Санкт-Петербургский государственный университет

Вильданов Эмир Флоридович

Выпускная квалификационная работа

Оптимизация распределённых JOIN-запросов в кластере Tarantool

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

Научный руководитель:
Доцент кафедры системного программирования, к.ф.-м.н., Д. В. Луцив

Консультант:
Ведущий программист ООО «Пикодата», Д. А. Смирнов

Рецензент:
Начальник отдела ООО «Пикодата», Д. Д. Кольцов

Санкт-Петербург
2023

Saint Petersburg State University

Emir Vildanov

Bachelor's Thesis

Optimization of distributed JOIN-queries in Tarantool cluster

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
Docent, C.Sc., D.V. Luciv

Consultant:
Senior developer Picodata LLC, D. A. Smirnov

Reviewer:
Team lead Picodata LLC, D. D. Koltsov

Saint Petersburg
2023

Оглавление

Введение

Распределённые базы данных — это популярная и востребованная область, поскольку в условиях дороговизны улучшения одной рабочей станции горизонтальное масштабирование является более выгодным вариантом, который в свою очередь повышает сохранность и доступность данных. Однако за повышенную гибкость подобных систем и их удобную функциональность приходится расплачиваться усложнением процесса разработки: из-за того, что данные оказываются распределены по нескольким вычислительным узлам, приходится иметь дело с перемещением данных по сети. Так JOIN-запрос в контексте распределённых баз данных — одна из наиболее трудоёмких операций. Поскольку эта операция в неоптимизированном виде представляет собой декартово произведение записей из двух таблиц, которые могут находиться на разных машинах, для выполнения JOIN-запроса необходимо производить перемещение этих записей с одного вычислительного узла на другой.

Проект Sbroad (SQL Broadcaster), который сейчас разрабатывается компанией Пикодата, представляет собой библиотеку распределённого SQL над кластером in-memory базы данных Тарантул. Он включает в себя парсер SQL, оптимизатор и исполнитель запросов. Проект сейчас находится на этапе активной разработки, поэтому JOIN-запросы в существующей архитектуре оптимизированы лишь в некоторой степени. Предполагается, что пользователь сам должен писать такие JOIN запросы, которые будут выполняться эффективно из того предположения, что правая таблица в JOIN-запросе всегда меньше левой. План исполнения распределённого запроса сейчас строится только на основании информации о том, по каким ключам производятся операции соединения: по колонкам распределения таблиц или по произвольным колонкам.

Подобно оптимизациям, основанным на логике реляционной алгебры, существует серия оценочных (Cost-Based) оптимизаций, также направленных на трансформацию SQL-запроса, но опирающихся на таб-

личную статистику или на эмпирические наблюдения. Cost-Based оптимизация направлена на построение такого плана исполнения SQL-запроса, который бы имел минимальную оценочную стоимость, определяемую, например, размером пересылаемых по сети данных.

Главным объектом данной работы является модуль Cost-Based оптимизации, опирающийся на табличную статистику и применяющий алгоритм переупорядочивания бинарных и N-арных JOIN-соединений в распределённых SQL-запросах. В работе приведён обзор существующих алгоритмов оценочной оптимизации и описана реализация модуля оценочной оптимизации, добавленного в библиотеку Sbroad.

Дата сборки: 26 мая 2023 г.

1. Постановка задачи

Целью данной работы является реализация модуля оценочной оптимизации JOIN-запросов в библиотеке распределённых SQL-запросов Sbroad. Для её достижения были поставлены следующие задачи.

1. Провести обзор существующих алгоритмов оценочной оптимизации, представленных в научных статьях и базах данных с открытым исходным кодом, среди которых выбрать ориентиры для заимствования логики.

2. Создать архитектуру дополнительного модуля оценочной оптимизации в библиотеке Sbroad.

3. В соответствии с созданной архитектурой реализовать модуль оптимизации на языке программирования Rust.

4. Провести эксперименты, оценивающие качество работы реализации.

2. Обзор

Задача Cost-Based оптимизации распределённых SQL-запросов и в частности JOIN-запросов — это довольно старая задача, и данная работа не ставит себе целью разработку уникальных алгоритмов оптимизации. Предположение, на котором основывается данная работа заключается в том, что необходимо лишь найти ориентиры для разработки: будь то алгоритмы, представленные в научных статьях, или готовые модули оценочной оптимизации, реализованные в базах данных с открытым исходным кодом. В главе **Особенности реализации** показано каким образом выбранные в качестве ориентиров алгоритмы оптимизации были адаптированы под архитектуру библиотеки Sbroad.

2.1. Алгоритмы переупорядочивания JOIN-операторов

Рассмотрим на примере запроса, представленного на листинге 1, какие вообще есть возможности для оптимизации при условии, что мы обладаем табличной статистикой.

Листинг 1: Пример JOIN-запроса

```
SELECT * FROM
(A LEFT JOIN B ON A.a = B.b INNER JOIN
(C INNER JOIN D ON C.c > D.d WHERE C.e ≤ 69))
RIGHT JOIN E ON B.j = E.h
WHERE E.i ≠ 42 OR E.k IN (1, 2, 3)
```

Первым делом заметим, что наш запрос содержит четыре JOIN-оператора. Одна из самых сложных [11] задач оптимизации JOIN-запроса — это определение порядка, в котором эти операторы необходимо исполнить. В зависимости от размеров таблиц и применяемых условий фильтрации (WHERE и ON), разные последовательности исполнения операторов приведут к разному количеству записей (*кардинальности*) в промежуточных соединениях. Чем меньше строк полу-

чается в промежуточных соединениях, тем меньше строк приходится пересылать по сети при исполнении последующих (родительских) операторов и тем меньше общее количество пересылаемых по сети данных.

Существует большое количество алгоритмов упорядочивания JOIN-операторов, с которыми можно ознакомиться в научной литературе [15, 1, 4]. Рассмотрим три крупные категории подобных алгоритмов.

- Детерминированные. Представляют собой алгоритмы, перебирающие тем или иным способом все возможные варианты исполнения сложного JOIN-запроса. Они отличаются высокой точностью и большим временем работы. Иногда поиск **самой** оптимальной последовательности JOIN-операторов может занимать значительную часть времени от исполнения самого запроса;

- Рандомизированные. Данные алгоритмы при оптимизации последовательности исполнения применяют набор правил, случайным образом преобразующих так называемое пространство решений. Зачастую условие остановки данных алгоритмов — отсутствие улучшения результата после применения шагов-правил определённого количества раз;

- Генетические. Данные алгоритмы работают с последовательностью JOIN-операторов в строковом представлении (где позиции операторов в строке определяют порядок их исполнения) при помощи генетических алгоритмов. Считается, что данные алгоритмы зачастую являются оптимальным решением, если требуется небольшое время работы оптимизации и более высокая (по сравнению с рандомизированными алгоритмами) точность.

В случае, когда необходимо оптимизировать запрос, включающий в себя разные типы JOIN-операторов (такие как INNER, OUTER, SEMI и ANTI) детерминированные алгоритмы показывают себя лучше всего, поскольку и у рандомизированных, и у генетических алгоритмов возникают трудности с определением корректности сгенерированной последовательности операторов относительно исходного SQL-запроса: не

все JOIN-операторы можно переставлять между собой с сохранением итогового результата исполнения запроса. По этой причине в модуле оптимизации, описанном в данной работе, используется именно детерминированный алгоритм переупорядочивания.

