DOI: 10.15514/ISPRAS–2020–32(5)–5

Динамическая компиляция пользовательских функций на языке PL/pgSQL

[1,2] *В.М. Джиджоев, ORCID: 0000-0002-0967-6766 <dzhivl@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,2] *А.Н. Томилин, ORCID: 0000-0002-4040-4179 <tom@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова 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*

Аннотация. Многие современные СУБД предоставляют процедурное расширение декларативного языка программирования SQL, которое позволяет выполнять сложные вычисления с условной логикой на стороне сервера. Данные расширения стимулируют модулярность и переиспользование кода, упрощают процесс программирования логики некоторых приложений, а также позволяют избавиться от накладных расходов на передачу данных по сети и повысить производительность вычислений. В большинстве случаев для исполнения SQL-запросов и процедурных расширений используется подход интерпретации, который сопряжен с существенными накладными расходами по неявному вызову функций и выполнению лишних обобщенных проверок. Ситуация ухудшается тем, что для выполнения SQL-запросов и процедурных расширений, как правило, применяются два разных движка-исполнителя, в рамках которых необходимо осуществлять дополнительные операции по переключению контекста для сохранения промежуточных вычислений и контроля используемых ресурсов. В совокупности, совместная интерпретация SQL-запросов и процедурных расширений может в значительной степени снижать общую производительность СУБД. Одно из решений этой проблемы – динамическая компиляция. В рамках данной работы рассматривается метод динамической компиляции процедурного расширения PL/pgSQL в СУБД PostgreSQL с использованием компиляторной инфраструктуры LLVM. Работа выполняется в рамках динамического компилятора SQL-запросов, реализованного в виде расширения к СУБД PostgreSQL. Предлагаемый метод позволяет избавиться от накладных расходов, связанных с интерпретацией. Результаты тестирования на некоторых синтетических тестах показывают ускорение выполнения SQL-запросов в несколько раз.

Ключевые слова: динамическая компиляция; JIT-компиляция; выполнение запросов; UDF; СУБД; PostgreSQL; PL/pgSQL; LLVM

Для цитирования: Джиджоев В.М., Бучацкий Р.А., Пантилимонов М.В., Томилин А.Н. Динамическая компиляция пользовательских функций на языке PL/pgSQL. Труды ИСП РАН, том 32, вып. 5, 2020 г., стр. 67-80. DOI: 10.15514/ISPRAS–2020–32(5)–5

Благодарности: Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 20-07-00877 А

Dynamic Compilation of User-Defined Functions in PL/pgSQL Language

[1,2] *V.M. Dzhidzhoev, ORCID: 0000-0002-0967-6766* <dzhivl@ispras.ru>
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,2] *A.N. Tomilin, ORCID: 0000-0002-4040-4179 <tom@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. Many modern RDBMS provide procedural extensions for SQL programming language, which allow users to perform server-side complex computations. Use of such extensions improves modularity and code reuse, simplifies programming of application logic, and helps developers to avoid network overhead and enhance performance. Interpretation is mostly used to execute SQL queries and procedural extensions code, resulting in significant computational overhead because of indirect function calls and performing of generic checks. Moreover, most RDBMS use different engines for SQL queries execution and procedural extensions code execution, and it is necessary to perform additional computations to switch between different engines. Thus, interpretation of SQL queries combined with interpretation of procedural extensions code may drastically degrade performance of RDBMS. One solution is to use a dynamic compilation technique. In this paper, we describe the technique of dynamic compilation of PL/pgSQL procedural language for the PostgreSQL database system using LLVM compiler infrastructure. Dynamic compiler of PL/pgSQL procedural language is developed as part of PostgreSQL queries dynamic compiler. Proposed technique helps to get rid of computational overhead caused by interpretation usage. Synthetic performance tests show that the developed solution speeds up SQL queries execution by several times.

Keywords: dynamic compilation; JIT-compilation; query execution; UDF; DBMS; PostgreSQL; PL/pgSQL, LLVM

For citation: Dzhidzhoev V.M., Buchatskiy R.A., Pantilimonov M.V., Tomilin A.N. Dynamic compilation of user-defined functions in PL/pgSQL language. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 5, 2020, pp. 67-80 (in Russian). DOI: 10.15514/ISPRAS–2020–32(5)–5

Acknowledgements. This work was supported by the Russian Foundation for Basic Research, project № 20-07-00877 A

1. Введение

При разработке приложений, использующих СУБД, в последние годы стало актуальным использование пользовательских функций (UDF) – фрагментов кода, хранимых в базе данных, которые можно вызывать из SQL-запросов. Одним из главных преимуществ использования пользовательских функций является механизм их исполнения – код пользовательских функций исполняется на стороне сервера СУБД. Такой механизм позволяет сократить обмен данными между сервером СУБД и приложением-клиентом, и, таким образом, увеличить эффективность использования СУБД. При этом, заметную долю времени выполнения SQL-запросов, содержащих вызовы пользовательских функций, занимает интерпретация кода пользовательских функций.

