УДК 004.657
DOI: 10.25559/SITITO.17.202104.972-987

Научная статья

Оптимизация аналитических запросов в гетерогенных системах

П. А. Курапов[1*], А. Ф. Мелик-Адамян[2]

[1] ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация

Адрес: 141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9

[*] kurapov@phystech.edu

[2] Intel Corporation, г. Санта-Клара, Соединенные Штаты Америки

Адрес: CA 95054-1549, Соединенные Штаты Америки, шт. Калифорния, г. Санта-Клара, бульвар Мишн-Колледж, д. 2200

Аннотация

Согласно оценке International Data Corporation (IDC) в 2020 году человечество произвело более 64 зеттабайт данных. В ближайшие 5 лет ожидается рост этого показателя на 23% в год, что превышает скорость роста производительности аппаратуры. Агрегация и анализ больших данных потребовали развития таких технологий как Hadoop и Spark. С увеличением количества данных и сложности запросов их производительности оказывается недостаточно и требуются новые пути повышения эффективности анализа. Один из таких методов – использование специализированных аппаратных ускорителей, таких как графический процессор, для выполнения запросов и расширение иерархии используемой памяти. В статье рассматривается проблема оптимизации аналитических запросов к СУБД в основной памяти с помощью использования аппаратных ускорителей и расширения иерархии памяти. Дан обзор классических подходов к оптимизации запросов и текущего состояния области исследования гетерогенного исполнения. Разобраны достоинства и недостатки существующих решений и сформулированы нерешенные проблемы. Предложен вариант эталонной архитектуры.

Ключевые слова: СУБД, графический процессор, оптимизация аналитических запросов, обработка аналитических запросов, гетерогенные вычисления

Авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Курапов П. А., Мелик-Адамян А. Ф. Оптимизация аналитических запросов в гетерогенных системах // Современные информационные технологии и ИТ-образование. 2021. Т. 17, № 4. С. 972-987. doi: https://doi.org/10.25559/SITITO.17.202104.972-987

Original article

Optimization of Analytical Queries in Heterogeneous Systems

P. A. Kurapov[a*], A. F. Melik-Adamyan[b]

[a] Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation

Address: 9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation

[*] kurapov@phystech.edu

[b] Intel Corporation, Santa Clara, United States of America

Address: 2200 Mission College Blvd., Santa Clara CA 95054-1549, California, United States of America

Abstract

According to International Data Corporation (IDC) reports humankind produced more than 64 zettabytes of data in 2020. Global data creation and replication are expected to grow 23% annually between 2020 and 2025. Such a pace exceeds the rate of hardware performance improvement. Technologies such as Hadoop and Spark can not keep up, and new ways of increasing analytic queries performance and efficiency are required. One such technique is expanding the memory hierarchy and using specialized hardware for query processing. The article discusses query optimization problems for in-memory DBMSs using hardware accelerators. We give an overview of classical approaches to query optimization and state-of-the-art in heterogeneous query execution research. The advantages and disadvantages of existing solutions are analyzed, and gaps are identified. A reference architecture for heterogeneous query processing is proposed.

Keywords: DBMS, GPU, analytical query optimization, OLAP, analytical query processing, in-memory databases, heterogeneous query processing

The authors declare no conflict of interest.

For citation: Kurapov P.A., Melik-Adamyan A.F. Optimization of Analytical Queries in Heterogeneous Systems. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education*. 2021; 17(4):972-987. doi: https://doi.org/10.25559/SITITO.17.202104.972-987

Введение

Как показано в [1], оптимизация запросов для распределенных систем является NP-трудной задачей, а при использовании гетерогенных вычислителей, пространство поиска оптимального плана запроса расширяется, что требует разработки новых алгоритмов поиска эффективных планов исполнения запросов.

Существует 3 основных вида запросов:

- OLTP (online transaction processing) – обработка единичных повторяющихся транзакций, например, изменение состояния счета в банке через банкомат. Для этого типа запросов характерны доступ к малой части данных (обновление одной записи), постоянное внесение изменений и, как следствие, возникновение конфликтов синхронизации при параллельной обработке;
- OLAP (online analytical processing) – аналитические запросы с целью выявления дополнительной информации из существующих данных, например, средние квартальные значения выручки от продаж определенного вида товаров. Для этого вида характерны вычислительно-ёмкие (большое количество источников данных и таблиц, высокая вложенность соединений, большое количество и сложность предикатов) запросы с обращением к большой части данных, доступ к данным только на чтение (read-only), и, как следствие, независимость запросов;
- HTAP (hybrid transaction/analytical processing) [2] – гибридная обработка транзакционных малых изменений (OLTP) и аналитических запросов (OLAP). Для этого вида характерны оба паттерна описанные выше. Совмещение процессоров запроса[1] в одной системе позволяет управлять ресурсами для обеспечения необходимой актуальности данных для OLAP запросов.

Типичный OLAP запрос состоит из следующих концептуальных шагов:

- поиск данных – задача нетривиальна в распределенных и при наличии множества источников данных (например, системы управления отношения с клиентами, системы планирования ресурсов предприятия и т. п.);
- чтение данных – доступ к дисковым накопителям и передача по сети. Данные могут быть распределены и сегментированы (sharding) по вычислительным узлам кластера;
- валидация и приведение данных к необходимому виду – предобработка для последующего анализа или использования сторонних библиотек;
- применение анализа к данным и доставка результатов пользователю.

В этой статье мы фокусируемся на аналитических запросах (OLAP) к большим данным в различных типах хранилищ, а именно, на оптимизации вычислительной его части, когда данные уже загружены в память (например, DRAM). Мы приводим обзор подходов к оптимизации запроса для гетерогенных сред исполнения и выделяем нерешенные проблемы.

Статья организованна следующим образом. В главе 2 мы даем определения аналитическому запросу и гетерогенной системе. Глава 3 описывает основные подходы к оптимизации запросов на всевозможных уровнях: от планирования до повышения эффективности исполнения. Глава 4 дает обзор существующих исследований, прототипов и коммерческих решений использования гетерогенных систем для оптимизации исполнения аналитических запросов. Мы определяем многокритериальную задачу оптимизации запроса для гетерогенной системы и даем заключение в главе 5.

Аналитические запросы

Аналитика над данными выполняет функцию поиска паттернов, корреляций и моделей для предсказания будущего (спроса на продукт в следующем месяце, пройдет ли пользователь по рекламной ссылке, вернется ли выданный займ, легитимна ли банковская транзакция и т. п.) и использования этих данных для принятия бизнес-решений[2]. Основой построения паттернов могут служить как простейшие аналитические модели, например, средних значений интересующих величин, так и более сложные – алгоритмы машинного обучения, работающие с данными больших размерностей. В больших масштабах значение точности прогноза с увеличением объема обрабатываемых данных возрастает, поэтому использование всех доступных данных становится предпочтительным.

Аналитические запросы характеризуются сложными, долго работающими процессами, которые обрабатывают большой объем данных из разных источников. Например, агрегация поверх всей таблицы или слияние нескольких больших таблиц в плоскую денормализованную таблицу с последующей обработкой. Масштаб изменений достаточно обширен, включая одновременное добавление, удаление и обработку большого числа строк.

Поскольку в бизнесе источников данных может быть много, создание единого хранилища данных (data warehouse) для анализа зачастую необходимость. Процесс агрегации данных в хранилищах называется extract, transform and load (ETL[3]) и, как показано выше, является частью OLAP. В этом смысле понятие OLAP запросов покрывает множество задач от работы исследователей данных (data scientist) до бизнес-аналитиков. Традиционно, аналитические задачи решались на уровне СУБД.

СУБД для аналитики

Реляционные СУБД предоставляют многочисленные возможности для оптимизации доступа и анализа агрегированных в хранилище данных. За счет абстрагирования методов доступа и алгоритмов обработки СУБД может контролировать способ хранения данных и оптимизировать план исполнения запроса. Различные виды запросов к одному хранилищу могут снижать общую производительность системы. Так, разделение процессов обработки запросов с целью обновления и анали-

[1] Компонента СУБД, отвечающая за исполнение плана.

[2] Silberschatz A., Korth H. F., Sudarshan S. 2020.Database System Concepts. Seventh Edition. McGraw-Hill Book Company, 2019. 1376 p.

[3] Extract Transform Load (ETL) [Электронный ресурс] // Databricks, 2021. URL: https://databricks.com/glossary/extract-transform-load (дата обращения: 13.10.2021).

за важно при повторяющихся запросах, например, при сборе данных для их онлайн визуализации и создания инструментальных панелей (dashboard) или переиспользовании данных (поиск новых паттернов и работа бизнес-аналитиков). Строго определенная схема хранилищ данных и специализированная структура хранения (например, физическое дублирование, колончатая организация данных) позволяют повышать производительность выполнения вычислительно-ёмких аналитических запросов (в силу природы аналитических запросов обращение обычно происходит ко всем доступным данным, к примеру, вычисление средних показателей величины столбца требует доступа ко всему столбцу). Количество данных требует распределенности хранения и вычисления, а сложность запросов требует еще и качественного планирования запроса с точки зрения выбора оптимальных реализаций алгоритмов и уровня параллелизма исполняемых операторов.

