Обзор методов динамической компиляции запросов

[1,2] *Е. Ю. Шарыгин <eush@ispras.ru>*
[1] *Р. А. Бучацкий <ruben@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Эффективное использование процессора является решающим фактором производительности аналитических систем, особенно с увеличением размеров обрабатываемых данных. В то же время возрастающие объёмы доступной основной памяти позволяют значительно сократить количество обращений к медленным дисковым хранилищам и тем самым отводят традиционные для большинства систем обработки данных оптимизации подсистемы ввода–вывода на второй план. Одним из наиболее эффективных способов повышения эффективности использования процессора и сокращения накладных расходов, прежде всего проявляющихся в затратах на интерпретацию планов запросов, является компиляция запросов в исполняемый код во время выполнения (динамическая компиляция). В последнее время наблюдается рост интереса к методам динамической компиляции запросов как в академических, так и в прикладных разработках. Данная статья является обзором литературы в области динамической компиляции запросов, в основном для реляционных СУБД. Представлены работы последних лет, описаны архитектурные особенности методов, сделана классификация работ, приведены основные результаты.

Ключевые слова: динамическая компиляция; JIT-компиляция; языки запросов; SQL; push-модель; специализация кода.

DOI: 10.15514/ISPRAS-2017-29(3)-11

Для цитирования: Шарыгин Е. Ю., Бучацкий Р. А. Обзор методов динамической компиляции запросов.Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 179-224. DOI: 10.15514/ISPRAS-2017-29(3)-11

1. Введение

Одной из основных функций современных СУБД [1] является предоставление интерфейса для выполнения операций по определению, изменению или выборке данных. Чаще всего такой интерфейс реализуется посредством специализированного языка запросов, например, SQL, что позволяет максимально изолировать задачи, решаемые СУБД, от других частей информационной системы.

Процесс обработки запросов в большинстве СУБД, таким образом, включает в себя несколько этапов:

1. синтаксический анализ, восстановление структуры запроса из текстового представления;
2. семантический анализ, определение используемых таблиц, атрибутов, типов данных, функций и т. д.;
3. составление и оптимизация плана выполнения запроса;
4. выполнение (интерпретация) плана запроса.

Можно заметить, что только затраты на выполнение этапа 4 (выполнения плана) зависят не только от сложности самого запроса, но и от размера данных, обрабатываемых СУБД, и поэтому доминируют над затратами на выполнение этапов 1–3 с увеличением размера данных.

Системы, работающие по приведённой выше схеме обработки запросов, в рамках данной статьи будем называть *классическими*. Недостатки классической схемы проявляются в следующих сценариях использования:

- Если информационная система отправляет много запросов, большинство из которых принадлежит одному из сравнительно небольшого числа различных классов, то этапы 1–3 в классической схеме для каждого класса запросов выполняются многократно. Современные СУБД решают эту проблему при помощи поддержки параметризованных запросов (prepared statements). Параметризованный запрос анализируется, транслируется в план выполнения и оптимизируется только один раз во время определения, и во время выполнения оптимизированный план переиспользуется для всех экземпляров запроса. В данной статье параметризованные запросы в дальнейшем не рассматриваются.
- Если большинство запросов, отправляемых информационной системой, принадлежит ограниченному числу запросов, известных во время разработки системы, то в классической схеме все перечисленные этапы выполняются многократно, несмотря на то, что как план выполнения, так и параметры запроса (если они есть) остаются на протяжении работы системы неизменными. Параметризация запросов позволяет избежать выполнения этапов 1–3, но не этапа 4 (собственно выполнения), алгоритмическая сложность которого доминирует над остальными этапами. Накладных расходов на интерпретацию можно избежать в этом случае при помощи (статической) компиляции запросов в машинный код [2–4] при сборке информационной системы. Статическая компиляция в дальнейшем также не рассматривается.
- Если запросы, отправляемые к СУБД, достаточно сложны и выполняются достаточно долго, накладные расходы на интерпретацию плана в классической схеме (этап 4) могут составлять значительную часть в общей времени обработки запроса. Решением этой проблемы является динамическая компиляция, методам реализации которой и посвящена данная статья.

