Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Куликов Даниил Витальевич

Выпускная квалификационная работа

Исследование влияния точности модели затрат на исполнение аналитических запросов к СУБД в основной памяти в гетерогенных системах

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель:
доцент, факультет математики и компьютерных наук, к.ф.-м.н., Д.С. Шалымов

Рецензент:
Руководитель лаборатории системного программирования и информационной безопасности, РАН, д.ф.-м.н. А. А. Белеванцев

Санкт-Петербург
2023 г.

Содержание

Введение

С течением времени количество информации, которое производит человечество, растет [1]. В связи с прогнозируемым ростом объема данных остро встает вопрос об анализе этих данных . Логично предположить, что при их росте потенциально будет возникать нужда в ускорении обработки. Для решения данной проблемы разрабатывается и исследуется множество разных подходов. Один из наиболее перспективных – использование аппаратных ускорителей, в частности графических процессоров. Уже сейчас графические устройства показывают хорошие результаты в области ускорения работы моделей машинного обучения, а тренды развития многообещающие [14].

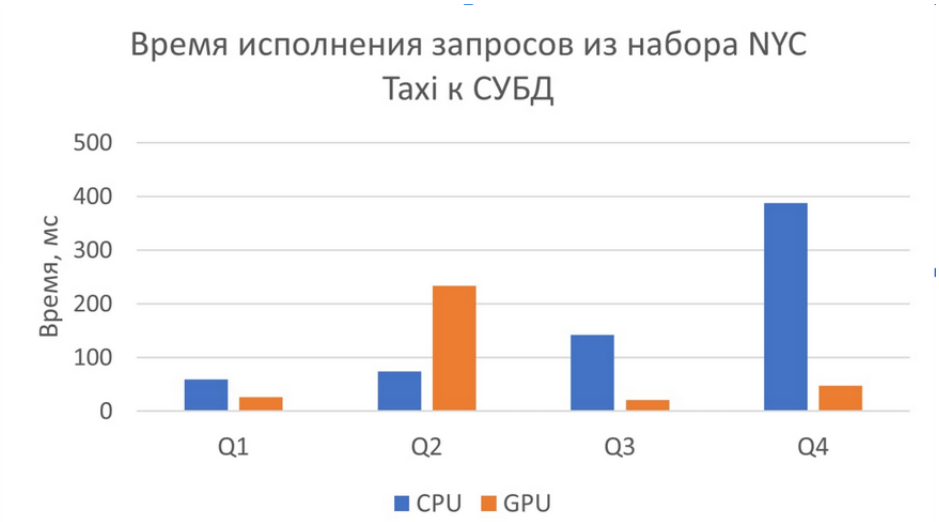

Рис. 1: Сравнение времени исполнения запросов системы бенчмаркинга NYC Taxi на ЦПУ и графическом процессоре.

В обработке данных графические процессоры также могут показывать неплохие результаты (**Рис. 1**). Однако в то же время сравнение может продемонстрировать отсутствие единственно оптимального устройства для исполнения аналитических запросов к СУБД. Это наблюдение показывает, что интеграция графического устройства в систему не решает все проблемы автоматически и требует дополнительных усилий.

Причины у таких результатов – разные архитектурные особенности центрального и графического процессоров, разные подходы к разработке,

разные программные модели и модели памяти.

Все эти наблюдения приводят к тому, что для успешного ускорения исполнения аналитических запросов с использованием графического устройства, необходимо разработать механизм распределения работ между устройствами. Формально говоря, возникает дополнительный параметр, который необходимо оптимизировать при поиске оптимального плана запроса.

Задача поиска оптимального плана в реляционных СУБД как правило решается использованием *моделей затрат*. Их задача давать оценку каждому физическому плану таким образом, чтобы оптимизация этой оценки приводила к оптимизации времени исполнения. Модели затрат могут давать как оценку на время исполнения, так и генерировать абстрактное значение, позволяющее оптимизировать время исполнения.

Проблема существующих решений в том, что они не учитывают возникший при использовании гетерогенного исполнения параметр, а их адаптация к новым архитектурам может быть затруднительной. Цель этой выпускной квалификационной работы изучить возможность использования модели затрат в целях оптимизации гетерогенных планов запросов и их влияние на качество этих планов.

Постановка задачи

Целью данной работы является разработка и исследование модели затрат для исполнения аналитических запросов в гетерогенных системах.

В рамках этой темы необходимо решить следующие **задачи**:

1. Реализовать модель затрат в *HDK* — низкоуровневом движке исполнения аналитических запросов.

2. Исследовать влияние точности модели затрат на исполнение аналитических запросов в гетерогенных системах.

1. Обзорный раздел по предметной области

1.1. Подходы к исполнению запросов

Существует несколько основных подходов к тому, как исполнять план запроса.

Итеративный подход подразумевает исполнение путем рекурсивного вызова входящих в данный узел плана операторов с целью получить следующую обрабатываемую запись. Основной проблемой данного подхода является низкая локальность данных и большое количество случайных обращений в память, что может негативно влиять на производительность системы [11].

Подход с полной промежуточной материализацией подразумевает последовательное исполнение операторов с вычислением полного промежуточного результата. Данный вариант хорошо подходит для систем с низкими размерами промежуточных данных (например OLTP), но неприемлем для СУБД, работающими с большими данными (например OLAP).

Векторный подход является итеративной моделью, но рекурсивно достаются не единичные записи, а блоки. Этот вариант позволяет эффективно эксплуатировать SIMD инструкции и показывает хорошие результаты при работе с запросами, у которых узким местом является доступ в память [12].

Кодогенерация – подход нацеленный на повышение локальности данных и увеличение времени пребывания данных в регистрах [11] [13]. Операторы плана запроса объединяются в конвейеры (шаги исполнения) и код, исполняющий данный шаг, генерируется системой. Тем самым вставляются элементы, соответствующие информации времени исполнения (части предикатов, количество фильтров и т.д.), обработка записей набором операторов объединяется в одном цикле, что позволяет обрабатывать данные непрерывно каждой операцией и достигать локальность данных. Конвейеры могут прерываться из-за различных физических операторов, требующих полной материализации (например, сортировки).

Потенциальные подходы в гетерогенных системах – векторный и ко-

догенерация. Исследования показывают, что оба подхода показывают сравнимые результаты и нет однозначно лучшего кандидата [12]. Подход с компиляцией запросов позволяет абстрагироваться от архитектурных и микроархитектурных особенностей устройств и эксплуатировать возможности современных компиляторных оптимизаций.

В данной работе будет обсуждаться конвейерный подход с компиляцией запросов.

1.2. Оптимизация планов запросов

Как правило, оптимизаторы запросов, основанные на оценке затрат, состоят из трёх основных компонент: оценки промежуточных размеров данных, оценки стоимости и оптимизации плана запроса.

Модель затрат — модуль, занимающийся оценкой стоимости запроса. Для оценки могут использоваться различные данные о таблице, столбцах, участвующих в запросе, разные характеристики устройств, информация об алгоритмах для исполнения операторов и др. Входными параметрами модели затрат также являются оценки промежуточных размеров данных; эти оценки имеют значительное влияние на качество предсказаний модели затрат [2]. Затем полученная оценка используется компонентом оптимизации плана запроса для перебора и поиска наилучшего решения.

Существуют разные подходы к генерации этой оценки и использовании метаинформации. В этой главе будет проведен обзор существующих решений оценки стоимости запроса.

1.3. Обзор литературы

1.3.1 Аналитическая модель затрат

Классические подходы, как правило, строятся на аналитическом вычислении стоимости, аппроксимации времени исполнения путем оценки различных аппаратных характеристик и стоимостей операций в СУБД [5].

Например, подход, использующийся в реляционной СУБД PostgreSQL [16], состоит в выделении основных значений, влияющих на время исполнения, и их последующем суммировании. Более формально модель затрат

задается вектором $c = (c\_tup, c\_ind, c\_op, c\_seq, c\_rnd)$, где

1. c_tup – стоимость обработки записи на центральном процессоре;
2. c_ind – стоимость обработки записи с помощью индекса на центральном процессоре;
3. c_op – стоимость выполнения операции на центральном процессоре (например, агрегация);
4. c_seq – стоимость последовательного обращения к странице памяти;
5. c_rnd – стоимость случайного обращения к странице памяти.

Для подсчета стоимости берется вектор n, в котором содержатся соответствующие количества данных (количество страниц, количество записей и т.д.), и считается скалярное произведение $\langle c, n \rangle$ [10].