Поисковый анализ данных

Типичный сценарий аналитики над данными с целью поиска невыявленных закономерностей отличается от медленно изменяющихся запросов для построения отчетов. Такой сценарий подразумевает произвольные (ad-hoc) запросы в поисковом режиме (exploratory data analysis)[4]. Случайная природа таких запросов накладывает ограничения на возможности оптимизации. К примеру, планы для построения инструментальных панелей и других периодических вычислений можно кэшировать и переиспользовать. В случае произвольных запросов полное переиспользование становится невозможным, однако техники переиспользования более гранулярных результатов оптимизации по-прежнему доступны [3-5].

Пример:

```
select
  sum(l_extendedprice * l_discount) as revenue,
from
  lineitem
where
  l_shipdate >= date '1994-01-01'
  AND l_shipdate < date '1994-01-01' + interval '1' year
  AND l_discount BETWEEN 0.06 - 0.01 AND 0.06 + 0.01
  AND l_quantity < 24;
```

Листинг 1. TPC-H Q6

Рассмотрим типичный пример аналитического запроса из стандартной системы бенчмаркинга СУБД TPC-H[5]. Запрос рассчитывает увеличение дохода при изменениях в политике предоставления скидок на продукты некоторой компании. Оценка производится на основе данных базовой таблицы lineitem, которая содержит данные о продажах и ценах. Для простоты рассмотрим выполнения на одном узле:

(1) Получить доступ к таблице lineitem, то есть найти место в памяти (указатель/дескриптор), по которому она размещена. Для базы данных в основной памяти (in-memory) данные уже доступны и обращение к диску не требуется;

(2) Выделить память для результирующей таблицы (в данном случае таблица вырождается в скалярное значение). Размер может быть подобран исходя из статистики, хранящейся в каталоге и, в случае недостатка аллоцированного участка, динамически перевыделен;

(3) В цикле обойти все записи таблицы, для каждого кортежа вычисляя значения предикатов и помещая запись в свободный слот буфера с результатом (либо применяя функцию агрегации). При усложнении запроса к каждой записи будут последовательно применяться несколько операторов. Кроме того, часть операторов требует полной материализации промежуточных результатов в памяти.

Таким образом, время исполнения запроса к реляционной БД складывается из чтения данных входных таблиц, вычисления предикатов и агрегаций, соединения таблиц, сортировки результатов и прочих операторов. Соответственно, оптимизация запроса концентрируется на сокращении времени исполнения последовательности операторов.

Все современные СУБД в том или ином виде поддерживают колончатую организацию данных с последующим использованием векторной обработки, которая за раз обрабатывает не скалярные значения кортежей, а вектор значений, также могут применяться техники компиляции, которые будут рассмотрены далее.

СУБД в основной памяти

За более чем 50-ти летнюю историю развития СУБД разработано множество принципов и подходов к оптимизации запросов на разных уровнях – от архитектурных решений до конкретных алгоритмических методов, см., например, [6-8]. Основной целью таких методов традиционно было повышение эффективности использования аппаратуры. Однако, по мере развития индустрии, а также стремительного увеличения объема данных, направленность оптимизаций менялась [9, 10]. Так, классические СУБД требуют от разработчика наибольшего внимания ко взаимодействию с относительно медленным дисковым накопителем. Минимизация количества обращений и времени доступа породила множество оптимизаций, например, по способам хранения данных (сжатие), по способам доступа к данным (индексирование), по точности ответа (фильтры Блума) и т. д. Также появилась модель выполнения операторов «производитель/потребитель», которая позволяет как можно дольше оставлять записи внутри процессора – кэши, регистры [7].

С другой стороны, увеличение объемов оперативной памяти до нескольких терабайт и ее удешевление привело к появлению баз данных в основной памяти. Использование основной памяти для хранения данных меняет характеристики производительности алгоритмов, разработанных для классических СУБД. Фокус на оптимизацию наиболее узкого места – доступа к данным на диске – смещается в сторону исполнения [11]. СУБД становится выгодно учитывать и использовать

[4] Tukey J. W. Exploratory Data Analysis. Addison-Wesley, 1977. 712 p.

[5] TPC-H Vesion 2 and Version 3 [Электронный ресурс] // TPC, 2021. URL: https://www.tpc.org/tpch (дата обращения: 13.10.2021).

иерархию памяти – от кэшей до диска, механизмы упреждающей выборки команд (prefetching). Использование вторичных индексов становится менее актуальным [12]. Количество инструкций, необходимых для выполнения запроса, становится важным фактором, наряду с производительностью подсистемы памяти [13]. Скорости доступа к данным хватает для задействования физического параллелизма уровня множества ядер и векторных инструкций [14]. Наконец, исчерпание ресурса этих техник [15] привело индустрию к исследованию возможностей повышения эффективности за счет использования специализированных аппаратных модулей.

Гетерогенные системы

Под гетерогенностью мы понимаем возможность совместного исполнения некоторой задачи на двух или более устройствах с различной архитектурой. Это могут быть ядра ЦПУ предоставляющие различные возможности (к примеру, наличие дополнительного векторного исполнителя, FP ALU), либо устройства с различной моделью программирования (как связка ЦПУ и графического процессора, тензорного процессора, ИИ-ускорителя, NPU, MIC и др.).

Различия в наборе инструкций и микроархитектурных особенностях позволяют адаптировать аппаратуру для оптимизации определенного класса задач. Например, графический процессор имеет большую пропускную способность памяти, чем ЦПУ и позволяет использовать значительно больше потоков одновременно. Однако, за счет относительной простоты каждого ядра, сложные арифметические операции (например, деление) и нетривиальный поток исполнения (ветвление) оказываются менее эффективны.

Из-за различий в наборе инструкций, для использования различных устройств в одной системе требуются пути для генерации исполняемого кода (кодогенерации) и поддержка со стороны среды исполнения (runtime). Модели программирования устройств могут отличаться. Например, в случае графического процессора, исполнением необходимо управлять с ЦПУ. Графический же процессор, выполняет специально упакованные программы, называемые ядрами (kernel).

Использование гетерогенных систем для оптимизации запросов позволило ускорить работу некоторых классов отдельных операторов [16] и запросов целиком [17, 18].

Проблемы использования гетерогенных систем

Переход на графический ускоритель в качестве основного исполнительного устройства вычисления запросов не дает автоматического прироста производительности по всем типам запросов. Напротив, в широком классе запросов графический процессор проигрывает обычному ЦПУ, даже если данные локальны по отношению к графическому процессору [19]. Зависимость производительности от типа нагрузки создает необходимость выбора соответствующего устройства/устройств для выполнения запроса или его специфических частей. Высокоскоростные сети в дата-центрах, новые типы быстрой неволатильной памяти и передача данных между устройствами в системе представляют новый класс задержек в системе с характерным временем – микросекунды [20]. При снижении времени ответа I/O операций и обращений в нелокальную память, пропускная способность вычислителей будет падать. Текущие методы скрытия задержек на обращение в память полагаются на параллелизм уровня инструкций, а I/O – на механизмы операционной системы, такие как смена контекста. Оба подхода плохо масштабируются на операции в микросекундном диапазоне: параллелизм уровня инструкций ограничен и его недостаточно [20], а смена контекста сравнима по времени с передачей данных из нелокальной памяти и будет вносить значимые задержки. Поэтому, необходима разработка новых алгоритмов эффективного использования гетерогенных систем. Кроме того, задача дополнительно усложняется в распределённых системах.

Таким образом, разработчику СУБД с использованием гетерогенных устройств необходимо решить следующие проблемы:

(1) Необходима эффективная модель для использования доступного параллелизма и векторизации. Модели программирования различных процессоров могут различаться. Архитектура СУБД должна учитывать эти различия и позволять эффективно планировать запрос;

(2) Передача данных между устройствами снижает производительность. Требуется механизм определения наилучшей конфигурации устройств и размещения входных и промежуточных данных в иерархии памяти для минимизации расходов связанных с перемещением данных;

(3) За счет появления неоднородностей ресурсов растет пространство поиска оптимального плана. Для оптимизатора требуются эффективные алгоритмы поиска, новые эвристики;

(4) За счет отличий в архитектурах устройств переиспользование моделей затрат для ЦПУ оказывается невозможным. Требуется единая модель затрат, которая учитывала бы удаленность данных для устройства, а также его специфические особенности. Производительность каждого отдельно взятого алгоритма зависит от определенных архитектурных и микроархитектурных свойств аппаратуры. Эффективная загрузка устройств требует точного планирования исполнения запроса;

(5) С увеличением количества используемых платформ усложняется поддержка и разработка СУБД. Необходимы методы снижения сложности и размеров исходного кода.

Прежде чем решать эти проблемы рассмотрим принципиальные классические механизмы и подходы к оптимизации запросов.