Для удобства последующего изложения определим здесь несколько важных понятий. *План исполнения* SQL-запроса (далее просто *план*) — это дерево реляционных операторов (например, Projection, Scan, Selection, Join), исполнение которых приводит к получению результата SQL-запроса. *Стоимость* плана — это функция, определяющая насколько оптимальным является конкретный план. В данной работе за стоимость плана принимается размер всех пересылаемых по сети данных. Предполагается, что в условиях распределённой среды размер пересылаемых данных оказывает наибольшее влияние на скорость исполнения запроса.

Построение плана с наименьшей стоимостью — это ключевая задача модуля оценочной оптимизации.

2.2. Селективность

Для понимания того, каким образом можно определить размеры таблиц, получающихся в процессе исполнения JOIN-операторов при наличии условий соединения и фильтрации, необходимо обратиться к такому понятию, как селективность операторов.

Селективность оператора с применённым условием (предикатом) — это отношение количества возвращаемых этим оператором записей к количеству записей, вернувшихся бы без применения дополнительных условий. Рассмотрим два вида селективности, полезных при оптимизации запроса, содержащего JOIN-операторы.

- Селективность SELECT-оператора (фильтрации). Это селективность запроса, определяющая, какую часть строк из таблицы вернёт запрос с использованием ключевого слова WHERE;
- Селективность JOIN-оператора (соединения). Это селективность,

определяющая отношение строк, вернувшихся при исполнении JOIN-запроса с ключом соединения между двумя таблицами к количеству строк, которое бы вернуло декартово произведение строк из этих двух таблиц.

Информация о селективности позволяет, например, определить размер соединения между таблицами C и D в запросе из листинга 1: условие WHERE даёт понять, что операция соединения будет происходить лишь с частью таблицы C, а условие соединения ON позволяет понять, что вернётся лишь часть строк из общего декартова произведения между таблицами. Маленькое значение селективности сообщит нам о том, что фильтрация и соединение возвращают лишь малую часть от запроса без этих ограничений.

2.3. Структуры данных

Можно заметить, что для определения селективности операторов необходимо оперировать статистикой по конкретным колонкам: только с её помощью мы сможем понять, какая часть записей из таблицы удовлетворяет конкретному условию фильтрации. Для хранения подобной статистики используют специальные структуры данных, которые хранят в себе информацию о распределении колоночных данных. С используемыми структурами данных (как и с алгоритмами их анализа) можно ознакомиться в статье-обзоре [10]. Здесь рассмотрим те из них, о которых наиболее часто идёт речь в научной литературе и которые встречаются в реализациях популярных баз данных (краткое сравнение представлено на рис. 1).

Вероятностные структуры данных (Sketches). Это легковесные структуры данных, которые хранят статистику по колонкам, опираясь на законы теории вероятностей. По сравнению с другими структурами данных, они требуют мало оперативной памяти, поддерживают возможность обновления прямо при исполнении запросов и с приемлемо маленькой погрешностью выдают информацию о табличной статистике. Более подробно с ними можно ознакомиться на веб-ресурсе [13] или

в научной литературе [8]. Среди подобных структур можно выделить наиболее популярные:

- HyperLogLog, который используется для определения количества различных (DISTINCT) элементов в колонке;
- ThetaSketch, который представляет собой оптимизацию HyperLogLog с поддержкой обеих операций пересечения и объединения с помощью операторов AND и OR;
- CountMinSketch, который применяют для определения частоты элементов в колонке.

Значительным минусом с точки зрения применимости подобных структур данных является сложность поддержания их в актуальном состоянии: для каждой колонки либо приходится хранить избыточную информацию большого размера, либо производить дорогостоящую по времени операцию чтения из базы данных для обновления статистики. Также подобные структуры рассчитаны только на подсчёт селективности фильтрации и соединения по равенству, в то время как удобного и точного метода работы с операторами неравенств они не предоставляют. Стоит заметить, что, например, в одной из оптимизаций CountMinSketch от Apache [2] реализована поддержка запросов по диапазонам, но упоминаний об её в базах данных с открытым исходным кодом найдено не было.

Сэмпилрование (Выборка/Sampling). Представляет собой методику выборки определённого набора записей из таблицы с целью уменьшения времени, затрачиваемого на чтение этих записей. При выполнении запроса чтение только части таблицы для определения селективности по фильтрам занимает значительно меньше времени чем чтение таблицы целиком. Структурой данных в данном случае является сэмп — часть таблицы, получающаяся после завершения алгоритма.

Главным преимуществом этого подхода является возможность обработки сложных запросов, включающих в себя несколько условий фильтрации. По сравнению, например, с вероятностными структурами дан-

ных, погрешность которых возрастает с количеством фильтров по разным колонкам, данная структура данных даёт более точные результаты за счёт того, что хранит информацию о всех колонках сразу. При определении селективности отпадает необходимость выявления функциональных зависимостей между колонками.

Из минусов данного подхода можно выделить необходимость выбора продвинутого алгоритма выборки, который бы сохранял в сэмпле распределение данных и выбирал элементы пропорционально их частоте появления в таблице (зачастую такие алгоритмы за счёт своей недетерминированности могут потреблять значительное количество памяти). Так же, как и вероятностные структуры данных, данный подход не решает проблему запросов по диапазонам.

Гистограммы (Histograms). Это структуры данных, которые представляют колоночную статистику в виде диапазонов значений, называемых бакетами (корзины/buckets/bins). Значение, хранящееся в этих бакетах — это количество попавших в них элементов (поэтому иногда данную структуру данных называют ещё частотным распределением). От того, каким образом выглядят данные бакеты, определяется вид гистограммы. Отметим некоторые из них.

- Equi-width. Это один из самых простых видов гистограмм, диапазон бакетов (диапазон значений, которые в этот бакет будут попадать) которой одинаков и равен заранее заданному значению. Данный вид гистограмм отличается простотой реализации, однако страдает от низкой точности вследствии того, что не учитывает наиболее частые элементы;

- Equi-height (equi-depth). Это гистограммы, у которых фиксированным значением вместо размера диапазона бакета является значение самого бакета, то есть количество попавших в него элементов. Данная гистограмма более сложна в построении, однако оказывается более точной за счёт того, что элементы с большей частотой оказываются в отдельных бакетах;

- Compressed. Данный вид диаграмм считается оптимизацией equi-

height гистограмм, где наиболее часто встречающиеся элементы выносятся из гистограммы в отдельную структуру данных, содержащую пары вида (элемент → его частота). Данная оптимизация позволят минимизировать погрешность в расчётах селективности фильтрации по равенству.

Основными минусами гистограммы являются необходимость чтения большого количества записей таблицы для их создания и большое количество занимаемой ими памяти (размер которой, например, зависит от выбранного в качестве параметра количества бакетов). Стоит однако отметить, что этот недостаток отчасти компенсируется применением алгоритма сэмплирования для уменьшения размера анализируемых записей.

Если ещё раз посмотреть на запрос, представленный в листинге 1, можно увидеть, что существует несколько возможных операторов соединения и фильтрации таблиц: равенства ($=$), неравенства ($\neq$, $>$, $\leq$), проверки вхождения (IN). Все эти условия соединения могут по-разному влиять на результирующий размер соединения. Основным преимуществом гистограмм является возможность определения по ним селективности операторов с фильтрами по равенству, неравенству и по другим возможным операторам. По этой причине в реализуемом модуле оптимизации было принято решение представлять статистику в виде Compressed-гистограмм.

Необходимость совмещения гистограмм по одной колонке, полученных с разных вычислительных узлов может быть достигнута с помощью применения алгоритма, описанного в статье [3].

