Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL

[1] *Е.Ю. Шарыгин <eush@ispras.ru>*
[1] *Р.А. Бучацкий <ruben@ispras.ru>*
[2] *Л.В. Скворцов <leonidxo@gmail.com>*
[1] *Р.А. Жуйков <zhroma@ispras.ru>*
[1] *Д.М. Мельник <dm@ispras.ru>*
[1]*Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1, стр. 52, факультет ВМК

Аннотация. В последние годы по мере увеличения производительности и роста объема оперативной и внешней памяти производительность СУБД для некоторых классов запросов определяется непосредственно скоростью обработки запросов процессором. В СУБД PostgreSQL для исполнения SQL-запросов традиционно используется механизм интерпретации, который приводит к накладным расходам, например, связанным с множественным ветвлением, неявными вызовами функций-обработчиков и выполнением лишних проверок, которых можно избежать, используя информацию, доступную только во время выполнения запроса.

В данной работе рассматривается метод динамической компиляции запросов, в частности, компиляция выражений оператора WHERE и метода последовательного сканирования таблиц SeqScan для СУБД PostgreSQL с помощью компиляторной инфраструктуры LLVM. Рассматривается оптимизация доступа к атрибутам, заключающаяся в вычислении смещений атрибутов кортежа во время компиляции запроса, а также метод автоматической трансляции встроенных функций PostgreSQL во внутреннее представление LLVM IR, что позволяет использовать один и тот же исходный код как для JIT-компилятора, так и для имеющегося интерпретатора. Генерация машинного кода во время выполнения запроса дает возможность избежать накладных расходов традиционной системы интерпретации.

Метод реализован в виде расширения к СУБД PostgreSQL и не требует изменения исходного кода СУБД. Результаты проведенного тестирования показывают, что динамическая компиляция запросов с помощью JIT-компилятора LLVM позволяет получить ускорение в несколько раз на синтетических тестах.

Ключевые слова: динамическая компиляция; JIT-компиляция; базы данных; PostgreSQL; LLVM; языки запросов

DOI: 10.15514/ISPRAS-2016-28(4)-13

Для цитирования: Шарыгин Е.Ю., Бучацкий Р.А., Скворцов Л.В., Жуйков Р.А., Мельник Д.М. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 217-240. DOI: 10.15514/ISPRAS-2016-28(4)-13

1. Введение

Среди систем управления базами данных идёт постоянная борьба за производительность. Работы по улучшению производительности большинства реляционных СУБД традиционно были в основном направлены на оптимизацию доступа к памяти ценой менее эффективного использования процессора. Кроме того, реализация в СУБД алгебры реляционных операторов и модели итераторов [1] позволяет упростить как построение и оптимизацию планов, так и реализацию реляционных операторов в отдельности, но в то же время приводит к значительным накладным расходам при выполнении плана.

С ростом объёмов и улучшением операционных характеристик доступа к оперативной памяти накладные расходы, связанные с неэффективным использованием процессора, становятся всё более заметными.

Одно из решений — динамическая компиляция запросов, которая позволяет во время выполнения получить эффективный машинный код, оптимизированный с учётом структуры конкретного запроса, используемых в нём типов данных и функций, и параметров базы данных, таких как размер и схема используемых таблиц, типы индексов и т.д.

В данной работе рассматривается динамическая компиляция выражений оператора WHERE и метода последовательного сканирования SeqScan для СУБД PostgreSQL [2] с помощью компиляторной инфраструктуры LLVM [3].

2. Обзор схожих работ

Увеличение эффективности использования процессора в реляционных СУБД является одним из перспективных направлений исследования в области разработки систем обработки данных.

Динамическая компиляция запросов [4,5] или микро-специализация кода СУБД [6] с использованием информации, доступной во время выполнения, в оптимизированный машинный код позволяет добиться более эффективного использования процессора, сохранив при этом общую архитектуру СУБД и её подсистем, изменив только модуль вычисления запросов. Кроме того, динамическая компиляция открывает новые возможности для оптимизации, связанные с подстановкой констант и вычислением арифметических выражений, традиционно выполняемом при помощи интерпретации.

В [4,5] описывается алгоритм генерации эффективного машинного кода для запросов к реляционной СУБД на языке SQL с использованием компиляторной инфраструктуры LLVM [3]. В работе аргументируется отказ от модели итераторов (Volcano-модели) [1] и приводятся экспериментальные данные,

согласно которым использование динамической компиляции запросов позволяет добиться ускорения в 2 раза, а смена модели выполнения на модель явных вложенных циклов (data-centric) — ещё в 3–4 раза.

Микро-специализация [6] предполагает замену «горячих» участков кода СУБД на версии, оптимизированные под конкретную таблицу, запрос или кортеж. Наличие различных степеней гранулярности позволяет, например, сохранять код, специализированный под таблицы или кортежи, для долгосрочного хранения и использования. В качестве одного из примеров, исследователи показывают, как можно использовать подход микро-специализации для ускорения процедуры обращения к атрибутам в СУБД PostgreSQL.

Для СУБД PostgreSQL разработано коммерческое расширение Vitesse DB [7] с закрытым исходным кодом, в котором реализована динамическая компиляция запросов с использованием LLVM. На запросе Q1 из набора тестов TPC-H [8] компиляция предикатов позволила получить ускорение в 2 раза, а компиляция всего запроса в одну функцию — ускорение в 8 раз. Дальнейшая оптимизация с привлечением параллелизма и колоночного хранилища позволила получить ускорение до 180 раз.

Компиляция арифметических выражений применяется в системах распределённой обработки данных Cloudera Impala [9] и Spark SQL [10].

В Cloudera Impala для компиляции критичных для производительности функций в оптимизированный машинный код используется инфраструктура LLVM. Эксперименты показали увеличение производительности до 5 раз. Особое внимание исследователи уделяют оптимизации косвенных и виртуальных вызовов функций.

