МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра систем информатики

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Компьютерные науки и системотехника

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Бухнера Марка Евгеньевича

Тема работы:

ИССЛЕДОВАНИЕ И ВНЕДРЕНИЕ ФИЛЬТРОВ ДЛЯ УСКОРЕНИЯ ЗАПРОСОВ К РАСПРЕДЕЛЕННОЙ БАЗЕ ТРАНЗАКЦИОННЫХ ДАННЫХ

«К защите допущена»
Заведующий кафедрой,
д.ф.-м.н., профессор
Лаврентьев М. М. / _____
«___» _____ 2024 г.

Руководитель ВКР
PhD, доцент
каф. ТК ММФ НГУ
ван Беверн Р. А. / _____
«___» _____ 2024 г.

Новосибирск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий

Кафедра систем информатики
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Компьютерные науки и системотехника

УТВЕРЖДАЮ
Зав. кафедрой Лаврентьев М. М.

_____
«23» октября 2023 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Бухнеру Марку Евгеньевичу, группы 20214

Тема: «Исследование и внедрение фильтров для ускорения запросов к распределенной базе транзакционных данных»
утверждена распоряжение проректора по учебной работе от 23 октября 2023 г. № 0377

Срок сдачи студентом готовой работы 31 мая 2024 г.

Исходные данные (или цель работы): ускорить выполнение запросов к распределенной базе данных Apache Hudi внедрением фильтров (структур данных), внедрить один или несколько фильтров в базу данных Apache Hudi.

Структурные части работы: исследование структур данных, ускоряющих запросы на диапазоне данных (запросы с условиями); внедрение избранных структур данных в базу данных Apache Hudi; проведение замеров производительности с использованием SQL-запросов бенчмарка TPC-H; сравнение данных с исходными.

Консультант по разделам ВКР: Цидулко Оксана Юрьевна — все разделы.

Руководитель ВКР
PhD, доцент
каф. ТК ММФ НГУ
ван Беверн Р. А. / _____
«23» октября 2023 г.

Задание принял к исполнению
Бухнер М. Е. / _____
«23» октября 2023 г.

АННОТАЦИЯ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Наименование темы: Исследование и внедрение фильтров для ускорения запросов к распределенной базе транзакционных данных

Выполнена студентом Бухнером Марком Евгеньевичем
Факультет информационных технологий, Новосибирский государственный университет
Кафедра систем информатики
Группа 20214
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Компьютерные науки и системотехника

Объем работы: 38 страниц
Количество иллюстраций: 4
Количество таблиц: 0
Количество литературных источников: 26
Количество приложений: 0
Ключевые слова: ключевые слова, через, запятую

В дипломной работе рассмотрены основные аспекты область исследования, проведен анализ существующих подходов к задача исследования. В работе представлен новый метод метод исследования, который позволяет достичь цель исследования.

Был разработан программный комплекс название программного комплекса, реализующий предложенный метод, и проведены экспериментальные исследования результаты экспериментальных исследований. Проведен анализ полученных результатов и сделаны выводы о выводы исследования.

Бухнер Марк Евгеньевич / _____
«___» _____ 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Платформы для анализа больших данных состоят из множества сервисов, среди них ссылки? выделяются системы для непосредственной обработки данных, системы для доступа к данным и сами хранилища данных.

Хранилища данных в таких системах являются внешними удаленными распределенными сервисами, что превращает ввод-вывод такого хранилища в одно из основных узких мест при обработке запросов. Существует множество подходов к организации файлов для их эффективного чтения, что позволяет сократить время оптимизировать? обработки запроса. Тем не менее, ускорение выполнения запросов возможно и другими путями, один из них — отсеивание файлов с данными, которые не удовлетворяют предикату запроса.

Существующие методы отсеивания файлов с данными используют упрощённые сводки (например, минимальные и максимальные значения каждого атрибута) для всех файлов данных, чтобы фильтровать те файлы, в которых не содержится записей, который удовлетворяют предикату запроса. Однако при работе с реальными данными такие подходы не всегда оказываются эффективными ввиду того, что каждый файл с данными может содержать большой диапазон значений, например — отметки времени только за январь и только за декабрь определенного года. Тогда для запросов, нацеленных на извлечение данных за определенные периоды, например, с июня по август того же года, такой подход приведет к необходимости чтения файлов, которые не содержат нужной информации.

Целью данной работы является разработка оптимальных методов индексации данных для ускорения обработки запросов с условиями в контексте платформ больших данных на основе анализа существующих подходов и их применимости на практике.

Задача исследования состоит в разработке структуры данных, которая позволит ускорить обработку интервальных запросов в платформах для обработки больших данных, а также во внедрении этой структуры данных в

одну из платформ с последующим проведением экспериментов на реальных наборах данных.

1 ПРЕДМЕТНАЯ ОБЛАСТЬ И ТЕКУЩИЕ РЕШЕНИЯ

1.1 Предметная область

В научной литературе существует множество определений термина **большие данные** [1, 2, 3]. Однако в рамках данной работы целесообразно выделить общие характеристики из всех предложенных определений.

Распространённой ошибкой при определении понятия большие данные является попытка придать данным численную оценку объёма, потому что объём данных, классифицируемых как большие, не только постоянно увеличивается, но и зависит от возрастающих вычислительных мощностей систем, на которых эти данные обрабатываются и хранятся.

Несмотря на необходимость дать численные оценки для данных в некоторых частях данной работы, это не противоречит основному свойству больших данных — их нельзя эффективно обработать или хранить на одном вычислительном узле. Следовательно, обработка таких объёмов данных требует использования множества вычислительных узлов, которые работают над одной задачей и видны пользователям как одна цельная система, такие системы называются **распределенными** [4].

Распределенные системы для обработки больших данных состоят из множества сервисов, однако их можно разделить на три уровня: [5]

1. **Системы обработки данных** — распределенная вычислительная система (кластер), которая отвечает на запросы пользователей.
2. **Система доступа к данным** — управляет организацией данных для более быстрого доступа к ним.
3. **Хранилище данных** — внешний распределенный сервис, доступ к которому осуществляется по сети.

Системы с такой организацией будем называть **платформами для обработки больших данных**.

Именно системы для доступа к данным организуют структуру файлов в хранилище данных для более быстрого доступа к ним. Например, одной из систем для хранения и доступа к данным является Apache Hudi — транзак-

ционная система для доступа к данным, которая поддерживает исторические запросы, а также отвечает за консистентность данных при записи и чтении. В первом приближении такую систему можно охарактеризовать как формат хранения данных. В качестве хранилища она использует файловую систему Hadoop Distributed File System (HDFS), являющуюся частью проекта Apache Hadoop. откуда информация?

Для систем существует два способа увеличения общей производительности, а также увеличения предельного объема данных, который возможно сохранить в системе: [6]

1. **Горизонтальное масштабирование** — увеличение количества отдельных вычислительных узлов в системе, выполняющих одну и ту же функцию.
2. **Вертикальное масштабирование** — увеличение производительности каждого вычислительного узла системы, путем использования более производительных компонентов и более объемных устройств хранения данных.

В распределенных системах используют именно горизонтальное масштабирование ввиду природы распределенных систем, а также дешевизны такого масштабирования, так как распределенная система по определению содержит множество узлов, и добавление новых является для таких систем естественным процессом [6].

Выбор конкретной реализации хранилища данных не важен в контексте данной работы, так как все они обладают общими свойствами — это внешние сервисы, взаимодействие с которыми происходит по сети. Такие системы используются в платформах для обработки больших данных ввиду того, что такие системы хорошо масштабируются горизонтально. Это необходимо для надежного хранения данных и возможности дешевого добавления памяти в файловую систему [7].

Такая архитектура приводит к тому, что самое узкое место всей платформы при обработке данных — доступ к данным, так как скорость чтения данных по сети на порядки ниже скорости чтения данных с ло-

кального запоминающего устройства или оперативной памяти [8]. Существует множество способов организации данных для более быстрого их чтения например, колоночный формат хранения данных?, что позволяет сократить время обработки запроса. Тем не менее, ускорение выполнения запросов возможно и другими путями, один из них — отсеивание файлов с данными, которые не удовлетворяют предикату запроса. Действительно, если для выполнения запроса нам нужны только записи за период, например, с января по март каждого года, то чтение файлов, которые содержат данные только за месяцы, отличные от заданных, можно пропустить. Такой способ ускорения запросов действительно эффективен, так как мы уже выяснили, что чтение данных с внешнего файлового хранилища по сети является узким местом.

Для дальнейшего определения предметной области необходимо дать несколько определений:

Определение 1. Запрос с условием — запрос к системе доступа к данным с фильтром (фильтрами) в виде предикатов, которые позволяют определить, какие данные будут включены в результирующий набор.

Определение 2. Нерелевантный файл — файл с данными, который не содержит данных, удовлетворяющих предикату запроса, то есть файл, чтение которого можно пропустить для ускорения выполнения условного запроса.

Не умаляя общности в дальнейшем будем рассматривать запросы с условиями, в которых содержатся предикаты только по одному атрибуту. Тогда запросы с условиями можно разделить на три вида:

1. **Точечный запрос** — запрос записей с предикатом равенства одному определенному значению.
2. **Интервальный ограниченный запрос** — запрос записей на интервале, ограниченном с обеих сторон.
3. **Интервальный неограниченный запрос** — запрос записей на интервале, ограниченном только с одной стороны.

Таким образом, любые другие составные запросы являются комбинацией перечисленных выше запросов.

Существующие методы отсеивания файлов с данными используют **упрощённые сводки** — минимальные и максимальные значения каждого атрибута для каждого файла данных, чтобы фильтровать те файлы, в которых не содержится записей, который удовлетворяют предикату запроса [9, 10].

Теорема 1 (Об оптимальности упрощенных свободок)**.** *Фильтр на основе упрощенных сводок является оптимальным по используемой памяти и времени на фильтрацию одного файла с данными для интервальных неограниченных запросов.*

Доказательство.

Дан файл, содержащий набор данных $\{v_1, v_2, \ldots, v_n\}$ из некоторого интервала $[a, b]$. Также задан предикат $P(x)$, который может быть либо лучем вправо: $P(x) = (x \geq c)$, либо лучем влево: $P(x) = (x \leq c)$, где c — некоторая константа.

Для хранения минимального и максимального значений файла требуется всего два элемента в памяти, что составляет $O(1)$ памяти.

Проверка предиката $P(x)$ на диапазон $[v_{\min}, v_{\max}]$ может быть выполнена за время $O(1)$:

- Если $P(x) = (x \geq c)$, то предикат истинно для всех значений в файле, если $v_{\max} \geq c$, и ложно, если $v_{\min} < c$.
- Если $P(x) = (x \leq c)$, то предикат истинно для всех значений в файле, если $v_{\min} \leq c$, и ложно, если $v_{\max} > c$.

Теорема 1 доказана.

Однако такая структура не является оптимальной для точечных и интервальных ограниченных запросов. Например, каждый файл с данными может содержать удаленные друг от друга значения — отметки времени только за январь и только за декабрь определенного года, но при этом не содержать значений внутри этого диапазона. Тогда для точечных запросов,

например, за конкретный день июля, или для запросов, нацеленных на извлечение данных за ограниченный с двух сторон период, например, с июня по август того же года, такой подход приведет к необходимости чтения файлов, которые не содержат нужной информации.

Соответственно задача фильтрации нерелевантных файлов остается актуальной для точечных и интервальных ограниченных запросов. В дальнейшем в данной работе интервальные ограниченные запросы будем называть интервальными запросами.

Таким образом, предметной областью данной работы является индексация и фильтрация нерелевантных файлов с данными для их более быстрого чтения в распределенных базах данных при выполнении интервальных запросов.

1.2 Устройство системы доступа к данным

Для формирования требований к структуре данных, позволяющей ускорить выполнение интервальных запросов путем отсеивания файлов с данными, не удовлетворяющими предикату, необходимо определить ключевые моменты устройства систем для доступа к данным в платформах для обработки больших данных, чтобы сформировать ряд ограничений, которые существуют при реализации структур данных в таких системах. Системы для доступа к данным используют одинаковые стратегии управления данными, поэтому, не умаляя общности, рассмотрим в данной работе устройство одной из популярных таких систем — **Apache Hudi** [11].

Apache Hudi (далее — Hudi) организует данные в таблицы, реализуя тем самым реляционную модель [12]. Hudi поддерживает работу с историческими запросами: при обновлении создаётся новый снимок обновленных данных, в то время как предыдущий остаётся доступным для исторических запросов. Каждой операции, изменяющей состояние данных, присваивается уникальная отметка на временной шкале. Используя концепцию временной шкалы, Hudi реализует журналирование всех операций над данными что это такое? может гарсия молина, используя этот журнал, реализуются

различные стратегии управления параллельным доступом к данным. На каждый момент времени приходится не более одного действия, создавая линейный порядок среди всех операций над таблицей. Каждое действие в журнале содержит информацию о времени (с точностью до миллисекунд), типе и состоянии операции. Каждой операции на шкале времени присуще одно из состояний: «запланировано», «в процессе» или «завершено». Таким образом, временная шкала — одна из основополагающих концепций для управления данными в Hudi [13].

В каждой таблице Hudi существует служебная директория, в которой находится вся метаинформация и журнал (временная шкала), необходимые для обслуживания данных. Метаданные организованы в виде таблиц, которые являются такими же таблицами Hudi, только вложенными в пользовательскую таблицу и скрытыми от пользователя. Эти таблица находится в служебной директории внутри расположения пользовательской таблицы, но недоступна для него. Именно в служебных таблицах хранятся существующие в Hudi структуры данных, такие как фильтр Блума и упрощенные сводки (о структуре которых изложено в параграфе 1.4) [14]. Такой подход является приемлемой практикой и используется в самых распространенных системах для доступа к данным [9].

В системах для доступа к данным существует две стратегии обновления данных, которые определяют внутреннюю структуру таблицы [11], которую необходимо рассмотреть, чтобы сформировать представление о том, какие файлы необходимо индексировать для решения задачи отсеивания нерелевантных файлов:

1. **Копирование при записи** — при таком подходе данные хранятся в файлах большого размера, в формате, оптимизированном для более быстрого чтения. При операции записи генерируется новая версия файла данных, которая создаётся путём слияния существующего файла данных с новыми поступившими данными, такой подход обеспечивает быстрое чтение данных, однако запись данных становится долгой, так как при

каждом обновлении необходимо копировать файл с данными большого размера [11].

2. **Слияние при чтении** — подход, при котором каждая операция записи данных создает отдельный файл небольшого размера, оптимизированный для более быстрой записи, такие файлы называются **дельта-файлы**. При чтении необходимо считывание всех данных дельта-файлов для формирования результата, такой подход обеспечивает более быструю запись, однако чтения данных становится долгим, так как необходимо считывать множество файлов и производить слияние данных их дельта-файлов при каждом чтении. Такой подход имеет особенность — очевидно, что нельзя допустить неограниченный рост количества дельта-файлов, поэтому при достижении определенного порога количества дельта-файлов, происходит процесс **компактизации** — слияние множества дельта-файлов в один файл большого размера, в формат, оптимизированный для чтения. [11].

Стратегия «Копирование при записи» является частным случаем стратегии «Слияние при чтении», при которой порог количества дельта-файлов равен нулю, а процесс компактизации происходит при каждой записи.

Устройство Hudi полностью реализует обе стратегии обновления данных следующим образом: для каждой стратегии существует тип таблицы с соответствующим названием. Тип таблицы задается при ее создании, а управление процессом обновления данных для каждого типа таблицы происходит по соответствующей стратегии, как описано выше [15].

Размер таблицы увеличивается при добавлении данных, как и размер файлов. Для размера файла существует предел, так как для реализации обеих стратегий необходимо создавать файлы, размер которых постоянно растет. Очевидно, что для реализации каждой из стратегий, необходимо иметь множество файлов с данными, каждый из которых содержит определенную часть из набора данных, который содержится в таблице. Таким образом процесс обновления данных для каждой из стратегий происходит более эффективно. Так как системы доступа к данным поддерживают возможность

исторических запросов, то каждый из файлов с данными представлен в нескольких вариантах, каждый из которых отвечает определенной отметке времени. Множество этих файлов называется **файловой группой** При достижении заданного конфигурируемого предела размера файла в файловой группе, формируется новая файловая группа. Независимо от типа таблицы, каждый файл данных ассоциирован с определённой файловой группой, а каждая файловая группа обладает уникальным случайно генерируемым идентификатором [16].

Структура данных, представляющая собой фильтр для отсеивания нерелевантных файлов, обновляется синхронно с обновлением набора данных в таблице, это обязательно, чтобы не допустить потери данных при неверном отсеивании файлов. Соответственно **таблица, содержащая фильтр, должна реализовывать ту же стратегию управления данными, что и пользовательская таблица**. Это верно, так как каждая операция записи в таблицу требует записи в фильтр, а каждая операция чтения данных требует чтение фильтра.

Файловые группы расположены в директориях файловой системы, путь к директории называется **партицией**. Партиция генерируется на основе заданной пользователем схемы партиционирования. Схема партиционирования включает в себя список атрибутов таблицы, значения которых определяют путь расположения файловой группы [16]. Партиционирование в системах для доступа к данным необходимо для решения множества задач, в том числе для ускорения интервальных запросов — необходимо отсеивать те пути к файловым группам, значение атрибута в которых не удовлетворяет предикату запроса. Данный подход является тривиальным, имеет множество ограничений и уже реализован в системах для доступа к данным [5], поэтому не рассматривается в данной работе.

Учитывая организацию файлов в системах для доступа к данным, можно прийти к выводу, что в действительности фильтр должен отсеивать не конкретные файлы с данными, а файловые группы, в силу того, что запрос может быть историческим. Выбор конкретной версии файла из файловой

группы зависит от момента времени исторического запроса. Таким образом, **расположение файла** — это упорядоченная пара (партиция, идентификатор файловой группы), которая однозначно определяет расположение файловой группы, выбор же конкретного файла из файловой группы зависит от момента времени, запрашиваемого в историческом запросе, этот процесс не зависит от условия интервального запроса, поэтому в дальнейшем в данной работе говоря о расположении файла, будем иметь ввиду именно партицию и файловую группу, а не конкретный файл.

1.3 Требования к внедряемой структуре данных

Далее рассмотрим основные понятия, связанные с индексами, а также сформируем ряд требований к структуре данных для отсеивания нерелевантных файлов для ее внедрения в систему для доступа к данным. Индексы представляют собой структуры данных, возникшие из необходимости обеспечить быстрый поиск информации, хранящейся в базах данных. В отсутствие индексов поиск информации по всем записям, удовлетворяющим определённым критериям, требовал бы последовательного доступа к каждой записи для проверки её соответствия условиям [17]. Для базы данных, содержащей N элементов, это потребовало бы времени порядка $O(N)$, что для современных баз данных является неэффективным.

Определение 3. Индекс — это структура данных, которая позволяет ускорить процесс поиска записей с определенным свойством за счёт использования дополнительной памяти и выполнения дополнительных операций записи для поддержания своей структуры [17].

Используя это определение, можно констатировать, что фильтры для отсеивания нерелевантных файлов являются индексами. В действительности, «Индекс» является более точным термином для описания структуры данных, позволяющей ускорить условные запросы, поэтому в рамках данной работы эти термины несут одинаковый смысл, а использование конкретного

определения по большей части зависит от уже существующего названия определенной структуры.

В контексте больших данных для сравнения методов индексации обычно рассматриваются три основные характеристики данных [18]:

1. **Объём данных**: Управление и индексация больших объёмов данных представляют собой наиболее очевидную проблему в данной предметной области [19]. Современные объёмы данных в облачных сервисах, обрабатываемые за приемлемое время, измеряются терабайтами и, как ожидается, в ближайшем будущем достигнут петабайт [20].
2. **Скорость изменения данных**: Данные постоянно изменяются и обновляются. Обработка данных в реальном времени, известная как «потоковая обработка», становится альтернативой традиционной пакетной обработке [19]. Секторы, такие как электронная коммерция, генерируют значительные объёмы данных, например, через веб-клики, которые непрерывно поступают в поток [21].
3. **Разнообразие данных**: Большие данные поступают из множества источников, включая веб-страницы, веб-журналы, социальные сети, электронные письма, документы и данные от сенсорных устройств. Различия в форматах и структуре этих данных создают проблемы, связанные с их разнообразием [22].

Объём индексируемых данных играет ключевую роль при выборе структуры данных для реализации в озерах данных. Даже если размер индекса значительно меньше объёма данных, например, в случае с индексированием данных объёмом в терабайты, размер индекса может достигать нескольких гигабайт. Это сопоставимо с объёмом оперативной памяти современных персональных компьютеров, что делает невозможным размещение таких индексов в оперативной памяти вычислительного узла. Кроме того, индекс обычно хранится в распределённой файловой системе, доступ к которой осуществляется по сети. Следовательно, загрузка индекса в оперативную память, даже если он теоретически помещается, может занять значительное время. Это делает даже самый точный и эффективный индекс

неэффективным при необходимости передавать большие объёмы данных по сети. Таким образом, важно оценивать размер персистентного хранения реализуемой структуры данных [20].

Скорость изменения данных влечёт за собой требования к возможности обновления индекса, включая поддержку операций обновления, добавления и удаления записей. При этом операции записи в индекс не должны сильно замедлять запись данных в таблицу. Отдельно следует упомянуть задачу индексации уже существующих наборов данных: при больших объёмах данных выполнение этой задачи за разумное время осуществимо не для каждой структуры данных.

Разнообразие данных в системах для доступа к данным не является большой проблемой, так как данные хранятся в формате, определённом схемой таблицы [23].

Таким образом, требования к внедряемой структуре данных в систему для доступа к данным следующие:

1. Индекс должен быть обновляемым, включая тот факт, что он должен поддерживать исторические запросы и операцию удаления данных.
2. Индекс должен отвечать на интервальные запросы.
3. Требуется индексировать большие объемы данных, поэтому индекс должен занимать мало места (более точные оценки приведены в следующем параграфе).
4. Индекс должен реализовывать ту же стратегию управления данными, что и пользовательская таблица.
5. Индекс должен поддерживать возможность индексации разных типов данных (Параграф почему??).

1.4 Анализ существующих решений

Для создания структуры данных, которая может помочь решить задачу пропуска нерелевантных файлов для интервальных запросов, необходимо ознакомиться с существующими решениями в платформах для доступа к данным. Существует множество специфичных индексов, решающих разные

задачи, однако одно из определенных нами требований — возможность индексировать разные типы данных, таким образом для рассмотрения не подходят индексы работающие, например, только со строковыми типами. Более того, для решения задачи необходима поддержка интервальных предикатов. В существующих платформах для доступа к данным таких решений несколько, это: упрощенные сводки, точные индексы и фильтры Блума [9]. Далее рассмотрим каждую из структур подробнее.

1.4.1 Упрощенные сводки

В параграфе 1.1 уже была рассмотрена структура данных «Упрощенные сводки». Более того, было доказано (Теорема 1), что такая структура данных является оптимальной для интервальных неограниченных запросов. Однако для интервальных ограниченных запросов использование такой структуры данных может приводить к чтение нерелевантных файлов. Например, каждый файл с данными может содержать удаленные друг от друга значения — отметки времени только за январь и только за декабрь определенного года, но при этом не содержать значений внутри этого диапазона. Тогда для точечных запросов, например, за конкретный день июля, или для запросов, нацеленных на извлечение данных за ограниченный с двух сторон период, например, с июня по август того же года, такой подход приведет к необходимости чтения файлов, которые не содержат нужной информации.

Упрощенные сводки — это тривиальная структура, которая не требует детального описания. Эти структуры просты в использовании и требуют минимальных затрат на хранение. Более того, размер такой структуры не зависит от объёма данных, который она описывает, что позволяет оценить затраты по памяти на хранение таких структур для каждого файла в O(1), так как для одного файла такая структура содержит всего 2 значения — минимальное и максимальное значение атрибута. Однако их эффективность такой структуры зависит от конкретного набора данных внутри индексируемого файла.

В действительности упрощенные сводки оказываются эффективными для упорядоченных атрибутов, которые не содержат пропуском, например, целочисленные первичные ключи. Это такие сценарии использования, где индексируемый атрибут в файле не содержит пропусков.

В дальнейшем в данной работе для краткости будем использовать следующее определение для индексируемых значений:

Определение 4. Ключ — конкретное значение индексируемого атрибута таблицы, то есть значение атрибута, по которому построен индекс для отсеивания нерелевантных файлов при заданном интервальном запросе, в котором интервал задан по значениям (ключам) этого же атрибута.

1.4.2 Точные индексы

Точные индексы в системах для доступа к данным отображают значение атрибута каждой отдельной записи на её расположение [24]. Используя такое определение можно сформировать семейство точных индексов, где каждая запись в индексе соответствует одной записи в таблице [17].

Основная проблема таких индексов заключается в том, что объём индекса становится соразмерным с объёмом пользовательских данных, их сложность по памяти есть O(N) [17]. Согласно документации Hudi, размер одной записи в точном индексе составляет приблизительно 50 байт [24]. Для оценки размера индекса относительно размера данных предположим, что размер одной записи составляет 100 КБ [24]. В таком случае для таблицы объёмом 10 ТБ размер индекса достигнет 5 ГБ, что приводит к необходимости чтения всего индекса размером 5 ГБ при любом запросе. Более того, индексируемый атрибут (ключ) может иметь значительно больший размер, чем принято в данных расчетах, например, это может быть строковый тип данных.

Точные индексы отображают значение **каждого ключа**, а значит, и индекс будет иметь размер не меньше, чем размер индексируемого атрибута.

Таким образом, размер индекса в значительной степени зависит от размера индексируемого атрибута.

В действительности, попытка оценить итоговый размер индекса относительно всего набора данных, содержащего помимо индексируемого атрибута множество других атрибутов, не является объективной. Достаточно увеличить количество атрибутов в схеме таблицы, и размер индекса станет меньше относительно размера всей таблицы, как итог, такая оценка не отражает реальный размер индекса, поэтому в дальнейшем в данной работе будем производить оценку размера индекса относительно размера индексируемого атрибута по следующей формуле:

$$\text{размер индекса на ключ} = \frac{\text{размер индекса в байтах}}{\text{кол-во ключей}} \tag{1}$$

Таким образом, итоговый размер точного индекса всегда не меньше размера индексируемого атрибута, а в действительности его размер будет больше, так как помимо самого ключа, необходимо хранить множество расположений данного ключа. Данные индексы созданы для решения другой задачи — точечного обновления записей [24].

1.4.3 Фильтр Блума

Фильтр Блума состоит из битового массива длиной m бит и набора k различных хеш-функций h, которые генерируют значения от 0 до $m - 1$, соответствующие позициям битов в массиве. В начальном состоянии, когда структура данных не содержит элементов, все биты массива установлены в ноль [25].

Фильтр Блума реализует реализует две основные операции: добавлением элемента в множество и проверка принадлежности элемента множеству. Для добавления элемента e необходимо установить в единицу биты на позициях $h_1(e), \ldots, h_k(e)$. Для проверки принадлежности элемента достаточно вычислить значения хеш-функций для этого элемента и убедиться, что все соответствующие биты установлены в единицу, что дает ответ «возможно»,

в случае, если дан ответ «возможно», когда в действительности элемент не принадлежит множеству, такой ответ является **ложноположительным**. Если хотя бы один бит не установлен в единицу, это означает, что элемент не принадлежит множеству — ответ "нет"[25, 26].

Основным преимуществом фильтра Блума является его простота реализации, что и обусловило его широкую популярность, фильтры Блума используются не только в контексте больших данных, но и в задачах дедупликации данных, в сетевых технологиях, файловых системах и других областях [26]. Несмотря на простоту, фильтр Блума остаётся очень компактной структурой данных. Его размер значительно меньше индексируемого набора данных. Например, для количества элементов $n = 1.000.000$ и вероятности ложноположительного ответа $p = 0.01$ при $k = 7$ размер индекса на ключ (1) будет равен примерно 10 бит [25].

Недостатком использования фильтра Блума для отсеивания нерелевантных файлов является необходимость прочитать либо все фильтры, если каждому файлу данных соответствует отдельный фильтр, либо прочитать весь фильтр, если он ассоциирован со всеми файлами данных, содержащими индексируемый атрибут, это следует из описания операций добавления элемента в множество и проверки принадлежности элемента множеству. Более, стандартная реализация фильтра Блума не предполагает возможность удаления элемента из индекса, однако поддержка такой операции необходима.

Однако наиболее значимым недостатком фильтра Блума является его неспособность обрабатывать интервальные запросы. Фильтр Блума эффективно отвечает на запросы о принадлежности одного ключа множеству, то есть на точечные запросы. Однако при необходимости проверить принадлежность каждого ключа из заданного интервала, временная сложность операции составит $O(N \cdot M)$, где N — количество проверяемых фильтров, а M — количество ключей в интервале.

Таким образом, фильтр Блума не подходят для решения задач отсеивания нерелевантных файлов из-за необходимости чтения всей структуры

данных и большой асимптотической сложности обработки интервальных запросов.

ВЫВОДЫ

В данной главе дано определение предметной области — это индексация и фильтрация нерелевантных файлов с данными для их более быстрого чтения в распределенных базах данных при выполнении интервальных запросов. Также описано устройство систем для доступа к данным, из чего сформирован ряд требований к структуре данных, призванной решить задачу пропуска нерелевантных файлов для интервальных запросов. В параграфе 1.4 были исследованы существующие фильтры для пропуска файлов, данные в которых не удовлетворяют предикату интервального запроса.

Существующие решения для пропуска нерелевантных файлов не применимы в рамках интервальных запросов ввиду их большого размера или большой асимптотической сложности для ответа на интервальные запросы.

2 СИТО-ИНДЕКС И ЕГО ВНЕДРЕНИЕ В СИСТЕМУ ДОСТУПА К ДАННЫМ

Для решения задачи пропуска нерелевантных файлов для интервальных запросов необходимо внедрить структуру данных, которая будет одновременно иметь небольшой размер относительно размера индексируемого атрибута ссылка на требования, то есть иметь размер индекса на ключ ссылка на формулу такой, при котором накладные расходы на хранение и доступ к индексу не приведут к тому, что выполнение запроса с использованием индекса будет медленнее выполнения запроса без использования этого же индекса.

Стоит обратить внимание, что в постановке задачи ссылка на параграф 1.1 данные в каждом файле имеют некоторую закономерность, например, еще раз рассмотрим случай, когда упрощенные сводки (1.4.1) дают ложноположительный ответ: файл может содержать в качестве значений отметки времени за только январь и только за декабрь определенного года. Тогда для запросов, нацеленных на извлечение данных за определенные периоды, например, с июня по август того же года, упрощенные сводки дадут ложноположительный ответ для данного файла.

Существуют эвристические подходы для построения индексов, которые используют закономерности в данных. Например, подход Sieve, который основан на наблюдении, что **близкие по значению ключи часто находятся в одном и том же наборе данных**, или распределены среди различных файлов данных [**?**]. Именно данное предположение отражают те случаи, когда упрощенные сводки дают ложноположительные ответы, так как упрощенные сводки никак не учитывают реальное расположение данных внутри каждого файла.

