DOI: 10.15514/ISPRAS-2020-32(1)-11

Кэширование машинного кода в динамическом компиляторе SQL-запросов для СУБД PostgreSQL

М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <*pantlimon*@ispras.ru>
Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <*ruben@ispras.ru*>
Р.А. Жуйков, ORCID: 0000-0002-0906-8146 <*zhroma@ispras.ru*>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. По мере увеличения производительности компьютеров и роста объёма оперативной и внешней памяти производительность СУБД для некоторых классов запросов всё чаще определяется характеристиками процессора и эффективностью его использования. Для исполнения SQL-запросов в реляционных СУБД используется различные модели выполнения, которые различаются характеристиками, но так или иначе подвержены существенным накладным расходам при интерпретации плана запроса. Накладные расходы связаны с большим количеством ветвлений, неявными вызовами функций-обработчиков и выполнением лишних проверок. Одно из решений – динамическая компиляция запросов, которая оправдана только в том случае, когда время, затрачиваемое на интерпретацию запроса, превосходит время, затрачиваемое на компиляцию и выполнение оптимизированного кода. Данное требование может быть удовлетворено только тогда, когда объем обрабатываемых запросом данных достаточно велик. Если время интерпретации запроса исчисляется миллисекундами, то затраты на динамическую компиляцию могут в сотни раз превосходить время выполнения сгенерированного машинного кода. Чтобы оправдать расходы, затрачиваемые на динамическую компиляцию таких запросов, необходимо иметь возможность повторного использования сгенерированного машинного кода в последующих выполнениях, тем самым избавившись от затратных операций по его оптимизации и компиляции. В рамках данной работы рассматривается метод кэширования машинного кода в динамическом компиляторе запросов СУБД PostgreSQL. Предлагаемый метод позволяет избавиться от накладных расходов, затрачиваемых на оптимизацию и компиляцию. Результаты проведенного тестирования показывают, что динамическая компиляция запросов с возможностью переиспользования машинного кода позволяет получить существенное ускорение на запросах типа OLTP.

Ключевые слова: динамическая компиляция; JIT-компиляция; кэширование кода; выполнение запросов; СУБД; PostgreSQL; LLVM

Для цитирования: Пантилимонов М.В., Бучацкий Р.А., Жуйков Р.А. Кэширование машинного кода в динамическом компиляторе SQL-запросов для СУБД PostgreSQL. Труды ИСП РАН, том 32, вып. 1, 2020 г., стр. 205-220. DOI: 10.15514/ISPRAS–2020–32(1)–11

Благодарности: Авторы выражают благодарность Е.Ю. Шарыгину, Д.М. Мельнику и А.Н. Томилину за помощь в выполнении научной работы.

Machine code caching in PostgreSQL query JIT-compiler

M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
R.A. Zhuykov, ORCID: 0000-0002-0906-8146 <zhroma@ispras.ru>

Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. As the efficiency of main and external memory grows, alongside with decreasing hardware costs, the performance of database management systems (DBMS) on certain kinds of queries is more determined by CPU characteristics and the way it is utilized. Relational DBMS utilize diverse execution models to run SQL queries. Those models have different properties, but in either way suffer from substantial overhead during query plan interpretation. The overhead comes from indirect calls to handler functions, runtime checks and large number of branch instructions. One way to solve this problem is dynamic query compilation that is reasonable only in those cases when query interpretation time is larger than the time of compilation and optimized machine code execution. This requirement can be satisfied only when the amount of data to be processed is large enough. If query interpretation takes milliseconds to finish, then the cost of dynamic compilation can be hundreds of times more than the execution time of generated machine code. To pay off the cost of dynamic compilation, the generated machine code has to be reused in subsequent executions, thus saving the cost of code compilation and optimization. In this paper, we examine the method of machine code caching in our query JIT-compiler for DBMS PostgreSQL. The proposed method allows us to eliminate compilation overhead. The results show that dynamic compilation of queries with machine code caching feature gives a significant speedup on OLTP queries.

Keywords: dynamic compilation; JIT-compilation; machine code caching; query execution; DBMS; PostgreSQL; LLVM

For citation: Pantilimonov M.V., Buchatskiy R.A., Zhuykov R.A. Machine code caching in PostgreSQL query JIT-compiler. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 1, 2020. pp. 205-220 (in Russian). DOI: 10.15514/ISPRAS-2020-32(1)-11

Acknowledgments: The authors are grateful to E.Y. Sharygin, D.M. Melnik and A.N, Tomilin for their help with the research.

1. Введение

Традиционно в реляционных системах управления базами данных (РСУБД) пользовательский запрос транслируется сначала в логический план запроса, представляющий собой дерево из операторов расширенной реляционной алгебры, а затем в физический, путём добавления метаинформации о выбранных методах доступа к данным и алгоритмах, реализующих реляционные операции. Готовый физический план передается на исполнение, где осуществляется его интерпретация с использованием заданной модели выполнения. Классическим примером последней является модель итераторов, также известная как Volcano-модель [1]. В рамках данной модели каждый алгебраический оператор преобразовывает входные данные в выходной поток кортежей, который управляется при помощи функции *next(),* являющейся частью общего интерфейса. Данная абстракция проста для понимания и удобна в реализации, однако неэффективным образом использует ресурсы современных центральных процессоров.