Методы динамической компиляции предполагают 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. Учёт структуры и константных данных запроса и 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 размер данных был достаточно велик: $N > N_0$.

Существующие методы динамической компиляции — и СУБД, их использующие, — можно разделить на несколько классов:

- компиляция выражений (раздел 2) — компиляция таких частей запросов, как арифметические и логические выражения и доступ к атрибутам, при сохранении интерпретации плана запроса;
- компиляция запросов (раздел 3) — компиляция запросов целиком, выполнение без интерпретации;
- автоматическая специализация (раздел 4) — специализация интерпретатора запросов во время выполнения к данным и структуре запроса.

2. Компиляция выражений и горячих участков кода

Динамическая компиляция выражений позволяет избежать накладных расходов на исполнение обобщённого кода СУБД благодаря компиляции некоторых «горячих» функций СУБД в машинный код во время выполнения с учётом конкретного запроса. Множество рассматриваемых функций-кандидатов включает в себя:

1. Функции, осуществляющие вычисление арифметических выражений. Так как выражения в запросе становятся известны только во время выполнения, вычисление выражений, как правило, реализуется в классических СУБД при помощи интерпретации, накладные расходы, связанные с которой, динамическая компиляция позволяет избежать.
2. Пользовательские функции. Многие системы позволяют использовать в запросах функции, определённые пользователем или на процедурных языках, поддержка которых встроена непосредственно в язык запросов, или через API (например, на C или C++). Вызов пользовательской функции во время выполнения запроса сопряжён с накладными расходами на поддержку абстракции: интерфейс, позволяющий пользовательской функции получать значения и типы параметров и устанавливать возвращаемое значение как правило реализован обобщённо. Кроме того, тело функции также обычно неизвестно до начала выполнения запроса. Динамическая компиляция позволяет встроить тело пользовательской функции непосредственно в машинный код вызывающей функции и удалить

накладные расходы за счёт оптимизации функции в месте вызова.

3. Функции доступа к атрибутам. В СУБД, использующих построчное хранение кортежей (N-ary Storage Model [1, 5]), атрибуты каждого конкретного кортежа располагаются в памяти последовательно. Смещение заданного атрибута в кортеже определяется его порядковым номером и типами предшествующих атрибутов (которые, в свою очередь, в случае реляционной модели данных определяются заголовком отношения) и потому вычисляется во время выполнения запроса. Динамическая компиляция позволяет предвычислить или существенно оптимизировать (в случаях, когда смещение атрибута также зависит и от *значений* предшествующих атрибутов [6]) вычисление смещений атрибутов, используя информацию о таблицах, к которым производится обращение.
4. Функции, использующие константные данные времени выполнения. Динамическая компиляция позволяет встроить в код значения не изменяемых во время выполнения глобальных переменных и полей структур, например, параметров работы СУБД или номера выполняемой транзакции (снимка таблицы) при использовании механизма многоверсионности (MVCC). Можно заметить, что в эту категорию входят также внутренние структуры интерпретатора выражений и заголовки таблиц, что делает описываемый здесь класс функций обобщением классов функций, описываемых выше.

При компиляции выражений сохраняется общий механизм интерпретации плана запроса, компиляции которого посвящена раздел 3.

2.1 Impala (2014)

Impala [7, 8] — это распределённая аналитическая система обработки данных для Apache Hadoop [9], использующая в качестве хранилища данных распределённую файловую систему Apache HDFS [9] или распределённую нереляционную СУБД Apache HBase [10].

Impala предоставляет интерфейс запросов на языке SQL и использует LLVM [11] для компиляции частей запроса в машинный код во время выполнения.

