Федеральное государственное бюджетное учреждение науки
Институт системного программирования имени В. П. Иванникова
Российской академии наук

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»

Диссертация на соискание учёной степени
кандидата технических наук

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

Москва — 2022

Оглавление

Стр.

Стр.

Введение

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.
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.

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

– 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».

Объем и структура работы. Диссертация состоит из введения, 4 глав, заключения и 2 приложений. Полный объём диссертации составляет 168 страниц, включая 38 рисунков и 16 таблиц. Список литературы содержит 80 наименований.

Глава 1. Обзор предметной области

Одной из основных функций современных СУБД [9] является предоставление интерфейса для выполнения операций по определению, изменению или выборке данных. Чаще всего такой интерфейс реализуется посредством специализированного языка запросов, например SQL, что позволяет максимально изолировать задачи, решаемые СУБД, от других частей информационной системы.

Основные этапы обработки запроса в СУБД показаны на рисунке 1.1. Хотя детали реализации могут различаться между разными СУБД, например, некоторые пропускают или объединяют шаги, процесс обработки запросов в большинстве СУБД состоит из следующих этапов:

1. Лексический и синтаксический анализ. На этом этапе входная строка-запрос пользователя обрабатывается лексическим и синтаксическим анализаторами и в результате получается дерево разбора. В процессе анализа выполняется только проверка синтаксиса, но не проверяется семантика. Например, если в запросе осуществляется обращение к таблице, которая не существует в базе данных, то ошибка не будет выдана.
2. Семантический анализ. Дерево разбора, полученное с предыдущей фазы, проходит через семантический анализ, в результате которого получается дерево запроса, дополненное различного рода метаинформацией: системными идентификаторами таблиц, типами и порядковыми номерами запрашиваемых полей, перечнем соединяемых таблиц и условий фильтраций в виде дерева и т.д.
3. Обработка системой правил. Выполняется поиск в системных каталогах применимых к дереву запроса правил и, обнаружив подходящие, выполняются преобразования, описанные в теле найденного правила. Примером преобразования является замена обращений к представлениям, так называемым виртуальным таблицам, на обращения к базовым таблицам из определения представления.
4. Планирование и оптимизации. Планировщик получает на вход структуру с деревом запроса. Используя вспомогательные структуры данных, называемыми путями, которые представляют собой упрощённые схемы планов, осуществляется выбор наиболее эффективного пути выполне-

ния запроса с точки зрения имеющихся оценок затрат и статистической информации на момент выполнения. Осуществляется выбор оптимального метода доступа к данным с заданным порядком соединений и алгоритмов для их выполнения. Выбранный вариант трансформируется в полноценный план запроса и передаётся исполнителю.

5. Выполнение итогового плана запроса. Исполнитель осуществляет рекурсивный обход по дереву плана и выполняет инкапсулированную логику соответствующего узла-оператора или выражения, формулируя на выходе ответ на запрос в виде множества строк.

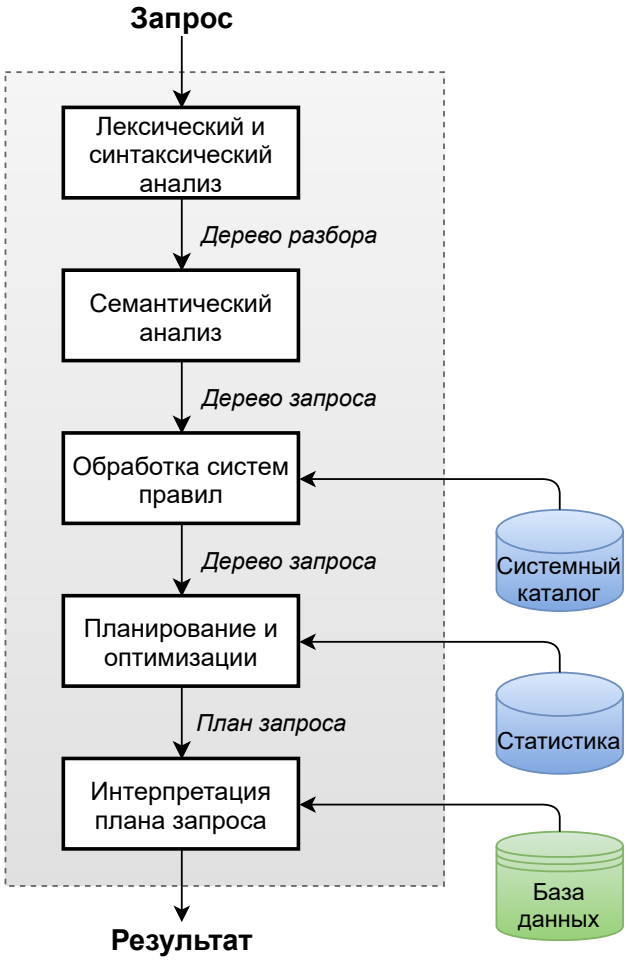


Рисунок 1.1 — Процесс обработки запроса в СУБД.

Большинство СУБД используют описанный подход для трансляции запроса пользователя сначала в логический план запроса, представляющий собой дерево из операторов расширенной реляционной алгебры, а затем в физический. Последний получается путём добавления метаинформации о выбранных методах доступа к отношениям в листовых узлах и о применяемых алгоритмах, реализующих реляционные операции во внутренних узлах. Примерами физических операторов являются сортировка, последовательное или индексное

сканирование, соединение хешированием, соединение методом вложенных циклов и т.д. Готовый алгебраический план выполняется исполнителем. На рисунке 1.2 представлен пример SQL-запроса, соответствующего логического плана и один из вариантов физического плана, где дуги в дереве представляют потоки данных между операторами. Логический план этого запроса состоит из оператора проекции π, который на заданное подмножество множества атрибутов отношения-операнда производит отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда, оператора соединения $\bowtie$, который по некоторому условию образует результирующее отношение, кортежи которого производятся путём объединения кортежей первого и второго отношений и оператора выборки σ, который выполняет фильтрацию данных из одной или нескольких таблиц, удовлетворяющих предикату.

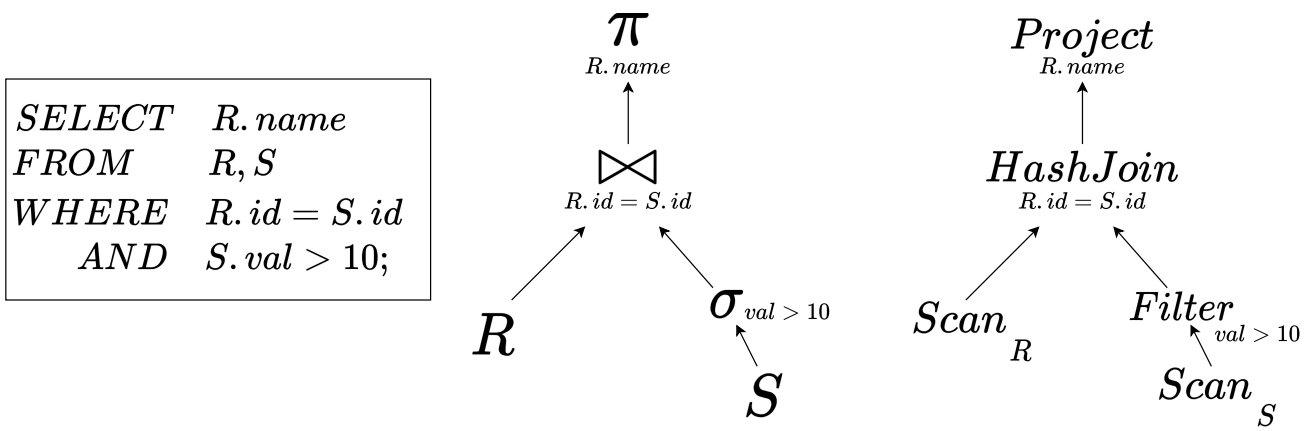


Рисунок 1.2 — Слева: пример SQL-запроса, по центру: логический план запроса, справа: физический план запроса.

Можно заметить, что только затраты на выполнение этапа 5 (выполнения плана запроса) зависят не только от сложности самого запроса, но и от размера данных, обрабатываемых СУБД, и поэтому доминируют над затратами на выполнение этапов 1–4 с увеличением размера данных.

Системы, работающие по приведённой выше схеме обработки запросов, в рамках данной работы будем называть *классическими*. Недостатки классической схемы проявляются в следующих сценариях использования:

– Если информационная система отправляет много запросов, большинство из которых принадлежит одному из сравнительно небольшого числа различных классов, то этапы 1–4 в классической схеме для каждого класса запросов выполняются многократно. Современные СУБД решают эту проблему при помощи поддержки параметризованных

запросов (prepared statements). Параметризованный запрос анализируется, транслируется в план выполнения и оптимизируется только один раз во время определения, и во время выполнения оптимизированный план переиспользуется для всех экземпляров запроса.

– Если большинство запросов, отправляемых информационной системой, принадлежит ограниченному числу запросов, известных во время разработки системы, то в классической схеме все перечисленные этапы выполняются многократно, несмотря на то, что как план выполнения, так и параметры запроса (если они есть) остаются на протяжении работы системы неизменными. Параметризация запросов позволяет избежать выполнения этапов 1–4, но не этапа 5 (собственно выполнения), алгоритмическая сложность которого доминирует над остальными этапами. Накладных расходов на интерпретацию можно избежать в этом случае при помощи (статической) компиляции запросов в машинный код [10—12] при сборке информационной системы. Статическая компиляция в дальнейшем не рассматривается.
– Если запросы, отправляемые к СУБД, достаточно сложны и выполняются достаточно долго, накладные расходы на интерпретацию плана в классической схеме (этап 5) могут составлять значительную часть в общей времени обработки запроса. Решением этой проблемы является динамическая компиляция, обзору методов реализации которой и посвящены следующие разделы.

1.1 Модели выполнения запросов

Модель выполнения запросов в СУБД определяет, как система выполняет план запроса. Существует четыре основных модели:

1. Модель итераторов [13; 14] (Volcano или pull, кортеж за раз)
2. Материализующая модель [15] (оператор за раз, колонка за раз)
3. Векторизующая модель [16] (векторная, блочная)
4. Модель явных циклов [17; 18] (push, кортеж за раз)

У каждой модели есть свои достоинства и недостатки в зависимости от типа рабочей нагрузки и используемого оборудования. В следующих подразделах мы подробно рассмотрим особенности каждой модели.

1.1.1 Модель итераторов

Традиционным способом выполнения запросов, используемым в интерпретаторах классических систем обработки запросов, является модель итераторов [19], также известная как Volcano-модель [13; 14].

Стандартный интерфейс оператора в модели Volcano состоит из трёх методов:

$Operator$

$open() \ : \ void$

$next() \ : \ Tuple*$

$close() \ : \ void$

Метод *open* инициализирует внутреннее состояние оператора, выделяет необходимые ресурсы и подготавливает оператор к созданию первого кортежа. Метод *next* создаёт и возвращает один следующий кортеж ($Tuple$), если он есть, и возвращает $NULL$, когда были созданы все кортежи. Метод *close* сбрасывает внутреннее 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.

Рассмотрим подробнее процесс выполнения запроса в модели Volcano на примере физического плана из рисунка 1.2. Подготовка каждого оператора перед началом выполнения запроса осуществляется путём вызова функции $open()$ у корневого оператора, который соответствующим образом инициализирует своё состояние и вызывает аналогичную функции у дочерних узлов. Процесс освобождения ресурсов выполняется аналогичным способом при помощи вызова функции $close()$. На рисунке 1.3 представлены этапы инициализации и освобождения ресурсов для примерного запроса.

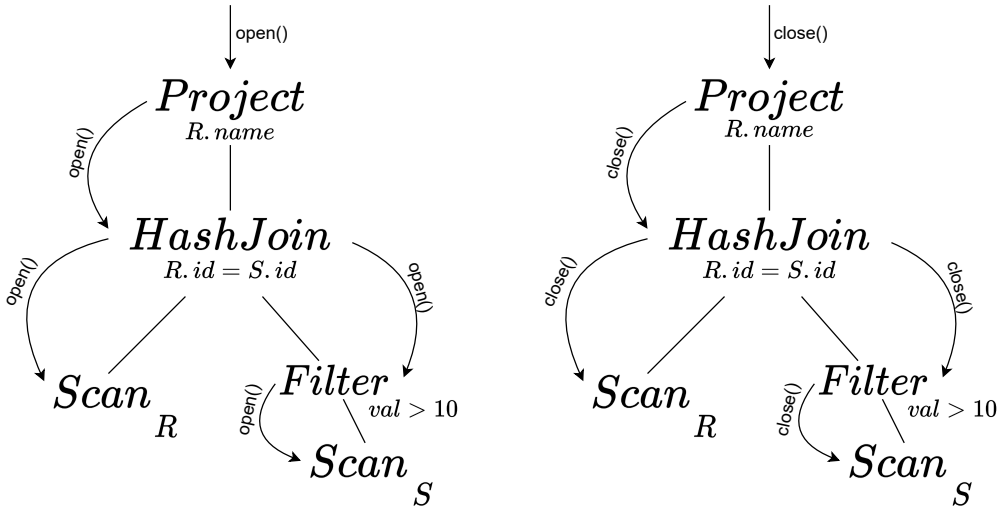


Рисунок 1.3 — Пример инициализации и освобождения операторов в физическом плане для запроса с рисунка 1.2.

После завершения стадии инициализации начинается этап выполнения запроса. Для выполнения запроса необходимо вызывать функцию $next()$ у корневого оператора для получения следующего кортежа до тех пор, пока не будет получен нулевой указатель вместо нового кортежа. В свою очередь, корневой оператор вызывает реализацию функции $next()$ у своих потомков, те у своих, и так вплоть до листовых узлов, которые получают входные данные из файлов таблиц и индексов, возвращая назад кортежи для последующей обработки. Визуально рекурсивный вызов функции $next()$ представлен на рисунке 1.4.

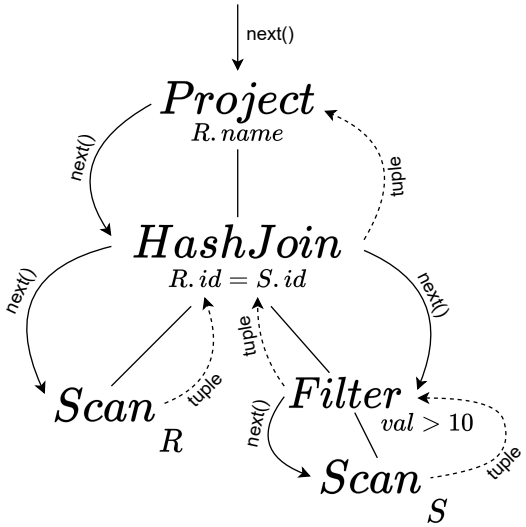


Рисунок 1.4 — Пример интерпретации в модели итераторов физического плана запроса с рисунка 1.2.

На рисунке 1.5 представлен псевдокод функций $next()$ для операторов SQL-запроса из рисунка 1.2, где пунктирными линиями обозначено направление

потока данных. Выполнение начинается с корневого узла, который вызывает функцию $next()$ от своего дочернего узла, оператора соединения, который в свою очередь вызывает реализацию функции $next()$ у своих потомков: для левого узла — это метод доступа, который итерируется по отношению R и генерируют кортежи, которые затем обрабатываются, для правого узла — это оператор фильтрации, который вызывает метод доступа для итерации по отношению S. После обработки всех кортежей возвращается нулевой указатель для оповещения родительских узлов о завершении выполнения.

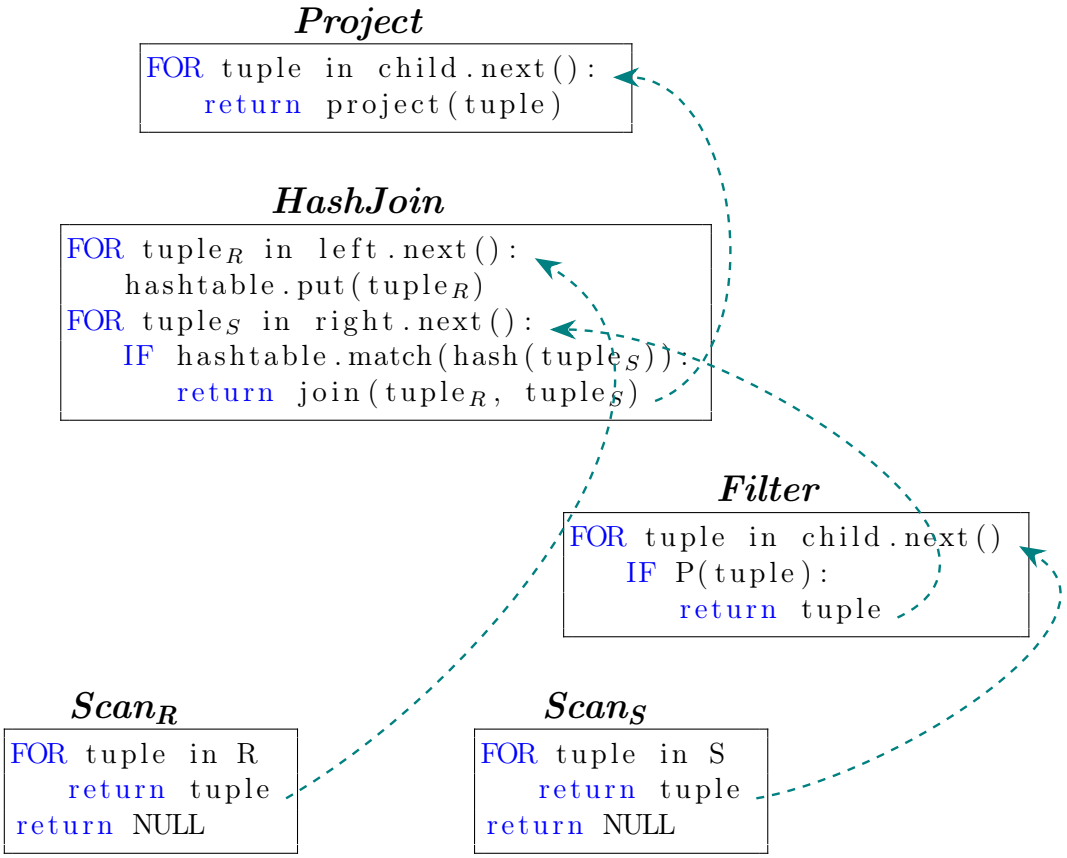


Рисунок 1.5 — Псевдокод операторов из SQL-запроса на рисунке 1.2 в модели Volcano.

Модель Volcano упрощает разработку операторов за счёт применения принципа декомпозиции, позволяет распределять операторы по различным вычислительным узлам и естественным образом объединяет вычисления последовательных операторов в запросе, поскольку вычисление каждого конкретного члена результирующей последовательности откладывается до того момента, как значения его атрибутов потребуются для вычисления очередного кортежа в родительском операторе, тем самым предоставляя механизм для построения конвейера по обработке данных.

Модель Volcano используется в большинстве современных СУБД, таких как PostgreSQL, MySQL, SQLite, Oracle, DB2 и других. Ее популярность объясняется простотой в реализации и универсальностью: любой оператор реляционной алгебры легко описать с помощью интерфейса *open*, *next* и *close*. При этом модель Volcano была разработана в то время, когда операции доступа к диску занимали практически всё время исполнения запроса. По мере того как характеристики аппаратных средств улучшались, неэффективное использование процессора в модели Volcano становилось всё заметнее.

Можно отметить следующие основные причины неэффективного использования процессора при выполнении запроса в модели Volcano:

- Невозможность применения оптимизации встраивания. Вызов $next()$ рекурсивен, и его встраивание (inlining) нецелесообразно: оно чревато «раздуванием» кода, а встраивание всего кода для произвольного запроса попросту невозможно. Причиной является то, что вызовы $next()$ косвенные, т.е. компилятор не знает, функция какого оператора будет фактически вызвана.
- Плохое предсказание переходов. Функция $next()$ является виртуальной и часто реализуется как косвенный вызов (по указателю), что плохо сочетается с конвейером современных ЦП. Даже в случае реализации с использованием условного оператора *switch*, где в зависимости от типа оператора выполняется вызов соответствующей функции, реализующий данный оператор, предсказывание переходов все равно затруднено, поскольку дерево операторов может быть произвольным.
- Плохая локальность по коду. За один вызов $next()$ оператор обрабатывает только один кортеж. Если в ходе своей работы запрос обрабатывает миллионы кортежей, то функция $next()$ будет вызвана аналогичное количество раз — это приводит к неэффективному использованию кэша инструкций процессора.
- Плохая локальность по данным. Подход «кортеж за раз» предполагает частое восстановление и сохранение состояния оператора между вызовами функции $next()$, то есть загрузку из динамической памяти в стек и регистры необходимых для работы оператора данных, а по завершении $next()$ сохранение их обратно в динамическую память, чтобы продолжить работу при следующем вызове. Это ведёт к неэффективному использованию кэша данных процессора по причинам, аналогичным

описанным в предыдущем пункте. Более того, сами инструкции копирования из динамической памяти на стек/в регистры и обратно занимают существенное время.

Существует несколько способов решения перечисленных проблем, отличающиеся своей эффективностью.

1.1.2 Материализующая модель

В рамках данной модели [15] 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. Для избежания чрезмерного сканирования кортежей в узлах разного уровня, оптимизатор запросов СУБД может передавать последующим операторам информацию о том, сколько кортежей необходимо. Оператор «материализует» свой вывод в виде единственного результата. Результатом может быть либо целый кортеж (при построчном хранении данных), либо подмножество столбцов (при колоночном хранении данных).

На рисунке 1.6 представлен псевдокод функций $next()$ в материализующей модели для операторов SQL-запроса из рисунка 1.2. Здесь, как и в модели итераторов, выполнение начинается с корневого узла вызовом функции *child.out()*. Далее нижестоящий оператор соединения вызывает функцию *child.out()* у своих потомков — операторов сканирования, которые возвращают все кортежи обратно.

Наибольшее распространение эта модель получила в СУБД, ориентированных на OLTP-нагрузку, где преобладающая часть запросов оперирует небольшим количеством кортежей. Для OLAP-нагрузки данная модель подходит не так хорошо, 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.

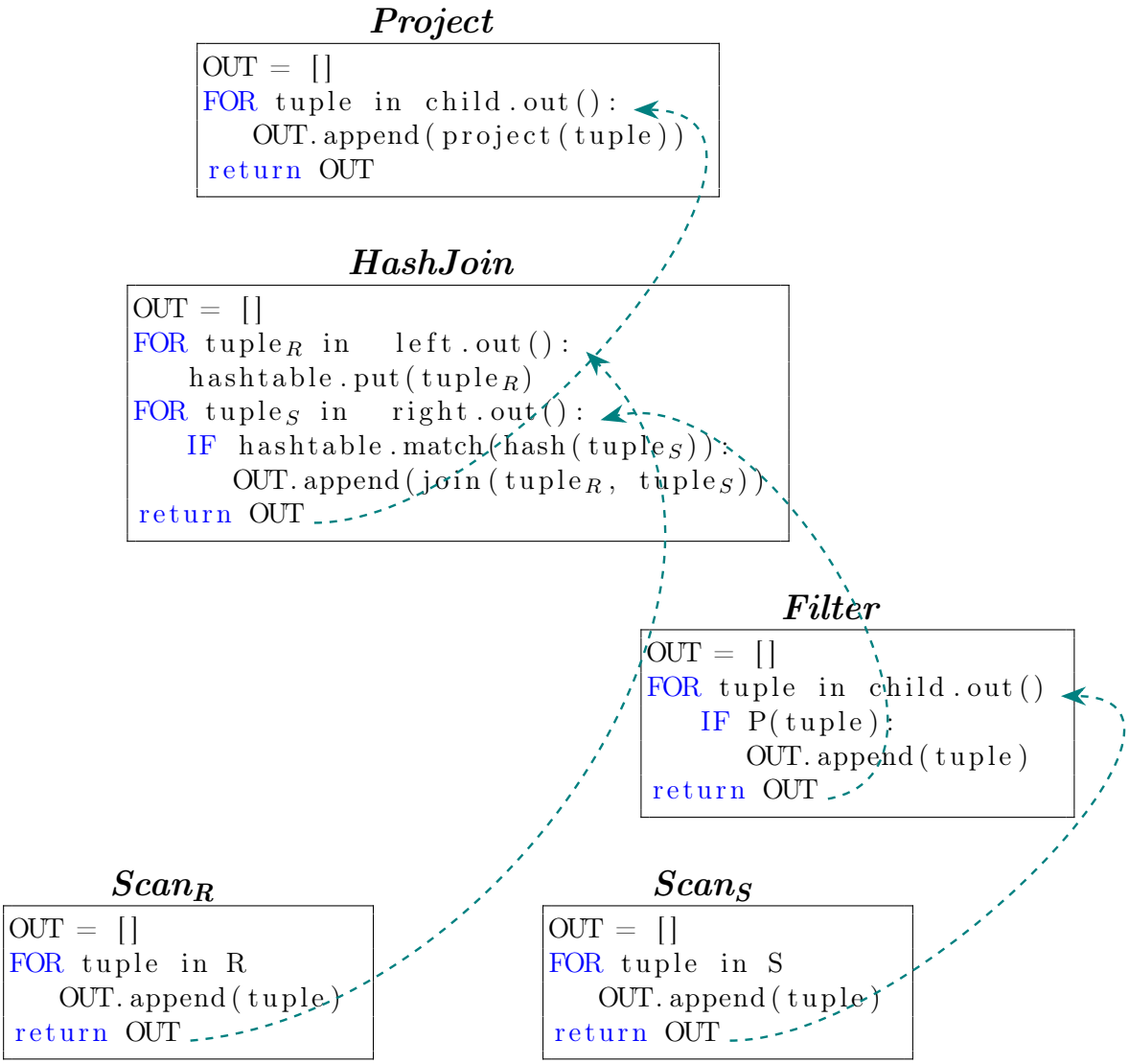


Рисунок 1.6 — Псевдокод операторов из SQL-запроса на рисунке 1.2 в материализующей модели.

Ярким представителем материализующей модели является резидентная СУБД VoltDB [20], ориентированная на обработку миллионов транзакций в секунду.

1.1.3 Векторизующая модель

Как и в модели Volcano, в векторизующей модели [16] 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 пе-

редавать целую партию (или вектор) кортежей за раз, варьируя её размер в зависимости от характеристик аппаратного обеспечения, доступных ресурсов и рекомендаций оптимизатора запросов.

На рисунке 1.7 представлен псевдокод функций $next()$ в векторизующей модели для операторов SQL-запроса из рисунка 1.2. Основное различие от модели Volcano в том, что для каждого оператора выходной буфер сравнивается с заданным размером вектора. Как только буфер наполняется, то пакет кортежей отправляется выше. При такой обработке стоимость вызовов $next()$ амортизируется на размер вектора и накладные расходы на них значительно уменьшаются.

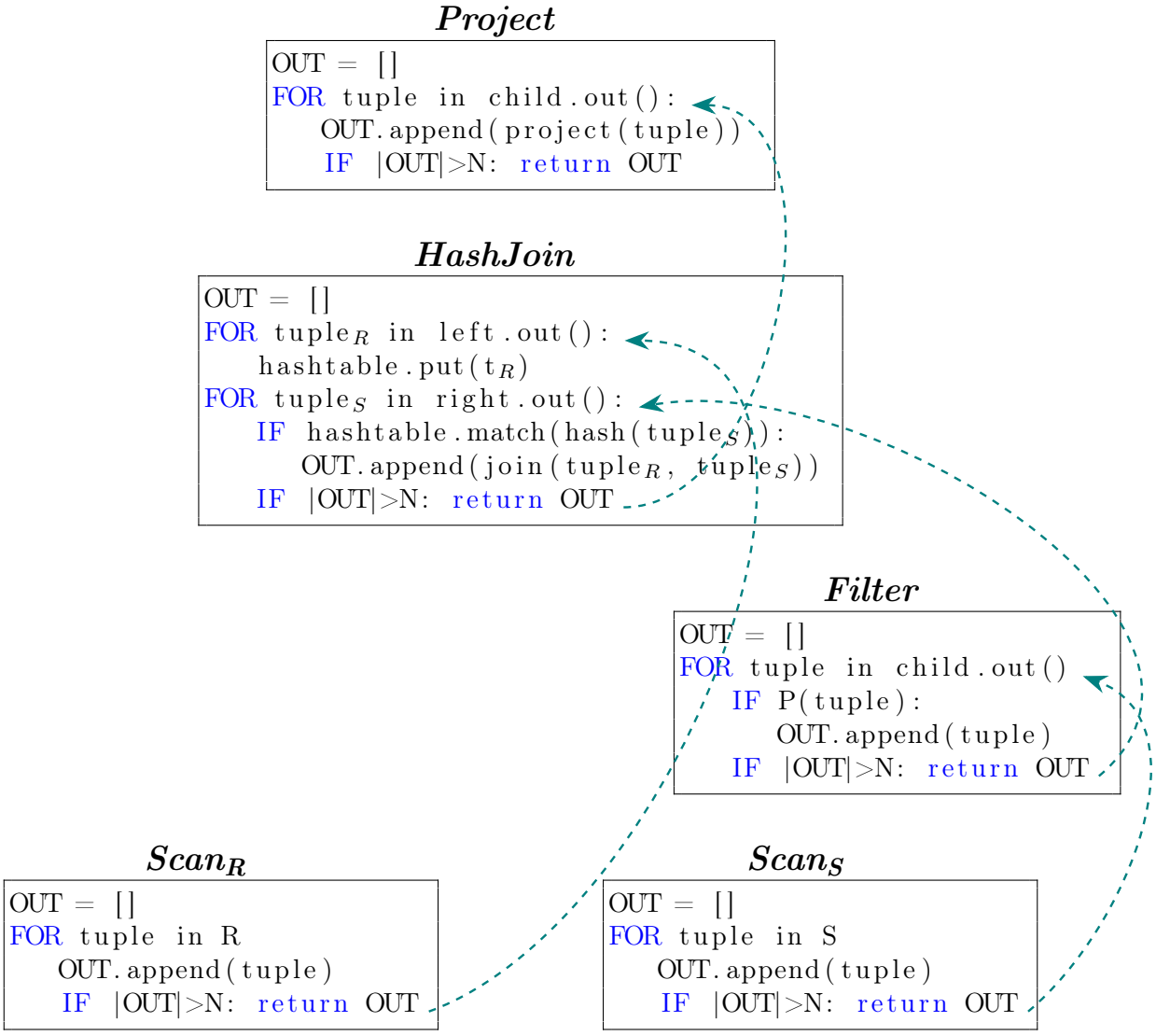


Рисунок 1.7 — Псевдокод операторов из SQL-запроса на рисунке 1.2 в векторизующей модели.

Модель в первую очередь ориентирована на OLAP-нагрузку, так как значительным образом уменьшает количество вызовов функции $next()$, аккуратно расходуя доступную память, а также позволяет повысить скорость обработки

за счёт применения аппаратных векторных инструкций (SIMD). Зачастую эта модель сочетается с колоночной организацией данных для максимальной утилизации векторных расширений, однако может использоваться и в традиционных строчно-ориентированных решениях. Данная модель применяется во многих современных СУБД с разным способом организации данных: SQL Server, Oracle, DB2, Presto, ActianVector и т.д.

1.1.4 Модель явных циклов

Материализующая и векторизующая модели выполнения запросов пытаются преодолеть проблему модели Volcano с количеством вызовов функции $next()$ (или её аналога) примерно одинаковым способом — использовать больше оперативной 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 [17; 18] *data-centric* или *push*. Основное её отличие от модели итераторов заключается в направлении обработки данных: дерево операторов обходится не сверху вниз, а снизу вверх. Таким образом, кортежи передаются в вышестоящий оператор без «запроса» сверху в виде функции $next()$, а как бы «проталкиваются» снизу вверх. Процесс инициализации и освобождения ресурсов не меняется, то есть 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. Схематично этот процесс для примерного запроса изображён на рисунке 1.8.

На этом запросе в модели явных циклов выполнение начинается снизу дерева, с оператора последовательного сканирования по отношению R: каждый кортеж из неё будет передан оператору HashJoin, который положит их в хеш-таблицу. Затем выполнение продолжится в операторе последовательного

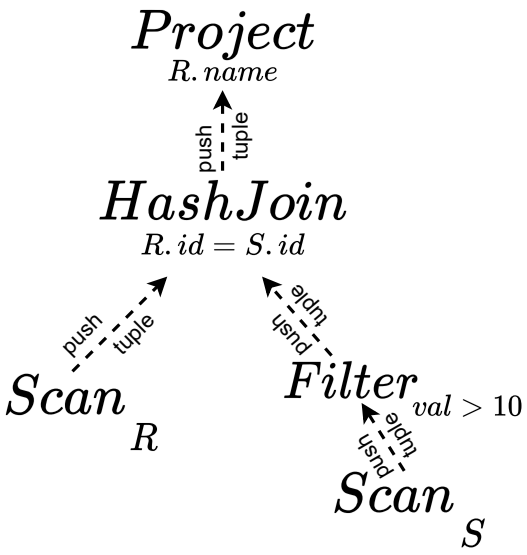


Рисунок 1.8 — Пример использования модели явных циклов на физическом плане для запроса с рисунка 1.2.

сканирования по отношению S, кортежи из неё будут отправлены в Filter для проверки условия, а затем в оператор HashJoin, который попытается найти каждому кортежу соответствующие пары и отправит их выше в Project для вывода результата запроса. Если в модели Volcano кортежи «вытягивались» из операторов-потомков, то здесь они «вталкиваются» в родительские операторы, именно поэтому модель явных циклов называют *push*-моделью. На рисунке 1.1 представлен псевдокод реализации модели явных циклов для запроса на рисунке 1.2.

Листинг 1.1: Псевдокод SQL-запроса на рисунке 1.2 в модели явных циклов.

```
FOR tuple_R in R:
    hashtable.put(tuple_R)
FOR tuple_S in S:
    IF P(tuple_S):
        IF hashtable.match(hash(tuple_S)):
            project(join(tuple_R, tuple_S))
```

С точки зрения эффективности, подобный «разворот» направления потока обработки данных решает только одну из перечисленных ранее проблем — проблему локальности по данным, то есть с частым сохранением и восстановлением состояния оператора, но не избавляет от основных накладных расходов, присущих модели Volcano. Однако модель явных циклов используется в динамических компиляторах запросов, что даёт возможность генерировать для конкретного запроса эффективный машинный код путём встраивания функций, удаления «мёртвого» с точки зрения запроса кода, подстановки констант и других оптимизаций.

1.2 Динамическая компиляция в СУБД

Вне зависимости от используемой модели выполнения, классический способ выполнения запроса сопряжён с накладными расходами по вызову виртуальных функций в ходе интерпретации его плана, состоящего из произвольной последовательности операторов, выражений и предикатов, что, в свою очередь, порождает значительное количество ложных предсказаний переходов. Также в ходе интерпретации могут выполняться проверки, которые избыточны для конкретного плана запроса. Всё чаще для решения этой проблемы применяется метод динамической компиляции, который заключается в кодогенерации специализированного кода под заданный план запроса. Производительность достигается за счёт применения оптимизаций встраивания функций, подстановки констант, вычисления арифметических выражений, замены косвенных вызовов на явные, удаления «мёртвого», с точки зрения плана запроса, кода и т.д.

Существующие методы динамической компиляции (как и СУБД, их использующие) можно разделить на два класса:

- компиляция выражений и горячих участков кода (1.2.1) — компиляция таких частей запросов, как арифметические и логические выражения и доступ к атрибутам, при сохранении интерпретации плана запроса;
- компиляция запросов (1.2.2) — компиляция запросов целиком, выполнение без интерпретации.

1.2.1 Динамическая компиляция выражений и горячих участков кода

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. Множество рассматриваемых функций-кандидатов включает в себя:

1. Функции, осуществляющие вычисление арифметических выражений. Так как выражения в запросе становятся известны только во вре-

мя выполнения, вычисление выражений, как правило, реализуется в классических СУБД при помощи интерпретации, накладные расходы, связанные с которой, динамическая компиляция позволяет избежать.

2. Пользовательские функции. Многие системы позволяют использовать в запросах функции, определённые пользователем или на процедурных языках, поддержка которых встроена непосредственно в язык запросов, или через API (например, на C или C++). Вызов пользовательской функции во время выполнения запроса сопряжён с накладными расходами на поддержку абстракции: интерфейс, позволяющий пользовательской функции получать значения и типы параметров и устанавливать возвращаемое значение, как правило, реализован обобщённо. Кроме того, тело функции также обычно неизвестно до начала выполнения запроса. Динамическая компиляция позволяет встроить тело пользовательской функции непосредственно в машинный код вызывающей функции и удалить накладные расходы за счёт оптимизации функции в месте вызова.
3. Функции доступа к атрибутам. В СУБД, использующих построчное хранение кортежей (N-ary Storage Model [9; 21]), атрибуты каждого конкретного кортежа располагаются в памяти последовательно. Смещение заданного атрибута в кортеже определяется его порядковым номером и типами предшествующих атрибутов (которые, в свою очередь, в реляционной модели данных определяются заголовком отношения) и потому вычисляется во время выполнения запроса. Динамическая компиляция позволяет предвычислить или существенно оптимизировать (в случаях, когда смещение атрибута также зависит и от значений предшествующих атрибутов [1]) вычисление смещений атрибутов, используя информацию о таблицах, к которым производится обращение.
4. Функции, использующие константные данные времени выполнения. Динамическая компиляция позволяет встроить в код значения не изменяемых во время выполнения глобальных переменных и полей структур, например, параметров работы СУБД или номера выполняемой транзакции (снимка таблицы) при использовании механизма многоверсионности (MVCC) [22]. Можно заметить, что в эту категорию входят также внутренние структуры интерпретатора выражений

и заголовки таблиц, что делает описываемый здесь класс функций обобщением классов функций, описываемых выше.

При компиляции выражений сохраняется общий механизм интерпретации плана запроса, компиляции которого посвящён раздел 1.2.2.

Далее мы проведём обзор систем обработки данных, в которых реализована динамическая компиляция выражений и горячих участков кода.

Apache Impala

Impala [23; 24] — это распределённая аналитическая система обработки данных для Apache Hadoop [25], использующая в качестве хранилища данных распределённую файловую систему Apache HDFS [25] или распределённую нереляционную СУБД Apache HBase [26].

Impala предоставляет интерфейс запросов на языке SQL и использует LLVM [27; 28] для компиляции частей запроса в машинный код во время выполнения. В качестве примера приводится функция доступа к атрибутам кортежа в памяти: динамическая компиляция позволяет специализировать общий код доступа к атрибутам, реализующий поддержку многих типов данных, с использованием доступной во время выполнения информации о том, к каким таблицам в данном запросе производится обращение. Поскольку эта функция вызывается для каждого обрабатываемого в цикле кортежа, даже удаление небольшого числа инструкций приводит к значительному приросту производительности.

Выделяются следующие оптимизаций, которые удалось применить во время динамической компиляции:

- подстановку констант времени выполнения;
- удаление условных переходов;
- удаление операций загрузки из памяти;
- разворачивание циклов;
- встраивание виртуальных вызовов.

В частности, арифметические и логические выражения в Impala представляются в виде иерархии классов с перегруженным методом вычисления значения, и встраивание вызовов виртуальных функций позволяет значительно сократить связанные с этим накладные расходы (см. рис. 1.9).

