DOI: 10.15514/ISPRAS-2022-34(4)-8

Настройка критериев планировщика СУБД с учётом динамической компиляции

[1,2] *Е.В. Долгодворов, ORCID: 0000-0001-6962-6448 <krym4s@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] *Д.М. Мельник,* ORCID: *0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский физико-технический институт*
141701, Московская область, г. Долгопрудный, Институтский переулок, д.9

Аннотация. С разработкой динамического компилятора запросов актуальной становится задача настройки критериев оптимизатора и планировщика СУБД для выбора оптимального, с точки зрения динамической компиляции, плана запроса. Необходимость настройки критериев оптимизатора обуславливается тем, что свойства различных моделей выполнения накладывают ограничения на эффективность выполнения плана запроса с использованием определённых узлов-операторов. Так, например, используемая в динамическом компиляторе push-модель даёт преимущество при выполнении запросов с использованием последовательного сканирования. При использовании динамической компиляции сканирование индекса может выполняться значительно менее эффективно, чем последовательное сканирование. Использование в плане вершин сканирования индекса уменьшает ценность метода динамической компиляции. Для преодоления указанных проблем предлагается выполнить настройку оптимизатора СУБД таким образом, чтобы тот оценивал и учитывал эффективность использования некоторых типов узлов при построении плана запроса с его последующей динамической компиляцией. В данной работе рассматривается модификация планировщика PostgreSQL для выбора наиболее эффективного пути выполнения запроса с учётом аппаратных характеристик и различий между интерпретируемой и компилируемой моделями выполнения узлов-операторов.

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

Для цитирования: Долгодворов Е.В., Бучацкий Р.А., Пантилимонов М.В., Мельник Д.М. Настройка критериев планировщика СУБД с учётом динамической компиляции. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 101-116. DOI: 10.15514/ISPRAS-2022-34(4)-8

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

JIT-aware DBMS planner configuration

[1,2] *E.V. Dolgodvorov, ORCID: 0000-0001-6962-6448 <krym4s@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] *D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russian Federation
[2] *Moscow Institute of Physics and Technology*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russian Federation

Abstract. Dynamic compilation of certain operator compositions might have a drastic impact on overall query performance, but not be considered during optimal query plan selection by DBMS planner due to lack of knowledge. To tackle this problem, we propose to extend cost model with criteria that make dynamic compilation overhead relevant. The necessity to set up the optimizer criteria is due to the fact that the properties of various execution models impose restrictions on the efficiency of query plan execution using certain operator nodes. For example, the push-model used in the dynamic compiler is advantageous when executing queries using sequential scan. So, dynamic compilation makes sequential scan more efficient than index scan. Using index nodes in such a plan makes the value of the dynamic compilation method diminishing. To overcome these problems, it is proposed to configure the DBMS optimizer, so that it evaluates and takes into account the efficiency of using certain types of nodes when building a query plan with its subsequent dynamic compilation. This paper discusses the modification of the PostgreSQL planner to select the most efficient query execution plan based on hardware characteristics and the execution model of operator nodes with interpretation or compilation.

Keywords: dynamic compilation; JIT-compilation; RDBMS; PostgreSQL; LLVM; query languages; planner.

For citation: Dolgodvorov E. V., Buchatskiy R. A., Pantilimonov M. V., Melnik D.M. JIT-aware DBMS planner configuration. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 101-116 (in Russian). DOI: 10.15514/ISPRAS-2022-34(4)-8

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

1. Введение

В ходе тестирования производительности разработанного в ИСП РАН [1-3] динамического компилятора запросов для СУБД PostgreSQL [6] на тестовом наборе TPC-H [9] было обнаружено, что для некоторых запросов выбор последовательного сканирования вместо сканирования индекса дает значительное ускорение при выполнении запроса с динамической компиляцией. Качественные различия в скорости выполнения рассмотрены на примере запроса Q06 (см. листинг 1). При тестировании использовалась база данных объемом 30 Гб, сервер с архитектурой ARM. Планы выполнения запроса Q06 с последовательным и индексным сканированием представлены в табл. 1.

```
SELECT
    SUM(l_extendedprice * l_discount) AS revenue
FROM
    lineitem
WHERE
    l_shipdate >= DATE '1996-01-01'
    AND l_shipdate < DATE '1996-01-01' + INTERVAL '1' YEAR
    AND l_discount BETWEEN 0.09 - 0.01 AND 0.09 + 0.01
    AND l_quantity < 24;
```

Листинг 1. Текст SQL-запроса Q06 из тестового набора TPC-H
Listing 1. Q06 SQL-query from the TPC-H test suite

Табл. 1. Планы выполнения запроса Q06 с использованием сканирования индекса (наверху) и последовательного сканирования (внизу)
Table 1. Q06 query execution plans using an index scan (top) and a sequential scan (bottom)

```
Aggregate
   -> Bitmap Heap Scan ON lineitem
     Recheck Cond: ((l_shipdate >= '1996-01-01'::DATE)
       AND (l_shipdate < '1997-01-01'::TIMESTAMP WITHOUT TIME zone))
     FILTER: ((l_discount >= '0.08'::DOUBLE PRECISION)
       AND (l_discount <= '0.1'::DOUBLE PRECISION))
   -> Bitmap INDEX Scan ON i_l_shipdate
     INDEX Cond: ((l_shipdate >= '1996-01-01'::DATE)
       AND (l_shipdate < '1997-01-01'::TIMESTAMP WITHOUT TIME zone))
```

```
Aggregate
  -> Seq Scan ON lineitem
    FILTER: ((l_shipdate >= '1996-01-01'::DATE)
    AND (l_shipdate < '1997-01-01'::TIMESTAMP WITHOUT TIME zone)
    AND (l_discount >= '0.08'::DOUBLE PRECISION)
    AND (l_discount <= '0.1'::DOUBLE PRECISION)
    AND (l_quantity < '24'::DOUBLE PRECISION))
```

Время выполнения измерялось путём многократного выполнения запроса и подсчёта медианы полученных результатов. Результаты представлены в табл. 2.

