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: 33 страниц
Количество иллюстраций: 0
Количество таблиц: 0
Количество литературных источников: 14
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 за январь по 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, 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 откуда инфа?.

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** [9].

Apache Hudi (далее — Hudi) организует данные в таблицы, реализуя тем самым реляционную модель [10]. 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 [11].

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) [12].

В системах для доступа к данным существует две стратегии обновления данных, которые определяют внутреннюю структуру таблицы [9], 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 [9].

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. [9].

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 [13].

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 [14].

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 [14]. 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, идентифи-

катор 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

Далее рассмотрим основные понятия, связанные с индексами. 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. Для базы данных, содержащей N элементов, это потребовало бы времени порядка O(N), что для современных баз данных является неэффективным.

Индекс — это структура данных, которая позволяет ускорить процесс поиска за счёт использования дополнительного пространства и выполнения дополнительных операций записи для поддержания своей структуры. Существует множество типов индексов, предназначенных для работы с различными типами данных, такими как пространственные, временные, текстовые, многомерные и другие. Выбор подходящего индекса для конкретной задачи является ключевым аспектом процесса оптимизации, поскольку это может значительно влиять на временную сложность поиска, которая варьируется от O(logN) до O(1).

Учитывая изложенное, можно заключить, что задача определения расположения файла данных по заданному атрибуту с использованием индекса сводится к нахождению файловой группы, в которой расположена данная запись. В дальнейшем, для целей данной работы, термин «файловая группа» будет интерпретирован как «местоположение файла данных». При этом выбор конкретной версии файла данных в файловой группе зависит

от типа запроса, включая исторические запросы. В рамках данной работы индекс будет отображать некоторые значения атрибутов на идентификатор файловой группы (расположение файла данных), в которой расположены соответствующие записи.

Учитывая специфику предметной области, необходимо разработать ряд требований к реализуемой структуре. Эти требования послужат основой для оценки анализируемых структур и определения необходимых доработок перед внедрением в платформу.

Ключевым аспектом является определение точных характеристик метода индексации, что имеет критическое значение для формулирования требований к индексации больших объёмов данных. В контексте больших данных для сравнения методов индексации обычно рассматриваются три основные характеристики, приводящие к сложностям в работе с данными [7]:

Объём данных: Управление и индексация больших объёмов данных представляют собой наиболее очевидную проблему в данной предметной области [8]. Современные объёмы аналитических данных в облачных сервисах, обрабатываемые за приемлемое время, измеряются терабайтами и, как ожидается, в ближайшем будущем достигнут петабайт [9].

Скорость изменения: Данные постоянно изменяются и обновляются. Обработка данных в реальном времени, известная как «потоковая обработка», становится альтернативой традиционной пакетной обработке [8]. Секторы, такие как электронная коммерция, генерируют значительные объёмы данных, например, через веб-клики, которые непрерывно поступают в поток [10].

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 [11].

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 некоторых 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.

Изменчивость данных влечёт за собой требования к обновляемости индекса, включая поддержку операций обновления, добавления и удаления записей. Эти операции должны выполняться эффективно в контексте работы с большими данными, что не всегда возможно для каждого типа индекса в рамках платформ озер данных. Отдельно следует упомянуть задачу индексации уже существующих наборов данных: при больших объёмах данных выполнение этой задачи за разумное время не всегда осуществимо для каждой структуры данных.

В предметной области, связанной с транзакционными платформами озер данных, обычно не приходится сталкиваться с задачами, связанными с разнообразием данных, так как данные хранятся в строгом формате, определённом схемой таблицы. Данные разделяются на три уровня: бронзовый, серебряный и золотой. Индексируемые данные в системе Hudi относятся к золотому уровню [12].

Реализуемая структура данных должна обеспечивать эффективность выполнения точечных и интервальных запросов по одному атрибуту. Важно подчеркнуть, что типичный сценарий использования платформ для анализа

данных заключается в интервальных запросах, поэтому оценка эффективности должна проводиться с учетом выполнения именно таких запросов.