Существуют и другие модели выполнения [2, 3, 4], которые пытаются нивелировать недостатки модели итераторов различным образом. Так или иначе, вне зависимости от используемой модели выполнения, классический способ выполнения запроса сопряжен 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 привлекается метод динамической компиляции, в рамках которого выполняется кодогенерация специализированного кода под заданный план запроса. Производительность достигается за счет встраивания функций, подстановки констант, вычисления арифметических выражений, замены косвенных вызовов на явные, удаления мёртвого, с точки зрения плана запроса, кода и другие. Метод динамической компиляции предполагает 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 и t_E – время выполнения. 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 [5], например, из тестового набора TPC-H [6], выполняющиеся на большом объеме данных, могут нивелировать время, затрачиваемое на компиляцию, и позволяют получить прирост производительности. В случае OLTP [7] запросов, например, из набора TPC-B [8], 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 путем сохранения и переиспользования сгенерированного машинного кода. Однако простого сохранения в общем случае недостаточно и необходимо выполнять кодогенерацию с возможностью применения патчей к сохраненному машинному коду по причине изменяемых значений и адресов структур в динамической памяти.

В данной работе рассматривается метод сохранения и переиспользования сгенерированного динамическим компилятором запросов машинного кода с целью уменьшения накладных расходов, затрачиваемых на компиляцию запросов. Работа выполняется с использованием компиляторной инфраструктуры LLVM [9] в динамическом компиляторе запросов [10, 11, 12] PostgreSQL [13], разрабатываемом в ИСП РАН [14].

2. Стандартный подход к автоматическому кэшированию запросов

В большинстве современных проприетарных РСУБД, таких как MS SQL (Microsoft), DB2 (IBM) и Oracle Database (Oracle), активным образом используется механизм кэширования планов исполняемых запросов в автоматическом режиме без использования явных синтаксических конструкций. Сохраненные планы запросов располагаются в общей памяти, которая может быть реализована по-разному в зависимости от используемой процессной модели. Таким образом, информация о планах запросов доступна всем обслуживающим клиентские подключения процессам/потокам и позволяет в общем случае уменьшить время отклика всей системы, увеличив ее производительность и пропускную способность за счет уменьшения накладных расходов на операции, связанные с построением физического плана для часто используемых запросов.

Однако у подхода автоматического кэширования в общей памяти имеются и свои минусы, в основном связанные со сложностью синхронизации доступа к разделяемым ресурсам, а

также большей чувствительностью системы к не оптимально построенным планам запросов, чем в случае локального кэширования, используемого в PostgreSQL и MySQL. Сохраненный план запроса может быть первоначально построен оптимизатором не оптимально из-за недостаточного объёма статистической метаинформации с асимметричным распределением данных, сложной конструкции самого запроса с большим количеством операций соединения, использования хранимых функций или процедур и т.д. Также может возникнуть ситуация, что оптимально построенный план запроса теряет свою актуальность после некоторого количества выполненных операций модификации базы данных, которые могли в различной степени изменить существующее распределение данных, от которого отталкивался оптимизатор при построении первоначального плана. До тех пор, пока данный план запроса не будет перестроен, все клиенты СУБД будут не оптимально расходовать ресурсы системы.

Для решения этой проблемы вендоры используют в своих СУБД различные подходы: автоматическое перестраивание плана запроса после некоторого процентного изменения данных в объекте (таблице) базы данных; сбор статистики во время выполнения, которая позволяет понять актуальность приблизительных оценок оптимизатора в момент первоначального построения; использование адаптивных планов запросов, которые могут заменять оператор-алгоритм в зависимости от количества получаемых данных, например, переходить от соединения по вложенным циклам на соединение по хэшу и т.д.

В общем случае задача динамической компиляции плана запроса слабо пересекается с проблемой его оптимального построения и может решаться независимо. По причине того, что операции по динамической генерации кода с его оптимизацией и компиляцией являются ресурсоемкими, амортизация стоимости затрачиваемых на них ресурсов становится важной задачей. В случае локального кэширования эта задача не может быть решена эффективно по причине того, что каждый процесс СУБД имеет доступ только к собственным сохраненным планам и не может переиспользовать результат работы другого процесса. В худшем случае каждый процесс может иметь абсолютную копию из некоторого набора часто выполняющихся запросов, каждый со своим динамически скомпилированным машинным кодом. При использовании такого подхода суммарные затраты на поддержание кэша планов будут линейно расти как по памяти, используемой для хранения собственной копии машинного кода плана, так и по тактам процессора, затрачиваемых на генерацию и компиляцию этой копии.

Таким образом, для реализации эффективного механизма кэширования динамически скомпилированных планов запросов в СУБД PostgreSQL его, как минимум, необходимо перенести из процесс-локальной памяти в разделяемую. Тем не менее, для преследуемых в данной работе целей указанное требование не является обязательным, и метод кэширования динамически сгенерированного машинного кода может быть исследован независимо.

3. Этапы обработки SQL запроса

Основной алгоритм выполнения SQL-запроса в реляционных СУБД состоит из следующих этапов:

1. Стадия лексического и синтаксического анализа. 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 выдана не будет.
2. Стадия семантического анализа. Дерево разбора, полученное на предыдущей фазе, 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. Стадия обработки системой правил. Далее выполняется поиск в системных каталогах правил, применимых к дереву запроса, и при обнаружении подходящих правил 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. Фаза планирования и оптимизации. Планировщик получает на вход структуру с деревом запроса. Используя вспомогательные структуры данных, называемыми путями, которые представляют собой упрощенные схемы планов, планировщик осуществляет выбор наиболее эффективного пути выполнения запроса с точки зрения имеющихся оценок затрат и статистической информации на момент выполнения. Производится выбор оптимального метода доступа к данным с заданным порядком соединений и алгоритмов для их выполнения. Выбранный вариант трансформируется в полноценный план запроса и передается исполнителю.
5. Фаза выполнения итогового плана запроса. Исполнитель осуществляет рекурсивный обход по дереву плана и выполняет инкапсулированную логику соответствующего узла-оператора или выражения, получая на выходе результирующее множество строк.

В большинстве СУБД, включая PostgreSQL, используется описанный подход для трансляции запроса пользователя в физический план запроса, пригодный для выполнения. В случае динамической компиляции к последней фазе добавляются накладные расходы, связанные с процессом кодогенерации, оптимизации и компиляции. Чтобы эти расходы были оправданы, итоговое время выполнения запроса в режиме интерпретации должно существенно превосходить время, затрачиваемое на динамическую компиляцию.

4. Анализ необходимости сохранения машинного кода

На рис. 1 представлен план, построенный оптимизатором СУБД PostgreSQL, для запроса Q1 из тестового набора TPC-H[1] в базе данных, сгенерированной с параметром SCALE=2. Генерация базы данных выполнялась с заменой типов CHAR(1) на ENUM и NUMERIC на DOUBLE PRECISION.

```
                           QUERY PLAN
-------------------------------------------------------------------
  Sort
    Sort Key: l_returnflag, l_linestatus
    ->  HashAggregate
        Group Key: l_returnflag, l_linestatus
        ->  Seq Scan on lineitem
            Filter: (l_shipdate <=
                '1998-09-29 00:00:00'::timestamp without time zone)
```

Рис. 1. План в PostgreSQL для запроса Q1 из набора TPC-H
Fig. 1. Query plan in PostgreSQL for query Q1 from TPC-H benchmark

В запросе Q1 оператор последовательного сканирования проверяет условие предиката для каждого кортежа таблицы lineitem, состоящей примерно из 12 млн. кортежей. Среднее время интерпретации данного запроса на машине с процессором Intel Core I7-6700HQ, когда база данных полностью располагается в основной памяти, составляет 7.7 секунд. Среднее суммарное время выполнения динамически скомпилированной версии этого же

[1] См. Q1 http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf

запроса составляет 2.1 секунды, где 350 мс занимает оптимизация сгенерированного во внутреннем представлении LLVM IR кода, 280 мс – его компиляция и 1.4 секунды – выполнение результирующего машинного кода. Таким образом, время выполнения запроса в режиме интерпретации существенно превосходит накладные расходы, затрачиваемые на динамическую компиляцию и выполнение сгенерированного машинного кода. Качество машинного кода в совокупности с размером обрабатываемых данных положительно сказываются на итоговом времени выполнения, что оправдывает ресурсы, затраченные на его генерацию.

Противоположную ситуацию можно наблюдать на примере следующего простого запроса:

```
select * from orders where o_custkey = 102022
      and o_orderdate between date '1992-11-01'
      and date '1994-01-01',
```

план которого представлен на рис. 2. Данный запрос извлекает данные о заказах клиента из соответствующей таблицы с использованием индекса и суммарно обрабатывает лишь небольшое количество кортежей. Среднее время его интерпретации на той же машине составляет примерно 0.110 мс. В случае динамической компиляции суммарное время составляет примерно 120 мс, а время выполнения сгенерированного машинного кода лишь 0.030 мс. Очевидно, что затраты на динамическую компиляцию данного запроса в сотни раз превосходят время его выполнения.

```
                           QUERY PLAN
-------------------------------------------------------------------
  Index Scan using i_o_custkey on orders
    Index Cond: (o_custkey = 102022)
    Filter: ((o_orderdate >= '1992-11-01'::date)
                      AND (o_orderdate <= '1994-01-01'::date))
```

Рис. 2. План в PostgreSQL для OLTP запроса
Fig. 2. Query plan in PostgreSQL for OLTP query

Можно сделать вывод, что для запросов такого вида стоимость динамической компиляции чрезвычайно высока, и чтобы оправдать ее использование, необходимо использовать сгенерированный машинный код повторно в последующих выполнениях, избавившись от затратных операций по его оптимизации и компиляции. В общем случае сохранение машинного кода должно осуществляться для запросов, которые СУБД самостоятельно кэширует с целью минимизации затрат, ассоциируемых с этапами его обработки до момента выполнения. В запросе, представленном на рис. 2, для вычисления результата используются литеральные значения-константы, что существенно уменьшает вероятность последующего повторного использования данного плана запроса с идентичными аргументами.

СУБД PostgreSQL не обладает функционалом по автоматическому кэшированию запросов, но предоставляет механизм ручного сохранения плана запроса в локальную память процесса, обслуживающего клиентское подключение. Таким образом, данный механизм может быть задействован для реализации возможности переиспользования сгенерированного машинного кода, ассоциируемого с конкретным планом запроса.

5. Генерация машинного кода с возможностью переиспользования

5.1 Кэширование плана запроса в PostgreSQL

Для кэширования плана запроса, поступающего в систему из внешнего источника, СУБД PostgreSQL предоставляет механизм его ручного сохранения с помощью команды PREPARE [15]. Эта команда позволяет создать подготовленный объект-оператор для

пользовательского запроса на стороне сервера, который проходит через первые три стадии стандартного процесса обработки, описанные в разд. 3: разбор, анализ и переписывание с использованием правил, а затем сохраняется в локальной памяти процесса – текущего сеанса работы с СУБД. Последующая работа с подготовленным объектом-оператором осуществляется путем использования команды EXECUTE [16], вместе с которой также передаются значения-параметры, если они были указаны в момент подготовки объекта. Для сохраненного объекта-оператора оптимизатор генерирует наилучший план в зависимости от переданных параметров, а затем передает его на выполнение. Таким образом, повторное использование подготовленного объекта-оператора позволяет нивелировать накладные расходы, затрачиваемые на первые три стадии обработки запроса: лексический, синтаксический и семантический анализ, а также обработку системой правил.

Подготовленный оператор может также использовать некоторый обобщенный (GENERIC) план, а не перестраивать его под каждый набор полученных параметров. Для подготовленных операторов без параметров это происходит сразу; иначе общий план выбирается после пяти и более выполнений, при которых получаются планы с ожидаемой средней стоимостью, превышающей оценку стоимости общего плана. Когда общий план выбран, используется до конца жизни подготовленного оператора. Обобщенный план запроса нивелирует все накладные расходы, ассоциируемые со всеми стадиями обработки запроса, кроме его выполнения путем интерпретации.

На рис. 3 представлен пример обобщенного плана для подготовленного командой PREPARE объекта-оператора:

```
PREPARE q1(int, date, date) as
        select * from orders where o_custkey = $1
                        and o_orderdate between $2 and $3;
```

```
                              QUERY PLAN
-------------------------------------------------------------------
 Index Scan using i_o_custkey on orders
   Index Cond: (o_custkey = $1)
   Filter: ((o_orderdate >= $2) AND (o_orderdate <= $3))
```

Рис. 3. Обобщенный план запроса в PostgreSQL
Fig. 3. Generic query plan in PostgreSQL

В данном случае оптимизатор PostgreSQL, используя накопленную статистику за первых 5 выполнений, посчитал, что стоимость обобщенного плана для данного запроса меньше, чем затраты на его планирование под каждый набор получаемых параметров. Значения $1, $2 и $3 соответствуют порядковым номерам параметров, указанных в команде PREPARE, и влияют на результат выполнения соответствующих фильтров-предикатов.

Таким образом, стандартный механизм PostgreSQL по созданию подготовленного объекта-оператора позволяет сохранить оптимизированный обобщенный план запроса для его последующего повторного переиспользования без расхода вычислительных ресурсов на стандартные фазы обработки. Данный механизм был использован в качестве базы для реализации возможности сохранения и переиспользования динамически скомпилированного машинного кода. Для этого потребовалось расширить существующие структуры PostgreSQL типа QueryDesc и CachedPlan таким образом, чтобы динамический компилятор запросов смог сохранить в них указатель на область памяти, содержащей сгенерированный машинный код. Сохраненная в заданных структурах информация

используется в динамическом компиляторе запросов для последующего переиспользования машинного кода с его предварительной модификацией, которая необходима для обновления адресов переменных PostgreSQL, используемых в процессе кодогенерации. Абсолютные адреса используемых структур и переменных PostgreSQL меняются после каждого выполнения сохраненного плана запроса и, соответственно, должны быть обновлены перед следующим запуском.

5.2 Инструменты LLVM для модификации машинного кода

Для реализации возможности переиспользования сгенерированного машинного кода некоторого обобщенного плана запроса необходимо располагать метаинформацией для установки соответствия между LLVM IR представлением, используемым в момент кодогенерации, и результирующим машинным кодом. Впоследствии данная информация может быть использована для модификации и патчинга динамически сгенерированных инструкций.

Для решения этой задачи инфраструктура LLVM предоставляет инструменты для контроля и модификации сгенерированного компонентом MCJIT [17] машинного кода – интринсики $llvm.experimental.stackmap$ и $llvm.experimental.patchpoint$. Оба этих интринсика во время компиляции представления LLVM в машинный код инициируют создание специальной секции данных, содержащей структуру $Stack\ Map$ [18]. В этой структуре сохраняется относительное смещение от начала функции в машинном коде, куда попадает вызов $stackmap/patchpoint$, а также местонахождение (слот стека, имя регистра, константа и т.п.) всех значений, переданных этим интринсикам в качестве параметров. Интринсик $llvm.experimental.patchpoint$, помимо тех же параметров, что и $llvm.experimental.stackmap$, принимает также адрес вызываемой функции. Помимо создания $Stack\ Map$, при компиляции $llvm.experimental.patchpoint$ в генерируемый код вставляется вызов этой функции в соответствии с заданным соглашением о вызовах. В дальнейшем, благодаря $Stack\ Map$, вызываемый объект можно будет подменить.

Для реализации метода кэширования сгенерированного машинного кода в динамическом компиляторе запросов используется только интринсик $llvm.experimental.patchpoint$, сигнатура которого представлена на рис. 4.

```
declare i64 @llvm.experimental.patchpoint.i64(i64<id>,
                i32<numBytes>, i8*<target>, i32<numArgs>, ...)
```

Рис. 4. Сигнатура интринсика llvm.experimental.patchpoint
Fig. 4. llvm.experimental.patchpoint intrinsic syntax

В процессе кодогенерации создаваемые инструкции в промежуточном представлении LLVM IR используют данные из структур PostgreSQL, располагающиеся в динамической памяти по некоторым абсолютным адресам, которые, в свою очередь, теряют актуальность после каждой итерации выполнения запроса. Таким образом, перед началом следующего выполнения сохраненного машинного кода необходимо актуализировать ранее используемые абсолютные адреса. В процессе динамической компиляции с целью переиспользования машинного кода для каждого обращения к полю структуры данных PostgreSQL осуществляется генерация вызова интринсика $llvm.experimental.patchpoint(ID, 13, 0x1234567890abcdef, 0)$, где ID – это уникальный идентификатор, 13 – количество резервируемых байт, $0x1234567890abcdef$ – адрес функции, которую будет вызывать сгенерированный код и 0 – количество параметров у вызываемой функции.

На рис. 5 представлен пример кодогенерации с использованием LLVM C API и его результирующее представление в LLVM IR и машинном коде.

В данном примере поле tts_nvalid принадлежит структуре $TupleTableSlot$, которая используется PostgreSQL для представления разных типов кортежей. Данная структура данных аллоцируется и освобождается при каждом выполнении плана запроса, вне зависимости от использования механизма ручного кэширования.

<table>
<tr><td>Вызов функций из LLVM C API внутри функции-генератора</td><td><pre>static LLVMValueRef
top_level_consume_codegen(LLVMModuleRef mod,
          LLVMBuilderRef builder, …){
  ...
  LLVMValueRef slot_nvalid_ptr;
  __LLVMPositionBuilderAtEnd(builder, entry_bb);
  slot_nvalid_ptr = GeneratePatchpoint(builder,
       __LLVMPointerType(__LLVMInt32TypeInContext
                   (llvm_ctx), 0),
       &inputslot->tupleslot->tts_nvalid);
  __LLVMBuildStore(builder,
       __LLVMConstNull(__LLVMInt32TypeInContext
               (llvm_ctx)), slot_nvalid_ptr);
  …
}</pre></td></tr>
<tr><td>Сгенерированный LLVM IR</td><td><pre>define internal i32 @llvm_top_level_consume() {
entry:
  %pp_ret = call i64 (i64, i32, i8*, i32, ...)
@llvm.experimental.patchpoint.i64(
      i64 0, i32 13, i8* inttoptr
           (i64 1311768467294899695 to i8*), i32 0)
  %pp_ret_pointer = inttoptr i64 %pp_ret to i32*
  store i32 0, i32* %pp_ret_pointer
  …
  ret i32 0
}</pre></td></tr>
<tr><td>Сгенерированный машинный код</td><td><pre><main+1966>:   49 bb ef cd ab 90 78 56 34 12
          movabs $0x1234567890abcdef,%r11
<main+1976>:   41 ff d3
callq *%r11
<main+1979>:   c7 00 00 00 00 00
movl $0x0,(%rax)</pre></td></tr>
<tr><td>Пропатченный машинный код</td><td><pre><main+1966>:   49 bb ef cd ab 90 78 56 34 12
          movabs $0x55b5b3195148,%rax
<main+1976>:   66 66 90
data16 xchg %ax,%ax
<main+1979>:   c7 00 00 00 00 00
movl $0x0,(%rax)</pre></td></tr>
</table>

Рис. 5. Пример кодогенерации с использованием интринсика $llvm.experimental.patchpoint$
Fig. 5. Example of code generation using $llvm.experimental.patchpoint$ intrinsic.

Адрес поля tts_nvalid в данной итерации кодогенерации равен $0x55b5b3195148$ и запоминается внутри функции $GeneratePatchpoint$ в глобальный массив $llvm_pp[]$ с индексом $llvm_pp_n$, который затем используется в качестве ID аргумента интринсика $llvm.experimental.patchpoint$. Впоследствии переданный индекс в функцию-интринсик будет сохранен в структуру $StkMapRecord$ внутри $Stack\ Map$ как $PatchPoint\ ID$. При последующем разборе структуры $Stack\ Map$ извлекаемое значение-индекс из поля $PatchPoint\ ID$ структуры $StkMapRecord$ позволит извлечь соответствующий адрес из массива $llvm_pp$ и использовать его для модификации кода. Результат выполнения патчинга представлен на рис. 5, где target-адрес вызываемой функции заменяется на адрес поля tts_nvalid, регистр $\%r11$ на $\%rax$, а вызов $callq$ на $nop\ (xchg\ \%ax, \%ax)$. Пропатченный код сохраняет семантику с точки зрения дальнейших инструкций, где работа осуществляется со значением регистра $\%rax$.

5.3 Реализация механизма модификации машинного кода в динамическом компиляторе запросов

В конструкции рассматриваемого в статьях [10, 11] динамического компилятора запросов с измененной моделью выполнения 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 машинного кода важно было избежать дублирования логики и сохранить существующий алгоритм кодогенерации в функциях-генераторах с целью упрощения дальнейшего процесса разработки и поддержки. Для реализации возможности генерации машинного кода с возможностью переиспользования были выполнены следующие изменения в существующем алгоритме кодогенерации.

- Добавлено глобальное состояние-режим кодогенерации – переменная $llvm_patchpoint$, которая задает поведение внутри функций-оберток над LLVM C API, функции $GeneratePatchpoint$ и некоторых других.
- Все использующиеся для кодогенерации функции из LLVM C API обернуты в функцию-обертку с аналогичным названием и дополнительным префиксом, которая возвращает разный результат в зависимости от глобального режима кодогенерации. Пример функции-обертки представлен на рис. 6.
- Все вызовы функции $LLVMConstIntToPtr$ из LLVM C API заменены на специальную функцию $GeneratePatchpoint$, псевдокод которой представлен на рис. 7.

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

Рис. 6. Пример функции-обертки над LLVM C API
Fig. 6. Example of LLVM C API wrapper function.

Перечислим возможные режимы кодогенерации.

- Одноразовая кодогенерация: машинный код будет использован один раз. Значение $llvm_patchpoint$ равно 0.
- Режим патчинга: кодогенерация не выполняется, а производится сбор и сохранение новых абсолютных адресов структур данных PostgreSQL для модификации сохраненного машинного кода. Значение $llvm_patchpoint$ равно 1.

- Режим кодогенерации с возможностью патчинга генерируемого машинного кода: в ходе кодогенерации сохраняются абсолютные адреса структур данных PostgreSQL как в режиме патчинга, а также генерируются вызовы интринсика $llvm.experimental.patchpoint$. После завершения процесса кодогенерации выполняется модификация сохраненного машинного кода. Значение $llvm_patchpoint$ равно 2.