Подходы к повышению производительности

Целевые параметры оптимизации достаточно различаются для разных типов задач [10]. Это может быть среднее время выполнения одной транзакции (время отклика), общая пропускная способность в транзакциях в секунду, количество задействованных ресурсов (сервера, узлы исполнения, потоки), используемая память, потребление энергии, и т. д. Для OLAP-сценария целью оптимизации является сразу два параметра:

- время исполнения отдельно взятого запроса – метрика эффективности оптимизатора, задающая “время ответа”

базы данных. Минимизация этого критерия критична для всех видов СУБД;
- цена исполнения – по сути, мера количества ресурсов, расходуемых для вычисления запроса.

Оптимизация аналитического запроса к базе данных начинается с построения логического плана запроса, который состоит из реляционных операторов и представляет собой направленный ациклический граф (DAG). Оптимизация плана заключается в:
- нахождении эффективных реализаций операторов логического плана с помощью физических операторов, реализованных (напрямую или генерируемых) в СУБД;
- выборе способа обращения к данным (выбор вторичного индекса, если требуется, способ размещения промежуточных результатов);
- выборе способа (алгоритма) и порядка соединения таблиц (join) в пространстве семантически эквивалентных физических планов;
- выборе уровня параллелизма.

Построенный физический план затем передается на исполнение. Эффективность этой стадии определяется моделью исполнения.

Модели исполнения

Существует несколько моделей исполнения физического плана:
- итеративная модель (volcano или pipeline, tuple-at-a-time) [13] – классическая реализация, где каждый физический оператор реализует функцию перехода к потомку в графе «next», которая вызывается для каждой записи. Next получает записи от оператора потомка в графе физического плана и рекурсивно спускаясь по графу, вызов доходит до листового оператора, который возвращает запись некоторой таблицы. Запись обрабатывается цепочкой операторов до тех пор, пока это возможно, прежде чем начать обработку следующего кортежа (поздняя материализация, late materialization [21]). Таким образом достигается минимизация доступа к относительно медленному диску;
- полная материализация (operator-at-a-time) [22] – каждый оператор сразу материализует все записи в памяти. Такой подход хорошо работает для OLTP запросов, когда промежуточный результат относительно мал и помещается в кэш;
- векторная обработка (vector-at-a-time) [23] – то же, что и итеративная модель, но вызов next возвращает не одну, а блок записей. Очень эффективно при обработке возможностями векторной аппаратуры, например, SIMD. Подходит также и для OLAP-запросов;
- компиляция и конвейерная обработка (обычно, operator-at-a-time/morsel [24]) – реализованная в таких системах, как Hekaton [25], MemSQL [26], SparkImpala. Авторы в работе[6] выделяют 3 класса динамической компиляции: компиляция отдельных выражений, запросов целиком и автоматическая специализация интерпретатора. Качество генерируемого кода в большой степени зависит от экспрессивности промежуточных представлений, то есть способности представления отразить детали алгоритма в виде, доступном для эффективного исполнения на аппаратуре [8]. С помощью генерации исходного кода или промежуточного представления для запроса достигается минимизация количества вызовов функций. Еще оптимизируется работа с кэшами за счет повышения локальности и применяются компиляторные оптимизации для снижения количества инструкций, необходимых для исполнения запроса.

За счет накладных расходов на материализацию промежуточных результатов (из-за ограничений по пропускной способности памяти), итеративная модель и полная материализация значительно уступают в производительности последним двум моделям исполнения [27].

Наиболее исчерпывающее исследование на тему сравнения векторного и скомпилированного исполнения для ЦПУ представлено в [14]. Результаты показывают, что между компиляцией и векторизацией нет однозначно лучшего варианта как с точки зрения производительности, так и удобства использования. Векторный подход предоставляет удобство отладки и профилирования, но требует ручной работы по оптимизации примитивов. Компиляция избавляет от низкоуровневых деталей и позволяет использовать всю мощь существующих компиляторных оптимизаций. Количество машинных инструкций может быть в разы меньше, чем в случае векторизации. Однако анализ сгенерированного кода цикла конвейера операторов значительно труднее, чем для заранее определенных примитивов.

В случае гетерогенных систем векторное исполнение может оказаться нецелесообразным. Например, для графического процессора накладные расходы на доступ в глобальную память могут превысить выгоду от векторизации – исполнение оператора зачастую не нуждается в дополнительном типе параллелизма, который мы рассматриваем в главе 3.2. С другой стороны, компиляция больших по объему кода конвейеров усиливает давление на регистры (мера количества свободных регистров во время исполнения программы) и может приводить к высокой конкуренции за ресурсы.

Автоматическая специализация интерпретатора предоставляет интересные возможности для взаимодействия сгенерированного кода запроса и остальной БД. Модель программирования графического процессора, однако, ограничивает возможности подобных оптимизаций в силу особенностей использования ускорителя: исполнение кода на графическом процессоре в режиме интерпретации крайне неэффективно.

Одно из наиболее важных направлений оптимизации – эффективное использование доступного многоуровневого (многопроцессорного, многопоточного, SIMD) параллелизма, предоставляемого современной аппаратурой.

[6] Шарыгин Е. Ю., Бучацкий Р. А. Обзор методов динамической компиляции запросов // Труды Института системного программирования РАН. 2017. Т. 29, № 3. С. 179-224. doi: https://doi.org/10.15514/ISPRAS-2017-29(3)-11

Использование параллелизма

Центральный процессор, как и другие вычислители, развивается по пути наращивания ресурсов для параллельного исполнения как с точки зрения уровня инструкций с глубиной конвейера с изменением последовательности команд (out-of-order), так и векторных вычислений. Использование параллелизма в OLAP СУБД более чем оправданно: запросы к базе данных независимы (каждый запрос может исполняться в собственном потоке – «interquery» параллелизм). Большинство операторов одного запроса можно исполнять в параллель – повышая утилизацию ресурсов и сокращая время доступа к данным.

Классическим способом контроля и использования параллелизма внутри одного запроса является введение дополнительного оператора (Exchange) в граф физического плана исполнения запроса [7].

Выделяют основные типы внутреннего параллелизма:

- горизонтальный параллелизм – один оператор может быть выполнен в несколько потоков. Например, scan-filter, заранее разделяя память (части одной таблицы) на участки, может быть исполнен в несколько потоков: каждый обращается к своему участку таблицы (partition). Полученные, таким образом, результаты необходимо агрегировать для сохранения единообразия интерфейса операторов.;
- вертикальный параллелизм – в отличие, от горизонтального, использует идею конвейеризации исполнения последовательных операторов в графе физического плана. Если следующий оператор не требует полной материализации результатов предыдущего, то он может начать исполнение, не дожидаясь окончания работы дочернего оператора – типичная модель производитель/потребитель. Другой вариант использования вертикального параллелизма – параллельное выполнение независимых операторов.

В распределенной системе время передачи данных от памяти к исполнителю значительно варьируется, поэтому важной задачей становится разделение и размещение данных в архитектуре с неоднородным доступом к памяти (NUMA) [24, 28] показывают, что явное адаптивное управление размещения “горячих” данных и планирование исполнения может значительно повысить пропускную способность системы.

Модель исполнения отвечает за производительность реализации стратегии вычисления запроса. При одной и той же стратегии семантически эквивалентные планы будут отличаться по производительности. СУБД необходимо найти наиболее выгодный план исполнения.

Организация поиска оптимального плана

Современные оптимизаторы, основанные на оценке затрат(cost-based), берут начало из 2х проектов: оптимизатор IBM System R (подход к перебору «снизу-вверх», bottom-up) и оптимизатор проекта Exodus («сверху-вниз», top-down). В первом случае для обхода альтернатив применяют технику динамического программирования. Сначала происходит оценка стоимости операций в листах дерева, то есть, обращений к таблицам (использовать ли вторичный индекс, если да, то какой из доступных). Затем, для каждого листа перебираются алгоритмы и вычисляется наименьшая стоимость обращения. Полученные значения используются для вычисления стоимости родительского узла, и так далее до корня. Во втором случае, с помощью каскадов [29], оптимизатор будет хранить просмотренные хешированные альтернативы в компактном представлении, и руководствоваться принципом Беллмана (любая часть оптимального плана – оптимальна) спускаясь по дереву, получая стоимость при достижении листов.

Классический подход

В силу сложности задачи оптимизация происходит поэтапно. Классический оптимизатор состоит из двух стадий:

(1) Применение эвристических правил с целью редактирования входного запроса (re-writer). Примеры: приведение запроса к стандартному виду, замена разреза данных (view) на их фактическое определение, снижение вложенности циклов;

(2) Поиск оптимального плана в пространстве эквивалентных. Для сравнения двух планов используется модель затрат. Оптимизатор оценивает стоимость исполнения операции за счет использования статистики, предоставляемой каталогом (содержит схемы таблиц и статистические метаданные), а также размер выходных данных в случае подхода «снизу-вверх».