Основные проблемы классического аналитического подхода в следующем:

1. Значения, использующиеся в качестве оценок-весов, калибруются, как правило, в ручном формате, а их изменение, основанное на недостаточно большом количестве данных, может критически повлиять на производительность системы [2]. В принципе ручная калибровка весов не идеальна. Это демонстрируется тем, что автоматизированный подбор значений может заметно улучшить качество оценки, а генерируемые веса могут сильно отличаться от заданных вручную [10].
2. При появлении устройства с новой архитектурой в системе, необходимо расширять метрики, использующиеся для оценки. Данное замечание является существенным для гетерогенных систем. Недостаточная гибкость в расширении используемых архитектур может негативно влиять на темпы развития гетерогенных СУБД.
3. При проектировании аналитических моделей как правило делаются упрощения, влияющие на качество генерируемых стоимостей [6].

1.3.2 Нейросетевой подход к оценке стоимости запроса

Так как аналитические модели обладают рядом недостатков относящихся не только к гетерогенным системам, были попытки использовать методы машинного обучения для генерации оценок. [6] описывает два подхода к генерации оценки:

1. **Моделирование запросов** – генерация стоимости происходит в два этапа: сначала происходит классификация запроса, а затем выбирается соответствующая этому классу нейронная сеть, в свою очередь генерирующая значение.

2. **Моделирование операторов** – на каждый физический оператор строится нейронная сеть и оценка в данном случае – агрегация результатов в узлах физического плана.

Входные данные для моделей могут быть разными: метаданные о предикате, оценка на селективность предиката, входные размеры данных, количество аттрибутов в записи (столбцов), минимум или максимум аттрибута и т.д.

Оба подхода в экспериментальных сравнениях показали свою эффективность по сравнению с аналитическим подходом, так как были способны выявить скрытые факторы, влияющие на производительность (например маленькие по размеру данных запросы помещались в кэш, чего не выявляла аналитическая модель).

Таким образом решается проблема, связанная с упрощениями аналитических подходов и их ручной калибровке.

В то же время данный подход не лишен недостатков:

1. Исследование базируется на теоретическом результате об аппроксимации функций нейронными сетями с прямой связью [7]. Однако данная работа не предоставляет метод построения аппроксиматоров. Таким образом процесс обучения модели затрат может происходить довольно долго, так как для построения подходящей структуры нейронной сети должно пройти несколько стадий: если построенная сеть

не удовлетворяет требованиям, то нейронная сеть переинициализируется новыми случайными значениями; если в результате нескольких неудачных попыток желаемый результат не достигнут, то меняется уже структура самой сети [6].

Более того в контексте гетерогенных систем данная проблема проявляет себя с новой стороны. Внедрение устройства с новой архитектурой или изменение конфигурации системы будет инициализировать процесс нового обучения, что, как уже было указано, может быть неприемлемо дорогим.

2. Второй подход по мнению авторов потенциально демонстрирует лучшие результаты, нежели метод с моделированием запросов. Однако моделирование операторов базируется на операторной модели, что неприменимо в СУБД с кодогенерацией запросов.

1.3.3 Оценка стоимости пользовательских методов в объектно-реляционных СУБД

В [8] вводится метод оценки определяемых пользователем методов. Нахождение оценки пользовательских методов является важной задачей, так как сложные предикаты или компараторы могут в значительной степени влиять на производительность исполнения запросов.

Авторы выделяют три основных класса оцениваемых методов: **константные** (время исполнения не зависит от аргументов), **монотонные** (время исполнения монотонно изменяется с ростом значений/размеров аргументов) и **немонотонные** (время исполнения с ростом значений/размеров аргументов меняется в произвольном порядке). Авторы утверждают, что их метод дает хорошие результаты на первых двух классах, однако на третьем может возникать существенная ошибка.

Метод заключается в том, чтобы перебирать аргументы метода, измерять его время исполнения при фиксированном наборе, а затем снизить размер полученных данных путем создания гистограмм.

Затем, чтобы получить стоимость исполнения метода, достаточно

идентифицировать клетку гистограммы, в которую попадает данный метод с данными аргументами.

Данная работа проделана исключительно для пользовательских функций и методов в СУБД, однако демонстрирует интересный подход к оценке времени исполнения путем "микробенчмаркинга"и использования полученных результатов для генерации стоимости.

1.3.4 Самонастраивающееся распределение операций СУБД на гибридных платформах

В [3] строится модель для гетерогенной системы. В данной работе рассматривается ситуация, когда в системе используется графический ускоритель и центральный процессор.

Методология строится на выделении набора алгоритмов AP_O для данной операции O в реляционной СУБД. Каждый алгоритм A_i имеет устройство для исполнения (центральный или графический процессор). Система собирает статистику о времени исполнения алгоритмов раз в какое-то время, а также сохраняет полученные данные для алгоритма A_i, если он был выбран для исполнения операции. Во избежание большого количества используемой памяти задается ограничение на хранимые данные. Выбор алгоритма осуществляется на аппроксимирующей оценке времени исполнения и берется алгоритм с наименьшей оценкой.

Также в работе аппроксимирующая функция пересчитывается не каждый раз при добавлении нового результата, а лишь с фиксированной периодичностью.

Такой подход интересен тем, что позволяет неявно использовать в оценке широкий спектр влияющих на производительность аппаратных характеристик. Причем важно отметить, что помимо статических факторов, также оцениваются и динамические, такие как загруженность устройств.

У данного подхода, однако, присутствуют определенные недостатки:

1. Для успешного начала работы системе необходимо несколько итераций на калибровку.

2. Так как акцент работы, в том числе сделан на учет нагруженности устройств, возникают дополнительные затраты на обработку новых данных и их сбор.

3. Интеграция данной системы в СУБД с генерацией исполняемого кода запроса может повлечь определенные трудности, так как заранее зафиксированный набор алгоритмов отсутствует, а оценки, сделанные в операторной модели, будут неверны .

4. Разработанная методология позволяет выбрать устройство, но не распределить работу в какой-то пропорции между ними.

1.4. Выводы

Суммируя все вышеописанные проблемы существующих работ, можно вывести основные требования к разрабатываемому решению:

1. **Генерация стоимости для гетерогенных планов** – главное требование к разрабатываемой модели затрат, так как основное назначение – оптимизация плана с точки зрения распределения исполнения между устройствами.

2. **Гибкость при добавлении устройств с новой архитектурой** – необходимо иметь возможность перестраивать систему для добавления новых устройств. При этом процесс перекалибровки не должен быть затратным.

3. **Возможность использования системы в СУБД с кодогенерацией** – модель затрат должна иметь возможность генерировать стоимость, которая будет верна для случая использования кодогенерации запросов.

При этом можно выделить интересные подходы у существующих методологий. Помимо аналитических моделей и методов машинного обучения, присутствуют подходы, использующие бенчмаркинг. Выделяется объект для генерации стоимости (операция в СУБД, пользовательские мето-

ды), выделяются характеристики, влияющие на производительность системы (размер входных данных, размер промежуточных данных) и производится сбор статистики с перебором возможных характеристик. Затем для подсчета стоимости используются собранные данные, соответствующие характеристикам оцениваемого случая.

2. Гетерогенные системы

В данной главе будет рассказано об основных различиях и особенностях центрального процессора и графического устройства, а затем об основных инструментах разработки гетерогенных систем.

2.1. Центральный процессор

Первым коммерческим микропроцессором является Intel 4004, выпущенный в 1971 году. 70-е можно считать точкой начала истории современных микропроцессоров.

С этого момента множество характеристик центральных процессоров стремительно улучшались, в том числе тактовая частота. Однако к 2000-м годам стало понятно, что бесконечно улучшать частоту не выйдет. С ростом этого показателя в значительной степени возрастало тепловыделение устройства, что требовало значительных затрат на охлаждающие системы. В связи с этим началось развитие такого понятия, как *параллелизм на уровне команд* (instruction-level parallelism, ILP); а затем коммерческие процессоры для персональных компьютеров стали получать дополнительные ядра, что открывало возможность работать с многопоточным исполнением.

Центральный процессор является вычислительным устройством общего назначения и решает широкий спектр задач: исполнение кода операционной системы, шифрование данных, исполнение бизнес логики и т.д. С целью оптимизации для многих задач в ЦП внедряются аппаратные модули, ускоряющие исполнение.

2.2. Графический процессор

С повышением требований программного обеспечения начали возникать новые микроархитектурные специализации как способ оптимизации вычислений. Одна из областей, требовавших ускорения исполнения, – отрисовка трехмерной геометрии (*рендеринг*).

Сначала для оптимизации выполнения были сделаны специальные

3D ускорители (например S3 ViRGE, Voodoo Rush). Затем появились более гибкие устройства, позволяющие программировать некоторые стадии графического конвейера.

Графический процессор спроектирован таким образом, чтобы делать большое количество однообразных вычислений над разными данными. Это продиктовано запросами отрисовки геометрии.

Графическое устройство позволяет решать более широкий спектр задач – задач обладающих *массовым параллелизмом*. Для реализации алгоритмов, решающих задачи, обладающие массовым параллелизмом, существуют специальные стандарты (например CUDA [22] или OpenCL [23]).