В данной работе предлагается подход, позволяющий уменьшить время выполнения таких запросов в СУБД PostgreSQL [1] с помощью динамической компиляции пользовательских функций. Альтернативным направлением оптимизаций времени выполнения пользовательских функций являются работы [2-4] посвященные автоматической трансформации процедурной логики в реляционные алгебраические выражения, которые затем встраиваются в тело SQL-запроса. При таком подходе планировщик запросов может использовать все доступные ему оптимизации для улучшения времени исполнения,

осуществлять распараллеливание вычислений, применять динамическую компиляцию, реализованную в SQL интерпретаторе.

Основным языком для написания хранимых процедур, пользовательских функций и обработчиков триггеров в PostgreSQL является PL/pgSQL [5]. PL/pgSQL – это процедурное расширение языка SQL в рамках структурной парадигмы программирования. Программа на языке PL/pgSQL состоит из операторов языка SQL, операторов ветвлений и циклов, может использовать переменные для передачи значений между операторами SQL, вызывать хранимые процедуры и функции.

Выполнение пользовательской функции, написанной на языке PL/pgSQL, в PostgreSQL осуществляется путём интерпретации абстрактного синтаксического дерева, соответствующего коду функции. В случае, когда при выполнении SQL-запроса пользовательская функция вызывается много раз, обход синтаксического дерева является повторным вычислением. Предлагаемый подход состоит в том, что все пользовательские функции языка PL/pgSQL, которые упоминаются в поступившем в PostgreSQL запросе, компилируются в машинный код путём однократного обхода их синтаксических деревьев. При исполнении поступившего запроса, вызовы интерпретатора функций PL/pgSQL заменяются на вызовы скомпилированного кода соответствующих функций. Таким образом, предлагаемый подход позволяет избавиться от повторных вычислений, связанных с многократным обходом абстрактных синтаксических деревьев пользовательских функций PL/pgSQL, и, предположительно, сократить время выполнения запросов, содержащих вызовы таких пользовательских функций.

Работа выполняется с использованием компиляторной инфраструктуры LLVM [6] в динамическом компиляторе [7-10] запросов PostgreSQL, разрабатываемом в ИСП РАН [11].

2. PL/pgSQL – процедурное расширение SQL

PL/pgSQL – процедурное расширение языка SQL для СУБД PostgreSQL. PL/pgSQL позволяет пользователям достаточно просто и безопасно определять функции и триггеры в PostgreSQL. С помощью него можно выполнять не только SQL-запросы, но и сложные вычисления. Функции PL/pgSQL могут использоваться везде, где допустимо использование встроенных функций PostgreSQL. PL/pgSQL в отличии от традиционного SQL, используемого в PostgreSQL и большинства других СУБД в качестве языка запросов, позволяет группировать блоки вычислений и последовательности запросов внутри сервера базы данных, что значительно увеличивает скорость обработки данных, решая такие проблемы как: излишнее взаимодействие клиента и сервера, передача промежуточных данных между исполнителями, увеличивающие время выполнения запросов, а также многократное выполнение одних и тех же запросов.

Все операторы PostgreSQL группируются в блоки операторов. Тело функции, написанной на PL/pgSQL так же является блоком операторов. Каждый блок операторов помечен некоторым идентификатором – меткой, содержит последовательность операторов, которые выполняются последовательно при выполнении блока, и может содержать объявления переменных, доступных для использования операторами блока. Блок операторов является оператором, то есть допустимо вложение блоков. Структура блока имеет следующий вид:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
END [ метка ];
```


Каждое объявление и каждый оператор в блоке должны завершаться символом ";"(точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, однако финальный END, завершающий тело функции, не требует точки с запятой.

Для определения функций, написанных на языки PL/pgSQL, используется команда CREATE FUNCTION. Например, функция gt, которая принимает на вход два аргумента типа integer и возвращает истину, если значение первого аргумента больше, чем значение второго, и ложь в противном случае, на языки PL/pgSQL выглядит так:

```
CREATE FUNCTION gt (x
integer, y integer)
RETURNS BOOLEAN AS $$
BEGIN
   IF x > y THEN
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL – типизированный язык, можно создавать переменные скалярных типов данных (тех же, что используются при написании выражений в SQL-запросах), типов кортежей с фиксированными типами элементов, типов кортежей с неизвестными во время компиляции типами элементов, типов массивов, типов структур.

Основными операторами PL/pgSQL являются:

- оператор присваивания переменной значения некоторого выражения;
- оператор выполнения фиксированного SQL-запроса;
- оператор выполнения SQL-запроса, формируемого в процессе выполнения функции;
- оператор ветвления;
- оператор цикла с логическим условием;
- оператор цикла по элементам массива;
- оператор цикла по результатам запроса;
- оператор возврата значения из функции.

2.1 Устройство интерпретатора PL/pgSQL

При вызове пользовательской функции, написанной на PL/pgSQL, осуществляется синтаксический и семантический анализ исходного кода функции. Результатом анализа является абстрактное синтаксическое дерево кода функции.

Каждая вершина абстрактного синтаксического дерева кода функции PL/pgSQL соответствует некоторому оператору в коде пользовательской функции. В зависимости от типа оператора, вершина хранит информацию, необходимую для выполнения соответствующего оператора – сам тип оператора, подвыражения, рёбра к связанным операторам.

Для выполнения пользовательской функции интерпретатор PL/pgSQL обходит абстрактное синтаксическое дерево функции. Каждому типу оператора PL/pgSQL соответствует

некоторая функция языка C в интерпретаторе, которая принимает на вход вершину абстрактного синтаксического дерева и выполняет вычисление, которое описано в вершине.

Пример абстрактного синтаксического дерева, соответствующего коду пользовательской функции на языке PL/pgSQL, приведен на рис 1. На этом рисунке также приведен граф вызовов функций интерпретатора, выполняющих вычисление пользовательской функции, представленной абстрактным синтаксическим деревом.

Так как функции интерпретатора, вычисляющие операторы PL/pgSQL, написаны в общем виде, без учёта специфики конкретного оператора PL/pgSQL (специфицирован только тип оператора, но не значения операндов, их типы и связанные операторы), при многократном вызове любого оператора осуществляются вычисления, результат которых определяется не данными, получаемыми во время вычисления оператора, а данными, известными после синтаксического и семантического анализа программы. Предлагаемое решение предполагает, что при кодогенерации известны вызываемые функции и операторы программы, известны типы переменных, выражений, и сгенерированный код операторов не будет содержать повторных вычислений, результат которых определяется информацией, известной во время кодогенерации.

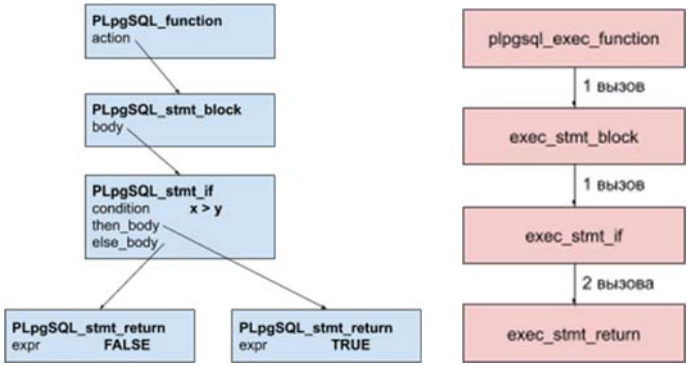


Рис. 1. Абстрактное синтаксическое дерево (слева) пользовательской функции gt, тело которой представлено в главе 2, и граф вызовов функций интерпретатора PL/pgSQL, осуществляющих вычисление функции gt (справа)
Fig. 1. Abstract syntax tree (left) of user defined function gt, which body is represented in chapter 2, and call graph of PL/pgSQL interpreter that executes gt (right)

3. Динамический компилятор запросов PostgreSQL, разрабатываемый в ИСП РАН

В ИСП РАН разрабатывается расширение [8] к СУБД PostgreSQL, реализующее динамическую компиляцию запросов с использованием компиляторной инфраструктуры LLVM.

Для исполнения запросов в PostgreSQL используется модель итераторов (Volcano-модель) [11]. Использование данной модели сопряжено с существенными накладными расходами, связанными с неявными вызовами функций и сопутствующими ошибками предсказания переходов, невозможноси выполнения оптимизаций и необходимостью сохранения состояния операторов между вызовами.

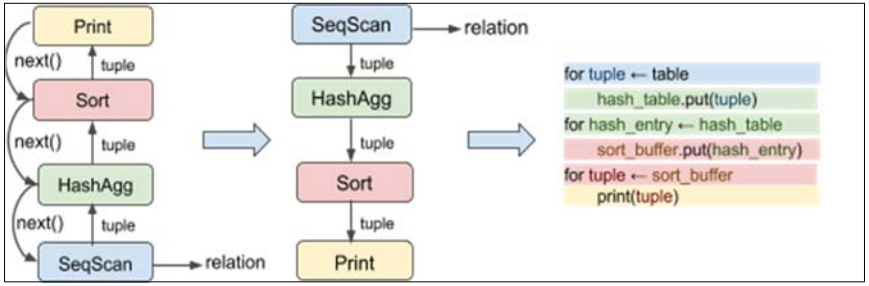


Рис. 2. Переход от модели итераторов (слева) к push-модели (посередине и справа) выполнения запроса
Fig. 2. Query plan transformation from iterator model into push-based model

В разработанном динамическом компиляторе запросов реализован переход от модели итераторов к модели явных циклов – push-модели (рис. 2) и алгоритм генерации кода в этой модели, основанный на сопоставленных каждому оператору функциях `consume` и `finalize`. Во время обход плана запроса выполняется генерации кода с использованием LLVM C API, при котором каждая вершина-оператор получает на вход функции `consume` и `finalize` от родительского оператора и генерирует код, в котором функция `consume` вызывается для каждого очередного возвращаемого родительскому оператору кортежа, а функция `finalize` – после возврата последнего кортежа. Генерации кода для внутренних операторов состоит в генерации функций `consume` и `finalize` (содержащих вызовы родительских функций `consume` и `finalize`) по одной на дочерний оператор и вызова генераторов дочерних операторов. В листовых операторах происходит генерация основных циклов обхода таблицы или индекса и вызов переданных функций `consume` и `finalize`.

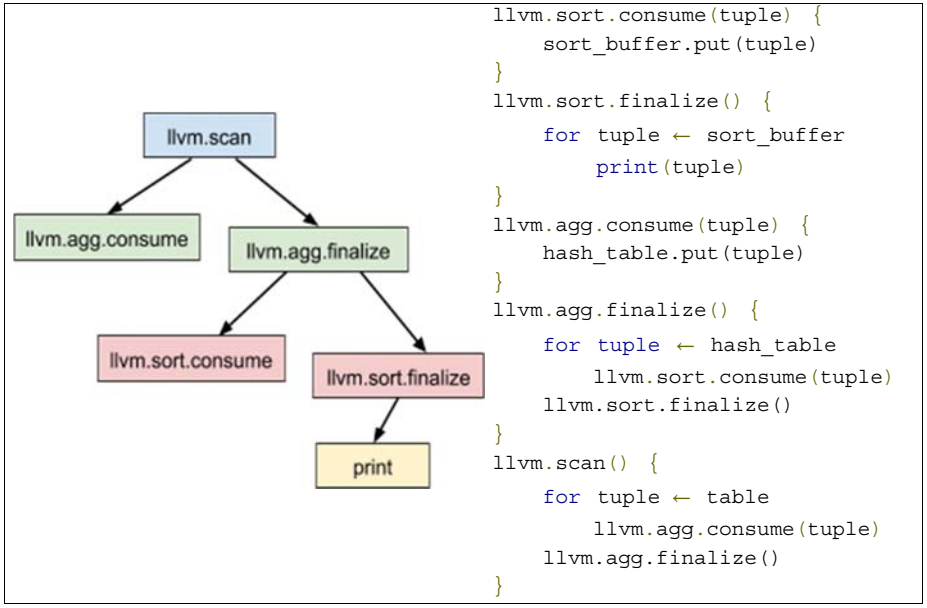


Рис. 3. Сгенерированный код в push-модели
Fig. 3. Generated code in push-based model

На рис. 3 показан результат генерации кода в push-модели. После встраивания получается код как на рис. 3 справа.

Также, в динамическом компиляторе запросов был разработан метод кэширования [10] и переиспользования сгенерированного динамическим компилятором машинного кода, позволяющий избавиться от накладных расходов на его создание (оптимизацию и компиляцию) при повторном использовании.

На бенчмарке TPC-H динамический компилятор запросов позволяет получить ускорение до 5.5 раз.

4. *Динамическая компиляция пользовательских функций на языке PL/pgSQL*

Предлагаемый в настоящей статье подход был реализован в виде дополнения к динамическому компилятору запросов к СУБД PostgreSQL, разрабатываемому в ИСП РАН (см. разд. 3). Предварительно был произведен анализ накладных расходов при интерпретации пользовательских функций на языке PL/pgSQL с целью идентификации наиболее горячих участков кода, которые можно оптимизировать с использованием рассматриваемого метода.

