На правах рукописи

Бучацкий Рубен Артурович

Метод динамической компиляции SQL-запросов для реляционных СУБД

Специальность 2.3.5 —
«Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2022

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской Академии Наук.

Научный руководитель: д.ф.-м.н., академик РАН
Аветисян Арутюн Ишханович

Официальные оппоненты: **Волконский Владимир Юрьевич,**
кандидат технических наук,
заместитель генерального директора по науке – направление «Системы программирования» ПАО «Институт электронных управляющих машин им. И.С. Брука»

Соколинский Леонид Борисович,
доктор физико-математических наук, профессор, проректор по информатизации ФГАОУ ВО «Южно-Уральский государственный университет (национальный исследовательский университет)»

Ведущая организация: Федеральное государственное бюджетное учреждение науки Институт проблем управления им. В.А. Трапезникова Российской академии наук

Защита состоится 8 декабря 2022 г. в 16 часов на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской Академии Наук по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования им. В.П. Иванникова РАН.

Отзывы на автореферат в двух экземплярах, заверенные печатью учреждения, просьба направлять по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25, учёному секретарю диссертационного совета 24.1.120.01.

Автореферат разослан «__» ________ 2022 года.

Учёный секретарь
диссертационного совета
24.1.120.01,
к.ф.-м.н. Зеленов С.В.

Общая характеристика работы

Актуальность темы. Системы управления реляционными базами данных (СУБД) широко используются в современных приложениях для обработки и анализа данных и позволяют выполнять сложные запросы на больших объемах данных. Эффективность обработчика запросов СУБД имеет решающее значение для большинства поддерживаемых СУБД приложений. Работы по улучшению производительности реляционных СУБД традиционно были направлены на оптимизацию доступа к памяти, однако в последние годы эффективное использование процессора является решающим фактором производительности аналитических систем. Это связано со снижением стоимости памяти на несколько порядков за последние десятилетия, что сделало возможным для современных СУБД хранить большую часть (если не всех) данных в основной памяти, тем самым отводя традиционные для большинства систем обработки данных оптимизации подсистемы ввода-вывода на второй план. Для сложных запросов дальнейшее ускорение их исполнения возможно за счёт оптимизации вычислений, выполняемых на процессоре, в том числе с применением компиляторных оптимизаций.

В большинстве реляционных СУБД для SQL-запроса сначала строится план выполнения (или план запроса), который по существу представляет собой дерево реляционных операторов, а затем выполняется его *интерпретация* для получения результата. Наиболее распространенным способом выполнения запросов, используемым в интерпретаторах классических систем обработки данных, является *модель итераторов*, также известная как Volcano-модель. В рамках данной модели каждый алгебраический оператор в дереве плана запроса (начиная с корня) извлекает и обрабатывает по одному кортежу за раз из своих дочерних узлов-операторов. Volcano-модель выполнения запросов используется в большинстве современных СУБД, таких как PostgreSQL, MySQL, SQLite и других.

Модель итераторов позволяет упростить как построение и оптимизацию плана запроса, так и реализацию реляционных операторов в отдельности. Однако, как показывают исследования, неэффективное расходование ресурсов процессора при использовании модели итераторов является узким местом для систем, хранящих данные в оперативной памяти (in-memory). Это связанно с использованием вызовов виртуальных функций и многократным сохранением и загрузкой состояний операторов в ходе интерпретации плана, состоящего из произвольной последовательности операторов, выражений и предикатов, что, в свою очередь, порождает значительное количество ложных предсказаний переходов и неэффективное использование кэша инструкций.

Одним из способов повышения эффективности использования процессора и сокращения накладных расходов, прежде всего проявляющихся

в затратах на интерпретацию запросов, является *динамическая компиляция*. Существующие методы динамической компиляции, применяемых в СУБД, можно разделить на два класса: *компиляция выражений и горячих участков кода* — компиляция таких частей запросов, как арифметические и логические выражения и доступ к атрибутам, при сохранении интерпретации плана запроса; *компиляция запросов целиком*, что подразумевает замену этапа интерпретации на генерацию по плану запроса специализированного кода в промежуточном представлении с дальнейшей трансляцией во время исполнения в машинный код, оптимизированный с учётом структуры конкретного запроса, используемых в нём типов данных, функций и параметров базы данных. Ускорение при динамической компиляции в основном достигается за счёт уменьшения количества инструкций, выполняемых процессором за один запрос, благодаря применению оптимизаций.

В последнее время актуальным становится применение методов динамической компиляции для ускорения выполнения запросов, такие методы разрабатываются как в академических, так и в прикладных и коммерческих системах.

Динамическая компиляция при сохранении Volcano-модели не избавляет от основных недостатков, присущих данной модели, а лишь позволяет нивелировать накладные расходы на исполнение обобщённого кода СУБД. Альтернативной моделью выполнения запросов является *модель явных циклов* или push-модель, которая используется во многих современных коммерческих резидентных СУБД для реализации динамической компиляции запросов, например в SQL Server (Hekaton), MemSQL (SingleStore), HyPer и т.д. В push-модели дочерний оператор возвращает результирующие кортежи родительскому оператору посредством вызова соответствующего обработчика, принимающего очередной кортеж и продолжающего его обработку. Здесь процессом выполнения плана запроса управляет не корневой, а один из листовых операторов сканирования таблицы или индекса. Динамический компилятор запросов получает преимущество от применения модели явных циклов и позволяет представить план в виде последовательности вложенных циклов, где самым первым и внешним является цикл сканирования таблицы. В этом случае повторная загрузка и сохранение состояния оператора не требуются.

Однако современные динамические компиляторы запросов не применимы к использующейся в большинстве СУБД Volcano-модели, соответственно *актуальной* является задача разработки метода динамической компиляции запросов в модели явных циклов, которая будет применима к СУБД с моделью Volcano. Решение этой задачи сводится к переходу из имеющейся Volcano-модели к модели явных циклов путём трансляции алгоритмов операторов по дереву плана запроса из Volcano-модели в код на

промежуточном представлении в модели явных циклов (в обратном порядке, где выполнение начинается с одного из листовых узлов сканирования) с дальнейшей оптимизацией и компиляцией в машинный код.

Динамическая компиляция запросов оправдана только в том случае, когда время интерпретации запроса превосходит суммарное время компиляции и выполнения оптимизированного кода. Данное требование может быть удовлетворено, когда объем обрабатываемых запросом данных достаточно велик, так как затраты на динамическую компиляцию могут в несколько раз превосходить время выполнения скомпилированного кода запроса. В данном контексте актуальна задача разработки эвристик на основе оценок стоимости динамической компиляции для принятия решения об интерпретации, компиляции или кэшировании сгенерированного динамическим компилятором кода с целью уменьшения накладных расходов на компиляцию запросов.

Целью данной работы является разработка и реализация метода динамической компиляции SQL-запросов в реляционных СУБД для оптимизации выполнения запросов для более эффективного использования процессора за счёт трансформации операторов плана запроса из Volcano-модели в модель явных циклов.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Разработать метод трансформации на лету операторов плана запроса из Volcano-модели в модель явных циклов для выполнения запросов с использованием динамического компилятора.
2. Разработать метод динамической компиляции выражений в SQL-запросах, который позволит использовать одну и ту же реализацию встроенных функций СУБД совместно с интерпретатором выражений за счёт применения открытой вставки функций и их совместной оптимизации, а также с использованием предкомпиляции в промежуточное представление встроенных функций СУБД.
3. Разработать эвристики стратегии выполнения запроса с учётом стоимости интерпретации и выполнения скомпилированного запроса; разработать метод кэширования кода, сгенерированного динамическим компилятором, для повторного применения в SQL-запросах.
4. Реализовать разработанный метод динамической компиляции SQL-запросов для СУБД и произвести оценку эффективности с точки зрения производительности.