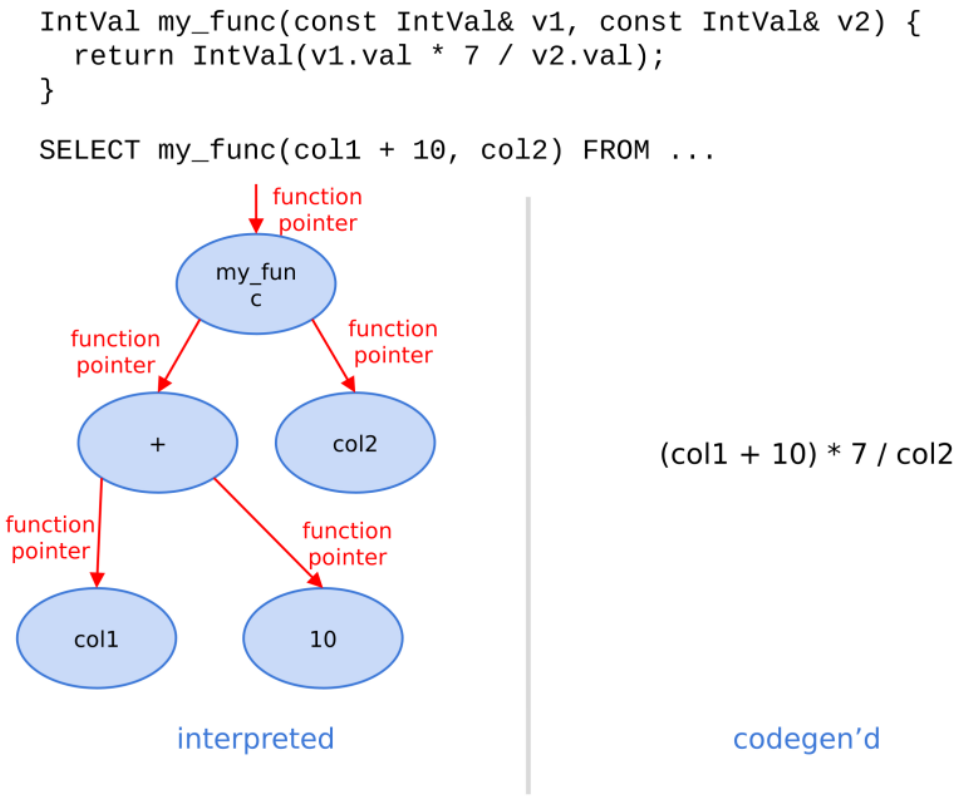


Рисунок 1.9 — Компиляция выражений в Apache Impala.

В [24] также описывается подход к разработке динамического компилятора с использованием инфраструктуры LLVM, при котором оптимизируемые функции реализуются на языке высокого уровня (C++) и компилируются статически в промежуточное представление LLVM IR и в объектный код, что позволяет переиспользовать один и тот же исходный код функций как в компиляторе, в котором производится связывание полученного кода на LLVM IR с генерируемым модулем (единицей трансляции) LLVM во время выполнения, так и в интерпретаторе, с которым полученный код связывается статически. Этот же метод также применяется и для оптимизации пользовательских функций на C++, которые связываются с использующим их кодом при использовании интерпретатора динамически через подгружаемую библиотеку, а при использовании компилятора — статически во время выполнения.

Динамическая компиляция выражений в Impala позволяет получить ускорение до 5.7 раз на запросе Q1 из тестового набора TPC-H [29] (запрос выполнялся на кластере из 10 узлов), при этом количество выполняемых инструкций и ветвлений сокращается в 4.29 и 3.76 раз, соответственно.

Spark SQL

Apache Spark [30] — это фреймворк для реализации распределённой обработки слабоструктурированных и неструктурированных данных для Apache Hadoop. Основными областями применения Spark являются потоковая обработка данных, обработка графовых данных и машинное обучение.

Spark SQL [31] — это модуль Apache Spark, реализующий поддержку реляционной модели обработки данных и декларативного языка запросов SQL.

В Spark SQL применяется динамическая компиляция арифметических и логических выражений. Для генерации промежуточного представления (абстрактного синтаксического дерева Scala) и компиляции его в байткод виртуальной машины Java используются встроенные средства языка Scala.

На простых запросах динамическая компиляция выражений в Spark SQL позволяет получить ускорение до 3.5 раз по сравнению с интерпретацией.

PostgreSQL

PostgreSQL [32] — высокопроизводительная объектно-реляционная СУБД, поддерживающая язык запросов SQL и расширяемая пользовательскими функциями, операторами, индексами, типами данных и процедурными языками. PostgreSQL использует построчное хранение кортежей (N-ary Storage Model [9; 21]).

В число СУБД, основанных на PostgreSQL [33], входят нереляционная СУБД ToroDB [34], колоночная СУБД Vertica [35], графовая СУБД AgensGraph [36], распределённые аналитические СУБД Redshift [37], Greenplum [38] и DeepgreenDB [39] и многие другие.

В PostgreSQL начиная с версии 11 [40] добавлена поддержка JIT-компиляции [41] с использованием LLVM для ускорения вычисления выражений и преобразования кортежей. Вычисление выражений производится при обработке предложений WHERE, целевых списков, агрегатов и проекций. Ускорение достигается за счёт генерации специализированного кода, предназначенного для каждого конкретного случая. Извлечение атрибутов кортежей — процедура пе-

ревода кортежа с диска в развёрнутое представление в памяти. Оно ускоряется за счёт создания функций, предназначенных для определённой структуры таблицы и количества извлекаемых столбцов.

На запросе Q1 из тестового набора TPC-H JIT-компиляция выражений и преобразования кортежей в PostgreSQL позволяет получить ускорение до 30% [42].

GreenPlum

В [38] описывается использование динамической компиляции запросов с использованием LLVM в СУБД Greenplum. Исследователи выделяют два способа применения динамической компиляции в СУБД: компиляция выражений и компиляция запросов — и разделяют некоторые существующие СУБД в соответствии с этим на два класса.

Динамическая компиляция в Greenplum используется на уровне выражений и горячих функций, что позволяет расширять множество компилируемых функций инкрементально в рамках существующей СУБД. Результат профилирования на запросах TPC-H [29] выделил три множества функций-кандидатов для компиляции: функции доступа к атрибутам, функции интерпретатора выражений и агрегатные функции.

Процесс замены каждой отдельной функции состоит из следующих шагов (рис. 1.10).

1. Замена непосредственных вызовов функции вызовами по указателю, помещённому во внутренние структуры данных СУБД.
2. Компиляция функции-кандидата в LLVM IR и в машинный код во время инициализации запроса.
3. Замена указателей в структурах данных на указатели в результирующий машинный код.
4. Освобождение скомпилированного кода и сбрасывание указателей во время финализации запроса.

Метод позволяет компилировать только те обобщённые функции, выполнение которых связано с существенными накладными расходами в профиле работы СУБД, а также вводить определённые ограничения при компиляции

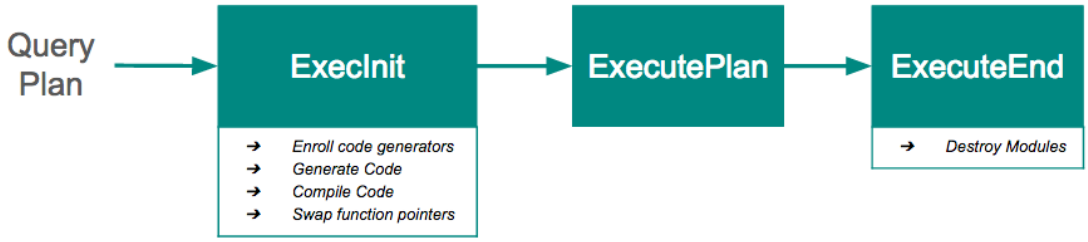


Рисунок 1.10 — Компиляция горячих функций в Greenplum.

каждой конкретной функции, при нарушении которых прерывать компиляцию до шага 3 (замены указателей), что приведёт к использованию при выполнении запроса обобщённой версии соответствующей функции.

Недостатком является необходимость сохранения интерфейсов между обобщённым и компилируемым кодом.

На запросе TPC-H Q1 применение динамической компиляции позволило получить двукратное ускорение, которое складывается в основном из ускорения функций интерпретатора выражений (1.25 раза) и агрегатных функций (1.35 раза).

1.2.2 Динамическая компиляция запросов

HyPer

HyPer [43] — резидентная СУБД, основанная на модели явных циклов и динамической компиляции запросов в машинный код с использованием инфраструктуры LLVM. HyPer был разработан коллективом Мюнхенского Технического Университета во главе с профессором Томасом Нойманом в 2011 году [44] и является коммерческой СУБД [45] с закрытым исходным кодом.

Исследователи утверждают [17; 18], что с ростом объёмов основной памяти производительность СУБД больше определяется эффективностью использования процессора и классическая Volcano-модель, несмотря на свою простоту и гибкость, не позволяет использовать процессор достаточно эффективно из-за нелокального доступа к памяти и ошибок в прогнозировании переходов.

Предлагаемый и реализованный в HyPer подход состоит в определении *границ конвейеризации (pipeline boundaries)* — операторов плана выполнения запроса, которые приводят к материализации кортежей (рисунок 1.11), и компиляции операторов в пределах границ конвейеризации в один цикл обработки данных (рисунок 1.12), оставляя таким образом границы между операторами только там, где необходимо: материализация кортежей происходит только на границах между циклами, которые определяются планом выполнения запроса и используемыми в нём алгоритмами обработки данных, а на каждой итерации одного цикла происходит применение нескольких операторов к одному кортежу, что позволяет максимально эффективно использовать для хранения атрибутов кортежа регистры процессора.

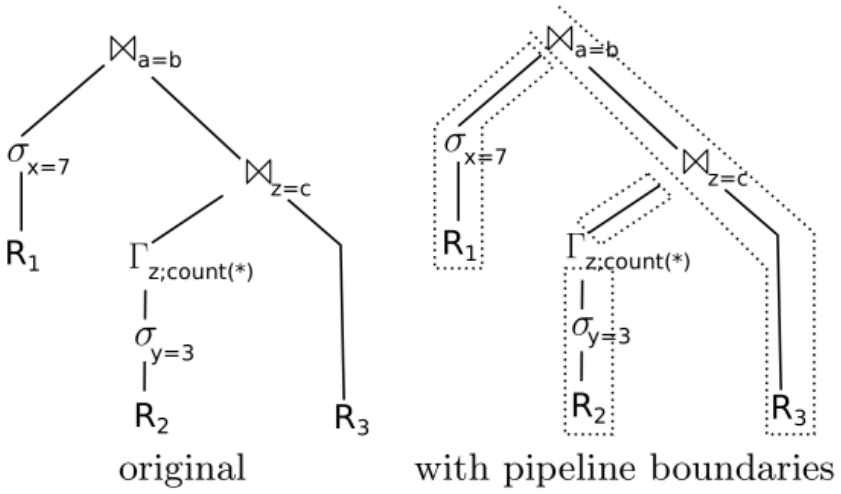


Рисунок 1.11 — Определение границ конвейеризации в HyPer.

Генерация кода выполняется при помощи методов `produce` и `consume`, сопоставленных операторам плана выполнения запроса. Метод `produce` определён для всех операторов и служит для построения кода, состоящего из одного или нескольких (в количестве листовых вершин) циклов и реализующего функциональность соответствующего поддерева плана выполнения запроса. `produce` рекурсивно вызывается для потомков внутренних операторов плана выполнения запроса. Когда выполнение доходит до листового оператора, генерируется заголовок цикла и вызывается метод `consume` родительского оператора, который служит для генерации кода, обрабатывающего приём очередного кортежа от дочернего оператора. Методы `consume` определены только для внутренних операторов плана в количестве, равном общему числу дуг в дереве плана выполнения запроса. Генерация кода для всего запроса производится посредством вызова метода `produce` на самом внешнем операторе плана. Вызовы `produce`

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

Рисунок 1.12 — Результат компиляции границ конвейеризации в HyPer (псевдокод).

и `consume`, таким образом, производят обход дерева плана в глубину, по завершении которого самый внешний вызов `produce` возвращает код, реализующий в модели явных циклов весь исходный запрос.

Интерфейс, представленный функциями `produce` и `consume`, подобен интерфейсу итераторов в Volcano-модели и упрощает независимую разработку реляционных операторов, но существует только во время компиляции запроса — результат компиляции не содержит вызовов этих функций и представлен всего несколькими императивными циклами (рисунок 1.12).

Для генерации машинного кода в HyPer используется инфраструктура LLVM [28] и промежуточное представление LLVM IR. Отмечаются следующие преимущества над альтернативными технологиями:

- приемлемое время компиляции (по сравнению с языками высокого уровня);
- достаточную низкоуровневость и контроль над результирующим машинным кодом (например, в части использования флагов арифметического переполнения);
- наличие неограниченного множества виртуальных регистров;
- платформо-независимость;
- типобезопасность;
- наличие большого числа встроенных оптимизаций.

Для максимизации продуктивности, упрощения поддержки и уменьшения времени компиляции существенная часть внутренней логики операторов реализована на C++ и вызывается из сгенерированного кода. Тем не менее, для обеспечения максимальной производительности необходимо, чтобы наиболее горячие участки кода, в особенности части, ответственные за обработку кортежей в циклах в пределах границ конвейеризации, были реализованы на LLVM IR во избежание накладных расходов на вызовы функций на C++, в частности на сохранение и загрузку регистров согласно используемому соглашению о вызовах.

В [17] исследователи отмечают некоторые особенности генерации кода для оптимизации производительности, в частности:

- загрузку атрибутов из памяти немного раньше, чем необходимо, для сокрытия задержек работы с памятью;
- использование циклов с постусловием вместо циклов с предусловием для улучшения предсказания переходов.

В статье также описывается обработка в цикле нескольких кортежей одновременно с использованием SIMD-инструкций, доступных в современных процессорах.

В [18] дополнительно описывается представление SQL-значений и операций в коде на LLVM IR, при этом как правило одному SQL-значению соответствует несколько LLVM-значений (например, число или указатель и флаг *null* или длина строки) и одной операции — несколько элементарных операций LLVM (например, сложение чисел и проверка переполнения); абстракция времени компиляции для генерации графа потока управления (условий и циклов) и абстракция времени компиляции для представления кортежей (в сжатом или дематериализованном виде). Использование абстракций времени компиляции существенно упрощает разработку и поддержку динамического компилятора и при этом не приводит к дополнительным затратам времени выполнения.

На синтетических запросах динамическая компиляция в HyPer позволяет получить ускорение от 2 до 8 раз в сравнении с интерпретацией в зависимости от запроса, а на бенчмарке TPC-H — до 3.7 раз в сравнении с колоночной СУБД VectorWise [46].

Hekaton

Hekaton [47; 48] — это расширение SQL Server [49], включающее в себя таблицы и индексы для данных в основной памяти, неблокирующие структуры данных для эффективного многопоточного выполнения и MVCC и компилятор запросов. Исследователи отмечают, что производительность СУБД при выполнении OLTP-запросов зависит от трёх основных параметров: количества выполняемых инструкций, количества циклов на инструкцию и коэффициента масштабируемости — причём последние два параметра суммарно могут дать только прирост производительности в 3–4 раза. Для ускорения СУБД в 10–100 раз необходимо, таким образом, существенно сократить количество выполняемых инструкций.

Компилятор запросов Hekaton принимает на вход структуры результат работы оптимизатора и генерирует код на промежуточном представлении PIT (Pure Imperative Tree), на основе которого после серии преобразований генерируется код на языке C, компилируемый и загружаемый в процесс SQL Server для выполнения.

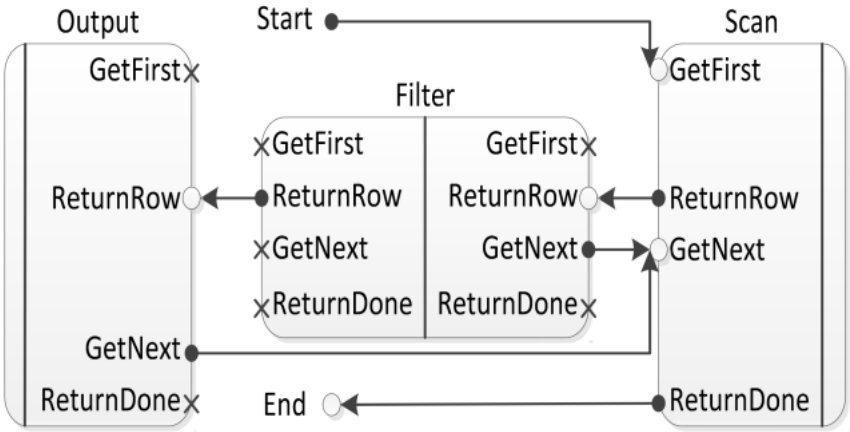


Рисунок 1.13 — Межоператорный поток управления в Hekaton.

Операторы в Hekaton реализуют интерфейс, состоящий из функций `GetFirst`, `GetNext`, `ReturnRow` и `ReturnDone`, что позволяет комбинировать операторы произвольным образом согласно плану выполнения запроса.

В Hekaton реализована модель явных циклов при помощи объединения кода операторов в одну функцию и трансляции переходов между ними — вызовов функций `GetFirst`, `GetNext` и т. д. — в безусловные переходы между соответствующими операторам базовыми блоками (см. рис. 1.13). Результирующий код

```
    /*Seek*/
l_17:; /*seek.GetFirst*/
    hr = (HkCursorHashGetFirst(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->Transaction),
        0, 0, 1,
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
l_20:; /*seek.1*/
    if ((hr == 0))
    {
        goto l_14 /*filter.child.ReturnRow*/ ;
    }
    else
    {
        goto l_12 /*query.ReturnDone*/ ;
    }
l_21:; /*seek.GetNext*/
    hr = (HkCursorHashGetNext(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->ErrorObject),
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
    goto l_20 /*seek.1*/ ;
    /*Filter*/
l_14:; /*filter.child.ReturnRow*/
    result_22 = ((rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ->hkc_1 /*[Id]*/ ) ==
        ((long)((valueArray[1 /*@id*/ ]).SignedIntData)));
    if (result_22)
    {
        goto l_13 /*output.child.ReturnRow*/ ;
    }
    else
    {
        goto l_21 /*seek.GetNext*/ ;
    }
```

Рисунок 1.14 — Пример результирующего кода в Hekaton.

(рис. 1.14) из соображений безопасности не содержит идентификаторов исходного запроса, что ещё больше затрудняет экспертный анализ, но проведённое исследование показало, что генерация кода в одну функцию позволяет минимизировать как число выполняемых инструкций, так и размер бинарного кода.

Сравнение с интерпретатором запросов показало сокращение количества выполняемых инструкций до 10–15 раз.

MemSQL

MemSQL [50] — резидентная СУБД, в которой реализована компиляция запросов в машинный код при помощи инфраструктуры LLVM. Исследовате-

ли выделяют три направления оптимизации производительности программных систем:

- оптимизация подсистемы ввода–вывода: планировка (scheduling), перемещение данных, распределение нагрузки;
- оптимизация потребления памяти вычислительных узлов;
- оптимизация использования ЦПУ для вычислений — в первую очередь, сокращение количества исполняемых инструкций.

В дисковых СУБД оптимизация ввода–вывода имеет гораздо более важное значение, потому что ввод–вывод в дисковых СУБД является самым «узким местом». В резидентных СУБД, напротив, оптимизации памяти и вычислений выходят на первый план и компиляция запросов в эффективный машинный код может дать существенный прирост производительности. В MemSQL для компиляции используются высокоуровневое промежуточное представление MPL (MemSQL Plan Language) и промежуточное представление среднего уровня MBC (MemSQL Bit Code), которое затем транслируется в низкоуровневое промежуточное представление LLVM IR.

Исследователи отмечают, что компиляция запросов даёт как количественные, так и качественные преимущества для конечного пользователя: например, возможность использования систем визуализации и мониторинга реального времени.

HIQUE

В HIQUE [51] рассматривается динамическая компиляция SQL-запросов на основе шаблонов кода.

Алгоритмы, используемые в традиционных СУБД, разработаны в первую очередь для оптимизации использования подсистемы ввода-вывода, что в современных условиях и при современных объёмах доступной памяти недостаточно. В случаях, когда вся база данных или значительная её часть умещается в оперативной памяти, узким местом производительности является процессор. Исследователи отмечают, что изменение подсистемы хранения для упрощения обработки данных — одно из предложенных решений — слишком радикально меняет архитектуру существующих СУБД. Более ортогональным решением яв-

ляется компиляция запросов в машинный код, которая позволяет значительно сократить накладные расходы на исполнение плана запроса по сравнению с интерпретаторами, реализующими Volcano-модель и основанными на абстракции итератора.

Предлагаемый подход *целостного выполнения запросов (holistic query evaluation)* основан на динамической компиляции плана выполнения запроса в программу на некотором языке программирования на основе шаблонов кода, разработанных для каждого оператора. Полученная программа оптимизируется как одно целое, при этом применяются как платформо-зависимые, так и межоператорные оптимизации, недоступные в интерпретаторе из-за существующих границ абстракции между различными операторами. Динамическая компиляция позволяет убрать эти границы абстракции, уменьшить число вызовов функций и увеличить локальность данных, в том числе за счёт более эффективного использования регистров процессора.

HIQUE использует построчное хранение кортежей (N-ary Storage Model). Архитектура подсистемы обработки запросов представлена на рис. 1.15.

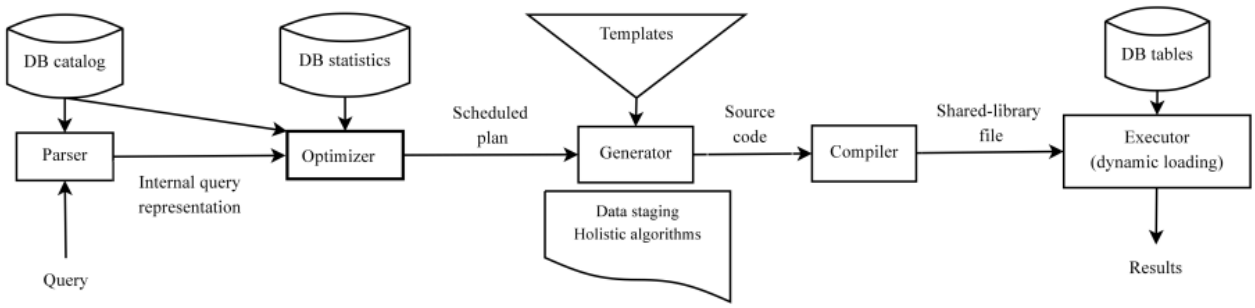


Рисунок 1.15 — Архитектура обработки запросов в HIQUE.

Результатом работы оптимизатора является топологически отсортированная последовательность операторов, в которой на вход оператор o_i принимает или выход оператора o_j, где $j < i$, или первичную таблицу в базе данных. Компиляция каждого оператора состоит из этапа предобработки, который заключается в определении используемых атрибутов и вычислении смещений, применении предикатов сканирования и вставки дополнительных операторов сортировки и секционирования, где необходимо, и следующего за ним этапа генерации кода, который заключается в инстанциировании соответствующего шаблона кода — в статье приведены примеры шаблонов для некоторых операторов. Промежуточные результаты работы каждого оператора сохраняются

во временные таблицы: между каждой парой операторов неявно вставляется точка материализации.

Результатом компиляции является функция на языке C, которая затем транслируется в объектный код и подгружается в основную программу для выполнения.

Экспериментальная оценка производительности на запроса Q1 из тестового набора TPC-H (масштаб базы 1ГБ) показывает прирост производительности от 4 раз (в сравнении с колоночной СУБД MonetDB [52] версии 5.8.2) до 167 раз (в сравнении с СУБД PostgreSQL версии 8.2.7). Стоит отметить, что HIQUE реализован как прототип, исполнитель запросов создан с нуля и не интегрировался в существующую систему. Также используемые в HIQUE шаблоны были написаны вручную, а не генерированы из исходного кода.

LegoBase

LegoBase [53] — компилятор запросов, реализованный на языке Scala. Это первая СУБД, реализованная с использованием фреймворка DBLAB [54] при помощи высокоуровневого программирования.

Для оптимизации производительности в LegoBase используется технология многоуровневой компиляции и фреймворк LMS [55]. Реализованы следующие оптимизации на уровне промежуточного представления LMS специально для LegoBase:

- оптимизация структур данных: частичное вычисление адресов и параметров обращений, замена ассоциативных структур данных (хеш-таблиц) линейными (массивами);
- изменение модели выполнение с Volcano-модели на push: исследователи отмечают, что реализация автоматического изменения модели выполнения оказалась возможна только благодаря достаточно высокому уровню абстракции реализации операторов на языке Scala;
- удаление лишней материализации кортежей на границе операторов;
- изменение схемы расположения данных таблиц: замена построчного хранения поколоночным.

В результате компиляции и оптимизации LMS генерирует код на языке Scala. Для достижения максимальной производительности в LegoBase реализована дополнительная трансляция результирующего кода в код на языке C. Исследователи отмечают, что трансляция в C потребовала решения двух проблем:

- Трансляции вызовов библиотечных функций и структур данных. Для решения этой проблемы в LegoBase используется библиотека GLib [56].
- Трансляции модели управления памятью. В языке Scala используется сборщик мусора, поэтому во время трансляции кода со Scala в C необходимо обеспечить своевременное освобождение неиспользуемой памяти во избежание утечек памяти. Для решения этой проблемы в LegoBase используется ручное управление памятью.

Полученный код на языке C компилируется при помощи Clang [57] и исполняется.

Исследователи отмечают следующие преимущества по сравнению с другими методами компиляции запросов:

- типобезопасность (обеспечивается использованием высокоуровневого языка);
- относительная простота реализации и поддержки;
- поддержка межоператорных оптимизаций (по сравнению с методами, основанными на шаблонах);
- поддержка высокоуровневых оптимизаций (обеспечивается более высоким уровнем промежуточного представления по сравнению с, например, промежуточным представлением LLVM).

Эксперименты с использованием бенчмарка TPC-H показывают, что LegoBase достигает большей производительности, чем СУБД HyPer [43], в которой для компиляции запросов используется LLVM [28]. Сравнение с HyPer указывает на упущенные оптимизационные возможности, вызванные слишком низким уровнем абстракции, предоставляемым LLVM IR, и реализованные при помощи использования более высокоуровневого промежуточного представления LMS.

1.3 Выводы

Из проведённого обзора работ в области динамической компиляции запросов можно заметить, что во всех рассмотренных динамических компиляторах для распределённых систем компиляция реализуется на уровне выражений и горячих функций. Здесь показателен подход Greenplum, основанный на замене указателей в структурах данных, изначально указывающих на обобщённый код, во время подготовки запроса к выполнению.

Основной подход к реализации компиляторов запросов — реализация генератора низкоуровневого кода запроса на основе модели явных циклов (HyPer, Hekaton, MemSQL). Реализация модели явных циклов (push-модели) в рамках динамического компилятора запросов описана в [17]. Она состоит в генерации по плану выполнения запроса 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.

В HIQUE реализована более простая модель, основанная на генерации кода операторов в топологическом порядке с материализацией во временные таблицы на каждом шаге. Тем не менее, даже такая модель позволила получить существенное ускорение относительно коммерческих СУБД.

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

В таблице 1 представлен список СУБД по типу реализованных динамических компиляторов.

Таблица 1 — Сводная таблица динамических компиляторов запросов

СУБД	**Проприе-тарная**	**Единица компиляции**	**Комментарий**
Apache Impala [23; 24]	нет	Выражения	доступ к атрибутам; LLVM
Greenplum [38]	нет	Выражения	горячие функции; LLVM
Spark SQL [30; 31]	нет	Выражения	—
PostgreSQL [41]	нет	Выражения	доступ к атрибутам; LLVM
HyPer [17; 43]	да	Запрос	push-модель; LLVM
SQL Server Hekaton [47; 48]	да	Запрос	push-модель; C
MemSQL [50]	да	Запрос	push-модель; многоуровневая трансляция; LLVM
DBLAB LegoBase [53]	нет	Запрос	push-модель; многоуровневая компиляция; Scala→C
HIQUE [51]	—	Запрос	шаблоны кода; C

Глава 2. Метод динамической компиляции SQL-запросов

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 кода в промежуточном представлении с дальнейшей оптимизацией и трансляцией в машинный код во время выполнения. Общая схема динамического компилятора запросов показана на рисунке 2.1.

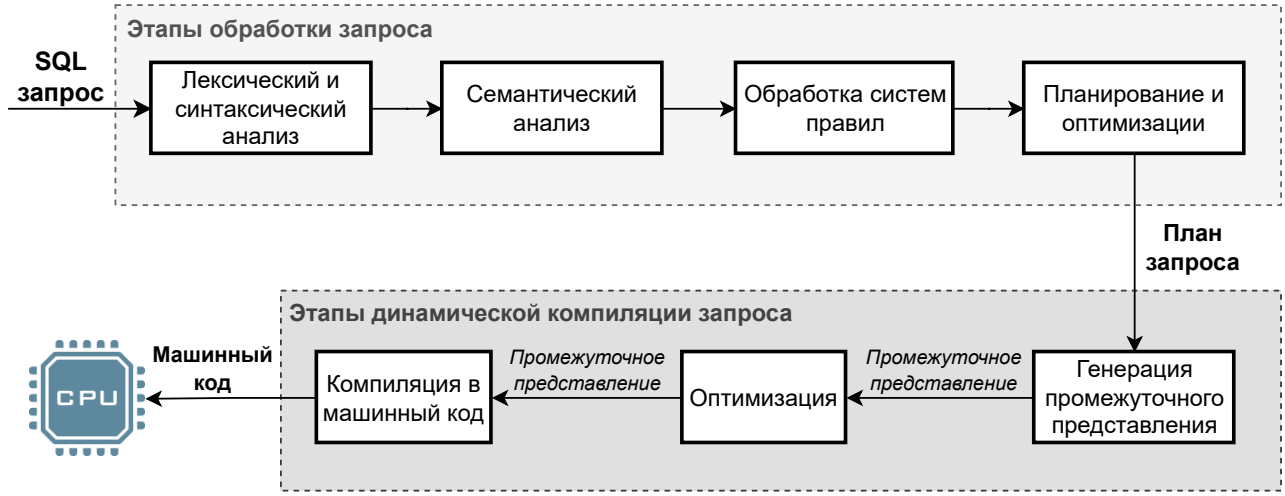


Рисунок 2.1 — Процесс обработки запроса в динамическом компиляторе.

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, полученный после этапов обработки.

В общем случае 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.

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 выпол-

няются, эти функции могут быть проверены и отлажены с использованием существующих отладчиков и визуализаторов.

Интерфейс оператора в модели явных циклов, принцип выделения этого интерфейса в контексте динамического компилятора и процесс трансформации описаны в разделе 2.1.

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.

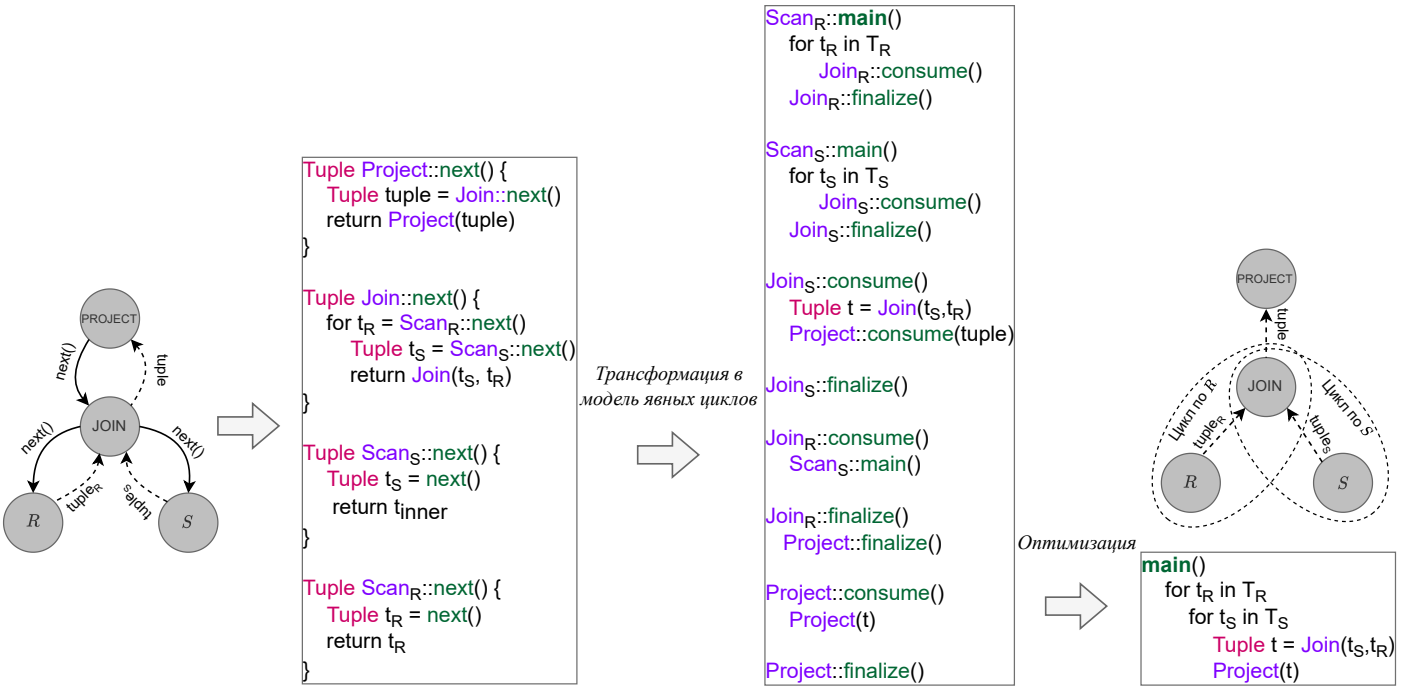
На третьем этапе выполняется *компиляция* сгенерированного оптимизированного промежуточного представления запроса в машинный код целевой архитектуры.

На рисунке 2.2 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()*.

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).

2.1 Трансформация операторов плана запроса в модель явных циклов

В ходе анализа модели явных циклов (1.1.4) и обоснования недостатков Volcano-модели (1.1.1) мы говорили, что компилятор не способен применить сложные оптимизации к функции *next()* по причине рекурсии и косвенных вызовов. Ключевая идея перехода с модели Volcano на модель явных циклов заключается в возможности применения метода динамической компиляции для



План запрос в Volcano-модели | Алгоритм операторов запроса в Volcano-модели | Сгенерированный код функций операторов в модели явных циклов | Оптимизированный код запроса в модели явных циклов

Рисунок 2.2 — Схема трансформации алгоритмов операторов плана запроса в модель явных циклов

генерации специализированного кода, который в большей степени подходит для применения оптимизации по встраиванию функций (inlining). Данная оптимизация нивелирует накладные расходы, ассоциируемые с вызовом функций, улучшает локальность кода, а также позволяет удерживать обрабатываемые данные в регистрах на более продолжительный период времени.

Далее мы опишем специальный интерфейс оператора в модели явных циклов и исследуем принцип выделения этого интерфейса в контексте динамического компилятора, необходимого для «разворота» потока обработки данных.

2.1.1 Интерфейс оператора в модели явных циклов

Определим интерфейс оператора в модели явных циклов. 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, имеющего 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 выполнения плана запроса в обратном порядке. Код, реализующих функции интерфейса в модели явных циклов, генерируется во время выполнения на основе имеющейся реализации алгоритма оператора в Volcano-модели. Декомпозиция алгоритмов операторов Volcano-модели на функции интерфейса в модели явных циклов подробно описана в разделе 2.1.4.

Чтобы проиллюстрировать этот интерфейс, мы на примере рассмотрим два оператора: последовательного сканирования таблицы (*Scan*) и соединения хешированием (*HashJoin*).

Ниже представлен псевдокод оператора *Scan*, реализующий в упрощённом виде описанный выше интерфейс оператора в модели явных циклов:

```
Scan::main()
    FOR tuple in T:
        IF P(tuple):
            call Parent::consume()
    call Parent::finalize()
```

В модели явных циклов выполнение начинается с листового сканирующего оператора, который в цикле сканирует таблицу, проверяет предикат *P(tuple)* и передаёт кортеж своему родителю посредством вызова функции *consume()*

родительского оператора. Так как оператор *Scan* является листовым узлом, то есть не имеет дочерних поддеревьев, соответственно у него не может быть функций *consume()* и *finalize()*, и реализация оператора находится в функции *main()*.

Оператор соединения хешированием является более сложным. В зависимости от того, с какой стороны пришли данные: из правого (внутреннего) или левого (внешнего) оператора, вызывается соответствующая функция *inner_consume()*, которая выполняет запись данных в хеш-таблицу, либо *outer_consume()*, которая вызывается, когда получены все данные от внутреннего оператора, и выполняет непосредственно соединение. По окончанию соединения вызывается функция *finalize()* от родительского узла-оператора для уведомления, что дочернее поддерево завершило производство новых кортежей.

Ниже представлен псевдокод оператора *HashJoin*, реализующий интерфейс оператора в модели явных циклов:

```
HashJoin::inner_consume()
    hashtable.put(tuple_inner)

HashJoin::outer_consume()
    FOR hashtable.match(hash(tuple_outer)):
        join(tuple_outer, tuple_inner)
        call Parent::consume()

HashJoin::inner_finalize()
    call Child_outer::main()

HashJoin::outer_finalize()
    call Parent::finalize()
```

Для простого запроса из рисунка 1.2, содержащего два сканирующих оператора и один оператор соединения хешированием, выполнение запроса начнётся путём вызова функции $Scan_{inner}$*::main()* внутреннего листового оператора. Эта функция в цикле по внутренней таблице вызовет *HashJoin::inner_consume()* для заполнения хеш-таблицы. После того, как все кортежи с внутренней стороны сохранены в хеш-таблице, управление вернётся оператору сканирования, который вызовет функцию *HashJoin::inner_finalize()*, которая, в свою очередь, вызовет функцию $Scan_{outer}$*::main()* для выполнения сканирования внешней таблицы и вызова функции *HashJoin::outer_consume()* для выполнения соединения.

Результирующий код в модели явных циклов после применения оптимизации встраивания представлен в виде двух последовательных циклов:

```
FOR tuple_inner in T_inner:
    IF P(tuple_inner):
        hashtable.put(tuple_inner)

FOR tuple_outer in T_outer:
    IF P(tuple_outer):
        FOR hashtable.match(hash(tuple_outer)):
            join(tuple_outer, tuple_inner)
```

Как можно заметить, ключевую роль для получения кода в виде вложенных циклов играет оптимизация по встраиванию функций, которая встраивает всю цепочку вызовов функций *consume()* до первого прерывающего оператора, как, например операторы сортировки или агрегации, которые аккумулируют все входные данные прежде чем выдать первый выходной кортеж. В получившемся коде мы предполагаем, что сначала правая сторона производит данные и сохраняет их в хеш-таблицу и после того, как все данные произведены, активируется генерация данных с левой стороны и выполняется поиск совпадений в хеш-таблице с дальнейшим соединением.

Далее мы опишем принцип генерации и выделения функций интерфейса для трансформации «на лету» операторов модели Volcano в модель явных циклов в рамках динамического компилятора запросов.

2.1.2 Генерация функций интерфейса оператора в модели явных циклов

Для реализации модели явных циклов у каждого оператора определены функции-генераторы, которые вызываются во время рекурсивного обхода дерева плана запроса в прямом порядке и «на лету» выполняют генерацию кода во внутреннем представлении описанных ранее функций интерфейса оператора *main()*, *consume()* и *finalize()*. Для каждого оператора функция-генератор вызывается один раз. Результатом генерации является одна императивная программа для каждого дерева плана запроса.