1.4 Анализ существующих решений

1.3.1. Точные индексы

Одним из наиболее распространённых методов в системах управления базами данных (СУБД) является создание индексов, отображающих значение атрибута каждой отдельной записи на её расположение в таблице. Такой подход приводит к формированию семейства индексов на уровне записей, где каждая запись в индексе соответствует записи в таблице, требуя выделения O(N) памяти [3]. Обычно для этих целей используются B-деревья.

Основная проблема данного подхода заключается в том, что объём индекса становится соразмерным с объёмом данных. Согласно документации Hudi, размер одной записи в точном индексе составляет приблизительно 50 байт [2]. Для оценки размера индекса относительно размера данных предположим, что размер записи составляет 100 КБ. В таком случае для таблицы объёмом 100 ТБ размер индекса достигнет 50 ГБ, что делает его непригодным для полной загрузки в оперативную память. При этом размер индекса линейно зависит от объёма данных. Данная оценка учитывает, что данные хранятся в сжатом виде, а метаданные — без избыточности.

Таким образом, использование подобных индексов ограничено данными небольшого размера по сравнению с типичными сценариями применения на платформах для работы с большими данными. Более того, принятый в расчётах размер записи в 100 КБ не является минимально возможным. В зависимости от кодировки и метода сжатия, каждая запись в такой таблице может содержать примерно 104 символов, что характерно для аналитических систем. Схема такой таблицы должна включать порядка 102 атрибутов или содержать большой объём текста при меньшем количестве атрибутов. Однако, если таблица содержит десятки атрибутов и средний размер значения атрибута в байтах не превышает сотни, то размер индекса становится сопоставим с размером всего набора данных. При фиксированном размере

индексной записи в 50 байт и размере записи в 100 байт, использование такого индекса становится нецелесообразным уже при размере таблицы в сотни гигабайт, что является типичным для классических СУБД, но не для озер данных.

Из анализа индексов, в которых количество записей соответствует количеству записей в таблице, следует важный вывод: хранение точного расположения каждой записи неосуществимо в реальных условиях использования платформ озёр данных. Для преодоления этой проблемы могут быть применены вероятностные структуры данных, рассмотрение которых представлено далее.

1.3.2. Фильтры Блума

Одним из наиболее распространенных вероятностных подходов к отсеиванию файлов, не содержащих данных, удовлетворяющих заданному предикату, является применение структуры данных, известной как «фильтр Блума» [4].

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.

Фильтр Блума реализует вероятностное множество с двумя основными операциями: добавлением элемента в множество и проверкой принадлежности элемента множеству. Для добавления элемента e необходимо установить в единицу биты на позициях h1(e), . . . , hk(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". Если хотя бы один бит не установлен в единицу, это означает, что элемент не принадлежит множеству — ответ "нет"[5].

Существует множество фильтров, основанных на концепции фильтра Блума, каждый из которых имеет свои преимущества и недостатки, напри-

мер, кукушечный фильтр [6]. Тем не менее, общая идея, а также основные преимущества и недостатки остаются неизменными.

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. Несмотря на простоту, фильтр Блума остаётся очень компактной структурой данных. Его размер значительно меньше индексируемого набора данных. Однако, для поддержания приемлемого уровня ошибок необходимо увеличивать размер битового массива при увеличении объема данных. В целом, фильтр Блума представляет собой достаточно компактную структуру данных, которая демонстрирует заметную экономию пространства.

Основным недостатком фильтра Блума в контексте отсеивания избыточных файлов данных 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. Кроме того, стандартная реализация фильтра Блума не предполагает возможность удаления элемента из индекса, поскольку это может привести к ложноотрицательным результатам, что недопустимо в данной задаче.

Однако наиболее значимым недостатком фильтра Блума является его неспособность обрабатывать интервальные запросы, которые часто используются в аналитических системах. Фильтр Блума эффективно отвечает на запросы о принадлежности одного ключа множеству, но при необходимости проверить принадлежность каждого ключа из заданного интервала, временная сложность операции составляет $O(N \times M)$, где N — количество проверяемых фильтров, а M — количество ключей в интервале. Учитывая, что размер множества ключей может быть неограничен для неограниченных интервальных запросов, это делает фильтр Блума непригодным для эффективной обработки таких запросов.

Таким образом, несмотря на то что структуры данных, основанные на фильтре Блума, широко используются в задачах, связанных с большими данными, они не подходят для решения задач отсеивания избыточных файлов данных из-за необходимости чтения всей структуры данных и невозможности обработки интервальных запросов, которые являются наиболее частыми в аналитических системах.