Научная новизна:

1. Разработан метод динамической компиляции запросов с трансформацией на лету операторов плана запроса из модели Volcano в модель явных циклов.

2. Разработан метод динамической компиляции выражений в SQL-запросах с применением открытой вставки предварительно скомпилированных встроенных функций СУБД.
3. Разработаны эвристики стратегии выполнения запроса на основе оценок затрат на динамическую компиляцию. Также разработан метод сохранения и переиспользования кода, сгенерированного динамическим компилятором, в SQL-запросах.
4. Разработано расширение к СУБД предложенных методов для динамической компиляции SQL-запросов.

Теоретическая и практическая значимость. Теоретическая значимость работы заключается в разработке метода генерации машинного кода специализированного под конкретный запрос к реляционной СУБД с трансформацией операторов плана запроса из Volcano-модели в модель явных циклов. Предложенный метод динамической компиляции может быть применён к СУБД с Volcano моделью выполнения для реализации динамического компилятора запросов.

Разработанный метод динамической компиляции запросов реализован с использованием компиляторной инфраструктуры LLVM в виде расширения к СУБД с открытым исходным кодом PostgreSQL и не требует изменения исходного кода самой СУБД. Исходный код разработанного динамического компилятора выражений опубликован в открытом доступе[1], а реализованные в нем элементы были использованы сообществом разработчиков СУБД PostgreSQL при реализации динамического компилятора выражений в версии 11.

Реализованный динамический компилятор запросов внедрён в научно-исследовательские и учебные проекты ИСП РАН, а также в компанию ООО «РусБИТех-Астра» для ускорения выполнения запросов СУБД PostgreSQL.

Методология и методы исследования. Результаты диссертационной работы получены с использованием методов и моделей, используемых при трансляции и оптимизации кода программ. Математическую основу данной работы составляют теория графов, теория множеств, теория алгоритмов и теория автоматов.

Основные положения, выносимые на защиту:

1. Метод динамической компиляции SQL-запросов в модели явных циклов в оптимизированный машинный код для реляционной СУБД.
2. Метод трансформации алгоритмов операторов плана запроса из Volcano-модели в модель явных циклов для выполнения запросов в рамках динамического компилятора.
3. Метод динамической компиляции выражений в SQL-запросах.

[1] https://github.com/ispras/postgres

4. Эвристики стратегии выполнения плана запроса и метод кэширования кода, сгенерированного динамическим компилятором SQL-запросов.
5. На основе предложенных методов реализован динамический компилятор SQL-запросов в качестве программного расширения к СУБД с открытым исходным кодом PostgreSQL с использованием компиляторной инфраструктуры LLVM. Проведена апробация реализованных методов на промышленных тестовых наборах TPC-H и TPC-DS.

Апробация работы. Результаты работы обсуждались на следующих конференциях:

– Конференция PgConf.Russia 2016, Москва, 3 – 5 февраля 2016 г.
– Международная конференция LLVM Cauldron 2016, Йоркшир, Великобритания, 8 сентября 2016 г.
– Международная конференция PostgreSQL Conference Europe 2016, Эстония, Таллин, 1 – 4 ноября 2016 г.
– Конференция «Технологии Баз Данных» 2016, Москва, 29 – 30 ноября 2016г.
– Научно-практическая «Открытая конференция ИСП РАН» 2016, Москва, 1 – 2 декабря 2016 г.
– Конференция PgConf.Russia 2017, Москва, 15 – 17 марта 2017 г.
– Международная конференция PGCon 2017, Оттава, Канада, 23 – 26 мая 2017 г.
– Международная конференция «Иванниковские чтения» 2019, Великий Новгород, 13 – 14 сентября 2019 г.

Личный вклад. Все представленные в работе результаты получены лично автором.

Публикации. По теме диссертации опубликовано 7 научных работ. Работы [1—4] опубликованы в журнале, входящем в список ВАК. Работы [5—7] опубликованы в научных журналах, индексируемых системами Web of Science и Scopus. Получено свидетельство о регистрации программы для ЭВМ [8].

В работах [1; 2] автору принадлежит описание метода динамической компиляции запросов и выражений и их реализация в СУБД с открытым исходным кодом PostgreSQL, в работе [3] — обзор современных компиляторов запросов, использующих модель явных циклов. В работах [5; 6] автору принадлежит реализация push-модели для СУБД PostgreSQL в компиляторно-независимой форме, что позволило использовать ее в тандеме с специализатором кода. В работах [4; 7] личный вклад автора заключается в описании метода кэширования скомпилированного кода запроса и её реализация в разработанном динамическом компиляторе запросов для СУБД PostgreSQL.

Диссертационная работа была выполнена при поддержке следующих грантов:

– Грант РФФИ 17-07-00759 А «Динамическая компиляция SQL-запросов для СУБД».
– Грант РФФИ 20-07-00877 А «Разработка специализированных методов оптимизации в динамическом компиляторе SQL-запросов для СУБД».

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, ставятся цели и задачи диссертационной работы, формулируется научная новизна и практическая значимость представляемой работы, а также приводятся основные положения, выносимые на защиту.

Первая глава посвящена обзору предметной области по применению методов динамической компиляции в современных СУБД. Производится обзор литературы и программных средств, связанных с решаемой задачей.

Описывается процесс обработки запросов в СУБД, который состоит из этапов лексического и синтаксического анализ, семантического анализа, обработки системой правил, этапов планирования и оптимизации и выполнения итогового плана запроса. На этапе планирования и оптимизации планировщик получает на вход дерево запроса и осуществляет выбор наиболее эффективного пути выполнения с точки зрения оценок затрат и информации на момент выполнения. Строится полноценный план запроса и передаётся исполнителю. Исполнитель осуществляет рекурсивный обход по дереву плана и выполняет инкапсулированную логику соответствующего узла-оператора или выражения, формулируя на выходе ответ на запрос.

Большинство СУБД используют описанный подход для трансляции запроса пользователя сначала в логический план, представляющий собой дерево из операторов расширенной реляционной алгебры, а затем в физический, который выполняется исполнителем.

В разделе 1.1 описывается четыре основных *модели выполнения запросов в СУБД*, определяющие, как система выполняет план запроса: модель итераторов (Volcano или pull, кортеж за раз); материализующая модель (оператор за раз, колонка за раз); векторизующая модель (векторная, блочная); модель явных циклов (push, кортеж за раз).

Интерфейс оператора в *модели Volcano* состоит из трёх методов: *open*, *next* и *close*. Метод *open* инициализирует внутреннее состояние оператора, выделяет необходимые ресурсы и подготавливает оператор к созданию первого кортежа. Метод *next* создаёт и возвращает один следующий кортеж, если он есть, и возвращает $NULL$, когда созданы все кортежи. Метод *close*

сбрасывает состояние оператора и окончательно высвобождает все ресурсы, необходимые оператору. Таким образом, оператор в модели Volcano используется как итератор, предоставляющий доступ к последовательности возвращаемых кортежей. Функцией каждого оператора является формирование последовательности возвращаемых кортежей из последовательностей кортежей, принимаемых на вход от дочерних операторов.