Генерация кода для запроса производится путём вызова функции-генератора от корневого оператора плана запроса, который производит обход дерева плана в глубину и вызов функций-генераторов от дочерних узлов-операторов.

По завершении функция-генератор возвращает код во внутреннем представлении, реализующий в модели явных циклов весь исходный запрос.

Генерация промежуточного представления происходит на основе имеющейся реализации алгоритма соответствующего оператора в конкретной СУБД. В зависимости от типа оператора порядок генерации его функций интерфейса и функций дочерних узлов может изменяться. Далее к сгенерированному внутреннему представлению применяются оптимизации и выполняется компиляция в машинный код.

На листинге 2.1 представлен пример функции-генератора для оператора *HashJoin*, где `IR` — это тип промежуточного представления (Intermediate Representation).

Листинг 2.1: Семантика функций-генераторов оператора HashJoin.

```
IR HashJoin::generate(IR Parent.consume, IR Parent.finalize) {

  HashJoin.outer_consume = generateOuterConsume(Parent.consume);
  HashJoin.outer_finalize = generateOuterFinalize(Parent.finalize);

  IR main_outer = Child_outer::generate(HashJoin.outer_consume, HashJoin.outer_finalize);

  HashJoin.inner_consume = generateInnerConsume();
  HashJoin.inner_finalize = generateInnerFinalize(main_outer);

  IR main_inner = Child_inner::generate(HashJoin.inner_consume, HashJoin.inner_finalize);

  return main_inner;
}
```

Функция *generate* у данного оператора получает на вход сгенерированные функции *consume* и *finalize* родительского оператора, выполняет генерацию во внутреннем представлении собственных функций *consume* и *finalize*, сохраняет их в соответствующие поля и вызывает функции-генераторы дочерних операторов. Из-за особенности оператора *HashJoin* мы сначала генерируем внешние функции *consume* и *finalize*, а потом передаём их генератору внешнего дочернего оператора, который сгенерирует цикл по внешней таблице. Далее происходит генерация внутренних функций *consume* и *finalize* и передача их генератору цикла по внутренней таблице. Итоговый сгенерированный код будет состоять из двух последовательных циклов: в первом цикле будет заполняться хеш-таблица, а во втором — поиск совпадений в хеш-таблице и соединение.

Стоит отметить, что генерация кода происходит согласно интерфейсу, описанному в разделе 2.1.1, и в данном примере оператор *HashJoin* передает генератору вышестоящего оператора сгенерированную функцию *main()* внутреннего листового узла.

Принцип декомпозиции алгоритмов операторов СУБД на функции интерфейса в модели явных циклов описан в разделе 2.1.4.

2.1.3 Механизм прерывания обработки запроса в модели явных циклов

Одно из основных различий между моделями Volcano и явных циклов заключается в том, каким способом они управляют выполнением запроса. В интерпретаторе с моделью Volcano корневой узел плана запроса выполняет вызов функции *next()*, который затем переходит к потомку, который, в свою очередь, может вызывать функцию *next()* у своего дочернего узла (унарный оператор) или дочерних узлов (бинарный оператор). Рекурсивные вызовы продолжаются до тех пор, пока не будут достигнуты листовые узлы. Если оператор возвращает *NULL* вместо нового кортежа, это означает, что поток исчерпан. Этот механизм позволяет любому оператору досрочно завершить обработку запроса.

Как было описано в разделе 2.1, в динамическом компиляторе запросов для управления преобразованием кода из модели Volcano в модель явных циклов алгоритм нелистового оператора разбивается на две части — *consume()* и *finalize()*. Эти этапы позволяют не только выполнять генерацию кода сверху вниз, но и составлять сгенерированный код таким образом, чтобы его можно было встроить после процедуры оптимизации.

Однако 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, что приводит к производству лишних кортежей, которые не будут обработаны. Для управления выполнением запросов в модели явных циклов с возможностью досрочного завершения 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. Значение статуса — это по сути количество вложенных циклов, которые необходимо прервать для продолжения обработки запроса. Механизм прерывания потока обработки данных активным образом используется в процессе работы некоторых операторов, когда оператор по собственной или родительской инициативе хочет приостановить генерацию новых кортежей и вернуть управление вызывающей стороне.

В зависимости от полученного статуса могут выполняться следующие действия:

- если статус равен `0`, то оператор продолжит процесс генерации новых кортежей для родительского узла;
- если статус равен `1`, то будет выполнен вызов функции *finalize()* родительского узла и возвращён её результат;
- если статус больше или равен `2`, то значение статуса будет скорректировано в зависимости от типа оператора и возвращено в качестве результата.

Возвращаемое значение статуса обрабатывается листовым оператором сканирования для принятия решения о продолжении или завершении потока обработки данных.

Далее приведём пример применения операции на конкретном плане запроса, но сначала рассмотрим ещё один компонент механизма — функцию для подсчёта вложенности, на которой вызывается функция *consume()*: $CD(n) - ConsumeDepth$:

$$CD(n) = \begin{cases} 1, & \text{если } n \in Scan \cup IndexScan \cup Sort \cup HashAgg \\ CD(n_{outer}), & \text{если } n \in Limit \\ CD(n_{outer}) + 1, & \text{если } n \in HashJoin \\ CD(n_{outer}) + CD(n_{inner}), & \text{если } n \in NestedLoop \cup MergeJoin \end{cases} \tag{2.1}$$

где:

n — узел-оператор плана запроса,

n_{outer} — левый дочерний узел оператора n,

n_{inner} — правый дочерний узел оператора n.

$CD(n)$ является функцией времени компиляции, и вычисленное ею значение является константой, которая используется в сгенерированном коде.

Например, функция *consume()* оператора *Limit* возвращает в качестве статуса $CD(n_{outer})$ для одновременного выхода из всех циклов в своём поддереве плана запроса — именно на такой глубине функция *consume()* будет вызвана.

На листинге 2.2 представлен псевдокод примера применения механизмов прерывания на примере запроса `select * from R join S limit 1`. Возможный план этого запроса состоит из двух операторов сканирования (*Scan*), оператора соединения вложенным циклом (*NestedLoop*) и оператора-ограничителя результата (*Limit*). В данном случае нас интересует только форма плана и типы узлов, а не результат его выполнения.

Выполнение начинается с вызова функции *main()* от внешнего листового оператора: $Scan_{outer}$*::main()*, которая в цикле по внешней таблице вызывает функцию *NestedLoop::outer_ consume()*, выполняющую сброс внутреннего состояния правого поддерева и запуск обработки кортежей с внутренней стороны: $Scan_{inner}$*::main()*. После получения первого кортежа с внутренней стороны и выполнения соединения в *NestedLoop::inner_ consume()* вызывается функция *Limit::consume()*, которая выполняет проверку параметра ограничения результата и возвращает соответствующий статус. Так как в нашем примере параметр *Limit* равен 1, то после получения первого же кортежа функция *Limit::consume()* вернёт статус 2 (см. формулу (2.1)):

$$CD(Limit) = CD(NestedLoop) = CD(Scan_{outer}) + CD(Scan_{inner}) = 2$$

— указывая, что нижестоящие операторы должны перейти к этапу завершения.

На листинге 2.3 представлен псевдокод запроса в модели явных циклов после применения оптимизации по встраиванию функций. Стоит отметить, что появление в коде оператора *GOTO* — это всего лишь артефакт оптимизации встраивания.

После применения таких оптимизирующих преобразований, как подстановка констант, вычисление арифметических выражений, удаление «мёртвого» кода и т.д., код запроса в модели явных циклов будет сведён к двум вложенным циклам (см. листинг 2.4) без лишних операций копирования и сравнения. Как можно увидеть, использование модели явных циклов в совокупности с динамической компиляцией увеличивает локальность кода и данных, что в перспективе приводит к ускорению выполнения запросов.

Для лучшего понимания раличий модели Volcano и модели явных циклов проиллюстрируем в сравнении порядок обхода плана запроса для примера выше в обеих моделях для получения одного кортежа. В модели Volcano поря-

Листинг 2.2: Работа механизма прерывания обработки в модели явных циклов.

```
Scan_inner::main()
    FOR tuple_inner in T_inner:
        IF P(tuple_inner):
            status = NestedLoop::inner_consume()
        ELSE
            status = 0
        SWITCH(status):
            CASE 0: continue
            CASE 1: break
            DEFAULT: return status - 1
    return NestedLoop::inner_finalize()

Scan_outer::main()
    FOR tuple_outer in T_outer:
        IF P(tuple_outer):
            status = NestedLoop::outer_consume()
        ELSE
            status = 0
        SWITCH(status):
            CASE 0: continue
            CASE 1: break
            DEFAULT: return status - 1
    return NestedLoop::outer_finalize()

NestedLoop::inner_consume()
    IF match(tuple_inner, tuple_outer):
        join(tuple_inner, tuple_outer)
        return Limit::consume()
    return 0

NestedLoop::outer_consume()
    RESET(T_inner)
    return Scan_inner::main()

NestedLoop::inner_finalize()
    return 0

NestedLoop::outer_finalize()
    return Limit::finalize()

Limit::consume()
    print(tuple)
    limit_index++
    IF limit_index > 1:
        return 2 % CD(NestedLoop) = 2
    return 0

Limit::finalize()
    return 0

main() = Scan_outer::main()
```

Листинг 2.3: Псевдокод запроса в модели явных циклов после применения оптимизации по встраиванию функций.

```
main()
    FOR tuple_outer in T_outer:
        IF P(tuple_outer):
            RESET(T_inner)
            FOR tuple_inner in T_inner:
                IF P(tuple_inner):
                    IF match(tuple_inner, tuple_outer):
                        join(tuple_inner, tuple_outer)
                        print(tuple)
                        limit_index++
                        IF limit_index > 1:
                            status = 2
                        ELSE
                            status = 0
                    ELSE
                        status = 0
                ELSE
                    status = 0
                SWITCH(status):
                    CASE 0: continue
                    CASE 1: break
                    DEFAULT: status = status - 1
                             GOTO break_inner
            status = 0
        ELSE
            status = 0
break_inner:
        SWITCH(status):
            CASE 0: continue
            CASE 1: break
            DEFAULT: return status - 1
    return 0
```

Листинг 2.4: Псевдокод запроса в модели явных циклов после всех оптимизаций.

```
main()
    FOR tuple_outer in T_outer:
        IF P(tuple_outer):
            RESET(T_inner)
            FOR tuple_inner in T_inner:
                IF P(tuple_inner):
                    IF match(tuple_inner, tuple_outer):
                        join(tuple_inner, tuple_outer)
                        print(tuple)
                        return 0
    return 0
```

док обхода дерева плана запроса визуально представлен на рисунке 2.3: обход начинается с вызова функции *next()* (порядковый номер 1) от узла *Limit*, обойдя дерево плана запроса (порядковые номера 2—7) и вернувшись в узел *Limit*, исполнитель отправит клиенту кортеж *tuple* (порядковый номер 8) в качестве результата. При следующем вызове *next()* оператор *Limit* самостоятельно прервёт процесс обработки, вернув клиенту *NULL*-указатель. Визуально процесс прерывания в модели Volcano представлен на рисунке 2.3 с порядковыми номерами 9 и 10.

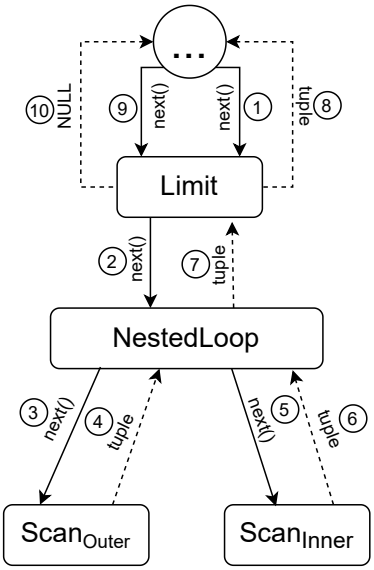


Рисунок 2.3 — Пример прерывания обработки запроса оператором Limit в модели Volcano.

В случае модели явных циклов, где основным процессом обработки руководит листовой оператор сканирования, располагающийся на произвольном уровне в дочернем поддереве оператора *Limit*, операция остановки выполняется при помощи интерпретации возвращаемого значения статуса. *Limit* должен сообщить руководящему оператору, которым в данном случае является $Scan_{outer}$, о том, что необходимо завершить выполнение запроса. Для этого он отправляет статус со значением, равным глубине оператора *NestedLoop*: $CD(NestedLoop) = 2$, которое должно интерпретироваться как команда прерывания внутреннего и внешнего циклов обработки с последующим вызовом функции *Limit::finalize()*.

Визуально этот процесс представлен на рисунке 2.4, где число в окружности является порядковым номером операции вызова функции. Оператор *NestedLoop* при получении от *Limit:consume()* статуса 2 должен протянуть

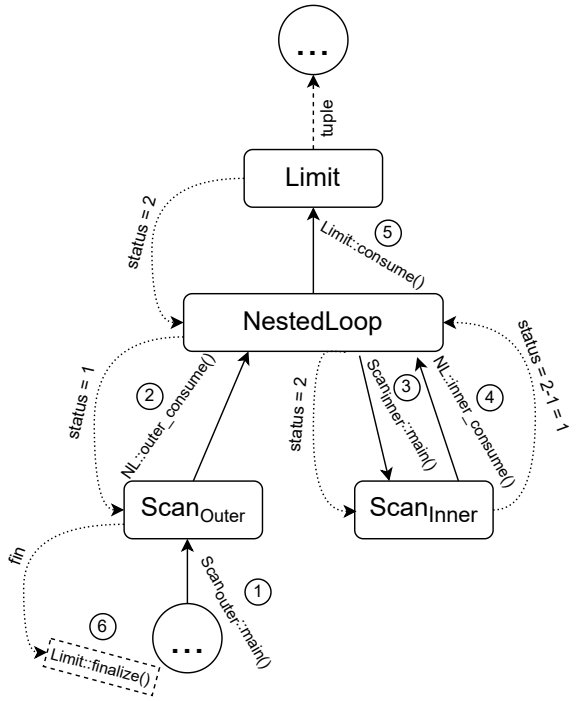


Рисунок 2.4 — Пример прерывания обработки запроса оператором Limit в модели явных циклов.

это значение через $Scan_{inner}$*::main()*, *NestedLoop::outer_consume()* оператору $Scan_{outer}$ для завершения выполнения запроса и вызова *Limit::finalize()*.

2.1.4 Декомпозиция алгоритмов операторов модели Volcano в модель явных циклов

В данной главе мы опишем процесс декомпозиции алгоритмов операторов итеративной Volcano-модели на функции интерфейса в модели явных циклов, которые генерирует код для данного запроса во время обхода в глубину дерева плана. Этот процесс позволяет трансформировать на лету операторы из Volcano-модели в модель явных циклов для выполнения запросов с использованием динамического компилятора.

В процессе декомпозиции алгоритмы операторов разбиваются на части: функцию *main()* для листовых узлов и функции *consume()* и *finalize()* для нелистовых узлов — с сохранением корректности за счёт генерации семантически эквивалентного внутреннего 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, выполняемые после обработки, зависят от типа оператора:
  - В общем случае, если часть алгоритма оператора, которую представляет данная функция *consume()*, не выполняет материализацию данных, то осуществляется вызов родительской функции *consume()* с последующим возвратом её статуса.
  - Иначе данные помещаются во временное хранилище и выполняется возврат соответствующего статуса.
- *finalize()* должна использоваться для выполнения части алгоритма оператора, отвечающего за постобработку аккумулированных данных:
  - В случае материализующего оператора функция *finalize()* должна содержать часть алгоритма, отвечающую за постобработку накопленных данных, с последующим циклом, где осуществляется производство новых кортежей. Для каждого созданного кортежа будет вызываться родительская функция *consume()*.
  - В случае конвейеризующего оператора функция *finalize()* выполняет связующую роль между операторами запроса: родительская функция *finalize()* вызывается из дочернего.
  - В случае бинарного оператора соединения функцию *finalize()* каждого дочернего узла целесообразно использовать для реализации заключающего этапа генерации данных. Это может быть необходимо при выполнении левого, правого или полного внешнего соединения.
- *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.

Опишем основные шаги, необходимые для выделения интерфейса:

1. Установить значение арности оператора, т.е. определить количество операндов, которые должны быть с ним связаны.
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 вместо функций *consume* и *finalize* будет функция *main*.

В общем случае количество функций *consume* и *finalize* равно общему числу дуг в дереве плана выполнения запроса.

Представление в СУБД плана запроса в виде двоичного дерева накладывает естественные ограничения на допустимое количество операндов у каждого оператора. Знание об арности оператора позволяет получить базовое представление о его возможностях и упростить процесс классификации. Так, нульарный оператор, являясь листовым узлом, всегда генерирует новые кортежи с использованием некоторого источника данных и для него генерируется только функция *main()*, которая будет использоваться для его запуска. Унарный оператор может быть либо конвейеризующим, либо материализующим. В первом случае будет выполняться кодогенерации функций интерфейса *consume()* и *finalize()*, где в функции *finalize()* будет вызываться родительская функция *finalize()*. Примером такого узла является оператор *Limit*. Во втором случае выполнится генерация функций *consume()* и *finalize()* с последующей их передачей дочернему узлу. Примерами унарных материализующих узлов являются операторы сортировки и материализации. Бинарные операторы зачастую представляют собой алгоритм соединения, который может быть материализующим (например HashJoin). В общем случае выполняется генерация двух экземпляров интерфейса *consume()* и *finalize()*, а затем устанавливается способ их взаимодействия. Так или иначе левый (внешний) оператор является ведущим, а правый (внутренний) ведомым. В случае n-арных операторов подход к кодогенерации схож с тем, что применяется для бинарных операторов.

Процесс декомпозиции алгоритмов реляционных операторов модели Volcano в модель явных циклов осуществляется с использованием информации, полученной в ходе этапов, описанных выше, 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 конечного пользователя. Для поддержки всего предоставляемого СУБД спектра функциональных возможностей динамическая компиляция должна по возможности входить в состав модулей основного ядра СУБД, тем самым накладывая ограничения на реализацию нового и поддержания старого функционала.

Далее мы опишем процесс декомпозиции алгоритмов операторов в модели Volcano на функции интерфейса в модели явных циклов для основных типов операторов СУБД.

Операторы сканирования

Операторы сканирования относятся к классу узлов, отвечающих за извлечение данных из файлов таблиц. В современных СУБД используется два режима сканирования таблиц: прямое, или последовательное, сканирование и сканирование через индекс.

При прямом сканировании всей таблицы каждый кортеж из таблицы считывается в последовательном порядке, определяемым типом сканирования, для поиска кортежей, удовлетворяющих заданным условиям. Прямое сканирование таблицы обычно является самым медленным методом сканирования из-за большого количества операций ввода-вывода, выполняемых над диском, и состоит из нескольких поисков, а также дорогостоящих операций по передаче данных с диска на память.

Сканирование через индекс используется с целью ускорения поиска данных. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск (например сбалансированного дерева).

На листинге 2.5 представлен псевдокод реализации функции *next()* оператора последовательного сканирования таблицы в модели Volcano, который выполняет чтение страниц данных таблицы и поиск следующего кортежа, удовлетворяющего некоторому условию.

Листинг 2.5: Алгоритм оператора последовательного сканирования таблицы в модели Volcano.

```
Scan::next()
    idx_page = state_page
    idx_tuple = state_tuple
    WHILE idx_page < T.length DO
        page = T[idx_page]
        WHILE idx_tuple < page.length DO
            tuple = page[idx_tuple]
            idx_tuple = idx_tuple + 1
            IF P(tuple):
                state_page = idx_page
                state_tuple = idx_tuple
                return tuple
        idx_page = idx_page + 1
        idx_tuple = 0
    return NULL
```

При каждом вызове функции *next()* восстанавливается значение позиции страницы данных в таблице (idx_{page}) и кортежа в этой странице (idx_{tuple}), зависящее от состояния оператора сканирования ($state_{page}$ и $state_{tuple}$), а по завершении вызова, перед возвратом результирующего кортежа, происходит обновление состояния оператора на основе изменённого значения текущей позиции в таблице. Именно это понимали под сохранением и загрузкой состояний, перечисляя проблемы модели Volcano в разделе 1.1.1.

Поскольку оператор последовательного сканирования является листовым узлом плана запроса, то его реализация в модели явных циклов будет находиться в единственной функции *main*. Так как в модели явных циклов основным процессом обработки руководит листовой оператор сканирования, используется механизм, описанный в разделе 2.1.3, для остановки выполнения с использованием статусов.

В представленном на листинге 2.6 алгоритме оператора последовательного сканирования в модели явных циклов реализованы циклы по страницам таблицы и кортежам в страницах, внутри которых выполняется проверка предиката $P(tuple)$ и вызов функции *consume()* родительского узла-оператора, тем самым избавляя от лишних инструкций копирования для сохранения состояния оператора сканирования.

Листинг 2.6: Алгоритм оператора последовательного сканирования таблицы декомпозированный на функции интерфейса в модели явных циклов.

```
Scan::main()
    FOR page in T:
        FOR tuple in page:
            IF P(tuple):
                status = Parent::consume()
            ELSE
                status = 0
            SWITCH(status):
                CASE 0: continue
                CASE 1: GOTO fin
                DEFAULT: return status - 1
fin:
    return Parent::finalize()
```

На листинге 2.7 представлен псевдокод реализации функции *next()* оператора сканирования таблицы через индекс в модели Volcano, который выполняет чтение данных по индексу и поиск следующего кортежа, удовлетворяющего некоторому условию.

Листинг 2.7: Алгоритм оператора сканирования через индекс в модели Volcano.

```
IndexScan::next()
    idx = state_index
    WHILE idx < index.length DO
        tuple = index[idx]
        idx = index.next
        IF P(tuple):
            state_index = idx
            return tuple
    return NULL
```

На листинге 2.8 представлен алгоритм оператора сканирования по индексу в модели явных циклов, аналогичный оператору последовательного сканирования, описанному выше.

Операторы соединения

Операция соединения JOIN предназначена для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор.

Листинг 2.8: Алгоритм оператора сканирования через индекс декомпозированного на функции интерфейса в модели явных циклов.

```
IndexScan::main()
   FOR tuple in index:
       IF P(tuple):
           status = Parent::consume()
       ELSE
           status = 0
       SWITCH(status):
           CASE 0: continue
           CASE 1: break
           DEFAULT: return status - 1
    return Parent::finalize()
```

Определение того, какие именно исходные кортежи войдут в результат и в каких сочетаниях, зависит от типа операции соединения и от явно заданного условия соединения. Условие соединения, то есть условие сопоставления кортежей исходных таблиц друг с другом, представляет собой логическое выражение (предикат).

В современных СУБД используются следующие три основных типа (стратегии) соединения:

1. *Соединение вложенным циклом (NestedLoop)*: правое отношение сканируется один раз для каждого кортежа, найденного в левом отношении.
2. *Соединение по хешу (HashJoin)*: сначала сканируется правое отношение и формируется хеш-таблица, ключ в которой вычисляется по атрибутам соединения. Затем сканируется левое отношение и по тем же атрибутам в каждом кортеже вычисляется ключ для поиска в этой хеш-таблице соответствующих кортежей справа.
3. *Соединение слиянием (MergeJoin)*: Каждое отношение сортируется по атрибутам соединения до начала соединения. Затем два отношения сканируются параллельно и соответствующие кортежи, объединяясь, формируют результирующие кортежи соединения.

Далее мы опишем процесс декомпозиции алгоритмов операторов соединения на функции интерфейса *consume()* и *finalize()*.

Соединение вложенным циклом. На листинге 2.9 представлен псевдокод алгоритма соединения вложенным циклом *NestedLoop* в модели Volcano. Ал-

Листинг 2.9: Алгоритм оператора соединения вложенным циклом в модели Volcano.

```
NestedLoop::next()
    WHILE true DO
        SWITCH(STATE_NestedLoop):
            CASE NEXT_OUTER:
                    tuple_outer = child_outer::next()
                    IF !tuple_outer:
                        return NULL
                    STATE_HashJoin = NEXT_INNER
            CASE NEXT_INNER:
                    tuple_inner = child_inner::next()
                    IF !tuple_inner:
                        % Сброс правого подплана
                        child_inner::close()
                        child_inner::open()
                        STATE_NestedLoop = NEXT_OUTER
                        continue
                    STATE_NestedLoop = MATCH
            CASE MATCH:
                    STATE_NestedLoop = NEXT_INNER
                    IF match(tuple_inner, tuple_outer):
                        return join(tuple_inner, tuple_outer)
```

горитм реализован в виде автомата состояний, где в цикле в зависимости от состояния оператора выполняются определённые действия.

Схема работы алгоритма следующая: на каждый кортеж, пришедший с внешней стороны ($NEXT\_OUTER$), извлекается кортеж с внутренней стороны ($NEXT\_INNER$) и выполняется проверка предикатов и соединение ($MATCH$). Когда все кортежи с внутренней стороны закончились, выполняется сброс состояния внутреннего поддерева с помощью последовательного вызова функций *close()* и *open()* оператора, тем самым заставляя его отдавать кортежи с самого начала для следующего внешнего кортежа.

На рисунке 2.5 проиллюстрирована работа алгоритма оператора *NestedLoop* на примере двух таблиц, которые представлены в виде лент *outer* и *inner*.

На листинге 2.10 представлен псевдокод алгоритма оператора соединения вложенным циклом *NestedLoop*, декомпозированого на функции интерфейса *consume()* и *finalize()*.

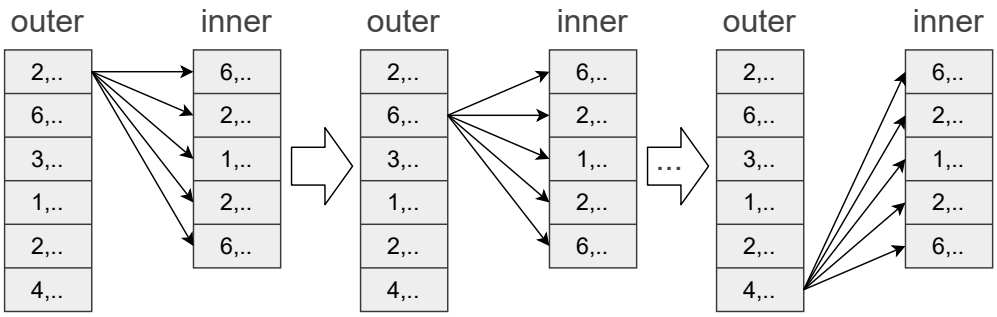


Рисунок 2.5 — Пример работы алгоритма соединения вложенным циклом.

Листинг 2.10: Алгоритм оператора соединения вложенным циклом, декомпозированного на функции интерфейса в модели явных циклов.

```
NestedLoop::inner_consume()
    IF match(tuple_inner, tuple_outer):
        join(tuple_inner, tuple_outer)
        return Parent::consume()
    return 0

NestedLoop::outer_consume()
    % Сброс состояния внутренней таблицы
    RESET(T_inner)
    return Child_inner::main()

NestedLoop::inner_finalize()
    return 0

NestedLoop::outer_finalize()
    return Parent::finalize()
```

NestedLoop является бинарным оператором, следовательно декомпозировать алгоритм необходимо на две функции (для внешнего и внутреннего поддеревьев) *consume()* и две *finalize()*. В функции *inner_consume()* выполняется проверка предиката соединения, соединение двух кортежей и возврат статуса от вызова функции *consume()* родительского узла-оператора. В *outer_consume()* вызывается функция *RESET()* для сброса состояния внутренней таблицы для извлечения заново всех кортежей.

Соединение хешированием. Алгоритм оператора соединения хешированием *HashJoin* в модели Volcano, так же как и у оператора *NestedLoop*, реализован в виде автомата состояний и работает следующим образом: на первом этапе (*BUILD*) строится хеш-таблица со всеми кортежами с внутренней (правой) сто-

роны, где ключ вычисляется по атрибутам соединения. На следующем этапе (*NEXT*) для каждого кортежа с внешней (левой) стороны вычисляется ключ соединения и по ее значению осуществляется поиск (*MATCH*) нужного кортежа в хеш-таблице внутренней стороны. В реальных системах хеш-таблица может не поместиться в оперативной памяти, поэтому применяют способы разбиения обрабатываемых кортежей на группы (batches) и построения хеш-таблиц для этих групп. Чтобы не усложнять псевдокод алгоритма соединения хешированием, далее будем рассматривать только случай, когда данные помещаются в память.

Псевдокод реализации алгоритма соединения хешированием в модели Volcano представлен на листинге 2.11. В большинстве современных СУБД для реализации соединения хешированием используется алгоритм гибридного соединения (hybrid hash join) [58].

Листинг 2.11: Алгоритм оператора соединение хешированием в модели Volcano.

```
HashJoin::next()
    WHILE true DO
        SWITCH(STATE_HashJoin):
            CASE BUILD:
                    WHILE tuple_inner = child_inner::next() DO
                        hashtable.put(tuple_inner)
                    STATE_HashJoin = NEXT
            CASE NEXT:
                    tuple_outer = child_outer::next()
                    IF !tuple_outer:
                        return NULL
                    STATE_HashJoin = MATCH
            CASE MATCH:
                    IF hashtable.match(hash(tuple_outer)):
                        return join(tuple_inner, tuple_outer)
                    STATE_HashJoin = NEXT
```

Так как оператор *HashJoin* является бинарным, то необходимо декомпозировать алгоритм на две функции *consume()* и две *finalize()*: для внешнего и для внутреннего узла.

На листинге 2.12 представлен алгоритм соединения хешированием, декомпозированный на функции интерфейса в модели явных циклов. Хронологически в работе *HashJoin* сначала полностью отрабатывает внутренняя сторона — то есть заполняется хеш-таблица — и только потом для каждого кортежа с внешней стороны ищутся подходящие пары, которые отправляются родителю.

Листинг 2.12: Алгоритм оператора соединения хешированием, декомпозированного на функции интерфейса в модели явных циклов.

```
HashJoin::inner_consume()
    hashtable.put(tuple_inner)
    return 0

HashJoin::outer_consume()
    FOR tuple_inner in hashtable.match(hash(tuple_outer)):
        join(tuple_inner, tuple_outer)
        status = Parent::consume()
        IF status > 0:
            return status - 1
    return 0

HashJoin::inner_finalize()
    return Child_outer::main()

HashJoin::outer_finalize()
    return Parent::finalize()
```

В функции *inner_consume()* выполняется заполнение хеш-таблицы, а в функции *outer_consume()* — поиск соответствующей пары в хеш-таблице, выполняется соединение и вызов функции *consume()* родительского узла-оператора. Также в функции *outer_consume()* выполняется обработка, корректировка и возврат статуса, полученного от родительского узла.

Соединение слиянием. Прежде, чем перейти к процессу анализа и декомпозиции алгоритма оператора соединения слиянием *MergeJoin* на функции интерфейса *consume()* и *finalize()*, хотелось бы отметить, что данный алгоритм очень органично вписывается в модель Volcano, но плохо подходит для модели явных циклов. Это связано с тем, что порядок работы оператора не может быть установлен заранее и определяется во время выполнения запроса в зависимости от входных данных. Следовательно одновременная конвейеризация данных из двух разных источников невозможна, и хотя бы один из циклов должен явно прерываться. Авторы исследования [59] рассматривают данную проблему в ходе анализа обеих моделей, а также утверждают, что указанный недостаток относится не только к оператору *MergeJoin*, но и к другим более сложным аналитическим операторам. Из-за этого при декомпозиции алгоритма оператора *MergeJoin* пришлось ввести дополнительную косвенность.

На рисунке 2.6 показана работа алгоритма оператора *MergeJoin* на примере двух таблиц, которые представлены в виде лент *outer* и *inner*.

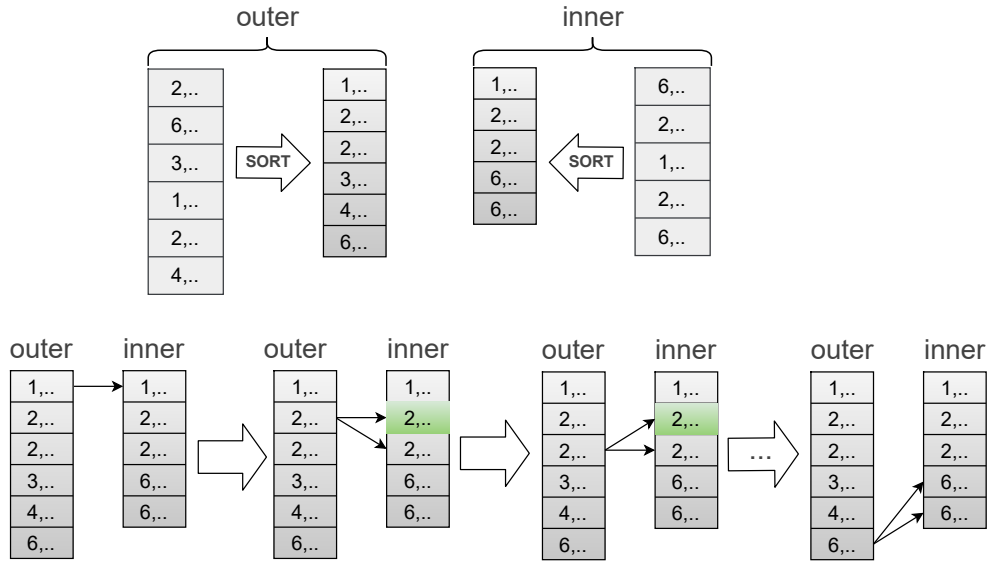


Рисунок 2.6 — Пример работы алгоритма соединения слиянием.

На листинге 2.13 представлен алгоритм оператора *MergeJoin* в модели Volcano. Оператор *MergeJoin* требует, чтобы входные данные были отсортированы по столбцам, участвующим в условии соединения. Это требование ограничивает пространство его применения, но в тоже время делает его крайне эффективным в определённых случаях.

В начале работы алгоритма выполняется сортировка входных таблиц. Далее над кортежами выполняется операция сравнения, результат которой определяет дальнейшее движение. Если *outer* кортеж меньше *inner* кортежа, то *outer* сдвигается на одну ячейку вперёд, если *inner* кортеж меньше *outer*, то сдвигается *inner*. Если кортежи из *outer* и *inner* равны, то выполняется соединение, а на ленте *inner* делается пометка и запоминается текущий *inner*-кортеж и его позиция. После соединения осуществляется сдвиг *inner* и выполняется проверка возможности повторного соединения. Сдвиг *inner* продолжается до тех пор, пока результат проверки положительный и соединение возможно. После остановки *inner* выполняется сдвиг на ленте *outer* и осуществляется сравнение текущего кортежа *outer* с помеченным кортежем из *inner*. Если условие выполняется, то совершается соединение кортежа *outer* с помеченным кортежем из *inner*, а *inner* совершает скачок назад на последнюю пометку и продолжает движение оттуда. В противном случае движение продолжается в обычном режиме.

Так как в модели явных циклов основным циклом работы управляет некоторый листовой оператор, а поток кортежей генерируется без дополнительных

Листинг 2.13: Алгоритм оператора соединение слиянием в модели Volcano.

```
MergeJoin::next()
  WHILE true DO
    SWITCH(STATE_MergeJoin):
      CASE INIT:
          tuple_outer = child_outer::next()
          IF !tuple_outer:
            STATE_MergeJoin = END
            continue
          tuple_inner = child_inner::next()
          IF !tuple_inner:
            STATE_MergeJoin = END
            continue
      CASE NEXT_OUTER:
          WHILE tuple_outer < tuple_inner DO
            tuple_outer = child_outer::next()
            IF !tuple_outer:
              STATE_MergeJoin = END
              continue
      CASE NEXT_INNER:
          WHILE tuple_outer > tuple_inner DO
            tuple_inner = child_inner::next()
            IF !tuple_inner:
              STATE_MergeJoin = END
              continue
      CASE MARK_INNER:
          mark_inner = tuple_inner
      CASE MATCH:
          IF tuple_outer = tuple_inner:
            tuple = join(tuple_outer, tuple_inner)
            tuple_inner = child_inner::next()
            IF !tuple_inner:
              STATE_MergeJoin = END
              continue
            ELSE
              STATE_MergeJoin = MATCH
            return tuple
          tuple_outer = child_outer::next()
          IF !tuple_outer:
            STATE_MergeJoin = END
            continue
      CASE MATCH_MARK:
          IF tuple_outer = mark_inner:
            tuple_inner = mark_inner
            STATE_MergeJoin = MATCH
          ELSE
            STATE_MergeJoin = NEXT_OUTER
          continue
      CASE END:
          return NULL
```

Листинг 2.14: Алгоритм оператора соединение слиянием декомпозированный на функции *inner_consume()* и *inner_finalize()* в модели явных циклов.

```
MergeJoin::inner_consume()
    WHILE true DO
        SWITCH(STATE):
            CASE TEST_INNER:
                    IF tuple_outer > tuple_inner:
                        STATE = TEST_INNER
                        return 0
                    IF tuple_outer < tuple_inner:
                        STATE = NEXT_OUTER
                        inner_break = TRUE
                        return CD(Child_inner) + 1
                    mark_inner = tuple_inner
                    STATE = MATCH
            CASE MATCH:
                    STATE = NEXT_INNER
                    join(tuple_inner, tuple_outer)
                    return Parent::consume()
            CASE NEXT_INNER:
                    IF tuple_outer = tuple_inner:
                        STATE = MATCH
                        break
                    STATE = NEXT_OUTER
                    inner_break = TRUE
                    return CD(Child_inner) + 1

MergeJoin::inner_finalize()
     STATE = NEXT_OUTER
    return 0
```

вызовов, то необходимо декомпозировать алгоритм оператора *MergeJoin* как минимум на две функции *consume()*, каждая из которых будет вызываться соответствующим дочерним узлом после генерации нового кортежа.

На листинге 2.14 представлен алгоритм оператора *MergeJoin*, декомпозированого на функции интерфейса *inner_consume()* и *inner_finalize()* для внутреннего цикла.

На представленном на листинге 2.15 алгоритме оператора *MergeJoin*, декомпозированого на функции интерфейса *outer_consume()* и *outer_finalize()* для внешнего цикла, функция *outer_consume()*, представляющая внешний цикл, для запуска внутреннего цикла выполняет соответствующий вызов сгенерированной для правого поддерева функции *main()*, где впоследствии

Листинг 2.15: Алгоритм оператора соединение слиянием декомпозированный на функции *outer_consume()* и *outer_finalize()* в модели явных циклов.

```
MergeJoin::outer_consume()
    WHILE true DO
        SWITCH(STATE):
            CASE INIT_OUTER:
                    STATE = TEST_INNER
            CASE NEXT_INNER:
                    inner_break = FALSE
                    status = Child_inner::main()
                    IF inner_break:
                        return 0
                    return status
            CASE TEST_OUTER:
                    IF tuple_outer > tuple_inner:
                        STATE = NEXT_INNER
                        break
                    IF tuple_outer < tuple_inner:
                        STATE = TEST_OUTER
                        return 0
                    mark_inner = tuple_inner
                    STATE = MATCH
            CASE MATCH:
                    STATE = NEXT_INNER
                    join(tuple_inner, tuple_outer)
                    status = Parent::consume()
                    IF !status:
                        break
                    return status
            CASE NEXT_OUTER:
                    IF tuple_outer = mark_inner:
                        tuple_inner = mark_inner
                        STATE = NEXT_INNER
                        join(tuple_inner, tuple_outer)
                        status = Parent::consume()
                        IF !status:
                            STATE = NEXT_INNER
                            break
                        return status
                    STATE = TEST_OUTER

MergeJoin::outer_finalize()
    return Parent::finalize()
```

вызывается функция *inner_consume()*, которая может выполнять соединение, перемещаться по ленте *inner* с использованием статус-кодов и прерывать внутренний цикл. Для продолжения выполнения внешнего цикла в случае прерывания внутреннего была введена константа *inner_break*, которая устанавливается в функции *inner_consume()* в случае, когда *inner*-кортеж больше *outer*-кортежа. Это позволяет отличить прерывание, вызванное родительским оператором (например при наличие оператора *Limit* над *MergeJoin*), от прерывания внутреннего цикла *MergeJoin*.