Алгоритм, описанный в статье [7], позволяет определять селективность JOIN-операторов с условием, использующим оператор неравенства.

2.4. Базы данных с открытым исходным кодом

В данном разделе описан обзор баз данных с открытым исходным кодом, которые рассматривались в качестве вариантов для заимствова-

	Скетчи	Сэмплирование	Гистограммы
Время сбора	+	+-	- (Оптимизируется сэмплированием)
Потребляемая память	+	+-	- (Оптимизируется сэмплированием)
Точность	-	+-	+
Поддерживаемые операторы фильтрации	- (Только =)	- (Только =)	+ (Все операторы: =, !=, <, >, <=, >=)

Рис. 1: Сравнение структур данных

ния логики оптимизации. Критерии, по которым оценивалась пригодность баз данных для выбора их в качестве ориентира для заимствования логики описаны ниже.

- Язык реализации. Ядро базы данных Tarantool и библиотека Sbroad написаны на языках программирования C и Rust соответственно. Поэтому в ходе обзора особое внимание было обращено на те базы данных, чьё ядро реализовано на языках программирования низкого уровня: переиспользование кода таких баз данных оказывается существенно проще за счёт одинакового уровня абстракции работы с памятью.

- Используемые алгоритмы и структуры данных. Внимание было обращено на те базы данных, которые используют детерминированный алгоритм переупорядочивания JOIN-операторов и гистограммы как структуры данных для хранения статистики.

- Полнота документации. Важным требованием для заимствования логики оптимизации из кода базы данных является понятность реализованных алгоритмов и их текстовое описание. Хорошо написанная документация позволяет быстрее осознать общую архитектуру оптимизации и подстроить её под существующую архи-

тектуру библиотеки.

Среди всех баз данных с открытым исходным кодом можно выделить такие, которые изначально разрабатывались с целью быть использованными в распределённых системах и такие, которые будучи первоначально централизованными со временем расширили свою функциональность ради способности быть использованными в распределённой среде. Обозревались базы данных (их краткое сравнение представлено на рис. 2) из обеих категорий, которые:

- были выданы поисковой системой при запросе о стоимостной оптимизации;
- являются популярными и активно используемыми.

MariaDB. Является примером базы данных, которая собирает базовую статистику по колонкам и генерирует гистограммы. С собираемой статистикой можно ознакомиться на сайте документации [14]. Информация по поводу алгоритмов Cost-Based оптимизаций найдена не была.

MySQL. Также собирает базовую статистику по колонкам и хранит гистограммы. С описанием принятых решений можно ознакомиться, изучив описания заголовочных файлов [16] и читая достаточно понятную документацию [17].

PostgreSQL. Среди всех рассмотренных баз данных содержит самую подробную и полную документацию. С собираемой статистикой (включающей в себя гистограммы) [19], формулами подсчёта селективности [20] и генетическим алгоритмом переупорядочивания JOIN-операторов [18] можно ознакомиться на сайте документации.

Spark. Использует equi-height гистограммы для представления статистики. Содержит большое количество видео с конференций (например, поясняющих принцип работы и пользу гистограмм [22]) и подробную документацию в коде. Использует детерминированный алгоритм динамического программирования для решения задачи упорядочивания JOIN-операторов [12]. Написана на языке программирования Java, поэтому как ориентир для заимствования логики не рассматривалась.

Presto. Является примером распределённой базы данных, которая хранит минимальную (не включающую в себя гистограммы) статистику по колонкам [21].

CockroachDB. Среди всех распределённых баз данных содержит самую подробную документацию. Среди прочей статистики собирает гистограммы [5]. С принципом работы Cost-Based оптимизатора можно ознакомиться на сайте документации [6]. Написана на языке Go, что при необходимости позволяет с небольшими усилиями переиспользовать реализацию. В качестве алгоритма переупорядочивания использует детерминированный алгоритм динамического программирования.

В результате обзора в качестве ориентиров для заимствования логики были выбраны следующие базы данных:

- PostgreSQL. Реализованные в базе данных алгоритмы сбора статистики и подсчёта на их основании селективности с учётом подробной документации оказались легко переносимыми в существующую архитектуру библиотеки на язык программирования Rust;
- CockroachDB. Используемый в базе данных алгоритм [9] переупорядочивания учитывает разнообразные типы JOIN-операторов, что отличает его от алгоритмов, используемых в других базах данных.

	Централизованные			**Распределённые**		
	MariaDB	**MySQL**	**PostgreSQL**	**Presto**	**Spark**	**CockroachDB**
Язык реализации	C++	C++	C	Java	Java	Go
Статистика	Equi-height гистограмма	Compressed гистограмма	Compressed гистограмма	Общая статистика	Equi-height гистограмма	Equi-height гистограмма
Алгоритм упорядочивания	Детерминированный	Детерминированный	Детерминированный/ генетический	Детерминированный	Детерминированный	Детерминированный (поддерживает разные типы JOIN-операторов)
Полнота документации	-	+-	+	-	+-	+

Рис. 2: Сравнение баз данных с открытым исходным кодом

2.5. Существующая оптимизация

В заключении обзора скажем, что из себя представляет существующая оптимизация распределённых JOIN-запросов.

В введении было сказано, что в существующей архитектуре выбор плана распределённого запроса, включающего в себя JOIN-оператор, основывается на информации о том, по каким ключам производится операции соединения. Рассмотрим более подробно, каким образом принимается решение о построении плана.

Поскольку данные всех таблиц в кластере базы данных оказываются распределены по разным вычислительным узлам, для определения расположения всех записей из таблицы применяется функция распределения. По заранее выбранному ключу распределения (колонке или группе колонок) данная функция по принципу, схожему с принципом работы хэш-функций, определяет расположение каждой записи таблицы базы данных.

Для того, чтобы понять, каким образом ключи соединения влияют на план, рассмотрим две таблицы: A и B, ключами распределения которых являются соответственно колонки a и b. Посмотрим, как в зависимости от выбора ключа соединения будет строиться план на примере трёх запросов, представленных на листинге 2 (с данной логикой можно ознакомиться в презентации по внутренней архитектуре библиотеки [23]):

Листинг 2: JOIN-операторы с разными ключами соединения

```
1) A JOIN B ON A.a = B.b;
2) A JOIN B ON A.a = B.c;
3) A JOIN B ON A.d = B.c.
```

В случае 1), где ключи соединения обеих таблиц совпадают с их ключами распределения, JOIN-оператор исполняется на узлах кластера локально: согласно принципу работы функции распределения, все удовлетворяющие условию записи обеих таблиц будут находиться на одних и тех же вычислительных узлах. В данном случае для исполнения запроса не происходит пересылка данных по сети.

Для двух других случаев реализовано две стратегии исполнения распределённого JOIN-запроса:

1. перераспределение (repartition);

2. пересылка целиком (broadcast).

В случае 2), где ключ соединения совпал только с ключом распределения таблицы A, будет принято решение о перераспределении записей из таблицы B по новому ключу распределения — колонке c. Информация о распределении записей таблицы A (которую обеспечивает функция распределения) позволяет распределить соответствующим образом колонки из таблицы B и оказаться в ситуации из случая 1).

В случае 3), где ключи соединения не совпали с ключами распределения ни одной из таблиц, будет принято решение целиком скопировать одну из таблиц на каждый из вычислительных узлов кластера. Мы вынуждены прибегнуть к такому решению вследствие того, что функция распределения не позволяет нам определить расположения записей ни одной из таблиц. Всегда предполагается, что таблица справа (правый ребёнок JOIN-оператора) имеет меньший размер, поэтому существующая реализация всегда копирует именно её.