В качестве примера приводится функция доступа к атрибутам кортежа в памяти: динамическая компиляция позволяет специализировать общий код доступа к атрибутам, реализующий поддержку многих типов данных, с использованием доступной во время выполнения информации о том, к каким таблицам в данном запросе производится обращение. Поскольку эта функция вызывается для каждого обрабатываемого в цикле кортежа, даже удаление небольшого числа инструкций приводит к значительному приросту производительности.

Среди оптимизаций, которые удалось применить во время динамической компиляции, авторы выделяют:

- подстановку констант времени выполнения;
- удаление условных переходов;
- удаление операций загрузки из памяти;
- разворачивание циклов;
- встраивание виртуальных вызовов.

В частности, арифметические и логические выражения в Impala представляются в виде иерархии классов с перегруженным методом вычисления значения, и встраивание вызовов виртуальных функций позволяет значительно сократить связанные с этим накладные расходы (см. рис. 1).

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}

SELECT my_func(col1 + 10, col2) FROM ...
```

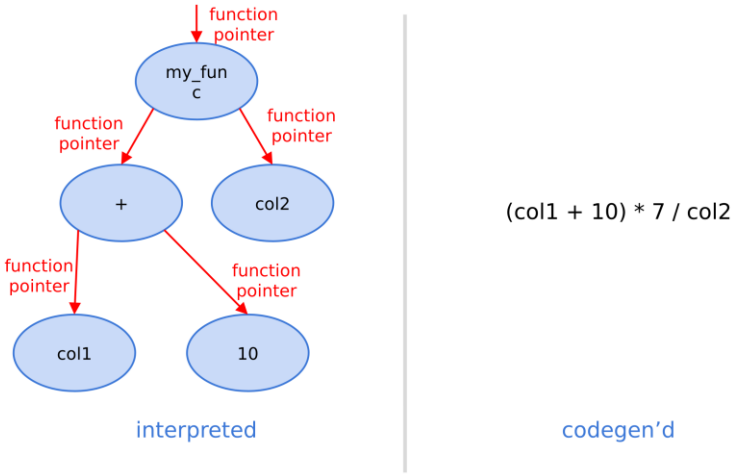


Рис. 1. Компиляция выражений в Impala
Fig. 1. Expression compilation in Impala

В [8] также описывается подход к разработке динамического компилятора с использованием инфраструктуры LLVM, при котором оптимизируемые функции реализуются на языке высокого уровня (C++) и компилируются статически в LLVM IR и в объектный код, что позволяет переиспользовать один и тот же исходный код функций как в компиляторе, в котором производится связывание полученного кода на LLVM IR с генерируемым модулем (единицей трансляции) LLVM во время выполнения, так и в интерпретаторе, с которым полученный код связывается статически. Этот же метод также применяется и для оптимизации пользовательских функций на C++, которые связываются с использующим их кодом при использовании интерпретатора динамически через динамически подгружаемую библиотеку, а

при использовании компилятора — статически во время выполнения.
Динамическая компиляция выражений в Impala позволяет получить ускорение до 5.7 раз на запросе Q1 (который входит в состав бенчмарка TPC-H [12]), при этом количество выполняемых инструкций и ветвлений сокращается в 4.29 и 3.76 раз, соответственно.

2.2 Spark SQL (2015)

Apache Spark [13] — это фреймворк для реализации распределённой обработки слабоструктурированных и неструктурированных данных для Apache Hadoop. Основными областями применения Spark являются потоковая обработка данных, обработка графовых данных и машинное обучение.
Spark SQL [14] — это модуль Apache Spark, реализующий поддержку реляционной модели обработки данных и декларативного языка запросов SQL.
В Spark SQL применяется динамическая компиляция арифметических и логических выражений. Для генерации промежуточного представления (абстрактного синтаксического дерева Scala) и компиляции его в байткод виртуальной машины Java используются встроенные средства языка Scala (quasiquotes).
На простых запросах динамическая компиляция выражений в Spark SQL позволяет получить ускорение до 3.5 раз по сравнению с интерпретацией.

2.3 Компиляция выражений в PostgreSQL

PostgreSQL [15] — высокопроизводительная объектно-реляционная СУБД, поддерживающая язык запросов SQL и расширяемая пользовательскими функциями, операторами, индексами, типами данных и процедурными языками. PostgreSQL использует построчное хранение кортежей (N-ary Storage Model [1, 5]).