Другие операторы СУБД

Далее мы опишем принцип декомпозиции на функции интерфейса в модели явных циклов алгоритма оператора-ограничителя результата *Limit*, оператора агрегации по хешу *HashAgg* и одного из материализующих операторов — оператора сортировки *Sort*.

Оператор-ограничитель результата служит для ограничения количества возвращаемых кортежей на основе заданного параметра. Псевдокод алгоритма оператора *Limit* представлен на листинге 2.16.

Листинг 2.16: Алгоритм оператора ограничитель результата в модели Volcano.

```
Limit::next()
    IF limit_index ⩾ N:
        return NULL
    limit_index++
    return child::next()
```

На листинге 2.17 представлен алгоритм оператора *Limit*, декомпозированного на функции интерфейса в модели явных циклов: *consume()* и *finalize()*. В функции *Limit::consume()* выполняется инкремент индекса-ограничителя результата и сравнение с заданным параметром с последующим возвратом соответствующего статуса. Функция *Limit::finalize()* выполняет связующую роль посредством вызова функции *finalize()* от родительского оператора.

Листинг 2.17: Алгоритм оператора-ограничителя результата, декомпозированного на функции интерфейса в модели явных циклов.

```
Limit::consume()
    IF limit_index ⩾ N:
        % CD – константа времени компиляции
        return CD(Child)
    limit_index++
    return Parent::consume()

Limit::finalize()
    return Parent::finalize()
```

Для завершения обработки запроса оператору *Limit* необходимо вернуть значение статуса, равное количеству вложенных циклов в своём потоке. Это значение вычисляется заранее, на этапе генерации кода. Используется разработанный механизм прерывания потока обработки на основе передачи статусов, где возвращаемое значение статуса обрабатывается вышестоящим оператором для принятия решения о продолжении или завершении потока обработки данных.

Оператор агрегации служит для группировки кортежей, используя агрегатные функции COUNT, SUM, MAX, MIN и AVG. В современных СУБД используются несколько стратегий реализации оператора агрегации: агрегация на основе сортировки и агрегации на основе хеширования. Мы опишем оператор агрегации по хешу, который в основном применяется при использовании GROUP BY совместно с агрегатными функциями. На листинге 2.18 представлен псевдокод алгоритм оператора агрегации по хешу HashAgg в модели Volcano.

Листинг 2.18: Алгоритм оператора агрегации по хешу в модели Volcano.

```
HashAgg::next()
    IF !done:
        WHILE tuple = child::next() DO
            IF hashtable[tuple_A]:
                hashtable[tuple_A].put(tuple)
            ELSE
                hashtable[tuple_A].create(tuple)
        FOR tuple_A in hashtable:
            hashtable[tuple_A].aggregate()
        done = true
    return hashtable.get()
```

Алгоритм оператора агрегации по хешу работает следующим образом: каждый входящий кортеж помещается в хеш-таблицу в соответствующую корзину, используя атрибут группировки в качестве ключа; после обработки всех кортежей выполняется сканирование хеш-таблицы и возврат по одному кортежу для каждого значения ключа с выполнением операции агрегации.

На листинге 2.19 представлен алгоритм оператора агрегации по хешу HashAgg, декомпозированный на функции интерфейса в модели явных циклов.

Листинг 2.19: Алгоритм оператора агрегации по хешу, декомпозированный на функции интерфейса в модели явных циклов.

```
HashAgg::consume()
    IF hashtable[tuple_A]
        hashtable[tuple_A].put(tuple)
    ELSE
        hashtable[tuple_A].create(tuple)
    return 0

HashAgg::finalize()
    FOR tuple_A in hashtable:
        aggregate(tuple_A)
        status = Parent::consume()
        SWITCH(status):
            CASE 0: continue
            CASE 1: break
            DEFAULT: return status - 1
    return Parent::finalize()
```

В функции *HashAgg::consume()* заключена логика по вставке кортежа в хеш-таблицу, а в функции *HashAgg::finalize()* осуществляется агрегация аккумулированных в хеш-таблице данных с последующим итерированием.

Оператор сортировки возвращает набор кортежей, отсортированных по значениям одного или более столбцов. Сортировка используется во многих операторах, например при выполнении соединения слиянием, агрегации с сортировкой, устранении дубликатов и т.д. Реализация алгоритма оператора сортировки в модели Volcano представлена на листинге 2.20.

На первом этапе выполняется загрузка кортежей во временный буфер в памяти, далее выполняется сортировка кортежей в этом буфере и возврат кортежей в отсортированном порядке при последующих вызовах *next()*.

Листинг 2.20: Алгоритм оператора сортировки в модели Volcano.

```
Sort::next()
    IF !done:
        WHILE tuple = child::next() DO
            sort_buffer.put(tuple)
        sort_buffer.sort()
        done = TRUE
    return sort_buffer.get()
```

Непосредственно используемый алгоритм сортировки зависит от реализации оператора сортировки в конкретной СУБД.

На листинге 2.21 представлен алгоритм оператора сортировки Sort, декомпозированный на функции интерфейса в модели явных циклов. В функции *Sort::consume()* выполняется вставка кортежа дочернего узла во временный буфер, в функции *Sort::finalize()* реализована непосредственно сортировка и процесс итерации по упорядоченному результату.

Листинг 2.21: Алгоритм оператора сортировки, декомпозированный на функции интерфейса в модели явных циклов.

```
Sort::consume()
    sort_buffer.put(t)
    return 0

Sort::finalize()
    sort_buffer.sort()
    FOR tuple in sort_buffer:
        status = Parent::consume()
        SWITCH(status):
            CASE 0: continue
            CASE 1: break
            DEFAULT: return status - 1
    return Parent::finalize()
```

2.2 Динамическая компиляция выражений в SQL-запросах

Для вычисления выражений, используемых в SQL-запросах, СУБД выполняет интерпретацию дерева выражений, построенного на этапе планирования

запроса, 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.

Примеры типов выражений, которые могут храниться в узлах дерева:

– Сравнения (=, <, >, !=)
– Конъюнкция (AND), дизъюнкция (OR)
– Арифметические операции (+, –, *, /, %)
– Константы и значения параметров
– Ссылки на атрибуты кортежей

Для вычисления результата выражения E 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. Неявные вызовы не позволяют выполнять оптимизацию встраивания функций (inlining), тем самым ограничивая возможности компилятора для дальнейшей оптимизации, например удаление общих подвыражений, распространение констант и тд. Для вычисления самих операций вызываются встроенные функции СУБД.

На рисунке 2.7 представлен пример дерева выражения для фильтра, где функция $Att()$ вызывается для чтения атрибутов, используемых в выражении.

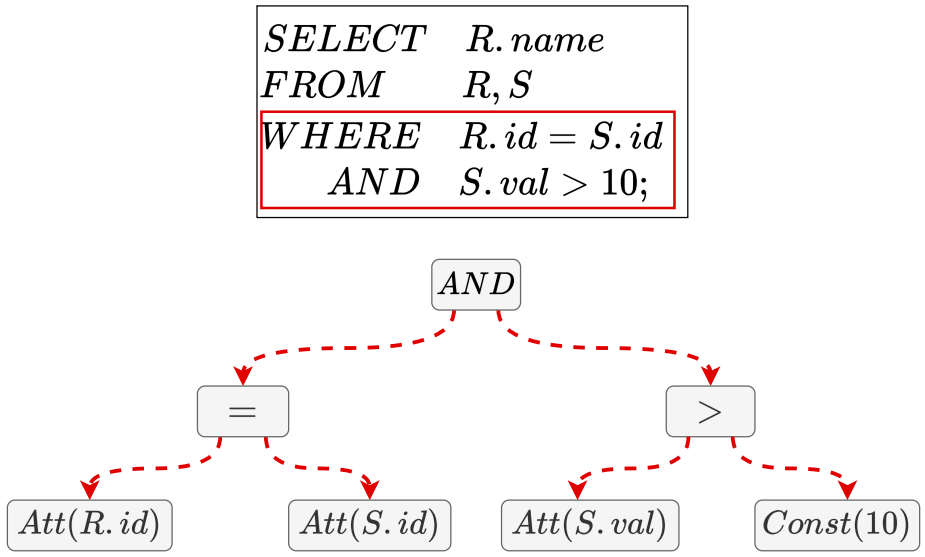


Рисунок 2.7 — Пример дерева выражения.

Поскольку во время выполнения доступна информация о вызываемых функциях и операциях, можно использовать кодогенерацию для замены неявных вызовов функций на явные, которые в дальнейшем могут быть встроены. Для этого предлагается использовать метод динамической компиляции выражений и предикатов, используемых в запросе.

Динамическая компиляция выражений выполняется путём рекурсивного обхода дерева выражений в обратном порядке и генерацией соответствующего кода для каждой функции или операции во внутреннем представлении. Таким образом код для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без расходов на неявный вызов функций.

Для нивелирования накладных расходов связанных с вызовом встроенных функций СУБД для вычисления операций при выполнении выражения предлагается использовать оптимизирующее компиляторное преобразование по открытой вставке функций, вставляющее код функции на место его вызова в тело вызывающей функции. Для получения кода встроенных функций СУБД во внутреннем представлении предлагается использовать метод предварительной компиляции. 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 во внутреннее представление встроенных функций СУБД позволит использовать в динамическом компиляторе одну и ту же реализацию встроенных функций СУБД совместно с интерпретатором выражений.

На рисунке 2.8 представлен пример сгенерированного кода для выражения в фильтре из примера на рисунке 2.7:

$E \equiv (Att(R.id) == Att(S.id))\ AND\ (Att(S.val) \geqslant Const(10))$.

Здесь функции EQ и GT являются встроенными функциями СУБД и возвращают истину или ложь в зависимости от значений атрибутов. Как можно заметить, после применения оптимизации открытой вставки встроенных функций СУБД в тело выражения в коде отсутствуют дополнительные вызовы функций.

К преимуществам подхода динамической компиляции выражений можно отнести следующее:

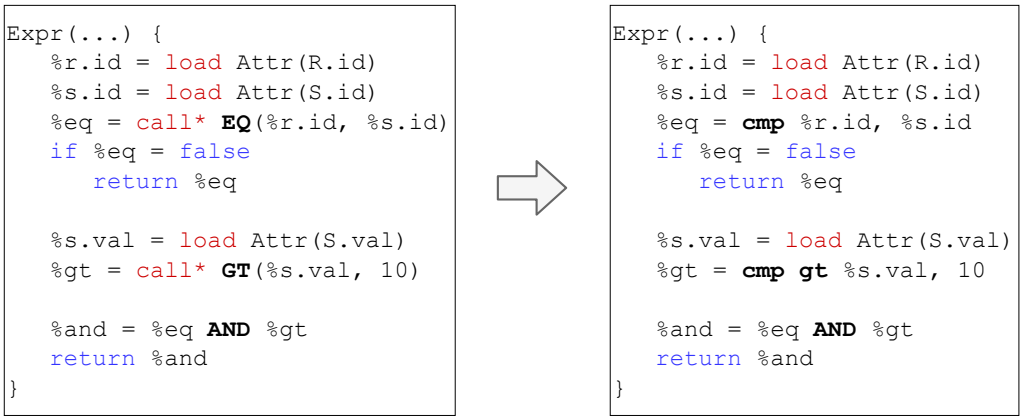


Рисунок 2.8 — Слева — сгенерированный код выражения динамическим компилятором, справа — кода после применения открытой вставки встроенных функций СУБД.

– Каждое выражение E рассматривается как единое целое, которое скомпилировано в отдельную функцию, и таким образом, чтобы вычислить результат выражения E, СУБД вызовет одну функцию.
– Модуль выполнения запросов СУБД может оставаться неизменным: выражения компилируются после этапа планирования и непосредственно перед началом выполнения плана запроса.
– Динамической компилятор и интерпретатор запросов могут вместе сосуществовать, то есть оба подхода могут быть использованы для вычисления выражения в запросе.
– Упрощённая поддержка благодаря использованию оптимизации открытой вставки и предкомпиляции встроенных функций, поскольку отпадает необходимость в ручной реализации каждой встроенной функции и отслеживании изменений в коде СУБД.

2.3 Эвристики стратегии выполнения запроса

В динамическом компиляторе запросов были разработаны эвристики стратегии выполнения запроса с учётом стоимости интерпретации и выполнения скомпилированного кода запроса. Эвристики описаны в виде конечного автомата состояний, представленный на рисунке 2.9, который используется для обработки каждого входящего запроса. Пунктирные линии, соединяю-

щие некоторые состояния, означают, что переход происходит только в случае ошибки. Состояния с пунктирными линиями указывают, что кэшированный запрос может продолжить выполнение из этих состояний после того, как его метаинформация была загружена из подсистемы кэширования планов. Под метаинформацией мы понимаем сохранённый план запроса, скомпилированный код для этого запроса и сохранённое состояние конечного автомата для этого запроса.

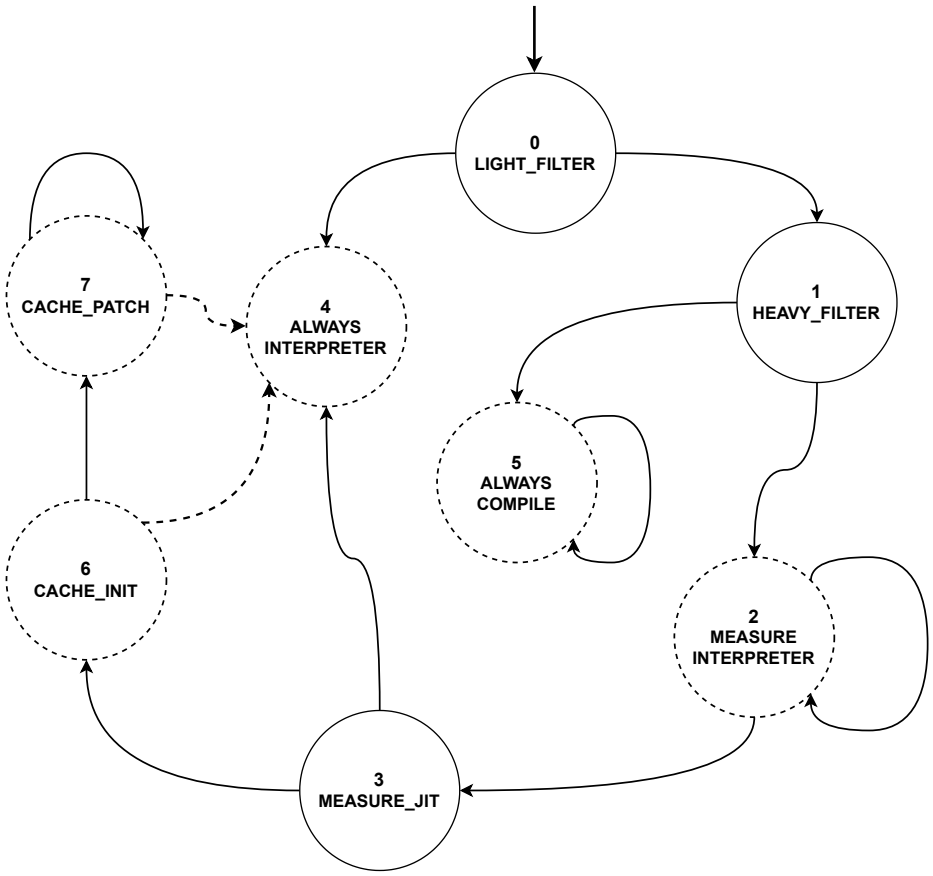


Рисунок 2.9 — Конечный автомат состояний эвристик стратегии выполнения запроса в динамическом компиляторе.

Чтобы минимизировать накладные расходы на выполнение, метаинформация загружается только в состоянии 1 — `HEAVY_FILTER`. Для достижения этого состояния запрос должен пройти два эвристических фильтра по стоимости, которые определяют, какие запросы должны выполняться интерпретатором, быть более тщательно профилированы или скомпилированы и выполнены сразу. Стоимость запроса $Cost(P)$, где P — это план запроса, рассчитывается оптимизатором СУБД и представляет собой относительную цену для СУБД на выполнения этого запроса. Состояния 0 и 1 представляют простой предикат $C_{low} < Cost(P) < C_{high}$, где C_{low} и C_{high} — это нижняя и верхняя оценка

для принятия решения об интерпретации, анализе или компиляции, который означает, что быстрые (короткие) запросы должны интерпретироваться, долгие (длинные) запросы должны быть скомпилированы сразу, а остальные должны быть проанализированы более тщательно для принятия окончательного решения. Если стоимость запроса больше чем верхняя оценка: $Cost(P) > C_{high}$, то запрос переходит в состояние 5 — `ALWAYS_COMPILE`, где сразу выполняется компиляция этого запроса.

Если же запрос удовлетворяет предикату $C_{low} < Cost(P) < C_{high}$ 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 мы сравниваем 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 $K\%$. Этим оценивается насколько динамический компилятор запросов улучшает производительность. Значение K задаётся пользователем. Если улучшение не достаточно, то при последующих выполнения этот запрос будет интерпретироваться (состояние 4 — `ALWAYS_INTERPRETER`).

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`. Также сохраняется состояние запроса для быстрого восстановления и продолжения выполнения из этого состояния. Более подробно о методе кэширования кода запроса, а также об алгоритме кэширования описано в следующем разделе.

2.3.1 Кэширования кода, сгенерированного динамическим компилятором запросов

Методы динамической компиляции предполагают замену в общем времени обработки запроса времени интерпретации $T_I(N)$ на суммарное время компиляции и выполнения скомпилированного кода $T_C + T_E(N)$, где N — размер

данных, обрабатываемых запросом, T_C — время компиляции. Учёт структуры и константных данных запроса и проводимые во время компиляции оптимизации делают скомпилированный код более эффективным: $T_E(N) < T_I(N)$, но чтобы динамическая компиляция имела смысл, необходимо, чтобы $T_C + T_E(N) < T_I(N)$, то есть чтобы время, затрачиваемое на интерпретацию запроса, превосходило время, затрачиваемое на компиляцию и выполнение оптимизированного кода. Данное требование может быть удовлетворено только в том случае, когда объем обрабатываемых запросом данных достаточно велик. Таким образом, только аналитические запросы типа OLAP [60], выполняющиеся на большом объёме данных, могут нивелировать время, затрачиваемое на компиляцию, и позволяют получить прирост производительности. В случае OLTP [61] запросов, где в основном обрабатывается небольшой объем данных, а время интерпретации может исчисляться микросекундами, метод динамической компиляции может оказаться неприемлемым по причине долгой оптимизации и компиляции динамически сгенерированного кода.

Проблема может быть решена путём *кэширования*, то есть сохранения и переиспользования сгенерированного динамическим компилятором запросов кода. Однако, как уже писали выше, не все запросы имеет смысл компилировать, так как, например, у быстрых запросов время затраченное на кодогенерацию может значительно превышать время интерпретации запроса, поэтому нужны гарантии, что накладные расходы на компиляцию окупятся, и не для всех запросов нужно сохранять код из-за дополнительной косвенности по обслуживанию кэшированного кода. Для этого разработаны эвристики стратегии выполнения запроса (см. главу 2.3).

Определим структуру кэша: она будет хранить динамически скомпилированный код запроса совместно с планом запроса и информацией о схеме используемых в этом запросе таблиц, типе атрибутов и т.д. Последняя информация необходима для корректного сопоставления плана запроса и сохранённого кода в случае изменений схемы таблиц или атрибутов. Размер кэша задаётся пользователем и зависит от имеющейся конфигурации по памяти.

Зачастую сохранённый код запроса может содержать абсолютные адреса структур данных, используемых в таблице, например, адрес дескриптора файла таблицы и т.д., которые могут изменяться в динамической памяти после каждого выполнения. Соответственно после загрузки кода запроса необходимо его подготовить к выполнению путём патчинга старых адресов, то есть замены ис-

пользуемых в коде адресов на новые на основе текущего контекста выполнения. Данный этап добавляет дополнительные накладные расходы по обслуживанию кэшированного кода запроса.

Общая схема метода кэширования динамически скомпилированного кода запроса следующая:

- После получения плана запроса выполняется поиск в структуре кэша кода для этого запроса.
- Если сохранённый код запроса был найден, то выполняется сопоставление сохранённой информации о схеме таблиц и атрибутов, используемых в запросе, далее выполняется загрузка кода и выполнение, тем самым экономя время затрачиваемое на компиляцию плана 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.

Ниже проиллюстрирована подробная процедура повторного использования ранее сохранённого кода плана запроса:

```
Cache(plan):
    entry = LFU.Cache → search(plan)
    if entry ≠ NULL then
        code ← PATCH(entry.code, context)
    else
        code ← Compile(plan)
        entry ← {plan, code}
        LFU.Cache → append(entry)
    end
    return code
```

Здесь $LFU.Cache$ — это структура кэша, выполняющая управление кэшем (поиск, загрузку/сохранение, и удаление); $PATCH(code, context)$ — функция выполняющая патчинг используемых в сохранённом коде запроса *code* абсолютных адресов, где *context* — это текущий контекст выполнения, содержащий новые адреса. Функция $Compile(plan)$ принимает на вход план запроса и выполняет генерацию и компиляцию с последующим возвратом динамически скомпилированного кода запроса.

Алгоритм кэширования

Для управления структурой кэша запросов был выбран алгоритма кэширования (вытеснения кэша) LFU (Least-Frequently Used, Наименее часто используемый). Мы использовали реализацию алгоритма LFU на основе двунаправленного списка [62], но модифицировали ее под наши цели по сохранению кода запросов. Реализация предлагаемого алгоритм кэширования на основе LFU имеет сложность выполнения $O(1)$ для каждой операций (вставка, поиск и удаление), которые могут выполняться в кэше LFU. Это достигается за счёт поддержки двух двунаправленных списков: один для частоты доступа и один для элементов с одинаковой частотой доступа. Для доступа к элементам по ключу используется хеш-таблица. Двунаправленный список, использующийся для связывания вместе узлов с одинаковой частотой доступа, будем называть *списком частот*. Набор узлов, имеющих одинаковую частоту доступа, представляет собой двунаправленный список таких узлов. Назовём этот список *списком кэшей или ключей*. Каждый узел в списке кэшей имеет указатель на свой родительский узел в списке частот.

Первоначально LFU кэш начинается как пустая хеш-таблица и пустой список частот. Когда *добавляется* первый элемент, создаётся единственный элемент в хеш-таблице, который указывает на этот новый элемент (по его ключу), и новый узел добавляется в список частот. Добавляется новый узел в первый список кэшей. Этот узел содержит указатель на узел в списке частот, членом которого он является. За счёт использования двунаправленного списка сложность вставки элементов во время выполнения составляет $O(1)$.

Когда к этому элементу обращаются ещё раз, выполняется *поиск* в списке частот узла и запрашивается значение его следующего элемента. Если следующего узла в списке частот не существует, а в текущем списке кэшей это не единственный элемент, то создаётся новый узел в списке частот и этот элемент вставляется в правильное место. Узел удаляется из текущего списка кэшей и вставляется в новый список кэшей списка частот. Указатель обновляется, чтобы указывать на новый узел в списке частот. Если же следующего узла в списке частот не существует, но при этом в текущем списке кэшей это единственный элемент, то новый узел в списке частот не создаётся, а возвращается текущий

элемент (это одна из модификаций реализации алгоритма). Таким образом, сложность доступа к элементам во время выполнения составляет $O(1)$.

Когда необходимо *удалить* элемент из кэша при его заполнении, то выбирается и удаляется крайний левый узел из списка частот, содержащий давно не используемые элементы и элементы с наименьшей частотой доступа. Ссылки на элементы из хеш-таблицы также удаляются. Следовательно, сложность удаления элементов во время выполнения составляет $O(1)$. В реализации операции удаления элемента и заключается основная модификация реализации алгоритма LFU, то есть в случае, когда кэш заполнен, то удаляется целый список элементов, а не только один (к которому последний раз обращались), тем самым освобождая больше места в кэше.

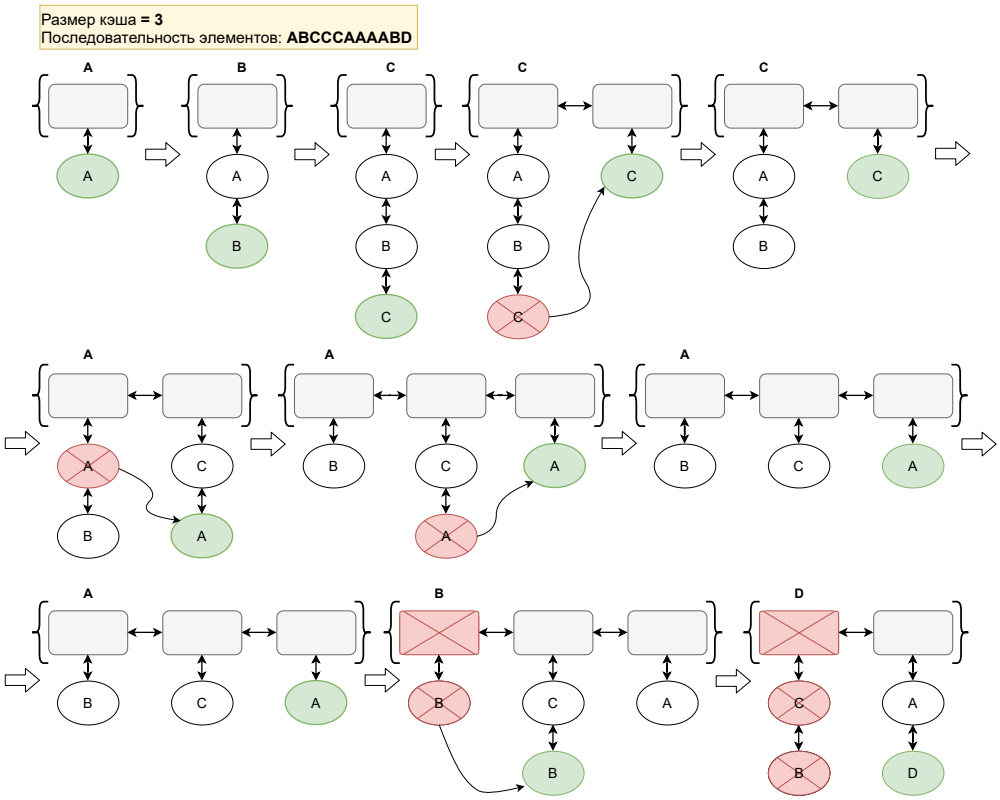


Рисунок 2.10 — Пример работы модифицированной реализации алгоритма LFU.

На рисунке 2.10 продемонстрирована работа модифицированной реализации алгоритма кэширования LFU на примере последовательности элементов (ключей) $ABCCCAAAABD$ в случае, когда размер кэша равен 3. Прямоугольники связаны друг с другом и обозначают узлы списка частот, а круги (A, B, C, D) также связаны друг с другом и обозначают узлы или блоки кэша списка кэшей. Таким образом, каждый раз, когда обращаемся к блоку в списке кэшей (например, A на рисунке 2.10), этот блок будет перемещаться на следующий узел в списке частот; если кэш заполнен, и мы пытаемся добавить новый блок

в кэш, то мы просто удаляем самый левый (головной) узел из списка частот с его списком кэшей. Таким образом, в приведённом выше примере в конце в кэше останутся только недавно использованный (D) и наиболее часто используемый (A) ключ. Такая реализация алгоритма кэширования LFU хорошо подходит как для часто выполняемых, так и для недавно выполненных запросов.

2.4 Выводы

Предложенный метод динамической компиляции запросов с трансформацией операторов плана запроса в модель явных циклов применим к использующейся в большинстве современных СУБД Volcano-модели. Во время обхода дерева плана запроса выполняется генерация в промежуточном представлении функций интерфейса операторов в модели явных циклов. После оптимизаций код запроса представляется в виде последовательности вложенных циклов. Описанный принцип декомпозиции алгоритмов операторов модели Volcano на функции интерфейса в модель явных циклов позволяет реализовать генераторы промежуточного представления на основе имеющейся реализации алгоритма соответствующего оператора в конкретной СУБД. Генерация кода в виде отдельных функций интерфейса для операторов плана запроса избавляет от основных недостатков присущих динамическим компиляторам запросов, описанных в разделе 1.2.2: отсутствует раздувание кода во время генерации и сгенерированные функции могут быть проверены и отлажены с использованием существующих отладчиков и визуализаторов.

Разработанный метод динамической компиляции выражений избавляет от накладных расходов на интерпретацию выражений, используемых в запросах. Для нивелирования накладных расходов связанных с вызовом встроенных функций СУБД для вычисления операций при выполнении выражения применяется оптимизирующее компиляторное преобразование по открытой вставке функций. Метод предварительной компиляции позволяет получить код встроенных функций СУБД в промежуточном представлении. Применение открытой вставки функций, а также метода предкомпиляции позволяет использовать в динамическом компиляторе одну и ту же реализацию встроенных функций СУБД совместно с интерпретатором выражений.

Разработанные эвристики стратегии выполнения запроса устанавливают стратегии, когда интерпретация выполняется для очень простых запросов, кэширование используется для средних запросов, а долго выполняющиеся запросы всегда компилируются. Разработанный метод кэширования позволяет повторно использовать сгенерированный динамическим компилятором код для одинаковых запросов, тем самым избавляя от расходов на компиляцию и оптимизацию.

Глава 3. Реализация динамического компилятора SQL-запросов

Предлагаемый метод динамической компиляции запросов реализован в виде программного расширения к СУБД с открытым исходным кодом PostgreSQL [32] с использованием компиляторной инфраструктуры LLVM [28].

В последующих разделах опишем устройство интерпретатора запросов СУБД PostgreSQL (раздел 3.1), компиляторную инфраструктуру LLVM (раздел 3.2) и архитектуру реализованного динамического компилятора запросов для СУБД PostgreSQL (раздел 3.3).

3.1 Интерпретатор запросов СУБД PostgreSQL

PostgreSQL [32] — свободная объектно-реляционная система управления базами данных (СУБД), которая базируется на языке SQL и поддерживает многие из возможностей стандарта SQL:2011.

В СУБД PostgreSQL для преобразования SQL-запроса из текстового представления в дерево плана запрос проходит несколько этапов обработки. На первом этапе выполняется синтаксический анализ запроса, который строит дерево синтаксического анализа. После этого анализатор выполняет семантический анализ дерева, полученного от этапа синтаксического анализа и создаёт новое представление, называемое деревом запроса. Планировщик (оптимизатор) использует дерево запросов для создания наиболее эффективного плана запроса. В конце исполнитель выполняет интерпретацию дерева плана запроса.

Для интерпретации запросов в СУБД PostgreSQL используется, описанная в разделе 1.1.1, модель Volcano в рамках которой каждый алгебраический оператор преобразовывает входные данные в выходной поток кортежей, который управляется при помощи функции *next()*. Интерфейса оператора в PostgreSQL состоит из функций *ExecInitNode*, *ExecEndNode* и *ExecProcNode*, соответствующих методам *open()*, *close()* и *next()* модели Volcano. Так как PostgreSQL написан на языке C, в котором нет виртуальных функций, за диспетчеризацию запросов по вызову нужной функции *next()* отвечает функция *ExecProcNode* с параметром в виде указателя на структуру *PlanState*

и с условным оператором *switch(PlanState.type)* внутри. Все узлы-операторы представляют собой структуру с различными полями, где первым всегда располагается структура типа *PlanState*, содержащий информацию о типе узла и описывающий состояние оператора.

Функции *ExecProcNode* имеет следующую сигнатуру:

```
TupleTableSlot *ExecProcNode(PlanState *node)
```

где *node* — указатель на базовую структуру *PlanState*, а *TupleTableSlot* — тип представления кортежа.

Таким образом, для того, чтобы вызвать нужную реализацию функции *next()* для соответствующего потомка, достаточно передать указатель на его *PlanState* в функцию *ExecProcNode*, где в последствии он будет приведён к нужному типу и передан в заданную реализацию функции *next()*. Так в PostgreSQL имитируется полиморфизм подтипов для реализации виртуальных функций.

Рассмотрим пример из документации PostgreSQL, демонстрирующий работу интерпретатора запросов СУБД. Рассматривается следующий запрос:

Листинг 3.1: Пример SQL-запроса.

```
SELECT DEPT.no_emps, EMP.age
FROM DEPT, EMP
WHERE EMP.name = DEMP.manager
      AND DEPT.name = "shoe";
```

Данный запрос выдаст возраст менеджера обувного отдела и количество сотрудников в этом отделе. Возможный план исполнения этого запроса представлен на рисунке 3.1.

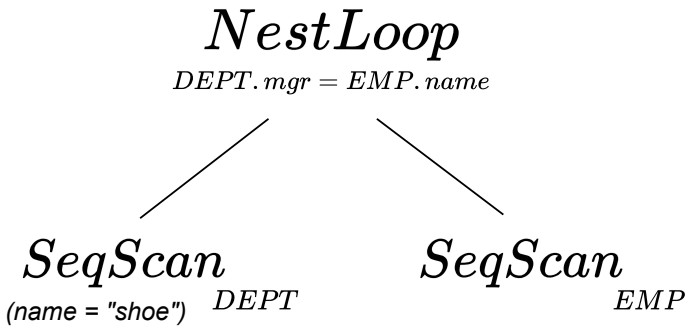


Рисунок 3.1 — План SQL-запроса с листинга 3.1.

Функция *ExecutorStart()* вызывается первой. Она вызывает *InitPlan()*, которая вызывает *ExecInitNode()* от корня плана запроса — оператора *NestLoop*. Далее вызывается *ExecInitNode()* от правого и левого подплана и так далее,

пока не будет инициализирован весь план. Результатом выполнения функции *ExecInitNode()* является дерево состояний плана, построенное по той же структуре, что и базовое дерево плана запроса.

Затем, когда вызывается функция *ExecutorRun()*, она вызывает *ExecutePlan()*, которая в цикле вызывает *ExecProcNode()* от корневого узла дерева состояний плана, в нашем примере происходит вызов *ExecProcNode()* оператора *NestedLoop* — *ExecNestLoop()*, который вызывает *ExecProcNode()* от своих подпланов. Каждый из этих подпланов представляет собой оператор последовательного сканирования, поэтому вызывается функция *ExecSeqScan()*. *ExecSeqScan()* возвращает кортежи *ExecNestLoop()*, который уже формирует итоговый кортеж и возвращает результат клиенту.

После завершение соединения *NestLoop* вызывается функция *ExecutorEnd()*, которая вызывает *ExecEndNode()*, которая вызывает *ExecEndNestLoop()*, которая, в свою очередь, вызывает *ExecEndNode()* от своих подпланов, что приводит к вызовам *ExecEndSeqScan()*.

3.2 Компиляторная инфраструктура LLVM

LLVM [28] – компиляторная инфраструктура для компиляции и оптимизации программ. В LLVM используется низкоуровневое типизированное платформонезависимое промежуточное представление LLVM IR [63], основанное на SSA-форме, которое, в свою очередь, может быть представлено и использовано тремя взаимозаменяемыми формами:

- граф структур данных, представляющих функции, базовые блоки и инструкции в оперативной памяти – используется для генерации, анализа и оптимизации программ;
- компактное закодированное бинарное представление, называемое битокодом LLVM, – используется в качестве общепринятого формата взаимодействия различных инструментов и утилит, составляющих инфраструктуру LLVM; имеет расширение **.bc*;
- человекочитаемое текстовое представление – используется для отладки; имеет расширение **.ll*.

Инфраструктура LLVM предоставляет богатый API на языках C, C++, Go, OCaml и Python для анализа и оптимизации программ. Для LLVM существует большое количество компиляторов, поддерживающих языки программирования: Си, Си++, D, Delphi, Fortran, Haskell, Julia, Lua, Objective C, Swift, Rust; поддерживается широкий набор целевых архитектур: ARM, MIPS, x86, PowerPC, SPARC, RISC-V, даже WebAssembly.

Инфраструктура LLVM содержит модули для кроссплатформенной динамической компиляции: MCJIT [64], ORC JIT [65] и LLJIT [66], в которых задействованы механизмы LLVM для машинно-зависимой оптимизации и генерации кода под различные платформы. Используя эти модули и LLVM API, можно динамически компилировать исполняемый код во время выполнения основной программы, что позволяет учитывать при оптимизации больше информации, например, типы переменных (для динамически типизированных языков). Пользовательский интерфейс перечисленных модулей для динамической компиляции состоит из функций, позволяющих:

- компилировать модули (единицы компиляции) LLVM IR в память и компоновать их с уже загруженными объектами;
- загружать в память и компоновать с программой существующие объектные модули;
- находить адреса загруженных в память символов по их имени (для вызова динамически скомпилированных функций и доступа к данным).

Динамическая компиляция и оптимизация с помощью LLVM используются во многих проектах. LLVM используют как при разработке новых языков программирования (Julia [67]), так и при создании более производительных реализаций существующих: JavaScript (JavaScriptCore [68] и V8 [69]), Python (Numba [70]), Ruby (MacRuby [71]) и др. Некоторый функционал Dropbox [72] и Facebook [73] ускорен с использованием LLVM. LLVM также используется для компиляции запросов в СУБД (HyPer [43], PostgreSQL [41] и др.) и в системах распределённой обработки данных (GreenPlum [38], Impala [23]).

3.3 Архитектура динамического компилятора запросов для СУБД PostgreSQL

С использованием компиляторной инфраструктуры LLVM задача реализации динамического компилятора запросов сводится к разработке программного компонента, транслирующего операторов по дереву плана запроса из модели Volcano в код на промежуточном представлении LLVM IR в модели явных циклов. Полученный код оптимизируется с использованием инструментов оптимизации, встроенных в LLVM. Компиляция в машинный код и подготовка к выполнению также осуществляется средствами LLVM.

Схема архитектуры реализованного динамического компилятора запросов представлена на рисунке 3.2.