Sieve предлагает следующую модель для определения закономерности в распределении данных:

Пусть k — ключ на множестве ключей K, тогда введем функцию R:

$$R(k) = \begin{cases} R(k-1), & \text{если множества расположений } k \text{ и } k-1 \text{ равны,} \\ R(k-1)+1, & \text{в остальных случая .} \end{cases} \tag{2}$$

Значение функции R — есть общее количество раз, когда набор расположений изменялся для двух соседних ключей. Далее производится линеаризация отдельных сегментов данной функции. Таким образом каждый сегмент отражает некоторую закономерность в распределении ключей между файлами. Например, если линеаризованный участок (сегмент) параллелен оси абсцисс, это значит, что для множество расположений ключей в данном сегменте не менялось, соответственно для того чтобы сохранить информацию о данном сегменте можно использовать меньше памяти, достаточно сохранить минимальное и максимальное значений ключей для данного сегмента и множество расположений, в которых ключи из этого промежутка расположены, и напротив, если тангенс угла линеаризованного участка равен единице, это значит, что расположения ключей различаются для каждых соседних ключей, то есть этот участком необходимо разделить на более мелкие блоки, каждый из которых должен содержать информацию о минимальном и максимальном значении ключей каждого блока и множество расположений этих ключей, для того чтобы снизить вероятность ложноположительных ответов для этого сегмента.

Существует множество вариантов линеаризации сегментов функции R, однако модель Sieve предлагает следующий вариант: какой вариант? + PR + ошибка

Используя данный подход множество ключей K разбивается на **сегменты**, каждый из которых свидетельствует о некоторой закономерности в распределении ключей, которые находится внутри данного сегмента. Далее каждый **сегмент разбивается на блоки** одинаковой длины, размер блока

b_{size} определяется как:

$$b_{size} = \frac{s_{size}}{R(s_{max}) - R(s_{min}) + 1} \tag{3}$$

Где s_{min} — минимальное значение ключа в сегменте, s_{max} — максимальное значение ключа в сегменте, а $s_{size} = s_{max} - s_{min}$ – размер сегмента.

Таким образом, вероятность ложноположительного ответа b_{fpr} для блока:

$$\begin{gathered} b_{fpr} = 1 - \frac{1}{R(b_{max}) - R(b_{min})} = \\ = 1 - \frac{1}{(PR(b_{max}) + \varepsilon) - (PR(b_{min}) - \varepsilon)} = 1 - \frac{1}{2\varepsilon} \end{gathered} \tag{4}$$

Любой ключ принадлежит ровно одному блоку, таким образом это есть вероятность ошибки для одного любого ключа.

Используя данный подход, реализуем структуру данных, которую назовем **«Сито-индекс»**. Данная структура будет состоять из сегментов, каждый из которых будет представлен отдельным файлом в хранилище данных. Каждый такой файл будет содержать в себе набор блоков. Далее подробнее опишем операции над данной структурой данных, принимая ввиду ограничения платформ для обработки больших данных 1.3.

что такое в итоге метаданные, сегмент и блок, + картинка Сито индекса?

2.1 Алгоритм построения

Алгоритм построения состоит из двух этапов:

1. Сортировка ключей.
2. Построение сегментов индекса.

Первый этап является вычислительно самым трудоемким, так как одно из ограничений в платформах для обработки больших данных — ограниченный размер оперативной памяти вычислительных узлов (1.3). Для того чтобы построить функцию R, необходимо отсортировать весь набор ключей для того чтобы отслеживать значение функции R. Так как размер

всего множества ключей K может не уместиться в оперативную память одного вычислительного узла и даже всех вычислительных узлов вместе взятых. Однако данную операцию возможно реализовать с использованием системы для обработки данных ссылка, то есть производить данные вычисления, используя вычислительные ресурсы всего кластера, так как система для обработки данных имеют возможность работы выполнения таких запросов как сортировка и агрегация данных, которые не умещаются в оперативную память, при этом может задействоваться локальная файловая система узлов кластера, при недостатке оперативной памяти вычислительных узлов ссылка спарк?. В данной работе системой для обработки данных является Apache Spark. Точкой входа для создания индекса является метод $createIndex$ в классе $SparkHoodieBackedTableMetadataWriter$:

Далее приводится алгоритм для формирования отсортированного набора данных, вычисление которого производится с использованием возможностей кластера Spark в распределенной среде:

1. Создание отношения: Используется адаптер Spark, предоставляемый Apache Hudi, для создания отношения. Для этого вызывается метод $createRelation$, которому передаются SQL-контекст, мета-клиент Hudi, схема данных для чтения и массив путей к файлам.
2. Преобразование отношения в набор данных: Полученное отношение преобразуется в набор данных с использованием метода $baseRelationToDataFrame$.
3. Из набора данных выбираются две колонки: целевая колонка для индексации и колонка с именами файлов.
4. Набор данных сортируется по значению колонки для индексации по возрастанию.
5. Группировка и агрегация данных: Выполняется группировка данных по колонке для индексации. Применяется агрегирующая функция к колонке с именем файла, которая позволяет собрать уникальные значения имен файлов в каждой группе.

Полученный набор данных, содержащий отсортированные и агрегированные данные, возвращается как результат.

Второй этап заключается в формировании сегментов и сохранению их в хранилище данных, получая из вычислительного кластера каждый следующий ключ вместе с множеством его расположений один за одним и отслеживая значение ошибки для текущего сегмента. Данную операция производится на одном вычислительном узле, так как для формирования сегментов индекса необходимо пройти все множество ключей по возрастанию один раз. Формирование очередного сегмента заканчивается при достижении им определенного размера (так как объем оперативной памяти вычислительного узла ограничен) или при достижении порога ошибки что это. После формирования очередного сегмента, необходимо вычислить размер блоков (3) внутри него и сформировать эти блоки, после чего сегмент сериализуется в хранилище данных и может быть выгружен из оперативной памяти. Таким образом возможно преодолеть ограничение оперативной памяти вычислительного узла и невозможность выполнить сегментацию на в системе обработки данных. Далее приведен алгоритм формирования сегментов Сито-Индекса:

1. Инициализация переменных:
   (a) Создается пустой список $segments$ для хранения метаданных сегментов индекса.
   (b) Инициализируется объект $segmentBuilder$ для построения блоков текущего сегмента.
   (c) Устанавливается значение погрешности сегмента $e = 100$.
   (d) Инициализируются переменные $R_{value} = 0$, $prevKey$ (пустой) и $prevIds$ (пустое множество).
   (e) Задаются начальные значения $slHigh = +inf$ и $slLow = 0$.
2. Итерация по набору данных из отсортированных ключей, для очередного ключа k:

(a) Если текущий $segmentBuilder$ пуст, то в него добавляется k и множество расположений k.

(b) Если сегмент не пуст:

i. Увеличивается значение $R_{value} = R_{value} + 1$, если текущий ключ отличается от предыдущего или если расположения ключа изменились.

ii. Вычисляется текущий наклон $sl = R_{value}/(k - segmentBuilder_{minkey})$.

iii. Если наклон sl выходит за границы $slHigh$ или $slLow$ или если сегмент заполнен:

A. Текущий сегмент сериализуется в хранилище данных, а его метаданные и добавляется в список $segments$.

B. Инициализируется новый объект $segmentBuilder$.

C. Значения $slHigh$, $slLow$ и R_{value} инициализируются заново.

iv. $slHigh = min(slHigh, (R_{value} + e)/(k - segmentBuilder_{minkey}))$

v. $slLow = max(slLow, (R_{value} - e)/(k - segmentBuilder_{minkey}))$

vi. $segmentBuilder.add(k, k)$

(c) $prevKey = k$

(d) $prevIds = k$

3. Если $segmentBuilder$ не пуст, то он сериализуется на файловое хранилище и его метаданные добавляются в список $segments$.
4. Список $segments$ сериализуется в хранилище данных — это есть метаданных Сито-индекса.

Реализация данного алгоритма в систему для доступа к данным Apache Hudi приведена в приложении В.

2.2 Поиск в индексе

Для поиска расположений файлов ссылка что это в которых содержатся ключи, удовлетворяющие предикату запроса, необходимо прочитать

с хранилища данных сегменты, которые отвечают за за данные ключи. В первую очередь, необходимо прочитать метаданные индекса, в которых содержится информация о существующих сегментах индекса.

Далее приведен алгоритм поиска расположений для **точечного запроса** по ключу k:

1. Список сегментов является отсортированным, соответственно за время $O(logN)$, где N - количество сегментов в индексе, возможно найти сегмент, который содержит информацию о ключе k, используя бинарный поиск по списку сегментов.
2. Сегмент, содержащий информацию о ключе k, загружается из хранилища данных.
3. В загруженном сегменте в массиве блоков необходимо найти блок, который содержит расположение файлов, в которых присутствует ключ k, сделать это возможно за время $O(1)$ вычислив номер соответствующего блока: $b_{num} = \frac{k - s_{min}}{b_{size}}$.
4. Список расположений из полученного блока — есть множество расположений, которые необходимо включить в рассмотрение для выполнения запроса.

Далее обработки **интервального запроса** по ключам из интервала $[l, r]$ необходимо выполнить следующие шаги:

1. За время $O(1)$ найти блок, содержащий информацию о ключе l, аналогично алгоритму поиска расположений для точечного запроса.
2. Далее последовательно считывать блоки, пока для очередного блока выполняется $r \leq b_{max}$.
3. Перейти к чтению блоков из следующего сегмента, если блоки в текущем сегменте закончились, а $r \leq b_{max}$ все еще выполняется.

Реализация данных алгоритмов в системе для доступа к данным Apache Hudi приведена в приложении В.

2.3 Обновление индекса

Существуют две операции обновления индекса, которые требуют модификации индекса для отражения актуального состоянии, чтобы не допустить ложноотрицательных ответов. При добавлении, обновлении или удалении записей возможны два следующих случая сценария:

1. Добавление ключа k в расположение r.
2. Удаление ключа k из расположения r.

В дальнейшем будем называть упорядоченную пару (k, r) **индексной записью**.

Сито-индекс на хранит в себе индексные записи, так как использует совершенно другую структуру, описывающую промежуток ключей, этой структурой является **блок** ссылка?. Однако определением «индексная запись» удобно оперировать в рамках операций обновления индекса.

С этой точки зрения, индекс должен быть синхронизирован с текущим состоянием таблицы. Для достижения этой синхронизации необходимо применять те же механизмы управления параллельными версиями (MVCC), которые используются для управления таблицей.

После накопления множества изменений (дельт) в индексе, необходимо провести процесс компактизации индекса в синхронизации с компактизацией индексируемой таблицы. Для поддержки фильтрации нерелевантных данных при выполнении исторических запросов требуется версионирование не только данных, но и индекса. При компактизации таблицы старая версия сегмента должна быть помечена тем же временным штампом, что и старая версия данных, после чего обновленный сегмент записывается в файловую систему.

2.3.1. Добавление записи Процесс добавления записи, ключевой атрибут которой отсутствует в индексе, включает добавление в соответствующую группу файлов, ассоциированную с сегментом дельта-файла. Дельта-файл содержит информацию о добавленных ключах и о расположении соответствующих файлов данных на файловой системе.

2.3.2. Удаление записи Большинство теоретических описаний подходов, используемых при создании Сито-индекса, не включает описание операции удаления ключа из индекса, хотя поддержка данной операции необходима. Подход к реализации этой операции аналогичен процессу добавления записи: необходимо записать в дельту, что ключ k был удален из индекса. В процессе компактизации следует удалить из индекса запись с указанием местоположения. Однако этот подход может быть некорректен при удалении ключа из блока, если местоположение, ассоциированное с удаленным ключом, соответствует также другим ключам, которые не были удалены. Это может привести к ложноотрицательным результатам, когда записи, соответствующие данному местоположению, окажутся утраченными.

Процесс удаления записи, ключевой атрибут которой присутствует в индексе, реализуется аналогично процессу добавления записи. Необходимо определить файловую группу, ассоциированную с сегментом, внести изменения и записать в дельта-файл информацию о удаленных ключах и их расположениях. Для предотвращения ложноотрицательных результатов в списке местоположений каждого блока следует хранить не только расположение файлов данных, но и количество ключей, которым соответствует данное расположение. Таким образом, удаление местоположения из подсегмента заключается в уменьшении второго элемента данной пары на одну единицу, если значение больше единицы, и в удалении пары из списка, если это значение равно единице. Аналогичные изменения необходимы также в процедуре добавления записей. Этот подход, называемый подсчетом (англ. counting), является общепринятым [16].

2.3.3. Обновление записи Обновление записи в системе Hudi представляет собой более сложную концепцию по сравнению с традиционными системами управления базами данных (СУБД). В отличие от стандартных СУБД, Hudi поддерживает выполнение исторических запросов. В общем случае, при обновлении набора записей сохраняется предыдущий актуальный файл данных, а также создается новый файл данных, который отражает актуальное состояние данных после обновления. Таким образом, местополо-

жение данных не изменяется с появлением нового файла данных, который соответствует последующему моменту времени на временной шкале. Для операций обновления записи по уже существующему ключу обновление индекса не требуется.

2.4 Результат внедрения

В данном разделе представлены результаты сравнения производительности индекса Sieve и полного сканирования таблицы при выполнении точечных и интервальных запросов с различной селективностью данных. Набор данных был сгенерирован с использованием бенчмарка TPC-H, причем запросы выполнялись на таблице «lineitem» по первичному ключу.

В рамках тестирования индекс был построен на двух наборах данных: первые 20 миллионов строк (набор 1) и первые 600 миллионов строк (набор 2) из таблицы "lineitem". Выбор данных объемов данных был обусловлен возможностями Spark в режиме standalone, где использовался один вычислительный узел с объемом выделенной памяти 2 ГБ. Это позволило проверить возможность индексации таблицы из 20 миллионов записей с полной загрузкой всех значений атрибута в память, в то время как загрузка 600 миллионов записей в память оказалась невозможной.

Размер индекса для первого набора данных составил 11 МБ при общем размере индексируемого атрибута 160 МБ, а для второго — 300 МБ при общем размере индексируемого атрибута 5 ГБ. В обоих случаях средний размер блока составил $p = 32$. При этом размер индекса составил всего около 6 от исходного размера индексируемого атрибута. Учитывая, что в реальных сценариях использования таблицы могут содержать десятки атрибутов, размер индекса будет составлять лишь десятые и сотые доли процента от общего размера таблицы. Далее представлены результаты тестирования в виде гистограмм, где по оси ординат отложено время выполнения запроса в секундах.

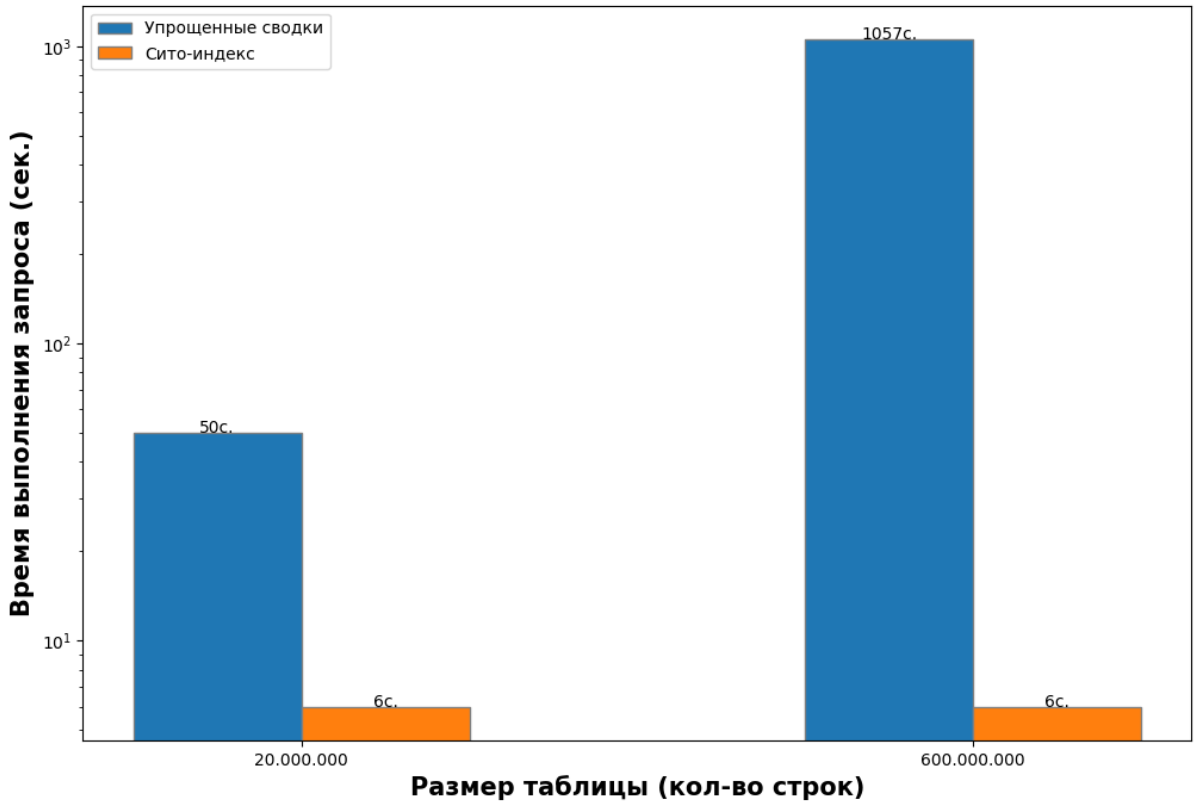

Рисунок 1 – Сравнение времени выполнения точечного запроса с использованием упрощенных сводок и Сито-индекса на двух таблицах разного размера

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы были рассмотрены и применены структуры данных, способствующие ускорению точечных и интервальных запросов в условиях распределенных баз данных, типичных для аналитических систем. Результатом работы стал синтез структуры данных Сито-индекс. Данная структура данных позволила значительно повысить эффективность обработки запросов с условиями за счет быстрого отсеивания блоков данных, которые не соответствуют заданным в запросе ограничениям. Эксперименты показали, что внедрение Sieve индекса привело к приросту производительности на аналитических запросах из бенчмарка TPC-H, подтверждая тем самым эффективность предложенных решений.

Выпускная квалификационная работа выполнена мной самостоятельно с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из

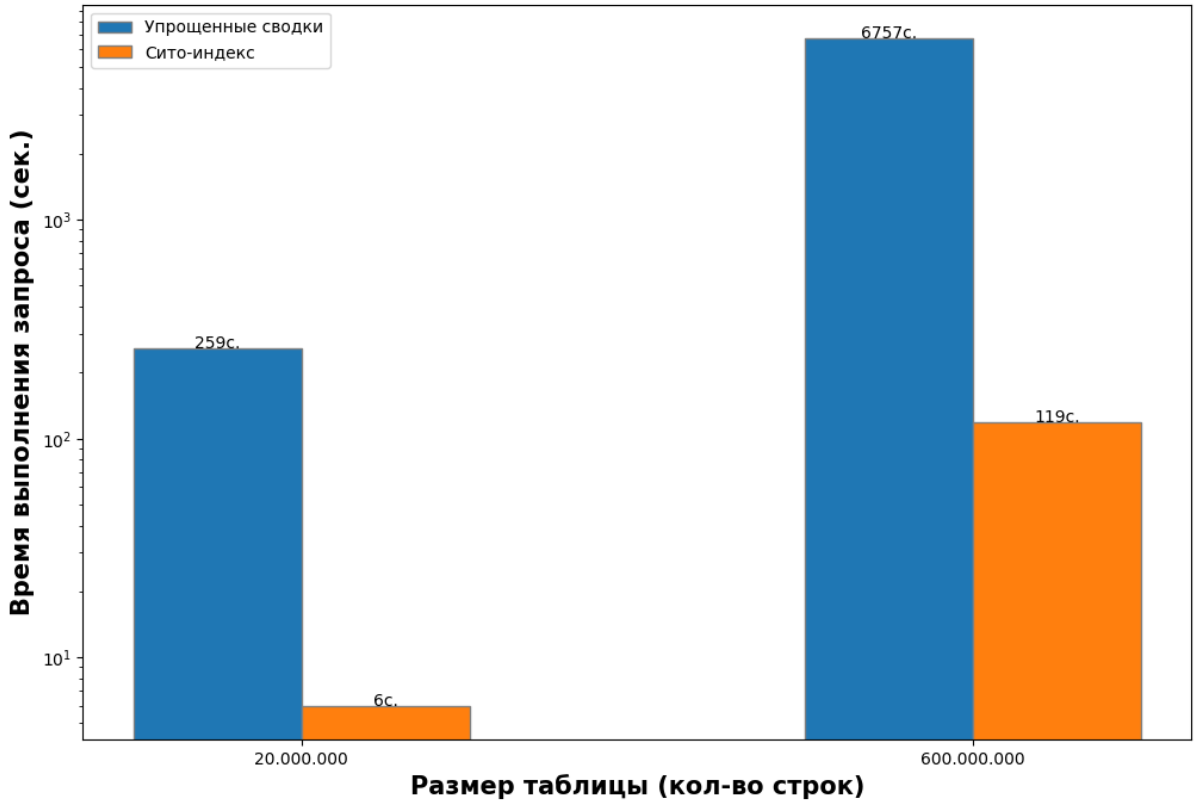

Рисунок 2 – Сравнение времени выполнения интервального запроса с использованием упрощенных сводок и Сито-индекса на двух таблицах разного размера

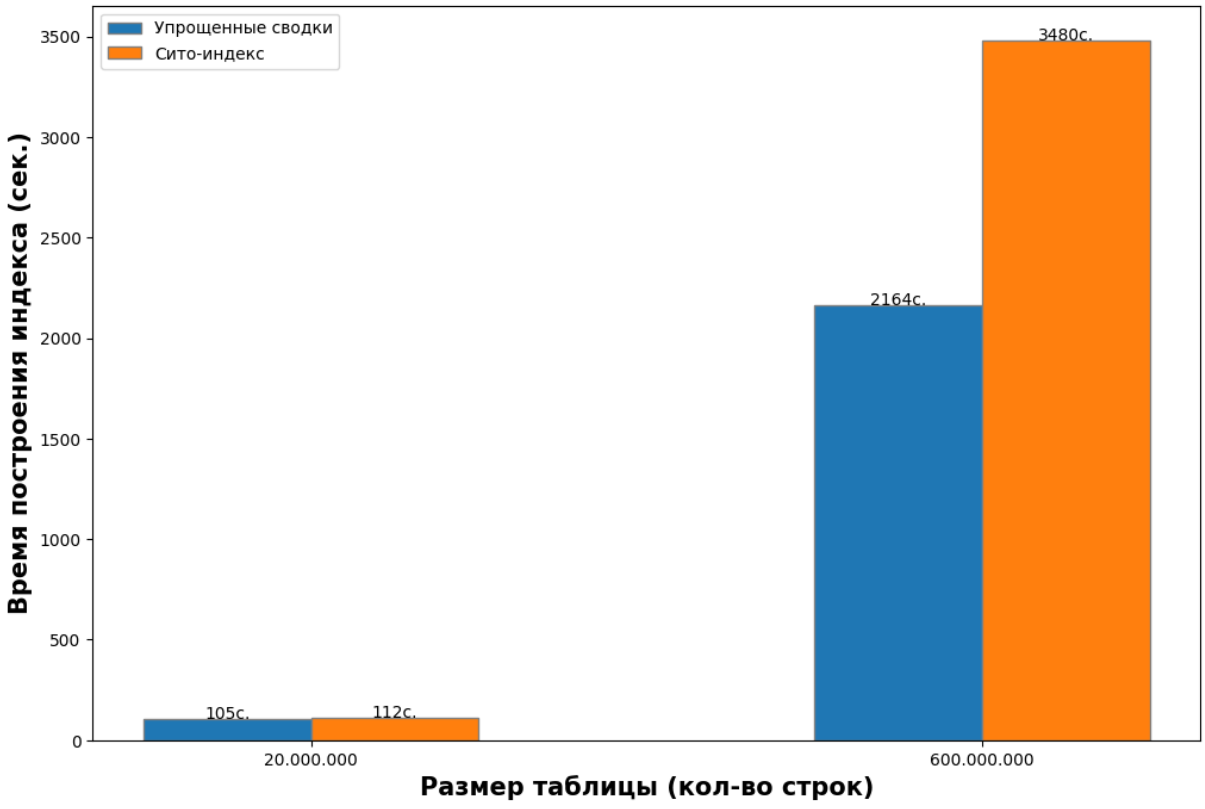

Рисунок 3 – Сравнение времени построения упрощенных сводок и Сито-индекса на двух таблицах разного размера

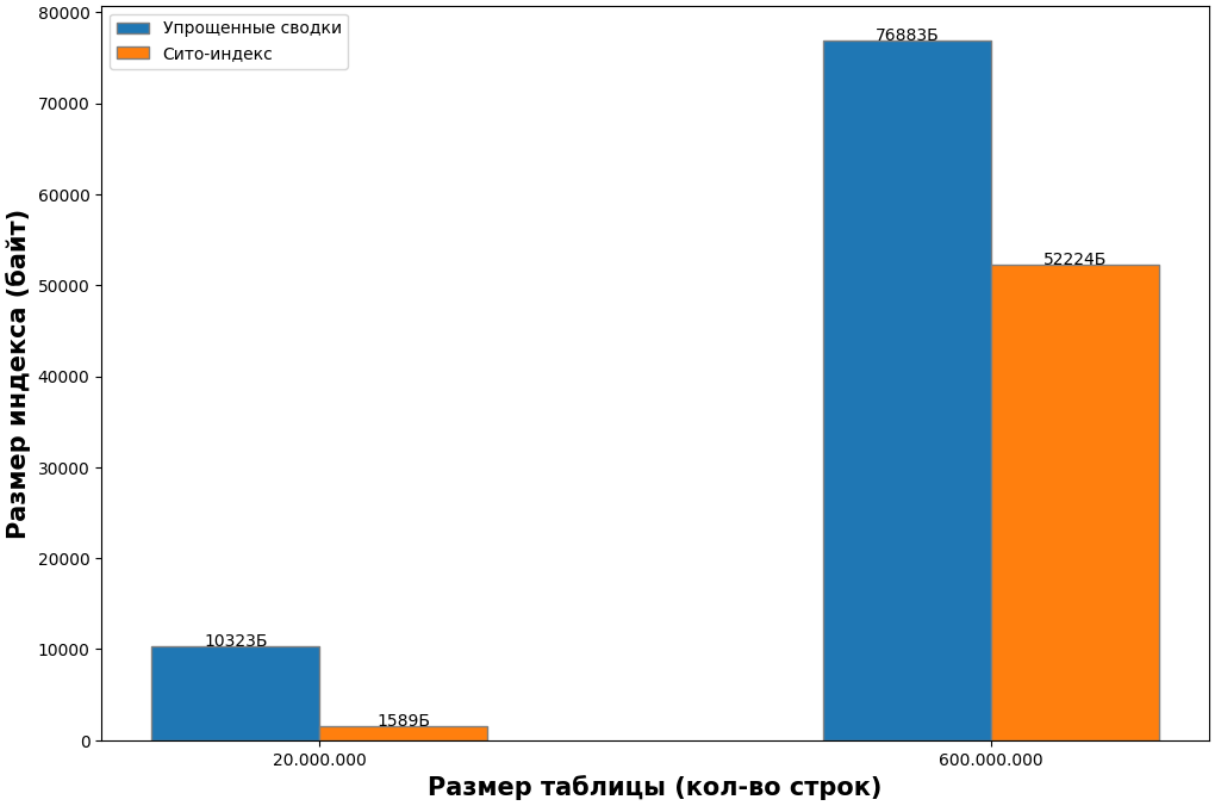

Рисунок 4 – Сравнение итогового размера упрощенных сводок и Сито-индекса на файловой системе для двух таблицах разного размера

опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Бухнер Марк Евгеньевич _____

(подпись)

«___» _____ 2024 г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Sagiroglu S. Sinanc D. Big data: A review // 2013 International conference on collaboration technologies and systems (CTS). — 2013. — P. 42.
2. Mohanty H. Big data: A Primer. — Springer India, 2015. — 198 p.
3. Fan J. Han F. Liu H. Challenges of big data analysis // National science review. — 2014. — Vol. 1, No. 2. — P. 293.
4. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM. — 1978. — Vol. 21, No. 7. — P. 558.
5. Errami S. A. et al. Spatial big data architecture: from data warehouses and data lakes to the Lakehouse // Journal of Parallel and Distributed Computing. — 2023. — Vol. 176. — P. 70–79.
6. El-Rewini H. Abd-El-Barr M. Advanced computer architecture and parallel processing. — John Wiley & Sons, 2005. — 272 p.
7. Pan X. Luo Z. Zhou L. Navigating the Landscape of Distributed File Systems: Architectures, Implementations, and Considerations // Innovations in Applied Engineering and Technology. — 2023. — P. 1–12.
8. Соловьев А. И. Хранение и обработка больших данных // Тенденции развития науки и образования. — 2018. — № 37-6. — С. 47–51.
9. Ta-Shma P. et al. Extensible data skipping // 2020 IEEE International Conference on Big Data. — 2020. — P. 372–382.
10. G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing // In VLDB. — 1998. — P. 476–487.
11. Jain P. et al. Analyzing and comparing lakehouse storage systems // In CIDR. — 2023. — P. 1–4.
12. Гарсиа-Молина Г. Ульман Д. Д. Уидом Д. Системы баз данных. Полный курс. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2003. — 1088 с.

13. Timeline | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/timeline (дата обращения: 01.05.2024).
14. Metadata Table | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/metadata (дата обращения: 01.05.2024).
15. Table & Query Types | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/table_types (дата обращения: 01.05.2024).
16. File Layouts | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/file_layouts (дата обращения: 01.05.2024).
17. Samoladas D. et al. Tree Data Structures and Efficient Indexing Techniques for Big Data Management: A Comprehensive Study // Proceedings of the 26th Pan-Hellenic Conference on Informatics. — 2022. — P. 123–125.
18. Patgiri R. Ahmed A. Big data: The v's of the game changer paradigm // 2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems. — 2016. — P. 17–24.
19. Chen J. et al. Big data challenge: a data management perspective // Frontiers of computer Science. — 2013. — Vol. 7. — P. 157–164.
20. Katal A. Wazid M. Goudar R. H. Big data: issues, challenges, tools and good practices // 2013 Sixth international conference on contemporary computing (IC3). — 2013. — P. 404–409.
21. Kaisler S. et al. Big data: Issues and challenges moving forward // 2013 46th Hawaii international conference on system sciences. — 2013. — P. 995–1004.
22. Vranopoulos G. Clarke N. Atkinson S. Addressing big data variety using an automated approach for data characterization // Journal of Big Data. — 2022. — Vol. 9, No. 1. — P. 8.

23. SQL DDL | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/sql_ddl (дата обращения: 01.05.2024).
24. Record Level Index | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/blog/2023/11/01/record-level-index/ (дата обращения: 01.05.2024).
25. H. Bloom B. Space/time trade-offs in hash coding with allowable errors // Communications of the ACM. — 1970. — Vol. 13, No. 7. — P. 422–426.
26. Patgiri R. Nayak S. Borgohain S. K. Role of bloom filter in big data research: A survey // International Journal of Advanced Computer Science and Applications. — 2018. — Vol. 9, No. 11. — P. 655–661.