2.3. Различие центрального и графического процессоров

2.3.1 Архитектура

В то время как современный центральный процессор представляет собой набор самостоятельных ядер, ядра графического процессора объединены в группы – *потоковые мультипроцессоры*.

В рамках одного потокового мультипроцессора помимо самих ядер содержатся общая локальная память, вычислительные единицы специальных функций (например sin или cos), аппаратные единицы для работы с памятью и модули, отвечающие за обработку инструкций.

Важной особенностью является то что ядра в рамках одного потокового мультипроцессора обладают общим указателем на текущую инструкцию. Это приводит к определенным ограничениям. Например, операторы управления потоком исполнения могут значительно снизить производительность, так как для имитации параллельного вычисления потоки, для которых предикат условного оператора неверен, будут неактивны, и обработка разных условных ветвлений будет происходить последовательно.

2.3.2 Задержка и пропускная способность. Программные модели

Центральный процессор нацелен на снижение задержки между вызовом некоторой операции или вводом данных и получением конечного результата. Это касается различного рода задач, так как ЦП – устройство общего назначения. Это достигается различными способами – аппаратные ускорения, повышенная тактовая частота, параллелизм на уровне команд и т.д. С появлением многоядерных центральных процессоров появилась возможность разбивать исполнение на разные потоки, в рамках которых можно решать независимые (или отчасти независимые) задачи, исполнять разные части бизнес-логики приложений и т.д.

Однако, несмотря на возможность параллельных вычислений на ЦП, стоит отметить, что количество потоков, которые можно использовать для оптимизации времени исполнения, ограничено, и слишком большое число потоков может негативно сказаться на производительности.

Одна из главных причин деградации производительности – дорогое обслуживание потоков. Одной из самых дорогих операций на центральном процессоре является смена контекста потоков [24]. К тому же большое количество потоков может сильно нагрузить операционную систему.

Таким образом программная модель разработки ПО под ЦП оптимизирована под снижение задержки, но имеет крайне малую пропускную способность (*throughput*).

Графические процессоры напротив, обладают высокой пропускной способностью и способны запускать большое количество потоков. Программная модель при разработке под графическое устройство основана на *параллелизме на уровне данных*. Это означает, что типичная программа, исполняющаяся на графическом устройстве, как правило, запускает большое количество потоков, параллельно исполняющих одинаковые операции над разными фрагментами данных.

Однако индивидуально взятое ядро графического процессора будет значительно уступать ядру центрального процессора в производительности. Графические ядра обладают более низкой частотой и, как правило,

проще устроены, нежели ядра ЦП. Это означает, что последовательное исполнение на графическом устройстве будет уступать по времени исполнения ЦП, так как все преимущество графического процессора по сравнению с центральными процессорами в массовом параллелизме, который недостижим на ЦП.

2.3.3 Устройство памяти

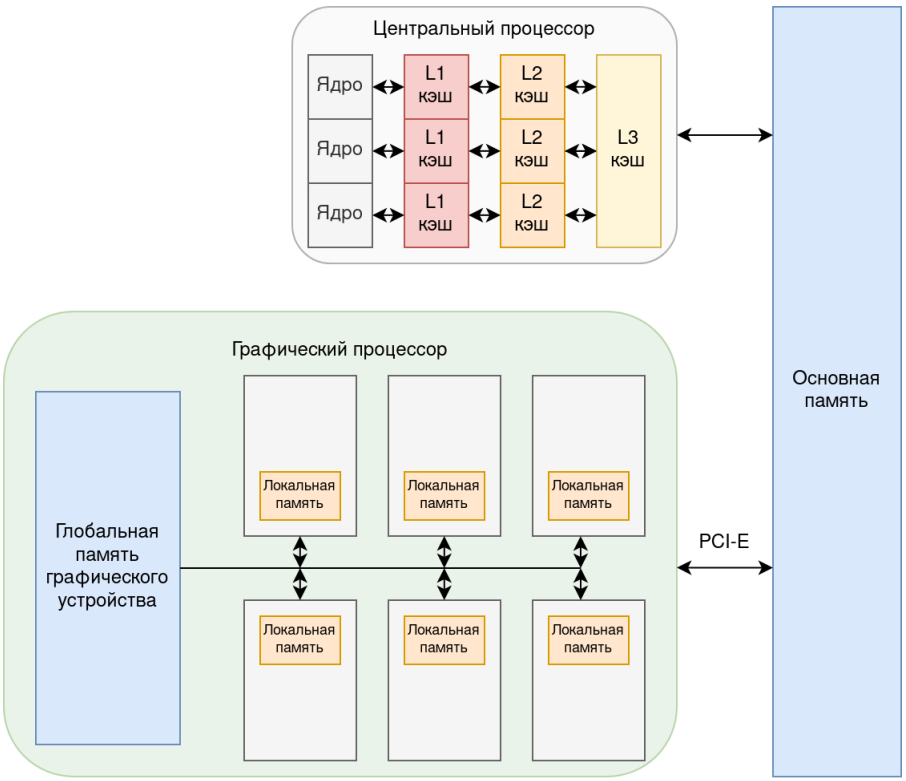

Рис. 2: Типичная топология памяти в современном ПК.

Архитектура памяти тоже влияет на подход к написанию кода под центральный процессор и графическое устройство. При использовании ЦП в распоряжении разработчика из энергозависимой памяти регистры, несколько уровней кэша (которые, как правило, напрямую не эксплуатируются) и ОЗУ. С появлением графического ускорителя иерархия памяти становится сложнее: потоковые мультипроцессоры графического устройства также содержат регистры, имеют локальную память, которая схожа по назначению с кэшем процессора (ее как раз в процессе разработки приходится использовать вручную), появляется глобальная память графического процессора,

и все эти элементы обладают разными пропускными способностями и размерами. Также появляются шины для подключения устройства в систему, которые тоже имеют ограничения на пропускную способность.

Глобальная память графического процессора, как правило, превосходит основную память в пропускной способности. Например, в 2020 была выпущена спецификация к пятому поколению ОЗУ DDR5. Из спецификации следует, что теоретическая пропускная способность памяти достигает 64 Гб/сек [25]. В том же году был анонсирован и выпущен ускоритель *A100* [27]. Заявленная пропускная способность глобальной памяти составляет 2039 Гб/сек.

Также стоит отметить, что хоть глобальная память графических устройств превосходит скорость основной памяти, ее размер сильно ограничен по сравнению с оперативной памятью.

Причины таких характеристик понятны: так как графические устройства строят свою модель вычислений на *параллелизме на уровне данных*, необходимо оборудовать устройство соответствующей памятью, способной обеспечить качество исполнения. Но в то же время чем быстрее память, тем она дороже, поэтому ее размеры часто оказываются значительно меньше.

Немаловажной частью модели памяти компьютерной системы с графическим устройством является шина подключения этого устройства (например, PCI-e). Заявленная скорость *A100* при использовании PCI-e – 64 Гб/сек. Стоит отметить, что на данный момент последним стандартом PCI-e является PCI-e 7.0, теоретическая пропускная способность которого может достигать 512 Гб/сек [26]. Чтобы избежать потери производительности из-за PCI-e шины, существуют высокопроизводительные шины. В случае *A100* это *NVLink* [36]. Заявленная пропускная способность при использовании NVLink – 600 Гб/сек.

Пропускная способность шины подключения часто воспринималась и до сих пор воспринимается как потенциальное узкое место. Однако с появлением специальных шин наподобие NVLink и развитием PCI-e, шины подключения скорее начинают восприниматься как неотъемлемая часть системы, которую следует иметь в виду, нежели потенциальное место деградации производительности.

Также стоит отметить особенности работы с внутренней памятью графического процессора. При обращении потока в глобальную память графического устройства будет подгружена кэш-линия данных. Если потоки будут обращаться к последовательному участку памяти, то подгрузка данных может быть осуществима за одну транзакцию. Если потоки обращаются в глобальную память в случайном порядке, то в худшем случае транзакции будут исполняться последовательно и все потоки будут ждать исполнение последней подгрузки данных, так как они обладают одним общим указателем на текущую инструкцию.

Таким образом недоэксплуатация пропускной способности глобальной памяти путем сложных шаблонов доступа в память может в разы снизить производительность на графическом устройстве.

2.4. Стандарты для программирования гетерогенных систем

2.4.1 OpenCL

OpenCL [23] – это открытый стандарт для программирования с использованием аппаратных ускорителей. Этот стандарт предоставляет унифицированные абстракции, что позволяет писать код единообразно для любого устройства, поддерживающего OpenCL.