Сложность моделей предсказания стоимости операций ограничена временем на оптимизацию запроса. Если время оптимизации будет значительно превышать время исполнения и не давать ощутимой выгоды, то СУБД выгоднее отключить оптимизации вовсе. Тем не менее, обычно, оптимизатор позволяет ускорять сложные запросы в разы [30].

Поскольку оптимизатор основывается на оценке стоимости операций и размерах выходных данных (cardinality), требуются точные оценки статистических параметров текущих данных (которые, к тому же, могут устаревать). Классические техники основываются на предположениях о данных: (1) независимость, (2) равномерность. И потому могут значительно ошибаться. При больших ошибках оценки размера выходных данных модель затрат практически не оказывает влияния на точность определения стоимости операции [30]. Поэтому существует множество методов улучшения этих оценок. Исчерпывающий обзор техник представлен в [31].

Добавление еще одного устройства в систему как минимум удваивает количество вариантов исполнения каждого отдельно взятого оператора. Для плана целиком это означает экспоненциальный рост пространства перебора. Для гетерогенных СУБД требуются новые подходы к оптимизации.

Альтернативные подходы

Организация перебора с эвристическим сокращением пространства перебора – неединственный способ поиска оптимального плана. К примеру, идея использования машинного обучения нашла развитие в применении моделей не только для оценки статистических параметров данных, но и как способа решения задачи оценки стоимости [32] или оптимизации запроса целиком [33]. Одной из основных проблем обучаемых

моделей является слишком низкое качество генерируемых планов на этапе «нулевого знания». NEO [33] решает эту проблему с помощью переиспользования опыта разработчиков оптимизаторов для быстрого обучения модели. Другая проблема – необходимость переобучения модели при изменении статистических параметров данных: модели склонны адаптироваться к текущему набору данных. BAO [34] решает эту задачу неинтрузивным образом. Как и [33], используется существующий оптимизатор. BAO управляет процессом за счет «подсказок» оптимизатору. Свёрточная модель на графе плана запроса выявляет паттерны и обучается выбору наиболее выгодного набора подсказок в нетривиальных случаях. Таким образом, сглаживаются ошибки эвристического оптимизатора. Подход BAO чувствителен к размерности пространства поиска и при значительном его увеличении эффективности оптимизатора оказывается недостаточно [34].

Такое обучение для оптимизации плана строится на основе обратной связи. Разумеется, применение методов машинного обучения – не единственный способ переиспользования исторической информации. Например, в Microsoft SQL Server существует техника оптимизации планов, построенная на идее объединения эффективных частей планов [3].

Другой подход – использование рандомизированных алгоритмов. Например, в PostgreSQL[7] генетический оптимизатор [35] рассматривает возможные решения задачи оптимизации как популяцию. Варианты планов кодируются с помощью числовых строк, которые обозначают порядок соединений таблиц. Особи с наихудшей оценкой стоимости заменяются в новой популяции. Новые особи создаются с помощью участков (генов) с известной низкой оценкой стоимости. Процесс продолжается до тех пор, пока оптимизатор не оценит заданное количество планов. В отличие от стандартного оптимизатора PostgreSQL, генетический алгоритм позволяет снижать количество перебираемых планов и эффективно искать оптимум для сложных запросов.

Выводы

В секции 2.6 мы обозначили 5 проблем использования гетерогенных систем. При использовании классических методов оптимизации аналитических запросов остаются нерешенными следующие:

(1) Классический подход предоставляет модель параллелизма, но требуется ее расширение для поддержки нескольких устройств;

(2) Существуют методы эффективной компиляции запроса такие как [24, 13], но необходима их адаптация для исполнения на устройствах;

(3) Требуется повышение эффективности оптимизатора;

(4) По-прежнему требуется единая модель затрат;

(5) По-прежнему необходимы методы снижения сложности поддержки платформ.

Обзор существующих решений

Исследования последнего десятилетия определили два основных подхода к интегрированию аппаратных ускорителей: использование ускорителя для решения узких задач, либо полноценное исполнение реляционных операторов на всех доступных устройствах.

Наиболее распространенным способом организации взаимодействия с ускорителем является использование шины PCIe для передачи данных и управления устройством. В такой конфигурации, для использования специализированной аппаратуры в качестве ускорителя запросов к базе данных ключевую роль играют 2 фактора ограничивающих потенциал использования [36]: ограниченная память устройства (по сравнению с основной) и относительно низкая скорость передачи данных между ЦПУ и устройством. Разница в пропускной способности памяти для центрального и графического процессора, при учете ограниченности скорости PCIe не позволяет использовать графический процессор как сопроцессор, эффективно исполняющий примитивы [37]. Вне зависимости от выбора ускорителя, для «холодного» запуска (то есть, когда данные не помещены в память устройства заранее), время передачи данных доминирует над исполнением.

Далее мы рассмотрим наиболее значимые подходы к использованию различных гетерогенных платформ для оптимизации запросов.

GDB

Целью реализованной системы GDB [38] было сравнение производительности аналитических запросов на ЦПУ и графическом процессоре. Поэтому, каждый из путей оптимизировался отдельно под устройство и на разных языках программирования, с учетом архитектурных особенностей. Чтобы обеспечить взаимозаменяемость реализаций операторов и возможность совместного исполнения одного оператора на разных устройствах, исполнение в GDB было построено на примитивах. Поскольку реализация на ЦПУ и графическом процессоре различаются, оптимизатору требуется расширение модели затрат, которая бы учитывала архитектурные различия и координацию между устройствами. Стоимость каждого примитива оценивалась калибровочным способом с помощью сопоставительного анализа – для каждой операции (например, map) считается удельная стоимость, как полное время исполнения, делённое на количество потоков. Задержки в памяти модель не учитывает, полагаясь на большой объем данных и возможность аппаратуры скрывать их за сменой контекста. Стоимость обращения в память каждого примитива оценивается аналитически, с учетом заранее известной пропускной способности шины передачи данных. Полная стоимость исполнения складывается из времени передачи данных и времени исполнения алгоритма.

В такой модели данные оператора, исполненного на сопроцессоре, копировались по относительно медленной PCIe шине при каждом вызове примитива. За счет этого модель показывала низкие значения прироста производительности целого запроса несмотря на значительное ускорение отдельных операторов. Кроме того, сама модель исполнения создания от-

[7] PostgreSQL: The World's Most Advanced Open Source Relational Database [Электронный ресурс] // The PostgreSQL Global Development Group, 2021. URL: https://www.postgresql.org (дата обращения: 13.10.2021).

дельных ядер для каждого оператора имеет низкую утилизацию ресурсов графического процессора [22]. Тем не менее, при верной стратегии размещения данных, использование графического процессора позволяет повысить производительность и энергоэффективность исполнения [37].

Другой не менее важной проблемой подхода является дублирование кода для поддержки сопроцессора. С точки зрения развития реальной системы такая архитектура будет значительно ограничивать возможности разработчика.

Таким образом, на основе данных первого прототипа гетерогенной СУБД можно выделить три аспекта проблем, обозначенных в параграфе 2.6:

(1) Дублирование работы по реализации алгоритмов для каждого оператора (или отсутствие единого пути компиляции, аспект проблемы 5);

(2) Низкая утилизация ресурсов графического процессора при пооператорном исполнении (проблема 1);

(3) Низкая эффективность взаимодействия ЦПУ и графического процессора (проблема 2).

Ocelot, HeroDB

Ocelot [39] решает проблему дублирования кода за счет использования аппаратно-независимых (hardware-oblivious) операторов, которые реализованы на OpenCL [40]. Сравнивая с базовой реализацией MonetDB [27], авторы показывают, что подход генерирует код на разных платформах, сопоставимый по производительности с оптимизированным при достаточно больших объемах данных (скрывает накладные расходы времени исполнения) и использовании плана запроса, изначально оптимизированного для исполнения на ЦПУ.

Преимущества подхода:

- переносимость. Реализация операторов на языке OpenCL позволяет использовать произвольную аппаратуру с поддержкой OpenCL интерфейса;
- простота в поддержке. Для исполнения запроса существует всего один путь в кодовой базе, соответственно, разработчикам не приходится поддерживать множество оптимизаций под специфичные архитектуры – генерация кода оператора полагается на компиляторные оптимизации, предоставляемые производителем.

С точки зрения производительности у подхода есть две основные проблемы:

(1) Реализация операторов предполагает возможность полного размещения необходимых данных в памяти устройства. Это, скорее искусственное, ограничение не позволяет модели воспользоваться преимуществом ускорения вычислений от аппаратного ускорения в случае больших размеров входных данных;

(2) Несмотря на ленивость вычислений, модель исполнения Ocelot – «operator-at-a-time» (см. главу 3.1). Это означает, что каждый оператор, в том числе, не являющийся границей конвейера (pipeline breaker), полностью материализует результат в памяти. Несмотря на высокий уровень параллелизма операторов это не является оптимальной стратегией исполнения (лишняя материализация промежуточных результатов между операторами, не являющимися границей конвейера). Кроме того, теряется преимущество способа исполнения через компиляцию – пользовательский код (например, условия предикатов) становится отдельным оператором вместо того, чтобы стать частью существующего. Другими словами, отсутствуют полноценные межоператорные оптимизации.

Тем не менее, такая модель позволяет принимать решение о размещении операторов на нужных устройствах динамически. Поскольку исполнение на ускорителе неизбежно связанно с накладными расходами, в зависимости от примитива и размера входных данных в каждом конкретном случае оптимальное размещение вычислений варьируется. На простых запросах [17] показывает, как с помощью простой «обучающейся» модели для определения размещения операторов получить значительный прирост производительности (до 4х раз), по сравнению со статическим размещением на одном из типов устройств (например, полностью на дискретном графическом процессоре). Эксперименты с распараллеливанием независимых операторов на уровне устройств не показали значительного прироста производительности по причине высоких накладных расходов в поставленном эксперименте, однако модель подразумевает потенциальный выигрыш от такого сценария использования.

Прирост производительности от эффективного использования всех доступных ресурсов демонстрирует He..roDB [18]. За счет совместного использования центрального процессора и интегрированного графического сопроцессора прототип демонстрирует примерно 2-х кратный прирост производительности на случайных запросах TPC-H [41] при малых размерах данных по сравнению с исполнением в режиме одного устройства (ЦПУ или сопроцессор). Стоит отметить, что авторы не указали конкретную модель используемого процессора, в то время как измерения [17] демонстрируют ожидаемый эффект снижения рабочих частот процессора и графических ядер в режиме совместного исполнения за счет разделения ресурсов (для экспериментов использовалась интегрированный графический процессор). К тому же прототип использует две отличных друг от друга реализации операторов, которые специфичны устройствам исполнения, что значительно затрудняет поддержку и развитие такой системы, хотя и позволяет использовать все доступные оптимизации.

В [17] для выбора устройства исполнения оптимизатор пользуется линейной экстраполяцией (по размеру входных данных) накопленных исторических данных о производительности, чтобы оценить время исполнения оператора, и эвристикой, которая максимизирует утилизацию устройств. Такой подход работает в случае пооператорного исполнения и слабо применим в случае совместной компиляции нескольких операторов, поскольку пространство вариантов исполнения возрастает (что означает значительное снижение количества экспериментальных данных для экстраполяции). Помимо этого недостатка, модель ограничена в выборе, если входные данные оператора не помещаются в память устройства. Так, оптимизатор теряет потенциальный выигрыш от использования оптимального устройства.

Hawk, CoGaDB, Hype, HorseQC

В этих системах используется порождение исходного кода запроса на языке, допускающем исполнение на различных плат-

формах. Такой подход по-прежнему позволяет иметь одну реализацию для всех устройств, но в то же время потенциально позволяет генерировать код для нескольких операторов сразу. HAWK [41] генерирует устройство-независимый OpenCL код для конвейеров запроса и пользуется существующими реализациями генераторов машинного кода для нужной платформы (OpenCL позволяет HAWK исполнять запросы как на ЦПУ, так и на графическом процессоре и других многоядерных архитектурах). В отличие от пооператорного исполнения такой подход позволяет сэкономить ресурсы на материализацию промежуточных результатов между операторами, не являющимися границей конвейера, передавая значения записей в регистрах. Более того, слияние нескольких операторов в одну единицу трансляции позволяет компилятору применять стандартные оптимизации.

оптимизатор реализовывал две стратегии: жадную (использование оптимального логического плана и последующая вставка операторов копирования данных для переноса на другое устройство) и перебор с ограничением по количеству переходов между устройствами. Однако, как отмечается позже, простых эвристических методов оказывается недостаточно даже для решения задачи выбора оптимального устройства исполнения [41].

HorseQC [22] позволяет склеивать ядра операторов для минимизации накладных расходов на запуск и копирование памяти для каждого оператора.

Совмещение генерации исходного кода и склеивания ядер операторов решает проблемы единого пути генерации кода запроса для разных устройств и эффективности получаемого кода запроса (проблемы 2 и 5). Однако, подход не удается использовать в реальных сценариях из-за высоких накладных расходов. Снижение уровня абстракции и генерация близкого к машинному коду представления делает подход более привлекательным.

Другой подход к снижению времени компиляции – классический подход JIT с различными ступенями оптимизаций [42]: интерпретация, компиляция без оптимизаций и компиляция с оптимизациями. Повышение уровня оптимизации для вычислительно-ёмких запросов позволяет решить проблему времени компиляции и не имеет проблемы с увеличением размера данных, однако техника подойдет не для любого устройства.

Voodoo

Менее затратный способ решения той же проблемы высоких накладных расходов на генерацию промежуточного представления и компиляцию – порождение промежуточного представления. Voodoo [43] алгебра позволяет транслировать граф реляционных операторов в абстрактную промежуточную алгебру, из которой получают бинарный код необходимого устройства с помощью Just-In-Time (JIT) компиляции. Как и [13], обход плана запроса формирует «фрагменты» с промежуточным представлением. Каждый фрагмент характеризуется двумя параметрами: степень параллелизма по данным (в терминах OpenCL может быть выражено размерностью итерационного пространства) и количество последовательных операторов. Операторы с одинаковой степенью параллелизма могут быть выполнены в пределах одной функции, то есть обойтись без дорогостоящей материализации промежуточного результата.

Собственная алгебра требует отдельного механизма понижения уровня абстракции в соответствующее представление. В отличие от, к примеру, промежуточного представления компилятора (LLVM)[8], отдельное промежуточное представление не позволяет напрямую использовать существующий JIT-компилятор – для него необходим отдельный модуль машинно-зависимых оптимизаций. Процесс набора фрагмента по степеням в большинстве случаев излишен. Оптимизатор заранее знает какие операторы должны быть объединены в один фрагмент, а реализации алгоритмов заранее известны. Этой информации должно быть достаточно для разделения исполнения на ядра в терминах OpenCL.

LegoBase

Другой способ высокоуровневого представления и использования примитивов операторов СУБД – встраивание их в язык и JIT компиляция [44] (LegoBase). Идея заключается в предоставлении разработчику СУБД возможности использования высокоуровневого языка (Scala) для описания оптимизаций и генеративного программирования. Большая часть конструкций языка Scala хорошо выражается с помощью концепций значительно более эффективного языка C. Поэтому LegoBase расширяет набор операций внутреннего представления JIT-компилятора и генерирует эффективный C код, который затем компилируется. LegoBase не использует эти операторы в классическом оптимизаторе, а применяет технику уже после получения физического плана.

Определяют основные проблемы предыдущих решений:

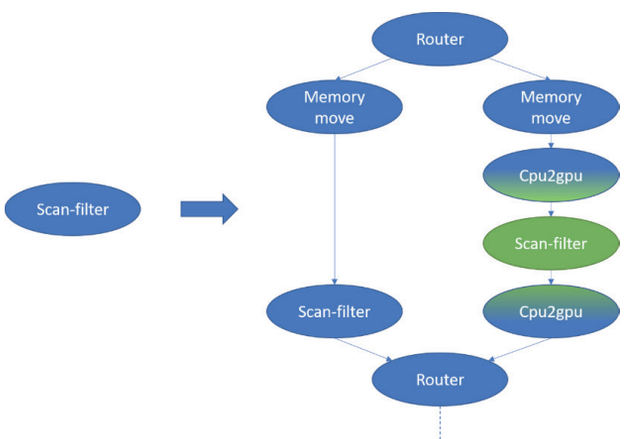
- разворачивание шаблонов не позволяет эффективно использовать кросс-операторные оптимизации;
- шаблоны тяжело реализовывать, поскольку они низкоуровневые и отсутствует строгая типизация (частично верно даже для генерации LLVM представления);
- компилятор запросов не может использовать оптимизации, направленные на включение (inline) кода запроса в саму базу данных, поскольку его область видимости ограничена запросом;
- LLVM и подобные позволяют применять оптимизации времени исполнения (напр. JIT), но они не позволяют использовать высокоуровневую информацию (селективность предиката).

Большинство аргументов против использования низкоуровневого представления с помощью LLVM указывает лишь на то, что в первых версиях оптимизатора [13] не было эффективной поддержки вычисления предикатов и динамических реализаций структур данных. Вместо этого использовались заранее скомпилированные модули.

Решение обладает двумя важными преимуществами, идеи которых переиспользуются в новых решениях.

[8] Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization : Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL., USA, 2002. URL: https://llvm.org/pubs/2002-12-LattnerMSThesis.html (дата обращения: 13.10.2021).

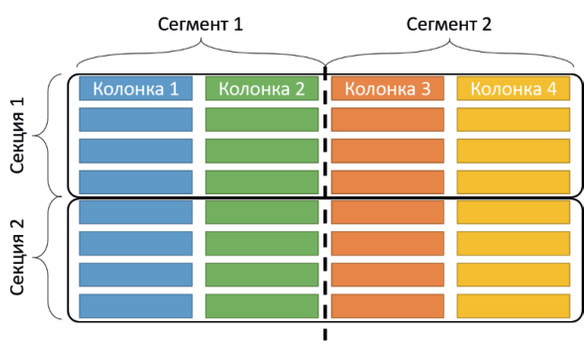
(1) Высокоуровневое представление операций со сложными структурами данных, таких как хеш-таблица, позволяет удобно управлять моделью исполнения и подробностями реализации. Это снижает сложность поддержки и повышает вариативность реализаций для повышения производительности в зависимости от динамических условий. В частности, это позволяет LegoBase удобно менять представление данных (колоночное и построчное);
(2) Использование единой системы JIT-компиляции и для запросов, и для самого исходного кода СУБД открывает дополнительные возможности кросс-функциональной оптимизации.



Р и с. 1. Пример преобразования оператора физического плана в гетерогенный
F i g. 1. An example of transforming a physical plane operator into a heterogeneous one

HetExchange

HetExchange [45] расширяет набор операторов физического плана запрос добавлением трех классов специализированных операторов: смена контекста исполняющего устройства, смена локализации данных и оператор управления потоком данных (рис. 1). При неизменных стандартных операторах такой фреймворк позволяет генерировать планы явно управляющие всеми доступными уровнями параллелизма и добиваться прироста производительности как относительно других гомогенных параллельных исполнителей, так и собственной реализации с исключительным устройством исполнения (ЦПУ или графический процессор). Планирование запроса ортогонально модели исполнения (векторизация против кодогенерации), и по словам авторов, модель может быть использована в любом типе.



Р и с. 2. Сегментирование и секционирование таблицы в памяти
F i g. 2. Segmenting and sectioning the table in memory

Нерешенной проблемой остается определение способа разделения данных и нагрузки в гетерогенной системе, то есть, необходима модель затрат, позволяющая задать параметры распределения данных между устройствами.

VOILA

Еще одно измерение расширения пространства поиска предлагает VOILA [46]. VOILA предлагает язык, который позволяет абстрагировать физический уровень реализации обработчика запросов со сравнимой производительностью по отношению к реальным системам, выполненной по заданному дизайну. В отличие от промежуточных представлений Voodoo [43], Weld [47] и подобных, VOILA предлагает абстрактное описание оператора и дополнительный уровень генерации машинного кода FUJI, которые позволяют описывать алгоритмические детали и менять их для создания новых вариантов исполнения. Для описания новых видов hash-join, Voodoo, например, необходимо реализовывать отдельные имплементации циклом. VOILA разбивает такие сложные задачи на компоненты: вычисление хеша, поиск нужной «корзины» (bucket lookup), проверка ключа на выполнение предиката, переход к следующей корзине, и т. д. Поскольку задача распараллеливания скалярного кода слишком трудна, каждый примитив всегда работает с векторным представлением по умолчанию. Чтобы реализовать скалярную стратегию исполнения размер вектора приравнивается к 1, а для поколоночного (column-at-a-time) подхода – бесконечности.
Для каждого конвейера запроса VOILA может применять наиболее подходящий способ исполнения, как если бы существовал обработчик запросов с именно такими характеристиками. VOILA также позволяет варьировать способ исполнения внутри каждого конвейера с помощью специального выражения смешения (Blend). Выражения смешения определяют дочерний вариант исполнения внутри произвольной области видимости (некоторый фрагмент программы вычисляющей запрос). На границах таких областей данные буферизуются, и подпрограмма выполняется в новом варианте. Такая гибкость

позволяет находить оптимальные реализации алгоритмов с точки зрения модели исполнения и получать многократное повышение производительности.
Цена такой гибкости – высокая стоимость поиска подходящей модели исполнения. Помимо перебора реализаций операторов в классической модели, VOILA добавляет измерения внутри каждого оператора. Такое пространство перебора потребовало от авторов введения ограничений даже на запросах из TPC-H.

OmniDB

OmniDB [48] предпринимает попытку решения проблемы задачи выбора оптимального распределения данных по устройствам, учитывая требования к эффективной разработке и поддержанию кодовой базы для множественных архитектур. Поскольку большинство компонент систем, специализированные под разные архитектуры, можно переиспользовать, авторы предлагают многоуровневую архитектуру на базе [38]. Для решения проблем переносимости и эффективности расширяемые параллельные ядра (примитивы, из которых составляется запрос, «qKernel») вычисляют запрос с помощью адаптеров, которые скрывают все подробности архитектуры устройства исполнения. Ядро, в свою очередь, основано на абстрактной параллельной модели вычислений (OpenCL) и не меняется в зависимости от устройства.
Модель совместной (ЦПУ/графический процессор) архитектуры заключается во введении N-поточных вычислителей (Parallel Processing Element, PPE), каждый из которых имеет свое пространство в памяти с блочным обращением к данным. На верхнем уровне планировщик создает ядра для операторов и разделяет работу, например обработку блока кортежей таблицы, между PPE. ЦПУ и графический процессор рассматриваются как PPE с учетом их возможностей: планировщик балансирует нагрузку и пытается выбрать наиболее выгодное устройство исполнения с точки зрения производительности. Оптимизация использует жадный алгоритм:

- оценивается пропускная способность каждого PPE;
- текущая загруженность устройства, то есть сколько блоков работы ожидает исполнения (статус «pending»);
- на основе полученных данных выбирается PPE с наибольшей пропускной способностью, который входит во множество недостаточно загруженных (уровень нагрузки определяется по пороговому значению).

Модель затрат строится на двух возможностях адаптеров: оценивать количество блоков памяти, к которым обращается ядро и количество используемых инструкций. Для I/O используют стандартный метод оценки[9]. Адаптеры, затем, калибруют размеры и скорость доступа к кэшам, пропускную способность PCIe шины, размер данных в блоке (на разных уровнях: запроса, оператора, OpenCL ядра).
Жадный алгоритм позволяет повысить эффективность планирования запроса в гетерогенной системе (до 30% эффективнее наивного распределения работы между устройствами), а специализация под архитектуру снижает количество промахов в кэш второго уровня на 20%. Однако, значительного прироста производительности прототип еще не показал. Кроме того, локальность принимаемых решений не позволяет оптимизатору достигать оптимальной эффективности.

OmniSciDB (Mapd)

СУБД в основной памяти OmniSciDB позволяет исполнять запросы на графическом ускорителе и центральном процессоре и совмещает в себе множество преимуществ описанных исследований. Логический план оптимизируется и представляется с помощью внутренней алгебры выражений (abstract syntax tree, AST). Выражения хранят в себе детали запроса, например, предикаты. После генерации физического плана OmniSci транслирует конвейеры операторов («шаг») плана в промежуточное представление компилятора (LLVM) и генерирует код для каждого шага под выбранную платформу. Полученное ядро исполняется над колоночным представлением данных (column-at-a-time). Единообразие принципа исполнения на ЦПУ (исполнение опирается на JIT) и графическом процессоре упрощает архитектуру СУБД. Пользуясь таким механизмом OmniSci склеивает операторы в одно ядро и применяет компиляторные оптимизации для межоператорного взаимодействия (то есть в том числе адаптирует технику [13] для графических процессоров). Важным преимуществом OmniSci является то, что СУБД минимизирует издержки на передачу данных от ЦПУ к графическому ускорителю за счет хранения горячих данных в памяти графического ускорителя между вызовами ядер шагов. Это снижает нагрузку на шину данных между графическим процессором и ЦПУ.
Основными недостатками подхода являются ограничение по памяти (OmniSci не позволяет исполнить запрос в описанном сценарии при отсутствии достаточного объема памяти устройства для промежуточных результатов) и отсутствие совместного исполнения шагов на различных устройствах. Некоторые классы запросов не поддерживаются на графических процессорах, однако СУБД предусматривает механизм восстановления (перезапуск части вычислений на ЦПУ) при возникновении ошибки исполнения на графическом процессоре – запрос всегда будет выполнен до конца.

Открытые проблемы

В существующих подходах предложены хорошие механизмы использования параллелизма (проблема 1). В частности, наиболее перспективным подходом является описанный в секции 4.6. Подход хорошо масштабируется, но в то же время, не слишком расширяет пространство поиска, как рассмотренный в параграфе 4.7. Совместная компиляция конвейеров операторов вида HorseQC (параграф 4.3) позволяет значительно снизить накладные расходы на перемещение данных (проблема 2). Одной компиляции, однако, недостаточно. Для эффективного исполнения по-прежнему требуется планирование (проблема 3). Методы, предложенные в секциях 4.8 и 4.3 опираются на простые эвристики и жадный алгоритм. Для эффективного оптимизатора этого недостаточно. Модель затрат, рассмотренная в параграфе 4.1 оценивает стоимость единичных операторов. Для использования совместной компиляции операторов тре-

[9] Silberschatz A., Korth H. F., Sudarshan S. 2020.Database System Concepts. Seventh Edition. McGraw-Hill Book Company, 2019. 1376 p.