Табл. 2. Время выполнения запроса Q06 с использованием сканирования индекса и последовательного сканирования
Table 2. Q06 query execution time using index scan and sequential scan

План запроса	Интерпретатор, сек.	Динамический компилятор, сек.
Bitmap Index Scan	21,48	18,77
Seq Scan	29,20	6,00

Стандартный планировщик выбирает сканирование индекса, не являющееся эффективным при выполнении с динамической компиляцией. Требуется разработать критерии выбора плана исполнения для PostgreSQL, учитывающие особенности динамической компиляции запросов. Такие критерии должны отдавать предпочтение планам, эффективным с точки зрения динамической компиляции, а не интерпретатора.

В данной работе предлагается настройка критериев планировщика, которая с большей точностью сопоставляет стоимость определённого плана со временем его выполнения при использовании динамического компилятора запросов.

2. Динамический компилятор запросов СУБД PostgreSQL

Разработанный в ИСП РАН динамический компилятор запросов [1–3] представляет собой расширение к СУБД PostgreSQL, генерирующий коды алгоритмов узлов-операторов на LLVM IR [4] с изменённой моделью выполнения. Эти изменения увеличивают эффективность выполнения узлов-операторов, что покрывает расходы на динамическую компиляцию.

2.1 Push-модель выполнения запросов

В динамическом компиляторе запросов мы переходим переход от используемой во многих современных СУБД *Volcano*-модели [8] исполнения запросов к модели явных циклов. План запроса в PostgreSQL представляет собой дерево, в котором листовые вершины – узлы сканирования, а их родительские вершины – узлы соединения и агрегации.

Volcano-модель, которую также называют *pull*-моделью, реализует каждый оператор с помощью итераторов, поддерживающих интерфейс *open()*, *next()*, *close()*. Первая и последняя

функции отвечают соответственно за инициализацию и освобождение ресурсов, нужных для работы узла-оператора. В этой модели вызов узлов выполняется от родителей к детям с помощью функции *next()*, после отработки узла оператор возвращает кортеж родительской вершине. Недостатки этой модели проявляются в том, что *next()* реализован через неявный вызов функции, что, как правило, вызывает ошибочное прогнозирование перехода (*branch misprediction*). Кроме того, *next()* для последовательного сканирования требует выгрузки переменных состояния, отвечающих за номер страницы и кортежа из памяти. Это может привести к большим накладным расходам. Схема вызова операторов в *pull*-модели показана на рис. 1.

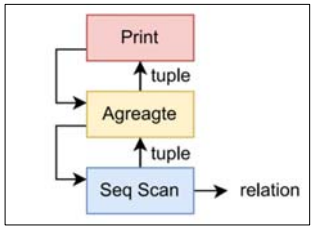

Рис. 1. Volcano-модель для запроса Q06
Fig. 1. Volcano model for Q06 query

Модель явных циклов, которую также называют *push*-моделью, предполагает прямой вызов операторов, а не рекурсивный. Выполнение всего запроса представляет собой проход по вложенным циклам, внешний из которых является циклом сканирования. В динамическом компиляторе запросов эта модель реализована с помощью функций *consume()* и *finalize()* для каждого оператора, где *consume()* вызывается для каждого произведенного кортежа и содержит часть алгоритма оператора, отвечающую за логику обработки получаемого кортежа, а *finalize()* выхывается для последнего кортежа и содержит части алгоритма оператора, отвечающего за постобработку данных. Итоговый код запроса представляется в виде последовательности вложенных циклов, в которых узлы-операторы вызываются явно. Кроме того, вызов узлов-операторов в циклах не требует сохранения и выгрузки контекста. Эти оптимизации позволяют получить ускорение до 5 раз на тестах из набора TPC-H по сравнению с выполнением запроса интерпретатором. Схема вызова узлов-операторов в *push*-модели показана на рис. 2.

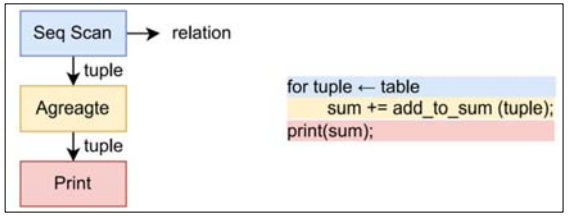

Рис. 2. Push-модель для запроса Q06
Fig. 2. Push-model for Q06 query

2.2 Динамическая компиляция выражения

В PostgreSQL обработка выражений представляет собой интерпретацию дерева выражений. Каждое такое дерево содержит в себе операторы и функции в качестве вершин и константы или переменные в качестве листовых вершин. Интерпретатор PostgreSQL осуществляет вызов функций и операторов неявно, что не даёт возможности для оптимизаций. Динамический компилятор запросов решает эту проблему, обходя это дерево рекурсивно в обратном порядке и генерируя явные вызовы функций в вышестоящих вершинах.

Таким образом, динамический компилятор запросов вносит значительные изменения в работу СУБД PostgreSQL на этапе выполнения плана, поэтому появляется необходимость выбора плана исполнения с учётом всех оптимизаций.

3. Планировщик СУБД PostgreSQL

SQL-запрос может быть выполнен разными способами, основываясь на плане запроса; при этом результат выполнения запроса будет одинаков вне зависимости от выбранного плана. Цель планировщика – построить наиболее эффективный план запроса. Планировщик PostgreSQL работает со структурами данных, называемыми путями. Каждый путь содержит информацию о сложности выполнения узла-оператора, сопоставляя ей стоимость. Собственная стоимость пути определяется суммарной стоимостью всех операций, которые выполняет узел.

При конструировании плана строится дерево путей, где стоимость узла задаётся суммой стоимости дочерних вершин и собственной стоимостью узла. Стоимость всего плана определяется стоимостью корневой вершины. Планировщик выбирает путь с наименьшей стоимостью. Стоимость операций в PostgreSQL задаётся константами, отношение которых даёт оценку отношения времени исполнения операций. Эта оценка не учитывает оптимизации при динамической компиляции, что ведёт к потере производительности. Так, последовательное чтение в случае динамической компиляции может оказаться более эффективным, нежели чтение по индексу, однако планировщик будет отдавать предпочтение второму.