В число СУБД, основанных на PostgreSQL [16], входят нереляционная СУБД ToroDB [17], колоночная СУБД Vertica [18], графовая СУБД AgensGraph [19], аналитическая СУБД VitesseDB [20], распределённые аналитические СУБД ParAccel [21], Redshift [22], Greenplum [23] и DeepgreenDB [24] и многие другие. Динамическая компиляция на уровне выражений или запросов используется по крайней мере в VitesseDB, ParAccel, Redshift, Greenplum и DeepgreenDB.
В этом подразделе рассмотрим некоторые существующие исследования по разработке и реализации динамической компиляции выражений для PostgreSQL.

2.3.1 Микроспециализация (2012)

Метод микроспециализации [25–27] заключается в замене некоторых

критических для производительности функций или участков кода СУБД на версии, специализированные под конкретную таблицу, запрос или кортеж на основе специально подготовленных шаблонов кода. [26] приводит три других способа специализации кода в СУБД:

- Архитектурная специализация, заключающаяся в адаптировании архитектуры СУБД под конкретный класс приложений. Примером является использование колоночного хранения или потоковой обработки данных.
- Компонентная специализация заключается в разработке специализированных компонентов СУБД, ориентированных под конкретные типы данных или сценарии использования. Примерами являются новые типы операторов и индексов.
- Пользовательская специализация, которая состоит в предоставлении пользователю средств специализации выполнения запросов за счёт пользовательских функций.

Исследователи рассматривают цикл обработки запросов в СУБД и отмечают, что разные переменные получают значение на разных этапах его выполнения. Например, переменные, определяемые данными таблицы, могут становиться известны только после чтения соответствующих буферов базы данных во время обработки запроса, переменные, связанные со структурой запроса, известны после получения и анализа запроса, в то время как переменные, описывающие заголовки таблиц и конфигурационные параметры СУБД, известны уже после начала работы СУБД и редко меняются.

В предлагаемой схеме разработчик определяет функции-кандидаты для специализации, удовлетворяющие следующим требованиям:

1. Функция-кандидат должна вызываться в цикле обработки запросов.
2. Функция-кандидат должна занимать существенную долю в общем времени выполнения запроса.
3. Функция-кандидат должна содержать обращения к переменным, значения которых инвариантны на всём протяжении выполнения тела цикла обработки запроса.
4. Функция-кандидат должна существенно зависеть от этих переменных и допускать эффективную специализацию при фиксированных значениях переменных.

Примерами функций-кандидатов являются функции вычисления выражений, функции доступа к атрибутам, функции, реализующие конкретные операторы плана выполнения [27]. Исследователи использовали профилировщик Callgrind [28] для выявления функций-кандидатов, удовлетворяющих условию 2.

В зависимости от того, на каком этапе выполнения СУБД становятся известны значения переменных, используемых в функции-кандидате, подразделяются три вида шаблонов специализированных функций (“bees” в терминологии [25–27]):

- relation bee (переменные определяются схемой таблицы),

- query bee (переменные определяются запросом) и
- tuple bee (переменные определяются значениями конкретных атрибутов в кортеже).

Например, на рис. 2 и рис. 3 представлен пример relation bee — функции доступа к атрибутам в СУБД PostgreSQL — до и после специализации, при этом цветом отмечены инвариантные переменные.