В режиме одноразовой кодогенерации функции обёртки над функциями LLVM C API возвращают результат вызова соответствующей LLVM функции, а $GeneratePatchpoint()$ возвращает результат вызова функции $LLVMConstIntToPtr$ без запоминания переданного адреса-аргумента.

В режиме патчинга выполняется обход плана запроса с использованием функций-генераторов, в ходе которого все вызовы функций-обертков над LLVM C API возвращают пустое значение, т. е. отсутствуют какая-либо кодогенерация. Единственная выполняемая работа – это сохранение адресов аргументов внутри функции $GeneratePatchpoint$ в глобальный массив $llvm_pp[]$. В конце выполняется патчинг машинного кода с использованием информации из структуры Stack Map.

В режиме кодогенерации с возможностью патчинга также выполняется обход плана запроса, где функция $GeneratePatchpoint$ запоминает адрес-аргумент в глобальный массив $llvm_pp[]$, а затем возвращает результат кодогенерации вызова интринсика $llvm.experimental.patchpoint$ с предварительным приведением к указателю нужного типа. После завершения процесса кодогенерации осуществляется патчинг адресов аналогично предыдущему режиму, а также одноразовое изменение инструкций, представленное ранее на рис. 5.

```
LLVMValueRef
GeneratePatchpoint(LLVMBuilderRef builder,
                   LLVMTypeRef type, void *address)
{
    if (llvm_patchpoint == 0)
        return LLVMConstIntToPtr(builder, address, type);
    // save address to the global array and use same index for
intrinsic in case of patchpoint gen.
    llvm_pp[llvm_pp_n] = (uintptr_t) address;
    if (llvm_patchpoint == 2)
    {
        args = { llvm_pp_n++, 13, 0x1234567890abcdef, 0 };
        ret = LLVMBuildCall(builder,
"llvm.experimental.patchpoint.i64", args);
        return LLVMBuildIntToPtr(builder, ret, type);
    }
    Assert(llvm_patchpoint == 1);
    llvm_pp_n++;
    return NULL;
}
```

Рис. 7. Псевдокод функции $GeneratePatchpoint()$
Fig. 7. $GeneratePatchpoint()$ *pseudocode*

Процесс патчинга, применяемый в описанных режимах, визуально представлен на рис. 8 и выполняется следующим образом:

1. Из структуры $Stack\ Map$ последовательно извлекается информация о количестве записей в массиве $StkSizeRecord$, каждая запись которого содержит адрес функции, размер стека и количество записей $StkMapRecord$.

2. Из каждой записи $StkMapRecord$ извлекается значение-индекс – $PatchPoint\ ID$, которое используется для поиска ячейки в массиве адресов, ранее собранных в процессе обхода плана запроса. Данный этап соответствует шагу 1 на рис. 8.
3. Помимо значения-индекса из $StkMapRecord$ также извлекается значение-отступ – $Instruction\ Offset$. Адрес функции и отступ позволяют получить указатель на область памяти, зарезервированную при генерации $llvm.experimental.patchpoint$. Данный этап соответствует шагу 2 на рис. 7.
4. На последнем этапе, который соответствует шагу 3 на рис. 7, выполняется патчинг с использованием нового адреса из массива $llvm_pp[]$.

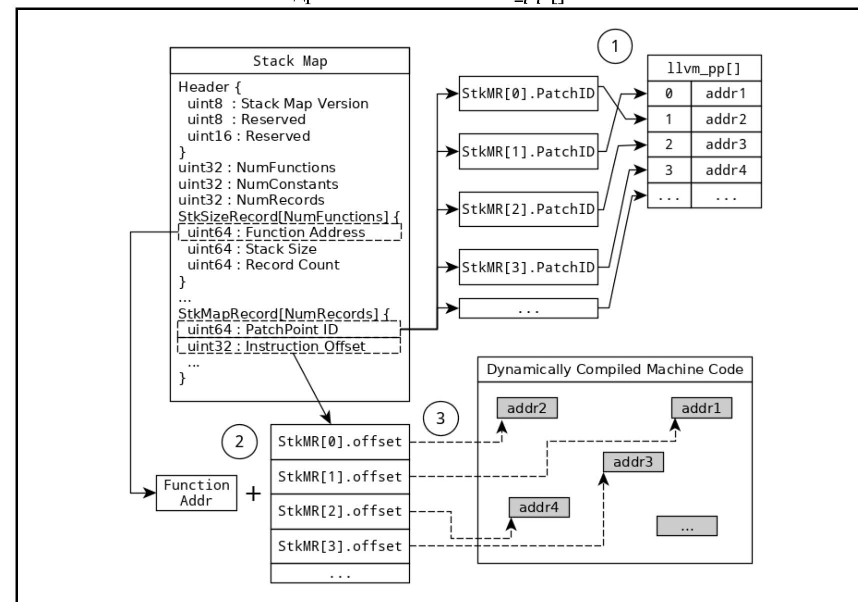

Рис. 8. Схема выполнения патчинга машинного кода
Fig. 8. Scheme of machine code patching

6. Результаты

Тестирование метода кэширования сгенерированного машинного кода с возможностью переиспользования осуществлялось на запросе Q1 из тестового набора TPC-H и запросах типа OLTP, представленных в табл. 1.

Для тестирования производительности механизма сохранения и патчинга сгенерированного машинного кода использовалась база данных из тестового набора TPC-H. Типы колонок базы данных были модифицированы следующим 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. База данных генерировалась с параметром SCALE=2. Суммарный объем директории с базой данных составил 6,4 Гб.