4.1 Анализ накладных расходов, связанных с вызовом PL/pgSQL функций

Для измерения расходов на интерпретацию пользовательских функций PL/pgSQL было проведено профилирование выполнения SQL-запросов, содержащих функции на языке PL/pgSQL, с помощью промышленного профилировщика perf [13], по результатам которого выяснилось, что на некоторых запросах доля времени вычисления пользовательских функций достигает более 75%. На рис. 4 представлен результат профилирования простого запроса "*select count(*) from tbl where gt(a,b)=true"*, где gt – это пользовательская функция на языке PL/pgSQL, представленная в разд. 2.

```
standard_ExecutorRun
 ExecProcNode
- ExecAgg
  - 97,65% fetch_input_tuple
    - ExecProcNode
      - ExecScan
        - 94,58% ExecQual
          - ExecMakeFunctionResultNoSets
            - 89,98% plpgsql_call_handler
              + 77,73% plpgsql_exec_function
              + 5,10% SPI_connect
              + 4,08% SPI_finish
              + 3,06% plpgsql_compile
            + 3,59% slot_getattr
+ 2,04% SeqNext
```

Рис. 4. Результат профилирования запроса "select count(*) from tbl where gt(a,b)=true;"
Fig. 4. Query "select count(*) from tbl where gt(a,b)=true;" profiling result

СУБД PostgreSQL выделяет память [14] в так называемых контекстах памяти (memory context), и тем самым реализует удобный способ управления выделением памяти в различных местах, с разными сроками жизни. При уничтожении контекста памяти освобождается вся выделенная в рамках данного контекст память.

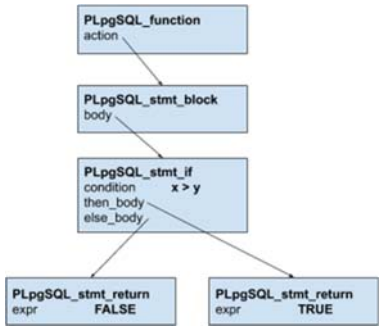
В интерпретаторе PL/pgSQL перед выполнением пользовательской функции вызывается функция SPI_connect, которая создает новый контекст памяти и делает его текущим, а после

выполнения вызывается функция SPI_finish для уничтожения контекста, созданного функцией SPI_connect, и восстановления предыдущего контекста памяти. Как видно из рис. 4, переключение контекста накладывает дополнительные расходы на время выполнения пользовательских функций, в дополнение к расходам, связанным с интерпретацией кода пользовательской функции (`plpgsql_exec_function`). Накладные расходы на переключение контекста могут быть уменьшены за счёт сокращения количества динамических аллокаций путем анализа графа вызова UDF функций, однако в данной работе эта оптимизация не рассматривается.

4.2 Реализация динамической компиляции пользовательских функций в компиляторе запросов СУБД PostgreSQL

Компиляция пользовательских функций производится при компиляции запроса, поступившего в СУБД. В случае, когда поступивший в СУБД SQL-запрос содержит вызов пользовательской функции PL/pgSQL, компилятор пользовательских функций производит построение и анализ абстрактного синтаксического дерева кода вызываемой пользовательской функции. Осуществляется обход абстрактного синтаксического дерева с целью проверки возможности компиляции пользовательской функции динамическим компилятором. В случае прохождения проверки, происходит компиляция пользовательской функции PL/pgSQL, иначе вызов обрабатывается интерпретатором PL/pgSQL, встроенным в PostgreSQL.

Компилятор пользовательских функций генерирует код на языке LLVM IR с использованием LLVM C API. Генерация кода пользовательской функции PL/pgSQL осуществляется путём обхода её абстрактного синтаксического дерева. Каждый узел этого дерева является представлением некоторого оператора PL/pgSQL, и для каждого узла при обходе дерева вызывается функция кодогенерации, соответствующая типу оператора, представленного данным узлом. Функция кодогенерации оператора каждого типа написана на основе функции интерпретации оператора соответствующего типа, содержащейся в интерпретаторе PL/pgSQL.




```
define  i64 @"llvm_gt"(%struct.FunctionCallInfoData*)
{entry:
  %args  = getelementptr
%struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %0, i32  0, i32  6
  %arg_ptr  = getelementptr  [100  x  i64],
[100  x  i64]* %args, i32  0, i32  0
  %arg  = load  i64, i64* %arg_ptr
          %arg_ptr2  = getelementptr  [100  x  i64],
          [100  x  i64]*%args, i32  0, i32  1
  %arg2  = load  i64, i64* %arg_ptr2
  %if_cond  = icmp gt  %arg, %arg2
  br  i1  %if_cond, label %then_body, label %else_body
then_body:
  ret  i64  1
else_body:
  ret  i64  0
}
```

Рис. 5. Слева – абстрактное синтаксическое дерево функции gt, справа – сгенерированный для этой функции LLVM IR
Fig. 5. On the left: abstract syntax tree of the gt function, on the right: generated LLVM IR for it.

При этом, если функция интерпретации оператора содержит вычисление, результат которого не зависит от значений времени выполнения пользовательской функции, функция кодогенерации оператора выполняет это вычисление во время кодогенерации, и скомпилированный код строится с учётом уже известного результата такого вычисления. Для тех вычислений в функции интерпретации оператора, результат которых нельзя определить во время кодогенерации, функция кодогенерации оператора генерирует код, осуществляющий данное вычисление во время выполнения пользовательской функции. В результате обхода абстрактного синтаксического дерева пользовательской функции генерируется функция на языке LLVM IR, которая динамически компилируется один раз с помощью LLVM и вызывается при выполнении SQL-запроса для вычисления пользовательской функции.

На рис. 5 приведён пример сгенерированного динамическим компилятором пользовательских функций LLVM IR для абстрактного синтаксического дерева функции gt, приведённой в разд. 2.

При динамической компиляции кода запроса и кода пользовательской функции на языке LLVM IR, компилятор LLVM осуществляет множество оптимизаций, одна из которых – встраивание кода вызываемой функции вместо её вызова. Такая оптимизация позволяет улучшить производительность скомпилированного кода за счёт избавления от вызова и проведения внутрипроцедурных оптимизаций кода. Пример того, как код пользовательской функции, сгенерированный динамическим компилятором PL/pgSQL, встраивается в код выполнения запроса, сгенерированный динамическим компилятором запросов к СУБД PostgreSQL, приведён на рис. 6.

1

```
llvm.ExecQual(tuple) {
   return
(*plpgsql_exec_function)(tuple)
}

llvm.scan() {
    for tuple ← table
        if llvm.ExecQual(tuple):
            print(tuple)
}
```

2

```
llvm_gt(tuple) {
    return tuple.a > tuple.b
}

llvm.ExecQual(tuple) {
    return llvm_gt(tuple)
}

llvm.scan() {
    for tuple ← table
        if llvm.ExecQual(tuple):
            print(tuple)
}
```

3

```
llvm.scan() {
    for tuple ← table
       if tuple.a > tuple.b:
            print(tuple)
}
```

Рис. 6. Слева – код запроса, сгенерированный динамическим компилятором запросов, с вызовом интерпретатора PL/pgSQL; справа – код, сгенерированный динамическим компилятором запросов и динамическим компилятором пользовательских функций: сверху – до оптимизации, снизу – после оптимизации встраивания функций

Fig. 6. On the left – code generated by JIT compiler with PL/pgSQL interpreter call; on the right – code generated by JIT compiler with PL/pgSQL compilation feature turned on: at the top before optimizations, at the bottom after inlining optimization.

Одной из ключевых задач при реализации компилятора пользовательских функций являлась реализация компиляции выражений, содержащихся в коде на языке PL/pgSQL. Сложность данной задачи заключалась в том, что типов операций, которые могут содержаться в выражении на языке PL/pgSQL, очень много. Для упрощения реализации и чтобы избежать дублирования кода, компиляция выражений была реализована следующим образом: если выражение на языке PL/pgSQL поддерживается в динамическом компиляторе [6], то выполняется соответствующая кодогенерация, иначе для его вычисления генерируется вызов интерпретатора PostgreSQL.

5. Результаты

Реализованный в рамках данной работы динамический компилятор поддерживает компиляцию пользовательских функций, написанных на подмножестве языка PL/pgSQL. Пользовательские функции, входящие в поддерживаемое подмножество языка, могут использовать переменные всех скалярных типов данных, поддерживаемых СУБД PostgreSQL, содержать операторы присваивания, ветвления, возврата значения из функции, простые циклы, выполнения произвольных фиксированных SQL-запросов.

Для тестирования производительности динамического компилятора были выбраны пользовательские функции, указанные в табл. 1. Было выполнено измерение времени выполнения запросов, содержащих вызов тестовых пользовательских функций, с использованием интерпретатора функций на языке PL/pgSQL и с использованием разработанного динамического компилятора функций на языке PL/pgSQL. Замер времени выполнения осуществлялся на компьютере с 16-ядерным процессором Intel Xeon CPU E5520 с ограничением тактовой частоты 2.27 ГГц под управлением операционной системы openSUSE Leap 15.1.