Предполагается, что стратегия полной пересылки одной таблицы является более оптимальным вариантом, чем перераспределение двух таблиц, однако в некоторых случаях это не так. Если размеры обеих таблиц оказываются примерно одинаковыми, выгоднее оказывается перераспределить по ключам соединения обе таблицы. Одной из задач реализованного модуля Cost-Based оптимизации является выбор наиболее оптимальной стратегии исполнения бинарного JOIN-оператора на основании собранной статистики.

В существующей архитектуре за перемещение данных по сети отвечает специальный узел дерева плана под названием **Motion**. Являясь ребёнком JOIN-оператора, этот узел с заданной стратегией символизирует перемещение записей таблиц с одного узла кластера на другой. О том, каким образом оптимизация была встроена в существующую архитектуру библиотеки написано в главе **Особенности реализации**.

3. Архитектура

В данной главе представлено описание архитектуры реализованного модуля оценочной оптимизации.

Модуль оптимизации написан на языке программирования Rust. Некоторые архитектурные решения были приняты в связи с такими используемыми языком концепциями как *borrow checking*, *ownership* и *lifetimes*. Строгость статического анализа кода компилятора Rust не всегда позволяет удобным образом оперировать структурами данных. Так, например, накладываются ограничения на использование ссылочных полей и рекурсивных (самоссылающихся) структур. Диаграммы, представленные ниже, скрывают излишнюю логику, возникающую для обхода ограничений компилятора, за более понятными представлениями.

На рис. 3 можно ознакомиться с концепцией реализованной оптимизации, представленной в виде диаграммы компонентов. Входными данными библиотеки Sbroad в целом является SQL-запрос. Компонент **MainOptimizer** (который содержит в себе уже реализованную логику библиотеки) трансформирует SQL-запрос в план, представленный структурой данных **Plan**.

Для работы алгоритма переупорядочивания JOIN-операторов информацию о запросе, содержащуюся в плане, необходимо расширить ссылками на используемые в каждом операторе таблицы. Компонент **PlanTransformer** отвечает за трансформацию плана в расширенную структуру данных **ExtendedPlan**.

Из расширенного плана далее компонент **OperatorsExtractor** создаёт структуру данных под названием **JoinOperatorsTree**. Она представляет собой дерево JOIN-операторов без информации об остальных реляционных узлах и с дополнительной специфической информацией, необходимой для работы алгоритма. Промежуточная сущность **ExtendedPlan** необходима для того, чтобы после завершения работы алгоритма оптимизации была возможность заново построить план, поскольку именно на этой структуре данных построена вся основная логика

библиотеки Sbroad.

В ходе своей работы алгоритм переупорядочивания (основная логика которого инкапсулирована в подкомпоненте **Reorderer**), оперируя структурой **JoinOperatorsTree**, генерирует корректные возможные последовательности исполнения запроса, представленные в виде структуры данных **ExtendedPlan**. Далее компонент **Coster** на основании доступной статистики (которую он получает путём обращения к компоненту **StatisticsFetcher**) присваивает сгенерированной последовательности стоимость и возвращает её компоненту **Reorderer**. Этот процесс продолжается до тех пор, пока алгоритм переупорядочивания не переберёт все возможные последовательности. В конце работы алгоритм возвращает план с наименьшей стоимостью исполнения.

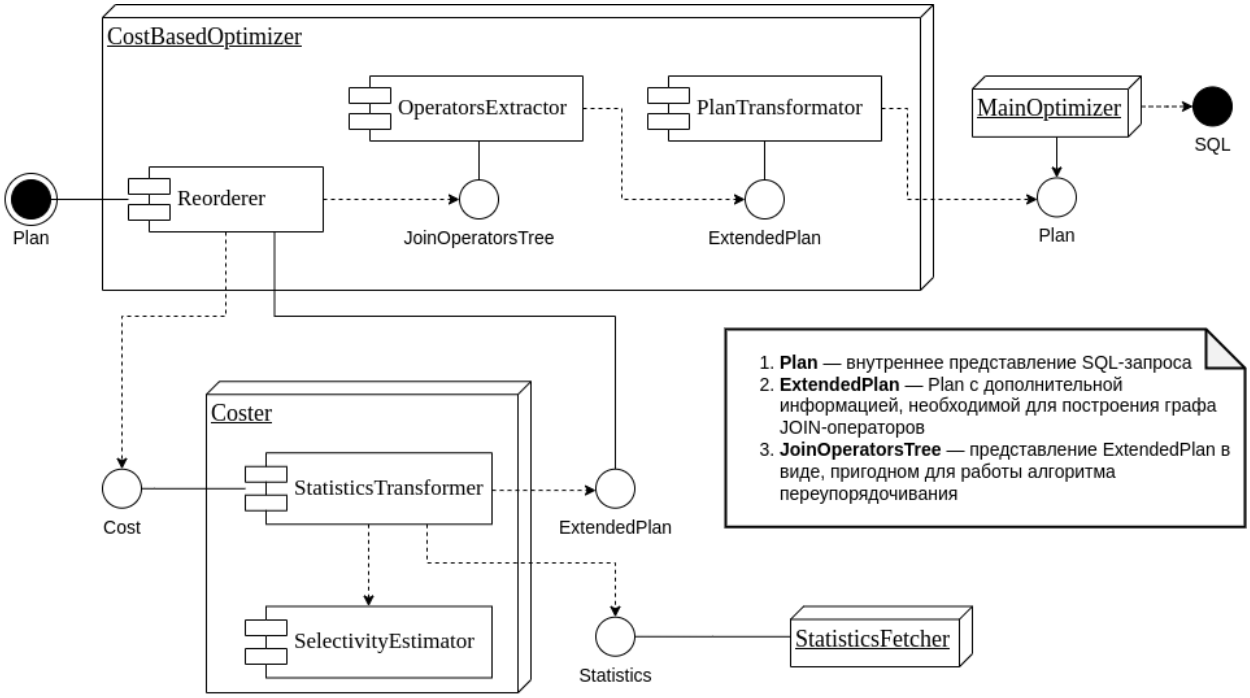

Рис. 3: Диаграмма компонентов реализованного модуля оптимизации

На рис. 4 представлена диаграмма классов, на которой можно более детально ознакомиться с описываемой реализацией. Зелёным цветом на ней отмечены классы, которые отвечают за стоимостную оптимизацию запроса. Серым — существующие классы библиотеки. Стоит сразу отметить, что именование классов лишь частично совпадает с именованием, представленным на диаграмме компонентов. Например,

на диаграмме можно изучить специфические поля и методы класса **JoinOperatorsTree** (например, *tes* и *applicable*), необходимые для работы алгоритма переупорядочивания.

Важной особенностью модуля оптимизации является тот факт, что помимо обработки последовательности JOIN-операторов приходится иметь дело ещё и с такими операторами, как Selection, UnionAll (называемыми на диаграмме классов *специальными вершинами*), обработку поддеревьев которых приходится поддерживать отдельно. Именно поэтому класс **CostBasedOptimizer** оперирует с очередью специальных вершин *optimization_ queue*: очерёдность обработки этих вершин совпадает с очерёдностью их появления в процессе обхода расширенного дерева плана в глубину.