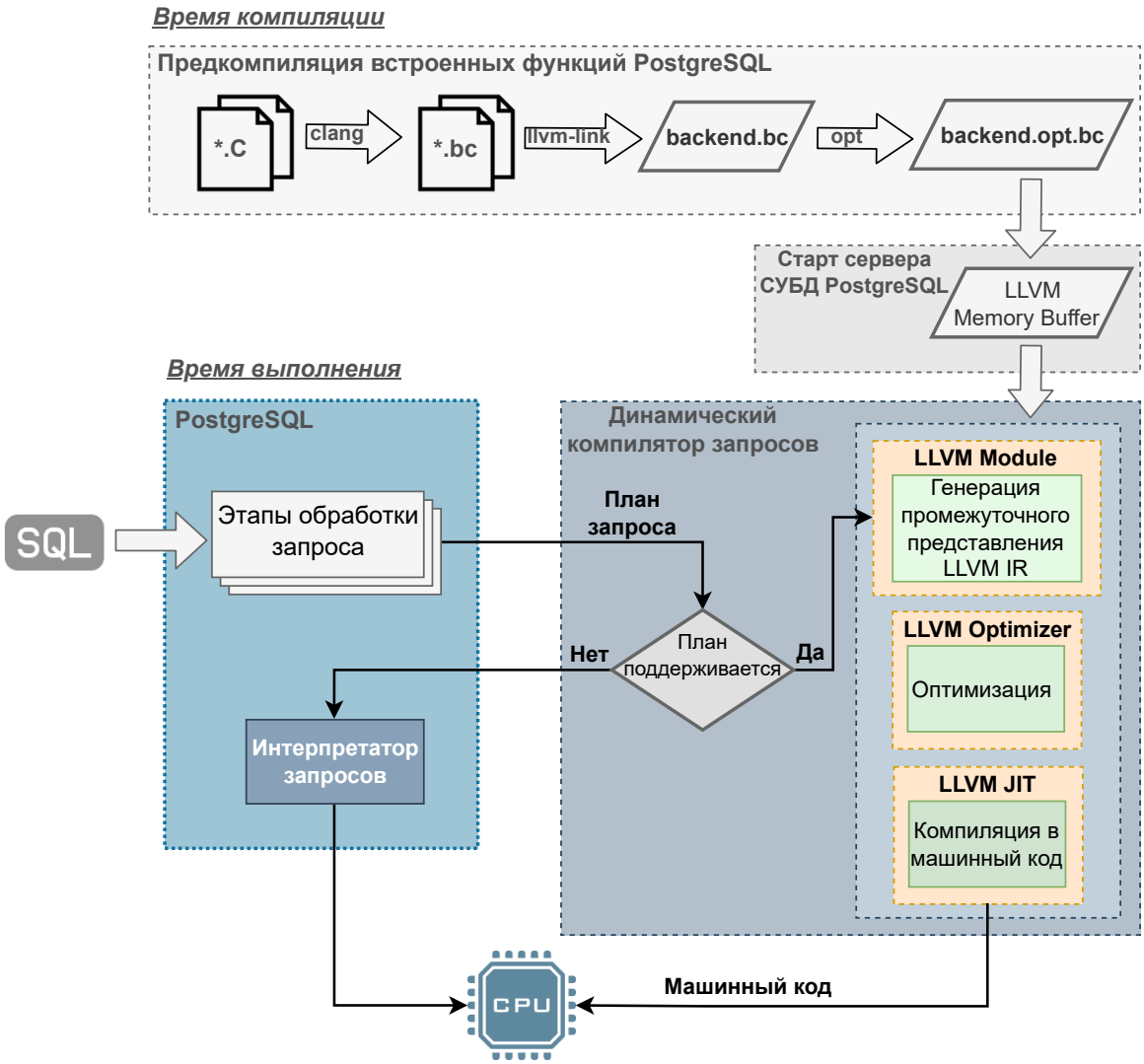


Рисунок 3.2 — Архитектура динамического компилятора запросов СУБД PostgreSQL.

Расширяемость, заложенная на архитектурном уровне в СУБД PostgreSQL, предоставляет весьма широкие возможности для определения новых типов данных, индексов, новых функций и операторов для использования в SQL-запросах, а также позволяет перехватывать управление на разных этапах обработки запроса при помощи регистрации функций-перехватчиков. Непосредственно в динамическом компиляторе запросов используется функция-перехватчик *ExecutorRun_hook*, которая вызывается перед выполнением плана запроса для ее перехвата.

В процессе сборки расширения для динамической компиляции запросов выполняется предварительная компиляция, оптимизация и компоновка встроенных функций СУБД PostgreSQL в единый LLVM модуль с использованием компилятора *clang* [57], статического оптимизатора промежуточного представления *opt* [74] и компоновщика LLVM-модулей *llvm-link* [75], которые входят в состав компиляторной инфраструктуры LLVM [28]. В полученном LLVM модуле содержаться все встроенные функции PostgreSQL в представлении LLVM IR. Метод предкомпиляции позволяет избавиться от ручной реализации функций-генераторов встроенных функции PostgreSQL, необходимых для динамической компиляции.

При запуске СУБД считывает специальный конфигурационный файл *postgresql.conf* с настройками и выполняет процесс инициализации. В этом файле можно задать параметры различного характера, такие как объём оперативной памяти для дискового кэша, допустимое количество подключений, а также пользовательские динамические библиотеки. Расширение для динамической компиляции запросов реализовано в виде динамической библиотеки *.so* (Shared Object), которая также содержит единый LLVM модуль, содержащий встроенные функции СУБД. Во время загрузки динамической библиотеки расширение регистрирует собственный обработчик выполнения запроса, используя функцию-перехватчик *ExecutorRun_hook*. Перехват управления осуществляется на этапе когда все фазы анализа запроса уже выполнены, а его план построен и оптимизирован. Схематически описанный процесс представлен на рисунке 3.3.

После перехвата выполнения запроса динамическим компилятором выполняется проверка поддержки всех выражений и операторов обработки данных, используемых в плане запроса, путём рекурсивного обхода плана и сбора информации о поддержки используемых в нем узлов. В случае если все выражения и

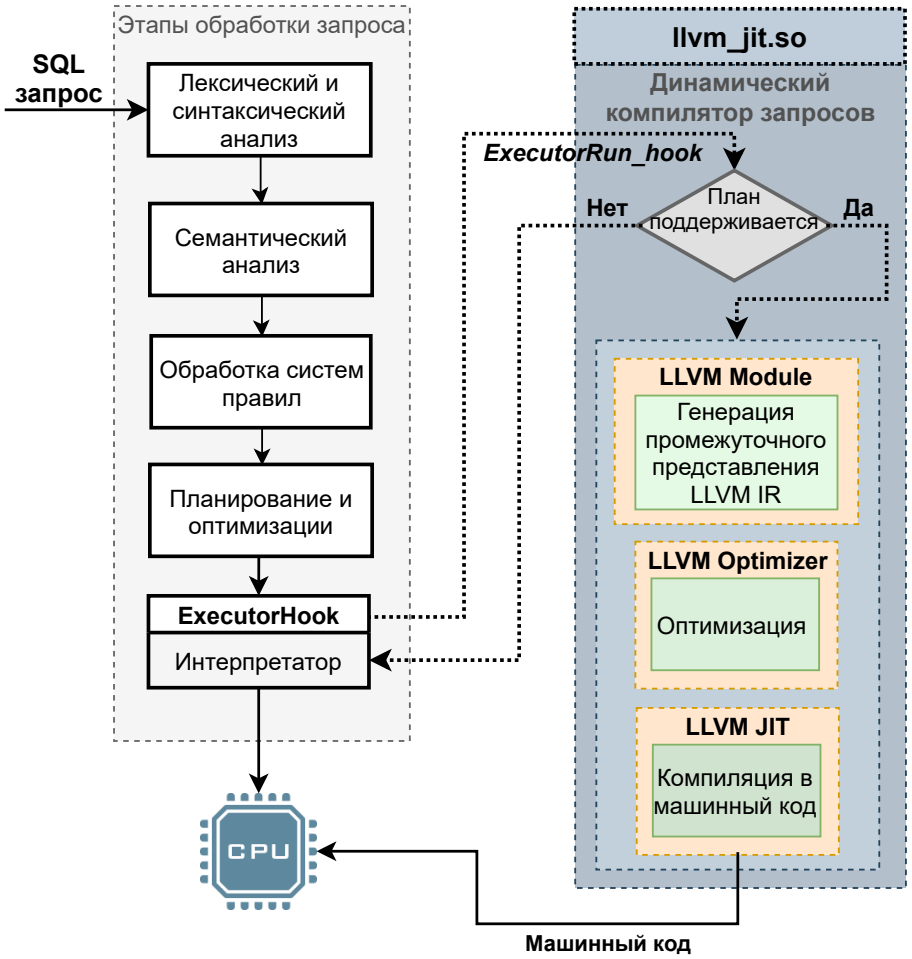


Рисунок 3.3 — Процесс перехвата этапа выполнения запроса динамическим компилятором.

операторы поддерживаются, то выполняется повторной обход плана запроса и генерация семантически эквивалентного внутреннего представления LLVM IR [63] исходного плана запроса с использованием программного интерфейса LLVM C API [76]. Сгенерированный код верифицируется, оптимизируется и динамически компилируется модулем LLVM для динамической компиляции MCJIT [64] или ORC JIT [65], а затем запускается на выполнение. Вызовы встроенных функций PostgreSQL в динамически сгенерированном коде запроса выполняются из загруженного единого LLVM модуля. В случае если хотя бы один элемент плана запроса не поддерживается, то осуществляется откат к стандартному интерпретатору запросов.

Динамическая генерация кода для поступающих в СУБД SQL запросов производится в модели явных циклов выполнения запросов, который был рассмотрен в разделе 1.1.4. Трансформация «на лету» операторов СУБД PostgreSQL из модели Volcano в модель явных циклов выполняется на основе подхода, описанного в разделе 2.1.

При разработке динамического компилятора были реализованы с использованием LLVM C API [76] большинство операторов PostgreSQL. Несмотря на несколько возросшую сложность это позволило:

– заменить используемую вычислительную модель с Volcano на модель явных циклов (раздел 3.3.1), более подходящую для генерации кода под конкретный запрос;
– реализовать динамическую компиляцию и оптимизацию вычисления арифметических выражений и предикатов (раздел 3.4);
– разработать и реализовать ряд оптимизаций, возможных только в динамически компилируемом окружении (разделы 3.4.1 и 3.4.2).

3.3.1 Реализация модели явных циклов в динамическом компиляторе запросов

В конструкции предлагаемого динамического компилятора запросов генерация кода происходит в модели явных циклов (процесс генерации описан в разделе 2.1.2) 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. Сгенерированные ими функции во внутреннем представлении передаются дочерним операторам запроса. Для каждого оператора соответствующие функции-генераторы интерфейса реализованы с использованием LLVM C API и вызываются для генерации реализующего его алгебраическую модель кода на LLVM IR, в котором функция *llvm.consume()* родительского оператора вызывается для каждого результирующего кортежа, а *llvm.finalize()* – после формирования последнего результирующего кортежа. Таким образом, после обхода дерева плана сгенерированный код после применения оптимизаций будет состоять из нескольких циклов, самым первым из которых является цикл одного из операторов сканирования таблицы.

В качестве примера рассмотрим запрос с листинга 3.2. План исполнения данного запроса, построенный оптимизатором СУБД PostgreSQL, представлен на листинге 3.3.

Листинг 3.2: Пример SQL-запроса для получения списка из 10 покупателей, сделавших наибольшее число заказов с 2000 года.

```
SELECT o_custkey, COUNT(*)
FROM orders
WHERE o_orderdate >= date '2000-01-01'
GROUP BY o_custkey
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Для выполнения этого плана требуется полностью просканировать таблицу *orders*, отфильтровать записи старше первого января 2000 года, выполнить группировку по ключу покупателя, а затем отсортировать полученные записи в порядке убывания результата агрегирующей функции `COUNT` и выбрать первые 10 записей.

Листинг 3.3: План SQL-запроса из листинга 3.2, построенного планировщиком СУБД PostgreSQL.

```
Limit
  -> Sort
       Sort Key: (COUNT(*)) DESC
       -> HashAggregate
            Group Key : o_custkey
            -> Seq Scan on orders
                 Filter: (o_orderdate >= '2000-01-01'::date))
```

Для запроса из листинга 3.2 генерация кода в промежуточном представлении LLVM IR в динамическом компиляторе запросов начинается с оператора вывода результата *Print*[1], для которого логика отправки кортежей реализуется в *llvm.print.main()*. Далее выполняется вызов функции-генератора дочернего узла *Limit* с передачей в качестве аргумента сгенерированной функции *llvm.print.main*. В свою очередь оператор *Limit* создаёт функции *llvm.limit.consume()* и *llvm.limit.finalize()*, в которых осуществляется учёт количества полученных кортежей и возвращается числовой статус, согласно модели из подраздела 2.1.3. Операция спуска в дочерний узел *Sort* повторяется с аргументами *llvm.limit.consume* и *llvm.limit.finalize*. Для материализующего оператора *Sort*, мы декомпозируем его алгоритм на две функции:

[1]В реализации динамического компилятора запросов отправка результатов запроса клиенту инкапсулирована в отдельном операторе, который отсутствует в оригинальном исполнителе.

llvm.sort.consume(), выполняющая вставку кортежа дочернего узла во временный буфер, *llvm.sort.finalize()*, непосредственно отвечающая за сортировку и процесс итерации по упорядоченному результату. Далее спуск повторяется и сгенерированные функции интерфейса передаются в нижестоящий оператор *HashAgg* и генерируется *llvm.hashAgg.consume()*, где заключается логика по вставке кортежа во временную хеш-таблицу, а затем *llvm.hashAgg.finalize()*, которая осуществляет агрегацию аккумулированных в хеш-таблице данных с последующим итерированием. Операция рекурсивного спуска повторяется и спустившись в листовой оператор *SeqScan*, создаём функцию *llvm.scan.main()*, в которой реализован внешний цикл по таблице, где производится последовательное чтение и распаковка кортежей и осуществляется операция по вычислению предиката и проекции. Для каждого отфильтрованного кортежа генерируется вызов функции *llvm.agg.consume()*. В завершении вставляется вызов функции *llvm.agg.finalize()*. Созданная функция *llvm.scan.main()* возвращается в качестве результата и является точкой входа программы для запуска выполнения запроса. На листинге 3.4 показан псевдокод сгенерированного кода для запроса из листинга 3.3 в модели явных циклов после применения оптимизации встраивания в *llvm.scan.main()* вызываемых ею функций.

Листинг 3.4: Псевдокод сгенерированного кода в модели явных циклов для плана запроса на листинге 3.3 после применения оптимизации встраивания.

```
llvm.scan.main() {
    for page in table
        for tuple_scan in page:
            if llvm.ExecQual(llvm.heap_deform_tuple(tuple_scan)):
                llvm.project(tuple_scan)
                hashtable.put(tuple_scan)
    hashtable.aggregate()
    for tuple_agg in hash_table:
        llvm.project(tuple_agg)
        sort_buffer.put(tuple_agg)
    sort_buffer.sort()
    for tuple_sort in sort_buffer:
        llvm.project(tuple_sort)
        if limit_threshold:
            print(tuple_sort)
        else
            break
    return 0
}
```

Следует отметить, что адреса большинства переменных и структур, используемых во время выполнения запроса, не изменяются, что позволяет выполнять кодогенерацию с их константными значениями, уменьшая количество разыменований и повышая производительность. Благодаря этому кортеж-приёмник для текущего оператора создаётся во время кодогенерации при помощи специальной функции, которая выполняет спуск в дочернее поддерево и возвращает подходящий слот, в который будут загружаться данные в ходе работы нижестоящего оператора. Это позволяет не выполнять передачу указателя на новый кортеж в качестве аргумента функции *consume()*, что упрощает для оптимизатора LLVM задачу по встраиванию функций.

К сожалению, объем сгенерированного кода во внутреннем представлении LLVM IR для реальных запросов значительный, что не позволяет привести здесь полный код плана запроса, но в качестве иллюстрации на листинге 3.5 представлен упрощённый (сокращённый) код LLVM IR для оператора последовательного сканирования таблицы *SeqScan*, который показывает основные механизмы модели явных циклов для плана из листинга 3.3.

Код на LLVM IR сначала загружает в буфер файл таблицы и в цикле поочерёдно извлекает страницы данных из буфера. Затем идёт цикл по кортежам, содержавшимся в текущей странице, и для каждого кортежа происходит загрузка используемых в запросе атрибутов (*llvm.heap_ deform_ tuple()*) в регистры и проверка предиката (*llvm.ExecQual()*). Если предикат ложен, цикл по кортежам продолжается. В противном случае выполняется вызов функции *consume()* родительского оператора, в данном примере *llvm.hashagg.consume()*, который заполнит хеш-таблицу. По окончанию сканирования из базового блока *finalize* выполнится вызов функции *llvm.hashagg.finalize()*, где для аккумулированных в хеш-таблице данных вычислится значение агрегирующей функции `COUNT`.

Листинг 3.5: Фрагмент сгенерированного кода LLVM IR функций интерфейса оператора сканирования в модели явных циклов.

```
define internal i32 @llvm.scan.filter() {
entry:
    %qual = call i1 @llvm.ExecQual()
    br i1 %qual, label %filtered, label %continue
continue:
    ret i32 0
filtered:
    %status = call i32 @llvm.hashagg.consume()
    ret i32 %status
}

define internal i32 @llvm.scan.main() {
entry:
    %buffer = load i32, i32* inttoptr (i64 93993580956300 to i32*)
    %page = 0
loop:
    %page = [ 0, %entry ], [ %page++, %next_page ]
    ...
    %page_in_bounds = icmp slt i32 %page, 2665
    br i1 %page_in_bounds, label %get_page, label %finalize
get_page:
    %arg = call i8* @BufferGetPage(i32 %buffer)
    %item_id = call i32* @PageGetItemId(%struct.PageHeaderData* %arg, i16 1)
    br label %next_tuple
next_tuple:
    %tuple_index = phi i16 [1, %get_page], [ %tuple_index++, %produce_tuple]
    %item_id2 = phi i32* [ %item_id, %get_page], [%item_id3, %produce_tuple]
    ...
    %tuple_index_in_bounds = icmp sle i16 %tuple_index, %7
    br i1 %tuple_index_in_bounds, label %produce_tuple, label %next_page
produce_tuple:
    %tuple_index++ = add i32 %tuple_index, 1
    %item_id3 =
        call i8* @PageGetItemId(%struct.PageHeaderData* %page,%tuple_index++)
    ....
    call void @llvm.heap_deform_tuple()
    %status = call i32 @llvm.scan.filter()
    %stop = icmp ne i32 %status, 0
    br i1 %stop, label %next_page, label %next_tuple
next_page:
    %exit_status = phi i32 [0, %next_tuple], [ %status, %produce_tuple]
    %page++ = add i32 %page, 1
    switch i32 %exit_status, label %break [
        i32 0, label %loop
        i32 1, label %finalize
    ]
finalize:
    %status1 = call i32 @llvm.hashagg.finalize()
    ret i32 %status1
break:
    %status2 = sub i32 %status, 1
    ret i32 %status2
}
```

Проверка предиката

Цикл по страницам данных

Цикл по кортежам страницы

Система переходов

Механизм прерывания обработки запроса

Для демонстрации реализованных генераторов кода операторов СУБД PostgreSQL на листинге 3.6 сверху представлен пример части кода функции СУБД PostgreSQL `ExecScan`, а снизу соответствующие функции-генераторы.

Листинг 3.6 — Сверху — фрагмент исходного кода на языке C функций `ExecScan` и `agg_fill_hash_table` СУБД PostgreSQL. Снизу — код функций-генераторов на LLVM C API соотвтетствующих функций.

```
TupleTableSlot *
ExecScan(...) {
    ...
    for (;;) {
        TupleTableSlot *slot = ExecScanFetch(node, accessMtd, recheckMtd);

        /* check that the current tuple satisfies the qual-clause */
        if (!qual || ExecQual(qual, econtext, false)) {
            /* Form a projection tuple */
            if (projInfo)
                return ExecProject(projInfo, &isDone);
            else
                return slot;
        }
    }
    ...
}

void
agg_fill_hash_table(...) {
    ...
    for (;;) {
        TupleTableSlot *outerslot = ExecScan(aggstate);
        ...
        entry = lookup_hash_entry(aggstate, outerslot);
        advance_aggregates(aggstate, entry->pergroup);
    }
    ...
}
```

```
LLVMValueRef
ScanConsumeCodegen(...) {
    ...
    /* Check filter condition */
    if (qual) {
        LLVMValueRef pred = GenerateQual(qual, econtext, scanslot, NULL, NULL);
        LLVMBuildCondBr(pred, filtered_bb, continue_bb);
    }

    /* Form a projection tuple */
    if (projInfo)
        GenerateProject(projInfo, scanslot, NULL, NULL);

    /* Produce slot */
    status = LLVMBuildCall(llvm.agg.consume, NULL, 0, "status");
    LLVMBuildRet(status);

    /* continue_bb */
    ...
    return llvm.scan.consume;
}

LLVMValueRef
HashAggConsumeCodegen(...) {
    ...
    /* entry */
    entry = LLVMBuildCall(llvm.lookup_hash_entry, NULL, 0, "entry");
    ret = LLVMConstInt(LLVMInt32Type(), offsetof(AggHashEntryData, pergroup), 0);
    pergroup = LLVMBuildGEP(entry, &ret, 1, "pergroup");
    LLVMBuildCall(llvm.advance_aggregates, &pergroup, 1, "");
    ...
    return llvm.agg.consume;
}
```

Функция `ExecScan` реализует последовательное сканирование таблицы, проверку предиката и проекцию, и функции `agg_fill_hash_table`, где выполняется чтение входящего кортежа и построение хеш-таблицы. Код функций-генераторов реализован на LLVM C API для генерации соответствующего кода во внутреннем представлении LLVM IR.

Таким образом, в динамическом компиляторе запросов были реализованы, с использованием LLVM C API, функции-генераторы большинства операторов СУБД PostgreSQL для генерации «на лету» специализированного семантически эквивалентного промежуточного представления LLVM IR кода запроса в модели явных циклов. Генерация кода в виде отдельных функций интерфейса для операторов плана запроса избавляет от основных недостатков присущих современным динамическим компиляторам запросов: отсутствует раздувание кода во время генерации и сгенерированные функции могут быть проверены и отлажены с использованием существующих отладчиков и визуализаторов, что является важным составляющим для промышелнного применения.

Далее мы опишем реализацию функций-генераторов интерфейса в модели явных циклов для основных типов операторов СУБД PostgreSQL.

3.3.2 Динамическая компиляция операторов сканирования

В СУБД PostgreSQL реализованы три основных типа сканирования таблицы:

- Последовательное сканирование (SeqScan)
- Индексное сканирование (IndexScan)
- Сканирование по битовой карте (BitmapScan)

Прежде чем перейти к описанию реализации операторов сканирования таблицы в динамическом компиляторе запросов, для улучшения читаемости и прояснения различных идей и изменений в алгоритмах, мы используем псевдокод. Чтобы уменьшить размер псевдокода и сконцентрироваться на частях, относящихся к главе, мы используем следующие абстракции:

- *next(Node)* — вызов функции, реализующий оператор, по указателю.
- *load(nodeState)* — функция, выполняющая загрузку состояния оператора.
- *save(nodeState)* — функция, выполняющая сохранение состояния оператора.

Под загрузкой состояния оператора понимается загрузка из динамической памяти в стек и регистры необходимых оператору данных для продолжения работы с сохранённой позиции, а под сохранением состояния — сохранение данных состояния оператора обратно в динамическую память. В СУБД PostgreSQL информация о состоянии оператора хранится в структуре типа *PlanState* соответствующего оператора запроса.

SeqScan

SeqScan — это оператор PostgreSQL, реализующий метод последовательного сканирования таблицы в базе данных. Последовательное сканирование определено для всех таблиц и является самым базовым методом доступа к данным в PostgreSQL. SeqScan работает, начиная сканирование с начала таблицы и просматривая все строки до конца таблицы. Для каждой строки в таблице SeqScan проверяет условия запроса (то есть условие WHERE); если условия вы-

полнены, то необходимые столбцы добавляются в набор результатов. Подробное рассмотрение данного метода и его реализации в динамическом компиляторе запросов предварим кратким описанием основной структуры данных, используемой в PostgreSQL, а именно *heap*-файла.

Heap-файл — это файл на диске, содержащий данные таблицы. Одна таблица в PostgreSQL может быть представлена несколькими *heap*-файлами, особенно если таблица большая: размер *heap*-файла ограничен и по умолчанию составляет 1 ГБ.

Heap-файл состоит из последовательности страниц фиксированного размера (по умолчанию 8 КБ). Структура страницы представлена на рисунке 3.4.

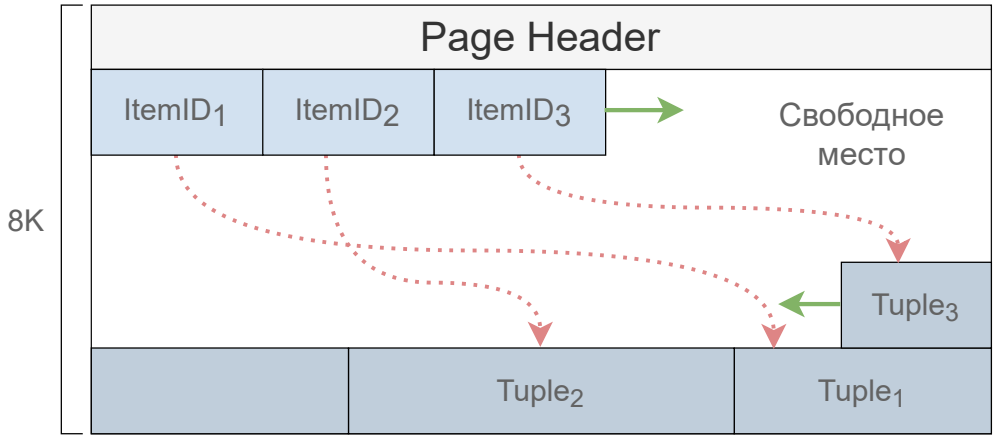


Рисунок 3.4 — Структура страницы в heap-файле.

Данные таблицы представляются набором кортежей (tuple), причём одной строке таблицы может соответствовать больше одного кортежа в силу используемого в PostgreSQL механизма многоверсионности (MVCC). Кортежи пишутся с конца страницы в начало, а с начала в конец пишутся указатели на кортежи (ItemId), состоящие из смещения на странице и длины кортежа, что позволяет эффективно последовательно обходить все кортежи, присутствующие на странице, — что и составляет основу оператора SeqScan. Таким образом, оператор SeqScan состоит из двух вложенных циклов:

1. Внешний цикл по всем страницам таблицы. Поддерживается выполнение и в параллельном режиме на нескольких процессах.
2. Внутренний цикл по всем ItemId и соответствующим им кортежам на странице. Для каждого кортежа производится проверка соответствия выполняемой транзакции и вычисление предиката условия WHERE.

Следуя модели Volcano, оператор SeqScan представляется итератором, с каждым вызовом соответствующего метода *next()* (SeqNext) производится вы-

полнение не более одной итерации внешнего цикла и не более одной итерации внутреннего и возврат следующего кортежа или индикатора конца потока. При этом каждый следующий вызов продолжает выполнение циклов с позиций, достигнутых предыдущим вызовом, и обновляет значения счётчиков, сохранённых в объекте состояния *SeqScanState* для того, чтобы выполнение можно было продолжить при следующем вызове. Для вычисления предиката условия WHERE используется интерпретатор выражений (подробнее в разделе 3.4). Для обеспечения доступа к участвующим в выражении атрибутам кортежа используется функция *slot_getattr*, которая по мере необходимости производит частичную десериализацию атрибутов кортежа. Псевдокод алгоритма последовательного сканирования SeqScan в СУБД PostgreSQL представлен на листинге 3.7.

Листинг 3.7: Псевдокод алгоритма оператора сканирования SeqScan в СУБД PostgreSQL.

```
ExecSeqScan(ExecScan) {
    while (tuple = (next(ExecScan(SeqNext)) != NULL)) {
        return ExecProject(tuple);
    }
    return NULL;
}
ExecScan(SeqNext) {
    tuple = next(SeqNext);
    if (ExecQual(slot_getattr(tuple)))
        return tuple;
}

SeqNext(SeqScanState) {
    tuple = heap_getnext(SeqScanState);
    return tuple;
}
heap_getnext(SeqScanState) {
    load(SeqScanState);
    while (page = NextPage(table) != NULL) {
        while (tuple = NextTuple(page) != NULL) {
            save(SeqScanState);
            return tuple;
        }
    }
    return NULL;
}
```

В динамическом компиляторе запросов для реализации оператора SeqScan использовался подход, описанный в разделе 2.1.4. Алгоритм оператора SeqScan был декомпозирован следующим образом: в функции *filter()* выполняется проверка предиката условия WHERE и вызов функции *consume()* родительского оператора, а в функции *main()* выполняются два вложенных цикла (внешний цикл по страницам таблицы, внутренний — по всем кортежам страницы), вызов функции *filter()* из внутреннего цикла и вызов функции *finalize()* родительского оператора после завершения сканирования. Для остановки выполнения используется механизм прерывания обработки запроса, описанный в разделе 2.1.3. Сгенерированный код во внутреннем представлении LLVM IR алгоритма оператора SeqScan, декомпозированого на функции интерфейса в модели явных циклов был представлен ранее на листинге 3.5. Функция *llvm.heap_deform_tuple()* выполняет извлечение необходимых атрибутов из кортежа с применением оптимизаций, описанных в разделе 3.4.2.

Код в таком виде позволяет представить счётчики циклов и другие переменные состояния локальными переменными на стеке или регистрах процессора и загружать их только при необходимости (например, при переходе на следующую страницу).

IndexScan

В отличие от последовательного сканирования, сканирование по индексу не извлекает все записи из таблицы последовательно, а использует специальную структуру данных (в зависимости от типа индекса), соответствующую индексу, участвующему в запросе, и находит требуемые данные (согласно предикату) за минимальное сканирование. Запись, найденная с помощью сканирования индекса, указывает непосредственно на *ItemId* в соответствующей страницы *heap*-файла. Всего в PostgreSQL поддерживается пять типов индексов [77]:

1. На основе B-дерева (B-tree index).
2. На основе хеширования (Hash index).
3. На основе обобщённого дерева поиска (GiST, SP-GiST index).
4. Обобщённые инвертированные индексы (GIN).
5. Индексы зон блоков (BRIN).

Индекс по сути представляет собой сопоставление некоторых значений ключей данных с идентификаторами кортежей, TID (Tuple Identifier), или версиями строк в основной таблице индекса. TID состоит из номера блока и номера записи в этом блоке. Этой информации достаточно, чтобы выбрать определённую версию строки из таблицы. Индексы сами по себе не знают, что в модели MVCC у одной логической строки может быть несколько существующих версий; для индекса каждый кортеж — независимый объект, которому нужна своя запись в индексе. Таким образом, при изменении строки для неё всегда заново создаются новые записи индекса, даже если значения ключа не изменились.

Наиболее часто используемым типом индекса в запросах PostgreSQL являются индексы на основе B-дерева. Этот тип индексов является типом по умолчанию, когда запрос CREATE INDEX отправляется без указания типа. Индексирование на основе B-дерева поддерживает основные операторы сравнения (=, <, <=, >, >=) и комбинации из них (например, BETWEEN, IS NULL).

Листинг 3.8: Псевдокод алгоритма оператора сканирования IndexScan в СУБД PostgreSQL.

```
ExecIndexScan(ExecScan) {
    while (tuple = (next(ExecScan(IndexNext)) != NULL)) {
        return ExecProject(tuple);
    }
    return NULL;
}
ExecScan(IndexNext) {
    tuple = next(IndexNext);
    if (ExecQual(slot_getattr(tuple)))
        return tuple;
}

IndexNext(IndexScanState) {
    tuple = index_getnext(IndexScanState);
    return tuple;
}

index_getnext() {
    load(IndexScanState);
    tuple = NextTuple(index_struct);
    save(IndexScanState);
    return tuple;
}
```

Методы последовательного сканирования и сканирования по индексам в PostgreSQL реализуются через одни и те же интерфейсы, различаясь только в некоторых конкретных методах, поэтому общий алгоритм работы метода сканирования по индексам, представленный на листинге 3.8, схож с алгоритмом работы последовательного сканирования. Отличие состоит в способе получения кортежей — в методе сканирования по индексам используется функция *index_getnext()*, которая возвращает кортеж по заранее созданной структуре индексов. Этот цикл продолжается пока из структуры индексов можно считать подходящий кортеж. Такой способ чтения позволяет значительно ускорить выполнение запросов, выводящих меньше кортежей, чем содержится в таблице.

Для динамической компиляции метода сканирования по индексам в рамках модели явных циклов применялись те же механизмы, что и для компиляции метода последовательного сканирования. В отличие от последовательного сканирования, в сканировании по индексам всего лишь один цикл – по кортежам, возвращаемым из структуры индексов. Код на LLVM IR алгоритма оператора IndexScan, декомпозированого на функции интерфейса в модели явных циклов представлен на листинге 3.9.

Листинг 3.9: Сгенерированный код LLVM IR функции интерфейса оператора сканирования по индексу в модели явных циклов.

```
define internal i32 @llvm.indexscan.main() {
fetch_tid:
   %tid = call %struct.ItemPointerData* @llvm.index_getnext_tid()
   %tid_is_null = icmp ne %struct.ItemPointerData* %tid, null
   br i1 %tid_is_null, label %fetch_tuple, label %finalize
fetch_tuple:
   %heap = call %struct.HeapTupleData *@index_fetch_heap()
   %has_tuple = icmp ne %struct.HeapTupleData* %heap, null
   br i1 %has_tuple, label %store_tuple, label %fetch_tid
store_tuple:
   call void @llvm_heap_deform_tuple()
   br label %produce_slot
produce_slot:
   %status = call i32 @llvm.parent.consume()
   switch i32 %status, label %break [
      i32 0, label %fetch_tid
      i32 1, label %finalize
   ]
finalize:
   %status2 = call i32 @llvm.parent.finalize()
   ret i32 %status2
break:
   %status3 = sub i32 %status, 1
   ret i32 %status3
}
```

Стоит отметить, что реализация в динамическом компиляторе запросов методов чтения структуры индексов по типу созданных индексов в рамках данной работы не рассматриваются.

BitmapHeapScan

В отличие от оператора последовательного сканирования SeqScan, который осуществляет полный просмотр таблицы, оператор сканирования по битовой карте BitmapHeapScan использует информацию, предоставляемую ему оператором BitmapIndexScan. Отличительной особенностью этого оператора является оптимизация, призванная уменьшить количество случайных обращений к диску при чтении данных из необходимой таблицы. В случаях, когда порядок сортировки между индексом и таблицей слабо коррелируют и требуется извлечь умеренное количество данных, оптимизатор запросов выберет стратегию с BitmapIndexScan вместо обычного сканирования по индексу.

Оператор BitmapHeapScan использует один или несколько индексов, выбор которых осуществляется на основе их совместимости с предикатными условиями. При использовании нескольких индексов, в зависимости от предикатов, применяется один из двух вспомогательных узлов: Bitmap And или Bitmap Or. Данные узлы выполняют конъюнкцию или дизъюнкцию над битовыми картами, полученными из каждого индекса по некоторому предикату. Битовая карта представляет собой массив из десяти 32 битных чисел, который позволяет записать в себя информацию о 320 кортежах, располагаемых в интересующей странице. Стоит отметить, что массив покрывает максимально возможное количество кортежей на одной 8кб странице, варьирующееся от 250 до 300.

После выполнения необходимой логической операции над битовыми картами в оператор BitmapHeapScan попадает упорядоченное множество идентификаторов на кортежи. Это множество отсортировано в порядке возрастания номера страницы, который определяет местоположение страницы относительно начала *heap*-файла. На основе этой информации оператор BitmapHeapScan может выполнить последовательное чтение данных из дисковой подсистемы с одноразовым просмотром необходимых страниц, что положительно сказывается на скорости работы жёстких дисков.

В отличие от оператора IndexScan, где алгоритм извлечения кортежей с использованием индекса инкапсулирован в одном узле, оператор BitmapHeapScan использует двухэтапный подход:

1. Создаётся битовая карта, содержащая информацию о расположении кортежей в файлах таблицы, удовлетворяющих определённому условию предиката. Эта операция выполняется дочерними узлами.
2. Осуществляется перебор элементов битовой карты и извлечение кортежей из heap-файлов. В случае использования «облегчённого» метода хранения данных в битовой карте из-за нехватки оперативной памяти может быть выполнена дополнительная проверка фильтрации для вычисления предиката.

Псевдокод алгоритма BitmapHeapScan в СУБД PostgreSQL представлен на листинге 3.10.

Листинг 3.10: Псевдокод алгоритма оператора сканирования BitmapHeapScan в СУБД PostgreSQL.

```
ExecBitmapHeapScan(ExecScan) {
    while (tuple = (next(ExecScan(BitmapHeapNext)) != NULL)) {
        return ExecProject(tuple);
    }
    return NULL;
}
ExecScan(BitmapHeapNext) {
    tuple = next(BitmapHeapNext);
    if (ExecQual(slot_getattr(tuple)))
        return tuple;
}

BitmapHeapNext(BitmapHeapScanState) {
    load(BitmapHeapScanState);
    if (bitmap == NULL)
        bitmap = buildBitmap();
    while (entry = NextBitmapEntry(bitmap) != NULL) {
        while (tuple_offset = NextTupleOffset(entry) != NULL){
            tuple <- tuple_offset[];
            save(BitmapHeapScanState);
            return tuple;
        }
    }
    return NULL;
}
```

Оператор сканирования BitmapHeapScan является своего рода листовым узлом, несмотря на то, что всегда является родителем таких узлов как BitmapIndexScan, Bitmap And или Bitmap Or. Это объясняется тем, что процесс извлечения из индекса указателей на кортежи таблицы и выполнения над ними логических операций осуществляются целиком, после чего отсортированный результат передаётся в оператор BitmapHeapScan.

Для динамической компиляции метода сканирования по битовой карте в рамках модели явных циклов применялись те же механизмы, что и для компиляции метода последовательного сканирования и сканирования по индексу. Псевдокод алгоритма оператора BitmapHeapScan, декомпозированого на функции интерфейса *consume()* и *finalize()*, представлен на листинге 3.11. В функции *finalize()* выполняется построение битовой карты, далее идёт цикл по элементам битовой карты с извлечением необходимых страниц и кортежей по соответствующему смещению. Как можно увидеть из псевдокода сохранение и восстановление состояние оператора BitmapHeapScan в реализации модели явных циклов не требуется.

3.3.3 Динамическая компиляция операторов соединения

В СУБД PostgreSQL реализованы следующие три основных типа соединения таблиц:

- соединение вложенным циклом (NestedLoop)
- соединение по хешу (HashJoin)
- соединение слиянием (MergeJoin)

NestedLoop

Соединение вложенным циклом (Nested Loop Join) — это базовый алгоритм соединения, в котором каждая запись внешнего отношения сопоставляется с каждой записью внутреннего отношения. NestedLoop является наиболее распространенным методом соединения, и его можно использовать практически

Листинг 3.11: Сгенерированный код LLVM IR функции интерфейса оператора сканирования по битовой карте в модели явных циклов.