3.1 Команда EXPLAIN ANALYZE

СУБД PostgreSQL предоставляет возможность профилирования плана запроса с помощью команды EXPLAIN ANALYZE [7]. Эта команда позволяет сравнить то, что предсказал планировщик, с реальными данными, полученными во время выполнения запроса. Планы запроса с EXPLAIN ANALYZE на примере запроса из листинга 1 представлены в табл. 3. Значение *cost* – это стоимость вершины, разбитая на стоимость получения первого кортежа и общую стоимость. Поле *actual time* хранит время выполнения вершины, оно разбито на два значения: время получения первого кортежа и общее время работы узла. EXPLAIN ANALYZE также выводит информацию о количестве кортежей, произведённых узлом и информацию, специфичную для каждого отдельного узла.

Табл. 3. Вывод команды EXPLAIN ANALYZE для запроса Q06 при использовании последовательного сканирования.
Table 3. Output of EXPLAIN ANALYZE command for Q06 query when using sequential scan.

```
Aggregate (cost=4455709.02..4455709.03 ROWS=1 width=8)
      (actual TIME=12463.297..12463.297 ROWS=1 loops=1)
  -> Seq Scan ON lineitem (cost=0.00..4437900.54 ROWS=3561696 width=16)
    (actual TIME=0.023..12233.865 ROWS=3573063 loops=1)
     FILTER: ((l_shipdate >= '1993-01-01'::DATE)
     AND (l_shipdate < '1994-01-01'::TIMESTAMP WITHOUT TIME zone)
     AND (l_discount >= '0.08'::DOUBLE PRECISION)
     AND (l_discount <= '0.1'::DOUBLE PRECISION)
     AND (l_quantity < '25'::DOUBLE PRECISION))
    ROWS Removed BY FILTER: 176425309
Planning TIME: 0.862 ms
Execution TIME: 12913.875 ms
```


4. Система для расчёта критериев планировщика с учётом динамической компиляции

Настройка планировщика PostgreSQL, учитывающая влияние динамической компиляции, проводится в два этапа: создание критериев и расчёт стоимости путей выполнения на основе этих критериев. Критерии, которые будут созданы на первом этапе, представляют собой набор констант, корректирующих формулы расчёта стоимости путей в планировщике СУБД с учётом динамической компиляции запросов. Константы вычисляются на основе информации о стоимости путей и времени их выполнения для тестовых запросов.

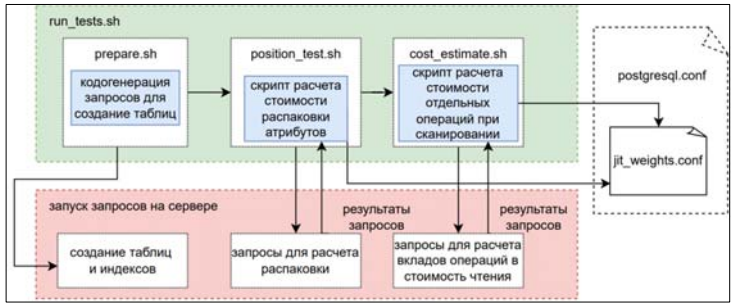

Рис. 3. Схема тестовой системы для расчёта критериев планировщика
Fig. 3. Scheme of testing system for criteria estimation of planner

Первый этап реализован в качестве тестовой системы, работа которой организована скриптом *run_tests.sh*. Изначально выполняется генерация и создание тестовых таблиц. Далее выполняется многократный запуск тестовых запросов с командой EXPLAIN ANALYZE с использованием интерпретатора СУБД PostgreSQL и динамического компилятора запросов. Результаты выполнения запросов используются для расчёта критериев для планировщика, учитывающие динамическую компиляцию запросов, после чего критерии заносятся в файл *jit_weights.conf*. Тестовые запросы сформированы на основе запросов из тестового набора TPC-H. Процесс расчёта критериев рассмотрен в подразделе 4.4. Для использования критериев необходимо записать содержимое *jit_weights.conf* в конец конфигурационного файла *postgresql.conf*. Схема работы тестовой системы представлена на рис. 3.

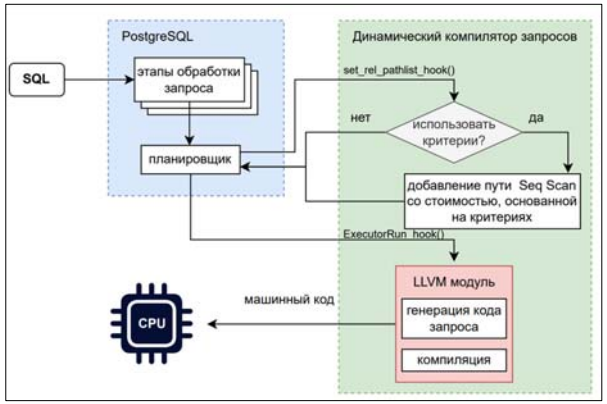

Рис. 4. Учет рассчитанных критериев планировщика при выполнении запроса динамическим компилятором
Fig. 4.Handling of estimated criteria for planner during execution with JIT-compilation

Второй этап выполняется непосредственно во время работы PostgreSQL. При запуске СУБД считывает конфигурационный файл *postgresql.conf* с настройками, инициализируя критерии для планировщика. Далее функция-перехватчик *set_rel_pathlist_hook()* на этапе планирования в зависимости от значения флага *use_jit_costs* добавляет для каждой таблицы, используемой в запросе, путь последовательного сканирования, стоимость которого была вычислена с учётом критериев. Итоговый план строится на основе путей, добавленных планировщиком PostgreSQL и путей, добавленных в функции-перехватчике. Далее происходит кодогенерация и динамическая компиляция запроса. Схема выполнения запроса с учётом рассчитанных критериев представлена на рис. 4.

4.1 Поддержка EXPLAIN ANALYZE в динамическом компиляторе запросов

В PostgreSQL для сбора статистики с помощью команды EXPLAIN ANALYZE [7] используются функции-счетчики: счётчики времени и количества кортежей, производимых узлом-оператором, счётчики количества кортежей, которые были удалены фильтром, и счётчики количества кортежей, видимость которых необходимо проверить в таблице, так как видимость не отображена в таблице индексов. Поддержка этих счётчиков динамическим компилятором запросов необходима для расчёта стоимости путей. Стоимость путей и время выполнения узлов будет использоваться при расчёте критериев для планировщика.

Счётчики времени представляют собой пару функций: *InstrStartNode()* и *InstrStopNode()*, они соответственно начинают и заканчивают отсчёт времени работы узла-оператора, а также подсчитывают число кортежей, им производимых. Эти счётчики должны поддерживаться каждой вершиной в плане выполнения запроса. Для их реализации в динамическом компиляторе запросов над каждым оператором была построена обёртка, как на листинге 2.

```
int consume_wrapper(node)
{
    InstrStartNode (node_instrument);
    ret = Call (node);
    InstrStopNode (node_instrument, 1);
    return ret;
}
```

Листинг 2. Реализация обёртки над оператором в динамическом компиляторе запросов
Listing 2. Node wrapper implementation in query JIT-compil.

В *push*-модели, используемой в динамическом компиляторе, каждому узлу-оператору соответствуют функции *consume* и *finalize*. В модели явных циклов после того, как оператор сформировал кортеж, он вызывает *consume* функцию родительского узла. Во время её исполнения счётчики не должны увеличивать время работы вершины и считать кортежи. Таким образом, вызов *consume* функции родительского узла оборачивается зеркальным кодом, что показано на листинге 3.

```
int ExecNode()
{
    /*...code...*/
    InstrStopNode(node_instrument, 1);
    parent_consume();
    InstrStartNode(node_instrument);
    /*...code...*/
}
```

Листинг 3. Реализация обертки над родительской consume() функцией в динамическом компиляторе запросов
Listing 3. Parent's consume() function wrapper implementation

Рассмотрим реализацию счётчиков на примере оператора Seq Scan, сгенерированный код которого во внутреннем представлении LLVM IR представлен на листинге 3. В

интерпретаторе СУБД PostgreSQL каждый узел оформлен в отдельную функцию, которая вызывается в обёртке *ExecProcNode()*, выставляющей счетчики *InstrStartNode()* до вызова узла и *InstrStopNode()* сразу по завершении его работы. Интерпретатор не содержит в себе обёрток наподобие той, что представлена в листинге 3, так как вышестоящие узлы не вызываются нижестоящими, лишь получая от них кортежи.

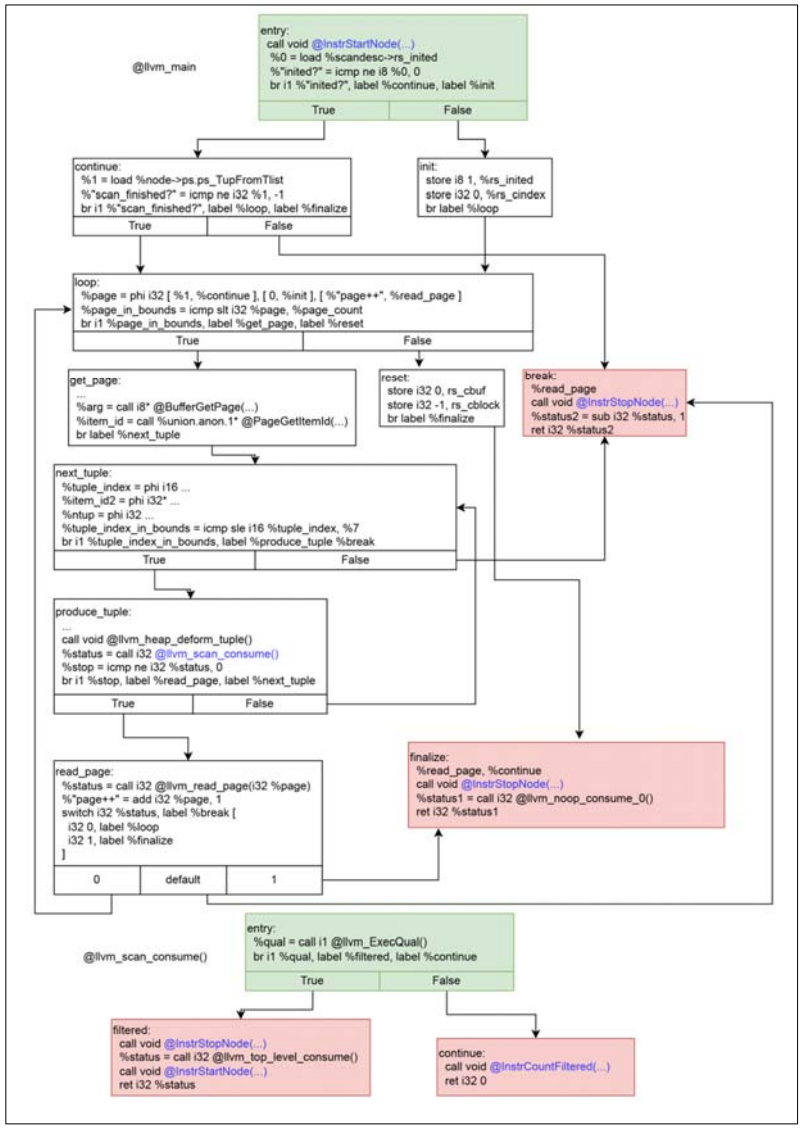

Рис. 5. Реализация оператора Seq Scan на LLVM IR
Fig. 5. LLVM Seq Scan implementation in JIT-compiler

На примере Seq Scan также рассмотрим добавление поддержки счётчика кортежей, удалённых фильтром. Поддержка этого счётчика осуществляется с помощью функции

InstrCountFiltered(). В EXPLAIN ANALYZE от него зависит поле *Rows removed by Filter* некоторых путей. Счётчик добавляется в обработку выражения. Реализация поддержки счётчиков в динамическом компиляторе запросов представлена на рис. 5.

Таким образом была реализована поддержка счётчиков в динамическом компиляторе запросов для основных узлов-операторов СУБД PostgreSQL.

4.2 Влияние распаковки атрибутов на скорость работы узла Seq Scan

Планировщик СУБД PostgreSQL не учитывает в стоимости вершин распаковку атрибутов. Доступ к элементу кортежа происходит в цикле, который суммирует смещения каждого предыдущего элемента относительно друг друга, чтобы получить итоговое смещение относительно начала кортежа. Для таблиц с большим числом столбцов время распаковки элементов оказывается значительным по отношению ко времени последовательного сканирования. Интерпретатор PostgreSQL совершает эти действия в функции *slot_deform_tuple()*. Подробно алгоритм распаковки атрибутов рассмотрен в статье [11]. Пример распаковки атрибутов в СУБД PostgreSQL представлен на рис. 6. В примере рассмотрена распаковка 3 и 8 атрибутов. Распаковка происходит в два этапа: с 1 по 3 атрибуты при первом вызове и с 4 по 8 атрибуты при втором вызове. В динамическом компиляторе запросов распаковка аргументов выполняется функцией *heap_deform_tuple()*, которая распаковывает все атрибуты вплоть до последнего необходимого за один вызов.

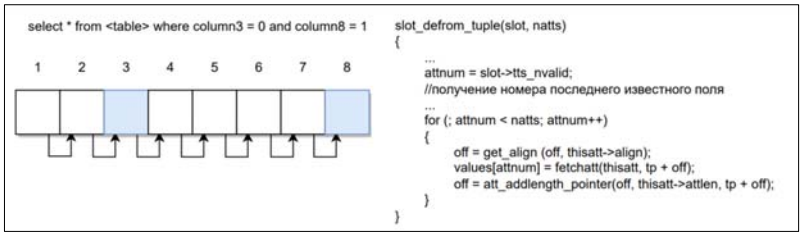

Рис. 6. Процесс распаковки атрибутов выражения в PostgreSQL
Fig. 6. Tuple deforming in PostgreSQL

При достаточно большом количестве столбцов может наблюдаться снижение эффективности последовательного сканирования в сравнении со сканированием индекса. Последнее происходит по специальной структуре, не требуя накладных расходов на распаковку.

Для демонстрации этого эффекта используется таблица с атрибутами типа *integer*. Каждая колонка этой таблицы содержит идентичные данные, чтобы обеспечить одинаковость времени выполнения при индексном сканировании. Пример построения таблицы и запросов, используемых для анализа, представлен в табл. 4. Скорость работы узлов Bitmap Index Scan и Index Scan зависит от *кардинальности запроса*, то есть количества кортежей, произведённых вершиной в ходе выполнения запроса. При увеличении кардинальности узел Bitmap Index Scan более эффективен, чем Index Scan, а Seq Scan более эффективен, чем Bitmap Index Scan. *cardinality_coef* – коэффициент, регулирующий кардинальность запроса.

Табл. 4. Запрос для анализа распаковки атрибута
Table 4. Query that is used for tuple deforming analysis

Таблица	Запрос
`CREATE TABLE deformingTest ( column_0 INT, column_1 INT, ... column_n INT );`	`EXPLAIN ANALYZE SELECT * FROM deformingTest WHERE column_[index] < [cardinality_coef];`


```
CREATE TABLE tmp
(
  index INT,
  rand DOUBLE PRECISION
);

INSERT INTO tmp (SELECT
code,
  1000 * random() FROM

generate_series(1,1000000)
  AS code);

INSERT INTO deformingTest(
  SELECT rand,
  ...
  rand FROM tmp);
```

Тестирование проводилось для 40 атрибутов. Для данной конфигурации таблицы и равномерно распределенных данных в столбцах 0, 19 и 39 время выполнения узлов Seq Scan и Bitmap Index Scan представлено в таблице 5. Время рассчитывалось как медиана времён при многократном запуске.

Табл. 5. Результаты тестирования распаковки
Table 5. Tuple deforming test results

Колонка №	Время Seq Scan, сек.	Время Bitmap Index Scan, сек.
0	15,78	15,92
19	17,75	16,03
39	20,15	16,19

Так как план не учитывает стоимость распаковки атрибутов, планировщик отдаёт предпочтение последовательному сканированию, которое не является эффективным во втором и третьем случаях.

В динамическом компиляторе запросов реализован учёт накладных расходов на распаковку атрибутов для увеличения точности оценки стоимости.

4.3 Влияние изменения модели выполнения на скорость работы узла Seq Scan

Динамический компилятор запросов, описанный в статьях [1-3], позволяет ускорить время выполнения запросов, содержащих узел Seq Scan, за счёт изменения модели выполнения с *Volcano* на модель явных циклов, что приводит к необходимости пересчёта стоимости этого пути, увеличивая его релевантность. Для этой цели формируются критерии.

В статье [12] подробно рассматривается оценка времени выполнения запроса на основе стоимости его выполнения, а также приводятся формулы, по которым происходит расчёт стоимости узлов-операторов. Так, стоимость вершины Seq Scan определяется планировщиком PostgreSQL следующей формулой:

$$total\_cost = startup\_cost + cpu\_run\_cost + disk\_run\_cost,$$

где:

$$cpu\_run\_cost = (cpu\_tuple\_cost + qpqual\_cost) \cdot rows.$$

Здесь:

$startup\_cost$ – стоимость получения первого кортежа, 0 для Seq Scan,
cpu_run_cost – стоимость обработки всех кортежей,
$disk\_run\_cost$ – стоимость чтения страниц памяти,

cpu_tuple_cost – стоимость обработки одного кортежа,
$qpqual\_cost$ – стоимость обработки выражения фильтра,
$rows$ – количество возвращённых кортежей.