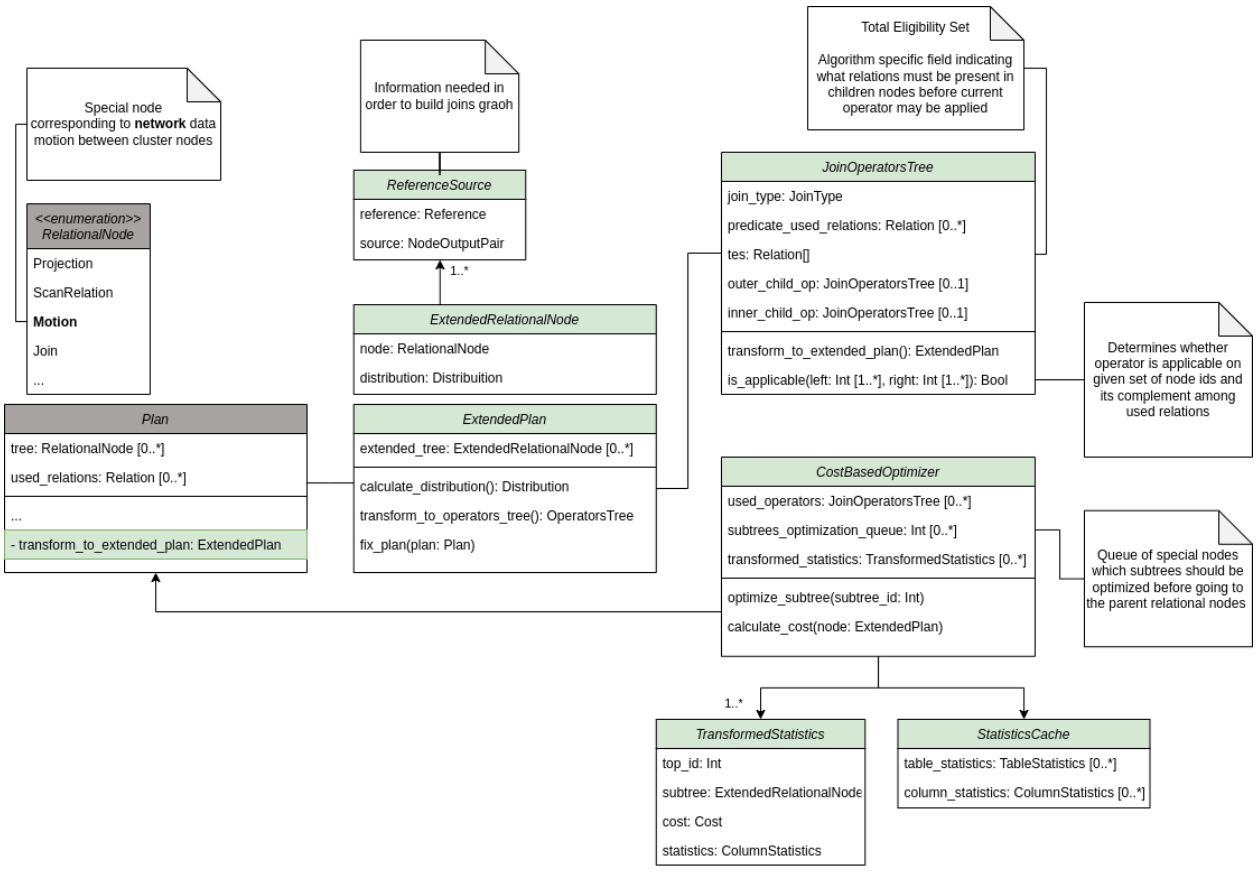

Рис. 4: Диаграмма классов реализованного модуля оптимизации

На рис. 5 можно увидеть, какие поля в себя включают структуры статистики.

- **Histogram**. Включает в себя массивы объектов **MostCommon**

(наиболее часто встречающиеся в колонке элементы с соответствующими частотами) и **Bucket** (бакеты гистограммы), фракцию нулевых элементов и фракцию уникальных элементов (которая, будучи умноженной на количество элементов в колонке, даст количество уникальных элементов).

- **ColumnStats**. Включает в себя минимальный и максимальный элементы колонки, количество элементов, средний размер элементов колонки и саму гистограмму.

Оба класса, **ColumnStats** и **Histogram**, реализуют два интерфейса:

- **SelectivityEstimator**. Который описывает логику подсчёта селективности определённых операторов фильтрации и соединения;
- **StatisticsTransformer**. Который описывает логику трансформации статистики в процессе применения специфических преобразований над колонками: их слияния при обработке оператора UnionAll, прибавлении к значениям колонки константного значения и других.

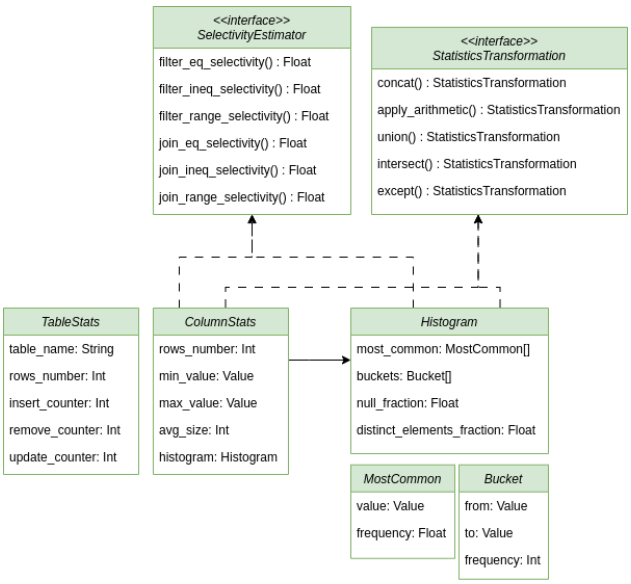

Рис. 5: Диаграмма классов используемой статистики

4. Особенности реализации

В данном разделе рассказано об особенностях реализованного модуля оптимизации. Описание работы определённой функциональности модуля ссылается на упомянутые в главе **Архитектура** классы, поля и методы.

4.1. Алгоритм переупорядочивания

Для работы любого алгоритма переупорядочивания (в данной работе реализован алгоритм под названием DPsube) необходимо построить граф JOIN-операторов, используемых в оптимизируемом запросе. В отличие от дерева реляционных операторов, которое является последовательностью исполнения запроса, граф операторов содержит в себе информацию о том, по каким ключам происходит соединение в каждом из операторов и как между собой связаны исходные таблицы, используемые в запросе. Пример запроса с соответствующим ему графом JOIN-операторов можно увидеть на рис.6.

В существующей реализации библиотеки Sbroad при построении плана не сохраняется информация, необходимая для построения графа операторов. С целью получения этой информации создаётся структура данных под названием **ExtendedPlan**. Для каждого JOIN-оператора из дерева плана во время его рекурсивного обхода сохраняется информация о корневых узлах, колонки которых используются в условии соединения данного JOIN-оператора.