В оптимизаторе запросов Spark SQL выполняется трансляция арифметических выражений в код на языке Scala, который затем динамически компилируется в JVM-байткод и запускается. В работе отмечается сокращение количества ветвлений и косвенных вызовов в скомпилированном коде.

Динамическая компиляция и оптимизация с помощью LLVM [3] используются во многих проектах. LLVM используют как при разработке новых языков программирования (Julia [11]), так и при создании более производительных реализаций существующих: JavaScript (JavaScriptCore FTL [12] и LLV8 [13]), Python (Pyston [14]), Ruby (MacRuby [15]) и других.

3. Архитектура обработки запроса в СУБД PostgreSQL

Основной алгоритм выполнения SQL-запроса в PostgreSQL состоит из четырёх этапов:

1. Разбор и анализ запроса.
2. Преобразование.
3. Составление плана.
4. Выполнение плана.

На первом этапе PostgreSQL выполняет лексический и синтаксический анализ SQL запроса и строит дерево разбора. На следующем шаге процедура преобразования принимает от анализатора дерево разбора и выполняет семантический анализ, необходимый для понимания, к каким именно таблицам, функциям и операторам обращается запрос. Структура данных, которая создаётся для представления этой информации, называется деревом запроса.

На третьем этапе PostgreSQL на основе дерева запроса составляет план выполнения запроса. Планировщик работает со структурами данных, называемыми путями, которые представляют собой упрощённые схемы планов, содержащие информацию, необходимую планировщику для принятия решений. После выбора наиболее эффективного пути выполнения строится дерево плана, которое передаётся исполнителю. При подготовке плана PostgreSQL также оценивает время, которое будет затрачено на выполнение того или иного шага выполнения запроса. Итоговый план является наиболее эффективным с точки зрения имеющихся оценок затрат на его выполнение.

Структуру плана выполнения можно вывести с помощью команды EXPLAIN. Например, на рис. 1 показан план выполнения для простого запроса “*select count(*) from rtbl where x+y < 1”;*

```
                QUERY PLAN
-------------------------------------------------
Aggregate
  ->  Seq Scan on rtbl
        Filter: ((x + y) < '1'::double precision)
(3 rows)
```

Рис. 1. Пример плана выполнения запроса

Fig. 1. Example of execution plan

Финальным этапом является выполнение плана, которое реализовано при помощи модели итератора, также известной как Volcano Style Processing [1]. В этой модели запрос состоит из множества операторов, каждый оператор является итератором с интерфейсом “open()”, “next()”, “close()”.

Исполнитель принимает план, созданный планировщиком, и обрабатывает его рекурсивно сверху вниз по дереву, при этом каждый узел в дереве плана вызывает метод “next()” от узлов ниже в дереве плана для получения входных данных, обрабатывает и возвращает один кортеж на узел выше.

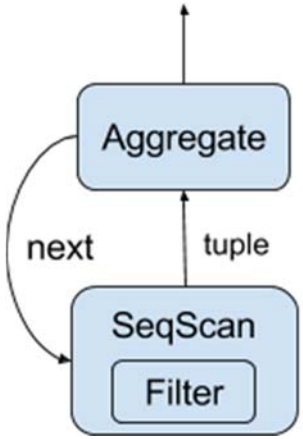

Рис. 2. Модель итераторов для запроса

Fig. 2. Iterator model for example query

В плане выполнения запроса на рис. 2 узел Aggregate будет обращаться за входными данными к дочернему узлу SeqScan, представляющему чтение таблицы. В результате выполнения узла SeqScan исполнитель выберет одну строку из таблицы и вернет её вызывающему узлу Aggregate. Надо заметить, что условие WHERE применено как фильтр (Filter на рис. 2) к узлу плана SeqScan, который проверяет условие для каждой прочтенной им строки и выводит только удовлетворяющие условию строки.

Динамической компиляции данного метода сканирования и фильтрации посвящена основная часть этой работы.

4. Компиляторная инфраструктура LLVM

LLVM [3] — компиляторная инфраструктура для компиляции и оптимизации программ. В LLVM используется низкоуровневое типизированное платформо-независимое промежуточное представление LLVM IR, основанное на SSA-форме, которое, в свою очередь, может быть представлено и использовано одним из трёх способов:

- как граф структур данных, представляющих функции, базовые блоки и инструкции в оперативной памяти — используется для генерации, анализа и оптимизации программ;
- как закодированное бинарное представление, называемое биткодом LLVM, — используется как формат ввода-вывода в различных инструментах, составляющих инфраструктуру LLVM;
- как человекочитаемое текстовое представление — используется для отладки.

Инфраструктура LLVM предоставляет богатый API на языках C++, C, Go, OCaml и Python для анализа и оптимизации программ. Кроме того, в её состав входит широкий набор инструментов, из которых в данной работе используются:

- Clang — компилятор с языков C, C++ и Objective-C во внутреннее представление LLVM;
- opt — статический оптимизатор LLVM-представления;
- llvm-link — компоновщик LLVM-модулей, позволяет скомпоновать несколько модулей в один;
- llc — статический компилятор из внутреннего представления LLVM под различные платформы (поддерживаются x86, x86_64, ARM и многие другие); в состав llc входит библиотека CPPBackend, которая позволяет компилировать в код на языке C++ с использованием LLVM C++ API.

Инфраструктура LLVM содержит модуль для динамической компиляции MCJIT [16], в котором задействованы механизмы LLVM для машинно-зависимой оптимизации и генерации кода под различные платформы. Используя MCJIT 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).

5. Особенности хранения данных и реализации последовательного сканирования в PostgreSQL

Рассмотрение реализации SeqScan в PostgreSQL предварим кратким описанием структуры данных, используемой в PostgreSQL для хранения таблиц, а именно heap-файла [17].

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 (по умолчанию) 1 ГБ.

Heap-файл состоит из последовательности страниц фиксированного размера (по умолчанию 8 КБ). Структура страницы представлена на рис. 3.

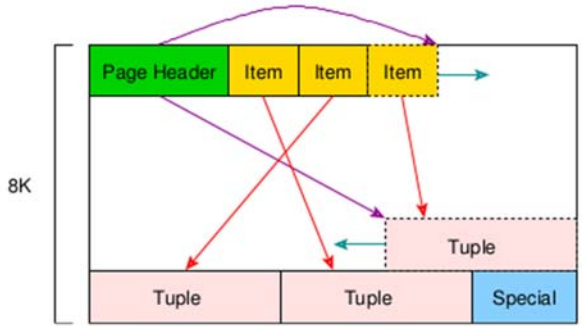

Рис. 3. Структура страницы в heap-файле

Fig. 3. Structure of a heap file page

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. Начиная с версии PostgreSQL 9.6 внешний цикл может выполняться 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 Volcano-модели, оператор SeqScan представляется итератором, с каждым вызовом соответствующего которому метода next() производится выполнение 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 6.2). Для обеспечения доступа к участвующим в выражении атрибутам кортежа используется функция slot_getattr, которая по

мере необходимости производит частичную десериализацию атрибутов кортежа.

Процесс десериализации 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 (в т.н. TOAST-таблице);
- атрибутов в кортеже 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.

6. Реализация динамической компиляции в PostgreSQL

JIT-компилятор запросов для PostgreSQL, о котором идёт речь в этой статье, реализован в виде расширения к СУБД.

Механизм расширений в PostgreSQL предоставляет весьма широкие возможности: при помощи расширений можно определять новые типы данных, типы индексов (access methods), новые функции и операторы для использования в SQL-запросах, а также перехватывать управление на определённых этапах обработки запроса при помощи регистрации функций-обработчиков.

Во время загрузки расширение регистрирует обработчик выполнения запроса, который вызывается после этапа оптимизации непосредственно перед выполнением плана. В обработчике проверяется, поддерживаются ли все выражения, используемые в запросе, в случае чего производится динамическая компиляция и выполнение кода, оптимизированного под конкретный запрос.

В 6.1 описывается динамическая компиляция метода сканирования SeqScan, а в 6.2 — выражений оператора WHERE.

6.1 Динамическая компиляция метода сканирования SeqScan

SeqScan — это один из методов сканирования таблиц, который выполняет последовательное сканирование таблицы в поисках подходящих под условие кортежей. Он регулярно обращается к фильтру, который вычисляет результат логического выражения, стоящего за оператором WHERE. Последовательное сканирование определено для всех таблиц и является базовым методом доступа к данным в PostgreSQL.

При разработке рассматриваемого в данной работе JIT-компилятора для PostgreSQL оператор SeqScan, реализация которого подробно описана в разд. 5, было решено переписать с использованием LLVM C API. Несмотря на несколько возросшую сложность, такое переписывание позволило:

- пересмотреть используемую вычислительную модель (раздел 6.1.1);
- спроектировать и реализовать ряд оптимизаций, возможных только в динамически компилируемом окружении (6.1.2, 6.1.3);
- динамически компилировать и оптимизировать код вычисления арифметических выражений и предикатов (6.2).

Рассмотрим некоторые самые значимые из применённых оптимизаций.

6.1.1 Отказ от итеративной модели

В используемой в PostgreSQL Volcano-модели [1] каждый оператор реализуется при помощи итератора, метод “next()” которого возвращает следующий кортеж. Реализация метода “next()” нелистового оператора в дереве запроса использует “next()” для получения данных от дочерних операторов. Таким образом, для каждого конкретного запроса операторы в Volcano-модели организуются в конвейер, в котором поток данных управляется корневым оператором запроса, через цепочку вызовов “next()” продвигающим циклы сканирования на следующую итерацию.

Описанная модель обладает следующими достоинствами:

1) позволяет останавливать выполнение дочерних операторов (например, из оператора Limit), что позволяет избежать лишних вычислений;

2) позволяет распределять вычисления на несколько вычислительных узлов [1].

Эти достоинства, однако, никак не проявляются на рассматриваемых в данной работе простых запросах вида:

select <columns> from <table> where <condition>;

В то же время, в недостатки итеративной модели, проявляющиеся даже на таких простых запросах, можно отнести существенные накладные расходы. Так, для оператора SeqScan необходимость сохранения состояния между вызовами next() означает, что для каждого считываемого из таблицы кортежа необходимо

вначале загрузить переменные состояния, в том числе счётчики циклов, из объекта SeqScanState и только потом продолжить выполнение с нужной итерации, при этом записав обновлённые значения переменных для последующих вызовов.

Без использования модели итераторов тот же запрос мог бы быть представлен непосредственно при помощи двух вложенных циклов:

```
for page ← table
    for tuple ← page
        if condition(tuple)
            print(columns(tuple))
```

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).

Оператор SeqScan в рассматриваемом в данной статье JIT-компиляторе реализован без использования модели итераторов.

6.1.2 Оптимизация доступа к атрибутам

Можно выделить следующие не реализованные в PostgreSQL возможности для оптимизации последовательного доступа к атрибутам:

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 нескольких атрибутов, которые имеют фиксированную длину и не могут принимать значение NULL.

2. Несмотря на то, что, начиная с атрибута, следующего за первым nullable или атрибутом переменной длины, смещения атрибутов фиксированными не являются, длину всякой последовательности not-nullable атрибутов фиксированной длины можно вычислить заранее, что позволяет пропускать такие последовательности, состоящие из атрибутов, не используемых в предикате.

3. Зная наперёд, 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.

Рассмотрим пример (рис. 4). Пусть в кортежах таблицы по 12 атрибутов, из которых атрибуты 5, 8 и 11 являются nullable или имеют переменную длину (помечены символом N/V). Пусть в запросе встречаются только атрибуты 3 и 8.

Рис. 4. Оптимизация доступа к атрибутам. Цветом выделены атрибуты, участвующие в запросе.

Fig. 4. Attribute access optimization. Highlighted: attributes involved in the query.

Тогда согласно оптимизации (1) смещение атрибута 3 вычисляется на этапе компиляции, согласно оптимизации (2) длина атрибута 4 и атрибутов 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: атрибуты 3, 5 и 8.

Для сравнения, используемый в PostgreSQL механизм доступа к атрибутам требует чтения восьми атрибутов 1–8 за два вызова функции slot_getattr (первый вызов считывает атрибуты 1–3, второй — 4–8).

Предложенные оптимизации возможны только при динамической компиляции кода под конкретный запрос или конкретную таблицу и реализованы в рассматриваемом в данной статье JIT-компиляторе.

6.1.3 Подстановка констант и дальнейшая специализация

Применение динамической компиляции позволяет учитывать при оптимизации кода информацию, доступную только во время выполнения, в частности подставлять в код константные параметры, такие как количество страниц, направление обхода и т.д.

Например, на рис. 5 представлен фрагмент исходного кода PostgreSQL, который отвечает за извлечение атрибута из кортежа.

Во время выполнения для каждого конкретного атрибута известны параметры attbyval (тип передачи атрибута) и attlen, что позволяет подставить эти значения непосредственно в код и избежать излишних ветвлений.

Оптимизация подстановки констант реализована через разделение всех переменных, используемых в коде JIT-компилятора, на два класса: переменные времени компиляции и переменные времени выполнения. Оптимизации свёртки и продвижения констант выполняются после генерации кода вместе с прочими оптимизациями LLVM.

Преимуществом такого подхода является то, что при правильной реализации подстановка констант будет применена единообразно для всего запроса и что значения переменных времени компиляции могут использоваться при генерации кода для, например, условной генерации части функций.

Недостатком такого подхода является необходимость разделения всех переменных на два класса вручную, что, во-первых, подвержено ошибкам реализации и, во-вторых, может приводить к неоптимальному результату (когда разработчик по причине отсутствия доказательства противного из соображений корректности отвёл переменную-константу в класс переменных времени выполнения).

```
#define fetch_att(T,attbyval,attlen) \
( \
   (attbyval) ? \
   ( \
      (attlen) == (int) sizeof(Datum) ? \
         *((Datum *)(T)) \
      : \
     ( \
      (attlen) == (int) sizeof(int32) ? \
         Int32GetDatum(*((int32 *)(T))) \
      : \
      ( \
         (attlen) == (int) sizeof(int16) ? \
            Int16GetDatum(*((int16 *)(T))) \
         : \
         ( \
            AssertMacro((attlen) == 1), \
            CharGetDatum(*((char *)(T))) \
         ) \
      ) \
     ) \
   ) \
   : \
   PointerGetDatum((char *) (T)) \
)
```

Рис. 5. Код извлечения атрибута из кортежа (PostgreSQL)

Fig. 5. Source code of attribute extraction routine (PostgreSQL)

Постановка констант вместе с дальнейшими оптимизациями позволяет получить код оператора SeqScan, специализированный под конкретный запрос и конкретные таблицы базы данных.

6.2 Динамическая компиляция выражений оператора WHERE

WHERE — необязательный оператор PostgreSQL, имеющий форму: WHERE <*предикат*>, где предикат — это любое выражение, возвращающее результат логического типа. Кортеж удовлетворяет условию предиката, если для значений атрибутов данного кортежа результат условия является истиной. Кортежи, для которых значение предиката оператора WHERE является ложью или NULL, исключаются из результата запроса.

Для определения доли времени выполнения оператора WHERE, по которому фильтруются кортежи, было проведено профилирование выполнения SQL запросов из тестового набора TPC-H [8], по результатам которого выяснилось, что на некоторых запросах из TPC-H доля времени вычисления предикатов оператора WHERE достигает более 50%.

Для вычисления предиката оператора WHERE PostgreSQL выполняет интерпретацию дерева выражений, где каждое выражение состоит из дерева отдельных операторов и функций, как показано в левой части рис. 6. Каждая вершина дерева вызывает функции соответствующих дочерних вершин неявным образом (через указатель на функцию). Для вычисления самих операций вызываются встроенные функции PostgreSQL, а для доступа к атрибутам используется функция slot_getattr, которая по мере необходимости извлекает атрибуты из кортежа. Это приводит к большим накладным расходам во время выполнения, так как оптимизация встраивания функций (inlining) не может быть выполнена, что позволило бы компилятору делать дальнейшие оптимизации, такие как удаление общих подвыражений (common subexpression elimination, CSE) и т.д.

Поскольку во время выполнения известны вызываемые функции и операции, можно использовать кодогенерацию для замены неявных вызовов функций на явные, которые в дальнейшем могут быть встроены.

Для динамической компиляции выражений, стоящих за оператором WHERE, сперва анализируется дерево выражений с целью проверки на то, реализована ли поддержка всех необходимых выражений и типов значений. В случае, если выражение не поддерживается, запрос выполнится с использованием интерпретации, как обычно в PostgreSQL.

Далее производится рекурсивный обход дерева выражений в обратном порядке и вызов функций-генераторов, реализованных с применением LLVM C API и использующих функции-генераторы встроенных функций PostgreSQL для генерации кода операций на языке LLVM IR. Используемые атрибуты загружаются заранее при считывании кортежа (как описано в разделе 6.1.2).

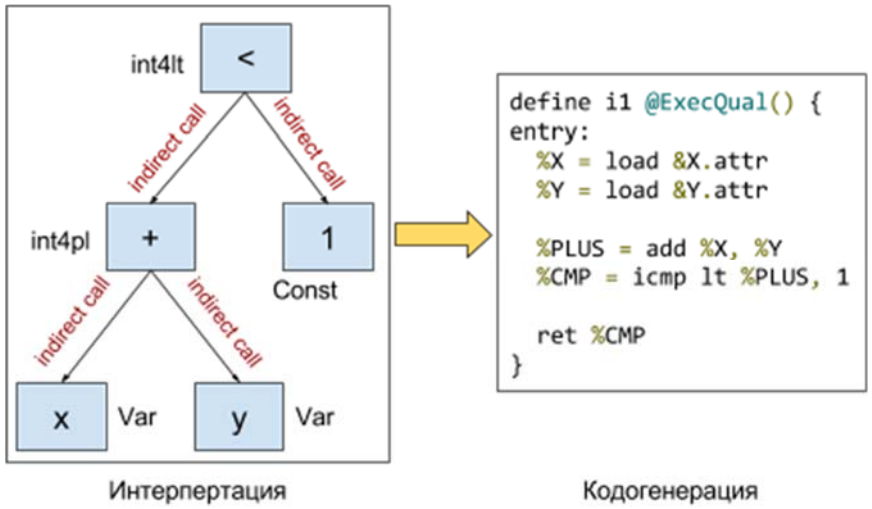

Рис. 6. Слева — интерпретация дерева выражений, справа — сгенерированный LLVM IR

Fig. 6. Left — interpretation of an expression tree, right — generated LLVM IR

В результате этого обхода генерируется код в виде функции на языке LLVM IR (ExecQual на рис. 6), которая будет вызвана из SeqScan для вычисления предикатов.

Как показано на правой части рис. 6, путем замены неявных вызовов функций на вызовы функций-генераторов LLVM IR и дальнейшей генерации кода с использованием оптимизации встраивания функций (inlining), код для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без каких-либо накладных расходов на неявные вызовы функций.

В данной работе рассматриваются два метода кодогенерации встроенных функций PostgreSQL, используемых при вычислении выражений: реализация вручную и предкомпиляция с использованием компилятора clang.

6.2.1 Реализация встроенных функций вручную

Одним из подходов к реализации функций-генераторов LLVM IR для выражений, используемых в операторе WHERE, является ручное переписывание встроенных функций PostgreSQL с использованием LLVM C API, для дальнейшей генерации кода на языке LLVM IR.

Например, на рис. 7 представлен фрагмент исходного кода PostgreSQL для функции сложения двух целых чисел (int4pl), а на рис. 8 — переписанная с использованием LLVM C API версия той же функции, при вызове которой сгенерируется код на языке LLVM IR.

```
Datum int4pl(PG_FUNCTION_ARGS)
{
  int32     arg1 = PG_GETARG_INT32(0);
  int32     arg2 = PG_GETARG_INT32(1);
  int32     result;

  result = arg1 + arg2;

  /*
   * Overflow check. If the inputs are of different signs then
   * their sum cannot overflow. If the inputs are of the same sign,
   * their sum had better be that sign too.
   */
  if (SAMESIGN(arg1, arg2) && !SAMESIGN(result, arg1))
    ereport(ERROR,
          (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),
           errmsg("integer out of range")));
  PG_RETURN_INT32(result);
}
```

Рис. 7. Исходный код встроенной функции int4pl (PostgreSQL)

Fig. 7. Source code of built-in function int4pl (PostgreSQL)

Реализация конкретных функций может отличаться от оригинальной версии, например, проверками на переполнение (в LLVM есть встроенные функции для быстрой проверки на переполнение арифметических операций: *llvm.sadd.with.overflow.i32* на рис. 8), но и функция, и её переписанная версия, должны возвращать одинаковый результат. Данный метод позволяет учитывать при генерации кода больше информации и тем самым дает возможность, в частности, избавиться или изменить множество проверок, которые накладывают дополнительные расходы во время выполнения.

К недостаткам данного метода можно отнести жёсткую привязанность к реализации функций на LLVM IR. Для поддержки всех встроенных функций PostgreSQL необходимо вручную написать функции-генераторы с помощью LLVM C API, отслеживать изменения в коде PostgreSQL и вносить соответствующие изменения в переписанные версии, что, учитывая суммарное число встроенных функций в PostgreSQL (больше 2000), является трудоёмкой задачей, чреватой возникновением ошибок при переписывании.

LLVM C API

```
LLVMValueRef define_int4pl (LLVMBuilderRef LLVMValueRef left,
LLVMValueRef right)
{
 LLVMValueRef int4pl = LLVMAddFunction("int4pl");
 LLVMBasicBlockRef entry_block, overflow_block, result_block;
 LLVMValueRef sadd_func, args[2] = {left, right}, result, over_bit, call;

 sadd_func = LLVMAddFunction("llvm.sadd.with.overflow.i32");

 LLVMPositionBuilderAtEnd(entry_block);
 call = LLVMBuildCall(sadd_func, args, 2);
 result = LLVMBuildExtractValue(call, 0);
 over_bit = LLVMBuildExtractValue(call, 1);
 over_bit = LLVMBuildIsNotNull(over_bit);
 LLVMBuildCondBr(over_bit, overflow_block, result_block);

 LLVMPositionBuilderAtEnd(overflow_block);
 LLVMBuildCall(OVERFLOW_ERROR_f, NULL, 0);
 LLVMBuildUnreachable();

 LLVMPositionBuilderAtEnd(result_block);
 LLVMBuildRet(result);
 return int4pl;
}
```

LLVM IR

```
define i64 @int4pl (i32 %0, i32 %1) {
entry_block:
 %call = call @llvm.sadd.with.overflow.i32(i32 %0, i32 %1)
 %result = extractvalue {i32, i1} %call, 0
 %over_bit = extractvalue {i32, i1} %call, 1
 %over_bit = icmp eq i1 %over_bit, null
 br i1 %over_bit, label %overflow_block, %result_block

overflow_block:
 %error = call @overflow_error
 unreachable

result_block:
 ret %result
}
```

Рис. 8. Функция-генератор LLVM IR для функции int4pl

Fig. 8. LLVM IR generator function for int4pl

Аналогично с типами переменных и констант, которые внутри PostgreSQL хранятся в виде 64-битных значений (Datum), что значит, что для каждого типа необходимо написать функцию, конвертирующую 64-битное значение в значение необходимого типа и обратно.

6.2.2 Предварительная компиляция встроенных функций

Недостаток метода ручной реализации всех встроенных функции PostgreSQL привел к необходимости рассмотрения альтернативных способов получения генераторов встроенных функций PostgreSQL на языке LLVM IR.

В состав LLVM (до версии 3.8) входила библиотека CPPBackend, которая переводит (транслирует) биткод LLVM в соответствующий код на языке C++, при выполнении которого вызываются функции из LLVM C++ API для генерации модуля исходного кода LLVM IR.

Используя библиотеку CPPBackend, был реализован метод автоматического получения функций-генераторов LLVM IR встроенных функций PostgreSQL путем предкомпиляции этих функций.

Метод работает следующим образом: множество файлов исходного кода PostgreSQL, содержащие встроенные функции, с помощью компилятора *clang* транслируются в объектные файлы биткода LLVM. Затем, с помощью компоновщика *llvm-link*, полученные файлы компонуются в единый биткод-файл, который оптимизируется модульным оптимизатором *opt*. На основе оптимизированного биткода статический компилятор *llc,* в котором реализован интерфейс библиотеки CPPBackend (-march=cpp), строит C++ файл, содержащий функции-генераторы на LLVM C++ API, вызовы которых сгенерируют код соответствующих встроенных функций исходного кода PostgreSQL на языке LLVM IR. Трансляция биткода в C++ код, содержащий вызовы функций из LLVM API на языке C++, показана на рис. 9. Общая схема метода показана на рис. 10.

Стоит отметить, что генерация биткод-файлов, их компоновка, оптимизация и трансляция в C++ файл и дальнейшая компиляция этого файла происходит один раз во время сборки расширения, что оставляет больше времени на оптимизацию кода, специфичного для конкретного запроса.

Другими преимуществами данного метода являются простота и универсальность реализации, упрощенная поддержка, поскольку отпадает необходимость в ручной реализации каждой встроенной функции и отслеживании изменений в коде PostgreSQL.

LLVM IR

```
define i64 @int4pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
 %1 = getelementptr %struct.FunctionCallInfoData, %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
 %2 = load i64, i64* %1
 %3 = trunc i64 %2 to i32
 %4 = getelementptr %struct.FunctionCallInfoData, %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
 %5 = load i64, i64* %4
 %6 = trunc i64 %5 to i32
 %7 = add nsw i32 %6, %3
 %.lobit = lshr i32 %3, 31
 %.lobit1 = lshr i32 %6, 31
 %8 = icmp ne i32 %.lobit, %.lobit1
 %.lobit2 = lshr i32 %7, 31
 %9 = icmp eq i32 %.lobit2, %.lobit
 %or.cond = or i1 %8, %9
 br i1 %or.cond, label %ret, label %overflow

overflow:
 tail call void @ereport(...)

ret:
 %10 = zext i32 %7 to i64
 ret i64 %18
}
```

LLVM C++ API

```
Function* define_int4pl(Module *mod) {

Function* func_int4pl = Function::Create(..., /*Name=*/"int4pl", mod);
// Block (entry)
Instruction* ptr_1 = GetElementPtrInst::Create(NULL, fcinfo, 0, "", entry);
LoadInst* int64_2 = new LoadInst(ptr_1, "", false, entry);
CastInst* int32_3 = new TruncInst(int64_2, IntegerType::get(..., 32), "", entry);
Instruction* ptr_4 = GetElementPtrInst::Create(NULL, fcinfo, 1, "", entry);
LoadInst* int64_5 = new LoadInst(ptr_4, "", false, entry);
CastInst* int32_6 = new TruncInst(int64_5, IntegerType::get(..., 32), "", entry);
BinaryOperator* int32_7 = BinaryOperator::Create(Add, int32_6, int32_3, "", entry);
BinaryOperator* lobit = BinaryOperator::Create(LShr, int32_3, 31, ".lobit", entry);
BinaryOperator* lobit1 = BinaryOperator::Create(LShr, int32_6, 31, ".lobit1", entry);
ICmpInst* int1_8 = new ICmpInst(*entry, ICMP_NE, lobit, lobit1, "");
BinaryOperator* lobit2 = BinaryOperator::Create(LShr, int32_7, 31, ".lobit2", entry);
ICmpInst* int1_9 = new ICmpInst(*entry, ICMP_EQ, lobit2, lobit, "");
BinaryOperator* int1_or_cond = BinaryOperator::Create(Or, int1_8, int1_9, "or.cond", entry);
BranchInst::Create(ret, overflow, int1_or_cond, entry);

// Block (overflow)
CallInst* void_err = CallInst::Create(func_erreport, void, "...", overflow);

// Block (ret)
CastInst* int64_10 = new ZExtInst(int32_7, IntegerType::get(..., 64), "", ret);
ReturnInst::Create(mod->getContext(), int64_10, ret);

return func_int4pl;
}
```

Рис. 9. Трансляция LLVM IR в LLVM C++ API с помощью CPPBackend

Fig. 9. Translation from LLVM IR into LLVM C++ API using CPPBackend library

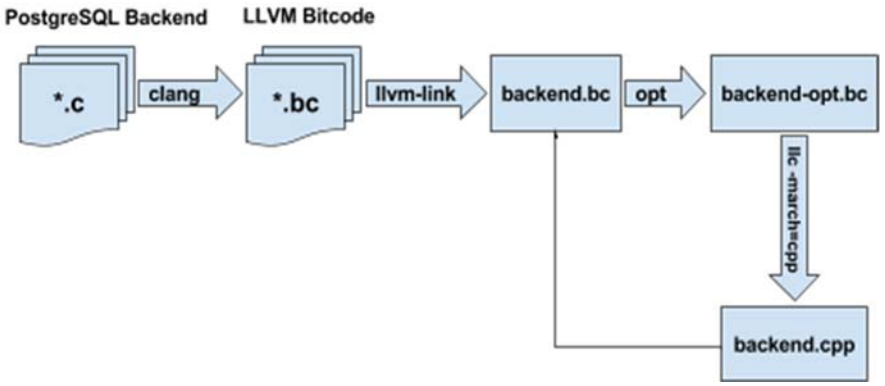

Рис. 10. Схема работы метода предкомпиляции функций PostgreSQL

Fig. 10. Scheme of the PostgreSQL backend function precompilation method

Одним из недостатков данного метода является то, что библиотека CPPBackend не обновлялась с версии LLVM 3.4, а с версии 3.9 не входит в состав LLVM. Реализация метода требует обновления CPPBackend до используемой в данной работе версии LLVM (3.7) и всех последующих версий.

К недостаткам данного метода также можно отнести ухудшение производительности по сравнению с реализацией вручную переписанных встроенных функций PostgreSQL. Ухудшение производительности является следствием того, что при ручном методе генерируется более специализированный под конкретный запрос (выражение) код.

7. Результаты

Для тестирования производительности был использован следующий SQL-запрос:

```
select a0 from widetbl where a199+a198 < 4;
```

Таблица widetbl содержит 201 столбец и 3000000 кортежей. Первый столбец имеет тип text (переменной длины), а остальные 200 имеют тип integer not null. Размер таблицы widetbl составляет 2605 МБ.

Динамическая компиляция оператора SeqScan (раздел 6.1) и предиката оператора WHERE (раздел 6.2) позволяет получить специализированный код под данный запрос, например, при выполнении данного запроса оптимизация доступа к атрибутам (раздел 6.1.2) позволяет пропускать при считывании кортежа атрибуты со 2-го по 199-й.

Тестирование производительности выполнялось на компьютере с четырёхъядерным процессором Intel Core i7 860 с тактовой частотой 2.80 ГГц и с 16 гигабайтами оперативной памяти под управлением 64-битной операционной системы Ubuntu Linux версии 14.04. Время выполнения

измерялось путём многократного выполнения запроса и подсчёта медианы полученных результатов.

Результаты тестирования отражены в табл. 1. Таким образом, время выполнения запроса сократилось в 4,32 раза с использованием метода предкомпиляции встроенных функций (раздел 6.2.2) и в 4,8 раза с использованием метода ручной реализации встроенных функций (раздел 6.2.1) по сравнению с версией PostgreSQL 9.6 Beta 2 с отключенным параллелизмом.

Табл. 1. Сравнение времени выполнения на первом запросе.

Table 1. Comparison of execution times on the first test query.

	PostgreSQL	Предкомпиляция	Ручная реализация
Время (мс)	2623,542	606,674	546,916
Ускорение (раз)	1,00	4,32	4,80

Для тестирования производительности был также использован следующий SQL-запрос:

```
select x, y from rtbl where sqrt((x-256)^2 + (y-128)^2) < 40;
```

Таблица rtbl содержит пять столбцов и 10000020 кортежей. Столбцы x и y имеют тип double precision. Размер таблицы rtbl составляет 1116 МБ.

Результаты тестирования отражены в табл. 2. Время выполнения этого запроса с использованием метода предкомпиляции сократилось в 4,7 раза. Результаты метода ручной реализации на этом запросе отсутствуют по причине отсутствия поддержки всех встроенных функций, используемых в запросе.

Табл. 2. Сравнение времени выполнения на втором запросе.

Table 2. Comparison of execution times on the second test query.

	PostgreSQL	Предкомпиляция	Ручная реализация
Время (мс)	3341,431	711,278	—
Ускорение (раз)	1,00	4,70	—

8. Заключение

В данной работе рассмотрен метод динамической компиляции запросов как одно из средств, позволяющих значительно увеличить производительность СУБД на запросах, скорость обработки которых в первую очередь определяется эффективностью использования процессора.

Метод применён к существующей СУБД PostgreSQL. Рассмотрена компиляция оператора последовательного сканирования SeqScan и выражений оператора WHERE. Метод позволяет совместить производительность, свойственную компилируемым языкам, с развитой инфраструктурой расширений и богатыми возможностями, предоставляемыми PostgreSQL.

Результаты проведенного тестирования показывают, что динамическая компиляция запросов с помощью JIT-компилятора LLVM позволяет получить ускорение в несколько раз на синтетических тестах.

В будущем планируется добавить поддержку нескольких операторов в запросе. Эта задача требует:

1) разработки LLVM-аналогов всех операторов, реализованных в PostgreSQL;

2) замены абстракции итератора (open(), next(), close()) на абстракцию, более подходящую для генерации кода под конкретный запрос и позволяющую реализовывать новые операторы и совмещать несколько операторов в рамках одного запроса.

Изменение модели вычислений в сочетании с применением динамической компиляции позволит получить более эффективный код. Недостатками, свойственными (1), являются:

- трудоёмкость: по существу, (1) требует переписывания части исходного кода PostgreSQL, ответственного за вычисление планов;
- сложность поддержки: альтернативные реализации реляционных операторов требуют постоянной поддержки и обновления до новых версий PostgreSQL.