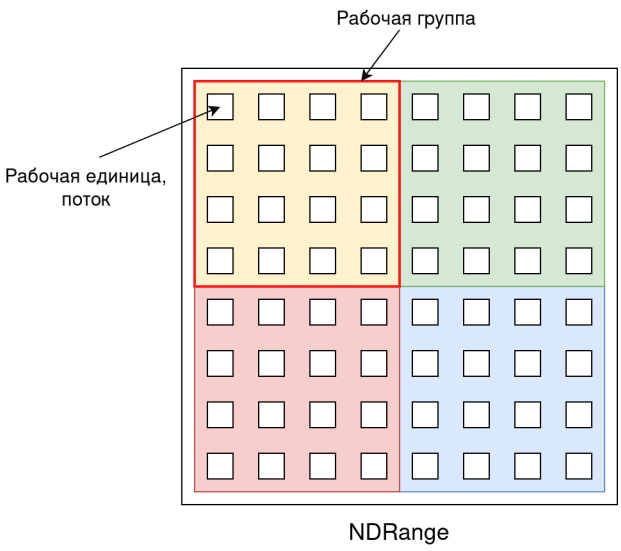

Рис. 3: Модель вычислений в OpenCL.

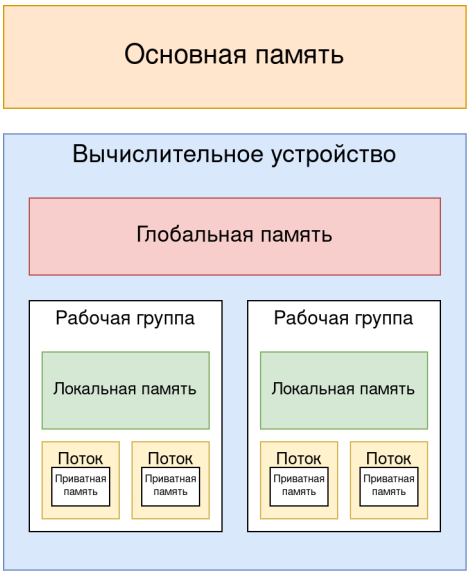

Рис. 4: Модель памяти в OpenCL.

Стандарт воспринимает аппаратные ускорители как устройства с набором вычислительных единиц, каждая из которых содержит в себе массивы вычислителей. Каждая вычислительная единица содержит в себе локальную память, которую можно использовать при написании алгоритмов. Каждое устройство также содержит в себе глобальную память, данные в которую подгружаются из основной памяти.

Потоки в OpenCL объединяются в рабочие группы. Они могут пользоваться общей быстрой локальной памятью и использовать барьеры для синхронизации исполнения. Рабочие группы в совокупности представляют собой *NDRange*. Он, как и рабочие группы, может быть одномерным, двумерным и трехмерным.

Синтаксически язык OpenCL является подъязыком ISO C99, но с определенными расширениями для параллелизма [28].

Компиляция OpenCL C происходит в две стадии: *offline* и *online*. Offline компиляция транслирует код в бинарный формат, затем на стадии online компиляции, бинарный код отдается драйверу устройства [28].

2.4.2 SYCL

Одна из проблем OpenCL – язык опирается на C-подобные синтаксис и семантику. При разработке на C++ всегда есть желание воспользоваться современными инструментами языка: шаблонами, RAII, ООП парадигмой. Эту проблему решает *SYCL* [29].

SYCL можно считать *высокоуровневой оболочкой*, позволяющей решать ту же задачу, что и OpenCL – разрабатывать приложения для гетерогенных систем. SYCL позволяет использовать для написания ISO C++ и предоставляет абстракции для работы с устройствами системы, управления ресурсами, подгрузки данных.

SYCL в первую очередь стандарт, он имеет несколько реализаций:

1. **DPC++** (*Data Parallel C++* [30]) – это реализация SYCL, разрабатываемая компанией Intel. DPC++ использует LLVM [31] для компиляции: код устройства переводится в LLVM IR, а затем в SPIR-V. После этого бинарный формат отдается драйверу устройства и пере-

водится в машинный код. Также DPC++ может использовать CUDA для исполнения на графических устройствах NVIDIA и Level Zero [17] для графических процессоров Intel.

2. **ComputeCpp** [19] – это реализация SYCL, разрабатываемая компанией Codeplay. Данная реализация так же использует SPIR-V для работы на большинстве графических устройств. Для работы с графическими процессорами NVIDIA используется NVPTX [18].

3. **OpenSYCL** [20] – данная реализация SYCL также использует CUDA для графических устройств NVIDIA и Level Zero для ГП Intel. Для графических процессоров AMD используется ROCm [21] – программный продукт компании AMD для разработки на графических устройствах.

2.5. Выводы

Для каждого устройства можно выделить основные типы операций и алгоритмов, которые будут быстрее исполняться на нем.

Для **центрального процессора** это:

1. Последовательные алгоритмы, обладающие сложными ветвлениями потока исполнения (Раздел 2.3.1 и Раздел 2.3.2).

2. Алгоритмы со случайным обращением в память (Раздел 2.3.3).

3. Алгоритмы с низким уровнем параллелизма (Раздел 2.3.2).

Для **графического устройства** это:

1. Алгоритмы, обладающие массовым параллелизмом (Раздел 2.3.2).

2. Алгоритмы с параллелизмом на уровне данных (Раздел 2.3.2).

3. Алгоритмы без случайных обращений в память (Раздел 2.3.3).

3. Аналитические шаблоны и Dwarf Bench

Для успешной оценки времени исполнения запроса, согласно прошлой главе, необходимо иметь возможность для извлечения информации об алгоритмах, написанных для графического устройства и центрального процессора. В общем случае проблема состоит в том, что потенциально алгоритмов может быть неограниченное количество. Однако в СУБД можно выделить конечное число алгоритмов, время исполнения которых можно будет использовать для оценки всех остальных случаев. Для этого можно заметить, что аналитические запросы можно разбить на *аналитические шаблоны*, каждый из которых можно исполнить конечным количеством алгоритмов [4].

В таком случае задачу оценки времени исполнения гетерогенного запроса можно свести к задаче оценки времени исполнения аналитических шаблонов. Для этого достаточно будет декомпозировать запрос на шаблоны, оценить каждый в отдельности и просуммировать.

В данной главе будут описаны аналитические шаблоны и представлен подход к измерению их времени исполнения.

3.1. Аналитические шаблоны

Можно выделить несколько основных аналитических шаблонов [4]:

1. Scan-filter – последовательное обращение к памяти одного или нескольких столбцов с применением предиката;
2. Hash build – вычисление хеша и построение хеш таблицы по записям таблицы;
3. Hash probe – поиск значения в построенной хеш таблице по ключу;
4. Hash-join – совместное применение Hash build и Hash probe (используется в основном для исполнения оператора *join*);
5. Group-By – группировка данных по заданному предикату;

6. Group-By-Aggregate – разделение данных на группы с последующим применением функции агрегации;

7. Sort – перестановка записей таблицы, заданная произвольным компаратором;

8. Reduction – применение функции агрегации к участку данных.

3.2. Dwarf Bench

Dwarf Bench – разработанная в рамках данной работы библиотека, реализующая основные аналитические шаблоны для исполнения на центральном процессоре и графическом устройстве. Для реализации используется OpenCL и SYCL (DPC++), что позволяет писать платформонезависимый код, способный исполняться на всех устройствах, которые поддерживают эти стандарты. Таким образом модуль может быть распространен не только на ЦП и графическое устройство, но и, например, на ПЛИС.

Dwarf Bench для каждого аналитического шаблона реализует ядро исполнения, которое может быть исполнено как на ЦП, так и на графическом устройстве. Это достигнуто благодаря вышеупомянутым стандартам.

3.3. Детали реализации

3.3.1 Scan-filter

Для симуляции шаблона scan-filter использовалась функция стандартной библиотеки DPC++ `std::copy_if`. Для каждого элемента вычисляется его индекс в массиве результатов (если предикат верен для этого элемента). Алгоритм построен следующим образом:

1. Входной массив разбивается на группы. Для каждого элемента высчитывается результат предиката и записывается в маску – массив бит, где i-тый бит возведен, если предикат верен на i-том элементе.

2. Маска и входной массив разбиваются на группы. В рамках каждой группы на маске считается локальная префиксная сумма.

3. На полученных группах считается глобальная префиксная сумма.

4. Берется элемент из входного массива. Если предикат для него верен (в маске возведен соответствующий бит), то по глобальной и локальной префиксным суммам можно высчитать его индекс (по сути это будет отступ относительно других элементов с верным предикатом).

3.3.2 Sort

Реализация шаблона Sort использует функцию стандартной библиотеки DPC++ `std::sort`. Как правило, классическим алгоритмом для реализации сортировки на графическом устройстве является поразрядная сортировка [15] [9] [33]. Почти все подходы к реализации так или иначе содержат три основных шага:

1. **Подсчет количества элементов** – алгоритм исполняется в несколько итераций и каждая итерация происходит над фиксированным количеством бит; таким образом подсчитывается количество для каждого возможного значения.

