Динамическая компиляция SQL-запросов для СУБД PostgreSQL

[1] *Р.А. Бучацкий <ruben@ispras.ru>*
[1, 2] *Е.Ю. Шарыгин <eush@ispras.ru>,*
[2] *Л.В. Скворцов <leonidxo@gmail.com>*
[1] *Р.А. Жуйков <zhroma@ispras.ru>,*
[1] *Д.М. Мельник <dm@ispras.ru>*
[3] *Р.В. Баев <baev@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.
[3] *Московский физико-технический институт (государственный университет)*
141701, Московская область, г. Долгопрудный, Институтский переулок, д.9

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. Для исполнения SQL-запросов в большинстве современных реляционных СУБД используется модель итераторов (Volcano-модель), которая удобна в реализации в рамках интерпретатора запросов, но сопряжена с существенными накладными расходами при выполнении плана, например, связанными с большим количеством ветвлений, неявными вызовами функций-обработчиков и выполнением лишних проверок, избежать которых довольно сложно при использовании механизма интерпретации. Одно из решений – динамическая компиляция запросов. В рамках данной работы рассматривается метод динамической компиляции запросов с применением альтернативной модели выполнения запроса в СУБД, что подразумевает отказ от используемой в PostgreSQL итеративной Volcano-модели, и его реализация для СУБД PostgreSQL с помощью компиляторной инфраструктуры LLVM. Динамический компилятор запросов 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 тестовом наборе TPC-H.

Ключевые слова: динамическая компиляция; JIT-компиляция; СУБД; PostgreSQL; LLVM; языки запросов.

DOI: 10.15514/ISPRAS-2016-28(6)-3

Для цитирования: Бучацкий Р.А., Шарыгин Е.Ю., Скворцов Л.В., Жуйков Р.А., Мельник Д.М., Баев Р.В. Динамическая компиляция SQL-запросов для СУБД PostgreSQL. Труды ИСП РАН, том 28, вып. 6, 2016, стр. 37-48. DOI: 10.15514/ISPRAS-2016-28(6)-3

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 внешней памяти, поскольку именно скорость доступа к данным обычно является узким местом при выполнении запросов. В последнее время, в связи с ростом объёмов и улучшением операционных характеристик доступа к оперативной памяти, стала актуальна задача более эффективного использования процессора.

Предлагаемый в данной работе метод динамической компиляции запросов позволит значительно увеличить производительность СУБД на запросах, скорость обработки которых в первую очередь определяется эффективностью использования процессора. Динамическая компиляция может быть выполнена с использованием информации, доступной только во время выполнения запроса, это позволяет генерировать машинный код, специализированный под конкретный запрос, и использовать модель выполнения, которая не накладывает дополнительных расходов во время выполнения. Этот подход 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 работе [1,2] описывается алгоритм генерации эффективного машинного кода для запросов к реляционной СУБД на языке SQL с использованием компиляторной инфраструктуры LLVM [3]. Аргументируется отказ от модели итераторов и приводятся экспериментальные данные, согласно которым использование динамической компиляции запросов позволяет добиться ускорения в 2 раза, а замена модели выполнения на модель явных циклов (data-centric) – ещё в 3–4 раза. Данные методы реализованы в коммерческой реляционной СУБД с закрытым исходным кодом HyPer [4].

Для СУБД PostgreSQL [5] разработано коммерческое расширение Vitesse DB [6] с закрытым исходным кодом, в котором реализована динамическая компиляция запросов с использованием LLVM. На запросе Q1 из набора тестов TPC-H [7] 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.

В [8] описывается метод компиляции выражений в PostgreSQL в машинный код с использованием инфраструктуры LLVM. В работе приведены результаты профилирования СУБД на тестовом наборе TPC-H, согласно которым вычисление арифметических выражений занимает от 32% до 70% от общего времени выполнения запроса. Исследователи не затрагивают вопросы оптимизации операторов и вызовов функций PostgreSQL, но предлагаемый в работе метод компиляции позволяет избежать повторной загрузки атрибутов из кортежа за счёт переиспользования регистров LLVM. На наборе тестов TPC-H представленный метод позволил получить ускорение до 37%.