Тестирование производительности выполнялось на компьютере с четырёхъядерным процессором Intel Core i7-6700HQ с ограничением тактовой частоты в 2.5 ГГц и с 16 гигабайтами оперативной памяти под управлением 64-битной операционной системы Ubuntu Linux версии 18.04. При тестировании база данных полностью располагалась в

оперативной памяти. Сравнение производительности интерпретатора и компилятора выполнялось с использованием СУБД PostgreSQL версии 9.6.3.

Для сбора статистических данных о результате выполнения запросов из табл. 1 использовалась программа $pgbench$ [19] – утилита, входящая в состав проекта PostgreSQL. Для каждого запроса выполнялось несколько запусков утилиты $pgbench$: $pgbench\ -n\ -M\ prepared\ -t\ 10000\ -l\ -f\ q[1,2,3].script\ -z\ 1$, где z – добавленный путём модификации исходного кода флаг, позволяющий выполнять инициализацию генератора случайных чисел одинаковым образом, а $q[1,2,3].script$ – файл с запросом.

Табл 1. SQL запросы для тестирования производительности метода кэширования
Table 1. SQL queries to test the performance of the machine code caching method

№ запроса	Текст запроса

1	`select customer.c_custkey, customer.c_name,`
	`customer.c_phone, customer.c_acctbal,`
	`orders.o_orderstatus, orders.o_totalprice,`
	`orders.o_orderdate, orders.o_clerk,`
	`lineitem.l_linenumber, lineitem.l_quantity,`
	`lineitem.l_discount, lineitem.l_tax,`
	`lineitem.l_shipdate, partsupp.ps_availqty,`
	`partsupp.ps_supplycost, part.p_name,`
	`part.p_brand, part.p_retailprice,`
	`supplier.s_name, supplier.s_address,`
	`supplier.s_phone`
	`from customer join orders on c_custkey = o_custkey`
	`join lineitem on l_orderkey = o_orderkey`
	`join partsupp on ps_partkey = l_partkey`
	`and ps_suppkey = l_suppkey`
	`join part on p_partkey = ps_partkey`
	`join supplier on s_suppkey = ps_suppkey`
	`where c_custkey between :bid1 and :bid1 + 20`
	`order by o_orderdate desc;`

2	`select l_returnflag, l_linestatus,`
	`sum(l_quantity), sum(l_extendedprice),`
	`sum(l_extendedprice * (1 - l_discount)),`
	`sum(l_extendedprice * (1 - l_discount)`
	`* (1 + l_tax)),`
	`avg(l_quantity), avg(l_extendedprice),`
	`avg(l_discount), count(*) as count_order`
	`from lineitem`
	`where l_shipdate <= date '1998-12-01'`
	`- interval '105 days'`
	`and l_partkey between :bid1 and :bid1 + 200`
	`group by l_returnflag, l_linestatus`
	`order by l_returnflag, l_linestatus;`

3	`select l_returnflag, l_linestatus,`
	`sum(l_quantity) as sum_qty,`
	`avg(l_discount) as avg_disc,`
	`count(*) as count_order`
	`from lineitem where l_shipdate <= date '1998-12-01'`
	`- interval '105 days'`
	`and l_partkey between :bid1`
	`and :bid1 + 200`
	`group by l_returnflag, l_linestatus`
	`order by l_returnflag, l_linestatus;`

Среднее число транзакций в секунду было получено в результате выполнения утилиты $pgbench$. Среднее время выполнения подсчитывалось на основе генерируемых pgbench лог-файлов. Результаты тестирования запросов из табл. 1 на 10000 транзакций отражены в табл. 2. Протокол $prepared$ в утилите $pgbench$ использует механизм кэширования, описанный в 4.1. Динамическая компиляция запроса с возможностью переиспользования машинного кода выполняется в момент создания оптимизатором PostgreSQL обобщенного (GENERIC) плана, т.е. при выполнении 6-ой транзакции. Дальнейшее выполнение запроса осуществляется путем повторного использования сгенерированного машинного кода. Таким образом, значение со средним количеством транзакций в секунду, подсчитанное утилитой $pgbench$, включает расходы, связанные с динамической компиляцией в 6-ой транзакции. Максимальное среднее ускорение без учета первых 6-ти транзакций в 1,78 раз было получено на запросе 2, где присутствует наибольшее число выражений.

Помимо тестирования запросов типа OLTP из табл. 1 был также протестирован запрос Q1 из набора TPC-H, результаты которого представлены в табл. 3. Сравнительное тестирование запроса Q1 в динамическом компиляторе запросов выполнялось с целью анализа влияния интринсика $llvm.experimental.patchpoint$ на качество результирующего машинного кода.

Табл. 2. Сравнение времени выполнения JIT компилятора с кэшированием машинного кода на тестовых запросах из табл. 1
Table 2. Comparison of the execution time of JIT compiler with machine code caching on test queries from table 1

Наименование единицы измерения	Запрос 1, vanilla PG	Запрос 1, JIT	Запрос 2, vanilla PG	Запрос 2, JIT	Запрос 3, vanilla PG	Запрос 3, JIT
Среднее кол-во транзакций в секунду (больше – лучше)	70,67	72,38	105.25	183,71	145,37	199,83
Компиляция обобщенного плана на 6-ой итерации, мс	-	1342,5	-	1118,9	-	997,2
Среднее время выполнения без учета первых 6 итераций, мс	14,12	13,65	9,49	5,31	6,87	4,89
Среднее ускорение выполнения без учета первых 6 итераций, X раз	1,03		1,78		1,40	

Среднее время выполнения запроса Q1 с использованием интерпретатора PostgreSQL составило 10 секунд, а время выполнения динамически скомпилированной версии этого же запроса составило 2.65 секунды, где 820 миллисекунд затрачивается на оптимизацию и компиляцию, а 1.73 секунд на выполнение машинного кода.