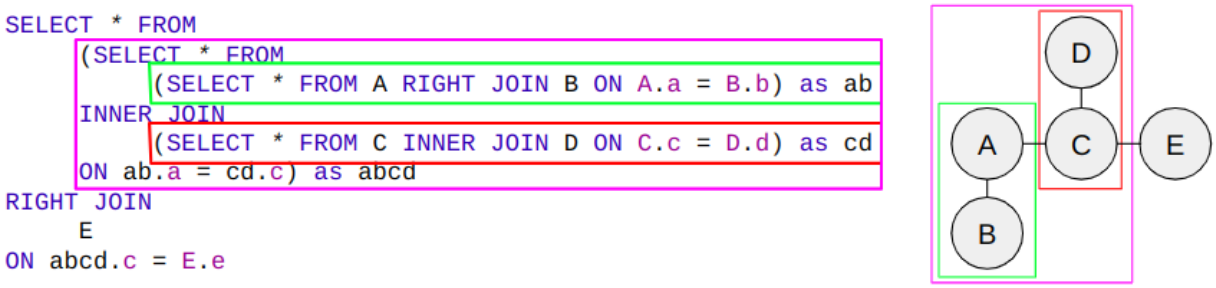

Рис. 6: Граф JOIN-операторов

Логика данного процесса усложняется в тех случаях, когда в плане

встречаются реляционные узлы (далее называемые *специальными*) вроде Selection и UnionAll, которые накладывают ограничения на переупорядочивание JOIN-операторов. Реализованная оптимизация оперирует данными специальными узлами (вместе с их поддеревьями) как с цельной таблицей, не давая возможности родительскому оператору переупорядочивать соединения их листовых узлов (которые чаще всего являются исходными таблицами базы данных). Пример добавления подобного специального Selection-узла изображён на рис.7. Если в дереве слева у алгоритма переупорядочивания была возможность любым возможным способом упорядочивать листовые узлы A, B, C, D и E, то в дереве справа корневому узлу приходится оперировать поддервом Selection-узла как с единой таблицей ABC. При этом перед оптимизацией всего дерева происходит оптимизация поддерева Selection-узла: поскольку оно само содержит JOIN-операторы, у него тоже есть возможность применить оптимизацию.

Очерёдность оптимизации подобных поддеревьев определяется путём обхода дерева реляционных операторов в глубину: специальная вершина добавляется в очередь *subtrees_ optimization_ queue* (о которой шла речь в главе **Архитектура**), когда заканчивается обход всех её детей.

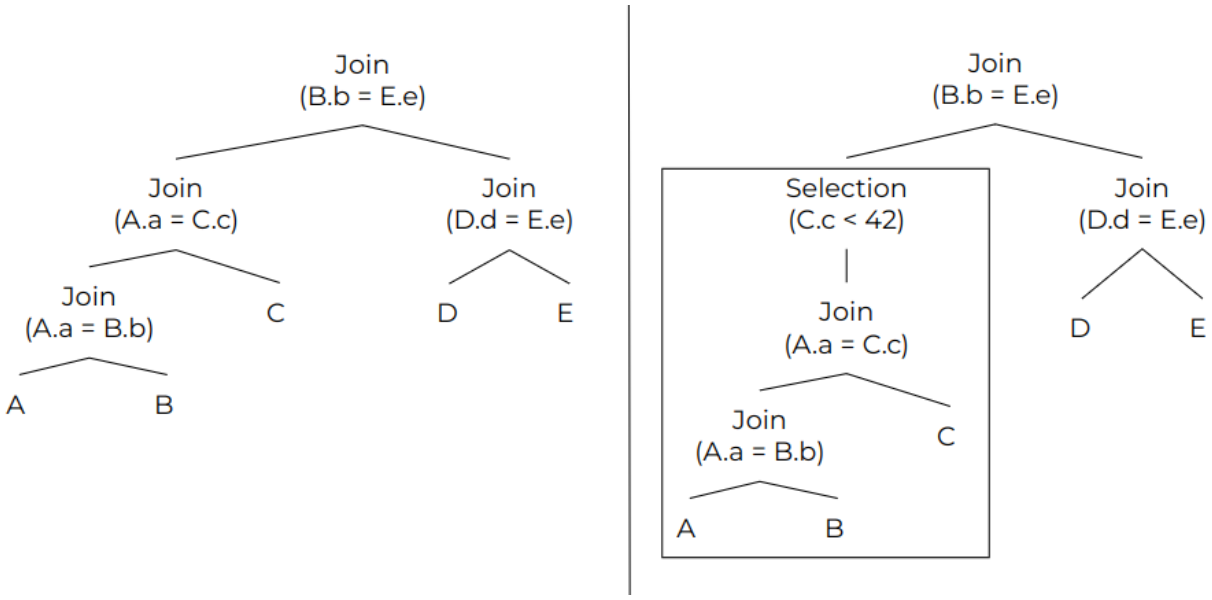

Рис. 7: Пример специальной вершины

Для применения алгоритма DPsube необходимо вычленить из всего дерева реляционных операторов узлы JOIN-операторов и дополнить их специфической для алгоритма информацией. **JoinOperatorsTree** – это структура данных, пригодная для работы алгоритма. В процессе обработки поддерева (будь это поддерево специальной вершины или всё дерево запроса) в этой структуре заполняются (помимо прочих) следующие поля:

- STO (SubTree Operators) — множество операторов, лежащих в поддереве данного оператора исходного запроса;
- predicate_used_relations — множество листовых узлов дерева (часто всего исходных таблиц базы данных), которые используются в условии соединения оператора;
- TES (Total Eligibility Set) — множество листовых узлов дерева, которые должны быть обработаны к моменту добавления данного оператора в последовательность исполнения (которую и строит алгоритм переупорядочивания). Оно определяется, исходя из информации об условии соединения оператора и типа оператора.

Данные деревья строятся для корневого узла и для всех специальных вершин до применения всех оптимизаций. В случае, когда листовым узлом одного из поддеревьев оказывается специальная вершина, она помечается специальным флагом и обрабатывается таким же образом, как если бы это была исходная таблица базы данных.

4.2. Подсчёт селективности

Логика определения селективности операторов WHERE и ON полностью заимствована из кода базы данных PostgreSQL. Поскольку данная функциональность никаким образом не связана с архитектурой дерева реляционных операторов (она связана только с обработкой структур данных, представляющих статистику), единственной задачей было переписать код с языка программирования C на язык программирования Rust.

4.3. Подсчёт стоимости плана

За стоимость плана в данной работе принимается размер данных, пересылаемых в процессе его исполнения. Как следствие, единственное место в дереве реляционных операторов, в котором возникает необходимость подсчёта стоимости — это JOIN-узлы. В случае, если оптимизатор принимает решение о перераспределении записей ребёнка JOIN-оператора, между оператором и ребёнком вставляется специальный Motion-узел с флагом, символизирующим выбранную стратегию перераспределения (Repartition или Broadcast).

Выбрав наиболее оптимальную стратегию распределения записей каждого из детей JOIN-оператора, алгоритм присваивает ему стоимость, равную сумме стоимостей пересылки записей каждого из его детей в соответствии с выбранной стратегией:

- $broadcast\_cost = table\_size$;
- $repartition\_cost = table\_size * \frac{N-1}{N}$, где N — количество вычислительных узлов в кластере.

Подсчёт стоимости всего плана осуществляется путём его рекурсивного обхода. Узлам, не имеющим дочерних Motion-вершин, приравнивается стоимость, равная сумме стоимостей своих детей.

4.4. Трансформация статистики

Подсчёт стоимости плана опирается на известную статистику колонок таблиц базы данных. Стоит сказать, что из себя представляет значение $table\_size$, фигурирующее в формулах для подсчёта стоимости. Если речь идёт о листовой вершине дерева реляционных операторов (то есть *Scan*-вершине, чтении таблицы прямо из базы данных), $table\_size$ вычисляется по формуле $rows\_number * rows\_avg\_size$, где $rows\_number$ — это количество записей в таблице, а $rows\_avg\_size$ — это средний размер записи таблицы, вычисляемый как сумма средних размеров каждой из колонок таблицы.