В данной работе рассматривается динамическая компиляция SQL-запросов для СУБД PostgreSQL с помощью компиляторной инфраструктуры LLVM.

2. Динамическая компиляция запросов в PostgreSQL

Основной алгоритм выполнения SQL-запроса в реляционных СУБД состоит из четырёх этапов. На первом этапе СУБД выполняет лексический и синтаксический анализ SQL запроса и строит дерево разбора. На следующем этапе процедура преобразования принимает от анализатора дерево разбора и выполняет семантический анализ с дальнейшим построением дерева запроса.

На третьем этапе на основе дерева запроса составляется план выполнения запроса путем выбора наиболее эффективного пути выполнения. Итоговый план является наиболее эффективным с точки зрения имеющихся оценок затрат на его выполнение.

Финальным этапом является выполнение плана, которое реализовано при помощи модели итераторов, также известной как Volcano Style Processing [9]. Модель итераторов подробно описана в разделе 2.1.

JIT-компилятор запросов для PostgreSQL, о котором идёт речь в этой статье, реализован с использованием компиляторной инфраструктуры LLVM. Инфраструктура LLVM предоставляет богатый API для анализа и оптимизации программ, а во встроенном модуле MCJIT [10] реализованы механизмы для машинно-зависимой оптимизации и динамической генерации кода под различные платформы. В LLVM используется низкоуровневое типизированное платформонезависимое промежуточное представление LLVM IR, основанное на SSA-форме.

JIT-компилятор 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

операторы, функции и выражения, используемые в запросе, в случае чего производится динамическая компиляция и выполнение кода, оптимизированного под конкретный запрос.

При разработке JIT-компилятора было решено переписать с использованием LLVM C API основные операторы PostgreSQL. Несмотря на несколько возросшую сложность, такое переписывание позволило:

- пересмотреть используемую вычислительную модель – замена абстракции модели итераторов (Volcano-модели) на абстракцию, более подходящую для генерации кода под конкретный запрос (разделы 2.1, 2.2);
- динамически компилировать и оптимизировать код вычисления арифметических выражений и предикатов (раздел 2.3);
- спроектировать и реализовать ряд оптимизаций, возможных только в динамически компилируемом окружении [11].

2.1 Переход к альтернативной модели выполнения запросов

В используемой в большинстве современных СУБД, в том числе PostgreSQL, Volcano-модели, также известной как *pull-based*, каждый оператор реализуется при помощи итератора с интерфейсом *open(), next(), close()*. Метод *open()* инициализирует внешние ресурсы, такие как память и открытие файла, а метод *close()* освобождает их.

Выполнение плана запроса происходит рекурсивно сверху вниз по дереву, при этом каждый узел в дереве плана вызывает метод *next()* от узлов ниже для получения входных данных, обрабатывает и возвращает один кортеж на узел выше. Вызов метода *next()* каждого оператора в дереве запроса производит один новый кортеж, полученный путем рекурсивного вызова метода *next()* 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 позволяет упростить реализацию и поддержку реляционных операторов за счёт простоты абстракции итератора, но приводит к существенным накладным расходам, проявляющимся даже на простых запросах. Во-первых, метод *next()* реализован с помощью неявного вызова функции, который представляет из себя косвенный переход, и, как правило, вызывает ошибочное прогнозирование перехода (branch misprediction), что может привести к большим накладным расходам.

Во-вторых, для оператора сканирования таблиц необходимо сохранение состояния между вызовами *next()*, что означает, что для каждого считываемого из таблицы кортежа необходимо вначале загрузить переменные состояния, в том числе счётчики циклов, и только потом продолжить

выполнение с нужной итерации, при этом записав обновлённые значения переменных для последующих вызовов.