Сравнение производительности интерпретатора и компилятора выполнялось с использованием СУБД PostgreSQL версии 9.6.3. Размер таблицы table, из которой осуществлялась выборка данных, составляет 5 Гб; таблица содержит пять столбцов и 40000000 кортежей. Столбцы a и b, использующиеся в тестовых запросах, имеют тип integer.

Запросы, на которых производились замеры времени выполнения и результаты замеров указаны в табл. 2. В столбце «PG» указано время выполнения запроса с использованием интерпретатора запросов PostgreSQL и интерпретатора функций PL/pgSQL. В столбце «SQL JIT» указано время выполнения запроса с использованием динамического компилятора SQL-запросов и интерпретатора пользовательских функций на языке PL/pgSQL. В столбце «PL/pgSQL JIT» указано время выполнения запроса с использованием динамического компилятора SQL-запросов и разработанного динамического компилятора пользовательских функций на языке PL/pgSQL.

Табл. 1. Пользовательские функции для тестирования производительности динамического компилятора
Table 1. User defined functions used for performance testing of dynamic compiler

Имя пользовательской функции	Код функции на языке PL/pgSQL

gt	`CREATE FUNCTION gt (x integer, y integer)`
	`RETURNS boolean AS $$`
	`BEGIN`
	`  IF x > y THEN`
	`  RETURN TRUE;`
	`  ELSE`
	`  RETURN FALSE;`
	`  END IF;`

	`END;` `$$ LANGUAGE plpgsql;`

inc	`CREATE FUNCTION inc (x integer)`
	`RETURNS integer AS $$`
	`BEGIN`
	`  RETURN x + 1;`
	`END;`
	`$$ LANGUAGE plpgsql;`

sumN	`CREATE FUNCTION sumN (n integer)`
	`RETURNS INTEGER AS $$`
	`DECLARE`
	`sum integer := 0;`
	`idx integer := 0;`
	`BEGIN`
	`  LOOP`
	`  IF idx > n THEN`
	`  RETURN sum;`
	`  END IF;`
	`  sum = sum + idx;`
	`  idx = idx + 1;`
	`  END LOOP;`
	`END;`
	`$$ LANGUAGE plpgsql;`

is_prime	`CREATE FUNCTION is_prime (n integer)`
	`RETURNS BOOLEAN AS $$`
	`DECLARE`
	`idx integer := 2;`
	`BEGIN`
	`  LOOP`
	`  IF idx > n/2 THEN`
	`  RETURN TRUE;`
	`  END IF;`
	`  IF mod(n, idx) = 0 THEN`
	`  RETURN FALSE;`
	`  END IF;`
	`  idx = idx + 1;`
	`  END LOOP;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Табл. 2. Сравнение времени выполнения запросов (в секундах) с использованием стандартного интерпретатора запросов PostgreSQL, динамического компилятора SQL-запросов и динамического компилятора SQL-запросов со включённым динамическим компилятором пользовательских функций на языке PL/pgSQL
Table 2. Comparison of execution times (in seconds) of PostgreSQL interpreter, SQL JIT compiler and SQL JIT compiler with enabled PL/pgSQL UDF compilation feature

№	Тестовый запрос	SQL JIT	PL/pgSQL JIT	PG	PG: SQL JIT	PG: PL/pgSQL JIT
1	`select is_prime(100000007);`	57.04	2.09	57.10	1.00	**27.32**
2	`select count(*) from table where gt(a,b)=true;`	77.99	16.01	95.22	1.22	**5.95**
3	`select sum(inc(a)) from table where gt(a,b)=true;`	137.99	25.54	167.72	1.22	**6.57**
4	`select sumN(a) from table where gt(a,b)=false;`	50.35	15.40	83.79	1.66	**5.44**

Можно отметить, что используемые для тестирования запросы являются синтетическими и в полной мере не могут отражать реальные задачи, но в то же время хорошо демонстрируют высокую производительность сгенерированного с помощью LLVM машинного кода, который сравним по эффективности с кодом на языке C. Например, в тестовом запросе №1 структура выбрана таким образом, чтобы минимизировать влияние используемого метода выполнения плана запроса на общее время выполнение запроса. Получается, что основной вклад в общее время выполнения данного запроса вносит время выполнения пользовательской функции. Исходя из структуры данного запроса и результатов замеров времени его выполнения, можно сделать вывод, что независимо от метода выполнения плана запроса, выполнение пользовательских функций PL/pgSQL методом динамической компиляции занимает на порядок меньше времени по сравнению с выполнением методом интерпретации.