```
1 void slot_deform_tuple(TupleTableSlot *slot, int natts) {
2   ...
3   tp = (char *) tup + tup->t_hoff;
4   for (; attnum < natts; attnum++) {
5     Form_pg_attribute thisatt = att[attnum];
6     if (hasnulls && att_isnull(attnum, bp)) {
7       values[attnum] = (Datum) 0;
8       isnull[attnum] = true;
9       slow = true;
10      continue;
11    }
12    isnull[attnum] = false;
13    if (!slow && thisatt->attcacheoff >= 0) {
14      off = thisatt->attcacheoff;
15    } else if (thisatt->attlen == -1) {
16      if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
17        thisatt->attcacheoff = off;
18      } else {
19        if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
20          thisatt->attcacheoff = off;
21        } else {
22          off = att_align_pointer(off, thisatt->attalign, -1, tp + off);
23          slow = true;
24        }
25      } else {
26        off = att_align_nominal(off, thisatt->attalign);
27        if (!slow)
28            thisatt->attcacheoff = off;
29      }
30      values[attnum] = fetchatt(thisatt, tp + off);
31      off = att_addlength_pointer(off, thisatt->attlen, tp + off);
32      if (thisatt->attlen <= 0)
33          slow = true;
34    }
35    ...
36  }
37 }
```

Рис. 2. Функция доступа к атрибутам в PostgreSQL
Fig. 2. Attribute access function in PostgreSQL

Шаблоны могут инстанциироваться или заранее, если возможные значения переменных, связанных, например, с конкретными атрибутами кортежей или параметрами отношений или запросов, принадлежат ограниченному диапазону, — в этом случае для каждой комбинации возможных значений переменных, участвующих в шаблоне, инстанциируется по отдельной специализированной функции — или непосредственно после получения переменными своих значений во время выполнения. Для relation bees это время определения схемы таблицы, для query bees — время построения плана выполнения запроса, и для tuple bees — время вставки или изменения данных таблицы.

Во время инстанциирования происходит заполнение «дыр» в шаблонах конкретными значениями переменных и дальнейшая оптимизация — свёртка и продвижение констант, удаление ветвлений.