Ещё одним возможным направлением исследований является автоматическое изменение порядка следования атрибутов в таблице, так чтобы атрибуты, которые могут принимать значение NULL и атрибуты переменной длины шли строго после атрибутов фиксированной длины. Это позволит вычислять во время компиляции запроса смещения всех используемых атрибутов фиксированной длины и получить константное время доступа к ним во время выполнения. Недостатками данного подхода являются:

- ограниченная применимость: одним из свойств оптимизации, описанной в главе 6.1.2, является то, что она не накладывает никаких ограничений на используемые в запросе таблицы базы данных;

- ресурсоемкость преобразования: изменение порядка атрибутов для существующих таблиц требует полной перезаписи heap-файла и пересоздания всех индексов, определённых для данной таблицы;
- недостаточная эффективность: подобная оптимизация порядка следования атрибутов уже входит в типичные сборники советов для DBA и поэтому зачастую выполняется ими вручную.

Описанный в данной статье динамический компилятор запросов находится в стадии подготовки исходного кода для опубликования в открытом доступе (open-source).

Список литературы

[1]. Graefe G. Volcano — an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng.,6(1): 120–135, 1994.
[2]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[3]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[4]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.
[5]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.
[6]. Zhang R., Debray S., and Snodgrass R.T. Micro-specialization: dynamic code specialization of database management systems. In Code Generation and Optimization, pages 73, 2012.
[7]. Tan CK. Vitesse DB: 100% Postgres, 100X Faster For Analytics. https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub
[8]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[9]. Kornacker M., Behm A. и другие. Impala: A Modern, Open-Source SQL Engine for Hadoop. CIDR, 2015.
[10]. Armbrust M., Xin R. S. и другие. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 1383-1394, 2015.
[11]. Julia, a high-level, high-performance dynamic programming language for technical computing. http://julialang.org/
[12]. FTL JIT, JavaScriptCore's top-tier optimizing compiler. https://trac.webkit.org/wiki/FTLJIT
[13]. Варданян В.Г., Иванишин В.А., Асрян С.А., Хачатрян А.А, Акопян Дж. А. Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM. Труды ИСП РАН, том 27 (выпуск 6), 2015 г., стр. 33-48. DOI: 10.15514/ISPRAS-2015-27(6)-3
[14]. Pyston, an open-source Python implementation using JIT techniques. https://pyston.org
[15]. 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. http://www.macruby.org

[16]. MCJIT Design and Implementation.
http://llvm.org/docs/MCJITDesignAndImplementation.html
[17]. Momjian B. PostgreSQL Internals through Pictures.
http://momjian.us/main/writings/pgsql/internalpics.pdf

Dynamic compilation of expressions in SQL queries for PostgreSQL

[1] *E.Y. Sharygin <eush@ispras.ru>*
[1] *R.A. Buchatskiy <ruben@ispras.ru>*
[2] *L.V. Skvortsov <leonidxo@gmail.com>*
[1] *R.A. Zhuykov <zhroma@ispras.ru>*
[1] *D.M. Melnik <dm@ispras.ru>*
[1]*Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University, CMC Department*
bldg. 52, GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. In recent years, as performance and capacity of main and external memory grow, performance of database management systems (DBMSes) on certain kinds of queries is more determined by raw CPU speed. Currently, PostgreSQL uses the interpreter to execute SQL queries. This yields an overhead caused by indirect calls to handler functions and runtime checks, which could be avoided if the query were compiled into native code "on-the-fly", i.e. just-in-time (JIT) compiled: at run time the specific table structure is known as well as data types and built-in functions used in the query as well as the query itself. This is especially important for complex queries, performance of which is CPU-bound.
We've developed a PostgreSQL extension that implements SQL query JIT compilation using LLVM compiler infrastructure. In this paper we show how to implement JIT compilation to speed up sequential scan operator (SeqScan) as well as expressions in WHERE clauses. We describe some important optimizations that are possible only with dynamic compilation, such as precomputing tuple attributes offsets only for attributes used by the query.
We also discuss the maintainability of our extension, i.e. the automation for translating PostgreSQL backend functions into LLVM IR, using the same source code both for our JIT compiler and the existing interpreter.
Currently, with LLVM JIT we achieve up to 5x speedup on synthetic tests as compared to original PostgreSQL interpreter.

Keywords: dynamic compilation; just-in-time compilation; database management system engines; PostgreSQL; LLVM; query languages.

DOI: 10.15514/ISPRAS-2016-28(4)-13

For citation: Sharygin E.Y., Buchatskiy R.A., Skvortsov L.V., Zhuykov R.A., Melnik D.M. Dynamic compilation of expressions in SQL queries for PostgreSQL. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 217-240 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-13

References

[1]. Graefe G. Volcano — an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng.,6(1): 120–135, 1994.
[2]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[3]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[4]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.
[5]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.
[6]. Zhang R., Debray S., Snodgrass R.T. Micro-specialization: dynamic code specialization of database management systems. In Code Generation and Optimization, pages 73, 2012.
[7]. Tan CK. Vitesse DB: 100% Postgres, 100X Faster For Analytics. https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub
[8]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[9]. Kornacker M., Behm A. et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. CIDR, 2015.
[10]. Armbrust M., Xin R.S. et al. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 1383-1394, 2015.
[11]. Julia, a high-level, high-performance dynamic programming language for technical computing. http://julialang.org/
[12]. FTL JIT, JavaScriptCore's top-tier optimizing compiler. https://trac.webkit.org/wiki/FTLJIT
[13]. Vardanyan V., Ivanishin V., Asryan S., Khachatryan A., Hakobyan J. Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation. Trudy ISP RAN / Proc. ISP RAS], vol. 27, issue 6, 2015. pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-3
[14]. bbPyston, an open-source Python implementation using JIT techniques. https://pyston.org
[15]. 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. http://www.macruby.org
[16]. MCJIT Design and Implementation. http://llvm.org/docs/MCJITDesignAndImplementation.html
[17]. Momjian B. PostgreSQL Internals through Pictures. http://momjian.us/main/writings/pgsql/internalpics.pdf