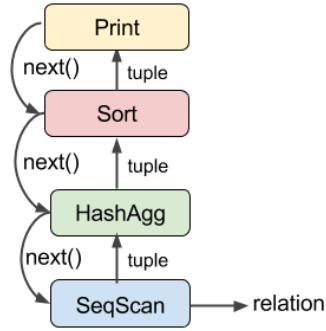


Рис. 1. Модель итераторов для запроса

Fig. 1. Iterator model for example query

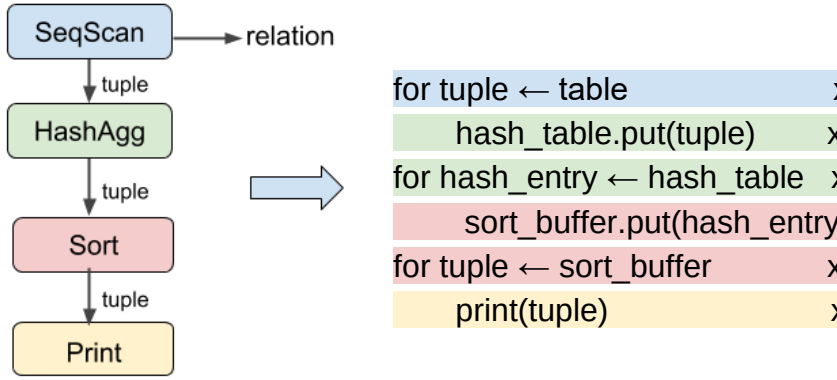


Рис. 2. Push-based модель для запроса

Fig. 2. Push-based model for example query

В плане выполнения запроса на примере СУБД PostgreSQL на рис. 1 для запроса вида select <columns> from <table> group by <column> order by <column>; узел Print будет обращаться за входными данными к дочернему узлу Sort, который, в свою очередь, будет обращаться к своему дочернему узлу HashAgg, который, в свою очередь, обратится к узлу SeqScan, представляющему последовательное чтение таблицы. В результате выполнения узла SeqScan исполнитель выберет одну строку из таблицы и вернет её вызывающему узлу HashAgg и так по цепочке до узла Print.

Альтернативой pull-based модели итераторов является модель явных циклов, также называемая push-based моделью (рис. 2).

Реализация этой модели позволяет представить счётчики циклов и другие переменные состояния локальными переменными на стеке или регистрах процессора и загружать их только при необходимости, но является существенно более сложной в реализации в рамках интерпретатора запросов.

Для JIT-компилятора запросов была выбрана именно push-based модель. Её реализации посвящен следующий раздел.

2.2 Реализация push-based модели в динамическом компиляторе

В конструкции предлагаемого в данной статье динамического компилятора запросов с использованием push-based модели генерация кода выполняется во время обхода дерева плана в прямом порядке, во время которого для каждого оператора вызываются функции *consume()* и *finalize()* и сгенерированные ими LLVM-функции передаются дочерним операторам запроса. Для каждого оператора соответствующие функции *consume()* и *finalize()* 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 будет состоять из нескольких циклов, самым первым из которых является цикл одного из операторов сканирования таблицы.

Для запроса, приведённого в разделе 2.1, для генерации кода на языке LLVM IR в JIT-компиляторе будут вызваны функции *Sort.consume* и *Sort.finalize*, результат выполнения которых будет использован в функциях *HashAgg.consume* и *HashAgg.finalize*, результат выполнения которых, в свою очередь, будет использован в функции SeqScan при генерации внешнего цикла по таблице (см. рис. 3).

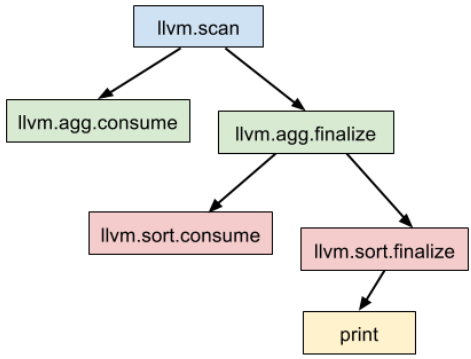
```
llvm.sort.consume = Sort.consume()
llvm.sort.finalize = Sort.finalize(print, null)
    llvm.agg.consume = HashAgg.consume()
    llvm.agg.finalize = HashAgg.finalize(llvm.sort.consume, llvm.sort.finalize)
        llvm.scan = SeqScan(llvm.agg.consume, llvm.agg.finalize)
```