```
define internal i32 @llvm.BitmapHeapScan.main() {
entry:
   %tbm = load %struct.TIDBitmap*, %struct.TIDBitmap** %bitmap
   %tbm_is_null = icmp eq %struct.TIDBitmap* %tbm, null
   br i1 %tbm_is_null, label %init_bitmap, label %loop
init_bitmap:
   %tbm = call %struct.TIDBitmap* @llvm.BitmapIndexScan()
   br label %loop
loop:
   %tbmRes = load %struct.TBMIterateResult*, %struct.TBMIterateResult** %tbmRes_ptr
    %tbmRes_is_null = icmp eq %struct.TBMIterateResult* %tbmRes, null
   br i1 %tbmres_is_null, label %next_page, label %prev_page
prev_page:
   %rs_cindex = add i32 %rs_cindex, 1
   br label %check_range
next_page:
   %tbmIter = call %struct.TBMIterator* @tbm_begin_iterate(%tbm)
   %tbmRes = call %struct.TBMIterateResult* @tbm_iterate(%tbmIter)
   %tbmRes_is_null = icmp eq %struct.TBMIterateResult* %tbmRes, null
   br i1 %tbmRes_is_null, label %finalize, label %next_page_block
next_page_block:
   %blockno = load i32 getelementptr %tbmRes, i32 0, i32 0
   %nblocks = load i32, i32* %nblocks_ptr
   %blockcmp = icmp uge i32 %blockno, %nblocks
   br i1 %blockcmp, label %next_page_block_true, label %next_page_fetch_heap
next_page_block_true:
   store %struct.TBMIterateResult* null, %struct.TBMIterateResult** tbmRes_ptr
   br label %loop
next_page_fetch_heap:
   call fastcc void @bitgetpage(%struct.TBMIterateResult* %tbmRes)
   br label %check_range
check_range:
   cmp = i8 checkRange(...)
   br i1 %cmp, label %next_page_block_true, label %fetch_tuple
fetch_tuple:
   %targoffset = load i16, i16* %rs_vistuples[rs_cindex]
   %arg = call %struct.PageHeaderData* @BufferGetPage()
   %item_id = call i8* @PageGetItemId(%arg, %targoffset)
   %item = call %struct.HeapTupleHeaderData* @PageGetItem(%arg, %item_id)
   store %struct.HeapTupleHeaderData* %item, %struct.HeapTupleHeaderData** %item_ptr
   %tbmres = load %struct.TBMIterateResult*, %struct.TBMIterateResult** %tbmres_ptr
   %blockno_ptr = getelementptr %tbmres, i32 0, i32 0
   %blockno = load i32, i32* %blockno_ptr
   ...
   br label %store_tuple
store_tuple:
   call void @llvm_heap_deform_tuple()
   %recheck = load i8, i8* %recheck_ptr
   %recheck_zero = icmp ne i8 %recheck, 0
   br i1 %recheck_zero, label %recheck_required, label %produce_slot
recheck_required:
   %qual = call i1 @llvm.ExecQual()
   br i1 %qual, label %produce_slot, label %loop
produce_slot:
   %status = call i32 @llvm.parent.consume()
   switch i32 %status, label %break [
      i32 0, label %loop
      i32 1, label %finalize
   ]
finalize:
   %status2 = call i32 @llvm.parent.finalize()
   ret i32 %status2
break:
   %status3 = sub i32 %status, 1
   ret i32 %status3
}
```

для любого набора данных с любым типом условия соединения. Поскольку этот алгоритм сканирует все кортежи внутреннего и внешнего отношения, он считается самой дорогостоящей операцией соединения.

Алгоритм работы оператора NestedLoop в СУБД PostgreSQL представлен на листинге 3.12. Функция ExecReScan сбрасывает состояние оператора внутреннего сканирования таблицы для выполнения сканирования с самого начала.

Листинг 3.12: Псевдокод алгоритма оператора соединения NestedLoop в СУБД PostgreSQL.

```
ExecNestLoop() {
    while (tuple_outer = (next(Scan_outer) != NULL)) {
        while (tuple_inner = (next(Scan_inner) != NULL)) {
            if (ExecQual(slot_getattr(tuple_outer), slot_getattr(tuple_inner)))
                return join(tuple_outer, tuple_inner);
        }
        ExecReScan(Scan_inner);
    }
}
```

Псевдокод алгоритма NestedLoop, декомпозированого на функции интерфейса *consume()* и *finalize()*, представлен на листинге 3.13.

Листинг 3.13: Сгенерированный код LLVM IR функций интерфейса оператора соединения вложенными циклами в модели явных циклов.

```
define internal i32 @llvm.nestedloop.inner.consume() {
entry:
    %qual = call i1 @llvm.ExecQual()
    br i1 %qual, label %project, label %continue
project:
    %prj = call i32 @llvm.ExecProject()
    %status = call i32 @llvm.parent.consume()
    ret i32 %status
continue:
    ret i32 0
}

define internal i32 @llvm.nestedloop.outer.consume() {
entry:
    call void @llvm.inner.rescan()
    %status = call i32 @llvm.child.inner.finalize()
    ret i32 %status
}

define internal i32 @llvm.nestedloop.inner.finalize() {
entry:
    ret i32 0
}

define internal i32 @llvm.nestedloop.outer.finalize() {
entry:
    %status = call i32 @llvm.parent.finalize()
    ret i32 %status
}
```

В динамическом компиляторе запросов алгоритм оператора соединения был декомпозирован на две функции *consume()*: для внешнего дочернего и внутреннего дочернего оператора.

– В функции *llvm.nestedloop.outer.consume()* для внешнего дочернего оператора выполняется сброс состояния и вызов функции *finalize()* внутреннего дочернего оператора.
– В функции *llvm.nestedloop.inner.consume()* для внутреннего дочернего оператора выполняется проверка предиката, соединение, вызов функции *consume()* родительского оператора и возврат полученного статуса.

HashJoin

В СУБД PostgreSQL в реализации оператора соединения по хешу HashJoin используется алгоритм гибридного соединения [58]. У HashJoin в качестве внутреннего потомка планировщик PostgreSQL всегда ставит оператор Hash, который отвечает за заполнение хеш-таблицы. Оператор Hash, получив от HashJoin сигнал, что хеш-таблица создана и нужно её наполнять, в цикле получает все кортежи от своего потомка и добавляет их в хеш-таблицу.

Оператор HashJoin с точки зрения взаимодействия со своим внешним потомком и родителем устроен значительно сложнее, чем рассмотренные ранее операторы. В коде СУБД PostgreSQL реализация оператора HashJoin фактически представляет собой конечный автомат: тело функции ExecHashJoin — это конструкция *switch-case* по состояниям автомата, обёрнутая в бесконечный цикл. Работа автомата проиллюстрирована на рисунке 3.5. Для упрощения описания автомата состояний оператора HashJoin мы будем рассматривать вариант соединения по-умолчанию (INNER), где опущены некоторые состояния из оригинального алгоритма в PostgreSQL ввиду их недостижимости.

Поясним каждое из состояний:

BUILD_HASHTABLE — осуществляется получение всех кортежей от внутреннего оператора и создание хеш-таблицы.

NEED_NEW_OUTER — извлекается следующий кортеж с внешней стороны.

SCAN_BUCKET — просматривается корзина, соответствующая внешнему кортежу, полученному в предыдущем состоянии на предмет соответствий. Если подходящая пара найдена вычисляется предикат, выполняется проекция и соединение кортежей.

NEED_NEW_BATCH — выполняется смена группы; содержимое хеш-таблицы уничтожается, с диска считываются внутренние кортежи следующей группы. Если групп больше нет, значит оператор закончил свою работу.

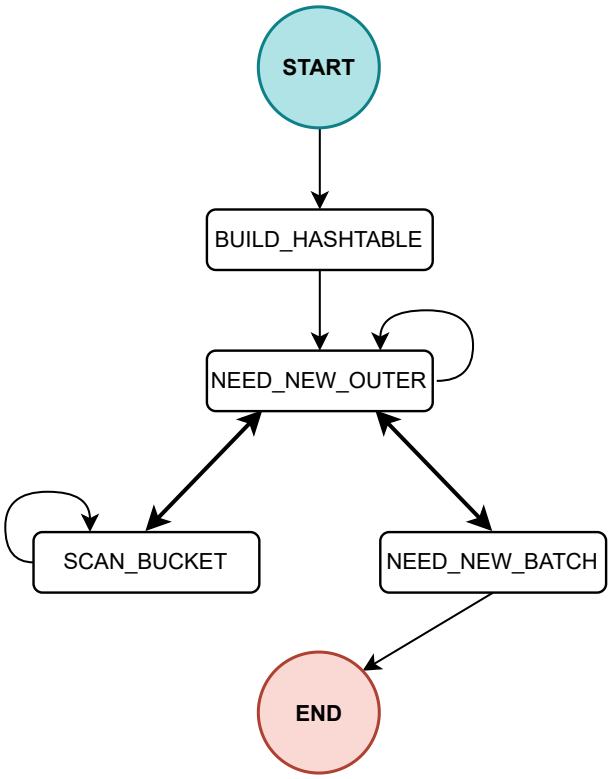


Рисунок 3.5 — Автомат состояний алгоритма соединения HashJoin в СУБД PostgreSQL.

В динамическом компиляторе запросов в модели явных циклов алгоритм оператора соединения HashJoin был декомпозирован (см. листинг 3.14) на две функции *consume()*, каждая из которых вызывается соответствующим дочерним узлом:

– В функции *llvm.hashjoin.outer.consume()* для внешнего дочернего оператора выполняется обработка внешнего кортежа, вычисляется предикат, выполняется проекция и соединение кортежей, а также выполняется смена группы.
– В функции *llvm.hashjoin.inner.consume()* для внутреннего дочернего оператора выполняется наполнение хеш-таблицы.

Листинг 3.14: Сгенерированный код LLVM IR функций интерфейса оператора соединения хешированием в модели явных циклов.

```
define internal i32 @llvm.hashjoin.outer.consume() {
entry:
    %hashvalue = call i1 @llvm.ExecHashGetHashValue(outer)
    call void @ExecHashGetBucketAndBatch(%hashvalue)
    br label %hashTuple
hashTuple:
    %CurBucketNo = load i32, i32* %CurBucketNo_ptr
    %bucket = getelementptr %struct.HashJoinTupleData*, i32 %CurBucketNo
    %hashTuple_llvm = load %struct.HashJoinTupleData*, %bucket
    %cmp = icmp eq %struct.HashJoinTupleData* %hashTuple_llvm, null
    br i1 %13, label %continue, label %matchTuple
matchTuple:
    %hashTuple_hashvalue = getelementptr i32, i32* %hashTuple_llvm
    %cmp_tuples = cmp eq i32 %hashTuple_hashvalue, %hashvalue
    br i1 %cmp_tuples, label %found, label %not_found
found:
    call void @llvm.heap_deform_tuple()
    %qual = call i1 @llvm.ExecQual()
    br i1 %qual, label %match, label %not_found
not_found:
    %hashTuple_next = load %struct.HashJoinTupleData* %hashTuple_next_ptr
    %cmp_null = icmp eq %struct.HashJoinTupleData* %hashTuple_next, null
     br i1 %cmp_null, label %continue, label %matchTuple
match:
    %prj = call i32 @llvm.ExecProject()
    %status = call i32 @llvm.parent.consume()
    %cmp_status = icmp eq i32 %status, 0
    br i1 %cmp_status, label %not_found, label %return
return:
    %status2 = sub i32 %status, #LLVM_CD(child)
    ret i32 %status2
continue:
    ret i32 0
}

define internal i32 @llvm.hashjoin.inner.consume() {
entry:
    %hashvalue = call i1 @llvm.ExecHashGetHashValue(inner)
    br i1 %hashvalue, %HashTableInsert, %exit
HashTableInsert:
    call void @ExecHashTableInsert(%hashvalue)
    br label %exit
exit:
    ret i32 0
}

define internal i32 @llvm.hashjoin.outer.finalize() {
entry:
    %status = call i32 llvm.parent.finalize()
    ret i32 %status
}

define internal i32 @llvm.hashjoin.inner.finalize() {
entry:
    %status = call i32 llvm.child.outer.finalize()
    ret i32 %status
}
```

MergeJoin

В СУБД PostgreSQL код оператор соединения слиянием MergeJoin реализован в форме конечного автомата и разбит на некоторое количество состояний. Тело функции ExecMergeJoin() — это условный оператор *switch*, где каждая метка представляет собой какое-то из состояний. Алгоритм оператора представлен в виде псевдокода на листинге 3.15, где на некоторых строках справа указано состояние, в котором выполняется соответствующая операция. Состояние автомата хранится в объекте состояния оператора MergeJoin и сохраняется между вызовами ExecMergeJoin.

Листинг 3.15: Псевдокод алгоритма оператора соединения MergeJoin в СУБД PostgreSQL.

```
ExecMergeJoin() {
    tuple_outer = next(node_outer);                          INITIALIZE_OUTER
    tuple_inner = next(node_inner);                          INITIALIZE_INNER

    while(true) {
        while (tuple_outer != tuple_inner) {                 SKIP_TEST
            if (tuple_outer < tuple_inner)
                tuple_outer = next(node_outer);              SKIPOUTER_ADVANCE
            else
                tuple_inner = next(node_inner);              SKIPINNER_ADVANCE
        }
        markedTuple_inner = tuple_inner;                     SKIP_TEST
        while(true) {
            while (tuple_outer = tuple_inner) {
                result = join(tuple_outer, tuple_inner);     JOINTUPLES
                tuple_inner = next(node_inner);              NEXTINNER
            }
            tuple_outer = next(node_outer);                  NEXTOUTER
            if (tuple_outer = markedTuple_inner)             TESTOUTER
                tuple_inner = markedTuple_inner;             TESTOUTER
            else
                break;
        }
    }
}
```

Как уже было сказано в разделе 2.1.3 оператор соединения слиянием плохо подходит для реализации в модели явных циклов в рамках динамического компилятора запросов, так как порядок работы оператора нельзя установить

заранее. Однако оператор MergeJoin реализован в динамическом компиляторе запросов и предложены некоторые оптимизации разной сложности для нивелирования недостатков модели явных циклов при декомпозиции алгоритма оператора *MergeJoin* на функции интерфейс *consume()* и *finalize()*.

На рисунке 3.6 представлен автомат состояний алгоритма соединения слиянием. Перечислим назначение состояний для дальнейшего рассмотрения процесса декомпозиции алгоритма оператора на соответствующие функции интерфейса:

INITIALIZE_OUTER — осуществляется получение первого кортежа внешнего оператора.

INITIALIZE_INNER — осуществляется получение первого кортежа внутреннего оператора.

JOINTUPLES — вычисляется предикат, выполняется проекция и соединение кортежей.

NEXTOUTER — осуществляется получение следующего кортежа внешнего оператора.

TESTOUTER — проверяется полученный кортеж из внешнего оператора на возможность соединения с маркированным в SKIP_TEST кортежем внутреннего оператора. В случае успеха выполняется откат внутреннего узла на позицию следующего кортежа после маркированного.

NEXTINNER — осуществляется получение следующего кортежа внутреннего оператора.

SKIP_TEST — выполняется сравнение двух кортежей и на основе результата выполняется сдвиг на следующий кортеж у узла, чей кортеж меньше. В случае, если кортежи равны выполняется маркировка позиции внутреннего узла.

SKIPOUTER_ADVANCE — выполняется сдвиг кортежа внешнего узла до тех пор, пока не найдётся подходящий для соединения кортеж.

SKIPINNER_ADVANCE — выполняется сдвиг кортежа внутреннего узла до тех пор, пока не найдётся подходящий для соединения кортеж.

ENDOUTER — означает, что закончились внешние кортежи.

ENDINNER — означает, что закончились внутренние кортежи.

Для упрощения описания разбиения автомата состояний оператора MergeJoin мы будем рассматривать вариант соединения по-умолчанию

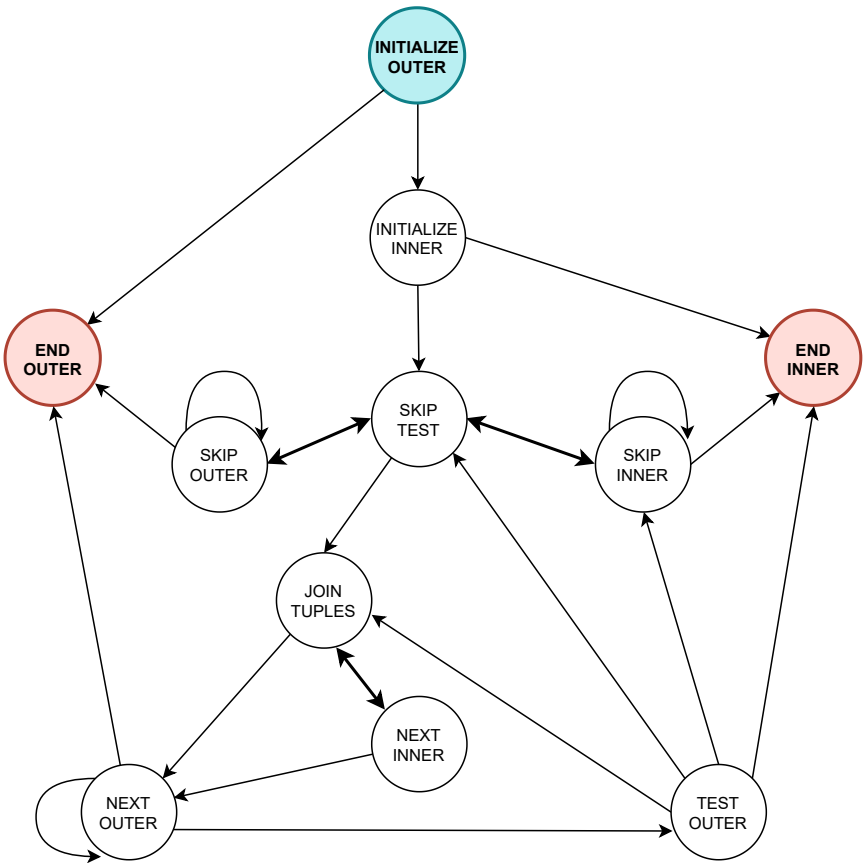


Рисунок 3.6 — Автомат состояний алгоритма соединения MergeJoin в СУБД PostgreSQL.

(INNER), где состояния ENDOUTER и ENDINNER не достижимы, и далее мы не будем принимать их во внимание.

В динамическом компиляторе запросов алгоритм бинарного оператора MergeJoin был декомпозирован на две функции *consume()*, каждая из которых вызывается соответствующим дочерним узлом после генерации нового кортежа. Для выполнения этой задачи потребовалось разбить автомат состояний на две части следующим образом:

– В функцию *llvm.mergejoin.outer.consume()* для внешнего дочернего оператора мы поместим все состояния, отвечающие за инициализацию, пропуск и обработку внешнего кортежа, а также состояния для получения внутреннего кортежа.
– В функцию *llvm.mergejoin.inner.consume()* для внутреннего дочернего оператора мы поместим аналогичные состояния, но для внутреннего кортежа, а также состояние NEXTOUTER, где будет выполняться возврат управления внешнему циклу.

Отметим, что для избежания прерывания внутреннего цикла соединения состояние JOINTUPLES было продублировано в функциях *llvm.mergejoin.outer.consume()* и *llvm.mergejoin.inner.consume()*. Это позволило существенно повысить эффективность оператора MergeJoin в рамках динамического компилятора запросов и модели явных циклов.

В общем случае для внешнего цикла не имеет значения в каком состоянии находится внутренний цикл. По этой причине любые операции, связанные с выполнением сдвига кортежа внешнего цикла, могут сводиться к вызову сгенерированной функции *finalize()* для правого дочернего узла. Таким образом состояния INITIALIZE_INNER, NEXTINNER и SKIPINNER_ADVANCE в *llvm.mergejoin.outer.consume()* могут быть объединены под общим названием OUTER_NEXTINNER. Любой переход в это состояние будет означать о потребности внешнего цикла в новом кортеже правого оператора. Выход из этого состояния предполагает потребность внутреннего цикла в новом кортеже. Внутренний цикл будет сообщать о необходимости в новом кортеже путём выполнения перехода в состояние INNER_NEXTOUTER в *llvm.mergejoin.inner.consume()*, где осуществиться возврат статуса со значением $LLVM\_CD(inner) + 1$ (вычисленной глубины правого поддерева с коррекцией, так как здесь мы хотим завершить внутренний цикл). Последний, согласно статусной модели, выполнит прерывание основного сканирующего цикла правого поддерева и вернёт управление в функцию *llvm.mergejoin.outer.consume()*. На рисунке 3.7 представлен автомат состояний каждой из функций после вышеуказанных преобразований.

Генерируемый с использованием LLVM C API код во внутреннем представлении LLVM IR для функций типа *consume()* выполняется в соответствии с графом состояний и в значительной степени совпадает с оригинальным алгоритмом оператора MergeJoin. Большая часть меток-состояний условного оператора *switch* заменяется на явные переходы в соответствующие базовые блоки. Функция *llvm.mergejoin.outer.consume()*, представляющая внешний цикл, для запуска внутреннего цикла выполняет соответствующий вызов сгенерированной для правого поддерева функции *finalize()*, где в последствии вызывается функция *llvm.mergejoin.inner.consume()*, которая может выполнять соединение, сдвиг позиции кортежа внутреннего узла с использованием статус кодов и прерывать внутренний цикл. Важно отметить, что для реализации типов объединения INNER и SEMI потребовалось выделить только одну функцию

Листинг 3.16: Сгенерированный код LLVM IR функции интерфейса оператора соединения слиянием для внутреннего цикла в модели явных циклов.

```
define internal i32 @llvm.mergejoin.inner.consume() {
entry:
    %state = load i32, i32* %State
    switch i32 %state [
        i32 #INITIALIZE_INNER, label %init_inner
        i32 #NEXTINNER, label %next_inner
        i32 #SKIPINNER_ADVANCE, label %skip_inner_advance  ]
init_inner:                ; INITIALIZE_INNER
    br label %skip_test
join_tuples:               ; JOINTUPLES
    store i32 #NEXT_INNER, i32* %State
    br label %join_qual
join_qual:
    %qual = call i1 @llvm.ExecQual()
    br i1 %qual, label %join_project, label %consume_next
join_project:
    %prj = call i32 @llvm.ExecProject()
    %status = call i32 @llvm.parent.consume()
    %statuszero = icmp eq i32 %status, 0
    br i1 %statuszero, label %consume_next, label %inner_join_return
inner_join_return:
    ret i32 %status
next_outer:                ; INNER_NEXTOUTER
    store i32 #NEXT_OUTER, i32* %State
    br label %break
next_inner:                ; NEXTINNER
    %compare = call fastcc i32 @MJCompare()
    %cmp = icmp eq i32 %compare, 0
    br i1 %cmp, label %join_tuples, label %next_outer
skip_test:                 ; SKIP_TEST
    %compare_skip = call fastcc i32 @MJCompare()
    %cmp_skip = icmp eq i32 %compareskip, 0
    br i1 %cmpskip, label %skip_test_equal, label %skip_test_notequal
skip_test_equal:
    call fastcc void @index_markpos()
    call fastcc void @index_marktuple()
    br label %join_tuples
skip_test_notequal:
    %compare_result = icmp slt i32 %compare_skip, 0
    br i1 %compare_result, label %skip_test_less, label %skip_test_more
skip_test_less:
    store i32 #SKIPOUTER_ADVANCE, i32* %State
    br label %break
skip_test_more:
    store i32 #SKIPINNER_ADVANCE, i32* %State
    br label %consume_next
skip_inner_advance:        ; SKIPINNER_ADVANCE
    br label %skip_test
consume_next:
    ret i32 0
finalize:
    ret i32 #LLVM_CD(inner)
break:
    store i1 true, i1* @llvm.mergejoin.inner_breaked
    ret i32 #LLVM_CD(inner) + 1
}

define internal i32 @llvm.mergejoin.inner.finalize() {
entry:
    store i32 #NEXTOUTER, i32* %State
    ret i32 0
}
```

Листинг 3.17: Сгенерированный код LLVM IR функции интерфейса оператора соединения слиянием для внешнего цикла в модели явных циклов.

```
define internal i32 @llvm.mergejoin.outer.consume() {
entry:
    %state = load i32, i32* %State
    switch i32 %state [
        i32 #INITIALIZE_OUTER, label %init_outer
        i32 #NEXTOUTER, label %next_outer
        i32 #SKIPOUTER_ADVANCE, label %skip_test  ]
init_outer:                ; INITIALIZE_OUTER
    br label %next_inner
join_tuples:               ; JOINTUPLES
    store i32 #NEXT_INNER, i32* %State
    br label %join_qual
join_qual:
    %qual = call i1 @llvm.ExecQual()
    br i1 %qual, label %join_project, label %next_inner
join_project:
    %prj = call i32 @llvm.ExecProject()
    %status = call i32 @llvm.parent.consume()
    %statuszero = icmp eq i32 %status, 0
    br i1 %statuszero, label %next_inner, label %outer_join_return
next_outer:                ; NEXTOUTER
    %compare = call fastcc i32 @MJCompare()
    %cmp = icmp eq i32 %compare, 0
    br i1 %cmp, label %test_outer_restore, label %skip_test
test_outer_restore:
    call fastcc void @index_restrpos()
    store i32 #NEXTINNER, i32* %State
    %qual_marked = call i1 @llvm.ExecQual.Marked()
    br i1 %qual_marked, label %join_project_marked, label %next_inner
join_project_marked:
    %prj_marked = call i32 @llvm.ExecProject.Marked()
    %status_marked = call i32 @llvm.parent.consume()
    %cmp_marked = icmp eq i32 %status_marked, 0
    br i1 %cmp_marked, label %next_inner, label %outer_join_return
outer_join_return:
    %stat = phi i32 [%status, %join_project], [%status_marked, %join_project_marked]
    ret i32 %stat
test_outer_nonmatch:
    store i32 #SKIPINNER_ADVANCE, i32* %State
    br label %next_inner
next_inner:                ; OUTER_NEXTINNER
    %run_inner_loop = call i32 @llvm.mergejoin.inner.main()
    %broke = load i1, i1* @llvm.mergejoin.inner_break
    %is_broken = icmp ne i1 %broke, false
    store i1 false, i1* @llvm.mergejoin.inner_break
    %inner_status = select i1 %is_broken, i32 0, i32 %run_inner_loop
    ret i32 %inner_status
skip_test:                 ; SKIP_TEST
    %compare_skip = call fastcc i32 @MJCompare()
    %cmp_skip = icmp eq i32 %compare_skip, 0
    br i1 %cmp_skip, label %skip_test_equal, label %skip_test_notequal
skip_test_equal:
    call fastcc void @index_markpos()
    call fastcc void @index_marktuple())
    br label %join_tuples
skip_test_notequal:
    %compare_result = icmp slt i32 %compare_skip, 0
    br i1 %compare_result, label %skip_test_less, label %skip_test_more
skip_test_less:
    store i32 #SKIPOUTER_ADVANCE, i32* %State
    br label %consume_next
skip_test_more:
    store i32 #SKIPINNER_ADVANCE, i32* %State
    br label %next_inner
consume_next:
    ret i32 0
finalize:
    ret i32 #LLVM_CD(outer)
}
define internal i32 @llvm.mergejoin.outer.finalize() {
entry:
    %status = call i32 @llvm.parent.finalize()
    ret i32 %status
}
```

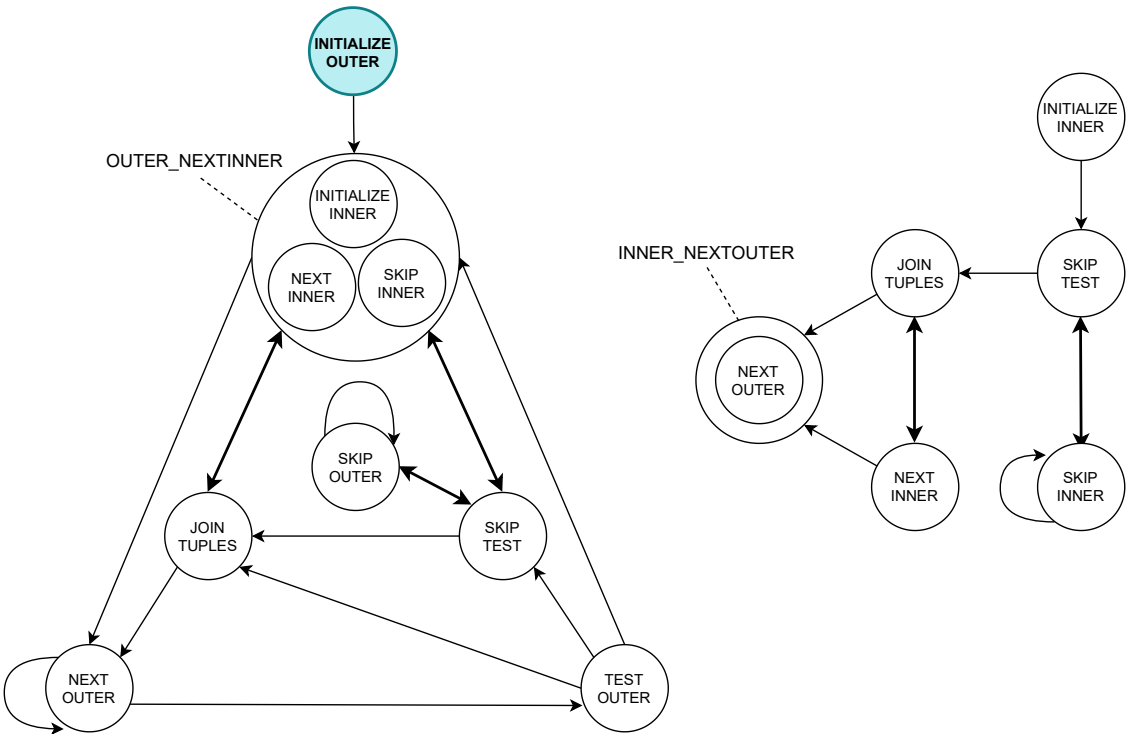


Рисунок 3.7 — Слева — автомат состояний функции *llvm.mergejoin.outer.consume()* оператора MergeJoin, справа — функции *llvm.mergejoin.inner.consume()*.

finalize() для внутреннего оператора, которая выполняет минимальное количество работы по обнулению нескольких переменных. Для реализации остальных типов потребовалось разместить логику соединения оставшихся кортежей на одной (в случае LEFT или RIGHT JOIN) или обеих (FULL JOIN) лентах с NULL кортежами в соответствующей функции *finalize()*.

Псевдокод алгоритма оператора MergeJoin, декомпозированого на функции интерфейса *consume()* и *finalize()*, представлен на листингах 3.16 и 3.17.

3.3.4 Динамическая компиляция оператора сортировки

В СУБД PostgreSQL оператор сортировки Sort выполняет упорядочивание кортежей из внешнего поддерева узла с помощью модуля *tuplesort*, который сохраняет результаты во временном файле или в памяти. После первоначального вызова при каждом последующем вызове возвращается кортеж из файла в упорядоченном порядке. Непосредственно сортировка выполняется с

использование алгоритма быстрой сортировки *qsort*. Алгоритм оператора Sort в PostgreSQL представлен на листинге 3.18.

Листинг 3.18: Псевдокод алгоритма оператора сортировки Sort в СУБД PostgreSQL.

```
ExecSort() {
    load(SortState);

    if (!sort_Done) {
        while (tuple = next(node_outer) != NULL)
            tuplesort_puttupleslot(SortState, tuple);

        tuplesort_performsort(SortState);
        sort_Done = true;
        save(SortState);
    }

    tuple = tuplesort_gettuple(SortState);
    return tuple;
}
```

Функция *tuplesort_puttuple* добавляет кортеж в SortState; *tuplesort_performsor* непосредственно выполняет сортировку; *tuplesort_gettuple* возвращает очередной кортеж из SortState в отсортированном порядке. В функции ExecSort в начале проверяется, была ли уже проведена сортировка. Если нет, оператор вытягивает все кортежи от своего потомка, добавляет их во временный файл и сортирует. Далее кортежи возвращаются по одному на вызов ExecSort.

В динамическом компиляторе запросов алгоритм оператора Sort был декомпозирован на функции *consume()* и *finalize()* следующим образом: в функции *consume()* выполняется вставка кортежа дочернего узла во временный файл, а в функции *finalize()* выполняется сортировка и итерация по упорядоченному результату, и вызов функции *consume()* родительского оператора. Для прерывания выполнения используется механизм статусной модели, описанный в разделе 2.1.3. В данной реализации оператора Sort в модели явных циклов состояние оператора не нужно загружать и сохранять при отправке каждого кортежа, а также не нужно проверять, была ли проведена сортировка, что было необходимо в функции ExecSort. Псевдокод алгоритма оператора Sort, декомпозированого на функции интерфейса *consume()* и *finalize()* представлен на листинге 3.19.

Листинг 3.19: Сгенерированный код LLVM IR функций интерфейса оператора сортировки в модели явных циклов.

```
define internal i32 @llvm.sort.consume() {
entry:
    call void @tuplesort_puttupleslot()
    ret i32 0
}

define internal i32 @llvm.sort.finalize() {
entry:
    call void @tuplesort_performsort()
    br label %get_tuple
get_tuple:
    %tuple = call fastcc i8 @tuplesort_gettuple_common()
    %next_tuple = icmp ne i8 %tuple, 0
    br i1 %next_tuple, label %produce_tuple, label %finalize
produce_tuple:
    call void @llvm.heap_deform_tuple()
    %status = call i32 @llvm.parent.consume()
    switch i32 %status, label %break [
       i32 0, label %get_tuple
       i32 1, label %finalize
    ]
finalize:
    %status2 = call i32 @llvm.parent.finalize()
    ret i32 %status2
break:
    %status3 = sub i32 %status, 1
    ret i32 %status3
}
```

3.3.5 Динамическая компиляция оператора агрегации

В СУБД PostgreSQL реализован оператор агрегации Agg, служащий для группировки кортежей. Оператор поддерживает три стратегии агрегации. Когда результат работы оператора состоит из одного кортежа, стратегия называется AGG_PLAIN. В противном случае используется либо AGG_SORTED, либо AGG_HASHED. Первая требует отсортированных по ключу агрегации входных кортежей, во второй группировка проводится с помощью хеш-таблицы. Оператор агрегации поддерживает несколько наборов группировки (grouping sets): можно разделить входные данные на несколько частей, провести агрегацию каждой и объединить в конце результаты. Есть возможность передавать кортежи агрегатной функции в заданном порядке и удалять дубликаты с помощью предложений ORDER BY и DISTINCT, причём разрешается использовать их вместе.

Реализация стратегии AGG_PLAIN достаточно проста, она не будет описана здесь. Рассмотрим стратегию AGG_HASHED, реализация которой находится в функции ExecAgg. При первом вызове функции ExecAgg заполняется хеш-таблица группированных кортежей: добавляется запись в хеш-таблицу если в ней ещё нет элемента с таким значением ключа агрегации (функция *lookup_hash_entry*); вызывается функция *advance_aggregates* для непосредственной агрегации. Далее сгруппированные кортежи извлекаются (по одному кортежу на вызов ExecAgg) из хеш-таблицы, проводится финальная обработка (функция *finalize_aggregates*), при необходимости выполняется проекция, и кортеж возвращается родительскому оператору. Алгоритм работы оператора агрегации по хешу HashAgg в PostgreSQL представлен на листинге 3.20.

Листинг 3.20: Псевдокод алгоритма оператора HashAgg в СУБД PostgreSQL.

```
ExecAgg() {
    load(AggState);

    if (!agg_done) {
        if (!table_filled)
            agg_fill_hash_table();
        result = agg_retrieve_hash_table();
        save(AggState);
        return result;
    }
}

agg_fill_hash_table() {
    while (tuple = next(node_outer) != NULL) {
        entry = lookup_hash_entry(tuple);
        advance_aggregates(entry);
    }
    table_filled = true;
}
agg_retrieve_hash_table() {
    while (!agg_done) {
        entry = ScanTupleHashTable()
        if (entry == NULL) {
            agg_done = TRUE;
            return NULL;
        }
        finalize_aggregates(entry);
        result = project_aggregates(entry);
        if (result)
            return result;
    }
}
```

В динамическом компиляторе запросов алгоритм оператора агрегации по хешу HashAgg был декомпозирован на функции *consume()* и *finalize()* следующим образом: в функции *consume()* выполняется вставка входного кортежа от дочернего узла в хеш-таблицу и продвижение агрегатного состояния, а в функции *finalize()* выполняется агрегацию аккумулированных в хеш-таблице данных с последующим итерированием и вызов функции *consume()* родительского оператора. Для прерывания выполнения используется механизм статусной модели, описанный в разделе 2.1.3. В данной реализации оператора HashAgg в модели явных циклов состояние оператора не нужно загружать и сохранять, что было необходимо в функции ExecAgg. Псевдокод алгоритма оператора HashAgg декомпозированого на функции интерфейса *consume()* и *finalize()* представлен на листинге 3.21.

Листинг 3.21: Сгенерированный код LLVM IR функций интерфейса оператора агрегации по хешу в модели явных циклов.

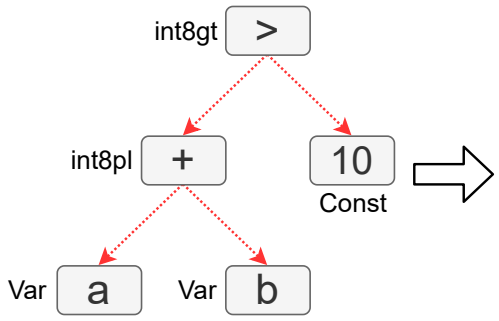
```
define internal i32 @llvm.hashagg.consume() {
entry:
    %pergroup = call %struct.TupleHashEntryData* @llvm.lookup_hash_entry()
    call void @llvm.advance_aggregates(%struct.AggStatePerGroupData* %pergroup)
    ret i32 0
}

define internal i32 @llvm.hashagg.finalize() {
entry:
    %hashtable = load %struct.HTAB*, %struct.HTAB** %HTAB
    br label %loop
loop:
    %hash_entry = call i8* @hash_seq_search(%struct.HTAB* %hashtable)
    %is_null = icmp ne i8* %hash_entry, null
    br i1 %is_null, label %finalize_group, label %finalize
finalize_group:
    call void @llvm.heap_deform_tuple()
    %pergroup = getelementptr i8, i8* %hash_entry, i32 8
    call void @llvm.finalize_aggregates(%struct.AggStatePerGroupData* %pergroup)
    %result = call i1 @llvm.project_aggregates()
    br i1 %result, label %produce_tuple, label %loop_head
produce_tuple:
    %status = call i32 @llvm.parent.consume()
    switch i32 %status, label %break [
        i32 0, label %loop
        i32 1, label %finalize
    ]
finalize:
    %status2 = call i32 @llvm.parent.finalize()
    ret i32 %status2
break:
    %status3 = sub i32 %status, 1
    ret i32 %status3
}
```

3.4 Динамическая компиляция выражений в СУБД PostgreSQL

В СУБД PostgreSQL выражение — это дерево узлов типа `Expr`. Существуют подклассы `Expr` для различных типов выражений: вызовов функций, вызовов операторов, констант, переменных и т.д. Для вычисления выражений этих подклассов вызываются соответствующие функции `Exec`. Семейство функций `Exec` вместе образуют интерпретатор выражений PostgreSQL, который обходит дерево выражений, где узлы операторов содержат указатель на функцию, которая при вызове рекурсивно вычисляет подвыражения, а также сам оператор. Листья этого дерева представляют литералы (например, `ExecEvalConst`), строковые переменные (например, `ExecScalarVar`) или обращения к столбцам (`slot_getattr`). Хотя такой интерпретатор выражений широко используется в современных СУБД, он считается [43; 78] ресурсоемким и расточительным для современных вычислительных систем и архитектур памяти.

На рисунке 3.8 слева представлен пример дерева для выражения $a * b > 10$, где a и b — это атрибуты таблицы, а справа — порядок вызова функций PostgreSQL соответствующих вершин для вычисления результата выражения. Функции `int8pl` и `int8gt` являются встроенными функциями СУБД PostgreSQL.