Табл. 3. Сравнение времени выполнения запроса Q1 из набора TPC-H в интерпретаторе PostgreSQL и динамическом компиляторе в режимах одноразовой кодогенерации и кэширования машинного кода

Table 3. Comparison of execution time of query Q1 from TPC-H benchmark in PostgreSQL interpreter and JIT compiler in one-time code generation mode and machine code caching.

vanilla, PG	LLVM JIT			LLVM JIT + PREPARE		
	компиляция + оптимизация	выполнение	сумма	компиляция + оптимизация	выполнение	сумма
10 сек	(370 + 450) мс	1.73 сек	2.65 сек	(380 + 560) мс	2.4 сек	3.4 сек
				0.140 мс	2.4 сек	2.4 сек

Динамическая компиляция подготовленного плана запроса с возможностью переиспользования сгенерированного машинного кода суммарно составила 3.4 сек, а его среднее время выполнения на всех итерациях 2.4 сек. В случае использования подготовленного плана запроса накладные расходы на компиляцию и оптимизацию на всех итерациях после подготовки близки к 0.

Таким образом производительность машинного кода, сгенерированного с использованием $llvm.experimental.patchpoint$, в среднем на 38% меньше, чем результат одноразовой кодогенерации. Это связано с тем, что использование $llvm.experimental.patchpoint$ ограничивает возможности компилятора по выполнению оптимизаций над LLVM IR.

7. Заключение

В рамках данной работы был разработан метод сохранения и переиспользования сгенерированного динамическим компилятором запросов машинного кода, позволяющий нивелировать накладные расходы на его создание при повторном использовании. Возможность повторной утилизации сгенерированного машинного кода позволяет применять метод динамической компиляции для SQL запросов типа OLTP, время интерпретации которых измеряется миллисекундами.

Метод реализован в динамическом компиляторе запросов СУБД PostgreSQL с использованием технологии $Stack\ Map$ из инфраструктуры LLVM. Результаты проведенного тестирования показывают, что динамическая компиляция запросов с помощью JIT-компилятора LLVM с возможностью дальнейшего переиспользования результирующего машинного кода позволяет получить существенное ускорение на OLTP-запросах с достаточным количество выражений.

В будущем планируется расширить существующий механизм сохранения и переиспользования сгенерированного машинного кода, реализовав его автоматическое сохранение для часто выполняющихся однотипных запросов с использованием стоимостных оценок и эвристик.

Список литературы / References

[1]. Graefe G. Volcano – an extensible and parallel query evaluation system. IEEE Transactions on Knowledge and Data Engineering, vol. 6, issue 1, 1994, pp. 120–135.

[2]. Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. Proceedings of the VLDB Endowment, vol. 2, 2009, pp. 1648-1653.

[3]. S. Padmanabhan, T. Malkemus, A. Jhingran and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In Proc. of the 17th International Conference on Data Engineering, 2001, pp. 567-574.

[4]. Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment, vol. 4, no. 9, 2011, pp. 539-550.

[5]. А.Н.Андреев. Классификация OLAP-систем вида xOLAP / A.N. Andreev. OLAP systems of XOLAP type classification. Available at: http://citforum.ru/consulting/BI/xolap_classification/, accessed: 25.07.2019 (in Russian).

[6]. TPC-H benchmark for testing OLAP workload. Available at: http://www.tpc.org/tpch/, accessed 25.07.2019.

[7]. What is an OLTP System? Available at: https://docs.oracle.com/database/121/VLDBG/GUID-0BC75680-5BD4-43A9-826F-CD8837D30EB2.htm#VLDBG1367, accessed: 25.07.2019.

[8]. TPC-B benchark for testing OLTP workload. Available at: http://www.tpc.org/tpcb/, accessed: 25.07.2019.

[9]. The LLVM Compiler Infrastructure. Available at: http://llvm.org/, accessed: 25.07.2019.

[10]. 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 / 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

[11]. 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 / 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

[12]. E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher. Runtime Specialization of PostgreSQL Query Executor. Lecture Notes in Computer Science, vol. 10742, pp. 375–386, 2018.

[13]. PostgreSQL official site. Available at: https://www.postgresql.org/, accessed: 25.07.2019.

[14]. ISP RAS website. Available at: https://www.ispras.ru/, accessed: 25.07.2019.

[15]. PREPARE command, PostgreSQL. Available at: https://www.postgresql.org/docs/9.6/sql-prepare.html, accessed: 25.07.2019.

[16]. EXECUTE command, PostgreSQL. Available at: https://www.postgresql.org/docs/9.6/sql-execute.html, accessed: 25.07.2019.

[17]. MCJIT Design and Implementation. Available at: https://releases.llvm.org/4.0.0/docs/MCJITDesignAndImplementation.html, accessed: 25.07.2019.

[18]. Stack maps and patch points in LLVM. Available at: https://llvm.org/docs/StackMaps.html, accessed: 25.07.2019.

[19]. pgbench utility. Available at: https://www.postgresql.org/docs/9.6/pgbench.html, accessed: 25.07.2019.

Информация об авторах / Information about authors

Михаил Вячеславович ПАНТИЛИМОНОВ – стажер-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, СУБД.

Michael Vyacheslavovich PANTILIMONOV – Researcher in Compiler Technology department. Research interests: compiler technologies, DBMS.

Рубен Артурович БУЧАЦКИЙ – младший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Роман Александрович ЖУЙКОВ – научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Roman Aleksandrovich ZHUYKOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.