Рис. 3. Цепочка вызовов функций-генераторов кода

Fig. 3. Calls to generator functions

Граф вызовов для сгенерированных LLVM-функции представлен на рис. 4 слева, а сами LLVM-функции – на рис. 4 справа. Предложенный подход позволяет избавиться от неявных вызовов функций и сохранения состояния

между вызовами. После встраивания в *llvm.scan* вызываемых ею функций (рис. 4, справа) получается код, эквивалентный показанному на рис. 2.




```
llvm.sort.consume(tuple) {
    sort_buffer.put(tuple)
}
llvm.sort.finalize() {
    for tuple ← sort_buffer
        print(tuple)
}
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.finalize() {
    for hash_entry ← hash_table
        llvm.sort.consume(tuple)
    llvm.sort.finalize()
}
llvm.scan() {
    for tuple ← table
        llvm.agg.consume(tuple)
    llvm.agg.finalize()
}
```

Рис. 4. Слева – граф вызовов LLVM-функций для запроса, справа – псевдокод LLVM-функций

Fig. 4. On the left: call graph for LLVM functions for example query, on the right: generated functions in pseudo-code

2.3 Динамическая компиляция выражений

Для вычисления выражений PostgreSQL выполняет интерпретацию дерева выражений, где каждое выражение состоит из дерева отдельных операторов и функций. Каждая вершина дерева вызывает функции соответствующих дочерних вершин неявным образом, через указатель на функцию. Это 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.

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 динамической компиляции выражений производится рекурсивный обход дерева выражений в обратном порядке и вызов функций-генераторов для генерации кода операций на языке LLVM IR.

В результате этого обхода генерируется код в виде функции на языке LLVM IR (ExecQual на рис. 5). С дальнейшим использованием оптимизации встраивания функций, код для дерева выражений становится линейным и

может быть динамически скомпилирован и выполнен без каких-либо накладных расходов на неявные вызовы функций.

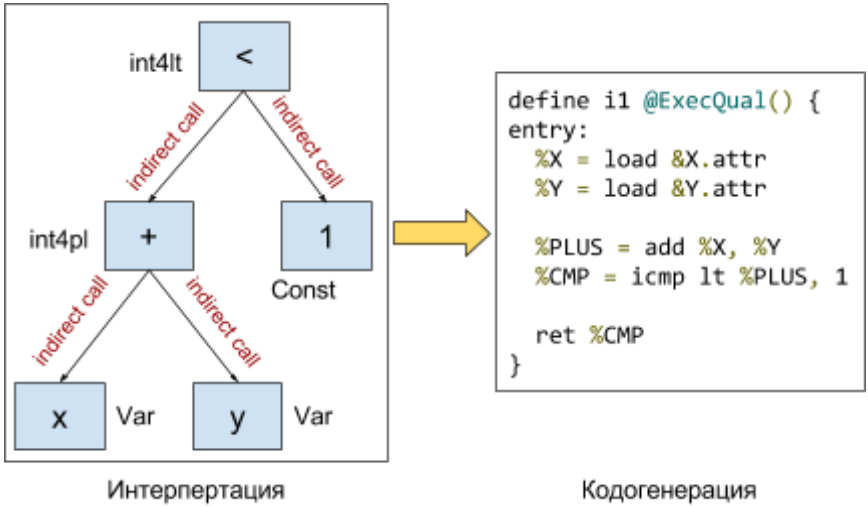


Рис. 5. Слева – интерпретация дерева выражений, справа – сгенерированный LLVM IR

Fig. 5. Left – interpretation of an expression tree, right – generated LLVM IR

Для вычисления операций вызываются встроенные функции PostgreSQL. Для получения функций-генераторов встроенных функций на языке LLVM IR был разработан метод предварительной компиляции с использованием библиотеки CPPBackend из состава LLVM (до версии 3.8), которая переводит LLVM-биткод в соответствующий код на языке C++, использующий функции LLVM C++ API для генерации модуля исходного кода LLVM IR.