Чтобы планировщик мог учитывать наличие динамической компиляции, введём корректирующий коэффициент ss_guc, обозначающий ускорение Seq Scan при выполнении запроса динамическим компилятором. В формуле его нужно учитывать именно перед ускорением обработки одного конкретного кортежа, так как это ускорение связано с использованием другой модели обработки запроса.

$$total\_cost = (\boldsymbol{ss\_guc} \cdot cpu\_tuple\_cost + qpqual\_cost) \cdot rows + disk\_run\_cost.$$

Необходимо учесть ускорение обработки выражения, для этого введём корректировочный коэффициент $qual\_guc$:

$$total\_cost = (ss\_guc \cdot cpu\_tuple\_cost + \boldsymbol{qual\_guc} \cdot qpqual\_cost) \cdot rows + disk\_run\_cost.$$

Ускорение чтения с диска никак не изменяется при динамической компиляции, поэтому нет необходимости вводить отдельные критерии. Как упоминалось в разд. 3, отношение констант, описывающих стоимость операций последовательного чтения, отличается для каждой конкретной машины, поэтому введём коэффициенты, корректирующие отношение этих констант. Сумма введённых коэффициентов равна 3, что соответствует 100%.

$$total\_cost = (\boldsymbol{cpu\_impact} \cdot ss\_guc \cdot cpu\_tuple\_cost + \boldsymbol{qual\_impact} \cdot qual\_guc \cdot qpqual\_cost) \cdot rows + \boldsymbol{disk\_impact} \cdot disk\_run\_cost.$$

В предыдущем разделе мы также показали, что необходимо учитывать стоимость распаковки атрибута. Таким образом, итоговая формула расчета стоимости узла Seq Scan представима как:

$$total\_cost = (cpu\_impact \cdot ss\_guc \cdot cpu\_tuple\_cost + qual\_impact \cdot qual\_guc \cdot (qpqual\_cost + \boldsymbol{deform\_cost})) \cdot rows + disk\_impact \cdot disk\_run\_cost \quad (1)$$

4.4 Критерии для планировщика PostgreSQL с динамическим компилятором запросов

Модификации стоимостной системы планировщика для исполнения запросов с учётом динамической компиляции, рассмотренные в предыдущих разделах, были реализованы с помощью учёта критериев на этапе планирования. Расчёт критериев производится на основе профиля исполнения запросов, сформированных на базе набора TPC-H. При расчёте стоимости вершин, критерии принимают значения, которые не изменяют стоимости вершины, по отношению к стоимости, рассчитанной оригинальным планировщиком, их значения представлены в листинге 4. Запросы выполняются повторно, чтобы обеспечить попадание данных в кеш. Результаты расчёта критериев заносятся в конфигурационный файл *jit_weights.conf*.

```
SET llvm_jit.jit_ss_cpu_tuple TO 1;
SET llvm_jit. ss_qual_cost TO 1;

SET llvm_jit.jit_qual_impact TO 1;
SET llvm_jit.jit_cpu_impact_nq TO 1;
SET llvm_jit.jit_cpu_impact_q TO 1;
SET llvm_jit.jit_disk_impact_nq TO 1;
SET llvm_jit.jit_disk_impact_q TO 1;
```

Листинг 4. Стартовые значения критериев при тестировании.
Listing 4. Starting values of criteria during testing.

Система рассматривает два типа запросов: содержащие выражения и не содержащие их. Во втором случае константа $qual\_impact$ считается равной единице. *Impact*-критерии вычисляются как корни системы уравнений на основе выражения 1:

$$\begin{cases} cpu\_impact\_nq \cdot ss\_guc \cdot cpu\_cost + disk\_impact\_nq \cdot disk\_cost = total\_cost \\ cpu\_impact\_nq + disk\_impact\_nq = 2 \end{cases}, \quad (2)$$

$$\begin{cases} cpu\_impact\_q \cdot ss\_guc \cdot cpu\_cost + qual\_impact \cdot qual\_guc \cdot qual\_cost + disk\_impact\_q \cdot disk\_cost = total\_cost \\ cpu\_impact\_q + disk\_impact\_q + qual\_impact = 3 \\ disk\_cpu\_prop = \dfrac{disk\_impact\_nq}{cpu\_impact\_nq} = \dfrac{disk\_impact\_q}{cpu\_impact\_q} \end{cases}, \quad (3)$$

где:

$cpu\_cost = cpu\_tuple\_cost \cdot rows$
$disk\_cost = disk\_run\_cost$
$qual\_cost = (qpqual\_cost + deform\_cost) \cdot rows$

В этих системах постфикс q обозначает критерии, рассчитанные для запросов с выражениями, nq – без выражений, $total\_cost$ – стоимость индексного сканирования, именно такую стоимость должен иметь узел Seq Scan, чтобы быть выбранным планировщиком. Третье уравнение системы 3 следует из того факта, что отношение скорости последовательного сканирования таблицы и скорости чтения данных с носителя остаётся постоянным на одной аппаратуре. Другие критерии и необходимые коэффициенты для системы уравнений рассчитываются на основе профиля выполнения тестовых запросов и результатов работы команды EXPLAIN ANALYZE. Примеры запросов, которые были использованы для расчета критериев, представлены в листинге 5.

```
EXPLAIN (analyze, format json) SELECT COUNT(*) FROM lineitem;
EXPLAIN (analyze, format json) SELECT COUNT(*)
    FROM lineitem WHERE l_shipdate >= DATE '1993-01-01';
```

Листинг 5. Примеры тестовых запросов для расчёта критериев
Listing 5. Test queries for calculating criteria

Критерий ss_guc вычисляется как отношение времени выполнения запроса без каких-либо выражений с использованием динамической компиляции и без неё:

$$ss\_guc = \frac{t_{\text{JIT}}}{t_{\text{INT}}}.$$

Критерий $qual\_guc$ вычисляется как отношение времени обработки выражения при динамической компиляции и без неё. Время обработки выражения вычисляется вычитанием времени выполнения запроса без выражений из времени выполнения запроса, содержащего выражения.

$$qual\_guc = \frac{t\_qual_{\text{JIT}}}{t\_qual_{\text{INT}}}.$$

Вычисление составляющих $cpu\_cost, disk\_cost$ и $qual\_cost$ стоимости узла Seq Scan производится на основе результата работы команды EXPLAIN ANALYZE. Стоимость каждой составляющей можно получить, выставив значение *impact*-переменной, являющиеся множителем перед составляющей, интересующей нас, в единицу и занулив все остальные *impact*-переменные.