2. **Префиксная сумма** – над собранной статистикой считается префиксная сумма. Это позволит узнать, сколько элементов, меньших чем данное, содержится в массиве, и использовать для вычисления индекса элемента.

3. **Перестановка элементов** – основываясь на префиксных суммах высчитывается индекс элемента. Причем необходимо также учесть элементы, равные по значению. Как правило, для этого осуществляется дополнительная локальная сортировка (она должна быть стабильной, чтобы обеспечить корректность поразрядной сортировки).

При этом важно отметить, что в СУБД сортировка не обязательно переставляет записи напрямую. Зачастую выходными данными сортировки являются индексы записей.

Также стоит учесть сложность компараторов и их влияние на производительность. Однако в Dwarf Bench сортировка происходит с использованием оператора "меньше".

3.3.3 Reduction

Reduction реализован с использованием реализованных языком DPC++ шаблонов гетерогенных алгоритмов, а именно редукцией.

Один из самых популярных подходов к реализации этого шаблона – древовидная редукция с использованием локальной памяти. В стандартной библиотеке DPC++ используется именно эта реализация.

Algorithm 1: Алгоритм древовидной редукции с использованием локальной памяти

Data: $local\_id$ // Локальный индекс потока в рамках рабочей группы
Data: $global\_id$ // Глобальный индекс потока
Data: wg_size // Размер рабочей группы
Data: $data, size$ // Массив данных и его размер
Result: $dest$ // Адрес результата
$local\_data[local\_id] \leftarrow data[global\_id]$; // Размер локальных данных равен размеру рабочей группы
$barrier()$;
for $level \leftarrow wg\_size$ ***to*** 1 ***by*** $level \leftarrow \frac{level}{2}$ **do**
  if $2 \cdot local\_id < level$ **then**
    $local\_data[local\_id] = local\_data[local\_id] + local\_data[local\_id + \frac{level}{2}]$;
  end
  $barrier()$;
end
if $local\_id == 0$ **then**
  $atomic\_add(dest, local\_data[0])$;
end

3.3.4 Hash-build, Hash-probe, Hash-join

В Dwarf Bench для реализации Join используется Equi join – когда предикат для соединения таблиц задается равенством значений столбцов.

Алгоритм для шаблона Hash-join имеет несколько стадий:

1. Заполняется буфер ключей, получаемых из первой таблицы.
2. Для каждого ключа подсчитывается количество значений, имеющих этот ключ.
3. Считается префиксная сумма на буфере количеств значений. Таким образом в результате получается массив отступов, которые нужно сделать для группы значений, имеющих конкретный ключ.
4. Строится массив, в котором последовательно перечисляются значения первой таблицы (а именно их индексы), согласно буферам отступа и количества.
5. Далее из второй таблицы осуществляется нахождение элементов по ключу. Для каждого ключа можно восстановить множество значений с этим ключом – из буфера префиксной суммы можно получить отступ в массиве индексов, а количество значений можно получить через буфер количества значений. Сами значения после этого можно получить из построенного массива индексов.

Стоит отметить, что данные, полученные при исполнении Hash-join разделяются на время построения хеш-таблицы и время нахождения соответствующих записей в этой хеш-таблице. Таким образом можно отдельно получать время исполнения Hash-build и Hash-probe.

3.3.5 Group-By и Group-By-Aggregate

Для реализации группировки данных используется хеш-таблица с открытой адресацией. Вместо записи значения, значение прибавляется к накопленному по заданному ключу. Таким образом в данной реализации фиксируется функция агрегации – суммирование.

Также доступна реализация с распределением данных между потоками, каждый из которых строит локальную хеш-таблицу, которые затем агрегируются в финальный результат.

3.4. Калибровка Dwarf Bench

Один из недостатков подхода – разница между результатами, предоставляемыми Dwarf Bench, и реальным временем исполнения аналитических шаблонов. Так может происходить из-за определенных упрощений, допущенных при реализации, с целью унифицировать алгоритмы для разных устройств.

Причина, по которой это является проблемой, – получаемая оценка времени исполнения используется для оптимизации гетерогенного плана. Если оценки будут сильно отличаться от реальных результатов, то оптимизация плана потенциально может часто выдавать неудовлетворительный по качеству план.

Чтобы ограничить ошибку результатов Dwarf Bench, после реализации основных аналитических шаблонов была проведена калибровка с реальной СУБД. Для этого использовались данные и запросы из системы бенчмаркинга TPC-H [37]. Брались конкретные запросы и разбивались на аналитические шаблоны. В СУБД исполнялись запросы, а в Dwarf Bench соответствующие аналитические шаблоны.

При наличии сильного различия результатов проводился сравнительный анализ для достижения лучших результатов. Он включал в себя как сравнение самих алгоритмов в СУБД и Dwarf Bench, так и генерируемое промежуточное представление компилятора (LLVM IR).

Таким образом некоторые из описанных алгоритмов были модифицированы или переписаны в процессе разработки.

3.5. Применение аналитических шаблонов и Dwarf Bench

Dwarf Bench является своего рода универсальной функцией, которая может выдавать какую-то оценку на время исполнения аналитических шаблонов. Основываясь на этом, можно построить модель затрат, которая будет генерировать стоимость используя получаемую оценку. Для этого необходимо лишь разбить план запроса на аналитические шаблоны, брать оценку для каждого из Dwarf Bench и агрегировать результат.

Для успешного применения Dwarf Bench необходимо решить следую-

щие задачи:

1. Экстраполировать данные, полученные из Dwarf Bench.
2. Построить алгоритм разбиения запроса на аналитические шаблоны.
3. Реализовать оптимизацию плана запроса.

Важно отметить, что подход с бенчмаркингом призван удовлетворить основные требования к модели затрат, которые были описаны в начале работы:

1. Подход с измерением времени исполнения аналитических шаблонов позволяет неявно собирать информацию о микроархитектурных особенностях устройств и предоставляет оценку на время исполнения, оптимизация которой должна приводить к оптимальному распределению работы.
2. При добавлении нового устройства в систему достаточно лишь запустить процесс бенчмаркинга на новом ускорителе и перекалибровать систему. При этом данный процесс занимает значительно меньше времени, чем ручная калибровка или переобучение нейронной сети. Таким образом возможно добавить любую архитектуру устройств, если они совместимы с языком SYCL, что гарантирует запрашиваемую гибкость.
3. Аналитические шаблоны выделены таким образом, чтобы покрывать план с учетом конвейерной модели и компиляции запросов. Исследования показали, что основной вклад в производительность вносит характер шага исполнения и размер обрабатываемых данных [4], поэтому для успешной оценки очередного конвейера нужно лишь выделить основной соответствующий ему шаблон (или несколько шаблонов).

Подробнее о том, как Dwarf Bench может быть применен в реальной гетерогенной СУБД, будет рассказано в следующей главе.

4. HDK и Модель затрат

В данной главе будет рассказано о том, как модуль Dwarf Bench может быть использован для построения модели затрат с целью оптимизации гетерогенного исполнения на примере библиотеки HDK.

4.1. Обзор HDK

HDK [34] является низкоуровневым движком для исполнения аналитических операций над данными. Проект является наследником реляционной базы данных OmniSciDB [35], позволяющей исполнять запросы не только на ЦП, но и на графическом устройстве.

Параллелизм на уровне данных достигается за счет разделения таблиц на фрагменты [13]. Фрагменты могут распределяться между устройствами и параллельно обрабатываться. Агрегация конечных результатов (за исключением промежуточных) происходит на центральном процессоре.

Код для исполнения аналитических запросов генерируется, используя LLVM и SPIR-V. Подобный подход уже был отмечен в стандартах SYCL и OpenCL. Таким образом удается единообразно генерировать промежуточное представление и для ЦПУ и для графического ускорителя.

Исполнение в HDK поддерживается в двух основных режимах: пользователь может выбрать конкретное устройство, на котором необходимо исполнить результат, а может выбрать гетерогенное исполнение и задать пропорцию, в которой необходимо распределить данные.

4.2. Модель затрат и интеграция Dwarf Bench

Модель затрат инициализируется в несколько этапов:

1. **Сбор данных** – сначала собираются данные о времени исполнения. В данном случае источником данных является Dwarf Bench, однако им может выступать любая произвольная сущность.

2. **Экстраполяция данных** – после сбора данных, их необходимо экстраполировать для любого размера входных данных.