В ходе рекурсивного обхода дерева операторов во время подсчёта стоимости плана статистика по колонкам от листовых Scan-вершин проходит через другие вершины, претерпевая трансформацию. Под трансформацией предполагается преобразование колоночной статистики в следующих случаях (некоторые из которых изображены на листинге 3):

1. построение нового выражения в проекции;
2. применение WHERE-условия;
3. применение ON-условия;
4. применение операции над множествами в случае обработки операторов UnionAll, Except и Intersect.

Листинг 3: Случаи трансформации статистики

```
1) SELECT a + b FROM ...
2) SELECT ... WHERE a != 42
3) ... JOIN ... ON a = b
4) ... T1 UNION ALL T2
```

Поскольку логика трансформации не была найдена ни в одной из баз данных, рассматриваемых как ориентиры для заимствования логики, данный алгоритм был реализован с нуля.

Применение трансформации предполагает изменение всех доступных полей статистики. Так, например, применение арифметической операции между колонкой и константным значением должно применить эту арифметическую операцию к:

- минимальному и максимальному значению;
- всем значениям массива *most_common*;
- всем значениям из массива бакетов.

Условие фильтрации, помимо прочего должно обновить значения массивов бакетов и *most_ common* таким образом, чтобы сохранились только те значения, которые удовлетворяют применяемому оператору.

Стоит отметить, что в процессе трансформации, некоторая колоночная статистика может потеряться. При применении в условии фильтрации любого оператора между двумя колонками таблиц (а не, например, колонкой и константным значением) мы лишаемся возможности предугадать, какие значения останутся в полях статистики и принимаем решение обнулить статистику по колонкам всех таблиц, участвующих в условии фильтрации. Единственные поля, которые всегда присутствуют в статистике колонок — это *rows_ number* и *avg_ size* (среднее значение в байтах по конкретной колонке). В случае, если алгоритм подсчёта селективности при обработке условия фильтрации встречает колонку с обнулённой статистикой, он принимает решение об использовании константного эвристического значения селективности.

Отдельно стоит упомянуть, что трансформация статистики возникает ещё и в том случае, когда нам необходимо собрать статистику по колонкам таблицы с нескольких узлов кластера. Поскольку данные всех таблиц оказываются распределены по нескольким узлам, то и статистика по их колонкам тоже оказывается распределена. Логика трансформации в таком случае совпадает с логикой трансформации оператора UnionAll: в обоих случаях происходит процесс объединения всех записей из всех таблиц (двух, в случае UnionAll).

4.5. Изменение плана запроса

После завершения работы алгоритма переупорядочивания возникает необходимость на основании вспомогательной структуры данных **ExtendedPlan** восстановить дерево реляционных операторов в виде структуры данных **Plan**, на которой построена вся логика работы библиотеки Sbroad.

Алгоритм в процессе своей работы не затрагивает первоначальный план, порождаемый основным модулем оптимизации библиотеки Sbro-

ad. По этой причине процесс построения (восстановления) оптимизированного плана оказывается проще: вместо построения плана с нуля необходимо только изменить изначальное дерево реляционных операторов таким образом, чтобы дети каждого оператора соответствовали детям этого же оператора в дереве расширенного плана.

5. Эксперименты

5.1. Постановка экспериментов

В качестве методов проведения замеров эффективности реализуемой оптимизации можно рассматривать два подхода: логический и физический.

Логический подход основывается на проверке эффективности преобразования SQL-запроса с помощью анализа его плана. План представляет собой дерево реляционных операторов, последовательное исполнение которых приводит к желаемому результату SQL-запроса. Одному SQL-запросу могут соответствовать несколько планов: например, в них могут отличаться последовательность исполнения JOIN-операторов и выбранные стратегии их распределённого исполнения. Сравнивая между собой планы до и после применения оптимизации и анализируя информацию об их подсчитанных стоимостях можно сделать некоторые предположения об эффективности оптимизации.

Физический подход основывается на реальном исполнении запросов в кластере и замере их эффективности. Такое тестирование является более естественным, потому что позволяет замерить показатели скорости обработки запросов и точность определения селективности. Замеры эффективности, произведённые на разных кластерах (например, развёрнутых локально или на реальном серверном окружении) сильно зависят от характеристик оборудования. Основное предположение, которое должно доказывать эффективность реализованной оптимизации заключается в том, что существует корреляция между оценочной стоимостью плана и фактическим временем его исполнения. Предполагается, что если такая корреляция будет наблюдаться в локально развёрнутом кластере, то она будет наблюдаться и в реальной среде.

Помимо стоимости оптимизированного плана и времени исполнения, мы также можем оценить время работы библиотеки Sbroad с опцией Cost-Based оптимизации и без неё. Исходя из предположения, что модуль оптимизации не должен генерировать планы с большей сто-

имостью, и анализируя время работы алгоритма, мы можем сделать некоторые предположения о том, насколько оправданным оказывается внедрение данного алгоритма с точки зрения издержек на оптимизацию.

В качестве тестовых запросов были выбраны запросы, включающие в себя один и более JOIN-операторов, которые используют клиенты компании Пикодата, поскольку именно на обеспечение эффективного исполнения этих запросов и направлена деятельность компании. Имена таблиц и колонок изменены из соображений конфиденциальности.

Эксперименты проводились на устройстве со следующими характеристиками.

- Операционная система — Ubuntu 20.04.5 LTS;
- Процессор — Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz;
- Оперативная память — 16 GB.

Развёрнутый кластер представляет из себя 5 процессов, запущенных на локальных портах.

5.2. Исследовательские вопросы (Research questions)

Сформулируем результаты, которые мы бы хотели увидеть от реализованного модуля оптимизации.

- Стоимость плана, получившегося после применения оптимизации должна быть ниже стоимости изначально плана.
- Время исполнения запроса с применённой оптимизацией должно быть меньше времени исполнения без неё.
- Время работы алгоритма переупорядочивания должно составлять достаточно маленькую часть от времени работы всей библиотеки Sbroad.

5.3. Метрики

Сравнение результатов с учётом озвученных выше исследовательских вопросов оказывается очевидным.

- Стоимости планов сравниваются как две численные величины. Чем меньше стоимость плана, тем, предположительно, эффективнее оказывается запрос.
- Время исполнения запроса измеряется в миллисекундах. Чем меньше время исполнения запроса, тем он оценивается более эффективным.
- Время работы алгоритма переупорядочивания и библиотеки Sbroad в целом измеряются в миллисекундах. Чем меньшую часть от всего времени работы составляет время работы алгоритма, тем он оценивается более эффективным с точки зрения накладных временных расходов.

5.4. Результаты

Эксперименты проводились на тестовых запросах №1 и №2 (изображённых соответственно на рис. 4 и рис. 5), которые оперируют тремя таблицами: *product*, *order* и *location*.

Ключи распределения и размеры таблиц представлены ниже.

- product: *p_id*, 24MB;
- order: *o_id*, 100MB;
- location: *l_id*, 200MB.

Для каждого из запросов было проведено 5 экспериментов, из результатов которых были подсчитаны средние значения и погрешность с учётом значения доверительного интервала, равного 95%.

Листинг 4: Тестовый запрос №1

```
SELECT * FROM
```

```
    order
INNER JOIN
    product
ON order.amount = product.price
```

На рис. 8 изображены результаты проведённых замеров для запроса №1. Можно отметить уменьшение как оценочной стоимости, так и фактического времени исполнения запроса. В данном случае в условии соединения фигурируют колонки, не совпадающие с ключами распределения ни одной из таблиц. Результаты оптимизации в данном случае обусловлены тем фактом, что при построении плана принимается решение вместо полной пересылки правой таблицы JOIN-оператора перераспределить по ключам соединения обе таблицы. Время работы библиотеки не сильно отличается с включенной и выключенной опцией оценочной оптимизации. Это обусловлено тем фактом, что в запросе участвует только один JOIN-оператор.