Составляющая $qual\_cost$ уточняется стоимостью распаковки атрибутов. Для расчёта критерия $deform\_cost$ используется табл. 4. Стоимость распаковки определяется отношением времени распаковки одной колонки ко времени выполнения вершины помноженным на стоимость выполнения вершины:

$$t_e = \frac{t_n - t_1}{n-1}, deform\_cost = \frac{t_e \cdot total\_cost}{t_1},$$

где

t_e – время распаковки одного столбца,

t_1 – время выполнения запроса, обращающегося к 1 колонке,

t_n – время выполнения запроса, обращающегося к колонке номер n,

$total\_cost$ – стоимость выполнения запроса, обращающегося к 1 колонке.

Необходимая информация вычисляется на основе профилирования запросов из листинга 6.

```
EXPLAIN (analyze, format json)
SELECT COUNT(*) FROM deformingTest WHERE column_0 == 0;

EXPLAIN (analyze, format json)
SELECT COUNT(*) FROM deformingTest WHERE column_n == 0;
```

Листинг 6. Тестовые запросы для расчета критерия распаковки атрибутов
Listing 6. Test queries to estimate deforming of attributes

Таким образом, вычисляются все критерии, которые будут использованы планировщиком при расчёте плана для выполнения с динамической компиляцией.

4.5 Модификация динамического компилятора запросов

Для построения плана планировщик СУБД PostgreSQL составляет список отношений (таблиц), участвующих в запросе. Для каждого отношения планировщик строит список путей — возможных узлов-операторов для выполнения плана. Пересчёт стоимости для выполнения с учётом динамической компиляции происходит за счёт добавления нового пути Seq Scan, стоимость которого рассчитана с учётом критериев (см. подразделы 4.3 и 4.4), в каждое отношение до момента выбора наилучшего пути. Введённые критерии считываются из конфигурационного файла во время старта сервера СУБД, и изменяют соответствующие константы. Расчёт стоимости пути с учётом введённых критериев производится по формуле 1. Добавление пути проходит в функции-перехватчике *set_rel_pathlist_hook()* (рис. 4). Последовательное сканирование является базовым методом доступа к данным, соответственно нет необходимости выполнять какие-либо проверки, для того, чтобы добавить этот путь. Далее планировщик выбирает наиболее эффективный путь выполнения запроса, основываясь на стоимости.

5. AQO как альтернативные критерии для планировщика СУБД PostgreSQL

При работе с динамическим компилятором запросов было интересно узнать, как на оптимизацию плана может повлиять расширение AQO. Adaptive Query Optimization [5] — расширение для СУБД PostgreSQL, которое методами машинного обучения улучшает предсказание кардинальности. Эта информация увеличивает точность, с которой происходит расчёт стоимости вершин сканирования индекса. Для сбора статистики AQO использует информацию, полученную с помощью команды EXPLAIN ANALYZE. Добавление поддержки EXPLAIN ANALYZE в динамическом компиляторе запросов позволило использовать это расширение совместно с динамическим компилятором.

На первом этапе AQO разделяет запросы на группы, основываясь на их структуре. Запросы попадают в одну группу, если тексты запросов отличаются только константами.

AQO собирает информацию о реальной кардинальности запросов после их выполнения для каждого типа запросов при значении поля *aqo.mode 'learn'* или *'intelligent'*. Запуск запросов при значении поля *aqo.mode = 'intelligent'* или *'controlled'* вычисляет для данного типа запроса и констант предположительную кардинальность вершин, основываясь на методе *Gradient K Nearest Neighbours*, рассмотренном в статье [10]. Эта информация используется планировщиком, чтобы более точно оценить стоимость JOIN-узлов. Таким образом, AQO применим для редко обновляющихся таблиц.

Выбирая план, AQO использует стоимостную систему, изменяя лишь стоимость узлов, зависящих от селективности запроса. AQO не будет конфликтовать с рассмотренным в этой статье динамическим компилятором запросов, так как Seq Scan не относится к такому типу вершин.

Применение AQO совместно с динамическим компилятором запросов позволило планировщику выбирать более эффективные планы для запросов с JOIN-узлами, позволяя выбрать альтернативный метод объединения из-за выбора узла сканирования, отличного от выбранного планировщиком PostgreSQL.

6. Результаты

Оценка работы планировщика, учитывающего динамическую компиляцию, проводилось на запросах из тестового набора TPC-H для базы данных объемом 30 Гб на сервере с архитектурой ARM. При тестировании запросы выполнялись многократно, бралось медианное значение времени.

В табл. 6 отражены результаты тестирования запросов из TPC-H, план которых был изменён при расчёте стоимости на основе критериев, рассчитанных тестовой системой. Ускорение вычислялось как отношение времени выполнения запросов с помощью динамической компиляции без критериев ко времени выполнения запросов с использованием динамической компиляции с критериями.

Табл. 6. Результаты тестирования динамического компилятора запросов при использовании критериев
Table 6. JIT-compiler test results using criteria

TPC-H 30 ГБ	без критериев		используя критерии		Ускорение, раз JIT $\frac{\text{без критериев}}{\text{с критериями}}$	Ускорение, раз $\frac{PG \text{ без критериев}}{JIT \text{ с критериями}}$
	PG, сек.	JIT, сек.	PG, сек.	JIT, сек.		
Q06	22,52	20,63	32,30	8,76	2,35	2,57
Q12	53,40	48,60	73,42	43,48	1,11	1,22
Q18	95,90	58,78	102,52	59,05	0,99	1,62

Для некоторых запросов время исполнения не изменилось, даже с учётом того, что поменялся план, что связано с близкой стоимостью узлов-операторов. На примере запроса Q06 наблюдается ускорение в 2,5 раза относительно интерпретатора и в 2,35 по сравнению со старым планом, использующим динамическую компиляцию. Для запроса Q12 наблюдается ускорение в 1,11 раза.

В табл. 7 представлены результаты тестирования с использованием AQO для запросов из таблицы выше. Ускорение вычислялось, как отношение времени выполнения запросов с динамической компиляцией без AQO ко времени выполнения запросов с использованием динамической компиляции с AQO.