Полученные результаты показывают, что динамическая компиляция пользовательских функций на языке PL/pgSQL в совокупности с динамической компиляцией запросов позволяет ускорить выполнение SQL-запросов в несколько раз, а на некоторых функциях, выполняющих целочисленные вычисления, на порядок.

6. Заключение

В данной работе демонстрируются промежуточные результаты исследования и реализации метода динамической компиляция процедурного расширения PL/pgSQL в СУБД PostgreSQL. Метод позволяет значительно увеличить производительность СУБД на SQL-запросах, использующих PL/pgSQL функции.

Динамическая компиляция подмножества языка PL/pgSQL реализована в динамическом компиляторе запросов СУБД PostgreSQL с использованием компиляторной инфраструктуры LLVM. Результаты тестирования динамической компиляция запросов с процедурными

расширениями на синтетических примерах демонстрируют ускорение времени выполнения запросов в несколько раз.

В будущем планируется обеспечить более широкую поддержку языка PL/pgSQL в динамическом компиляторе запросов – реализовать компиляцию функций, содержащих сложные операторы языка, рекурсивные вызовы и использующих переменные типов массивов и структур. Для увеличения производительности разработанного решения возможна оптимизации использования контекстов памяти, уменьшающая накладные расходы на переключение контекстов, а также применение кэширования динамически сгенерированного кода пользовательских функций. Особый интерес вызывает сравнительное тестирование производительности разработанного решения с производительностью других процедурных расширений СУБД PostgreSQL: PL/V8 [15], PL/Python [16], PL/Perl [17].

Список литературы / References

[1]. PostgreSQL official site. Available at: https://www.postgresql.org/, accessed: 20.10.2020.

[2]. Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, Conor Cunningham Froid. Optimization of Imperative Programs in a Relational Database. Proceedings of the VLDB Endowment, vol. 11, no. 4, 2017, pp. 432-444.

[3]. Christian Duta, Denis Hirn, and Torsten Grust. Compiling PL/SQL Away. In Proc. of the 10th Annual Conference on Innovative Data Systems Research (CIDR'20), 2020, 8 p.

[4]. Denis Hirn and Torsten Grust. PL/SQL Without the PL. In Proc. of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), 2020, pp. 2677–2680.

[5]. PL/pgSQL – SQL Procedural Language. Available at: https://www.postgresql.org/docs/9.6/plpgsql-overview.html, accessed: 20.10.2020

[6]. The LLVM Compiler Infrastructure. Available at: http://llvm.org/, accessed: 20.10.2020.

[7]. 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 / 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).

[8]. 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 / 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).

[9]. E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher. Runtime specialization of postgresql query executor. Lecture Notes in Computer Science, vol. 11, 2018, pp. 375–386.

[10]. 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, стр. 205-220. DOI: 10.15514/ISPRAS-2020-32(1)-11 / 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. 205-202 (in Russian).

[11]. ISP RAS website. Available at: https://www.ispras.ru/en/, accessed: 20.10.2020.

[12]. Graefe G. Volcano – an extensible and parallel query evaluation system. IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, 1994, pp. 120-135.

[13]. Perf profiler website. Available at: https://perf.wiki.kernel.org/index.php/Main_Page, accessed: 20.10.2020.

[14]. PostgreSQL SPI Memory Management. Available at: https://www.postgresql.org/docs/9.6/spi-memory.html, accessed: 20.10.2020.

[15]. PLV8 – A Procedural Language in Javascript powered by V8. Available at: https://github.com/plv8/plv8, accessed: 20.10.2020.

[16]. PL/Python – Python Procedural Language. Available at: https://www.postgresql.org/docs/9.6/plpython.html, accessed: 20.10.2020.

[17]. PL/Perl – Perl Procedural Language, Available at: https://www.postgresql.org/docs/9.6/plperl.html, accessed: 20.10.2020.

Информация об авторах / Information about authors

Владислав Муратович ДЖИДЖОЕВ – лаборант отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Vladislav Muratovich DZHIDZHOYEV – laboratory assistant at Compiler Technologies Department. Research interests: compiler technologies, optimization.

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

Ruben Arturovich BUCHATSKIY – Researcher at Compiler Technology Department. Research interests: compiler technologies, optimizations.

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.

Александр Николаевич ТОМИЛИН – доктор физико-математических наук, профессор, главный научный сотрудник ИСП РАН, профессор МГУ. Научные интересы: операционные системы, архитектура и структура вычислительных систем, компиляторные технологии.

Alexander Nikolaevich TOMILIN – Doctor of Physical and Mathematical Sciences, Professor, Chief Researcher at ISP RAS, Professor at MSU. Research interests: Operating Systems, Architecture and structure of computing systems, compiler technologies.