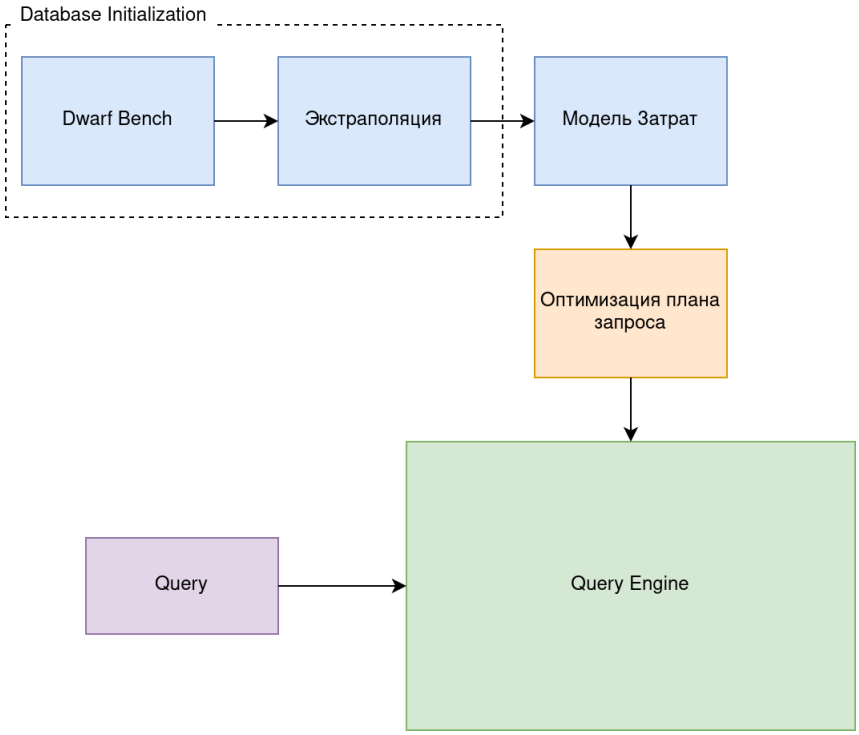

Рис. 5: Высокоуровневая схема модели затрат.

Оценка стоимости и оптимизация происходят по ходу исполнения запроса. Каждый раз, когда движок запросов выделяет очередной шаг исполнения, модель затрат разбивает его на аналитические шаблоны и дает оценку, основываясь на экстраполированных данных. Таким образом нет необходимости каждый раз запускать бенчмаркинг аналитических шаблонов – достаточно собрать информацию один раз при калибровке системы.

На основе полученных оценок на время исполнения оптимизируется пропорция, с которой будет исполнен данный шаг.

Далее каждая часть реализации будет обсуждена более детально.

4.2.1 Сбор данных

С помощью предоставленного Dwarf Bench программного интерфейса движок делает запрос на исполнение основных аналитических шаблонов на выбранных устройствах с разными размерами данных. Полученные результаты экстраполируются и сохраняются в хеш-таблицу с шаблоном и устройством в качестве ключа.

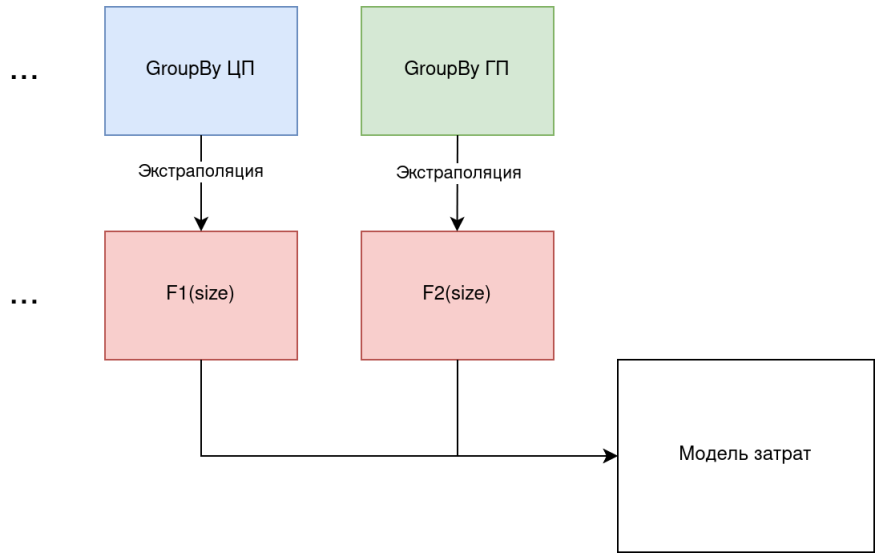

Рис. 6: Схема сбора данных о времени исполнения аналитических шаблонов.

4.2.2 Экстраполяция

Экстраполяция данных необходима для получения оценки для любого размера данных и заключается в построении функции $F : \mathbb{N} \to \mathbb{R}$, получающей на вход размер данных и возвращающей оценку на время исполнения.

В качестве подхода к экстраполяции данных была выбрана *линейная регрессия*. В общем случае экстраполяция линейной функцией является некорректной, однако на рассматриваемых размерах и запросах (время выполнения которых определяется свободной памятью, необходимой для хранения рабочих данных; *memory-bound*) линейная модель, согласно проведенным экспериментам, является хорошей аппроксимацией [4].

Линейная регрессия в данной реализации характеризуется одним признаком – размером данных. Формально говоря полученная модель имеет вид

$$t(s) = w_0 + s \cdot w_1 = x^T w, \quad x = (1, s)$$

В перспективе данной модели можно добавить дополнительных параметров, влияющих на время исполнения, но в таком случае необходимо провести дополнительное исследование, ведь от новых параметров время

исполнения не обязательно будет вести себя линейно.

Процесс построения функций линейной регрессии происходит для каждого аналитического шаблона и для каждого его запуска на устройствах.

Затем полученные экстраполированные данные передаются в модель затрат.

4.2.3 Выделение аналитических шаблонов и оценка времени исполнения

Как уже было отмечено ранее, основной вклад в производительность вносит характер шага исполнения и размер обрабатываемых данных. Например, добавление еще одной агрегации в шаблон GroupBy будет влиять на время исполнения исключительно при добавлении нового аттрибута (столбца). Это так, потому что дополнительная функция агрегации над уже присутствующими данными в запросе внесет дополнительные арифметические операции, которые по сравнению с затратами на обращение в память незначительны [4]. Тем не менее это верно в случае, если их количество мало – при неограниченном росте вклад может оказаться серьезным, что потенциально является причиной добавить параметризацию шаблона количеством агрегационных функций.

Таким образом при получении на вход шага исполнения достаточно выделить основной аналитический шаблон и получить на вход размер данных. К примеру шаблон scan-filter учитывается лишь если присутствует самостоятельно. В противном случае обращение к данным будет уже учтено объемлющим шаблоном (например GroupBy).

Для выделения шаблонов модель затрат проходит по узлам дерева, соответствующим шагу исполнения, и выделяет аналитические шаблоны согласно вышеописанным рассуждениям.

Оценка времени происходит путем сложения оценок для каждого шаблона в шаге исполнения.

4.2.4 Оптимизация гетерогенного плана

Для проведения экспериментов с разработанным подходом было необходимо реализовать оптимизацию запроса с использованием оценок модели затрат. В качестве подхода к поиску оптимальной пропорции для шага исполнения был выбран следующий алгоритм:

1. Пространство размера данных делится на равные части. Таким образом получается набор потенциальных пропорций.
2. Возможные пропорции перебираются и оцениваются моделью затрат.
3. В качестве выбранной пропорции выбирается пропорция с наименьшей стоимостью.

Algorithm 2: Алгоритм оптимизации пропорции

Data: $C()$ // Модель затрат. Формально - функция от размера данных, набора шаблонов и устройства
Data: $size$ // Размер входных данных
Data: $templs$ // Аналитические шаблоны
Data: $iters$ // Количество итераций / потенциальных пропорций
Result: $best\_cpu\_prop$, $best\_gpu\_prop$ // Соответствующие пропорции

$best\_prediction \leftarrow \infty$;
$best\_cpu\_prop \leftarrow 0$;
$best\_gpu\_prop \leftarrow 0$;
$opt\_step \leftarrow \lceil \frac{size}{iters} \rceil$;
for $cur\_size \leftarrow 0$ ***to*** $size$ ***by*** opt_step **do**
    $cpu\_size \leftarrow cur\_size$;
    $gpu\_size \leftarrow size - cur\_size$;
    $cpu\_cost \leftarrow C(cpu\_size, templs, CPU)$;
    $gpu\_cost \leftarrow C(gpu\_size, templs, GPU)$;
    $cur\_prediction \leftarrow max(cpu\_cost, gpu\_cost)$;
    if $cur\_prediction < best\_prediction$ **then**
        $best\_prediction \leftarrow cur\_prediction$;
        $best\_cpu\_prop \leftarrow cpu\_size$;
        $best\_gpu\_prop \leftarrow gpu\_size$;
    end
end

5. Результаты

Результаты собирались в системе с центральным процессором Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz и графическим устройством NVIDIA GeForce GTX 1650 SUPER. В качестве реализации модели затрат использовалась описанная в прошлой главе модель в HDK.