Табл. 7. Результаты тестирования динамического компилятора запросов с использованием AQO
Table 7. JIT-compiler test results using AQO

TPC-H 30 ГБ	не используя AQO		используя AQO		Ускорение, раз JIT $\frac{\text{без } AQO}{\text{с } AQO}$	Ускорение, раз $\frac{PG \text{ без } AQO}{JIT \text{ с } AQO}$
	PG, сек.	JIT, сек.	PG, сек.	JIT, сек.		
Q06	22,52	20,63	22,52	20,56	1,00	1,09
Q12	53,40	48,60	53,45	47,47	1,02	1,12
Q18	95,90	58,78	65,15	44,49	1,32	2,15

Для запроса Q18 было получено ускорение в 1.32 раз при использовании AQO совместно с динамическим компилятором запросов относительно динамического компилятора без AQO.

Это объясняется тем, что в плане запроса последовательное сканирование таблицы было заменено индексным, что позволило ускорить соединение таблиц. Выбор плана выполнения запроса, зависящий от распаковки атрибутов, рассмотрен на примере выполнения запроса из листинга 7.

```
EXPLAIN (analyze) SELECT COUNT(*) FROM deformingTest WHERE COLUMN[index] < 18;
```

Листинг 7. Тестовый запрос оценки работы планировщика в зависимости от распаковки атрибуто.
Listing 7. Test query that examines planner choice depending on attribute extraction

Для каждого столбца таблицы *deformingTest* был создан индекс, что позволило планировщику выбирать между сканированием по индексу и последовательным сканированием. Тестирование было выполнено при использовании динамической компиляции с учётом критериев. Время выполнения запросов и выбранный планировщиком путь представлены в табл. 8.

Табл. 8. Результаты тестирования времени распаковки динамическим компилятором с использованием критериев
Table 8. Deforming test results using JIT-complier

index	JIT без критериев, сек.	JIT с критериями, сек.
0	10,75 (Bitmap Index Scan)	9,97 (Seq Scan)
39	10,72 (Bitmap Index Scan)	10,74 (Bitmap Index Scan)

Планировщик, не использующий критерии, выбирает путь Bitmap Index Scan для этого запроса вне зависимости от номера атрибута в выражении, что приводит к потере эффективности при обращении к нескольким первым столбцам. Выбор узла Seq Scan приводит к потере эффективности при обращении к атрибутам с номером больше некоторого порогового значения. Таким образом, планировщик, учитывающий сформированные критерии, выбирает оптимальный по времени план, основываясь, в том числе и на стоимости распаковки атрибутов.

7. Заключение

В ходе данной работы были рассмотрены эвристические методы выбора плана для СУБД PostgreSQL, учитывающие исполнение запроса с помощью динамической компиляции. Была добавлена поддержка команды EXPLAIN с параметром ANALYZE для динамического компилятора запросов для СУБД PostgreSQL, использованная для профилирования выполнения запросов. На основе профиля выполнения запроса были сформированы критерии оптимизации выполнения, которые уточняли стоимость узла-оператора Seq Scan при выполнении запросов с динамической компиляцией. В будущем планируется обобщение алгоритмов для расчёта критериев для различных узлов-операторов.

Планировщик PostgreSQL использует стоимостную модель, которая была адаптирована для расчёта эффективных планов для выполнения запросов с динамической компиляцией. Был сформирован набор запросов на основе TPC-H, позволяющих уточнить стоимость узлов-операторов, основываясь на результатах профилирования. При использовании критериев для выполнения планов из набора TPC-H на сервере с архитектурой ARM, выбирались планы более эффективные с точки зрения динамической компиляции, что позволило уменьшить время выполнения вплоть до двух раз.

В ходе работы были также рассмотрены альтернативные критерии для выбора оптимальных планов, например, использование расширения AQO. Разработанный эвристический метод позволяет также использовать модели совместно с ним и иные алгоритмы. Так, например, планировщик PostgreSQL, используя расширение AQO с введёнными критериями, уменьшает время выполнения вплоть до 1,32 раз за счёт изменения плана.

Список литературы / References

[1] Бучацкий Р.А., Шарыгин Е.Ю. и др. Динамическая компиляция SQL-запросов для СУБД PostgreSQL. Труды ИСП РАН, том 28, вып. 6, 2016, стр. 37-48 / Buchatskiy R.A., Sharygin E.Y. et al. 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.

[2] Шарыгин Е.Ю., Бучацкий Р.А. и др. 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. et al. 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.

[3] Sharygin E., Buchatskiy R., et al. Runtime specialization of PostgreSQL query executor. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 375-386.

[4] The LLVM Compiler Infrastructure. Available at: https://llvm.org/docs/, accessed 13.07.2022

[5] Adaptive Query Optimization. Available at: https://github.com/postgrespro/aqo, accessed 13.07.2022

[6] PostgreSQL 9.6.24 Documentation. Available at: https://postgrespro.com/docs/postgresql/9.6/index, accessed 13.07.2022

[7] Using Explain Analyze in PostgreSQL. Available at: https://postgrespro.ru/docs/enterprise/9.6/using-explain\#using-explain-analyze, accessed 13.07.2022.

[8] 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.

[9] TPC-H, an ad-hoc, decision support benchmark. Available at: http://www.tpc.org/tpch/, accessed 13.07.2022.

[10] Ivanov O., Bartunov S. Adaptive Cardinality Estimation. arXiv:1711.08330, 2017, 12 p.

[11] Zhang R., Debray S., Snodgrass R.T. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. In Proc. of the Tenth International Symposium on Code Generation and Optimization, 2012, pp. 63-73.

[12] Wu W., Chi Y. et al. Predicting query execution time: Are optimizer cost models really unusable? In Proc. of the IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 1081-1092.

Информация об авторах / Information about authors

Егор Викторович ДОЛГОДВОРОВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: компиляторные технологии, оптимизации.

Egor Viktorovich DOLGODVOROV – A Student at MIPT, Laboratory Assistant in Compiler Technology department at ISP RAS. Research interests: compiler technologies, optimizations.

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

Ruben Arturovich BUCHATSKIY – PhD, Researcher in 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.

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Dmitry Mikhailovich MELNIK – Senior Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.