буется ее расширение (проблема 4). Наконец, единая модель программирования гетерогенных систем, рассмотренная в 4.2 снижает сложность разработки (проблема 5). Недостатком такого решения являются более низкая скорость компиляции и низкая выразительность промежуточного представления (частично решены с переходом к генерации промежуточного представления).

Таким образом, нерешенными остаются следующие задачи:

- эффективный поиск оптимального плана в расширенном пространстве поиска;
- расширение модели затрат для учета совместной компиляции операторов;
- повышение скорости единого пути кодогенерации для гетерогенных платформ.

Центральной задачей остается быстрый поиск эффективных планов и расширение модели затрат. Эта задача актуальная для платформ, состоящих из ЦПУ и графического процессора в силу большой распространенности. Интегрированный графический процессор встречается в большинстве современных настольных процессоров, а большинство серверов предоставляет возможность подключения мощных дискретных графических процессоров общего назначения (GPGPU[10]). И значительная часть операторов (например, вычисление сложных предикатов) может быть эффективно исполнена на параллельной архитектуре графических процессоров.

Заключение

В заключении рассмотрим задачу многокритериальной оптимизации генерации оптимального плана для гетерогенного исполнения. Как показано в главе 3 критериями оптимизации являются: время исполнения запроса и цена как мера используемых ресурсов. Гетерогенное исполнение запросов снижает время исполнения, а управление уровнем параллелизма позволяет контролировать стоимость исполнения. Для простоты ограничимся задачей для одного узла. Без ограничения общности будем считать, что платформа состоит из двух устройств, например ЦПУ и графического процессора. Каждый из процессоров обладает собственной локальной памятью, обращения в которую значительно эффективнее доступа к памяти другого устройства. Будем рассматривать гетерогенный план совместного исполнения, получаемый с помощью стандартного оптимизатора аналогично [45]. План запроса будет состоять из «шагов», которые представляют собой последовательности совместно компилируемых операторов, не требующих полной промежуточной материализации. Промежуточные результаты можно как оставлять на текущем устройстве для последующей обработки дочерними узлами графа, так и перемещать на другое устройство. Для каждого шага СУБД генерирует ядро (или несколько ядер), которое позволяет асинхронно выполнить последовательность операторов на обоих устройствах. Таблицы БД разделены на фрагменты или, что тоже самое, секции (рис. 2). Каждое ядро может обрабатывать часть таблицы (один и более фрагментов).

Полное время исполнения складывается из времени оптимизации плана, компиляции ядер, перемещении данных и фактического исполнения на устройствах. Время оптимизации контролируется скоростью сходимости алгоритма поиска и граничными условиями (например, не более 200 миллисекунд на оптимизацию). Стоимость компиляции приблизительно линейна относительно размера ядра и не рассматривается. Перемещение данных контролируется специальными операторами плана и является частью оптимизации (мы выносим это в отдельный этап после построения оптимального плана для одного устройства с целью снижения пространства перебора). Наконец, время исполнения зависит от модели исполнения и от выбора устройства, что тоже является частью оптимизации. Таким образом, задача заключается в следующем: по заданному гетерогенному плану (строится из оптимизированного логического плана), выбрать такое распределение данных между устройствами для каждого шага, чтобы минимизировать полное время исполнения с учетом ограничений на время оптимизации.

Задача относится к классу задач нелинейной многомерной многокритериальной оптимизации и оценивается как NP-трудная.

References

[1] Wang C., Chen M.-S. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering.* 1996; 8(4):650-662. (In Eng.) doi: https://doi.org/10.1109/69.536256

[2] Appuswamy R., Karpathiotakis M., Porobic D., Ailamaki A. The Case For Heterogeneous HTAP. *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'17).* Chaminade, California, USA; 2017. Available at: https://www.cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf (accessed 13.10.2021). (In Eng.)

[3] Ding B., Das S., Wu W., Chaudhuri S., Narasayya V. Plan Stitch: Harnessing the Best of Many Plans. *Proceedings of the VLDB Endowment.* 2018; 11(10):1123-1136. (In Eng.) doi: https://doi.org/10.14778/3231751.3231761

[4] Giannikis G., Makreshanski D., Alonso G., Kossmann D. Shared Workload Optimization. *Proceedings of the VLDB Endowment*. 2014; 7(6):429-440. (In Eng.) doi: https://doi.org/10.14778/2732279.2732280

[5] Mishra A., et al. Accelerating Analytics with Dynamic In-Memory Expressions. *Proceedings of the VLDB Endowment*. 2016; 9(13):1437-1448. (In Eng.) doi: https://doi.org/10.14778/3007263.3007280

[6] Barthels C., Müller I., Schneider T., Alonso G., Hoefler T. Distributed Join Algorithms on Thousands of Cores. *Proceedings of the VLDB Endowment*. 2017; 10(5):517-528. (In Eng.) doi: https://doi.org/10.14778/3055540.3055545

[10] GPGPU Definition [Электронный ресурс] // HEAVY.AI, 2021. URL: https://www.omnisci.com/technical-glossary/gpgpu (дата обращения: 13.10.2021).

[7] Graefe G. Encapsulation of Parallelism in the Volcano Query Processing System. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (Atlantic City, New Jersey, USA) (SIGMOD'90).* Association for Computing Machinery, New York, NY, USA; 199. p. 102-111. (In Eng.) doi: https://doi.org/10.1145/93597.98720

[8] Shaikhha A., Klonatos Ya., Parreaux L., Brown L., Dashti M., Koch C. How to Architect a Query Compiler. *Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD'16).* Association for Computing Machinery, New York, NY, USA; 2016. p. 1907-1922. (In Eng.) doi: https://doi.org/10.1145/2882903.2915244

[9] DeWitt D.J., Katz R.H., Olken F., Shapiro L.D., Stonebraker M.R., Wood D.A. Implementation Techniques for Main Memory Database Systems. *ACM SIGMOD Record*. 1984; 14(2):1-8. (In Eng.) doi: https://doi.org/10.1145/971697.602261_

[10] Stonebraker M., Cetintemel U. "One Size Fits All": An Idea Whose Time Has Come and Gone. *Proceedings of the 21st International Conference on Data Engineering (ICDE'05).* IEEE Computer Society, USA; 2005. p. 2-11. (In Eng.) doi: https://doi.org/10.1109/ICDE.2005.1

[11] Ailamaki A., DeWitt D.J., Hill M.D., Wood D.A. DBMSs on a Modern Processor: Where Does Time Go? *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA; 1999. p. 266-277. Available at: https://www.vldb.org/conf/1999/P28.pdf (accessed 13.10.2021). (In Eng.)

[12] Kester M.S., Athanassoulis M., Idreos S. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? *Proceedings of the 2017 ACM International Conference on Management of Data (Chicago, Illinois, USA) (SIGMOD'17).* Association for Computing Machinery, New York, NY, USA; 2017. p. 715-730. (In Eng.) doi: https://doi.org/10.1145/3035918.3064049

[13] Neumann T. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*. 2011; 4(9):539-550. (In Eng.) doi: https://doi.org/10.14778/2002938.2002940

[14] Kersten T., Leis V., Kemper A., Neumann T., Pavlo A., Boncz P. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proceedings of the VLDB Endowment*. 2018; 11(13):2209-2222. (In Eng.) doi: https://doi.org/10.14778/3275366.3284966

[15] Lottarini A., Ramirez A., Coburn J., Kim M.A., Ranganathan P., Stodolsky D., Wachsler M. Vbench: Benchmarking Video Transcoding in the Cloud. *ACM SIGPLAN Notices.* 2018; 53(2):797-809. (In Eng.) doi: https://doi.org/10.1145/3296957.3173207

[16] Kara K., Giceva J., Alonso G. FPGA-Based Data Partitioning. *Proceedings of the 2017 ACM International Conference on Management of Data (Chicago, Illinois, USA) (SIGMOD'17).* Association for Computing Machinery, New York, NY, USA; 2017. p. 433-445. (In Eng.) doi: https://doi.org/10.1145/3035918.3035946

[17] Karnagel T., Habich D., Schlegel B., Lehner W. Heterogeneity-Aware Operator Placement in Column-Store DBMS. *Datenbank-Spektrum*. 2014; 14(3):211-221. (In Eng.) doi: https://doi.org/10.1007/s13222-014-0167-9

[18] Müller M., Leich T., Pionteck T., Saake G., Teubner J., Spinczyk O. He..ro DB: A Concept for Parallel Data Processing on Heterogeneous Hardware. In: Brinkmann A., Karl W., Lankes S., Tomforde S., Pionteck T., Trinitis C. (eds.) *Architecture of Computing Systems – ARCS 2020. ARCS 2020. Lecture Notes in Computer Science*. Vol. 12155. Springer, Cham; 2020. p. 82096. (In Eng.) doi: https://doi.org/10.1007/978-3-030-52794-5_7

[19] Zhang Ya., Zhang Yu, Lu J., Wang Sh., Liu Z., Han R. One size does not fit all: accelerating OLAP workloads with GPUs. *Distributed and Parallel Databases*. 2020; 38(4):995-1037. (In Eng.) doi: https://doi.org/10.1007/s10619-020-07304-z