Для проведения экспериментов использовалась система бенчмаркинга NYC Taxi. Все запросы являются агрегациями с вариацией столбцов, типов столбцов, наличием сортировки.

Всего проводилось два эксперимента: общее сравнение производительности и сравнение результатов при увеличении точности предсказаний.

В дальнейшем обозначение (c, g) будет обозначать пропорцию, где первое число – доля ЦП, а второе – графического устройства.

5.1. Общее сравнение производительности

В сравнении участвовали следующие конфигурации исполнения:

1. **Исполнение с использованием модели затрат** – запрос исполняется и распределяется между устройствами с помощью модели затрат, реализованной в прошлой главе.

2. **Исполнение на центральном процессоре** – запрос полностью исполняется на центральном процессоре.

3. **Исполнение на графическом процессоре** – запрос полностью исполняется на графическом процессоре.

4. **Исполнение с идеальной пропорцией** – запрос исполняется с пропорцией, на которой время исполнения минимально.

В качестве результатов рассматривается среднее время исполнения без учета выбросов, возникающих из-за инициализации движка.

Для отображения результатов идеальной конфигурации был произведен перебор пропорций и выбрана та, на которой среднее время исполнения минимально.

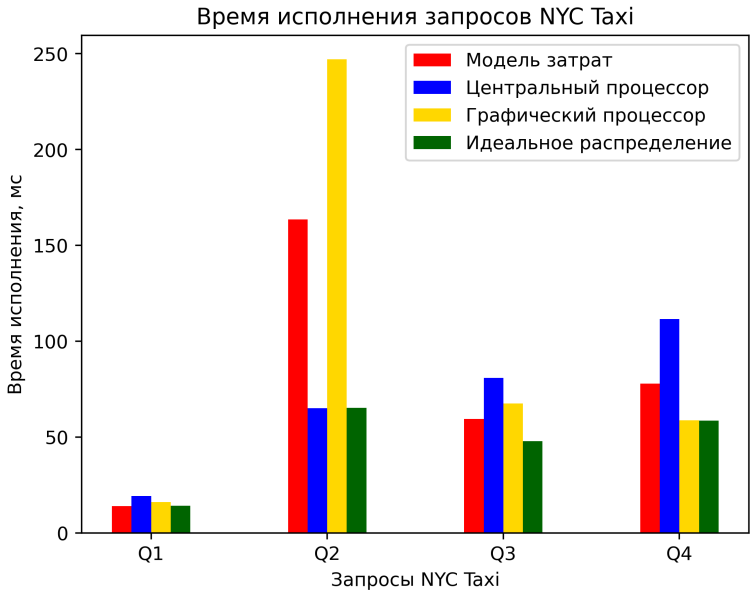

Рис. 7: Результаты времени исполнения с различными конфигурациями HDK.

На запросе Q1 модель затрат подобрала правильный результат – идеальной пропорцией является $(4, 6)$ и именно эта пропорция была получена при оптимизации запроса. Сам запрос Q1 представляет собой группировку по одному столбцу и подсчет количества данных в качестве агрегационной функции.

```
1 SELECT cab_type, count(*) as cnt
2     FROM trips
3     GROUP BY cab_type;
```

Запрос, соответственно, был разбит на один аналитический шаблон – Group By. С простым запросом, состоящим из одного аналитического шаблона, модель затрат справилась отлично.

Остальные запросы, помимо группировки данных, также сортируют их.

Запрос Q2 отличается от остальных запросов функцией агрегации. Так как это единственное отличие, вероятнее всего, что это является причиной низкой производительности на графическом устройстве.

```
SELECT  passenger_count,
        AVG(total_amount) as total_amount_avg
    FROM trips
    GROUP BY passenger_count
    ORDER BY passenger_count;
```

В таком случае можно сделать вывод, что на самом деле характер функции агрегации тоже может внести значительный вклад в производительность исполнения запроса. В силу того, что Dwarf Bench не позволяет варьировать функции агрегации, модели затрат не удалось выявить сложность запроса для графического устройства. Модель затрат определила идеальное распределение как $(4, 6)$, однако оптимальным было отправить запрос полностью на центральный процессор.

На запросе Q3 модель затрат позволяет получить результат, превосходящий наилучшее устройство для исполнения, несмотря на то что ей не удалось достичь идеальной пропорции. Оптимальным распределением в данном случае являлось $(2, 8)$, оптимизация запроса распределила работу в пропорции $(4, 6)$.

Этот результат совместно с результатом на запросе Q1 демонстрирует актуальность дальнейшего исследования в этом направлении, так как в определенных случаях данный подход проявляет себя неплохо.

На запросе Q4 модель затрат ошиблась с идеальной пропорцией: система выдала $(3, 7)$, в то время как оптимальной являлась $(0, 10)$. Но несмотря на это ошибка по времени исполнения не является критичной.

Можно отметить, что почти все предлагаемые моделью затрат пропорции похожи. Это как правило излишнее доверие исполнения графическому устройству и недостаточное центральному процессору. Наиболее вероятные причины такого поведения заключаются в следующем: все запросы NYC Taxi, использованные для получения результатов, характерно похожи между собой и различаются лишь размером данных. Как уже было отмечено, построенная модель затрат не различает дополнительные характеристики запросов, такие как функция агрегации. Поэтому различия между четырьмя запросами для нее минимальны.

5.2. Увеличение точности предсказаний

Зафиксируем какой-нибудь запрос и конкретный шаг исполнения в нем. Предположим, что реальное время исполнения шага – r, а результат, получаемый из Dwarf Bench – $r + \varepsilon$, где ε – это ошибка. Реальное время исполнения пропорции можно получить аналогично поиску идеальной конфигурации – перебрать пропорции.

В таком случае можно провести линейную интерполяцию данных из Dwarf Bench и реальных результатов. В построенной модели это будет соответствовать

$$(1 - t)(r + \varepsilon) + t \cdot r = r + (1 - t) \cdot \varepsilon$$

В таком случае параметр t отвечает как раз за точность предсказаний, что позволит изучить тенденцию времени исполнения запроса при понижении ошибки.

Однако стоит отметить, что важным являются относительные результаты, а не абсолютные. Поэтому интерполировались нормированные результаты. Более формально подбор наилучшего распределения с помощью модели затрат и список реальных результатов времени исполнения на разных пропорциях являются векторами v_{cm} и v_r соответственно. Вместо интерполяции $(1 - t) \cdot v_{cm} + t \cdot v_r$ использовалась интерполяция $(1 - t) \cdot \frac{v_{cm}}{||v_{cm}||} + t \cdot \frac{v_r}{||v_r||}$.

Эксперименты были проведены на запросах Q2, Q3 и Q4.

Запрос	0.0	0.2	0.4	0.6	0.8	1.0
Q2	(4, 6)	(4, 6)	(5, 5)	(8, 2)	(8, 2)	(10, 0)
Q3	(4, 6)	(4, 6)	(4, 6)	(3, 7)	(2, 8)	(2, 8)
Q4	(3, 7)	(3, 7)	(3, 7)	(3, 7)	(0, 10)	(0, 10)

Таблица 1: Изменение пропорций при увеличении точности.

При сравнении результатов между собой можно отметить, что на запросах, на которых модель затрат проявила себя хуже (Q2), время исполнения улучшается с более высоким темпом, чем на запросах, которые и без того хорошо были оптимизированы.

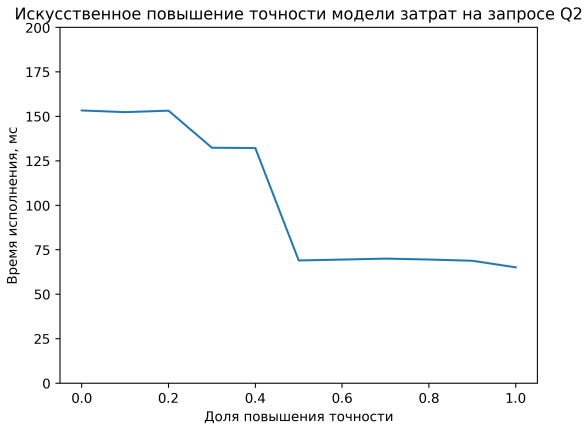

Рис. 8: Результаты увеличения точности на запросе Q2.

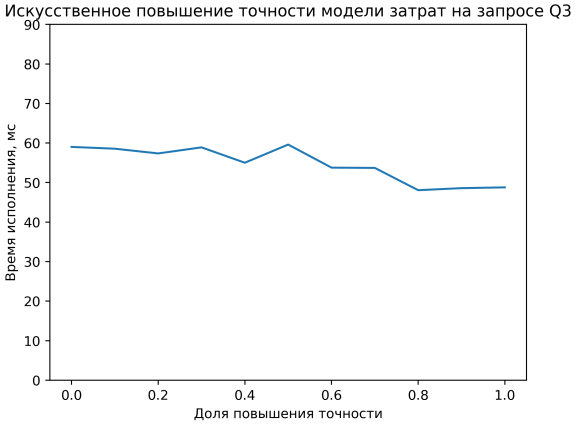

Рис. 9: Результаты увеличения точности на запросе Q3.

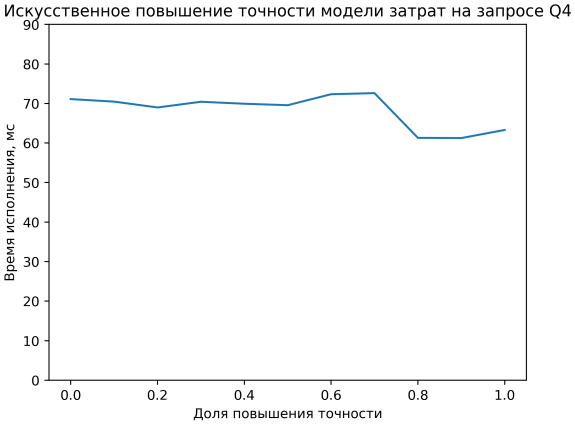

Рис. 10: Результаты увеличения точности на запросе Q4.

На запросе Q2 улучшение времени исполнения наблюдается начиная с повышения точности на 30%. При этом оптимизация ошибки в два раза позволяет получить практически наилучший результат с приростом производительности практически в 2.5 раза. Изменение пропорции с $(8, 2)$ до $(10, 0)$ не дало значимых результатов.

Запросы Q3 и Q4 улучшались с меньшими темпами. Существенный результат был достигнут лишь при оптимизации ошибки более чем в два раза.

При этом важным выводом также является способность разработанной модели затрат способствовать оптимизации времени исполнения вплоть до идеального или практически идеального. Причем результат этот дости-

жим при понижении ошибки менее чем на 100%, что можно видеть на запросе Q2.

Также стоит отметить, что были дополнительно проведены эксперименты с улучшением абсолютных значений. Однако в данном случае улучшение результатов происходило лишь при практически полном устранении ошибки – существенные улучшения возникали при параметре интерполяции равном 0.95 - 1.0. При этом оптимизация относительных результатов демонстрирует улучшение при уменьшении ошибок в два раза (параметр интерполяции равен 0.5), что может сказать о необходимости сосредоточить внимание именно на относительных результатах, нежели на абсолютных.

5.3. Выводы

Основные выводы из полученных результатов заключаются в следующем:

1. Полученная реализация модели затрат уже позволяет получить значительный прирост производительности.

2. Dwarf Bench нуждается в параметризации некоторых аналитических шаблонов.

3. При снижении ошибки относительных стоимостей модель затрат позволяет достичь оптимального времени исполнения. Причем получение таких результатов возможно без полной ликвидации ошибок генерируемых стоимостей.

4. Относительные результаты важнее абсолютных – несмотря на допускаемые моделью затрат ошибки, на запросе Q1 удается достичь оптимального решения засчет верных относительных результатов, несмотря на большие ошибки в абсолютных оценках на время исполнения. К тому же эксперимент с улучшением точности предсказаний показал, что оптимизация относительных ошибок лучше оптимизирует время исполнения, нежели оптимизация абсолютных оценок.

Заключение

Основные результаты работы заключаются в следующем:

1. Была разработана библиотека аналитических шаблонов.
2. Была разработана и реализована модель затрат в HDK, позволяющая оценивать стоимость планов в гетерогенной системе.
3. Разработанная модель затрат позволяет получать повышение производительности.
4. Была исследована зависимость времени исполнения от точности предсказаний модели затрат. На запросах, которые модель затрат оценивала хуже, замечено более активное улучшение результатов при повышении точности. Также отмечено, что модель затрат позволяет получать оптимальные результаты без полной ликвидации ошибок.

Таким образом в выпускной квалификационной работе проведено исследование влияния точности модели затрат на исполнение аналитических запросов к СУБД в основной памяти в гетерогенных системах, что является актуальным результатом для области анализа данных.

Благодарность

Автор выражает особую благодарность Курапову Петру Александровичу за вклад, наставления и советы при написании данной выпускной квалификационной работы.

Список литературы

[1] High Data Growth and Modern Applications Drive New Storage Requirements in Digitally Transformed Enterprises. URL: https://www.delltechnologies.com/asset/en-us/products/storage/industry-market/h19267-wp-idc-storage-reqs-digital-enterprise.pdf

[2] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? Proc. VLDB Endow. 9, 3 (November 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[3] S. Bress, et al., Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms, 2012

[4] Курапов П.А., Куликов Д.В., Мелик-Адамянх А.Ф. МОДЕЛЬ ЗАТРАТ ДЛЯ ОПТИМИЗАЦИИ АНАЛИТИЧЕСКИХ ЗАПРОСОВ В ГЕТЕРОГЕННЫХ СИСТЕМАХ // International Journal of Open Information Technologies. 2022. №4.

[5] Du, Weimin & Krishnamurthy, Ravi & Shan, Ming-Chien. (1992). Query Optimization in a Heterogeneous DBMS.. 277-291.

[6] Boulos, Jihad, Yann Viémont and Kinji Ono. “A Neural Networks Approach for Query Cost Evaluation.” (1997).

[7] Kurt Hornik, Maxwell Stinchcombe, Halbert White: Multilayer feedforward networks are universal approximators, Neural Networks, Volume 2, Issue 5, Pages 359-366 (1989)

[8] Jihad Boulos and Kinji Ono. 1999. Cost estimation of user-defined methods in object-relational database systems. SIGMOD Rec. 28, 3 (Sept. 1999), 22–28. https://doi.org/10.1145/333607.333610

[9] Harada, Takahiro and Lee W. Howes. “Introduction to GPU Radix Sort.” (2011).

[10] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs and J. F. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?,"2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, QLD, Australia, 2013, pp. 1081-1092, doi: 10.1109/ICDE.2013.6544899.

[11] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[12] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proc. VLDB Endow. 11, 13 (September 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

[13] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14). Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[14] Marius Hobbhahn and Tamay Besiroglu (2022), "Trends in GPU price-performance". Published online at epochai.org. Retrieved from: https://epochai.org/blog/trends-in-gpu-price-performance [online resource]

[15] Andy Adinets, Duane Merrill. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs https://doi.org/10.48550/arXiv.2206.01784

[16] PostgreSQL home page. URL: https://www.postgresql.org/

[17] Intel® oneAPI Level Zero. URL: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-1/intel-oneapi-level-zero.html

[18] NVPTX Back-end in LLVM. URL: https://llvm.org/docs/NVPTXUsage.html

[19] ComputeCpp. URL: https://developer.codeplay.com/products/computecpp/ce/home

[20] OpenSYCL repository. URL: https://github.com/OpenSYCL/OpenSYCL

[21] ROCm. URL: https://www.amd.com/en/graphics/servers-solutions-rocm

[22] CUDA Toolkit. URL: https://developer.nvidia.com/cuda-toolkit

[23] OpenCL. URL: https://www.khronos.org/opencl/

[24] Infographics: Operation Costs in CPU Clock Cycles. URL: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

[25] Smith, Ryan (July 14, 2020). "DDR5 Memory Specification Released: Setting the Stage for DDR5-6400 And Beyond". AnandTech. URL: https://www.kingston.com/en/blog/pc-performance/ddr5-overview

[26] PCI Express 7.0 Specification. URL: https://pcisig.com/specifications/pci-express-70-specification

[27] NVIDIA A100 Tensor Core GPU. URL: https://www.nvidia.com/en-us/data-center/a100/

[28] OpenCL Guide page. URL: https://github.com/KhronosGroup/OpenCL-Guide/

[29] SYCL. URL: https://www.khronos.org/sycl/

[30] DPC++. URL: https://www.intel.com/content/www/us/en/developer/videos/dpc-part-1-introduction-to-new-programming-model.html

[31] LLVM. URL: https://llvm.org/

[32] Intel® oneAPI Threading Building Blocks. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html

[33] oneAPI DPC++ Library (oneDPL). URL: https://github.com/oneapi-src/oneDPL

[34] HDK - Heterogeneous Data Kernels. URL: https://github.com/intel-ai/hdk

[35] HeavyDB (formerly OmniSciDB). URL: https://github.com/heavyai/heavydb

[36] NVIDIA NVLink. URL: https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/

[37] TPC-H home page. URL: https://www.tpc.org/tpch/