно, то можно потерять все спаны, которые были созданы внутри этого потока, т.к. у него не будет доступа до trace id. С предложенным в работе методом потоки автоматически могут узнать о корневом спане, поэтому спаны никогда не потеряются. Однако, конечно, можно осуществить аккуратный рефакторинг кода и при добавлении к запросу нового потока добавлять информацию про спан-контекст везде и явным образом. Это избавит от необходимости держать корневой спан в общем query_context.

5.2.3. Получение трейса

В качестве способа, который позволил бы автоматизировать получение информации про трейс и при этом пока не добавлять Jaeger, был выбран уже упоминавшийся системный лог QueryLog. Достаточно было переиспользовать код, который обычно добавляет записи в QueryLog. После этого появилась возможность смотреть на спаны как на строки системной

таблицы, что помогло при дальнейшей отладке. Этот способ хранить трейсы до сих пор остался в коде, т.к. его можно использовать при написании тестов.

5.3. Поддержка трейса, начавшегося вне ClickHouse.

Одно из преимуществ общепринятого OpenTracing API для распределенной трассировки заключается в том, что это позволяет стереть границы между сервисами. Если все сервисы-участники запроса поддерживают OpenTracing API, то в едином трейсе можно наблюдать за всей логикой исполнения этого запроса. Для этого со стороны ClickHouse нужно поддержать получение спан-контекста извне.

Перед этим имеет смысл решить, какой формат будет у представления спан-контекста в ClickHouse. Удобно и правильно в таком случае ориентироваться на формат, используемый в Jaeger, а именно 128-битное число на trace id (или два 64-битных числа), 64-битное число на span id.

Спан-контекст извне можно получать с использованием специального заголовка HTTP запроса, которых есть несколько видов. Был выбран заголовок Traceparent [9]. Важно, что формат Traceparent заголовка соответствует внутреннему представлению спан-контекста, выбранному в ClickHouse.

5.4. Детализация трассировки с использованием CurrentMetrics и добавлением тэгов к спанам.

5.4.1. Переиспользование механизма CurrentMetrics

Описанное к настоящему решение позволяет трассировать лишь целые подзапросы. Используя возможность хранить спаны в переменной типа ThreadStatus, можно уменьшить гранулярность спанов.

Вместо того, чтобы добавлять создание спанов в каждый интересный файл кодовый базы, можно воспользоваться готовыми решениями, которые уже были добавлены в нужные и интересные места кода. Таким решением явилось CurrentMetrics, про которое уже говорилось в другой главе. Напомним только, что большинство счетчиков, поддерживаемых CurrentMetrics работают через создание гардов (guards). Как потенциально полезные для трассировки, были выбраны счетчики числа чтений с диска и числа записей на диск. В коде во всех местах, где выполняется работа с диском, уже есть гарды, которые инкрементируют этот счетчик, если в текущем потоке выполняется чтение с диска, а когда чтение завершено, то гард разрушается, декрементируя счетчик.

Чтобы переиспользовать эту механику, в код были добавлены гарды, которые создают новый спан, объявляют его предком текущего, переключают текущий спан в ThreadStatus на новый спан-предок, а внутри себя хранят родительский спан. Смысл в том, чтобы в thread local переменной всегда хранился текущий спан, в котором идет работа (в данном потоке). Это

нужно, т.к. хочется иметь возможность в любом другом месте кода создать новый спан-предок или добавить к текущему тэги и логи, о которых речь также пойдет в этой главе. Спан завершается при вызове деструктора у соответствующего гарда, и в этом деструкторе родительский спан возвращается на свое место в thread local переменную - состояние дерева спанов возвращается к такому, которое было до создания последнего спана. Таким образом, на гардах по сути удается построить стек спанов. В этой схеме важно соблюдать то правило, что если один гард был создан позже другого, то должен быть разрушен раньше другого.

Этот новый гард для спанов был добавлен как поле класса счетчиков, считающих CurrentMetrics. И каждый счетчик мог в конструкторе проинициализировать гард и таким образом создать новый спан. Для счетчиков работы с диском это означает, что мы получили спаны, время жизни которых соответствует времени одной сессии работы с диском.

К сожалению, есть одна особенность, которая не позволяет использовать эту технику на всех CurrentMetrics. Описанный подход подразумевает, что объект самого счетчика имеет поведение обычного гарда, однако в коде это не всегда так. Например, счетчик числа файлов, открытых на запись является полем класса WriteBufferFromFile. В первом приближении работу с объектами подобных классов в ClickHouse можно описать следующим образом: создается множество таких объектов (например, в соответствии с числом колонок, которые хранятся как независимые файлы), с которыми затем можно работать как с потоками, переключаясь с записи в

один файл на запись в другой файл и обратно. Таким образом, ни о какой одной thread local переменной речи быть не может. Подобную логику исполнения уже не так просто трассировать в автоматическом режиме, не обращаясь к конкретному коду и конкретному классу, который в нем используется. При этом в общем случае, конечно, хочется избегать явных зависимостей от кода реализующего трассировку, потому что любой рефакторинг, любое добавление новой функциональности потребует учитывать трассировку, а если про нее забыть, то появится код, который не будет полноценно трассироваться. Тем не менее, в этой работе еще будет рассмотрено, в каком случае точно следует добавить трассировку в исполняющий запрос код.

5.4.2. Добавление к спанам тэгов и логов.

В OpenTracing API существуют два довольно важных понятия, которые пока не были освещены в данной работе. Это тэги (tags) и логи (logs). Они являются дополнительной информацией, которой можно обогатить спан. Затем при рассмотрении трейса в Jaeger, можно будет увидеть все тэги и логи спанов. По структуре и тэг, и лог представляют собой пару ключ-значение строкового типа (если более точно, то значение тэга может быть числом или булевой переменной), однако логи также имеют временную метку и на самом деле они сгруппированы по этим временным меткам. Смысл тэгов в том, чтобы обогатить спан какой-то дополнительной информацией о запросе, например адресом хоста или значением какого-то внутреннего состояния.

Использование логов призвано отражать важные события, которые произошли во время жизни спана.

В коде, чтобы добавить тэги и логи пользователю нужно вызвать функцию и передать туда ключ и значение. Внутри функции в thread local переменной находится текущий спан, которому будут приписаны эти тэги и логи. Затем, после того, как спан завершился, с точки зрения семантики уже никакие тэги и логи ему приписываться не должны. Спан готов к отправке в Jaeger.

5.5. Подключение Jaeger

5.5.1. Как Jaeger собирает данные

В Jaeger есть два сервиса, которые в каком-то виде могут принимать спаны. Это jaeger-agent и jaeger-collector. Jaeger-agent работает на каждой машине кластера, принимает спаны от jaeger-client, собирает их в батчи (batches) и отправляет в jaeger-collector, который уже складывает спаны в хранилище.

Также есть способ общаться непосредственно с jaeger-collector из клиента. Это может быть полезно, если вариант с jaeger-agent по каким-то причинам не подходит или не удобен. В данной работе был выбран именно такой вариант.

В разных сценариях используются разные способы совершать удаленный вызов процедур (RPC) и разные протоколы:

- Клиент общается с агентом, используя Apache Thrift поверх UDP.
- Агент общается с коллектором, используя gRPC с protobuf.
- Клиент общается с коллектором, используя Apache Thrift поверх HTTP.

Чтобы использовать Apache Thrift, необходимо сгенерировать код, описывающий протокол общения. Для этого в репозитории Jaeger на GitHub есть файл с расширением .thrift - это расширение для файлов на языке описания интерфейсов (IDL) для Apache Thrift.

Чтобы получать результат трейса, используется jaeger-query. Предполагается работа с этим сервисом через браузер.

Есть два способа запустить все сервисы Jaeger - каждый сервис отдельно или используя единый бинарный файл jaeger-all-in-one. Для целей данной работы второго способа достаточно.

5.5.2. Подключение Jaeger

Был написан код, который при завершении спана упаковывает его в формат, описанный в .thrift файле, а затем создает HTTP-сессию и посылает спан в jaeger-collector. Конечно, такое поведение недопустимо в production коде, однако подходит для целей демонстрации работоспособности. Демонстрация результатов будет представлена в следующей главе.

5.6. Детализация трассировки с использованием Processors

5.6.1. Обработка запроса

В ClickHouse перед тем, как перейти к непосредственному процессингу запроса, происходит анализ abstract syntax tree (AST), выделяются компоненты запроса, строится граф зависимостей. Сфокусируемся на запросе вида SELECT (демонстрация работы также будет с некоторым распределенным SELECT-запросом). Выделение компонентов запроса из AST происходит в том, что называется интерпретаторами, в случае с SELECT это файл InterpreterSelectQuery. Работа с данными и процессинг запроса происходят в файле PipelineExecutor (которому уже не важен конкретный тип запроса). В нем все компоненты запроса представлены в виде так называемых процессоров. ClickHouse - большая система с большой функциональностью и богатым SQL-подобным языком запросов, поэтому различных процессоров тоже большое множество. В этом можно убедиться, прочитав вышеупомянутый файл InterpreterSelectQuery.

Получается, что логику исполнения запроса в ClickHouse можно выразить в терминах работы с процессорами - например, что сначала работает процессор, который посылает распределенный запрос, в котором другой процессор вычитывает данные с диска, следующий - обрабатывает согласно ORDER BY и так далее. Поэтому появилось желание научиться трассировать эти процессоры, представляя их работу в качестве отдельных спанов.

5.6.2. Processors в ClickHouse

Процессоры - довольно новый интерфейс в ClickHouse, который появился в рамках большого рефакторинга. Работа с процессорами напоминает работу с fiber-aware кодом, но процессоры обладают специфичным для ClickHouse интерфейсом. Файл PipelineExecutor как раз является точкой, в которой происходит шедулинг процессоров - множество исполняющих потоков достает из очереди процессоры, готовые к обработке (например, процессор мог стать готовым, когда другой процессор доставил нужные данные), и один поток начинает исполнять этот процессор. Зависимости процессоров друг от друга описываются через некоторый граф.

5.6.3. Трассировка процессоров

Нетрудным образом, используя гарды, удалось трассировать методы интерпретатора SELECT запроса, в котором происходит подготовка пайплайна.

Важные для трассировки детали PipelineExecutor заключаются в том, что во-первых в этом файле происходит работа с состояниями процессора, а во-вторых, что как и для файберов, выполнение различных этапов работы процессора может исполняться в различных потоках. Важные для нас состояния это Idle, которое означает, что процессор ждет данные, чтобы начать работать, и Finished, когда процессор завершает работу и больше нет смысла его вызывать.

Чтобы выразить работу процессора через спан, каждому процессору как поле класса был добавлен спан. Однако, инициализировать этот спан нужно не при инициализации процессора, а когда процессор первый раз выходит из состояния Idle, а завершать спан - когда процессор переходит в состояние Finished. Если бы время жизни спана соответствовало времени жизни процессора, то для них всех спаны в Jaeger выглядели одинаково, т.к. Создаются они в интерпретаторе, а завершаются при завершении работы в PipelineExecutor.

Заметим, что в PipelineExecutor также есть работа, которая осуществляется не в процессорах - это работа потоков по подготовке и поиску готовых процессоров. Такую работу тоже хочется трассировать, поэтому для этих потоков (они привязаны к исполнению запроса только на время работы PipelineExecutor) также создаются спаны. Осталось понять, каким образом осуществить переключение спана, потому что между спанами потоков и спанами процессоров нет ChildOf зависимости. Для этого в код был добавлен еще один гард, который паркует текущий спан, а на его место в thread local помещает другой спан, переданный ему в конструкторе. В деструкторе этот гард возвращает припаркованный спан на место. Такой гард в PipelineExecutor создается от спана процессора, паркует спан потока, а в конце возвращает все на место.

Внутри выполнения процессоров можно теперь создавать спаны, соответствующие конкретным методом процессоров. Это такие методы, как prepare, work, expandPipeline. У всех вызовов методов одного процессора

будет один родительский спан, поэтому можно будет удобным образом наблюдать это в Jaeger.

6. Демонстрация работы трассировки

В этой главе будет показан наглядный результат работы трассировки. К сожалению, не очень просто показывать по изображениям работу такого интерактивного инструмента, как Jaeger, но мы постараемся обратить внимание на самые интересные места, которые подробно были описаны в предыдущей главе.

В качестве примера рассматривается SELECT запрос к заранее подготовленной таблице:

```
SELECT DISTINCT s FROM remote('127.0.0.{1,2}', currentDatabase(), data) ORDER BY x + y, s LIMIT 10;
```

remote отвечает за распределенной запрос, в данном случае будет выполнено два SELECT запроса на двух шардах таблицы. Также отработают процессоры, соответствующие функциям DISTINCT, ORDER BY, LIMIT.

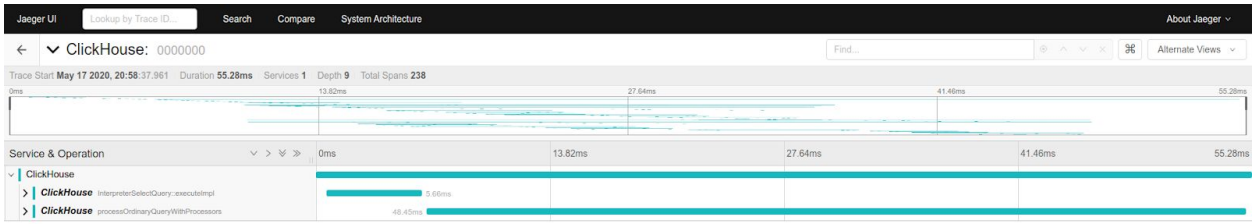

Рис. 6.1. Демонстрация работы трассировки

На рисунке 6.1. видно интерфейс Jaeger с нашим запросом. Выполнялся запрос 55 миллисекунд, всего в трейсе 238 спанов. На изображении видим верхний спан - спан всего запроса. Спан слева - это спан интерпретатора, который готовит запрос. Спан справа - спан исполнения запроса.

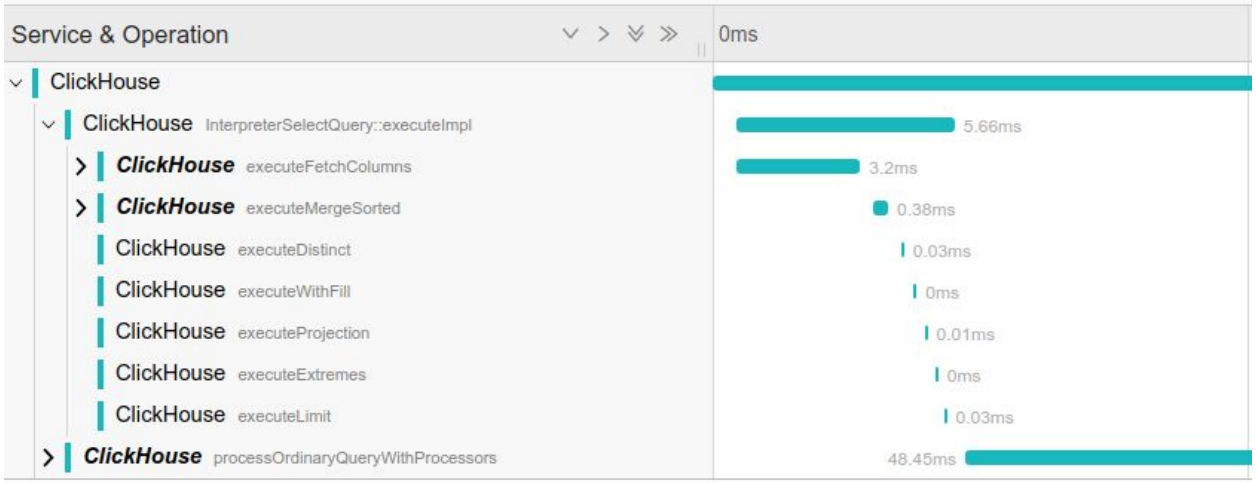

Рис. 6.2. Демонстрация работы трассировки

На рисунке 6.2. мы раскрыли спан интерпретатора и видим, что дольше всего выполнялась команда FetchColumns.

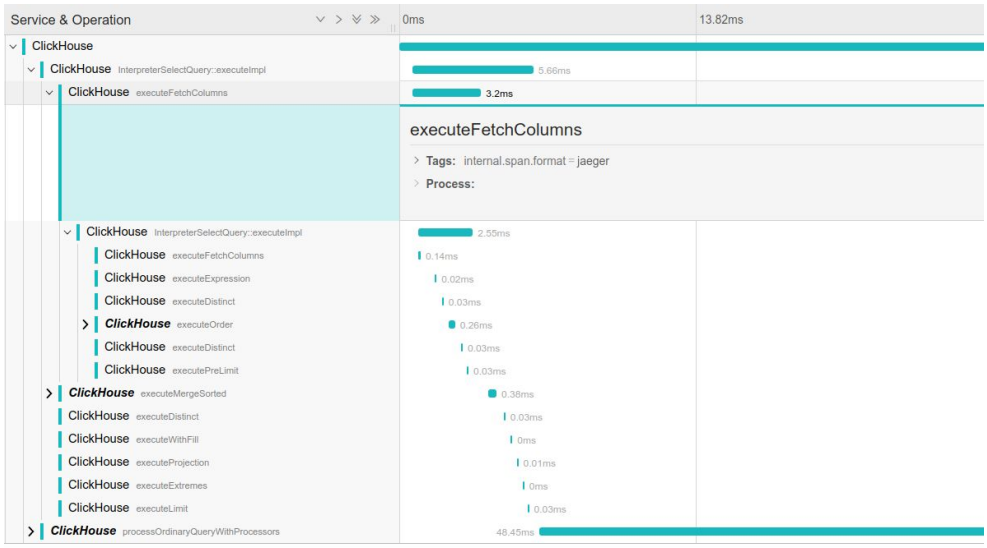

Рис. 6.3. Демонстрация работы трассировки

На каждый спан можно нажать и посмотреть его тэги. Так, на рисунке 6.3. видим тэг format=jaeger.

Теперь посмотрим на выполнение запроса

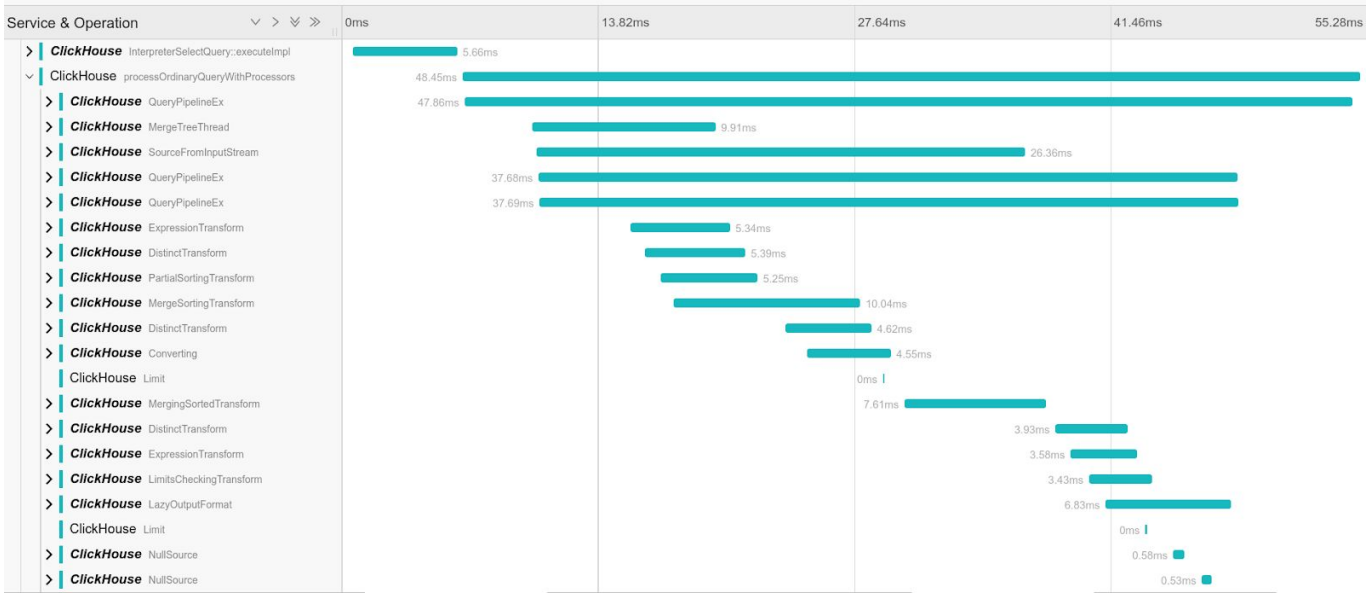

Рис. 6.4. Демонстрация работы трассировки

Серые надписи слева - это operation_name спана. Здесь видны спаны двух видов. Во-первых три спана QueryPipelineEx - это спаны, соответствующие трем потокам, в которых исполнялся запрос. Все остальные спаны, большинство из которых имеют Transform в названии - это и есть реализации интерфейса процессоров.

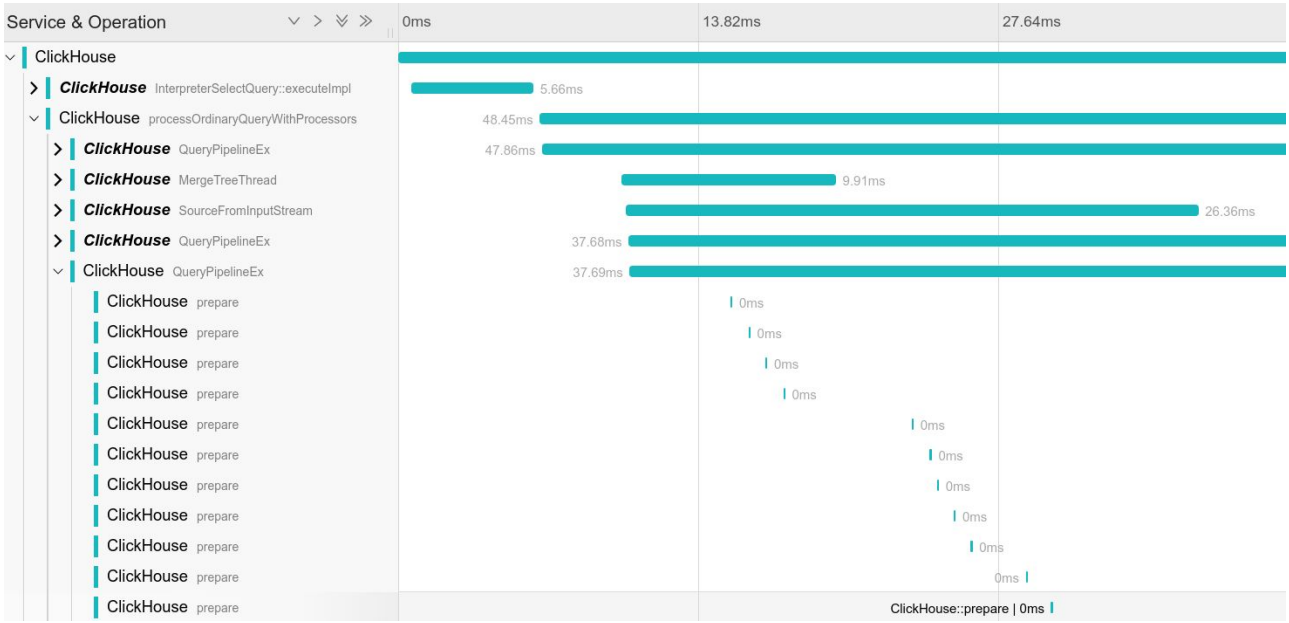

Рис. 6.5. Демонстрация работы трассировки

На рисунке 6.5. выше видно, что в потоке QueryPipelineEx есть спаны prepare - это как раз спаны, которые обозначают то, что мы подготавливали процессор, который был в состоянии Idle.

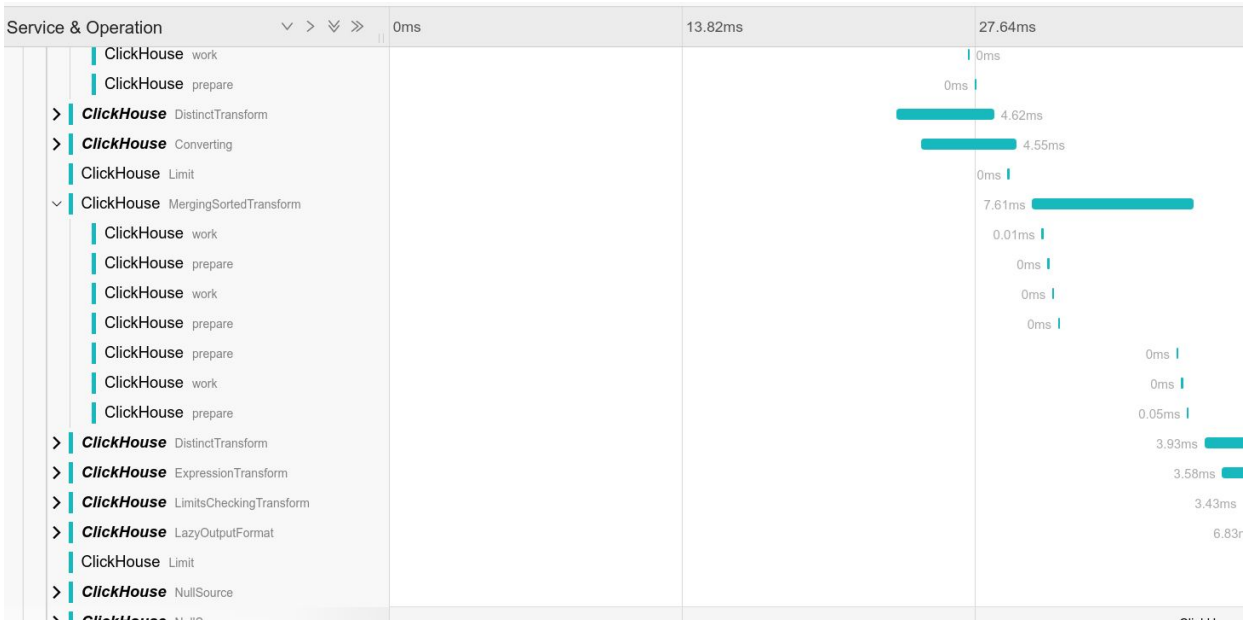

Рис. 6.6. Демонстрация работы трассировки

На рисунке 6.6 мы раскрыли спан MergingSortedTransform. Здесь можно увидить, что мы трижды исполняли данный процесс (метод work).

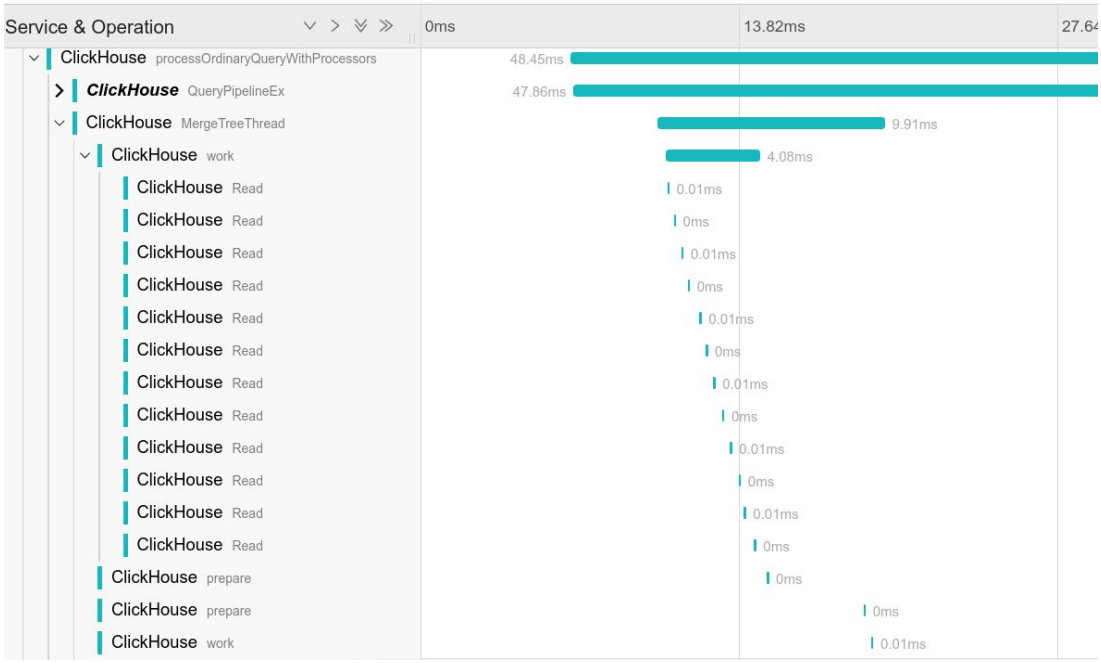

Рис. 6.7. Демонстрация работы трассировки

Раскрыли спан MergeTreeThread, который работает с таблицей типа MergeTree. Он читает из нее данные - тут мы видим обращения к диску, т.е. Read спаны (реализованные через CurrentMetrics). Обращений так много, потому что ClickHouse - колоночная СУБД, различные колонки хранятся независимо.

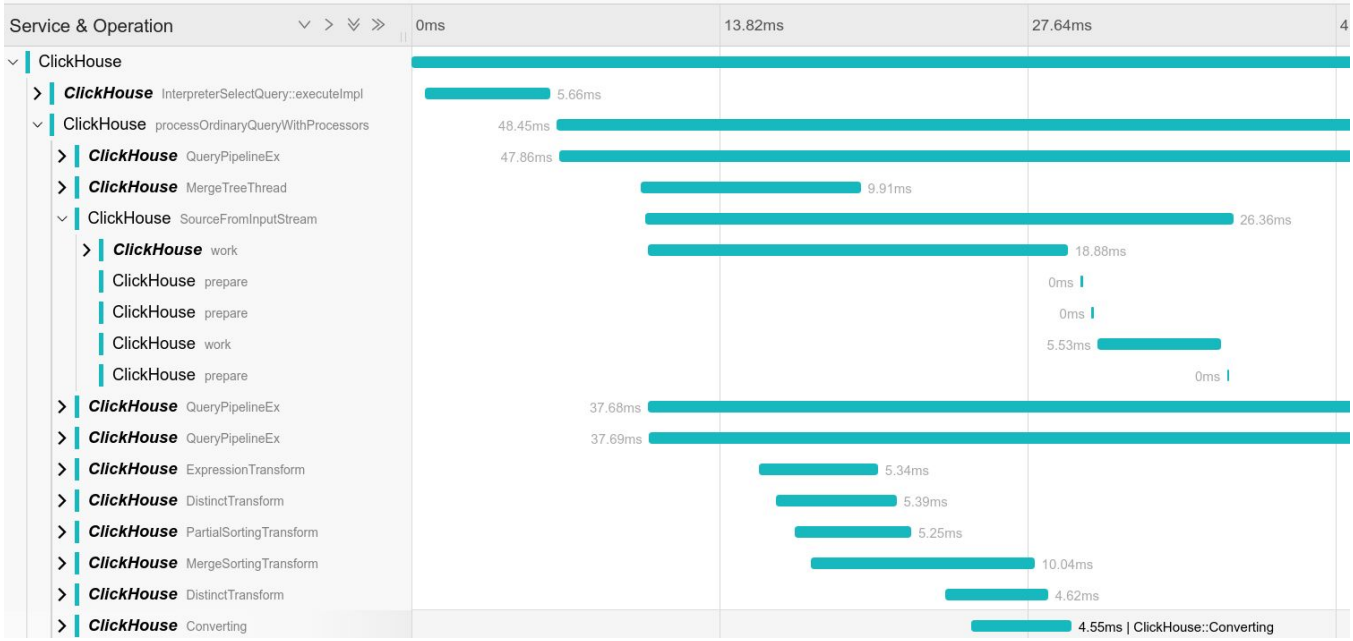

Рис. 6.8. Демонстрация работы трассировки

До сих пор мы не видели распределенный запрос, а видели лишь запрос к одному из шардов, расположенных локально по отношению к точке инициации запроса. На рисунке 6.8. в спане SourceFromInputStream в первом work как раз осуществляется этот распределенный запрос. Посмотрим на это подробнее.