	Время работы библиотеки	Стоимость плана	Время исполнения запроса
Без оптимизации	0.014s (±2%)	120MB	2.059s (±6%)
С оптимизацией	0.015s (±2%)	16MB	1.424s (±5%)

Доверительный интервал — 95%

Рис. 8: Результаты тестового запроса №1

На рис. 9 изображены результаты проведённых замеров для запроса №2. Можно также отметить уменьшение как оценочной стоимости планов, так и фактического времени исполнения запроса. Увеличение времени исполнения запроса (как и его оценочной стоимости) связано с увеличением количества записей в результирующем соединении. Алгоритм упорядочивания также не вносит большого вклада во время работы библиотеки, при этом составляя заметно малую часть от времени исполнения самого запроса.

Листинг 5: Тестовый запрос №2

```
SELECT * FROM
    product
INNER JOIN
    (SELECT * FROM
        location
    INNER JOIN
        order
    ON location.order_id = order.o_id) as interm
ON product.p_id = interm.product_units
```

	Время работы библиотеки	Стоимость плана	Время исполнения запроса
Без оптимизации	0.055s (±2%)	2800MB	11.760s (±10%)
С оптимизацией	0.059s (±1%)	442MB	7.411s (±11%)

Доверительный интервал — 95%

Рис. 9: Результаты тестового запроса №2

Заключение

В ходе проделанной выпускной квалификационной работы были достигнуты следующие результаты.

- Проведён обзор структур данных и алгоритмов переупорядочивания последовательности JOIN-операторов, используемых для реализации оценочной оптимизации. В качестве ориентиров для заимствования логики были выбраны базы данных PostgreSQL и CockroachDB.
- Создана архитектура дополнительного модуля оценочной оптимизации в библиотеке Sbroad.
- На языке программирования Rust реализован модуль оценочной оптимизации, использующий алгоритм динамического программирования для переупорядочивания JOIN-операторов и подсчёта селективности с использованием статистики, представленной в виде Compressed гистограмм.
- Проведены эксперименты, оценивающие качество работы оптимизации.
- Результаты представлены на конференции «Современные технологии в теории и практике программирования».

С реализованным модулем оптимизации можно ознакомиться на GitLab, перейдя в файл sbroad-core/src/cbo.rs.

Задачи, связанные с реализацией модуля Cost-Based оптимизации, помечены ярлыком «Cost-Based». Изучив их, можно ознакомиться с процессом разработки.

Список литературы

[1] Allam Julian Reddy. Evaluation of a greedy join-order optimization approach using the IMDB dataset. — URL: https://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisAllam19.pdf (online; accessed: 2022-12-12).

[2] Apache. Apache Madlib GitHub. — URL: https://github.com/apache/madlib/blob/4987e8fe5367bb823afb1bd4020fd6f0fa603258/methods/sketch/src/pg_gp/countmin.c (online; accessed: 2022-12-12).

[3] Burak Yıldız Tolga Buy¨ uktanır, Emekci Fatih. Equi-depth Histogram Construction for Big Data with Quality Guarantees. — URL: https://arxiv.org/pdf/1606.05633.pdf (online; accessed: 2022-12-12).

[4] Christina Pavlopoulou Michael J. Carey Vassilis J. Tsotras. Revisiting Runtime Dynamic Optimization for Join Qeries in Big Data Management Systems. — URL: https://openproceedings.org/2022/conf/edbt/paper-6.pdf (online; accessed: 2022-12-12).

[5] CockroachDB. CockroachDB GitHub. — URL: https://github.com/cockroachdb/cockroach/blob/master/pkg/sql/stats/histogram.go (online; accessed: 2022-12-12).

[6] CockroachDB. Cost-Based Optimizer. — URL: https://www.cockroachlabs.com/docs/stable/cost-based-optimizer.html (online; accessed: 2022-12-12).

[7] Diogo Repas Zhicheng Luo Maxime Schoemans Mahmoud Sakr. Selectivity Estimation of Inequality Joins In Databases. — URL: https://arxiv.org/pdf/2206.07396.pdf (online; accessed: 2022-12-12).

[8] Florin Rusu Alin Dobra. Statistical Analysis of Sketch Estimators. — URL: https://www.cise.ufl.edu/~adobra/Publications/sigmod-2007-statistics.pdf (online; accessed: 2022-12-12).

[9] Guido Moerkotte Pit Fender Marius Eich. On the correct and complete enumeration of the core search space. — URL: https://www.researchgate.net/publication/262216932_On_the_correct_and_complete_enumeration_of_the_core_search_space#fullTextFileContent (online; accessed: 2022-12-12).

[10] Hai Lan Zhifeng Bao Yuwei Peng. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. — URL: https://arxiv.org/pdf/2101.01507.pdf (online; accessed: 2022-12-12).

[11] Jaffray Justin. An Introduction to Join Ordering. — URL: https://www.cockroachlabs.com/blog/join-ordering-pt1/ (online; accessed: 2022-12-12).

[12] KONIECZNY BARTOSZ. Reorder JOIN optimizer - cost-based optimization. — URL: https://www.waitingforcode.com/apache-spark-sql/reorder-join-optimizer-cost-based-optimization/read (online; accessed: 2022-12-12).

[13] Katsov Ilya. Probabilistic data structures for web analytics and data mining. — URL: https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/ (online; accessed: 2022-12-12).

[14] MariaDB. Column Stats Table. — URL: https://mariadb.com/kb/en/mysqlcolumn_stats-table/ (online; accessed: 2022-12-12).

[15] Michael Steinbrunn Guido Moerkott Alfons Kemp. Heuristic and Randomized Optimization for the Join Ordering Problem. — URL: https://www.csd.uoc.gr/~hy460/pdf/HeuristicandRandomizedOptimizationfortheJoinOrdering%20Problem.pdf (online; accessed: 2022-12-12).

[16] MySQL. MySQL Source Code Documentation. — URL: https://dev.mysql.com/doc/dev/mysql-server/latest/equi__height_8h.html (online; accessed: 2022-12-12).

[17] MySQL. The Optimizer Cost Model. — URL: https://dev.mysql.com/doc/refman/5.7/en/cost-model.html (online; accessed: 2022-12-12).

[18] PostgreSQL. PostgreSQL GitHub. — URL: https://github.com/postgres/postgres/blob/master/src/backend/optimizer/geqo/geqo_main.c (online; accessed: 2022-12-12).

[19] PostgreSQL. PostgreSQL pgstats View. — URL: https://www.postgresql.org/docs/current/view-pg-stats.html (online; accessed: 2022-12-12).

[20] PostgreSQL. Row Estimation Examples. — URL: https://www.postgresql.org/docs/current/row-estimation-examples.html (online; accessed: 2022-12-12).

[21] Presto. Introduction to Trino Cost-Based Optimizer. — URL: https://www.starburst.io/blog/introduction-to-presto-cost-based-optimizer/ (online; accessed: 2022-12-12).

[22] Ron Hu Zhenhua Wang. Cardinality Estimation through Histogram in Apache Spark 2.3. — URL: https://www.databricks.com/session/cardinality-estimation-through-histogram-in-apache-spark-2-3 (online; accessed: 2022-12-12).

[23] Smirnov Denis. Sbroad Internal design. — URL: https://git.picodata.io/picodata/picodata/sbroad/-/blob/main/doc/design/sbroad.pdf (online; accessed: 2022-12-12).