[20] Barroso L., Marty M., Patterson D., Ranganathan P. Attack of the Killer Microseconds. *Communications of the ACM.* 2017; 60(4):48-54. (In Eng.) doi: https://doi.org/10.1145/3015146

[21] Chernishev G.A., Galaktionov V.A., Grigorev V.D., Klyuchikov E.S., Smirnov K.K. PosDB: An Architecture Overview. *Programming and Computer Software.* 2018; 44(1):62-74. (In Eng.) doi: https://doi.org/10.1134/S0361768818010024

[22] Funke H., Breß S., Noll S., Markl V., Teubner J. Pipelined Query Processing in Coprocessor Environments. *Proceedings of the 2018 International Conference on Management of Data (Houston, TX, USA) (SIGMOD'18).* Association for Computing Machinery, New York, NY, USA; 2018. p. 1603-1618. (In Eng.) doi: https://doi.org/10.1145/3183713.3183734

[23] Răducanu B., Boncz P., Zukowski M. Micro Adaptivity in Vectorwise. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, New York, USA) (SIGMOD'13).* Association for Computing Machinery, New York, NY, USA; 2013. p. 1231-1242. (In Eng.) doi: https://doi.org/10.1145/2463676.2465292

[24] Leis V., Boncz P., Kemper A., Neumann T. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD'14).* Association for Computing Machinery, New York, NY, USA; 2014. p. 743-754. (In Eng.) doi: https://doi.org/10.1145/2588555.2610507

[25] Diaconu C., Freedman C., Ismert E., Larson P.-A., Mittal P., Stonecipher R., Verma N., Zwilling M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, New York, USA) (SIGMOD'13).* Association for Computing Machinery, New York, NY, USA; 2013. p. 1243-1254. (In Eng.) doi: https://doi.org/10.1145/2463676.2463710

[26] Chen J., Jindel S., Walzer R., Sen R., Jimsheleishvilli N., Andrews M. The MemSQL Query Optimizer: A Modern Optimizer for Real-Time Analytics in a Distributed Database. *Proceedings of the VLDB Endowment*. 2016; 9(13):1401-1412. (In Eng.) doi: https://doi.org/10.14778/3007263.3007277

[27] Boncz P., Zukowski M., Nes N. MonetDB/X100: Hyper-Pipelining Query Execution. *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05).* Asilomar, CA, USA; 2005. Available at: http://cidrdb.org/cidr2005/papers/P19.pdf (accessed 13.10.2021). (In Eng.)

[28] Psaroudakis I., Scheuer T., May N., Sellami A., Ailamaki A. Scaling up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement. *Proceedings of the VLDB Endowment.* 2015; 8(12):1442-1453. (In Eng.) doi: https://doi.org/10.14778/2824032.2824043

[29] Graefe G. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*. 1995; 18(3):19-29. Available at: https://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/Papers/Cascades-graefe.pdf (accessed 13.10.2021). (In Eng.)

[30] Leis V., Gubichev A., Mirchev A., Boncz P., Kemper A., Neumann T. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment*. 2015; 9(3):204-215. (In Eng.) doi: https://doi.org/10.14778/2850583.2850594

[31] Cormode G., Garofalakis M., Haas P.J., Jermaine C. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*. 2012; 4(1-3):1-294. (In Eng.) doi: https://doi.org/10.1561/1900000004

[32] Sun J., Li G. An End-to-End Learning-Based Cost Estimator. *Proceedings of the VLDB Endowment*. 2019; 13(3):307-319. (In Eng.) doi: https://doi.org/10.14778/3368289.3368296

[33] Marcus R., Negi P., Mao H., Zhang C., Alizadeh M., Kraska T., Papaemmanouil O., Tatbul N. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment*. 2019; 12(11):1705-1718. (In Eng.) doi: https://doi.org/10.14778/3342263.3342644

[34] Negi P., Interlandi M., Marcus R., Alizadeh M., Kraska T., Friedman M., Jindal A. Steering Query Optimizers: A Practical Take on Big Data Workloads. *Proceedings of the 2021 International Conference on Management of Data (Virtual Event, China) (SIGMOD/PODS'21).* Association for Computing Machinery, New York, NY, USA; 2021. p. 2557-2569. (In Eng.) doi: https://doi.org/10.1145/3448016.3457568

[35] Horng J.-T., Kao C.-Y., Liu B.-J. A genetic algorithm for database query optimization. *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.* Vol. 1. IEEE Computer Society; 1994. p. 350-355. (In Eng.) doi: https://doi.org/10.1109/ICEC.1994.349926

[36] Lutz C., Breß S., Zeuch S., Rabl T., Markl V. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD'20).* Association for Computing Machinery, New York, NY, USA; 2020. p. 1633-1649. (In Eng.) doi: https://doi.org/10.1145/3318464.3389705

[37] Shanbhag A., Madden S., Yu X. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD'20).* Association for Computing Machinery, New York, NY, USA; 2020. p. 1617-1632. (In Eng.) doi: https://doi.org/10.1145/3318464.3380595

[38] He B., Lu M., Yang K., Fang R., Govindaraju N.K., Luo Q., Sander P.V. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems.* 2009; 34(4):21. (In Eng.) doi: https://doi.org/10.1145/1620585.1620588

[39] Heimel M., Saecker M., Pirk H., Manegold S., Markl V. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *Proceedings of the VLDB Endowment*. 2013; 6(9):709-720. (In Eng.) doi: https://doi.org/10.14778/2536360.2536370

[40] Stone J.E., Gohara D., Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*. 2010; 12(3):66-73. (In Eng.) doi: https://doi.org/10.1109/MCSE.2010.69

[41] Breß S., Köcher B., Funke H., Zeuch S., Rabl T., Markl V. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal.* 2018; 27(6):797-822. (In Eng.) doi: https://doi.org/10.1007/s00778-018-0512-y

[42] Kohn A., Leis V., Neumann T. Making Compiling Query Engines Practical. *IEEE Transactions on Knowledge and Data Engineering.* 2021; 33(2):597-612. (In Eng.) doi: https://doi.org/10.1109/TKDE.2019.2905235

[43] Pirk H., Moll O., Zaharia M., Madden S. Voodoo – a Vector Algebra for Portable Database Performance on Modern Hardware. *Proceedings of the VLDB Endowment*. 2016; 9(14):1707-1718. (In Eng.) doi: https://doi.org/10.14778/3007328.3007336

[44] Shaikhha A., Klonatos Ya., Koch C. Building Efficient Query Engines in a High-Level Language. *ACM Transactions on Database Systems*. 2018; 43(1):4. (In Eng.) doi: https://doi.org/10.1145/3183653

[45] Chrysogelos P., Karpathiotakis M., Appuswamy R., Ailamaki A. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proceedings of the VLDB Endowment*. 2019; 12(5):544-556. (In Eng.) doi: https://doi.org/10.14778/3303753.3303760

[46] Gubner T., Boncz P. Charting the Design Space of Query Execution Using VOILA. *Proceedings of the VLDB Endowment*. 2021; 14(6):1067-1079. (In Eng.) doi: https://doi.org/10.14778/3447689.3447709

[47] Palkar S., Thomas J.J., Shanbhag A., Narayanan D., Pirk H., Schwarzkopf M., Amarasinghe S., Zaharia M. Weld: A Common Runtime for High Performance Data Analytics. *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'17).* Chaminade, California, USA; 2017. Available at: https://www.cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf (accessed 13.10.2021). (In Eng.)

[48] Zhang S., He J., He B., Lu M. OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures. *Proceedings of the VLDB Endowment.* 2013; 6(12):1374-1377. (In Eng.) doi: https://doi.org/10.14778/2536274.2536319

Поступила 13.10.2021; одобрена после рецензирования 23.11.2021; принята к публикации 30.11.2021.
Submitted 13.10.2021; approved after reviewing 23.11.2021; accepted for publication 30.11.2021.

Об авторах:

Курапов Петр Александрович, аспирант, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0000-0001-6774-7588,** kurapov@phystech.edu

Мелик-Адамян Арег Фрикович, руководитель отдела Cloud-Native Languages and Runtimes, Intel Corporation (CA 95054-1549, Соединенные Штаты Америки, шт. Калифорния, г. Санта-Клара, бульвар Мишн-Колледж, д. 2200), кандидат технических наук, **ORCID: https://orcid.org/0000-0003-2140-6221,** areg.melik-adamyan@intel.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Petr A. Kurapov, Postgraduate Student, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0000-0001-6774-7588,** kurapov@phystech.edu

Areg F. Melik-Adamyan, Senior Director, Cloud-Native Languages and Runtimes, Intel Corporation (2200 Mission College Blvd., Santa Clara CA 95054-1549, California, United States of America), Cand. Sci. (Tech.), **ORCID: https://orcid.org/0000-0003-2140-6221,** areg.melik-adamyan@intel.com

All authors have read and approved the final manuscript.