```
ExecQual() {
   gt = indirect call ExecEvalFunc() {
      pl = indirect call ExecEvalFunc() {
         a = indirect call ExecEvalVar();
         b = indirect call ExecEvalVar();
         return int8pl(a, b);
      }
      const = indirect call ExecEvalConst(10);
      return int8gt(pl, const);
   }
   return gt;
}
```

Рисунок 3.8 — Слева – интерпретируемое дерево выражений, справа – цепочка вызовов функций СУБД PostgreSQL соответствующих вершин.

Для заданного выражения во время выполнения запроса интерпретатор PostgreSQL будет многократно обходить дерево и вызывать одни и те же функции `Exec` в том же порядке. Задача динамической компиляции выражений

состоит в том, чтобы превратить эти повторяющиеся усилия во время выполнения в одноразовую задачу времени компиляции.

Поскольку во время выполнения доступна информация о вызываемых функциях и операциях, можно использовать кодогенерацию для замены неявных вызовов функций на явные, которые в дальнейшем могут быть встроены. В динамическом компиляторе запросов была реализована динамическая компиляция выражений и предикатов, используемых в запросе. Динамическая компиляция выражений выполнена путём рекурсивного обхода дерева выражений в обратном порядке и генерацией, с использованием LLVM C API, соответствующего кода для каждой функции или операции во внутреннем представлении LLVM IR. Таким образом код для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без расходов на неявный вызов функций. На листинге 3.22 представлен результат динамической кодогенерации в виде функции во внутреннем представлении LLVM IR для интерпретируемого выражения из рисунка 3.8.

Листинг 3.22: Сгенерированный код на LLVM IR для выражения $a * b > 10$.

```
define internal i1 @llvm.ExecQual() {
entry:
  %a.attr.value = load i64, i64* %a.attr
  %b.attr.value = load i64, i64* %b.attr

  %ml = mul i64 %a.attr.value, %b.attr.value
  %cmp = icmp sgt i64 %ml, 10

  ret %cmp
}
```

3.4.1 Метод предкомпиляции встроенных функций СУБД PostgreSQL

Для получения кода встроенных функций PostgreSQL во внутреннем представлении LLVM IR был разработан метод предварительной компиляции, 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 биткод LLVM (*.bc*) с помощью компилятора `clang`, а затем компонуются линковщиком `llvm-link` в единый биткод файл (`backend.bc`). Последний оптимизируется оптимизатором LLVM `opt`. Далее, во время старта сервера СУБД PostgreSQL, оптимизированный биткод файл `backend.opt.bc` загружается в виде LLVM модуля, содержащего все встроенные функции PostgreSQL. В процессе генерации кода выражения выполняется вызов сгенерированных встроенных функций из полученного модуля, где все функции помечены атрибутом *always_inline*. После генерации кода запроса выполняется оптимизация по встраиванию функций, которая вставляет код функции в тело кода выражений. Схема работы метода предкомпиляции показана на рисунке 3.9.

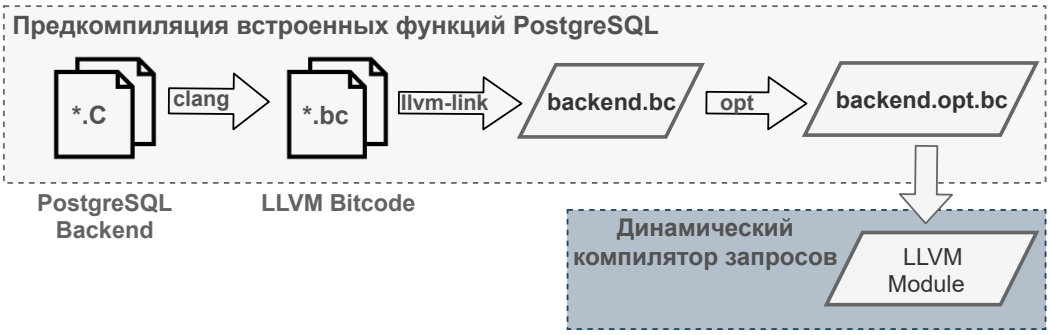


Рисунок 3.9 — Схема метода предкомпиляции встроенных функций PostgreSQL.

На листинге 3.23 показан пример предварительно скомпилированной в LLVM IR встроенной функции `int8pl` (сложение двух целых чисел) из файла `int8.c`, который является частью исходного кода бекенда СУБД PostgreSQL.

Преимуществами метода предварительной компиляции являются простота и универсальность реализации, упрощённая поддержка, поскольку отпадает необходимость в ручной реализации с использованием LLVM C API генератора каждой встроенной функции и отслеживании изменений в коде СУБД PostgreSQL (биткод файл можно быстро перекомпилировать в случае изменений в исходном коде).

Листинг 3.23 — Сверху — исходный код на языке C встроенной функции `int8pl` СУБД PostgreSQL. Снизу — код функции `int8pl` во внутреннем представлении LLVM IR.

```
Datum
int8pl(FunctionCallInfo fcinfo)
{
    int64    arg1 = fcinfo->arg[0];
    int64    arg2 = fcinfo->arg[1];
    int64    result;

    result = arg1 + arg2;

    /*
    * Overflow check.
    */
    if (SAMESIGN(arg1, arg2) && !SAMESIGN(result, arg1))
        ereport(ERROR, (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),
            errmsg("bigint out of range")));
    PG_RETURN_INT64(result);
}
```

```
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
    %1 = getelementptr %struct.FunctionCallInfoData,
    %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
    %2 = load i64, i64* %1
    %3 = getelementptr %struct.FunctionCallInfoData,
    %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
    %4 = load i64, i64* %3
    %5 = add nsw i64 %4, %2
    %.lobit = lshr i64 %2, 63
    %.lobit1 = lshr i64 %4, 63
    %6 = icmp ne i64 %.lobit, %.lobit1
    %.lobit2 = lshr i64 %5, 31
    %7 = icmp eq i64 %.lobit2, %.lobit
    %or.cond = or i1 %6, %7
    br i1 %or.cond, label %exit, label %overflow
overflow:
    call void @ereport(..., "bigint out of range")
exit:
    ret i64 %5
}
```

3.4.2 Оптимизация доступа к атрибутам

В СУБД PostgreSQL процесс десериализации атрибутов предполагает последовательный обход кортежа как байтового массива и восстановление атрибутов одного за другим. В механизме частичной десериализации для каждого следующего атрибута этот обход запускается, начиная с позиции последнего восстановленного атрибута, что позволяет частично избежать восстановления не используемых в запросе атрибутов. Неэффективность процедуры доступа к атрибутам кортежа является прямым следствием того, насколько компактно таблицы представляются с помощью heap-файлов. Размер кортежа и расположение атрибутов могут варьироваться в зависимости от хранимых данных:

- значения NULL хранятся в битовой маске в заголовке кортежа, а при сериализации атрибутов кортежа пропускаются;
- атрибуты переменной длины (varlena — variable-length attributes; например, строки) могут храниться как в самом кортеже, так и вне его;
- атрибутов может оказаться меньше, чем колонок в таблице, — в таком случае значения остальных атрибутов считаются равными NULL.

Таким образом, в общем случае смещение атрибута с порядковым номером N определяется наличием NULL-атрибутов и длиной атрибутов переменной длины среди атрибутов с номерами от 1 до $N-1$ и потому требует линейного обхода по всем атрибутам с меньшими номерами.

По итогам анализ алгоритма доступа к атрибутам извлекаемого кортежа в PostgreSQL были выделены следующие возможности для оптимизации последовательного доступа к атрибутам возможных при динамической компиляции кода под конкретный запрос.

1. Заголовок таблицы содержит свойства *attnotnull* (атрибуты, у которых это свойство не установлено, называются *nullable* и могут принимать значения NULL) и *attlen* (длина атрибута, отрицательна для атрибутов переменной длины) для каждого атрибута. Это позволяет вычислить заранее смещения первых N атрибутов, где $N = max\{N \geqslant 1 | \forall n < N : attnotnull(n) \wedge attlen(n) > 0\}$, т.е. которые не могут принимать значение NULL и имеют фиксированную длину.
2. Несмотря на то, что, начиная с атрибута, следующего за первым *nullable* или атрибутом переменной длины, смещения атрибутов фикси-

рованными не являются, длину всякой последовательности не *nullable* атрибутов фиксированной длины можно вычислить заранее, что позволяет пропускать такие последовательности, состоящие из атрибутов, не используемых в предикате.

3. Зная заранее, какие атрибуты той или иной таблицы используются в запросе, а какие нет, можно извлекать только используемые атрибуты при первом считывании кортежа и отказаться от затратного механизма ленивой десериалиации.

Основная идея заключается в том, что в момент выполнения запроса, в базе данных присутствует информация о типах колонок таблиц, их длине и поддержке значения NULL. Данная метаинформация может быть использована во время динамической компиляции для предварительного расчёта смещения атрибутов внутри кортежа, что позволит уменьшить количество ветвлений и обрабатывать только необходимые для данного запроса атрибуты.

Рассмотрим пример таблицы с 12 атрибутами, из которых атрибуты 5, 8 и 11 являются *nullable*, то есть допускают NULL значение или имеют переменную длину. Эти атрибуты помечаются символами N/V. Допустим в запрос используются атрибуты 3 и 8. На рисунке 3.10 представлена используемая таблица и предполагаемый к ней запрос.

```
SELECT at3, at8 FROM table;
```

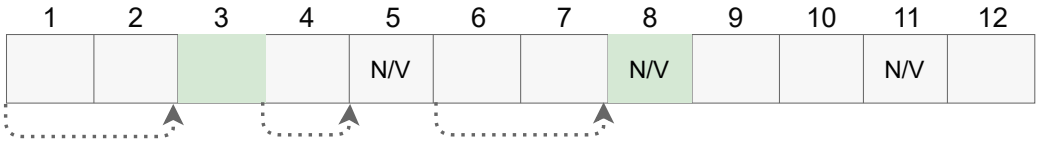


Рисунок 3.10 — Пример применения оптимизации доступа к атрибутам кортежа.

Согласно оптимизации (1) смещение атрибута 3 вычисляется на этапе компиляции, согласно оптимизации (2) длина атрибутов 4, 6–7 также вычисляются на этапе компиляции, и согласно оптимизации (3) атрибуты 9–12 не считываются и десериализация происходит не во время вычисления выражения, а во время загрузки кортежа. Таким образом, при обработке кортежа требуется прочитать всего три атрибута: 3, 5 и 8. Для сравнения, оригинальный алгоритм PostgreSQL выполнит чтение первых восьми атрибутов за два вызова функции *slot_getattr()*: первый вызов считывает атрибуты 1–3, второй — 4–8.

3.5 Реализация эвристик стратегии выполнения запроса в динамическом компиляторе для СУБД PostgreSQL

Для реализация эвристик стратегии выполнения в динамическом компиляторе запросов для СУБД PostgreSQL был использован описанный в разделе 2.3 конечный автомат состояний, который реализован в виде конструкции *switch-case* по состояниям автомата. В ходе реализации этот автомат состояний был расширен тремя дополнительными состояниями: `ESTIMATED_CODEGEN_OVERHEAD`, `ACCUMULATE_COMPILATION_COST` и `CODEGEN_ERROR`. Расширенный автомат состояний представлен на рисунке 3.11. Далее мы опишем основные дополнения в реализованном автомате состояний.

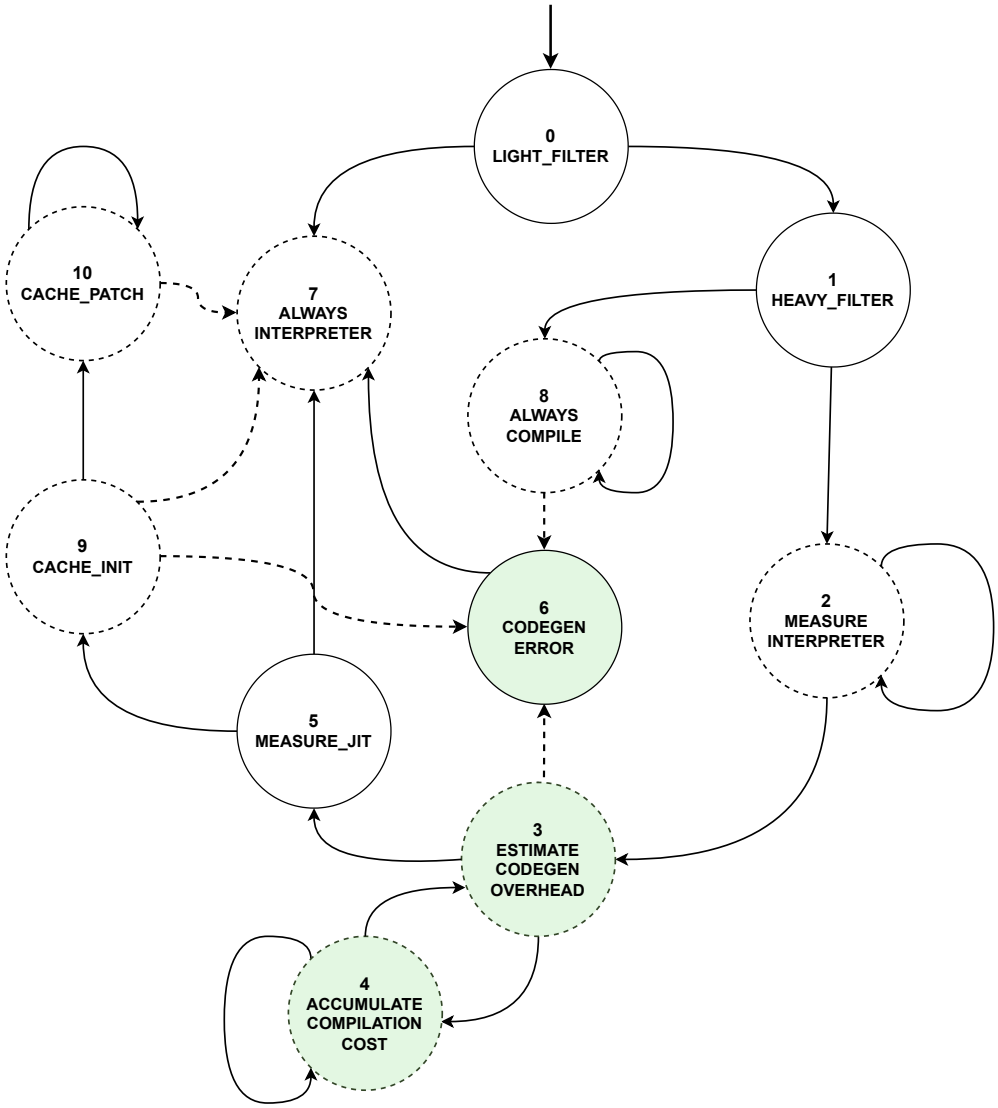


Рисунок 3.11 — Конечный автомат состояний эвристик стратегии выполнения запроса в динамическом компиляторе.

В состоянии 2 — `MEASURE_INTERPRETER` измеряется время выполнения запроса интерпретатором T_I два раза и берётся минимум в качестве базового значения: $T_I = min(T_{I_1}, T_{I_2})$. Как правило, первое выполнение происходит медленнее по многим причинам (в основном, из-за холодного кэша данных), поэтому требуется одно дополнительное выполнение.

После измерения времени интерпретации, выполнение запроса переходит в новое состояние 3 — `ESTIMATED_CODEGEN_OVERHEAD`, в котором мы можем дополнительно оценить время затраченное на компиляцию кода T_C, используя линейную регрессию для нахождения зависимости количества инструкций в сгенерированном LLVM модуле конкретного запроса от времени компиляции этого модуля. Для разработки регрессионной модели мы собрали образцы данных с помощью регрессионных тестов PostgreSQL, тестов TPC-H и TPC-DS. Каждая выборка данных представляет собой пару (M_{inst}, M_{comp}), где M_{inst} — количество инструкций в модуле, M_{comp} — время компиляции модуля. Смысл этой эвристики заключается в том, что в некоторых случаях время затрачиваемое на компиляцию кода может значительно превышать время интерпретации запроса, поэтому нам нужна какая-то гарантия того, что эти накладные расходы не будут избыточными и окупятся. Следовательно, мы оцениваем время компиляции и интерпретируем запрос $N = \lceil \frac{T_C}{T_I} \rceil$ раз. Последнее соответствует состоянию 4 — `ACCUMULATE_COMPILATION_COST`.

Состояние 6 — `CODEGEN_ERROR` служит для отката состояния динамического компилятора и возврат к выполнению запроса интерпретатором в случае возникновения ошибки на каком либо из этапов компиляции запроса. Откат к интерпретатору реализован с помощью имеющегося в СУБД PostgreSQL механизма перехвата ошибок PG_TRY()/PG_CATCH().

В состоянии 5 — `MEASURE_JIT`, помимо измерения времени выполнения скомпилированного запроса, дополнительно оцениваем снижение производительности времени выполнения, связанное с кэшированием запросов: $T_E * K_D > T_C$, умножив время выполнения динамически скомпилированного кода T_E на некоторый коэффициент деградации K_D и сравнив его с сохраненным временем T_C, затраченным на компиляцию и оптимизацию. Смысл этой эвристики заключается в том, что для кэширования и повторного использования кода запроса мы вводится дополнительный уровень косвенности, который позволяет обновлять адреса структур данных PostgreSQL, от которых зависит скомпилированный код запроса. Такая косвенность может ухудшить производительность

запросов до 10%, но в среднем ухудшение составляет порядка 5%. В случае долго выполняющихся запросов эта деградация может быть намного выше, чем время затраченное на компиляцию и выполнение запроса, поэтому эффективнее компилировать такие запросы при каждом выполнении. В следующем разделе описана реализация метода кэширования кода запроса в динамическом компиляторе СУБД PostgreSQL, где подробно рассмотрена косвенность из-за которой ухудшается производительность выполнения запросов.

3.5.1 Реализация механизма кэширования динамически скомпилированного кода запросов

Для нивелирования накладных расходов при динамической компиляции и оптимизации сгенерированного кода запроса был реализован метод кэширования кода. На рисунке 3.12 представлена схема работы метода кэширования динамически скомпилированного кода запросов. В качестве сохраняемого кода был выбран объектный код запроса.

Во время первого выполнения запроса, наряду с сохранением объектного кода, мы сохраняем адрес глобального массива адресов, который содержит адреса переменных и структур PostgreSQL, использующиеся в сгенерированном динамическим компилятором запросов коде, поскольку значения переменных и адреса структур изменяются в динамической памяти при каждом выполнении запроса. Таким образом, перед началом следующего выполнения сохранённого кода запроса необходимо обновить ранее используемые абсолютные адреса. После второго и последующих выполнений того же запроса мы загружаем сохранённый объектный код и адрес глобального массива адресов и обновляем адреса в этом массиве путём обхода дерева плана запроса так, как это было сделано при первом выполнении этого запроса, но без фактической генерации кода. Таким образом, единственное различие между сохраненным кодом и кодом, созданным для однократного выполнения (без кэширования), заключается в том, что сохранённый код использует дополнительную *косвенность*, загружая требуемый адрес из определённой ячейки глобального массива адресов, что может незначительно повлиять на производительности, но при этом переиспользова-

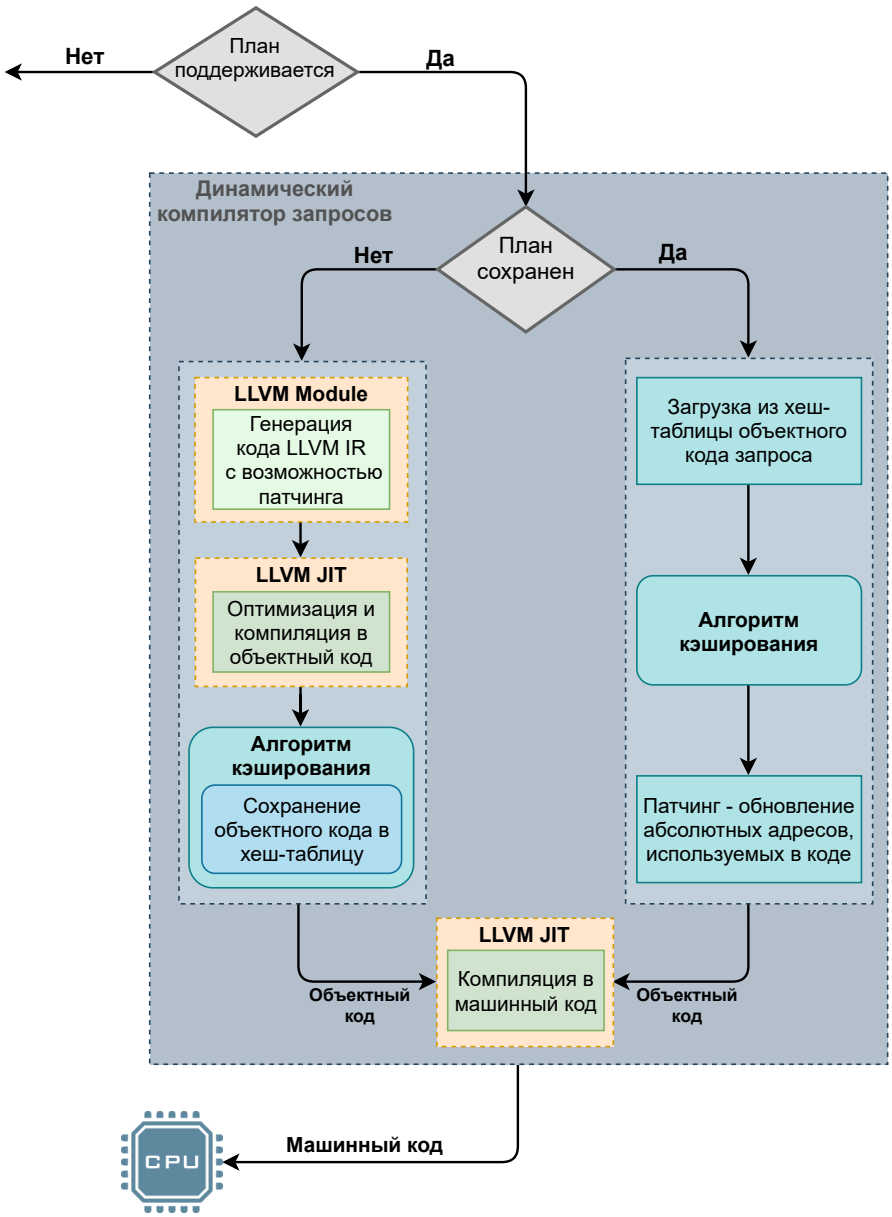


Рисунок 3.12 — Схема работы метода кэширования скомпилированного кода запроса в динамическом компиляторе запросов.

ние сохранённого кода запроса избавляет от накладных расходов, связанных с этапами генерации и компиляции кода запроса.

В конструкции динамического компилятора запросов с моделью явных циклов генерация кода выполняется во время обхода дерева плана в прямом порядке, во время которого для каждого оператора вызываются функции-генераторы. Для каждого оператора соответствующие функции-генераторы реализованы с использованием LLVM C API и вызываются для генерации реализующего его алгебраическую модель кода на LLVM IR.

При реализации метода кэширования динамически скомпилированного кода запроса важно было избежать дублирования логики и сохранить существу-

ющий алгоритм кодогенерации в функциях-генераторах с целью упрощения дальнейшего процесса разработки и поддержки. Однако простого сохранения динамически сгенерированного кода запроса в общем случае недостаточно, так как в сгенерированном коде используются переменные и структуры PostgreSQL, абсолютные адреса которых меняются после каждого выполнения плана запроса и, соответственно, нужен механизм, позволяющий выполнять кодогенерацию с возможностью обновления адресов в сохранённом коде.

Для реализации были выполнены следующие изменения в существующем алгоритме кодогенерации:

- Все использующиеся для кодогенерации функции из LLVM C API обёрнуты в функцию-обёртку с аналогичным названием и дополнительным префиксом, которая возвращает разный результат в зависимости от глобального режима кодогенерации. Пример функции-обёртки представлен на листинге 3.24.
- Добавлен глобальный режим генерации кода — глобальная переменная *llvm_patchpoint*, которая задаёт поведение внутри функций-обёрток над LLVM C API.
- Все вызовы функции LLVM C API, которая генерирует доступ к переменным PostgreSQL по адресу (*LLVMConstIntToPtr*), заменены на специальную функцию *GeneratePatchpoint*, код которой представлен на листинге 3.25.

Листинг 3.24: Пример функции-обертки над LLVM C API в динамическом компиляторе запросов СУБД PostgreSQL.

```
static LLVMValueRef inline
__LLVMBuildStore(LLVMBuilderRef B,LLVMValueRef Val,LLVMValueRef Ptr)
{
    Assert(llvm_patchpoint >= 0 && llvm_patchpoint < 3);
    if (llvm_patchpoint == 1)
    {
        Assert(B == NULL);
        Assert(Val == NULL);
        Assert(Ptr == NULL);
        return NULL;
    }
    else
        return LLVMBuildStore(B, Val, Ptr);
}
```

Перечислим возможные режимы кодогенерации:

1. Одноразовая кодогенерация (*llvm_patchpoint=0*): динамически скомпилированный код запроса будет использован один раз.
2. Патчинг (*llvm_patchpoint=1*): кодогенерация не выполняется, а производится сбор и обновление абсолютных адресов структур данных PostgreSQL, используемых в коде. Адреса сохраняются в глобальный массив.
3. Кодогенерация с возможностью патчинга (*llvm_patchpoint=2*): в ходе кодогенерации сохраняются абсолютные адреса структур данных PostgreSQL, используемых в коде, как в режиме патчинга и обновляется глобальный массива адресов.

Листинг 3.25: Исходный код функции GeneratePatchpoint в динамическом компиляторе запросов СУБД PostgreSQL.

```
LLVMValueRef
GeneratePatchpoint(LLVMTypeRef type, void *address)
{
    /* Генерация доступа к переменной по адресу */
    if (llvm_patchpoint == 0)
        return __LLVMConstIntToPtr(address, type);

    /* Сохранение нового адреса в глобальный массив адресов */
    llvm_pp[llvm_pp_n] = (uintptr_t)address;

    /* Генерация кода загрузки из глобального массива адресов */
    if (llvm_patchpoint == 2)
    {
        LLVMValueRef vals[1], val_off, val;
        LLVMValueRef llvm_pp_ptr = LLVMGetNamedGlobal("llvm_pp");
        LLVMValueRef llvm_pp_ = __LLVMBuildLoad(llvm_pp_ptr);
        vals[0] = llvm_int64_const(llvm_pp_n);
        val_off = __LLVMBuildGEP(llvm_pp_, vals, 1);
        val = __LLVMBuildLoad(val_off);
        llvm_pp_n++;
        return val;
    }