Метод работает следующим образом: множество файлов исходного кода PostgreSQL, содержащих встроенные функции, с помощью компилятора *clang* [12] транслируется в объектные файлы биткода LLVM, которые компонуются в единый биткод-файл и оптимизируются модульным оптимизатором *opt* [13]. На основе оптимизированного биткода статический компилятор *llc* [14], в котором реализован интерфейс библиотеки CPPBackend (-march=cpp), строит файл на языке C++, содержащий функции-генераторы на LLVM C++ API, вызовы которых генерируют код на языке LLVM IR соответствующих встроенных функций PostgreSQL. Общая схема метода показана на рис. 6.

Стоит отметить, что генерация объектных (биткод) файлов, их компоновка, оптимизация, трансляция в C++-файл и дальнейшая компиляция этого файла происходят один раз во время сборки расширения.

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.

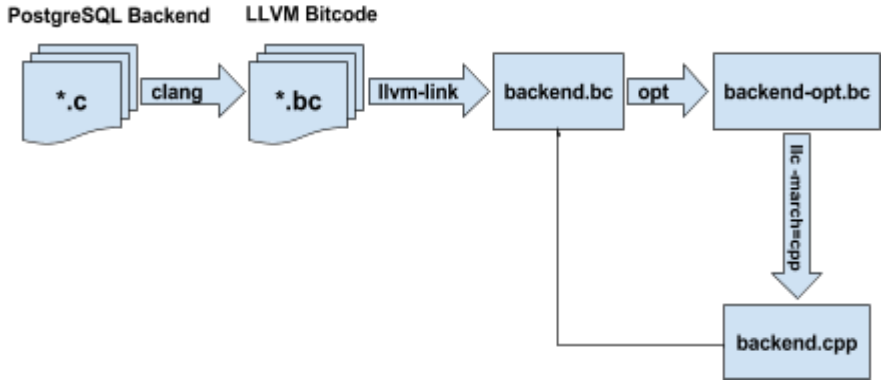


Рис. 6. Схема работы метода предкомпиляции функций PostgreSQL

Fig. 6. Scheme of the PostgreSQL backend function precompilation method

3. Результаты

Для тестирования производительности динамического компилятора (глава 2) использовался набор тестов TPC-H. Этот набор состоит из аналитических запросов и считается достаточно близким к реальным запросам к СУБД.

Тестирование производительности выполнялось на компьютере с 18-ядерным процессором Intel Xeon E7-8890 с тактовой частотой 2.5 ГГц и с 3 терабайтами оперативной памяти под управлением 64-битной операционной системы CentOS Linux release 7.2.1511. Размер базы данных TPC-H – 100 гигабайт, при тестировании база данных располагалась в оперативной памяти. Надо отметить, что типы колонок в таблицах базы данных TPC-H были модифицированы соответствующим образом: тип CHAR(1) был изменен на тип ENUM, тип NUMERIC на DOUBLE PRECISION. Данная модификация позволяет использовать встроенные типы LLVM во время динамической компиляции.

Табл. 1. Сравнение времени выполнения JIT компилятора (глава 2) на тестовом наборе TPC-H.

Table 1. Comparison of execution times of JIT compiler (chapter 2) on TPC-H benchmark.

TPC-H 100 Гбайт	Q1	Q3	Q6	Q13	Q14	Q17	Q19	Q22
PG, сек	431,81	212,06	112,52	252,17	127,36	163,56	9,03	16,47
JIT, сек	100,52	103,38	36,71	175,93	44,43	100,4	7,07	15,29
Ускорение, раз	**4,30**	**2,05**	**3,07**	1,43	**2,87**	1,63	1,28	1,08

Время выполнения измерялось путём многократного выполнения запроса и подсчёта медианы полученных результатов. Результаты тестирования для некоторых запросов из тестового набора TPC-H отражены в табл. 1. Таким образом, время выполнения запроса Q1 сократилось в 4,3 раза с использованием динамического компилятора запросов по сравнению с версией PostgreSQL 9.6 Beta 2 с отключенным параллелизмом.

Тестирование запросов Q6 и Q14 проводилось с отключенными операторами PostgreSQL BitmapHeapScan, Material и MergeJoin, так как данные операторы на момент написания статьи не были реализованы в динамическом компиляторе запросов.