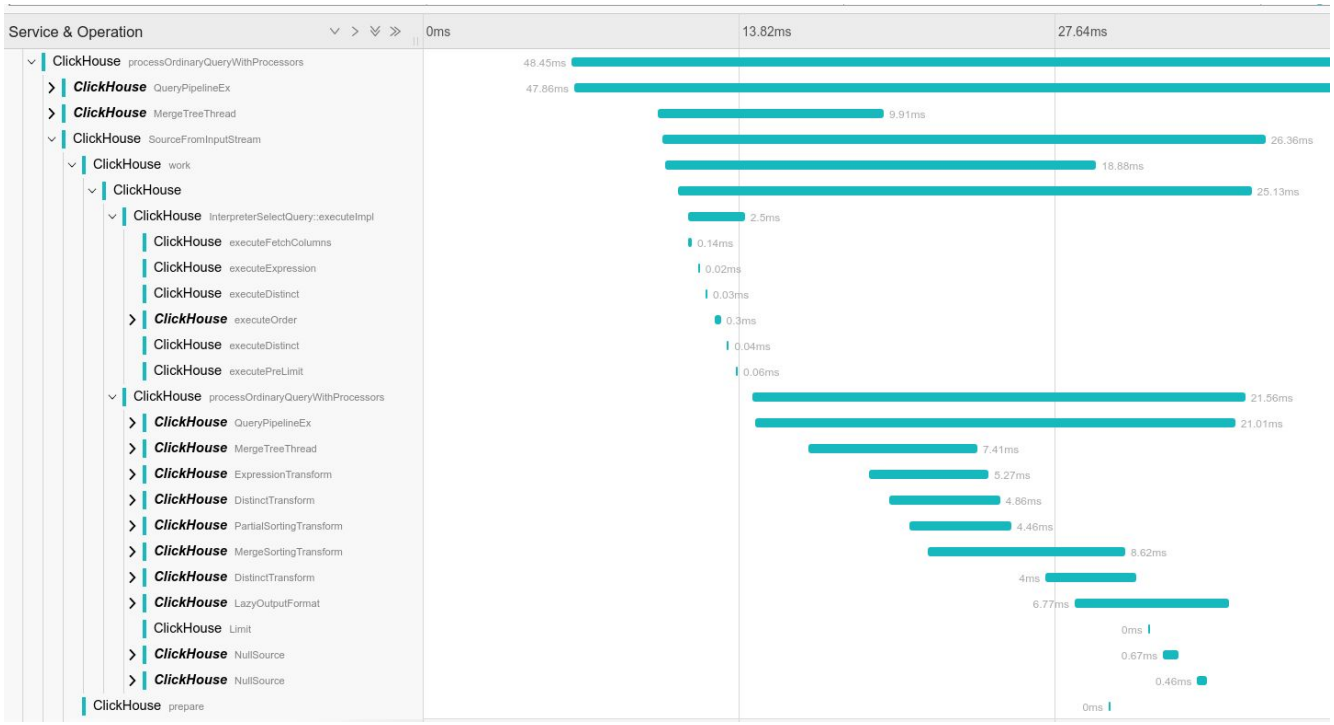

Рис. 6.9. Демонстрация работы трассировки

На этом изображении мы показали запрос к удаленному шарду, который требовал передачи данных (в том числе спан-контекста) по сети. По сути мы видим те же две фазы работы interpreter-a и executor-a, что логично, потому что запросы к шардам были одинаковыми.

Рис. 6.10. Демонстрация работы трассировки

Завершается запрос слиянием ответов с двух шардов и выводом результата.

7. Заключение

В ClickHouse была добавлена возможность распределенной трассировки и на примере распределенного SELECT запроса продемонстрирована состоятельность выбранного подхода, т.к. довольно генеричным образом удалось добавить трассировку на уровне подзапросов к шардам таблицы, на уровне CurrentMetrics, например таких, как чтение и запись на диск, а также на уровне логического исполнения запроса с использованием так называемых процессоров. Построенный способ распределенной трассировки легко расширяем под нужны разработчиков и пользователей: чаще всего во все интересные места исполнения можно добавлять спаны, используя технику гардов, а спаны обогащать специфичной информацией благодаря тэгам и логам. Таким образом, распределенная трассировка обладает широкими возможностями для интроспекции запроса и хорошей интерпретируемостью, которая не зависит от того, насколько много компонентов кластера поучаствовало в запросе. Некоторую работу по приведению написанного кода к production виду еще предстоит сделать, но все точки роста мы уже обсудили с командой разработчиков ClickHouse.

8. Список литературы

[1] ClickHouse, “ClickHouse website”, 2020. [Online]. Available: https://clickhouse.tech/

[2] ClickHouse, “Performance comparison of analytical DBMS”, 2020. [Online]. Available: https://clickhouse.tech/benchmark/dbms/

[3] OpenTracing team, “OpenTracing API website”, 2020. [Online]. Available: https://opentracing.io/

[4] Zipkin, “Zipkin website”, 2020. [Online]. Available: https://zipkin.io/

[5] Jaeger Tracing, “Jaeger Tracing website”, 2020. [Online]. Available: https://www.jaegertracing.io/

[6] Jaeger Tracing, “Jaeger GitHub repository”, 2020. [Online]. Available: https://github.com/jaegertracing/jaeger

[7] OpenTelemetry team, "Merging OpenTracing and OpenCensus”, 2020. [Online]. Available: https://medium.com/opentracing/merging-opentracing-and-opencensus-f0fe9c7ca6f0

[8] ClickHouse community, “support of open tracing in clickhouse GitHub issue”, 2020. [Online]. Available: https://github.com/ClickHouse/ClickHouse/issues/5182

[9] W3C, “World Wide Web Consortium”, 2020. [Online]. Available: https://www.w3.org/TR/trace-context/#traceparent-header