1.3.3. Блочные индексы

Платформы для работы с большими данными применяют распределённые файловые системы, что превращает операции ввода-вывода с удалённого хранилища в одни из наиболее ресурсоёмких аспектов обработки запросов. Эти системы организуют записи данных в блоки, каждый из которых может содержать миллионы записей, что способствует максимизации степени сжатия данных. Для оптимизации пропускной способности и минимизации количества операций ввода-вывода в секунду, минимальной единицей ввода-вывода из удалённого хранилища является блок или подмножество столбцов из данного блока.

Для повышения эффективности обработки запросов облачные сервисы анализа данных обычно применяют легковесные агрегации [14] для каждого блока, чтобы избежать необязательного доступа к нерелевантным блокам данных в процессе обработки запроса. Одной из наиболее распространённых форм легковесных агрегаций является ZoneMap, которая включает в себя хранение минимальных и максимальных значений, а также количества нулевых записей для каждого столбца в блоке данных. Например, если в блоке данных ZoneMap указано, что диапазон дат охватывает период с апреля по май, и запрос ищет записи за февраль, то данный блок может быть исключён из процесса чтения данных из хранилища, так как он не содержит необходимых данных.

ZoneMap — 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 как O(1). Однако их эффективность зависит от конкретного расположения данных внутри блока.

В действительности ZoneMap оказывается эффективным для упорядоченных атрибутов, таких как целочисленные первичные ключи, однако такие атрибуты не являются ключевыми в сценариях использования больших данных для интервальных запросов. Эти атрибуты необходимы для корректной операции вставки данных в соответствующий файл данных для поддержки исторических запросов, но этот аспект выходит за рамки данной работы. Стоит отметить, что для управления такими операциями обычно подходят точные структуры данных, так как ложноположительные ответы могут нивелировать преимущества небольшого размера структур. Это связано с тем, что операция вставки обновлённой версии существующей записи является точечной операцией. Для неупорядоченных атрибутов, которые являются более типичным сценарием использования, такие структуры неэффективны, поскольку диапазоны значений в блоках могут охватывать большую часть предикатов запроса, что приводит к необходимости полного сканирования множества нерелевантных блоков.

Несмотря на ограничения легковесных агрегаций, этот подход избавлен от недостатков точных индексов и фильтров Блума. Важно заключить, что в контексте больших данных такие структуры могут оказаться неэффективными, однако они применимы для таблиц большого размера в традиционных СУБД, где их использование напоминает блочные индексы [15].

В то же время данный подход может оказаться полезным, если его адаптировать не к физическим файлам данных, а к общему множеству ключей. Эффективно разделить все упорядоченное множество ключей на непересекающиеся блоки, где каждый блок содержит информацию о минимальном и максимальном значениях, а необходимые данные для пропуска нерелевантных блоков определяются расположением файлов данных, ассоциированных с данным множеством ключей. Такой подход, хоть и сохраняет легковесность, имеет размер индекса, который растет линейно относительно количества записей в индексируемом наборе данных.

Существуют также эвристические подходы, которые используют закономерности в данных. Например, подход Sieve основан на наблюдении, что близкие по значению ключи часто находятся в одном и том же наборе данных, или распределены среди различных файлов данных [1]. Sieve использует кумулятивную распределительную функцию для определения размера следующего индексного блока: если значения функции для ключей k и k-1 указывают на одинаковые множества расположений, значение функции равно 0; в противном случае — 1. Sieve группирует множество ключей, относящихся к одному и тому же набору блоков, в отдельный сегмент. Эта идея схожа с принципом легковесных агрегаций, однако предлагает большую гибкость в построении индексов.

На данном этапе возможно синтезировать структуру данных, объединяющую преимущества различных подходов, с учетом существующих ограничений предметной области. Это позволит описать структуру, эффективно отсеивающую ненужные файлы данных и ускоряющую выполнение запросов.

Целесообразно использовать подход Sieve для выявления трендов в распределении ключей между файлами данных. К каждому полученному сегменту применяется классический метод блочных индексов. Каждый сегмент делится на блоки равной длины, содержащие информацию о расположении каждого ключа в интервале [l; r].

Для быстрого нахождения необходимого множества блоков отмечается, что при выполнении интервальных запросов в такой структуре требуется последовательное считывание расположенных подряд блоков. Для решения этой задачи в классических СУБД обычно используются B-деревья и их модификации. Оптимальная скорость сканирования в B-деревьях достигается за счет использования структуры в виде плоского дерева, где каждый узел представляет собой сегмент. Учитывая, что размер каждого блока внутри сегмента фиксирован, точечный запрос в такой структуре можно выполнить за время O(logN), а затем найти нужный блок можно за константное время. Для интервальных запросов требуется последовательное чтение следующих

блоков. Такой подход к индексации данных будем называть "Сито а реализуемую структуру данных — "Сито-индексом".

ВЫВОДЫ

Каждый подход к созданию индексных структур обладает определенными преимуществами и недостатками. Однако в условиях обработки больших объемов данных возможно эффективно сочетать наилучшие характеристики каждой системы для формирования оптимальной легковесной структуры. Такая структура должна быть не только эффективной с точки зрения использования памяти, но и обеспечивать приемлемый уровень точности ответов на реальных данных с минимальным количеством ложноположительных результатов.

Синтезированная структура, названная Сито-индексом, объединяет подходы, описанные в параграфах 1.3.1 и 1.3.3, и представляет собой вероятностную структуру данных. Она использует кумулятивную распределительную функцию для определения размера индексных блоков, применяет метод блочных индексов для группировки ключей и образует структуру B-дерева, что облегчает поиск и ускоряет выполнение запросов.

Сито-индекс обеспечивает гибкость при работе с большими объемами данных, путем подбора оптимального размера блока для разных участков данных и может быть оптимизирован для различных сценариев использования в зависимости от специфических требований.

2 ВНЕДРЕНИЕ СИТО-ИНДЕКСА В СИСТЕМУ ДОСТУПА К ДАННЫМ

Теоретическое определение Сито-индекса разработано с учетом всех ограничений предметной области, однако для его внедрения в конкретную платформу управления данными, такую как Apache Hudi, необходимо детально формализовать все операции по обновлению индекса и специфицировать алгоритм его построения в распределенной среде.

2.1 Построение индекса

Задача построения индекса формулируется следующим образом: для заданного индексируемого атрибута требуется создать и реализовать персистентное хранение индекса в таблице. Важно отметить, что в общем случае индексируемый атрибут не умещается в оперативной памяти вычислительного узла.

При построении индекса работаем с отношением, содержащим два атрибута: индексируемый атрибут (ключ) и расположение файла данных, содержащего данный ключ. Уникальность ключей в общем случае не гарантируется, что делает необходимым сортировку данного набора для построения кумулятивной распределительной функции.

Процесс построения индекса различается для новых и существующих таблиц. Для новой таблицы построение индекса начинается с декларации его использования, после чего происходит обновление индекса согласно пункту 2.4.

Для существующих таблиц необходимо, чтобы набор ключей был отсортирован, причем уникальность каждого ключа не гарантируется. Простейшее решение заключается в применении любого метода сортировки, при этом стабильность сортировки не является обязательной. Сложность задачи построения индекса для существующей таблицы возрастает из-за необходимости сортировать и агрегировать большой объем данных для определения расположения каждого ключа. Проблема вытекает из необходимости созда-

ния упорядоченного массива ключей, на основе которого будет сформирована кумулятивная распределительная функция.

Построение индекса для существующей таблицы представляет собой более общий случай. Основная задача заключается в формировании упорядоченного массива ключей для создания кумулятивной распределительной функции.

Для решения этой задачи применим возможности распределенного движка Spark, используемого в Hudi. С его помощью осуществляется сортировка ключей и агрегация, что позволяет получить набор расположений для каждого уникального ключа. Этот метод является оптимальным, так как задача сортировки большого объема данных, который не умещается в оперативной или даже постоянной памяти компьютера, имеет известные решения, но остается нетривиальной.

После сортировки ключей достаточно в однопоточном режиме пройтись по каждому из них, поддерживая текущее кумулятивное значение функции для определения момента изменения тренда данных и необходимости формирования нового сегмента.

Существует вероятность, что весь набор данных будет иметь общий тренд, и с точки зрения подхода Sieve целесообразно создать один большой сегмент. Однако это невозможно из-за ограничений оперативной памяти вычислительного узла, что требует прекращения генерации сегмента при достижении его определенного размера. Экспериментально было установлено, что размер сегмента не должен превышать 3 миллиона записей при выделенных 2 ГБ оперативной памяти на вычислительный узел. Далее следует определить оптимальный размер блока, при котором вероятность ошибки окажется ниже установленного порога. Каждый сегмент делится на блоки, в которых сохраняется информация о местоположениях, ассоциированных с данным интервалом ключей. После завершения этих операций сегмент сериализуется на файловую систему.

Таким образом, благодаря возможностям распределенного движка и особенностям однопроходного алгоритма сегментации данных, можно эф-

фективно построить Сито-индекс для объемного набора данных, который не умещается в память компьютера.

2.2 Поиск в индексе

Задача поиска нужных блоков в Сито-индексе для отфильтровывания нерелевантных данных аналогична поиску блоков данных в B-дереве. Основное различие между обработкой интервального запроса и точечным запросом заключается в следующем: для точечного запроса требуется найти и прочитать один блок, тогда как для интервального запроса необходимо определить один начальный блок, содержащий ключ из предиката. Например, для предиката a>5 необходимо найти блок, содержащий ключ 5, и последовательно прочитать все блоки, в которых находятся ключи с большими значениями. Эта операция эффективна, поскольку блоки расположены последовательно, что исключает необходимость поиска отдельных блоков в структуре индекса.

Сегменты индекса маркируются минимальным и максимальным значениями ключей, содержащихся в каждом сегменте, что позволяет определить соответствующий сегмент для заданного предиката за время O(logN) с использованием алгоритма бинарного поиска.

Номер соответствующего блока внутри сегмента может быть вычислен по формуле $(key - min_key)block_size$, где деление выполняется как целочисленное, key это искомый ключ, minkey минимальный ключ в сегменте, a blocksize известный размер блока для текущего сегмента.

2.3 Обновление индекса

Существуют три типа операций обновления индекса: добавление записи, ключевой атрибут которой отсутствует в индексе; удаление записи, ключевой атрибут которой присутствует в индексе; и обновление существующей записи, когда ключевой атрибут уже занесен в индекс. Кроме того, осуществляется операция удаления записи из индекса.

Для эффективного обновления индекса целесообразно использовать подход Merge-On-Read, поскольку обновление сегмента при каждой операции вставки является ресурсоемким.

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, местоположение 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.

картинки-графики

ВЫВОДЫ

Следует подчеркнуть потенциал внедрения структуры данных Сито-индекс в современные платформы озер данных, такие как Apache Hudi, Iceberg и DeltaLake. Принципы работы с данными в них имеют много общего, что делает возможным применение Сито-индекса также и в других платформах.

Особенностью Сито-индекса является его легковесность, позволяющая полностью загружать его в оперативную память. Это становится возможным благодаря тому, что размер индекса составляет всего десятые доли процента от общего объема данных. Такая эффективность открывает перспективы использования Сито-индекса не только в платформах озер данных, но и в

SQL-движках, предназначенных для обработки больших данных, например, в Apache Presto.

Индексация данных в рамках платформы Apache Hudi показала, что даже при работе с большими объемами данных (до 600 миллионов строк) возможно эффективное построение индекса, что подтверждает его применимость в условиях ограниченной памяти.

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 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. Jain P. et al. Analyzing and comparing lakehouse storage systems // In CIDR. — 2023. — P. 1–4.
10. Гарсиа-Молина Г. Ульман Д. Д. Уидом Д. Системы баз данных. Полный курс. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2003. — 1088 с.
11. Timeline | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/timeline` (дата обращения: 01.05.2024).
12. Metadata Table | Apache Hudi // The Apache Software Foundation. — URL: `https://hudi.apache.org/docs/0.14.0/metadata` (дата обращения: 01.05.2024).

13. 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).
14. File Layouts | Apache Hudi // The Apache Software Foundation. — URL: https://hudi.apache.org/docs/0.14.0/file_layouts (дата обращения: 01.05.2024).