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

В данной работе были разработаны и реализованы LLVM-аналоги основных операторов PostgreSQL, была заменена абстракция модели итераторов (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 выполнения в сочетании с применением динамической компиляции позволило получить более эффективный код.

Исходный код динамического компилятора выражений, описанного в разделе 2.3, опубликован в открытом доступе (open source) [15]. Компилятор запросов, включающий, помимо компиляции выражений, также измененную модель выполнения запроса (разделы 2.1, 2.2), находится в стадии подготовки к публикации.

Список литературы

[1]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.

[2]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.

[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]. HyPer – A Hybrid OLTP&OLAP High Performance DBMS, http://www.hyper-db.de

[5]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[6]. Vitesse DB, сайт проекта. http://vitessedata.com/vitesse-db
[7]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[8]. Butterstein D., Grust T. Precision Performance Surgery for PostgreSQL – LLVM-based Expression Compilation, Just in Time. Proceedings of the 42nd Int'l Conference on Very Large Databases (VLDB 2016), New Delhi, India, August 2016.
[9]. Graefe G. Volcano – an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng., 6(1): 120–135, 1994.
[10]. MCJIT Design and Implementation. http://llvm.org/docs/MCJITDesignAndImplementation.html
[11]. 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, стр. 217-240. DOI: 10.15514/ISPRAS-2016-28(4)-13
[12]. clang – a C language family frontend for LLVM, http://clang.llvm.org/
[13]. opt – modular LLVM optimizer and analyzer, http://llvm.org/releases/3.7.1/docs/CommandGuide/opt.html
[14]. llc – LLVM static compiler, http://llvm.org/releases/3.7.1/docs/CommandGuide/llc.html
[15]. PostgreSQL с динамической компиляцией выражений – исходные коды проекта на сайте github.com. https://github.com/ispras/postgres/tree/llvm-expressions

Dynamic compilation of SQL queries for PostgreSQL

[1] *R.A. Buchatskiy <ruben@ispras.ru>*
[1,2] *E.Y. Sharygin <eush@ispras.ru>*
[2] *L.V. Skvortsov <leonidxo@gmail.com>*
[1] *R.A. Zhuykov <zhroma@ispras.ru>*
[1] *D.M. Melnik <dm@ispras.ru>*
[3] *R.V. Baev <baev@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.*
[3] *Moscow Institute of Physics and Technology (State University) 9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, 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 have developed a PostgreSQL extension that implements SQL query JIT compilation using LLVM compiler infrastructure. In this paper we show how to implement LLVM-analogues of the main operators of the PostgreSQL, how to replace Volcano iterator model abstraction (open(), next(), close()) by the abstraction that is more suitable to generate code for a particular query. Currently, with LLVM JIT we achieve up to 4.3x speedup on TPC-H Q1 query 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(6)-3

For citation: Buchatskiy R.A., Sharygin E.Y., Skvortsov L.V., Zhuykov R.A., Melnik D.M., Baev R.V. Dynamic compilation of SQL queries for PostgreSQL. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 37-48 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-3

References

[1]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.
[2]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.
[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]. HyPer – A Hybrid OLTP&OLAP High Performance DBMS, http://www.hyper-db.de
[5]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[6]. Vitesse DB website. http://vitessedata.com/vitesse-db
[7]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[8]. Butterstein D., Grust T. Precision Performance Surgery for PostgreSQL – LLVM-based Expression Compilation, Just in Time. Proceedings of the 42nd Int'l Conference on Very Large Databases (VLDB 2016), New Delhi, India, August 2016.
[9]. Graefe G. Volcano – an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng., 6(1): 120–135, 1994.
[10]. MCJIT Design and Implementation. http://llvm.org/docs/MCJITDesignAndImplementation.html
[11]. 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
[12]. clang – a C language family frontend for LLVM, http://clang.llvm.org/
[13]. opt – modular LLVM optimizer and analyzer, http://llvm.org/releases/3.7.1/docs/CommandGuide/opt.html
[14]. llc – LLVM static compiler, http://llvm.org/releases/3.7.1/docs/CommandGuide/llc.html
PostgreSQL with JIT compiler for expressions – project source code at github.com website. https://github.com/ispras/postgres/tree/llvm-expressions