Основные недостатки модели Volcano: невозможность применения оптимизации встраивания и плохое предсказание переходов из-за того, что функция *next()* является виртуальной и часто реализуется как косвенный вызов (по указателю), что плохо сочетается с конвейером современных ЦП; плохая локальность по коду и данным из-за частого сохранения и восстановления состояния оператора между вызовами функции *next()*.

В рамках *материализующей модели* каждый оператор выполняет обработку всех входных данных целиком, сохраняя промежуточный результат в оперативной или внешней памяти, а затем передаёт его следующему оператору. Функция *next()* оператора, которая в модели Volcano возвращает один кортеж, в данной модели возвращает все свои кортежи за раз. Однако для OLAP-нагрузки данная модель подходит плохо, так как промежуточные результаты операторов, в зависимости от их размера, могут быстро исчерпать весь объём доступной оперативной памяти, в таком случае СУБД, возможно, придётся сохранять на диск эти результаты для дальнейшей передачи между операторами.

В *векторизующей модели* каждый оператор реализует интерфейс взаимодействия в виде функции *next()*, однако ключевое отличие заключается в количестве передаваемых кортежей за один вызов *next()*. Внутренний алгоритм каждого оператора может обрабатывать и передавать целую партию (или вектор) кортежей за раз, варьируя её размер в зависимости от характеристик аппаратного обеспечения, доступных ресурсов и рекомендаций оптимизатора запросов. Зачастую эта модель сочетается с колоночной организацией данных для максимальной утилизации векторных расширений, однако может использоваться и в традиционных строчно-ориентированных решениях.

Материализующая и векторизующая модели пытаются преодолеть проблему модели Volcano с количеством вызовов функции $next()$ одинаковым способом — используя больше памяти. В свою очередь, это означает, что упомянутые оптимизации нивелируют основное преимущество модели итераторов — возможность конвейеризации процесса обработки данных. В некоторых случаях конвейеризация позволяет оператором избегать излишних операций копирования и материализации передаваемых данных, что положительно сказывается на объёме используемой памяти.

Существует альтернатива Volcano-модели, которую мы будем называть *моделью явных циклов*, также данную модель называют *data-centric*

или *push*. Основное её отличие от модели итераторов заключается в направлении обработки данных: дерево операторов обходится не сверху вниз, а снизу вверх. Кортежи передаются в вышестоящий оператор без «запроса» сверху в виде функции $next()$, а «проталкиваются» снизу вверх. Процесс инициализации и освобождения ресурсов не меняется: методы *open()* и *close()* остаются и сохраняют свой смысл. Выполнение начинается снизу дерева, листовые операторы в одном цикле сканируют таблицу и передают кортежи своим родителям, те — своим, и так до корня. С точки зрения эффективности, подобный «разворот» направления потока обработки данных решает только проблему локальности по данным, то есть с частым сохранением и восстановлением состояния оператора, но не избавляет от основных накладных расходов, присущих модели Volcano. Модель явных циклов используется при реализации динамических компиляторов запросов.

В разделе 1.2 описываются *существующие методы динамической компиляции*, применяемые в СУБД. Проводится обзор работ.

Динамическая компиляция выражений позволяет избежать накладных расходов на исполнение обобщённого кода СУБД благодаря компиляции некоторых «горячих» функций СУБД в машинный код во время выполнения с учётом конкретного запроса. Во всех рассмотренных динамических компиляторах для распределённых систем компиляция реализуется на уровне выражений и горячих функций (Apache Impala, Spark SQL). Показателен подход Greenplum, основанный на замене указателей в структурах данных на указатели в результирующий машинный код, изначально указывающих на обобщённый код, во время подготовки запроса к выполнению. Динамическая компиляция выражений позволяет получить ускорение в несколько раз на простых запросах, а на TPC-H — до 30%.

Основной подход к реализации *динамических компиляторов запросов* — реализация генератора низкоуровневого кода запроса на основе модели явных циклов (HyPer, Hekaton, MemSQL). Реализация модели явных циклов в рамках динамического компилятора запросов описана в работе, посвящённой архитектуре исполнителя СУБД HyPer. Она состоит в генерации по плану выполнения запроса одного блока императивного кода со вложенными циклами, в котором одна из листовых вершин плана выполнения (сканирование таблицы или индекса) является самым внешним циклом, а из самого внутреннего цикла вызывается код, отвечающий за приём кортежей самой внешней (корневой) вершиной плана. На синтетических запросах динамическая компиляция в HyPer позволяет получить ускорение до 8 раз в сравнении с интерпретацией, на бенчмарке TPC-H — до 3.7 раз в сравнении с колоночной СУБД VectorWise.

Однако такая генерация кода является неоптимальной, так как сгенерированный код сложнее проверять и отлаживать, потому что границы между операторами размыты, также допускается раздувание кода: один и тот же фрагмент кода может генерироваться множество раз в одном блоке.

Также из обзора можно отметить, что для реализации динамического компилятора в большинстве работ выбрана инфраструктура LLVM, предоставляющая широкий набор инструментов для создания компиляторов.

Вторая глава посвящена разработке *метода динамической компиляции SQL-запросов* с трансформацией на лету операторов плана запроса из модели Volcano в модель явных циклов.

Рассмотренные в обзоре динамические компиляторы запросов не применимы к использующейся в большинстве современных СУБД Volcano-модели, соответственно актуальной является задача разработки метода динамической компиляции запросов, которая будет применима к СУБД с моделью Volcano. Для этого необходимо организовать переход к модели явных циклов, где направление обработки данных не сверху вниз, как в Volcano-модели, а снизу вверх, а также адаптировать имеющиеся реализации алгоритмов операторов в Volcano-модели под модель явных циклов в рамках динамического компилятора без изменения кода интерпретатора.

Метод динамической компиляции запросов подразумевает замену этапа интерпретации на генерацию по плану запроса специализированного низкоуровнего кода запроса в модели явных циклов с дальнейшей оптимизацией и трансляцией в машинный код во время выполнения. Общая схема метода показана на рисунке 1.

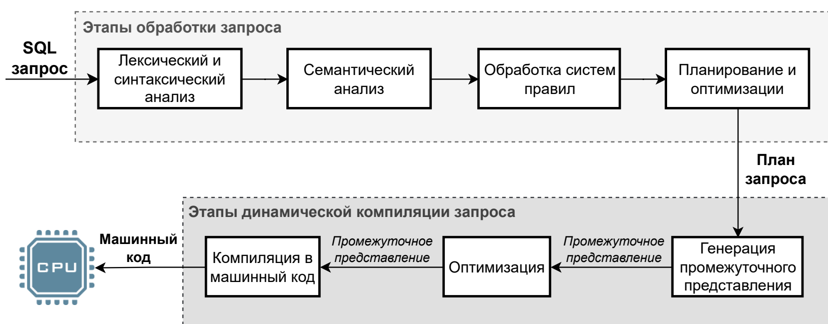

Рис. 1 — Процесс обработки запроса в динамическом компиляторе.