    llvm_pp_n++;
    return NULL;
}
```

В режиме одноразовой кодогенерации (*llvm_patchpoint=0*) функции обёртки над функциями LLVM C API возвращают результат вызова соответствующей LLVM функции, а *GeneratePatchpoint()* генерирует код для доступа

по адресу к переменным PostgreSQL без запоминания переданного адреса-аргумента (см. листинг 3.25).

В режиме патчинга (*llvm_patchpoint=1*) выполняется обход плана запроса с использованием функций-генераторов, в ходе которого все вызовы функций-обёрток над LLVM C API возвращают пустое значение, т.е. отсутствуют какая-либо кодогенерация. Для сохранения корректности обход выполняется в том же порядке в каком выполнялся для сохранённого кода. Единственная выполняемая работа в функции *GeneratePatchpoint* — это сбор новых адресов, путём сохранения их в глобальный массив для обновление старых адресов.

В режиме кодогенерации с возможностью патчинга (*llvm_patchpoint=2*) выполняется обход плана запроса и генерация кода для него, где функция *GeneratePatchpoint* записывает адрес-аргумент в глобальный массив адресов и генерируется код загрузки значений из этого массива по смещению. После генерации кода плана запроса на LLVM IR выполняется компиляция в объектный код и сохранение его в структуру кэша.

В качестве структуры кэша была выбрана структура данных хеш-таблица, реализация которой имеется в СУБД PostgreSQL. Для управления кэшем был реализован описанный в разделе 2.3.1 модифицированный алгоритм вытеснения кэша LRU. Ключом элемента хеш-таблицы является контрольная сумма от текстового представления физического плана запроса, но без учёта используемых в ней констант. Таким образом один и тот же сохранённый код может быть использован для множества однотипных запросов, различающихся лишь значениями константных параметров, используемых в запросе. В текстового представления плана запроса также хранится информация о схеме используемых таблиц, типе атрибутов и т.д., таким образов сохраняется корректность сохранённого кода запроса относительно плана запроса в случае, если, например, изменяется схема таблиц в запросе, но при этом сам запрос останется прежним, то сохранённый код для этого запроса не будет загружен.

3.6 Выводы

Предложенный в главе 2 метод динамической компиляции запросов был применён к исполнителю запросов СУБД с открытым исходным кодом

PostgreSQL. Реализован динамический компилятор запросов в виде программного расширения к СУБД с использованием компиляторной инфраструктуры LLVM.

В процессе сборки расширения для динамической компиляции запросов выполняется предварительная компиляция, оптимизация и компоновка встроенных функций СУБД 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 PostgreSQL, необходимых при динамической компиляции.

Перед началом выполнения запроса динамическим компилятором производится проверка поддержки всех выражений и операторов, используемых в плане запроса. Далее выполняется обход дерева плана запроса и генерация семантически эквивалентного внутреннего представления LLVM IR исходного плана запроса с использованием программного интерфейса LLVM C API. Во время генерации для каждого оператора вызываются функции-генераторы декомпозированого на интерфейс в модели явных циклов алгоритма оператора 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. Сгенерированный код после оптимизаций будет состоять из нескольких циклов, самым первым из которых — цикл одного из операторов сканирования. Далее сгенерированный код компилируется модулем LLVM для динамической компиляции MCJIT или ORC JIT и запускается на выполнение. Реализованы с использованием LLVM C API генераторы для большинства операторов СУБД PostgreSQL.

В динамическом компиляторе запросов реализована компиляция выражений 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.

Реализация в динамическом компиляторе запросов эвристик стратегии выполнения и механизма кэширования динамически скомпилированного кода запроса позволила уменьшить накладные расходы на компиляцию и установить стратегии выполнения для запросов разной сложность.

Глава 4. Тестирование и анализ результатов

Динамический компилятор запросов реализован в качестве программного расширения для СУБД PostgreSQL версии 9.6.3. В качестве JIT-компилятора был использован модуль ORC JIT из компиляторной инфраструктуры LLVM версии 9.0.1. В таблице 2 представлен список реализованных операторов СУБД PostgreSQL в динамическом компиляторе запросов.

Таблица 2 — Перечень поддерживаемых операторов в динамическом компиляторе запросов для СУБД PostgreSQL.

– Qual & Project	– NestLoop (Inner, Anti, Semi)	– Sort
– SeqScan	– HashJoin (Inner, Right, Semi)	– Material
– IndexScan	– MergeJoin (Inner, Semi)	– Result
– IndexOnlyScan	– Subplan	– Limit
– SubqueryScan	– Hash	– Bitmap (OR,AND)
– BitmapHeapScan	– Agg (Plain, Hash, Sort)	
– BitmapIndexScan	– WindowAgg	– Group
– CteScan	– Append	– Unique

Для оценки эффективности метода динамической компиляции запросов с применением модели явных циклов выполнения было произведено сравнительное тестирование на запросах из промышленных тестовых наборов TPC-H [29] и TPC-DS [79]. Тестовый набор TPC-H состоит из 22-х преимущественно бизнес-ориентированных аналитических SQL-запросов, где существенная часть запросов из набора обрабатывает большой объём данных в процессе вычисления конечного результата. Тестовый набор TPC-DS состоит из 99 запросов, содержащих как аналитические запросы, так и запросы, использующиеся для поддержки процесса принятия решений. Оба тестовых набора считаются достаточно близкими к реальным запросам к СУБД.

Для генерации базы данных использовались адаптированные под СУБД PostgreSQL программы *DBGen* и *QGen*, оригиналы которых доступны на официальном сайте TPC [80]. Программа *DBGen* позволяет установить параметр масштаба (SF — Scale Factor) для генерации базы нужного размера. Размер

базы данных определяется с учётом коэффициента 1 (т.е. SF = 1; примерно 1ГБ), минимально необходимого размера для тестовой базы данных. Данный коэффициент влияет практически на все таблицы, за исключением некоторых справочников. На рисунке 4.1 представлена схема базы данных TPC-H, где над каждой таблицей указано либо число, либо операция умножения, которая показывает влияние параметра масштаба на конечное количество записей.

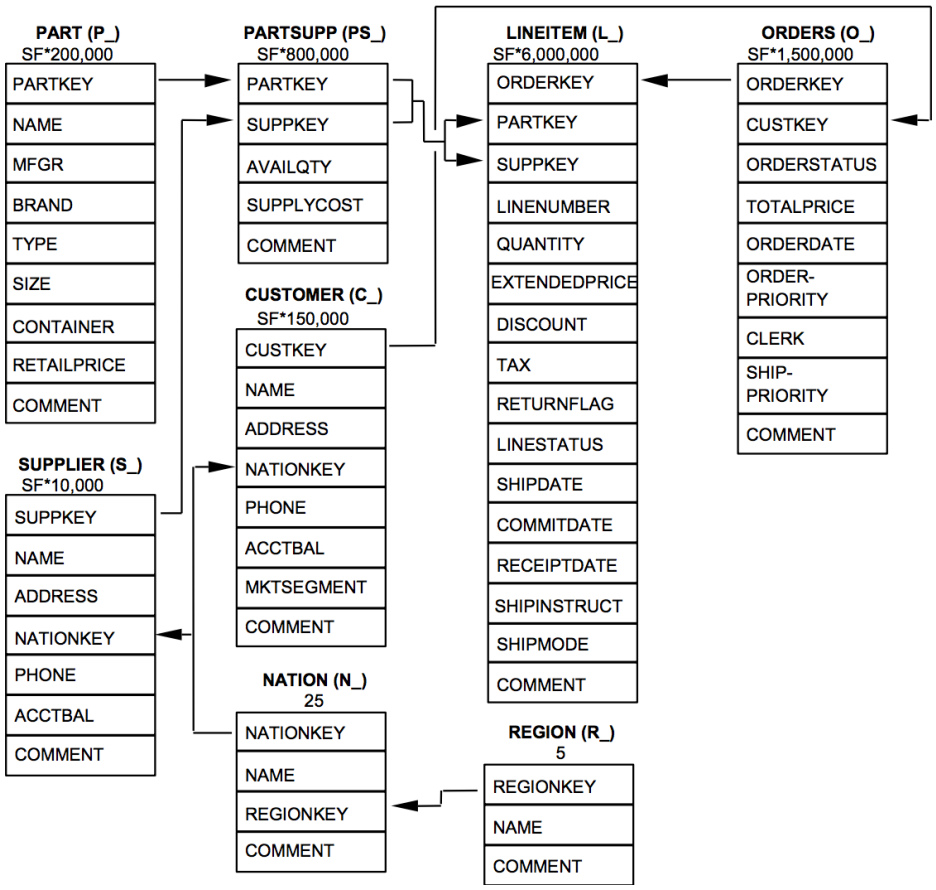


Рисунок 4.1 — Схема базы данных TPC-H.

Тестирование производительности выполнялась на платформах x86-64 и ARM (aarch64):

– x86-64 — сервер с процессором Intel Xeon E5-2699 v3, 72Core @ 2,3 ГГц с 256 ГБ ОЗУ (8*32ГБ DDR4 2133MHz) под управлением 64-битной операционной системы openSUSE Leap 42.3.
– ARM — сервер с процессором Kunpeng 920, 2*48Core @ 2,6 ГГц с 256 ГБ ОЗУ (8*32ГБ DDR4 2933MHz) под управлением 64-битной операционной системы CentOS Linux 8.1.1911.

В качестве параметра масштаба SF базы данных для TPC-H были выбраны числа 75, 20 и 10, для TPC-DS — 50, 20 (отдельные запросы были протестированы на масштабах 10 и 1).

Тестовый набор TPC-H имеет два режима генерации данных: с использованием оригинальных типов СУБД (NUMERIC, CHAR(1)) в колонках и нативных типов данных (DOUBLE вместо NUMERIC, ENUM вместо CHAR(1)). Далее в таблицах с результатами TPC-H используемый режим генерации данных по типу отмечен в колонке `Тип` соответственно `original` или `native`.

Каждый запрос из наборов TPC-H и TPC-DS запускался по 10 раз. Переход к выполнению следующего запроса сопровождался перезапуском СУБД. Итоговое значение для каждого запроса вычислялось методом подсчёта медианы полученных результатов.

В таблицах 3 и 4 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 TPC-H с помощью интерпретатора PostgreSQL, динамического компилятора и динамического компилятора с кэшированием кода запроса соответственно. Ускорение отражает на сколько процентов ускорилось время выполнения динамически скомпилированного запроса относительно интерпретации (колонки $\frac{PG}{JIT}$ и $\frac{PG}{JIT+\text{кэш}}$).

Таблица 3 — 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

Время динамической компиляции без кэширования кода запроса также включает накладные расходы на оптимизацию и компиляцию в машинный код. На запросах из тестового набора TPC-H суммарное время оптимизации и компиляции составляет от 0.4 до 2.2 секунд в зависимости от сложности запроса.

Как можно заметить из результатов тестирования среднее ускорение от динамической компиляции без кэширования кода лучше на большом объёме данных. А использование метода кэширования кода в динамическом компиляторе позволяет компенсировать ухудшение производительности на небольшом

Таблица 4 — 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	$\frac{PG}{JIT}$,%	$\frac{PG}{JIT+кэш}$,%	$\frac{JIT}{JIT+кэш}$,%
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

объёме данных за счёт экономии на времени компиляции. Также 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 таблицах 5 и 6 отражены результаты тестирования запросов из набора TPC-DS на платформах x86-64 и ARM. Также как и для TPC-H в колоках `PG`, `JIT` и `JIT+кэш` указано среднее время выполнения всех запросов TPC-DS.

Таблица 5 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-DS на платформе x86-64.

TPC-DS x86-64

Масштаб	PG,сек	JIT,сек	JIT+кэш,сек	$\frac{PG}{JIT}$,%	$\frac{PG}{JIT+кэш}$,%	$\frac{JIT}{JIT+кэш}$,%
50	62.50	49.41	47.58	26.50	31.35	3.83
20	27.47	20.75	18.67	32.36	47.17	11.19

Таблица 6 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-DS на платформе ARM.

TPC-DS ARM

Масштаб	PG,сек	JIT,сек	JIT+кэш,сек	$\frac{PG}{JIT}$,%	$\frac{PG}{JIT+кэш}$,%	$\frac{JIT}{JIT+кэш}$,%
50	87.94	62.22	59.44	41.33	47.94	4.68
20	36.05	26.62	23.32	35.42	54.59	14.15

Для лучшей демонстрации полученных результатов и оценки влияния метода динамической компиляции на отдельные запросы в таблице 7 приведены результаты выполнения всех запросов из набора TPC-H на платформе x86-64 при масштабе 75 с использованием нативных типов данных. Здесь ускорение вычислялось путём деления времени выполнения интерпретатора на время выполнения динамически скомпилированного запроса и показывает во сколько раз ускорился запрос (колонки $\frac{PG}{JIT+\text{кэш}}$ и $\frac{PG}{JIT}$). Результаты тестирования показывают, что метод динамической компиляции позволяет значительно (**до 5 раз**) увеличить производительность СУБД на запросах, скорость обработки которых в первую очередь определяется эффективностью использования процессора.

Таблица 7 — 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 можно увидеть из результатов тестирования, 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 7, но в среднем ухудшение производительности составляет 5%. Поэтому в некоторых случаях может быть более эффективным всегда компилировать запрос. Разработанные эвристики устанавливают баланс между стратегиями «всегда кэшировать» и «всегда компилировать», поэтому динамическая компиляция не применяется для очень простых запросов, кэширование используется для средних запросов, а долго выполняющиеся запросы всегда компилируются.

Для тестирования разработанных эвристик стратегии выполнения в динамическом компиляторе были выбраны тестовые наборы TPC-DS в масштабе 50 и TPC-H (с использованием нативных типов) в масштабах 75 и 20. Тестирования проводилось на платформе x86-64. Результаты показаны в таблице 8. Применение эвристик позволило получить лучший результат.

Таблица 8 — Сравнение времени выполнения интерпретатора PostgreSQL и динамического компилятора с использованием эвристик кэширования на тестовых наборах TPC-DS и TPC-H на платформе x86-64.

x86-64	TPC-DS 50	TPC-H 75	TPC-H 20
PG, сек	62.50	108.76	18.71
JIT, сек	49.41	80.14	12.63
JIT+кэш, сек	47.58	80.31	11.99
JIT+кэш+эвр, сек	46.49	79.61	11.70
PG/JIT, %	26.49	35.71	48.14
PG/JIT+кэш, %	31.36	35.43	56.05
PG/JIT+кэш+эвр, %	34.44	36.62	59.91

Полные результаты тестирования динамического компилятора на запросах из тестового наборов TPC-H и TPC-DS приведены в приложении A.

В таблице 16 из приложения A показаны результаты тестирования динамического компилятора запросов на тестовом наборе TPC-H в масштабе 75

на платформе ARM с использованием режима *параллельного выполнения запросов* в СУБД PostgreSQL. Тестирование проводилось на 8 параллельных рабочих процессах (ядрах процессора). Полученные результаты показывают, что при распараллеливании запросов ускорение от динамической компиляции для многих запроса из тестового набора TPC-H пропорционально количеству используемых ядер процессора. Однако для некоторых запросов параллельное выполнение не даёт особого выигрыша. Это может быть связано с ограничениями текущей реализации параллельного выполнения запросов в PostgreSQL, либо с невозможностью построить параллельный план, который был бы быстрее последовательного, либо, в случае с динамическим компилятором, с накладными расходами на компиляцию и оптимизацию кода запросов. Например, на запросе $Q1$ использование параллельного выполнения даёт ускорение в **34 раза** по сравнению с интерпретатором PostgreSQL без параллелизма (колонка $\frac{PG}{JIT_{\text{пар.}}}$), но в **4.5 раза** по сравнению с PostgreSQL с параллелизмом (колонка $\frac{PG_{\text{пар.}}}{JIT_{\text{пар.}}}$).

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

Заключение

Основные результаты работы заключаются в следующем.

1. Разработан метод динамической компиляции запросов с трансформаций 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. 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.

В качестве дальнейших направлений исследования по тематике данной работы перспективным направлением является реализация векторного исполнителя совместно с динамическим компилятором с возможностью применение аппаратных векторных инструкций (SIMD) для ускорения выполнения запросов. Также перспективным является разработка и реализация методов для построения динамических компиляторов SQL-запросов с использованием предметно-ориентированных языков.

Список литературы

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.
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.
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.
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.
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.
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.
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.
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).
9. *Кузнецов, С. Д.* Основы современных баз данных [Электронный ресурс] [Текст] / С. Д. Кузнецов // Информационно-аналитические ма-

териалы Центра Информационных Технологий. —. — Режим доступа: http://citforum.ru/database/osbd/contents.shtml.

10. A History and Evaluation of System R [Текст] / D. D. Chamberlin [и др.] // Commun. ACM. — New York, NY, USA, 1981. — Окт. — Т. 24, № 10. — С. 632—646. — URL: https://doi.org/10.1145/358769.358784.

11. *Wade, B. W.* Compiling SQL into System/370 Machine Language [Текст] / B. W. Wade // IEEE Annals of the History of Computing. — 2012. — Т. 34, № 4. — С. 49—50.

12. *Greer, R.* Daytona and the Fourth-Generation Language Cymbal [Текст] / R. Greer // Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data. — Philadelphia, Pennsylvania, USA : Association for Computing Machinery, 1999. — С. 525—526. — (SIGMOD '99). — URL: https://doi.org/10.1145/304182.304242.

13. *Graefe, G.* Encapsulation of Parallelism in the Volcano Query Processing System [Текст] / G. Graefe // SIGMOD Rec. — New York, NY, USA, 1990. — Май. — Т. 19, № 2. — С. 102—111. — URL: https://doi.org/10.1145/93605.98720.

14. *Graefe, G.* Volcano/spl minus/an extensible and parallel query evaluation system [Текст] / G. Graefe // IEEE Transactions on Knowledge and Data Engineering. — 1994. — Т. 6, № 1. — С. 120—135.

15. *Manegold, S.* Database Architecture Evolution: Mammals Flourished Long before Dinosaurs Became Extinct [Текст] / S. Manegold, M. L. Kersten, P. Boncz // Proc. VLDB Endow. — 2009. — Авг. — Т. 2, № 2. — С. 1648—1653. — URL: https://doi.org/10.14778/1687553.1687618.

16. Block oriented processing of relational database operations in modern computer architectures [Текст] / S. Padmanabhan [и др.] // Proceedings 17th International Conference on Data Engineering. — 2001. — С. 567—574.

17. *Neumann, T.* Efficiently Compiling Efficient Query Plans for Modern Hardware [Текст] / T. Neumann // Proc. VLDB Endow. — 2011. — Июнь. — Т. 4, № 9. — С. 539—550. — URL: https://doi.org/10.14778/2002938.2002940.

18. *Neumann, T.* Compiling Database Queries into Machine Code [Текст] / T. Neumann, V. Leis // IEEE Data Eng. Bull. — 2014. — Т. 37. — С. 3—11.

19. *Lorie, R. A.* XRM - An Extended (N-ary) Relational Memory [Текст] / R. A. Lorie // IBM Research Report. — 1974. — Т. G320—2096.

20. *Stonebraker, M.* The VoltDB Main Memory DBMS. [Текст] / M. Stonebraker, A. Weisberg // IEEE Data Eng. Bull. — 2013. — Т. 36, № 2. — С. 21—27.

21. *Copeland, G. P.* A Decomposition Storage Model [Текст] / G. P. Copeland, S. N. Khoshafian // SIGMOD Rec. — New York, NY, USA, 1985. — Май. — Т. 14, № 4. — С. 268—279. — URL: https://doi.org/10.1145/971699.318923.

22. *Bernstein, P. A.* Concurrency Control in Distributed Database Systems [Текст] / P. A. Bernstein, N. Goodman // ACM Comput. Surv. — New York, NY, USA, 1981. — Июнь. — Т. 13, № 2. — С. 185—221. — URL: https://doi.org/10.1145/356842.356846.

23. Impala: A Modern, Open-Source SQL Engine for Hadoop. [Текст] / M. Kornacker [и др.] // Cidr. Т. 1. — 2015. — С. 9.

24. *Wanderman-Milne, S.* Runtime Code Generation in Cloudera Impala. [Текст] / S. Wanderman-Milne, N. Li // IEEE Data Eng. Bull. — 2014. — Т. 37, № 1. — С. 31—37.

25. Apache Hadoop, open-source software for reliable, scalable, distributed computing. The Apache Software Foundation [Электронный ресурс]. — URL: http://hadoop.apache.org (visited on 10/01/2021).

26. Apache HBase, the Hadoop database, a distributed, scalable, big data store. The Apache Software Foundation [Электронный ресурс]. — URL: http://hbase.apache.org (visited on 10/01/2021).

27. *Lattner, C.* LLVM: A compilation framework for lifelong program analysis & transformation [Текст] / C. Lattner, V. Adve // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — IEEE. 2004. — С. 75—86.

28. The LLVM Compiler Infrastructure. The LLVM Foundation [Электронный ресурс]. — URL: http://llvm.org (visited on 10/01/2021).

29. TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council [Электронный ресурс]. — URL: http://tpc.org/tpch (visited on 10/01/2021).

30. Apache Spark, a fast and general engine for large-scale data processing. The Apache Software Foundation [Электронный ресурс]. — URL: https://spark.apache.org (visited on 10/01/2021).

31. Spark sql: Relational data processing in spark [Текст] / M. Armbrust [и др.] // Proceedings of the 2015 ACM SIGMOD international conference on management of data. — 2015. — С. 1383—1394.

32. PostgreSQL, an open source object-relational database system. The PostgreSQL Global Development Group [Электронный ресурс]. — URL: https://www.postgresql.org (visited on 10/01/2021).

33. PostgreSQL derived databases. PostgreSQL wiki [Электронный ресурс]. — URL: https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases (visited on 10/01/2021).

34. ToroDB Stampede, a database bridging NoSQL and SQL. 8Kdata [Электронный ресурс]. — URL: https://www.torodb.com (visited on 10/01/2021).

35. Vertica, a "shared nothing" distributed analytical database. Hewlett Packard Enterprise Development [Электронный ресурс]. — URL: https://www.vertica.com (visited on 10/01/2021).

36. AgensGraph, a highly optimized, multi-model graph database for the modern, complex connected data environment. Bitnine Global [Электронный ресурс]. — URL: http://www.agensgraph.com (visited on 10/01/2021).

37. Amazon Redshift and the Case for Simpler Data Warehouses [Текст] / A. Gupta [и др.] // Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. — Melbourne, Victoria, Australia : Association for Computing Machinery, 2015. — С. 1917—1923. — (SIGMOD '15). — URL: https://doi.org/10.1145/2723372.2742795.

38. Greenplum Database (GPDB), an advanced, fully featured, open source data warehouse, based on PostgreSQL [Электронный ресурс]. — URL: https://greenplum.org/ (visited on 10/01/2021).

39. DeepgreenDB, a scalable MPP data warehouse solution derived from the open source Greenplum database project. Vitesse Data [Электронный ресурс]. — URL: https://vitessedata.com/products/deepgreen-db/ (visited on 10/01/2021).

40. PostgreSQL 11.0 Release Notes [Электронный ресурс]. — URL: https://www.postgresql.org/docs/release/11.0/ (visited on 10/01/2021).

41. PostgreSQL 11 Documentation: Just-In-Time Compilation (JIT) [Электронный ресурс]. — URL: https://www.postgresql.org/docs/11/jit.html (visited on 10/01/2021).

42. PostgreSQL 11 and Just In Time Compilation of Queries [Электронный ресурс]. — URL: https://www.citusdata.com/blog/2018/09/11/postgresql-11-just-in-time/ (visited on 10/01/2021).

43. HyPer – A Hybrid OLTP&OLAP High Performance DBMS [Электронный ресурс]. — URL: https://hyper-db.de/ (visited on 10/01/2021).

44. *Kemper, A.* HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots [Текст] / A. Kemper, T. Neumann // 2011 IEEE 27th International Conference on Data Engineering. — 2011. — С. 195—206.

45. Faster analytics with Hyper. TABLEAU SOFTWARE [Электронный ресурс]. — URL: https://www.tableau.com/products/new-features/hyper (visited on 10/01/2021).

46. Actian Vector (former VectorWise), a relational vectorized columnar analytic database. Actian Corporation [Электронный ресурс]. — URL: https://www.actian.com/analytic-database/vector-analytic-database/ (visited on 10/01/2021).

47. Hekaton: SQL Server's Memory-Optimized OLTP Engine [Текст] / C. Diaconu [и др.] // Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. — New York, New York, USA : Association for Computing Machinery, 2013. — С. 1243—1254. — (SIGMOD '13). — URL: https://doi.org/10.1145/2463676.2463710.

48. Compilation in the Microsoft SQL Server Hekaton Engine. [Текст] / C. Freedman, E. Ismert, P.-Å. Larson [и др.] // IEEE Data Eng. Bull. — 2014. — Т. 37, № 1. — С. 22—30.

49. SQLServer, a relational database. Microsoft [Электронный ресурс]. — URL: https://www.microsoft.com/en-us/sql-server (visited on 10/01/2021).

50. *Paroski, D.* Code generation: The inner sanctum of database performance [Текст] / D. Paroski // High Scalability; http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-ofdatabase-performance.html. — 2016.

51. *Krikellas, K.* Generating code for holistic query evaluation [Текст] / K. Krikellas, S. D. Viglas, M. Cintra // 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). — 2010. — С. 613—624.

52. MonetDB, an open source column-oriented database. MonetDB B.V. [Электронный ресурс]. — URL: https://www.monetdb.org/ (visited on 10/01/2021).

53. Building Efficient Query Engines in a High-Level Language [Текст] / Y. Klonatos [и др.] // Proc. VLDB Endow. — 2014. — Июнь. — Т. 7, № 10. — С. 853—864. — URL: https://doi.org/10.14778/2732951.2732959.

54. *Tahboub, R. Y.* How to Architect a Query Compiler, Revisited [Текст] / R. Y. Tahboub, G. M. Essertel, T. Rompf // Proceedings of the 2018 International Conference on Management of Data. — Houston, TX, USA : Association for Computing Machinery, 2018. — С. 307—322. — (SIGMOD '18). — URL: https://doi.org/10.1145/3183713.3196893.

55. *Rompf, T.* Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs [Текст] / T. Rompf, M. Odersky // Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. — Eindhoven, The Netherlands : Association for Computing Machinery, 2010. — С. 127—136. — (GPCE '10). — URL: https://doi.org/10.1145/1868294.1868314.

56. GLib, a general-purpose utility library. The GNOME Foundation [Электронный ресурс]. — URL: https://docs.gtk.org/glib/ (visited on 10/01/2021).

57. Clang: A C language family frontend for LLVM. The LLVM Foundation [Электронный ресурс]. — URL: https://clang.llvm.org/ (visited on 10/01/2021).

58. Implementation Techniques for Main Memory Database Systems [Текст] / D. J. DeWitt [и др.] // SIGMOD Rec. — New York, NY, USA, 1984. — Июнь. — Т. 14, № 2. — С. 1—8. — URL: https://doi.org/10.1145/971697.602261.

59. *SHAIKHHA, A.* Push versus pull-based loop fusion in query engines [Текст] / A. SHAIKHHA, M. DASHTI, C. KOCH // Journal of Functional Programming. — 2018. — Т. 28. — e10.

60. *Андреев, А. Н.* Классификация OLAP-систем вида xOLAP [Текст] / А. Н. Андреев // Рязанский государственный радиотехнический университет. — 2010.

61. What Is an OLTP System? ORACLE Help Center. [Электронный ресурс]. — URL: https://docs.oracle.com/database/121/VLDBG/GUID-0BC75680-5BD4-43A9-826F-CD8837D30EB2.htm (visited on 10/01/2021).

62. *Ketan, S. A.* An O (1) algorithm for implementing the LFU cache eviction scheme [Текст] / S. A. Ketan, M. D. Matani. — 2010.

63. LLVM Language Reference Manual. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/LangRef.html (visited on 10/01/2021).

64. MCJIT Design and Implementation. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/MCJITDesignAndImplementation.html (visited on 10/01/2021).

65. ORC Design and Implementation. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/ORCv2.html (visited on 10/01/2021).

66. LLJIT and LLLazyJIT. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/ORCv2.html#lljit-and-lllazyjit (visited on 10/01/2021).

67. The Julia Programming Language. Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah [Электронный ресурс]. — URL: https://julialang.org/ (visited on 10/01/2021).

68. Introducing the WebKit FTL JIT. Apple Inc [Электронный ресурс]. — URL: https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/ (visited on 10/01/2021).

69. Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM [Текст] / В. Г. Варданян [и др.] // Труды Института системного программирования РАН. — 2015. — Т. 27, № 6. — С. 33—48.

70. Numba: A High Performance Python Compiler [Электронный ресурс]. — URL: http://numba.pydata.org/ (visited on 10/01/2021).

71. MacRuby, an implementation of Ruby 1.9 directly on top of Mac OS X core technologies such as the Objective-C runtime and garbage collector, the LLVM compiler infrastructure and the Foundation and ICU frameworks. Apple Inc [Электронный ресурс]. — URL: http://macruby.org/ (visited on 10/01/2021).

72. Pyston: JIT-based Python implementation. Dropbox Tech [Электронный ресурс]. — URL: https://dropbox.tech/infrastructure/introducing-pyston-an-upcoming-jit-based-python-implementation (visited on 10/01/2021).

73. BOLT: A Practical Binary Optimizer for Data Centers and Beyond [Текст] / M. Panchenko [и др.] // Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. — Washington, DC, USA : IEEE Press, 2019. — С. 2—14. — (CGO 2019).

74. opt - LLVM optimizer. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/CommandGuide/opt.html (visited on 10/01/2021).

75. llvm-link - LLVM bitcode linker. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/docs/CommandGuide/llvm-link.html (visited on 10/01/2021).

76. LLVM-C: C interface to LLVM. The LLVM Foundation [Электронный ресурс]. — URL: https://llvm.org/doxygen/group__LLVMC.html (visited on 10/01/2021).

77. PostgreSQL 11 Documentation: Index Types. The PostgreSQL Global Development Group [Электронный ресурс]. — URL: https://www.postgresql.org/docs/11/indexes-types.html (visited on 10/01/2021).

78. *Boncz, P. A.* MonetDB/X100: Hyper-Pipelining Query Execution [Текст] / P. A. Boncz, M. Zukowski, N. Nes // CIDR. — 2005.

79. TPC-DS, decision support benchmark. Transaction Processing Performance Council [Электронный ресурс]. — URL: http://tpc.org/tpcds (visited on 10/01/2021).

80. Transaction Processing Performance Council [Электронный ресурс]. — URL: http://tpc.org (visited on 10/01/2021).

Список рисунков

Список таблиц

Приложение А

Результаты тестирования на запросах из тестовых наборов TPC-DS и TPC-H

В таблицах 9 — 15 приводятся полные результаты тестирования разработанного динамического компилятора запросов для СУБД PostgreSQL на тестовых наборах TPC-DS в масштабе 50 и TPC-H в масштабах 75, 20 и 10 на платформе x86-64. Ускорение вычислялось путём деления времени выполнения интерпретатора на время выполнения динамически скомпилированного запроса и показывает во сколько раз ускорился запрос.

В таблице 16 показаны результаты тестирования динамического компилятора запросов на тестовом наборе TPC-H в масштабе 75 при параллельном выполнении на 8 ядрах на платформе ARM.

Таблица 9 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-DS на платформе x86-64 в масштабе 50.

TPC-DS | SF=50 | x86-64							
№ запроса	JIT,сек	JIT+кэш,сек	JIT+кэш+эвристики,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+кэш}$	$\frac{PG}{JIT+кэш+эвристики}$
q2	44.52	51.94	45.17	141.90	3.19	2.73	3.14
q3	1.83	0.21	0.22	0.24	0.13	1.14	1.09
q5	28.66	24.26	14.48	26.98	0.94	1.11	1.86
q7	19.56	18.70	27.37	28.24	1.44	1.51	1.03
q8	25.50	24.08	24.06	24.77	0.97	1.03	1.03
q9	42.91	40.01	42.58	57.49	1.34	1.44	1.35
q10	2.34	0.39	0.34	0.43	0.18	1.10	1.26
q12	6.46	4.65	4.74	5.70	0.88	1.23	1.20
q13	16.92	12.92	13.76	13.80	0.82	1.07	1.00
q14a	395.15	405.45	391.93	523.79	1.33	1.29	1.34
q14b	413.75	356.99	388.58	501.87	1.21	1.41	1.29
q15	12.98	13.25	14.77	16.24	1.25	1.23	1.10
q16	3.08	1.18	1.41	1.39	0.45	1.18	0.99
q17	29.29	33.27	43.84	45.53	1.55	1.37	1.04
q18	9.32	4.72	5.22	6.16	0.66	1.31	1.18
q19	23.60	22.86	21.64	26.88	1.14	1.18	1.24
q20	2.57	0.60	0.71	0.78	0.30	1.30	1.10
q21	1.66	0.39	0.57	0.59	0.36	1.51	1.04
q22	43.87	39.98	41.53	48.80	1.11	1.22	1.18
q23a	268.64	275.89	208.34	349.18	1.30	1.27	1.68
q23b	265.64	279.07	213.54	367.08	1.38	1.32	1.72

q24a	3.19	0.77	0.85	0.91	0.29	1.18	1.07
q24b	3.23	0.77	0.87	0.90	0.28	1.17	1.03
q25	52.42	50.45	49.39	50.99	0.97	1.01	1.03
q26	5.11	4.01	8.15	8.47	1.66	2.11	1.04
q27	19.76	18.80	18.08	23.61	1.19	1.26	1.31
q28	18.04	15.15	15.35	22.39	1.24	1.48	1.46
q29	26.83	26.18	33.65	32.34	1.21	1.24	0.96
q31	57.85	63.18	58.87	154.30	2.67	2.44	2.62
q32	24.08	23.56	22.73	41.55	1.73	1.76	1.83
q33	25.05	22.58	22.50	26.87	1.07	1.19	1.19
q34	14.47	13.78	13.53	21.37	1.48	1.55	1.58
q35	29.53	29.96	31.50	43.96	1.49	1.47	1.40
q36	77.29	76.01	78.45	83.41	1.08	1.10	1.06
q37	1.16	0.18	0.18	0.18	0.16	1.00	1.00
q38	39.35	39.70	40.95	61.58	1.56	1.55	1.50
q39a	23.98	23.68	21.83	30.59	1.28	1.29	1.40
q39b	25.87	24.86	22.45	31.35	1.21	1.26	1.40
q40	2.54	0.93	1.20	1.30	0.51	1.40	1.08
q41	1.81	0.03	0.04	0.03	0.02	1.00	0.75
q42	21.42	21.10	21.34	25.33	1.18	1.20	1.19
q43	19.36	18.23	18.05	30.86	1.59	1.69	1.71
q44	7.66	4.22	4.50	4.89	0.64	1.16	1.09
q45	9.48	8.12	8.29	10.13	1.07	1.25	1.22
q46	32.61	32.98	31.94	39.95	1.23	1.21	1.25
q47	187.91	193.90	187.69	208.31	1.11	1.07	1.11
q48	43.40	42.47	37.67	48.48	1.12	1.14	1.29
q49	9.20	4.98	4.86	6.53	0.71	1.31	1.34
q50	380.15	394.03	447.23	477.70	1.26	1.21	1.07
q51	52.83	54.59	56.74	74.83	1.42	1.37	1.32
q52	22.54	21.59	21.06	24.33	1.08	1.13	1.16
q53	9.66	7.92	7.43	8.71	0.90	1.10	1.17
q54	3.53	0.89	0.95	0.97	0.27	1.09	1.02
q55	2.74	1.53	1.53	1.60	0.58	1.05	1.05
q56	9.06	5.09	5.36	7.65	0.84	1.50	1.43
q57	77.26	75.21	73.72	81.04	1.05	1.08	1.10
q58	4.57	1.19	1.38	1.31	0.29	1.10	0.95
q59	45.72	44.92	45.21	142.59	3.12	3.17	3.15
q60	36.67	32.07	31.83	39.46	1.08	1.23	1.24
q61	2.71	0.01	0.01	0.01	0.00	1.00	1.00
q62	7.40	6.31	6.47	16.48	2.23	2.61	2.55
q63	9.31	7.43	7.37	8.62	0.93	1.16	1.17
q64	31.34	25.60	26.45	35.59	1.14	1.39	1.35
q65	53.74	53.12	50.96	79.15	1.47	1.49	1.55
q66	24.79	14.49	13.99	22.55	0.91	1.56	1.61
q67	428.69	426.68	443.94	476.30	1.11	1.12	1.07

q68	21.99	21.90	21.29	24.30	1.11	1.11	1.14
q69	6.26	4.50	4.16	6.34	1.01	1.41	1.52
q70	53.10	51.20	50.80	65.85	1.24	1.29	1.30
q71	33.74	33.77	32.59	38.77	1.15	1.15	1.19
q72	354.30	328.58	277.11	369.61	1.04	1.12	1.33
q73	16.53	15.73	15.70	23.12	1.40	1.47	1.47
q75	75.63	69.70	67.77	103.64	1.37	1.49	1.53
q76	9.46	6.89	7.28	8.64	0.91	1.25	1.19
q77	30.40	26.08	25.19	26.07	0.86	1.00	1.03
q79	28.70	28.86	27.60	32.60	1.14	1.13	1.18
q80	37.31	32.77	31.21	32.08	0.86	0.98	1.03
q82	1.50	0.34	0.48	0.49	0.33	1.44	1.02
q83	4.20	0.41	0.50	0.51	0.12	1.24	1.02
q84	1.14	0.13	0.13	0.13	0.11	1.00	1.00
q85	4.85	1.59	1.79	1.85	0.38	1.16	1.03
q86	19.44	18.16	18.46	19.34	0.99	1.06	1.05
q87	39.33	38.74	42.48	59.96	1.52	1.55	1.41
q88	85.61	87.94	86.70	138.84	1.62	1.58	1.60
q89	23.66	22.58	21.79	31.00	1.31	1.37	1.42
q90	2.78	1.26	1.20	1.55	0.56	1.23	1.29
q91	2.27	0.27	0.30	0.35	0.15	1.30	1.17
q92	15.81	13.76	14.21	23.62	1.49	1.72	1.66
q93	3.20	1.99	2.04	2.04	0.64	1.03	1.00
q94	7.81	6.03	5.60	5.75	0.74	0.95	1.03
q96	11.19	10.97	10.83	16.58	1.48	1.51	1.53
q97	77.47	75.27	72.37	93.36	1.21	1.24	1.29
q98	18.09	16.56	16.75	19.71	1.09	1.19	1.18
q99	15.12	12.92	12.60	31.87	2.11	2.47	2.53
AVG	49.41	47.58	46.49	62.50	1.27	1.31	1.34

Таблица 10 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 20.

TPC-H | SF=20 | Native | x86-64							
№ запроса	JIT,сек	JIT+кэш,сек	JIT+кэш+эвристики,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{PG}{JIT+\text{кэш+эвристики}}$
Q1	15.49	18.53	15.69	73.18	4.72	3.95	4.66
Q2	7.79	4.30	4.31	5.53	0.71	1.29	1.28
Q3	14.09	17.55	14.30	29.25	2.08	1.67	2.05
Q4	4.87	3.60	3.67	4.01	0.82	1.11	1.09
Q5	32.92	34.62	32.60	32.98	1.00	0.95	1.01
Q6	10.29	10.40	10.57	11.06	1.07	1.06	1.05
Q7	12.76	12.94	12.79	16.87	1.32	1.30	1.32
Q8	7.56	4.74	4.51	5.00	0.66	1.05	1.11
Q9	21.08	19.78	19.27	33.02	1.57	1.67	1.71
Q10	14.21	12.44	12.64	26.78	1.88	2.15	2.12
Q11	2.54	0.89	1.34	1.35	0.53	1.52	1.01
Q12	17.94	17.48	17.29	22.09	1.23	1.26	1.28
Q13	23.44	25.74	23.67	34.37	1.47	1.34	1.45
Q14	3.60	2.62	2.59	3.85	1.07	1.47	1.49
Q15	7.97	8.10	9.21	9.20	1.15	1.14	1.00
Q16	6.86	5.52	7.72	7.77	1.13	1.41	1.01
Q17	2.03	1.34	2.03	1.93	0.95	1.44	0.95
Q18	18.44	16.17	16.19	30.00	1.63	1.86	1.85
Q19	2.12	0.78	0.78	1.46	0.69	1.87	1.87
Q20	6.89	6.35	6.35	7.04	1.02	1.11	1.11
Q21	41.68	37.54	37.51	52.05	1.25	1.39	1.39
Q22	3.25	2.41	2.42	2.73	0.84	1.13	1.13
AVG	12.63	11.99	11.70	18.71	1.48	1.56	1.60

Таблица 11 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 20.

TPC-H | SF=20 | Original | x86-64						
№ запроса	JIT,сек	JIT+кэш,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{JIT+\text{кэш}}{JIT}$
Q1	111.91	112.56	169.58	1.52	1.51	1.01
Q2	6.87	4.53	5.30	0.77	1.17	0.66
Q3	15.57	13.34	16.03	1.03	1.20	0.86
Q4	5.09	3.52	4.06	0.80	1.15	0.69
Q5	7.62	6.56	7.13	0.94	1.09	0.86
Q6	12.37	10.44	11.31	0.91	1.08	0.84
Q7	15.57	13.63	13.57	0.87	1.00	0.88
Q8	6.32	4.02	4.82	0.76	1.20	0.64
Q9	29.61	30.02	35.58	1.20	1.19	1.01
Q10	19.50	17.45	26.27	1.35	1.51	0.89
Q11	4.38	1.33	1.70	0.39	1.28	0.30
Q12	17.60	17.39	21.39	1.22	1.23	0.99
Q13	23.81	25.83	32.98	1.39	1.28	1.08
Q14	4.41	3.14	4.13	0.94	1.32	0.71
Q15	11.39	12.35	12.64	1.11	1.02	1.08
Q16	6.72	5.52	7.43	1.11	1.35	0.82
Q17	2.43	1.61	2.11	0.87	1.31	0.66
Q18	27.23	23.86	39.47	1.45	1.65	0.88
Q19	2.33	0.79	1.25	0.54	1.58	0.34
Q20	3.37	2.17	2.17	0.64	1.00	0.64
Q21	32.46	29.71	31.71	0.98	1.07	0.92
Q22	3.91	2.65	2.95	0.75	1.11	0.68
AVG	16.84	15.56	20.62	1.22	1.33	0.92

Таблица 12 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 10.

TPC-H | SF=10 | Native | x86-64						
№ запроса	JIT,сек	JIT+кэш,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{JIT+\text{кэш}}{JIT}$
Q1	8.24	8.38	36.05	4.38	4.30	1.02
Q2	4.74	2.12	2.49	0.53	1.17	0.45
Q3	6.66	5.67	9.43	1.42	1.66	0.85
Q4	2.76	1.99	2.32	0.84	1.17	0.72
Q5	4.33	2.83	3.38	0.78	1.19	0.65
Q6	4.35	4.10	5.24	1.20	1.28	0.94
Q7	7.98	6.33	6.42	0.80	1.01	0.79
Q8	3.88	1.90	2.22	0.57	1.17	0.49
Q9	10.88	9.88	13.28	1.22	1.34	0.91
Q10	8.69	6.89	12.08	1.39	1.75	0.79
Q11	1.94	0.45	0.62	0.32	1.38	0.23
Q12	8.65	7.74	11.70	1.35	1.51	0.89
Q13	11.51	11.78	16.53	1.44	1.40	1.02
Q14	2.07	1.23	1.72	0.83	1.40	0.59
Q15	4.11	5.01	4.09	1.00	0.82	1.22
Q16	3.99	2.71	3.83	0.96	1.41	0.68
Q17	1.40	0.62	0.93	0.66	1.50	0.44
Q18	8.88	7.81	16.33	1.84	2.09	0.88
Q19	1.74	0.45	0.64	0.37	1.42	0.26
Q20	2.28	0.95	1.10	0.48	1.16	0.42
Q21	14.37	14.40	16.05	1.12	1.11	1.00
Q22	2.30	1.23	1.38	0.60	1.12	0.53
AVG	5.72	4.75	7.63	1.33	1.61	0.83

Таблица 13 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 10.

TPC-H | SF=10 | Original | x86-64						
№ запроса	JIT,сек	JIT+кэш,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{JIT+\text{кэш}}{JIT}$
Q1	58.58	55.02	84.34	1.44	1.53	0.94
Q2	4.60	2.22	2.61	0.57	1.18	0.48
Q3	7.00	6.25	8.28	1.18	1.32	0.89
Q4	13.28	12.37	21.39	1.61	1.73	0.93
Q5	4.88	2.91	3.79	0.78	1.30	0.60
Q6	6.48	4.58	5.43	0.84	1.19	0.71
Q7	7.18	6.16	8.07	1.12	1.31	0.86
Q8	4.10	2.05	2.34	0.57	1.14	0.50
Q9	11.85	11.87	15.31	1.29	1.29	1.00
Q10	10.41	9.60	13.03	1.25	1.36	0.92
Q11	2.32	0.69	0.81	0.35	1.17	0.30
Q12	8.89	8.00	10.50	1.18	1.31	0.90
Q13	11.27	12.92	16.63	1.48	1.29	1.15
Q14	2.70	1.55	1.97	0.73	1.27	0.57
Q15	6.11	6.47	5.60	0.92	0.87	1.06
Q16	3.94	2.86	3.79	0.96	1.33	0.73
Q17	1.69	0.72	1.13	0.67	1.57	0.43
Q18	13.68	13.38	19.24	1.41	1.44	0.98
Q19	1.92	0.41	0.65	0.34	1.59	0.21
Q20	2.35	0.89	1.16	0.49	1.30	0.38
Q21	14.15	13.13	16.70	1.18	1.27	0.93
Q22	2.80	1.38	1.59	0.57	1.15	0.49
AVG	9.10	7.97	11.11	1.22	1.39	0.88

Таблица 14 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 75.

TPC-H | SF=75 | Native | x86-64							
№ запроса	JIT,сек	JIT+кэш,сек	JIT+кэш+эвристики,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{PG}{JIT+\text{кэш+эвристики}}$
Q1	54.62	60.29	55.63	278.86	5.11	4.63	5.01
Q2	20.57	19.67	22.20	21.07	1.02	1.07	0.95
Q3	66.31	63.01	62.85	120.42	1.82	1.91	1.92
Q4	36.74	36.36	36.56	41.87	1.14	1.15	1.15
Q5	392.43	386.62	391.61	407.89	1.04	1.06	1.04
Q6	67.77	63.53	64.03	71.41	1.05	1.12	1.12
Q7	165.24	157.03	160.36	185.96	1.13	1.18	1.16
Q8	40.37	42.24	39.66	43.56	1.08	1.03	1.10
Q9	167.65	170.45	167.72	205.41	1.23	1.21	1.22
Q10	53.65	58.80	53.45	95.96	1.79	1.63	1.80
Q11	5.48	4.18	5.41	5.65	1.03	1.35	1.04
Q12	100.84	100.78	100.51	116.38	1.15	1.15	1.16
Q13	93.21	95.06	93.63	130.57	1.40	1.37	1.39
Q14	24.41	23.46	27.90	27.95	1.15	1.19	1.00
Q15	80.33	83.93	80.05	97.91	1.22	1.17	1.22
Q16	24.02	23.94	23.96	29.67	1.24	1.24	1.24
Q17	5.37	4.81	5.78	7.26	1.35	1.51	1.26
Q18	45.53	47.11	45.44	100.27	2.20	2.13	2.21
Q19	4.04	2.77	2.93	5.02	1.24	1.81	1.71
Q20	84.07	81.20	81.07	86.09	1.02	1.06	1.06
Q21	221.48	231.99	221.49	302.96	1.37	1.31	1.37
Q22	9.00	9.63	9.07	10.47	1.16	1.09	1.15
AVG	80.14	80.31	79.61	108.76	1.36	1.35	1.37

Таблица 15 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе x86-64 в масштабе 10.

TPC-H | SF=75 | Original | x86-64						
№ запроса	JIT,сек	JIT+кэш,сек	PG,сек	$\frac{PG}{JIT}$	$\frac{PG}{JIT+\text{кэш}}$	$\frac{JIT+\text{кэш}}{JIT}$
Q1	407.41	422.15	615.42	1.51	1.46	1.04
Q2	21.22	19.77	20.72	0.98	1.05	0.93
Q3	70.62	77.26	121.20	1.72	1.57	1.09
Q4	40.08	36.33	39.61	0.99	1.09	0.91
Q5	51.23	50.83	55.19	1.08	1.09	0.99
Q6	70.83	71.02	78.30	1.11	1.10	1.00
Q7	162.27	164.49	176.34	1.09	1.07	1.01
Q8	39.08	41.80	46.77	1.20	1.12	1.07
Q9	177.66	189.90	214.01	1.20	1.13	1.07
Q10	66.57	70.72	102.47	1.54	1.45	1.06
Q11	7.02	5.22	6.56	0.93	1.26	0.74
Q12	99.19	100.73	114.73	1.16	1.14	1.02
Q13	99.05	98.60	132.79	1.34	1.35	1.00
Q14	27.60	26.69	30.21	1.09	1.13	0.97
Q15	89.66	98.58	99.06	1.10	1.00	1.10
Q16	23.68	22.33	30.17	1.27	1.35	0.94
Q17	6.56	5.63	8.10	1.23	1.44	0.86
Q18	113.55	112.80	157.73	1.39	1.40	0.99
Q19	4.74	3.52	4.97	1.05	1.41	0.74
Q20	25.96	23.86	26.78	1.03	1.12	0.92
Q21	89.46	94.86	144.01	1.61	1.52	1.06
Q22	9.59	10.36	11.56	1.21	1.12	1.08
AVG	77.41	79.43	101.67	1.31	1.28	1.03

Таблица 16 — Время выполнения интерпретатора PostgreSQL и динамического компилятора на тестовом наборе TPC-H на платформе ARM в масштабе 75 при параллельном выполнении на 8 ядрах.

TPC-H | SF=75 | Native | ARM | parallel_workers=8								
№ запроса	JIT,сек	$JIT_{\text{пар.}}$,сек	PG,сек	$PG_{\text{пар.}}$,сек	$\frac{PG}{PG_{\text{пар.}}}$	$\frac{PG}{JIT_{\text{пар.}}}$	$\frac{PG}{JIT}$	$\frac{PG_{\text{пар.}}}{JIT_{\text{пар.}}}$
Q1	70.12	11.45	389.33	51.99	7.49	34.00	5.55	4.54
Q2	27.54	21.38	29.52	23.69	1.25	1.38	1.07	1.11
Q3	78.44	44.45	159.74	50.94	3.14	3.59	2.04	1.15
Q4	49.88	9.33	51.59	9.75	5.29	5.53	1.03	1.05
Q5	66.54	20.14	72.85	22.55	3.23	3.62	1.09	1.12
Q6	79.92	7.98	91.38	14.57	6.27	11.45	1.14	1.83
Q7	239.64	32.75	259.82	37.15	6.99	7.93	1.08	1.13
Q8	65.23	11.66	66.48	11.58	5.74	5.70	1.02	0.99
Q9	223.65	68.19	273.35	81.18	3.37	4.01	1.22	1.19
Q10	68.13	16.36	128.11	17.77	7.21	7.83	1.88	1.09
Q11	6.38	5.33	8.44	4.50	1.88	1.58	1.32	0.84
Q12	117.07	12.44	149.97	21.41	7.00	12.06	1.28	1.72
Q13	123.21	119.39	171.31	169.89	1.01	1.43	1.39	1.42
Q14	30.68	26.39	33.06	33.15	1.00	1.25	1.08	1.26
Q15	99.91	92.49	118.38	117.54	1.01	1.28	1.18	1.27
Q16	33.20	39.66	45.21	44.29	1.02	1.14	1.36	1.12
Q17	6.92	6.97	10.07	10.09	1.00	1.44	1.46	1.45
Q18	79.61	75.99	200.71	199.41	1.01	2.64	2.52	2.62
Q19	5.51	2.57	7.04	1.31	5.37	2.74	1.28	0.51
Q20	38.30	31.60	42.36	33.86	1.25	1.34	1.11	1.07
Q21	124.04	81.36	214.98	141.26	1.52	2.64	1.73	1.74
Q22	11.53	7.77	14.84	7.85	1.89	1.91	1.29	1.01
AVG	74.79	33.89	115.39	50.26	2.30	3.40	1.54	1.48

Приложение Б

Реализация метода динамической компиляции выражений в SQL-запросах для СУБД PostgreSQL

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2017660035

«Реализация метода динамической компиляции выражений в SQL-запросах для СУБД PostgreSQL»

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук (RU)***

Авторы: ***Бучацкий Рубен Артурович (RU), Жуйков Роман Александрович (RU), Шарыгин Евгений Юрьевич (RU), Скворцов Леонид Владленович (RU), Баев Роман Вячеславович (RU), Мельник Дмитрий Михайлович (RU)***

Заявка № **2017616982**

Дата поступления **17 июля 2017 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***13 сентября 2017 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев