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: 42 страницы
Количество иллюстраций: 4
Количество таблиц: 0
Количество литературных источников: 36
Количество приложений: 2
Ключевые слова: базы данных, распределенные системы, фильтры, индексы, структуры данных, ускорение запросов, точечные запросы, интервальные запросы, SQL.

Объектом исследования данной работы являются структуры данных, позволяющие отсеивать файлы, данные в которых не удовлетворяют предикату точечных или интервальных запросов. Цель работы — ускорение точечных и интервальных запросов в контексте систем для обработки больших данных.

Распределенные базы транзакционных данных предназначены для хранения объемов данных, которые невозможно разместить на одной физической вычислительной машине. В рамках данной работы фильтры внедряются и сравниваются в рамках системы для доступа к данным Apache Hudi. Актуальность работы состоит в том, что такие системы не имеют развитых инструментов для отсеивания нерелевантных файлов. Отчего существует множество сценариев, когда такие запросы выполняются медленно.

Результатом данной работы является разработка и внедрение структуры данных «Сито-индекс» для пропуска нерелевантных файлов, в итоге с помощью данного фильтра удалось добиться прироста скорости выполнения точечных запросов на порядки, а интервальных запросов на порядок для тех сценариев использования, когда имеющиеся в Apache Hudi фильтры для отсеивания нерелевантных файлов — упрощенные сводки были не способны отсеивать файлы.

Областью применения такой структуры данных являются системы для доступа к данным в платформах для обработки больших данных.

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

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Платформы для анализа больших данных состоят из множества сервисов, среди них 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 трудоемкость операций ввода-вывода к такому хранилищу в одно из основных узких мест при обработке запросов. Существует множество подходов к организации файлов для их эффективного чтения, позволяющих 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.

Существующие методы отсеивания файлов с данными используют упрощенные сводки (например, минимальные и максимальные значения каждого атрибута) для всех файлов данных, чтобы фильтровать те файлы, в которых не содержится записей, удовлетворяющих 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 и внедрение эффективных методов индексации данных для ускорения обработки запросов с предикатами в контексте платформ больших данных, с последующим проведением экспериментов на реальных наборах данных.

Задача исследования состоит в анализе существующих подходов и разработке структуры данных, которая позволит ускорить обработку интервальных запросов в платформах для обработки больших 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.

Распространенной ошибкой при определении понятия большие данные является попытка придать данным численную оценку объема, потому что объем 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 объемов 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. **Система обработки данных** — распределенная вычислительная система (кластер), которая отвечает на запросы пользователей.
2. **Система доступа к данным** — управляет организацией данных для 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 [6] —

транзакционная система 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 [6] файловую систему Hadoop Distributed File System (HDFS), являющуюся частью проекта Apache Hadoop.

Для систем существует два способа увеличения общей производительности, а также увеличения предельного объема данных, который возможно сохранить в системе [7]:

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 [7] 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.

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, они системы хорошо масштабируются горизонтально. Это необходимо для надежного хранения данных и возможности дешевого добавления памяти в файловую систему [8].

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 локального

запоминающего устройства или оперативной памяти [9]. Существует разные способы организации данных для быстрого их чтения [10], позволяющие 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 эффективен, так как мы уже выяснили, что чтение данных с внешнего файлового хранилища по сети является узким местом.

Для дальнейшего описания предметной области необходимо дать несколько определений:

Определение 1. Запрос с условием — запрос к системе доступа к данным с условием (условиями) 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 запросов с предикатами.

Без ограничения 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 атрибута одному определенному значению.
2. **Интервальный ограниченный запрос** — запрос записей со значениями атрибута из интервала, ограниченного с обеих сторон.
3. **Интервальный неограниченный запрос** — запрос записей со значениями атрибута из интервала, ограниченного только с одной стороны.

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

Существующие методы отсеивания файлов с данными используют **упрощенные сводки** — минимальные и максимальные значения каждого атрибута для каждого файла данных, чтобы фильтровать те файлы, в которых не содержится записей, который удовлетворяют предикату запроса [11, 12]. Данная структура является оптимальной по памяти и времени для интервальных неограниченных запросов. Вполне вероятно, что такое утверждение уже описано в литературе, однако автор данной работы не нашел формального доказательства этого факта, поэтому далее приводится авторское формальное доказательство:

Теорема 1 (Об оптимальности упрощенных сводок)**.** *Для неограниченных интервальных запросов фильтрация файлов на основе упрощенных сводок требует $O(1)$ дополнительной памяти и $O(1)$ времени на фильтрацию одного файла с данными. Таким образом, для неограниченных интервальных запросов такая фильтрация является асимптотически оптимальной.*

Доказательство.

Дан файл, содержащий набор данных $\{v_1, v_2, \ldots, v_n\}$ из некоторого интервала $[a, b]$. Также задан предикат запроса $P(x)$, 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 хранить всего два элемента в памяти, что составляет $O(1)$ памяти.

Проверка пустоты пересечения диапазона $[v_{\min}, v_{\max}]$ с диапазоном, заданным предикатом $P(x)$ может быть выполнена за время $O(1)$:

- Если $P(x) = (x \geq c)$, то для файла предикат ложен только при $v_{\max} < c$, и в этом случае файл можно пропустить.
- Если $P(x) = (x \leq c)$, то для файла предикат ложен только при $v_{\max} > c$, и в этом случае файл можно пропустить.

Теорема 1 доказана.

Однако такая структура не является асимптотически 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 файлов и фильтрация нерелевантных файлов с данными для быстрого выполнения интервальных запросов чтения в распределенных базах данных.

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, без ограничения общности [13], рассмотрим в данной работе устройство одной из таких систем — **Apache Hudi** [13].

Apache Hudi (далее — Hudi) организует данные в таблицы, реализуя тем самым реляционную модель [14]. Hudi поддерживает работу с историческими запросами: при обновлении данных создается новый снимок обновленных данных в виде отдельного файла, в то время как предыдущий остается доступным для исторических запросов. Каждой операции, изменяющей состояние данных, присваивается уникальная отметка на временной шкале. Ис-

пользуя концепцию временной шкалы, Hudi реализует журналирование всех операций над данным [15], используя этот журнал, реализуются различные стратегии управления параллельным доступом к данным. На каждый момент времени приходится не более одного действия, что создает линейный порядок на всех операциях 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 [16].

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 таблицы находятся в служебной директории внутри директории с пользовательской таблицей, но недоступны для пользователя [17]. Именно в служебных таблицах хранятся существующие в Hudi структуры данных, такие как фильтр Блума и упрощенные сводки (о структуре которых изложено в разделе 1.4). Такой подход является приемлемой практикой и используется в самых распространенных системах для доступа к данным [11].

В системах для доступа к данным существует две стратегии обновления данных, которые определяют внутреннюю структуру таблицы [13], 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 создается путем слияния существующего файла данных с новыми поступившими данными. Такой подход обеспечивает быстрое

чтение данных, однако запись данных становится медленной, так как при каждом обновлении необходимо копировать файл с данными большого размера [13].

2. **Слияние при чтении** — подход, при котором каждая операция записи данных создает отдельный файл небольшого размера, в котором хранятся не все данные, а только изменения, внесенные операцией обновления, такие файлы оптимизированы для быстрой записи и называются **дельта-файлами**. При чтении необходимо считывание всех данных дельта-файлов для формирования результата, такой подход обеспечивает более быструю запись, однако чтение данных становится медленным, 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** — слияния множества дельта-файлов в один файл большого размера, в формат, оптимизированный для чтения [13].

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 [18].

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 определенной файловой группой, а каждая файловая группа обладает уникальным случайно генерируемым идентификатором [19].

Структура данных, представляющая собой фильтр для отсеивания нерелевантных файлов, обновляется синхронно с обновлением набора данных в таблице, это необходимо 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 [19]. 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, поэтому введем следующее определение.

Определение 3. Расположение файла (или расположение) — 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 определенным критериям, требовал бы последовательного доступа к каждой записи для проверки ее соответствия условиям [20]. Для базы данных, содержащей N элементов, это потребовало бы времени порядка $O(N)$, что для современных баз данных является неэффективным.

Определение 4. Индекс — это структура данных, которая позволяет ускорить процесс поиска записей с определенным свойством за счет использования дополнительной памяти и выполнения дополнительных операций записи для поддержания своей структуры [20].

Используя это определение, можно констатировать, что фильтры для отсеивания нерелевантных файлов являются индексами. В действительно-

сти, «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 [21]:

1. **Объем данных**: Управление и индексация больших объемов данных представляют собой наиболее очевидную проблему в данной предметной области [22]. Современные объемы данных в облачных сервисах, обрабатываемые за приемлемое время, измеряются терабайтами и, как ожидается, в ближайшем будущем достигнут петабайт [23].
2. **Скорость изменения данных**: Данные постоянно изменяются и обновляются. Обработка данных в реальном времени, известная как «потоковая обработка», становится альтернативой традиционной пакетной обработке [22]. Многие отрасли, такие как электронная коммерция, генерируют значительные объемы данных которые непрерывно (в потоке) поступают систему хранения данных [24].
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 [25].

Объем индексируемых данных играет ключевую роль при выборе структуры данных для реализации в системе доступа к данным. Даже если размер индекса значительно меньше объема данных, например, в случае с индексированием данных объемом в терабайты, размер индекса может достигать нескольких гигабайт. Это сопоставимо с объемом оперативной памяти современных персональных компьютеров, что делает невозможным размещение таких индексов в оперативной памяти вычислительного узла. Кроме того, индекс обычно хранится в распределенной 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 объемы данных по сети. Таким образом, важно оценивать размер персистентного хранения реализуемой структуры данных [23].

Скорость изменения данных влечет 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 объемах 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, определенном схемой таблицы [26].

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 разделе 1.4).
4. Индекс должен реализовывать ту же стратегию управления данными, что и пользовательская таблица.

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 [11]. Далее рассмотрим каждую из структур подробнее.

1.4.1 Упрощенные сводки

В разделе 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 чтению 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 объема 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 эффективность такой структуры зависит от конкретного набора данных внутри индексируемого файла.

В действительности упрощенные сводки оказываются эффективными для упорядоченных атрибутов, которые не содержат пропусков, например, упорядоченные целочисленные первичные ключи.

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

Определение 5. 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 ее расположение [27]. Используя такое определение можно сформировать семейство точных индексов, где каждая запись в индексе соответствует одной записи в таблице [20].

Основная проблема таких индексов заключается в том, что размер индекса становится соразмерен пользовательским данным, их сложность по памяти — O(N), где N - количество записей в пользовательской таблице [20]. Согласно документации Hudi, размер одной записи в точном индексе составляет приблизительно 50 байт [27]. Для оценки размера индекса относительно размера данных предположим, что размер одной записи составляет 100 КБ [27]. В таком случае для таблицы объемом 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 количества записей в пользовательской таблице.

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 (Определение 3), в которых содержится данный ключ. Данные индексы созданы для решения другой задачи — точечного обновления записей [27].

1.4.3 Фильтр Блума

Фильтр Блума [28] состоит из битового массива длиной m бит и набора k независимых хеш-функций $h_1, \ldots, h_k$, каждая из которых генерирует значения из $0, \ldots, m-1$, соответствующие позициям битов в массиве. В начальном состоянии, когда структура данных не содержит элементов, все биты равны нулю.

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 равны единицам, 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 равен единице, это означает, что элемент не принадлежит множеству — ответ "нет".

Основным преимуществом фильтра Блума является его точность и простота реализации, что и обусловило его широкую популярность, фильтры Блума используются не только в контексте больших данных, но и в задачах дедупликации данных, в задачах передачи данных по сети, файловых системах и других областях [29]. Несмотря на простоту, фильтр Блума остается очень компактной структурой данных. Не смотря на то, что число бит m, использованных в фильтре Блума, в несколько раз больше чем число ключей в индексируемом наборе, размер фильтра Блума сильно меньше, чем длина записи всех ключей из набор,. Например, для количества элементов $n = 1.000.000$, и вероятности ложноположительного ответа $p = 0.01$, оптимальным значением количества функций будет $k = 7$, а размер индекса на ключ (1) будет равен примерно 3 битам, при размере ключа 32 бита. [28].

Недостатком использования фильтра Блума для отсеивания нерелевантных файлов является необходимость прочитать либо все фильтры, если каждому файлу данных соответствует отдельный фильтр, либо прочитать один большой фильтр, ассоциированный со всеми файлами данных, содержащими индексируемый атрибут. Более того, 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.

Стоит отметить, что существуют модификации фильтра блума, которые поддерживают операции удаления [30] и интервальные запросы [31],

однако существующие реализации фильтра Блума в системах доступа к данным не подходят для решения задач отсеивания нерелевантных файлов из-за необходимости чтения всей структуры данных и большой временной 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 фильтрации нерелевантных файлов для интервальных запросов. В разделе 1.4 были исследованы существующие в системах доступа к данным фильтры для пропуска файлов, данные в которых не удовлетворяют предикату интервального запроса.

Существующие решения для фильтрации нерелевантных файлов не применимы для ускорения выполнения интервальных запросов ввиду их большого размера или большой временной 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 разработать структуру данных, которая будет одновременно иметь небольшой размер относительно размера индексируемого атрибута (см. раздел 1.3), то есть иметь размер индекса на ключ (1) 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.

В данной работе для системы доступа к данным Apache Hudi была разработана и внедрена структура данных, названная **Сито-индекс**. Сито-индекс является модификацией структуры данных Sieve [32], адаптированной для индексации больших наборов данных: часть построения структуры данных проводится распределенно с использованием системы для обработки данных Apache Spark; добавлена поддержка операций обновления индекса (добавление и удаление записей), реализованное с учетом стратегий управления данными «копирование при записи» и «слияние при чтении»; добавлена поддержка ответов на интервальные запросы.

Данная глава организована следующим образом. В разделе 2.1 приводятся необходимые идеи разбиения индексируемых данных из работы [32] о структуре данных Sieve, в разделе 2.2 описан алгоритм построения Сито-индекса, в разделе 2.3 описан алгоритм поиска в Сито-индексе, в разделе 2.4 описан алгоритм обновления Сито-индекса, а в разделе 2.5 представлены численные результаты экспериментов.

2.1 Идея Сито-индекса

Стоит обратить внимание, что в постановке задачи (см. раздел 1.1) данные в каждом файле имеют некоторую закономерность, например, еще раз рассмотрим случай, когда упрощенные сводки (см. раздел 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. Отсюда следует вывод, что данные за январь и декабрь текущего года расположены именно в этом файле (и, возможно, в каких-то других), а данные за период с февраля по ноябрь точно содержатся в других файлах.

Существуют эвристические подходы для построения индексов, которые используют такие закономерности в данных. Например, подход Sieve [32], который основан на наблюдении, что **близкие по значению ключи часто находятся в одних и тех же файлах**. 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.

Напомним, что под расположением в данной работе имеется ввиду путь к файлу и его идентификатор (Определение 3). **Множество расположений** ключа k — набор расположений (файлов), в которых присутствует значение k.

Sieve предлагает следующую модель для определения закономерности в распределении данных:

Пусть k — ключ (Определение 5) из множества ключей, представимых в виде целого числа (целые числа, даты, отметки времени) K, тогда введем функцию R:

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

Значение функции R в точке k — есть общее количество раз, когда множество расположений изменилось для ключей k и $k-1$. Далее для построения модели производится линеаризация отдельных сегментов данной функции. Таким образом каждый сегмент отражает некоторую закономер-

ность в распределении ключей между файлами. Например, если линеаризованный участок (сегмент) параллелен оси абсцисс, это значит, что 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 участок 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.

Введем следующие обозначения: e - допустимая ошибка сегмента (задается пользователем), s — сегмент (множество ключей из интервала $[s_{min}, s_{max}]$), s_{min}, s_{max} — соответственно минимальное и максимальное значения ключа в сегменте, $s_{size} = s_{max} - s_{min}$, $s.E$ — заданное ограничение на число дополнительных ложноположительных ответов для сегмента, b — блок (часть сегмента, множество ключей из интервала $[b_{min}, b_{max}]$, где $s_{min} \leq b_{min} \leq b_{max} \leq s_{max}$), b_{min}, b_{max} — соответственно минимальное и максимальное значение ключа в блоке.

Существует множество вариантов линеаризации сегментов функции R, однако модель Sieve [32] предлагает следующий алгоритм, этот алгоритм будет являться вспомогательным для реализации алгоритма построения Сито-индекса (см. раздел 2.2).

Алгоритм для выделения сегментов:

1. s_{min} — значение, получаемое в процессе алгоритма построения индекса (см. раздел 2.2.)
2. $slLow = 0, slHigh = \infty$.
3. Для очередного ключа k и значения функции $R(k)$ вычислим $sl = R(k)/(k - s_{min})$.

(a) Если $slLow < sl < slHigh$, обновить значения:

$slHigh = min(slHigh, (R(k) + e)/(k - s_{min}))$,

$slLow = max(slLow, (R(k) - e)/(k - s_{min}))$ и перейти к шагу 3.

(b) Иначе сегмент s выделен, начать построение нового сегмента перейдя к шагу 1.

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} внутри сегмента определяется как:

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

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

$$b_{fpr} = 1 - \frac{1}{R(b_{max}) - R(b_{min})} \tag{4}$$

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

2.2 Алгоритм построения Сито-индекса

Структура данных Сито-индекс, как и Sieve, состоит из сегментов, каждый из которых представляет собой отдельный файл в хранилище данных. Каждый такой файл содержит в себе множество блоков. Также отдельно вводится понятие **метаданные** Сито-индекса, это отдельный файл, в котором содержится список сегментов.

Каждый блок содержит в себе множество расположений $b.L$, в которых располагаются ключи из интервала $[b_{min}, b_{max}]$. Добавим для каждого существующего расположения loc счетчик $b.L(loc)$, которым будем вести подсчет количества ключей, которые содержатся в данном расположении loc. Такое решение необходимо для реализации операций обновления индекса.

Далее разработаем операции над данной структурой данных, принимая ввиду ограничения платформ для обработки больших данных (см. раздел 1.3).

Алгоритм построения состоит из двух этапов: сортировка ключей и 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 (см. раздел 1.3). Для того чтобы построить функцию R, необходимо отсортировать весь набор ключей для того чтобы отслеживать изменение значения 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 распределенно, с использованием системы для обработки данных (см. раздел 1.1), 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 [33]. В данной работе системой для обработки данных является Apache Spark.

Структура данных «Sieve» имеет реализованный алгоритм построения индекса, однако его реализация не рассматривает ограничение размера оперативной памяти для построения сегментов. Реализуем модифицированную версию данного алгоритма.

Сперва необходимо выполнить первый этап и получить набор данных df, содержащий отсортированные и агрегированные данные. Второй этап заключается в формировании сегментов и сохранении их в хранилище данных. Этот процесс происходит итеративно, получая из вычислительного кластера каждый следующий ключ вместе с множеством его расположений один за одним и отслеживая значение ошибки для текущего сегмента.

Данная 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, согласно алгоритму для выделения сегментов. После формирования очередного сегмента, необходимо вычислить размер блоков (3) внутри него и сформировать эти блоки, используя алгоритм формирования блоков (представлен далее в этом разделе), после чего сегмент сохраняется в хранилище данных и может быть выгружен из оперативной памяти. Таким образом возможно преодолеть ограничение оперативной памяти вычислительного узла и невозможность выполнить сегментацию в системе обработки данных.

Алгоритм построения Сито-индекса:

На входе: $indexColumn$ — индексируемый атрибут, $locations$ — атрибут расположения записи.

На выходе: метаданные Сито-индекса, множество построенных сегментов S.

1. $df = select(indexColumn, locations).sort(indexColumn)$
   $.groupBy(indexColumn).add(collect(locations))$
2. Создается пустой список S для хранения метаданных сегментов индекса.
3. Итерация по набору данных df из отсортированных ключей, для очередного ключа k и его множества расположений loc:
   (a) Если текущий сегмент s не содержит ключей, то в него добавляется k и множество расположений loc.
   (b) Если значение sl достигло определенной ошибки (шаг 3 алгоритма для выделения сегментов), производится формирование блоков для этого сегмента и он сохраняется в файловое хранилище, а его метаданные в S, после чего он может быть выгружен из памяти.
   (c) Иначе k вместе с его расположениями loc сохраняется в сегменте s для дальнейшего формирования блоков.

4. Если последний строящийся s содержит ключи, производится формирование блоков для этого сегмента и он сохраняется в файловое хранилище, а его метаданные в S.
5. В файловое хранилище сохраняются метаданные индекса, в виде списка метаданных доступных сегментов S, каждый элемент такого списка S_i содержит лишь значения $S_{(i)min}$ и $S_{(i)max}$. Сегменты сохраняются упорядоченно $S_{(i)max} < S_{(i+1)min}$.

Алгоритм формирования блоков:

На входе: выделенный сегмент s.

На выходе: построенный сегмент s с массивом блоков $s.B$.

1. Вычислить размер b_{size} по формуле (3).
2. Вычислить кол-во блоков для данного сегмента $b_{count} = s_{size}/b_{size}$.
3. Сформировать массив блоков B размеров b_{count}.
4. Для каждого ключа k и его множества расположений loc:
   (a) Вычислить $i = (k - s_{min})/b_{size}$.
   (b) Добавить в блок B_i все расположения loc, обновить счетчики расположений $B_i.L$ для каждого расположения из loc.
5. Сохранить массив блоков B, остальная информация может быть выгружена из памяти.
6. Установить значение предела дополнительных ложноположительных ответов для сегмента $s.E = e \cdot b_{count}$

Инициировать создание Сито-индекса возможно с помощью SQL-запроса, этот процесс подробно описан в руководстве пользователя (Приложение А) и описании программы (Приложение Б).

2.3 Алгоритм поиска в Сито-индексе

Для поиска расположений файлов, в которых содержатся ключи, удовлетворяющие предикату запроса, в первую очередь, необходимо прочитать метаданные индекса, в которых содержится информация о существующих сегментах индекса, чтобы получить список сегментов S.

Поиск расположений для точечного запроса схож с алгоритмом поиска, используемом для структуры данных Sieve [32].

Алгоритм поиска расположений для точечного запроса по ключу k:

На входе: ключ k.

На выходе: множество расположений ключа k.

1. Список сегментов является отсортированным, N - количество сегментов в индексе. Сегмент s, содержащий информацию о ключе k, находится бинарным поиском по списку сегментов время $O(logN)$.
2. Сегмент s, содержащий информацию о ключе k, загружается из хранилища данных.
3. Чтобы в загруженном массиве найти блок, содержащий расположения файлов, в которых присутствует ключ k, за время $O(1)$ вычислим номер соответствующего блока $b_i = \frac{k - s_{min}}{b_{size}}$.
4. Список расположений из полученного блока $loc = b_i.L$ — есть множество расположений, которые необходимо проверить при выполнении запроса.

Далее разработаем алгоритм обработки интервального запроса.

Алгоритм поиска расположений для интервального запроса по ключам из интервала $[l, r]$:

На входе: ключи l, r.

На выходе: множество расположений ключей из интервала $[l, r]$.

1. Список сегментов является отсортированным, N - количество сегментов в индексе. Сегмент s, содержащий информацию о ключе l, находится бинарным поиском по списку сегментов время $O(logN)$.
2. Сегмент s, содержащий информацию о ключе l, загружается из хранилища данных.
3. Чтобы в загруженном массиве найти блок, содержащий расположения файлов, в которых присутствует ключ k, за время $O(1)$ вычислим номер соответствующего блока $b_i = \frac{l - s_{min}}{b_{size}}$.

4. Далее последовательно считываем следующие за ним блоки, пока для очередного блока выполняется $r \leq b_{max}$, если не выполняется, алгоритм завершен.
5. После чтения последнего блока в сегменте, переходим к чтению блоков из следующего сегмента (шаг 4), предыдущий сегмент может быть выгружен из памяти.

2.4 Алгоритм обновления Сито-индекса

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

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

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

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

Индекс должен поддерживать исторические запросы (см. раздел 1.3), в разделе 1.2 уже были описаны методы управления данными для их версионирования. На протяжении всей работы факт того, что для каждого файла может хранится несколько исторических версий был опущен, однако при рассмотрении операций обновления индекса этот процесс стоит рассмотреть детальнее. Для работы с историческими запросами индекс должен сам использовать ту же стратегию управления своим содержимым, что и пользовательские данные (см. раздел 1.2).

Исходя из структуры хранения сегментов Сито-индекса в хранилище данных возможно реализовать стратегии управления данными следующим образом:

1. Для реализации стратегии **«копирование при записи»** при поступлении индексной записи генерируется новая версия файла с сегментом, которая создается путем слияния существующего сегмента с поступившими индексными записями.
2. Для реализации стратегии **«слияние при чтении»** каждая поступающая индексная запись сохраняется в дельта-файл (см. раздел 1.1), каждый из которых ассоциирован с файлом сегмента. При компактизации пользовательской таблицы производится слияние файла сегмента и дельта файлов поступивших индексных записей, тот же процесс происходит и при обращении к индексу.

Стратегия «Копирование при записи» является частным случаем стратегии «Слияние при чтении», при котором процесс компактизации происходит сразу же, без создания дельта-файлов. В действительности, процесс управления данными индекса работает согласно этому же процессу для пользовательских данных (см. раздел 1.2).

Таким образом операция добавления или удаления индексной записи сводится к реализации процесса компактизации для сегмента и поступивших индексных записей. Данный процесс может происходить как для обновления индекса при реализации обеих стратегий управления данными, так и при чтении индекса (см. раздел 2.3), чтобы получить актуальное состояние сегментов в запрашиваемый момент времени в историческом запросе.

Алгоритм компактизации сегмента s при добавлении индексной записи (k, loc):

На входе: сегмент s, индексная запись (k, loc).

На выходе: обновленный сегмент s.

1. Необходимо найти и считать блок b (см. раздел 2.3) из сегмента s, который содержит информацию о диапазоне ключей, в который входит ключ k.
2. Если расположение loc уже присутствует в множестве расположений блока $b.L$, необходимо увеличить счетчик расположений $b.L(loc)$ на единицу. В данном случае вероятность ложноположительного ответа для

данного блока не возрастает, так как расположение loc уже присутствует в данном блоке.

3. Если расположение loc отсутствует в множестве расположений блока $b.L$, то необходимо добавить его в это множество, а счетчик расположений $b.L(loc)$ установить равным 1. Если размер блока $b_{size} > 1$ и это операция обновления индекса, то как минимум для одного ключа ответ по нему будет ложноположительный, количество дополнительных ложноположительных ответов сегмента s увеличится: $s.e = s.e + b_{size} - 1$.

Алгоритм компактизации сегмента s при удалении индексной записи (k, r):

На входе: сегмент s, индексная запись (k, loc).

На выходе: обновленный сегмент s.

1. Необходимо обнаружить блок b (см. раздел 2.3) из сегмента s, который содержит информацию о диапазоне ключей, в который входит ключ k.
2. Если счетчик расположений $b.R(r) = 1$, необходимо удалить расположение и его счетчик из множества расположений данного блока. Количество дополнительных ложноположительных ответов в данном случае не увеличивается.
3. Если счетчик расположений $b.R(r) > 1$, необходимо уменьшить его на единицу. Количество дополнительных ложноположительных ответов в данном случае не увеличивается.

При достижении предела количества дополнительных ложноположительных ответов $s.e \geq s.E$, сегмент необходимо построить заново в соответствии с алгоритмом построения индекса (см. раздел 2.2).

2.5 Результат внедрения

В рамках данной работы структура данных Сито-индекс была внедрена [34] в платформу для доступа к данным Apache Hudi.

В данном разделе представлены результаты сравнения скорости работы точечных и интервальных запросов с использованием упрощенных сводок и Сито-индекса для таблиц разного размера. Тестовый набор данных был

сгенерирован с использованием специализированной библиотеки примеров для больших данных TPC-H [35], запросы производились к таблице «lineitem» по атрибуту «shipdate».

В процессе тестирования индекс был построен для двух таблиц разного размера: первые 20 миллионов строк (первый набор данных) и первые 600 миллионов строк (второй набор данных) из таблицы «lineitem». Тестовый вычислительный кластер Spark состоял из одного узла, который был ограничен объемом оперативной памяти в 2 ГБ. Это позволило проверить возможность индексации таблицы из 20 миллионов записей с полной загрузкой всех значений атрибута в память и возможность построения индекса для таблицы из 600 миллионов записей, размер индексируемого атрибута такой таблицы не умещается в 2 ГБ оперативной памяти.

Размер индекса для первого набора данных составил 1,5 МБ при общем размере индексируемого атрибута в 80 МБ (20 миллионов значений по 4 Б), а для второго — 52 МБ при общем размере индексируемого атрибута 2,2 ГБ (600 миллионов значений по 4 Б) (Рисунок 1). При этом размер индекса составил примерно 6% от исходного размера индексируемого атрибута.

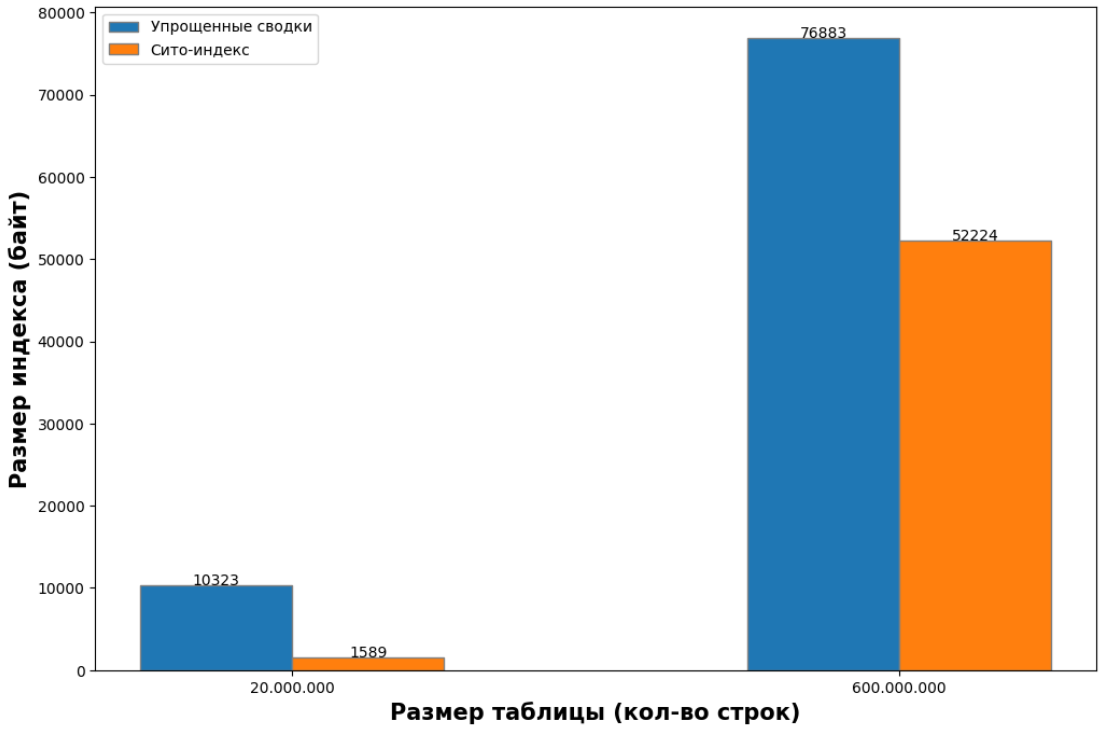

Рисунок 1 – Итоговые размера упрощенных сводок и Сито-индекса на файловой системе для двух таблиц разного размера

Причина, по которой итоговый размер упрощенных сводок получился больше размера Сито-индекса заключается в реализации упрощенных сводок в Apache Hudi [36], каждая запись содержит полное название файловой группы (Определение 3), в то время как авторская реализация Сито-индекса кодирует каждый идентификатор файловой группы используя ассоциативный массив, что позволяет сократить использование памяти.

Далее была проведены серии из 50 точечных запросов и 20 интервальных запросов для двух наборов данных, сперва они были проиндексированы упрощенными сводками, затем Сито-индексом. На рисунках ниже представлено сравнение средней скорости выполнения точечных (Рисунок 2) и интервальных (Рисунок 3) запросов. Разброс значений в сериях запросов составил не более 3%.

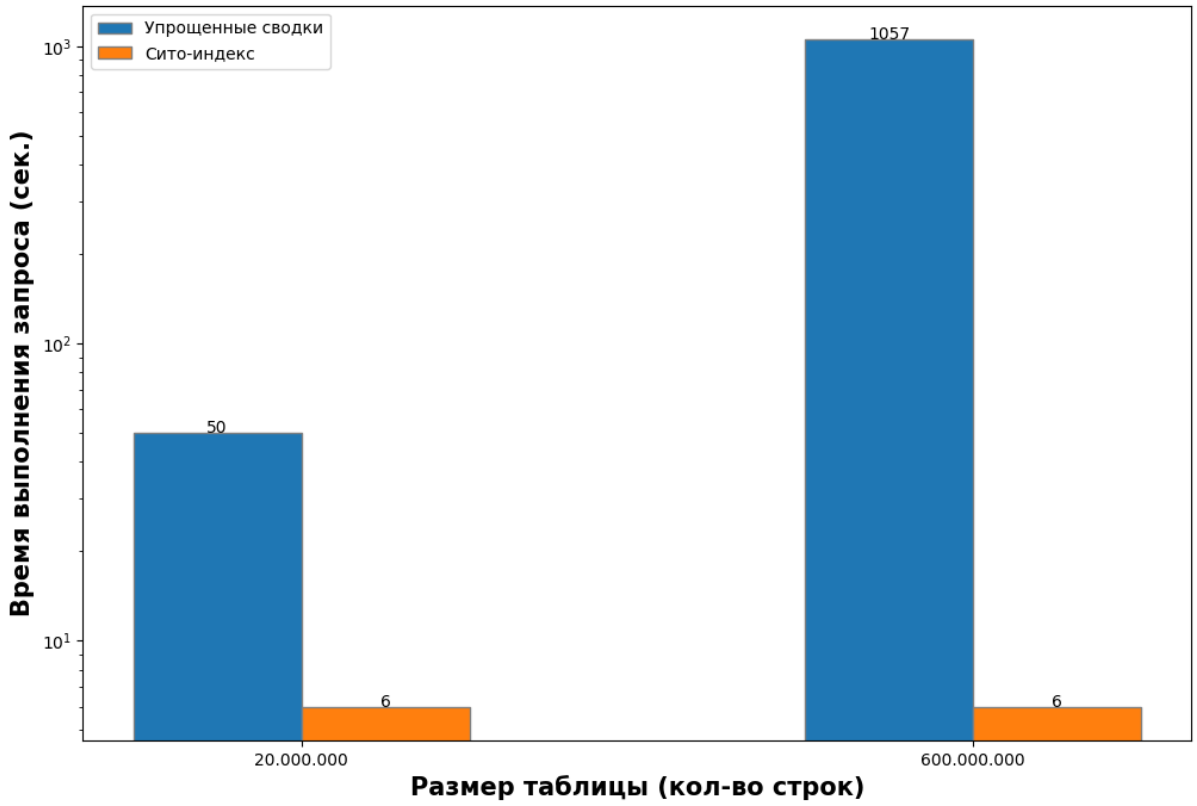

Рисунок 2 – Среднее время выполнения точечного запроса с использованием упрощенных сводок и Сито-индекса для двух таблиц разного размера

В наиболее выигрышном для Сито-индекса случае, почти каждый файл с данными в обеих наборах содержал минимальное значение даты — 1 января 1992 года, а максимальное — 1 января 2008 года, при этом почти каждый

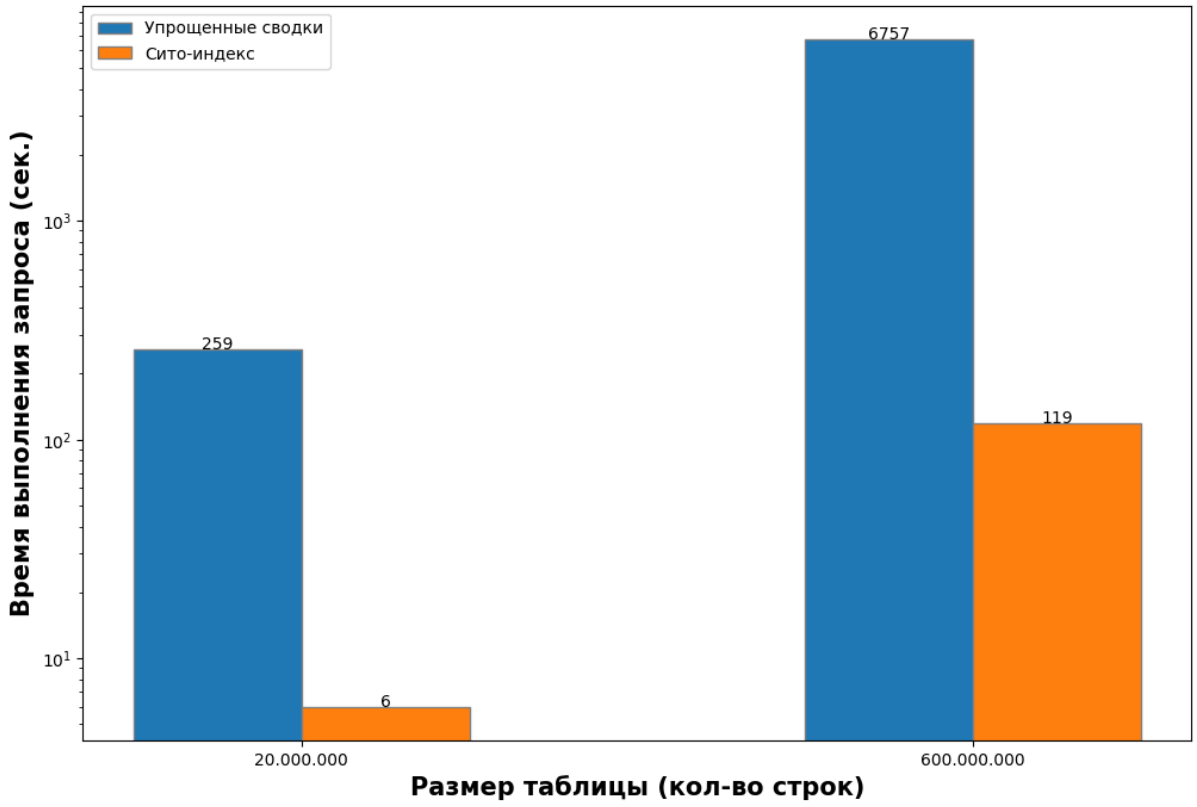

Рисунок 3 – Среднее время выполнения интервального запроса с использованием упрощенных сводок и Сито-индекса для двух таблиц разного размера

файл сдержал каждую дату из интервала с 1 января 1992 года по 1 января 1998 года и каждую дату из интервала с 1 января 2007 года по 1 января 2008 года, промежуток с 1999 года по 2006 год оставался пустым. Интервальные запросы были выполнены с предикатом 1 января 2003г. $\leq$ *shipdate* $\leq$ 31 января 2003г., а точечные с предикатом *shipdate* = 1 января 2003г. Тогда индекс с упрощенными сводками приводил к чтению всех файлов, а Сито-индекс пропускал чтение почти всех файлов.

В наименее выигрышном для Сито-индекса случае, когда выполнения запросов с предикатом подразумевало чтение всех файлов, время выполнения запроса было одинаковым для Сито-индекса и упрощенных сводок.

Построение Сито-индекса занимает больше времени ввиду большей вычислительной сложности алгоритма построения индекса. Однако является сопоставимым с построением упрощенных сводок, так как для построения упрощенных сводок необходимо прочитать весь индексируемый атрибут, в то время как для построения Сито-индекса его необходимо еще и отсортировать.

Далее представлено сравнение времени построения индекса для двух наборов данных (Рисунок 4).

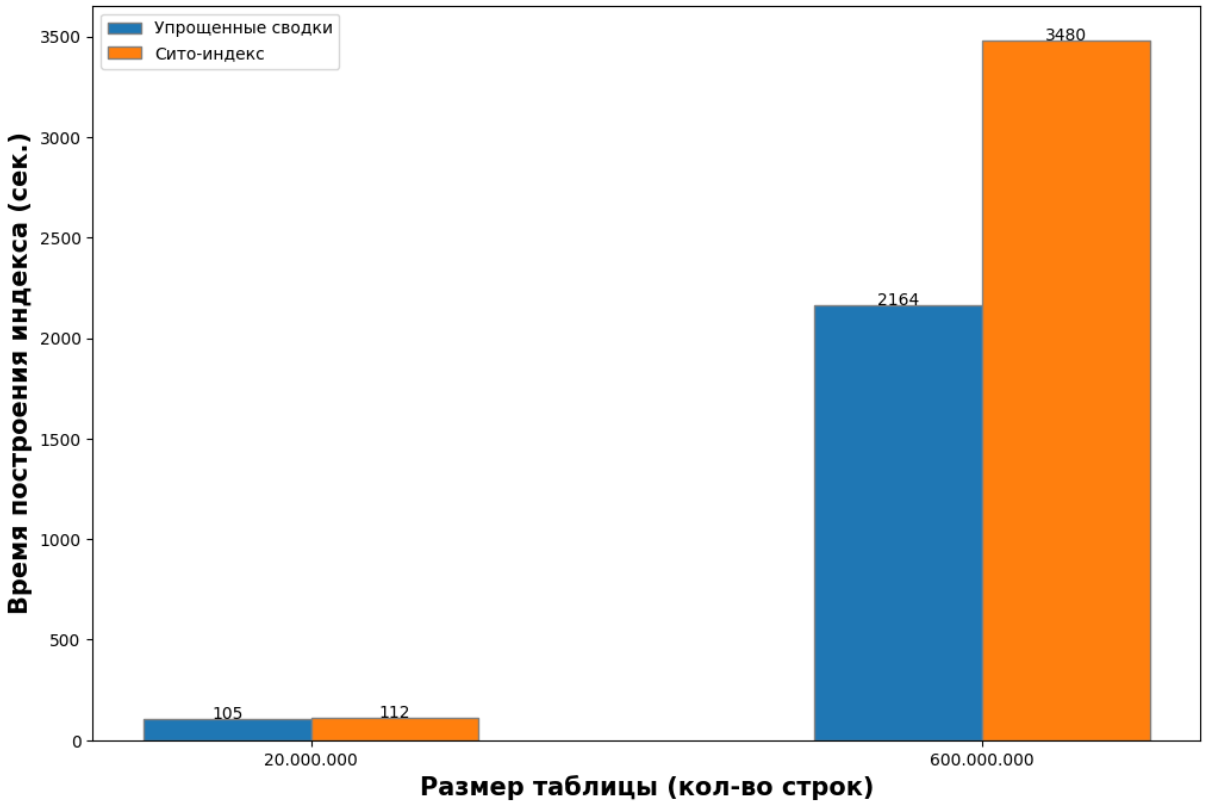

Рисунок 4 – Сравнение времени построения упрощенных сводок и Сито-индекса для двух таблицах разного размера

ВЫВОДЫ

Благодаря разработке структуры данных Сито-индекс, которая построена на основе структуры данных Sieve и является ее модернизацией, и внедрению ее в систему для доступа к данным с открытым исходным кодом Apache Hudi, удалось добиться прироста скорости выполнения точечных запросов на порядки, а интервальных запросов на порядок для тех сценариев использования, когда имеющиеся в Apache Hudi фильтры для отсеивания нерелевантных файлов упрощенные сводки были не способны отсеивать файлы.

В худших же сценариях для использования Сито-индекса время выполнения запросов с данной структурой было не больше времени выполнения тех же запросов с использованием упрощенных сводок.

ЗАКЛЮЧЕНИЕ

В рамках данной выпускной квалификационной работы были рассмотрены имеющиеся структуры данных в системах для доступа к данным, способствующие ускорению точечных и интервальных запросов. Результатом работы стала разработка и внедрение структуры данных Сито-индекс в существующую систему для доступа к данным с открытым исходным кодом. Данная структура данных позволила значительно повысить эффективность обработки запросов с условиями за счет отсеивания файлов, которые не содержат данных, которые удовлетворяют предикату точечного или интервального запроса.

С помощью разработанной структуры данных удалось решить поставленную задачу фильтрации нерелевантных файлов для запросов с предикатами. Эксперименты показали, что внедрение структуры данных Сито-индекс привело к приросту производительности точечных и интервальных запросов в системе для доступа данных Apache Hudi.

Выпускная квалификационная работа выполнена мной самостоятельно с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из 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. Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/releases/release-0.7.0/` (дата обращения: 01.05.2024).
7. El-Rewini H. Abd-El-Barr M. Advanced computer architecture and parallel processing. — John Wiley & Sons, 2005. — 272 p.
8. 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.
9. Соловьев А. И. Хранение и обработка больших данных // Тенденции развития науки и образования. — 2018. — № 37-6. — С. 47–51.
10. Abadi D. J. Boncz P. A. Harizopoulos S. Column-oriented database systems // Proceedings of the VLDB Endowment. — 2009. — Vol. 2, No. 2. — P. 1664–1665.
11. Ta-Shma P. et al. Extensible data skipping // 2020 IEEE International Conference on Big Data. — 2020. — P. 372–382.

12. Moerkotte G. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing // In VLDB. — 1998. — P. 476–487.
13. Jain P. et al. Analyzing and comparing lakehouse storage systems // In CIDR. — 2023. — P. 1–4.
14. Гарсиа-Молина Г. Ульман Д. Д. Уидом Д. Системы баз данных. Полный курс. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2003. — 1088 с.
15. Sivathanu M. et al. A Logic of File Systems // FAST. — 2005. — Vol. 5. — P. 1–15.
16. Timeline | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/timeline` (дата обращения: 01.05.2024).
17. Metadata Table | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/metadata` (дата обращения: 01.05.2024).
18. 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).
19. File Layouts | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/file_layouts` (дата обращения: 01.05.2024).
20. 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.
21. 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.

22. Chen J. et al. Big data challenge: a data management perspective // Frontiers of computer Science. — 2013. — Vol. 7. — P. 157–164.
23. 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.
24. Kaisler S. et al. Big data: Issues and challenges moving forward // 2013 46th Hawaii international conference on system sciences. — 2013. — P. 995–1004.
25. 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.
26. SQL DDL | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/sql_ddl` (дата обращения: 01.05.2024).
27. 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).
28. Bloom B. H. Space/time trade-offs in hash coding with allowable errors // Communications of the ACM. — 1970. — Vol. 13, No. 7. — P. 422–426.
29. 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.
30. Bonomi F. et al. An improved construction for counting bloom filters // Algorithms–ESA 2006: : 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14. – Springer Berlin Heidelberg. — 2006. — Vol. 5. — P. 684–695.
31. Cormode G. Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications // Journal of Algorithms. — 2005. — Vol. 55, No. 1. — P. 58–75.

32. Tong Y. et al. Sieve: A Learned Data-Skipping Index for Data Analytics // Proceedings of the VLDB Endowment. — 2023. — Vol. 16, No. 11. — P. 3214–3226.
33. Han S. et al. Impact of memory size on bigdata processing based on hadoop and spark // Proceedings of the international conference on research in adaptive and convergent systems. — 2017. — P. 275–280.
34. Sieve implementation // GitHub, Inc. — URL: `https://github.com/Alowator/hudi/commits/sieve/` (дата обращения: 20.05.2024).
35. Sieve implementation // TPC. — URL: `https://tpc.org/tpch/` (дата обращения: 01.05.2024).
36. HoodieMetadataPayload.java // GitHub, Inc. — URL: `https://github.com/apache/hudi/blob/release-0.14.0/hudi-common/src/main/java/org/apache/hudi/metadata/HoodieMetadataPayload.java` (дата обращения: 01.05.2024).

ПРИЛОЖЕНИЕ А

.

.

.

.

.

.

.

ПРИЛОЖЕНИЕ Б

.

.

.

.

.

.

.

.

.