Предлагаемый метод состоит из трёх последовательных этапов: генерация во время выполнения кода запроса в промежуточном представлении в модели явных циклов, оптимизация и компиляция в машинный код. Входным параметром для первого этапа является план запроса.

Для генерации кода в модели явных циклов необходимо:

– Определить интерфейс оператора в модели явных циклов.
– Декомпозировать алгоритм операторов Volcano-модели на функции интерфейса в модели явных циклов.
– Реализовать функции-генераторы интерфейса операторов в модели явных циклов.

Процесс генерация кода в модели явных циклов состоит из одного рекурсивного прохода по дереву плана запроса, в процессе которого вызываются функции-генераторы соответствующих узлов-операторов СУБД для генерации на основе имеющейся реализации алгоритмов операторов в Volcano-модели специализированного промежуточного представления плана запроса в модели явных циклов:

1. Для каждого нелистового узла в дереве плана генерируются две функции:
   – *consume()*, которая вызывается один раз для каждого нового кортежа, созданного из дочернего узла, и
   – *finalize()*, которая вызывается после создания последнего кортежа из дочернего узла.

   Для листового узла в дереве плана генерируется одна функция:
   – *main()*, которая вызывает по цепочке функции *consume()* и *finalize()* от родительских узлов.
2. Сгенерированные функции *consume()* и *finalize()* передаются соответствующим дочерним узлам и вызываются из функций, сгенерированных для этих узлов.
3. Каждый узел возвращает сгенерированную функцию одного из своих дочерних узлов, в зависимости от того, какой цикл сканирования является внешним для этого конкретного узла.

Результатом генерации является набор функций в промежуточном представлении (из которых одна функция выделена как точка входа), представляющий специальный интерфейс операторов запроса в модели явных циклов. Такая генерация функций интерфейса позволяет трансформировать на лету операторы плана запроса из модели Volcano в модель явных циклов, реализовывать новые операторы и совмещать несколько операторов в рамках одного запроса. Для выделения этого интерфейса необходимо декомпозировать имеющийся алгоритм оператора в Volcano-модели с сохранением корректности на соответствующие функции интерфейса в модели явных циклов. Преимуществом генерации отдельных функций является то, что, если оптимизации не выполняются, эти функции могут быть проверены и отлажены с использованием существующих отладчиков.

После первого этапа над сгенерированным промежуточным представлением проводятся межоператорные *оптимизации*, выполняется встраивание функций. Код запроса после оптимизаций представляется в виде последовательности вложенных циклов, где самым первым и внешним является цикл сканирования таблицы.

На третьем этапе выполняется *компиляция* сгенерированного оптимизированного промежуточного представления запроса в машинный код целевой архитектуры.

На рисунке 2 представлен процесс генерации кода в модели явных циклов на примере дерева плана запроса, где генерируется код в промежуточном представлении по плану запроса функций интерфейса операторов в модели явных циклов. Как можно заметить, ключевую роль для получения кода в виде вложенных циклов играет оптимизация по встраиванию функций, которая встраивает всю цепочку вызовов функций *consume()*.

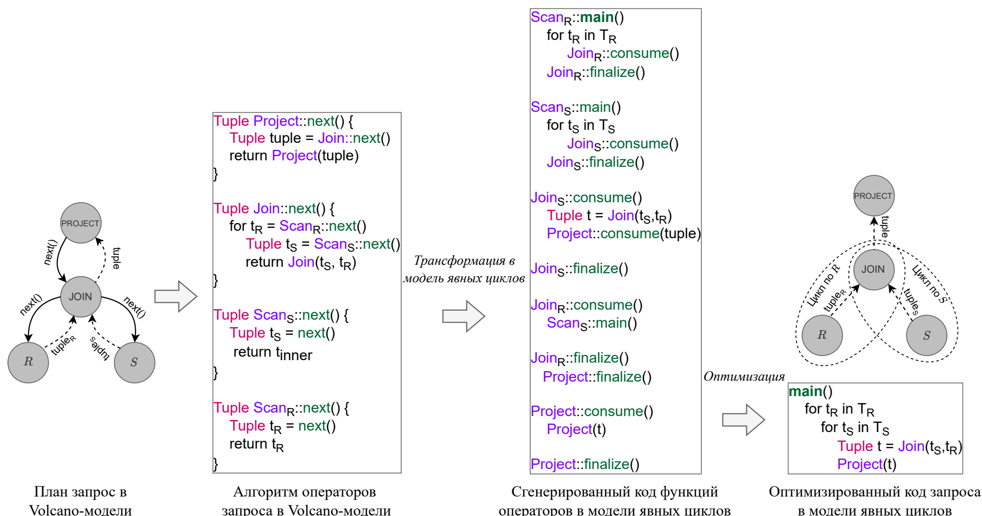

Рис. 2 — Схема трансформации алгоритмов операторов плана запроса в модель явных циклов

Задача генерации специализированного кода по плану запроса в модели явных циклов сводится к трансляции алгоритмов операторов по дереву плана запроса из Volcano-модели в код на промежуточном представлении в модели явных циклов (в обратном порядке, где выполнение начинается с одного из листовых узлов сканирования).

В разделе 2.1 описывается *метод трансформации операторов Volcano-модели в модель явных циклов* в контексте динамической компиляции, который состоит из следующих этапов:

1. описание интерфейса оператора в модели явных циклов, который позволит трансформировать операторы плана запроса из модели Volcano в модель явных циклов,
2. выделение интерфейса для операторов СУБД путём декомпозиции, на основе имеющейся реализации, алгоритма операторов Volcano-модели на функции интерфейса в модели явных циклов.

Далее определяется *интерфейс оператора в модели явных циклов* и принцип выделения этого интерфейса в контексте динамического компилятора, необходимого для «разворота» потока обработки данных. Для нелистовых операторов ($Operator \notin ScanOperator$) интерфейс состоит из двух функций: *consume* и *finalize* — со следующими сигнатурами:

Operator

consume() : *int32*

finalize() : *int32*

consume() — это функция «уведомитель-обработчик», которая уведомляет вызывающую сторону о том, что дочернее поддерево произвело новый кортеж.

finalize() — это функция «уведомитель-завершитель», которая используется для уведомления о том, что дочернее поддерево закончило производство кортежей.

Для листовых операторов сканирования таблицы (*ScanOperator*), интерфейс состоит из одной функции, имеющей следующую сигнатуру:

ScanOperator

main() : *int32*

main() — это функция, которая является входной точкой для вызова по цепочке функций *consume()* и *finalize()* от вышестоящих операторов, реализующих код операторов запроса в модели явных циклов.

Фактически же эти функции представляют реализацию алгоритма оператора СУБД в модели явных циклов и являются ключевым механизмом для трансформации операторов Volcano-модели в модель явных циклов для выполнения плана запроса в обратном порядке. Код функций интерфейса в модели явных циклов генерируется во время выполнения на основе имеющейся реализации алгоритма оператора в Volcano-модели.

В разделе 2.1.2 описывается процесс *генерации функций интерфейса* оператора в модели явных циклов. Для реализации модели явных циклов у каждого оператора определены функции-генераторы, которые на основе имеющейся реализации алгоритмов операторов в модели Volcano, «на лету» выполняют генерацию кода в промежуточном представлении описанных ранее функций интерфейса оператора *main()*, *consume()* и *finalize()* во время рекурсивного обхода дерева плана запроса в прямом порядке. Для каждого оператора функция-генератор вызывается один раз. Результатом генерации является одна императивная программа для каждого дерева плана запроса. Генерация промежуточного представления происходит на основе имеющейся реализации алгоритма соответствующего оператора в конкретной СУБД. В процессе генерации кода используется информация, доступная только во время выполнения, в частности константные параметры, такие как количество страниц, параметры атрибутов, направление обхода и т.д., что позволяет выполнять оптимизации по разворачиванию циклов, подстановке констант и прочие. Далее к сгенерированному промежуточному представлению применяются компиляторные оптимизации и выполняется компиляция в машинный код.

В разделе 2.1.3 описывается *механизм прерывания обработки запроса* в модели явных циклов, необходимый для досрочного завершения выполнения. Модель явных циклов имеет сложности с операторами,

прерывающими цикл обработки (например, с оператором-ограничителем результата Limit), так как по своей природе эта модель не позволяет досрочно завершать итерацию обработки запроса. Это связано с тем, что в этой модели операторы не могут контролировать, когда данные больше не должны генерироваться вышестоящим оператором. Для управления выполнением запросов в модели явных циклов был разработан механизм прерывания процесса обработки при помощи интерпретации возвращаемых функциями *consume()* и *finalize()* числовых статусов.

Результат вызова функций *consume()* и *finalize()* интерпретируется как количество циклов (последовательностью которых в модели явных циклов представляется код запроса), которые нужно завершить для продолжения обработки запроса. Возвращаемое значение статуса обрабатывается листовым оператором сканирования для принятия решения о продолжении или завершении потока обработки данных.

В разделе 2.1.4 описывается процесс *декомпозиции алгоритмов операторов* итеративной Volcano-модели на функции интерфейса в модели явных циклов, который генерирует код для данного запроса во время обхода в глубину дерева плана. В процессе декомпозиции алгоритмы операторов разбиваются на части: функцию *main()* для листовых узлов и функции *consume()* и *finalize()* для нелистовых узлов. Корректность алгоритма оператора после декомпозиции сохраняется за счёт генерации семантически эквивалентного промежуточного представления оператора на основе имеющейся реализации алгоритма данного оператора в СУБД.

Определим назначение каждой функции интерфейса:

- *consume()* должна содержать часть алгоритма оператора, отвечающую за логику обработки получаемого из дочернего поддерева кортежа. Дальнейшие действия зависят от типа оператора.
- *finalize()* должна использоваться для выполнения части алгоритма оператора, отвечающего за постобработку аккумулированных данных.
- *main()* для листовых операторов должна содержать логику операторов сканирования для получения кортежа. В случае, если листовых операторов несколько, то только одна из них будет являться точкой входа для выполнения запроса.

Основные шаги, необходимые для выделения интерфейса:

1. Установить значение арности оператора.
2. Классифицировать тип оператора, выделить его основные свойства и определить его функциональные возможности.
3. Выполнить декомпозицию алгоритма обработки на соответствующее количество экземпляров функций интерфейса *consume* и *finalize*. Если оператор нульарный, то будет функция *main*.

Процесс декомпозиции алгоритмов реляционных операторов модели Volcano в модель явных циклов зависит от реализации данного оператора в конкретной СУБД и не может быть полностью формализован для всех возможных операторов. Это связано с тем, что метод динамической компиляции должен сохранять обратную совместимость и семантическую целостность компилируемого запроса. Задача усложняется тем, что современные СУБД имеют большое количество разнообразного функционала, часть из которого может плохо вписываться в существующую архитектуру исполнителя, но реализована по причине удобства для пользователя.

Далее в этом разделе подробно описывается процесс декомпозиции алгоритмов основных типов операторов СУБД в модели Volcano на функции интерфейса в модели явных циклов.

Раздел 2.2 посвящён описанию *метода динамической компиляции выражений* в SQL-запросах. На этапе планирования СУБД строит дерево выражений, используемых в запросе, где каждое выражение состоит из дерева отдельных операторов и функций. Вычисление выражений широко применяется при выполнении плана запроса: сканирование таблиц, фильтры, агрегаты, проекции и объединения (которые не имеют поддержки индексов) по своей сути полагаются на интерпретатор выражений.

Для вычисления результата выражения во время выполнения запроса интерпретатор выражений СУБД обходит дерево выражений и для каждой вершины вызывает функции соответствующих дочерних вершин, реализующих обобщённый код конкретного узла. Эти вызовы выполняются неявным образом, через указатель на функцию, что приводит к большим накладным расходам во время выполнения. Для вычисления самих операций вызываются встроенные функции СУБД.

Метод динамической компиляции заключается в следующем: после получения дерева выражения выполняется рекурсивный обход этого дерева в обратном порядке и генерация кода в промежуточном представлении для каждой функции или операции; для встроенных функций СУБД, используемых в выражении, применяется оптимизирующее компиляторное преобразование по *открытой вставке* предварительно скомпилированных функций. Таким образом код для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без расходов на неявный вызов функций. Оптимизация открытой вставки выполняется во время генерации кода выражения, во время которой при генерации вызова встроенной функции производится поиск этой функции в списке предварительно скомпилированных функций СУБД и выполняется вставка кода функции в тело кода выражения. Применение открытой вставки функций, а также использование предкомпиляции встроенных функций СУБД позволит использовать в динамическом компиляторе одну и ту же реализацию встроенных функций совместно с интерпретатором выражений, тем самым нивелирую накладные расходы, связанные с вызовом этих функций.

В разделе 2.3 описываются разработанные *эвристики стратегии выполнения* с учётом стоимости интерпретации и выполнения скомпилированного кода запроса. Эвристики стратегии выполнения описаны в виде конечного автомата состояний, представленный на рисунке 3, который используется для обработки каждого входящего запроса. Пунктирные линии, соединяющие некоторые состояния, означают, что переход происходит только в случае ошибки. Состояния пунктирными линиями указывают, что кэшированный запрос может продолжить выполнение из этих состояний.

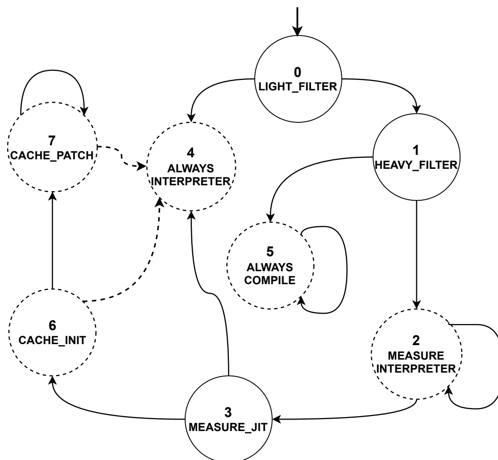

Рис. 3 — Конечный автомат состояний эвристик стратегии выполнения запроса в динамическом компиляторе.

Состояния 0 и 1 представляют предикат $C_{low} < Cost(P) < C_{high}$, где P — это план запроса, $Cost(P)$ — стоимость запроса, которая рассчитывается оптимизатором СУБД и представляет собой относительную цену для СУБД на выполнения этого запроса, C_{low} и C_{high} — это нижняя и верхняя оценка для принятия решения об интерпретации, анализе или компиляции, который означает, что быстрые запросы должны интерпретироваться, долгие запросы должны быть скомпилированы сразу, а остальные должны быть проанализированы для принятия решения. Если запрос удовлетворяет этому предикату и переходит в состояние 2 — `MEASURE_INTERPRETER`, то измеряется время его выполнения T_I интерпретатором.

После измерения времени интерпретации, выполнение запроса переходит в состояние 3 — `MEASURE_JIT`, где измеряется время выполнения скомпилированного запроса. В этом состоянии сравнивается время выполнения динамически скомпилированного кода (T_E) со временем интерпретатора (T_I) из состояния 2, и если $(\frac{T_I}{T_E} - 1) * 100 < K\%$, то есть если относительное ускорение составляет менее $K\%$, где K задаётся пользователем, то при последующих выполнениях этот запрос будет интерпретироваться (состояние 4 — `ALWAYS_INTERPRETER`). Если стоимость запроса больше, чем

верхняя оценка: $Cost(P) > C_{high}$, то запрос переходит в состояние 5 — `ALWAYS_COMPILE`, где сразу выполняется компиляция. Как только запрос достигает состояния 6 — `CACHE_INIT`, выполняется компиляция плана запроса и сохранение динамически скомпилированного кода, чтобы его можно было повторно использовать позже в состоянии 7 — `CACHE_PATCH`.

Эвристики устанавливают стратегии, когда интерпретация выполняется для очень простых запросов, кэширование используется для средних запросов, а долго выполняющиеся запросы всегда компилируются.

В разделе 2.3.1 описывается *метод кэширования кода*, сгенерированного динамическим компилятором запросов. Чтобы динамическая компиляция имела смысл, необходимо, чтобы время, затрачиваемое на интерпретацию запроса, превосходило время, затрачиваемое на компиляцию и выполнение оптимизированного кода. Данное требование может быть удовлетворено в случае, когда объем обрабатываемых запросом данных достаточно велик. Таким образом, только аналитические запросы типа OLAP могут нивелировать время, затрачиваемое на компиляцию, и позволяют получить прирост производительности. В случае OLTP запросов, где обрабатывается небольшой объем данных, метод динамической компиляции может оказаться неприемлемым по причине долгой компиляции.

Проблема решается путём кэширования сгенерированного динамическим компилятором кода. Однако не для всех запросов нужно сохранять код из-за расходов по обслуживанию кэшированного кода. Для этого применяются эвристики стратегии выполнения. Алгоритм работы метода кэширования динамически скомпилированного кода запроса следующий:

- После получения плана запроса выполняется поиск в структуре кэша сохранённого кода для этого плана.
- Если сохранённый код запроса был найден, то выполняется сопоставление сохранённой информации о схеме таблиц и атрибутов, используемых в запросе, выполняется загрузка кода и выполнение, тем самым экономя время, затрачиваемое на компиляцию запроса.
- Если сохранённого кода для этого запроса нет в структуре кэша, то выполняется генерация и компиляция кода плана запроса и сохранение его в структуре кэша вместе с информацией о схеме используемых в этом запросе таблиц, типе атрибутов и т.п.

Для управления структурой кэша выбран алгоритм кэширования LFU (Least-Frequently Used, Наименее часто используемый). Используется реализация алгоритма LFU на основе двунаправленного списка.

Третья глава посвящена реализации предлагаемого метода динамической компиляции запросов в виде программного расширения к СУБД с открытым исходным кодом PostgreSQL с использованием компиляторной инфраструктуры LLVM.

В разделе 3.1 описывается *устройство интерпретатора запросов СУБД PostgreSQL*, который для выполнения использует модель Volcano. В

разделе 3.2 описывается *компиляторная инфраструктура LLVM* для компиляции и оптимизации программ, которая используется для разработки динамического компилятора запросов для СУБД PostgreSQL.

В разделе 3.3 описывается *архитектура разработанного динамического компилятора запросов* для СУБД PostgreSQL. С использованием компиляторной инфраструктуры LLVM задача реализации динамического компилятора запросов сводится к разработке программного компонента, транслирующего алгоритм операторов по дереву плана запроса из модели Volcano в код на промежуточном представлении LLVM IR в модели явных циклов. Полученный код оптимизируется с использованием оптимизатора LLVM. Компиляция в машинный код и подготовка к выполнению также осуществляется средствами LLVM. Схема архитектуры реализованного динамического компилятора запросов представлена на рисунке 4.

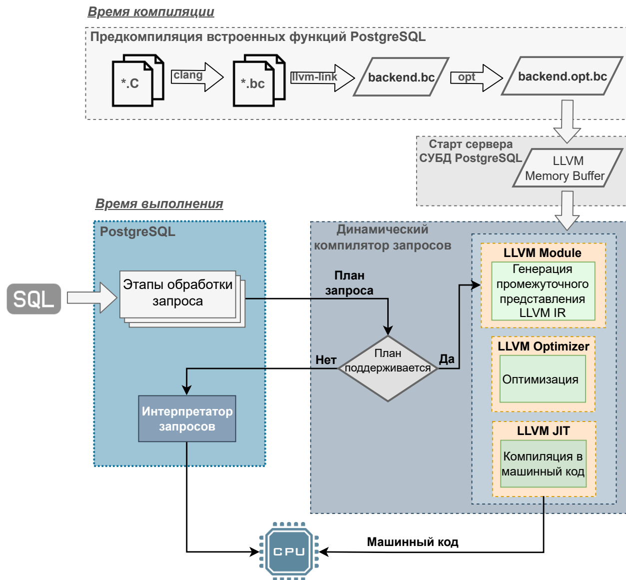

Рис. 4 — Архитектура динамического компилятора запросов PostgreSQL.

В процессе сборки расширения выполняется предварительная компиляция, оптимизация и компоновка встроенных функций PostgreSQL в единый LLVM-модуль, в котором содержатся все встроенные функции PostgreSQL в представлении LLVM IR. Метод предкомпиляции позволяет избавиться от ручной реализации функций-генераторов встроенных функции СУБД, необходимых при динамической компиляции.

Перед началом выполнения запроса динамическим компилятором производится проверка поддержки всех выражений и операторов, используемых в плане запроса. Далее выполняется обход плана запроса и генерация семантически эквивалентного внутреннего представления LLVM

IR исходного плана запроса с использованием программного интерфейса LLVM C API. Сгенерированный код оптимизируется и динамически компилируется модулем LLVM для динамической компиляции MCJIT или ORC JIT, а затем запускается на выполнение.

В динамическом компиляторе были реализованы с использованием LLVM C API большинство операторов СУБД PostgreSQL. Это позволило:

– заменить используемую вычислительную модель с Volcano на модель явных циклов;
– реализовать динамическую компиляцию и оптимизацию вычисления арифметических выражений и предикатов;
– разработать и реализовать ряд оптимизаций, возможных только в динамически компилируемом окружении.

В разделе 3.3.1 описывается *реализация модели явных циклов в динамическом компиляторе запросов* для СУБД PostgreSQL. В конструкции предлагаемого динамического компилятора запросов генерация кода происходит в модели явных циклов путём обхода дерева плана запроса в прямом порядке, во время которого для каждого оператора вызываются функции-генераторы, выполняющие генерацию во внутреннем представлении, декомпозированого на интерфейс в модели явных циклов, алгоритма оператора. Для каждого оператора соответствующие функции-генераторы интерфейса реализованы с использованием LLVM C API и вызываются для генерации реализующей его алгебраической модели кода на LLVM IR, в котором функция *llvm.consume()* родительского оператора вызывается для каждого результирующего кортежа, а *llvm.finalize()* – после формирования последнего результирующего кортежа. После обхода дерева плана сгенерированный код после оптимизаций будет состоять из нескольких циклов, самым первым из которых — цикл одного из операторов сканирования.

В разделах 3.3.2—3.3.5 описывается *динамическая компиляция операторов* сканирования, соединения, сортировки и агрегации в СУБД PostgreSQL. Описывается реализация с использованием LLVM IR интерфейса модели явных циклов для этих операторов.

В разделе 3.4 описывается *реализованный метод динамический компиляции выражений* в СУБД PostgreSQL. Для вычисления выражений в СУБД PostgreSQL используется интерпретатор, который обходит дерево выражений, где узлы операторов содержат указатель на функцию, которая при вызове рекурсивно вычисляет подвыражения, а также сам оператор. Поскольку во время выполнения доступна информация о вызываемых функциях и операциях, можно использовать кодогенерации для замены неявных вызовов функций на явные, которые в дальнейшем могут быть встроены. Динамическая компиляция выражений выполнена путём рекурсивного обхода дерева выражений в обратном порядке и генерации, с использованием LLVM C API, соответствующего кода для каждой функции или операции во внутреннем представлении LLVM IR. По итогу код

для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без расходов на неявный вызов функций.

В разделе 3.4.1 описывается *реализованный метод предкомпиляции встроенных функций* СУБД PostgreSQL, позволяющий избавиться от ручной реализации на LLVM C API функций-генераторов встроенных функции СУБД, используемых для вычисления выражений в запросах. Метод работает следующим образом: множество файлов исходного кода СУБД транслируются в объектные файлы формата биткода LLVM (*.bc*), а затем компонуются в единый биткод-файл. Последний оптимизируется оптимизатором LLVM. Во время старта сервера СУБД PostgreSQL, биткод-файл загружается в виде LLVM-модуля, содержащего все встроенные функции PostgreSQL. В процессе генерации кода выражения выполняется вызов сгенерированных встроенных функций из этого модуля. После генерации кода запроса выполняется оптимизация по встраиванию функций.

В разделе 3.4.2 описываются *оптимизации доступа к атрибутам кортежа*, реализованные в динамическом компиляторе запросов, путём предварительного расчёта смещения атрибутов внутри кортежа, что позволило уменьшить количество ветвлений и обрабатывать только необходимые для данного запроса атрибуты.

В разделе 3.5 описывается *реализация эвристик стратегии выполнения* в динамическом компиляторе запросов для СУБД PostgreSQL. Автомат состояний расширен состояниями для дополнительной оценки времени, затраченного на генерацию кода и снижения производительности, связанного с исполнением кэшированного кода запроса.

В разделе 3.5.1 описывается *реализация механизма кэширования* динамически скомпилированного кода запросов, позволяющая нивелировать накладные расходы при динамической компиляции и оптимизации. В качестве сохраняемого кода был выбран объектный кода запроса. Простого сохранения сгенерированного кода запроса в общем случае недостаточно, так как в этом коде используются переменные и структуры PostgreSQL, абсолютные адреса которых меняются после каждого выполнения плана запроса. Для решения этой проблемы в код сохранённого запроса вставляется дополнительная косвенность: в местах использования этих адресов вставляются загрузки соответствующих значений из глобального массива, который содержит адреса переменных и структур PostgreSQL, использующихся в сгенерированном коде. Во время первого выполнения запроса, наряду с сохранением объектного кода, сохраняется адрес глобального массива адресов. После второго и последующих выполнений того же запроса выполняется загрузка сохранённого объектного кода и адреса глобального массива адресов, выполняется обновление значений адресов в этом массиве. Использование такой косвенности незначительно влияет на производительность, но при этом кэширование кода запроса избавляет от накладных расходов, связанных с этапами генерации и компиляции.

В **четвёртой главе** приведены результаты тестирования реализованного динамического компилятора запросов для СУБД PostgreSQL на запросах из промышленных тестовых наборов TPC-H и TPC-DS, которые считаются достаточно близкими к реальным запросам к СУБД.

Тестирование производительности выполнялось на платформах x86-64 и ARM (aarch64). Размер базы данных определяется с учётом параметра масштаба 1 (примерно 1 ГБ), минимально необходимого размера для тестовой базы данных. В качестве параметра масштаба базы данных для TPC-H были выбраны значения 75, 20 и 10, для TPC-DS — 50 и 20.

В таблицах 1 и 2 отражены результаты тестирования запросов из набора TPC-H на платформах x86-64 и ARM. В колонках `PG`, `JIT` и `JIT+кэш` указано среднее (арифметическое) время выполнения запросов с помощью интерпретатора PostgreSQL, динамического компилятора и динамического компилятора с кэшированием кода соответственно. В колонке `Тип` отмечен режим генерации данных: с использованием оригинальных типов СУБД (NUMERIC, CHAR(1)) и нативных типов данных (DOUBLE, ENUM).

Таблица 1 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64.

TPC-H x86-64							
Масштаб	Тип	PG,сек	JIT,сек	JIT+кэш,сек	$\frac{PG}{JIT}$,%	$\frac{PG}{JIT+\text{кэш}}$,%	$\frac{JIT}{JIT+\text{кэш}}$,%
75	native	108.76	80.14	80.31	35.71	35.43	−0.21
75	original	101.67	77.41	79.43	31.34	28.00	−2.54
20	native	18.71	12.63	11.99	48.14	56.05	5.34
20	original	20.62	16.84	15.56	22.45	32.52	8.23
10	native	7.63	5.72	4.75	33.39	60.63	20.42
10	original	11.11	9.10	7.97	22.09	39.40	14.18

Таблица 2 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе ARM.

TPC-H ARM							
Масштаб	Тип	PG,сек	JIT,сек	JIT+кэш,сек	$\frac{PG}{JIT}$,%	$\frac{PG}{JIT+\text{кэш}}$,%	$\frac{JIT}{JIT+\text{кэш}}$,%
75	native	115.39	74.79	76.75	54.29	50.35	−2.55
75	original	140.67	94.15	93.31	49.41	50.76	0.90
20	native	20.03	13.49	11.54	48.48	73.57	16.90
20	original	28.65	20.61	18.89	39.01	51.67	9.11
10	native	11.58	8.31	7.15	39.35	61.96	16.22
10	original	15.23	12.09	9.85	25.97	54.62	22.74

Ускорение отражает на сколько процентов ускорилось время выполнения динамически скомпилированного запроса относительно интерпретации (колонки $\frac{PG}{JIT}$ и $\frac{PG}{JIT+\text{кэш}}$). Надо отметить, что время динамической компиляции без кэширования кода запроса включает накладные расходы на оптимизацию и компиляцию в машинный код, которые на запросах из

TPC-H составляют от 0.4 до 2.2 сек. в зависимости от сложности запроса. Среднее ускорение от динамической компиляции без кэширования кода выше на большом объёме данных. Использование метода кэширования кода позволяет компенсировать ухудшение производительности на небольшом объёме данных за счёт экономии на времени компиляции. При тестировании с использованием нативных типов данных ускорение от динамической компиляции лучше чем при использовании оригинальных типов СУБД. Это обусловлено тем, что во время динамической компиляции используются соответствующие встроенные типы LLVM, что позволяет лучше оптимизировать код. Различие результатов на платформах x86-64 и ARM объясняется различием в скорости работы памяти и операций доступа к данным на соответствующих аппаратных платформах.

В таблице 3 приведены результаты выполнения всех запросов из набора TPC-H на платформе x86-64 при масштабе 75 с использованием нативных типов данных. Ускорение вычислялось как отношение времени выполнения интерпретатора ко времени выполнения динамически скомпилированного запроса и показывает во сколько раз ускорился запрос (колонки $\frac{PG}{JIT+\text{кэш}}$ и $\frac{PG}{JIT}$).

Таблица 3 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора на запросах из TPC-H на платформе x86-64.

TPC-H | 75 | Native | x86-64					
№ запроса	PG,сек	JIT,сек	JIT+кэш,сек	$\frac{PG}{JIT+\text{кэш}}$, раз	$\frac{PG}{JIT}$, раз
Q1	278.86	54.62	60.29	**4.63**	**5.11**
Q2	21.07	20.57	19.67	1.07	1.02
Q3	120.42	66.31	63.01	**1.91**	**1.82**
Q4	41.87	36.74	36.36	1.15	1.14
Q5	407.89	392.43	386.62	1.06	1.04
Q6	71.41	67.77	63.53	1.12	1.05
Q7	185.96	165.24	157.03	1.18	1.13
Q8	43.56	40.37	42.24	1.03	1.08
Q9	205.41	167.65	170.45	1.21	1.23
Q10	95.96	53.65	58.80	**1.63**	**1.79**
Q11	5.65	5.48	4.18	1.35	1.03
Q12	116.38	100.84	100.78	1.15	1.15
Q13	130.57	93.21	95.06	1.37	1.40
Q14	27.95	24.41	23.46	1.19	1.15
Q15	97.91	80.33	83.93	1.17	1.22
Q16	29.67	24.02	23.94	1.24	1.24
Q17	7.26	5.37	4.81	**1.51**	1.35
Q18	100.27	45.53	47.11	**2.13**	**2.20**
Q19	5.02	4.04	2.77	**1.81**	1.24
Q20	86.09	84.07	81.20	1.06	1.02
Q21	302.96	221.48	231.99	1.31	1.37
Q22	10.47	9.00	9.63	1.09	1.16
AVG	108.76	80.14	80.31	1.35	1.36

Как следует из результатов, на отдельных запросах из TPC-H ускорение от применения динамического компилятора достигает **до 5 раз**, в среднем ускорение составляет 1.36 раз по сравнению с интерпретатором СУБД PostgreSQL. Также было проведено тестирование с включённым режимом *параллельного выполнения запросов* в СУБД PostgreSQL на тестовом наборе TPC-H в масштабе 75 на платформе ARM, которое показало ускорение от динамической компиляции до **34 раз** на 8 параллельных рабочих процессах (ядрах процессора) по сравнению с интерпретатором PostgreSQL без параллелизма и до **4.5 раза** по сравнению с PostgreSQL с параллелизмом. На запросах из тестового набора TPC-DS среднее ускорение составляет от 1.31 до 1.55 раз в зависимости от масштаба базы.

Можно отметить, что короткие запросы выигрывают от кэширования скомпилированного кода за счёт экономии на времени компиляции. Однако на долгих запросах дополнительный уровень косвенности, используемый при кэшировании запросов, может снизить производительность до 10%, как, например, на запросе $Q1$ из таблицы 3, в среднем ухудшение производительности составляет 5%. Разработанные эвристики устанавливают баланс между стратегиями «всегда кэшировать» и «всегда компилировать»: динамическая компиляция не применяется для очень простых запросов, кэширование используется для средних запросов, а долго выполняющиеся запросы всегда компилируются.

Полученные результаты тестирования демонстрируют положительное влияние метода динамической компиляции на время выполнения запроса, что доказывает перспективность этого подхода в контексте СУБД.

В **заключении** приведены основные результаты работы, которые заключаются в следующем:

1. Разработан метод динамической компиляции запросов с трансформацией на лету операторов плана запроса из модели Volcano в модель явных циклов.
2. Разработан метод динамической компиляции выражений в SQL-запросах с применением открытой вставки предварительно скомпилированных встроенных функций СУБД.
3. Разработаны эвристики стратегии выполнения запроса на основе оценок затрат на динамическую компиляцию. Также разработан метод кэширования кода, сгенерированного динамическим компилятором, в SQL-запросах.
4. На основе предложенных методов реализован динамический компилятор SQL-запросов в качестве программного расширения к СУБД с открытым исходным кодом PostgreSQL с использованием компиляторной инфраструктуры LLVM. Проведена апробация реализованных методов на промышленных тестовых наборах TPC-H и TPC-DS, показывающая перспективность этого подхода в контексте СУБД.

Публикации автора по теме диссертации

1. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL [Текст] / Е. Ю. Шарыгин, Р. А. Бучацкий [и др.] // Труды Института системного программирования РАН. — 2016. — Т. 28, № 4. — С. 217—240. — (ВАК).
2. Динамическая компиляция SQL-запросов для СУБД PostgreSQL [Текст] / Р. А. Бучацкий, Е. Ю. Шарыгин [и др.] // Труды Института системного программирования РАН. — 2016. — Т. 28, № 6. — С. 37—48. — (ВАК).
3. *Шарыгин, Е. Ю.* Обзор методов динамической компиляции запросов [Текст] / Е. Ю. Шарыгин, Р. А. Бучацкий // Труды Института системного программирования РАН. — 2017. — Т. 29, № 3. — С. 179—224. — (ВАК).
4. Кэширование машинного кода в динамическом компиляторе SQL-запросов для СУБД PostgreSQL [Текст] / М. В. Пантилимонов, Р. А. Бучацкий [и др.] // Труды Института системного программирования РАН. — 2020. — Т. 32, № 1. — С. 205—220. — (ВАК).
5. Query compilation in PostgreSQL by specialization of the DBMS source code [Текст] / E. Sharygin, R. Buchatskiy, [et al.] // Programming and Computer Software. — 2017. — Vol. 43. — P. 353—365. — (Scopus), (WoS).
6. Runtime Specialization of PostgreSQL Query Executor [Текст] / E. Sharygin, R. Buchatskiy, [et al.] // Perspectives of System Informatics / ed. by A. K. Petrenko, A. Voronkov. — Cham : Springer International Publishing, 2018. — P. 375—386. — (Scopus).
7. Machine Code Caching in PostgreSQL Query JIT-Compiler [Текст] / M. Pantilimonov, R. Buchatskiy, [et al.] // 2019 Ivannikov Memorial Workshop (IVMEM). — 2019. — P. 18—25. — (Scopus).
8. *Свидетельство о государственной регистрации программы для ЭВМ.* Реализация метода динамической компиляции выражений в SQL-запросах для СУБД PostgreSQL [Текст] / Р. А. Бучацкий [и др.] ; ФГБУН Институт системного программирования РАН. — № 2017616982 ; заявл. 17.07.2017 (Рос. Федерация).

Бучацкий Рубен Артурович

Метод динамической компиляции SQL-запросов для реляционных СУБД

Автореф. дис. на соискание ученой степени канд. тех. наук

Подписано в печать ___.___._____. Заказ № ____________
Формат 60×90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография ______________________________________