```
1 void GetColumnsToLongs(char bee_id, int address, char* data, int* start_att,
2                        int* offset, bool* isnull, Datum* values) {
3   *(long*)isnull = 0;
4   isnull[8] = 0;
5   values[0] = *(int*)data;
6   values[1] = *(int*)(data + 4);
7   values[2] = (long)(address + bee_id * 32 + 1000);
8   *start_att = 3;
9   if (end_att < 4) return;
10  *offset = 8;
11  if (*offset != (((long)(*offset) + 3) & ~((long)3)))
12    if (!(*(char*)(data + *offset)))
13        *offset = (long)(*offset + 3) & ~(long)3;
14  values[3] = (long)(data + *offset);
15  *offset += VARSIZE_ANY(data + *offset);
16  *offset = ((long)(*offset) + 3) & ~((long)3);
17  values[4] = (*(long*)(data + *offset)) & 0xffffffff;
18  *offset += 4;
19  values[5] = (long)(address + bee_id * 32 + 1001);
20  *start_att = 6;
21  if (end_att < 7) return;
22  if (!(*(char*)(data + *offset)))
23      *offset = (long)(*offset + 3) & ~(long)3;
24  values[6] = (long)(data + *offset);
25  *offset += VARSIZE_ANY(data + *offset);
26  values[7] = *(int*)(address + bee_id * 32 + 1002);
27  if (!(*(char*)(data + *offset)))
28      *offset = (long)(*offset + 3) & ~(long)3;
29  values[8] = (long)(data + *offset);
30  *start_att = 9;
31 }
```

Рис. 3. Функция доступа к атрибутам после специализации
Fig. 3. Specialized attribute access function

В [25, 26] описывается архитектура системы HIVE, которая служит для построения, компиляции, инстанциирования шаблонов кода, а также кеширования и удаления инстанциированных шаблонов при удалении соответствующих объектов (схем, таблиц) из базы данных.

В [27] описывается механизм горячей замены специализированных функций, который позволяет совмещать выполнение нескольких специализированных функций во время выполнения одного запроса. Для этого при изменении значений переменных, используемых в специализированной функции, происходит изменение кода вызывающей функции для замены адресов функций в инструкциях вызова. Механизм позволяет эффективно специализировать код операторов с несколькими внутренними состояниями: например, оператор JOIN, состояния которого определяют, из какого дочернего оператора происходит считывание кортежей.

Подход реализован для СУБД PostgreSQL и на бенчмарке TPC-H показывает средний прирост производительности в 12.4%, при этом на разных запросах прирост производительности составил от 1.4% до 32.8%. На бенчмарке TPC-C [29] средний прирост производительности превысил 11%.

2.3.2 Butterstein, Grust (2016)

В работе [30] приведены результаты профилирования на запросе TPC-H

Q1 [12], согласно которому вычисление арифметических и логических выражений занимает до 70% от суммарного времени обработки запроса.

В работе предлагается метод компиляции выражений в запросах в машинный код с использованием инфраструктуры LLVM [11]. Метод основан на «дырах» в генерируемом коде на LLVM IR, которые заполняются по мере генерации кода для всего выражения.

В работе предлагается оптимизация, направленная на минимизацию количества вызовов функции `slot_getattr`, предоставляющей в PostgreSQL доступ к значениям атрибутов кортежа, при помощи разделения «дыр» по множеству атрибутов, значения которых к моменту достижения «дыр» во время выполнения гарантировано извлечены, и дублированию кода под разные «дыры» с добавлением необходимых вызовов `slot_getattr` где необходимо.

Например, на рис. 4 приведён результат компиляции логического выражения $\big(p_1(A) \wedge p_2(B)\big) \vee p_3(A,B)$ с двумя дырами F_1 и F_2, которые соответствуют случаям, когда первый дизъюнкт принимает значение «ложь», и при этом загружены или только атрибут `A`, или и `A`, и `B`. Код, сгенерированный в эти «дыры» во время компиляции оператора `OR`, отличается наличием в случае F_1 загрузки атрибута `A`, в то время как в случае F_2 это не требуется.

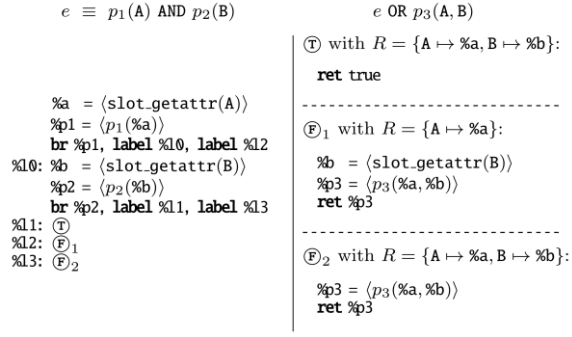


Рис. 4. Разделение «дыр» при компиляции логического выражения
Fig. 4. Expression compilation with hole splitting

Метод реализован в СУБД PostgreSQL и показывает ускорение до 37% на бенчмарке TPC-H.

2.3.3 Компиляция выражений в компиляторе, разрабатываемом в ИСП РАН (2016)

В ИСП РАН разрабатывается компилятор запросов для PostgreSQL с использованием LLVM, частью которого является компилятор выражений [6].

В работе приведены результаты профилирования выполнения запросов

из бенчмарка TPC-H [12], согласно которым на некоторых запросах вычисление предикатов оператора WHERE занимает более 50% от общего времени выполнения.

При компиляции выражения осуществляется обход представляющего его синтаксического дерева и генерация соответствующего кода на LLVM IR. Вопрос необходимости переписывания на LLVM API всего множества функций, доступных для вызова из выражений в запросах на варианте языка SQL, реализованном в PostgreSQL, решён при помощи метода предкомпиляции (рис. 5), который предполагает предварительное преобразование исходного кода PostgreSQL, реализующего встроенные функции, в вид, доступный кодогенератору. Преобразование осуществляется в два этапа:

1. Трансляция исходного кода PostgreSQL в биткод LLVM при помощи компилятора Clang [31].
2. Трансляция полученного биткода функций в последовательности вызовов LLVM API на языке C++ для воссоздания соответствующего промежуточного представления в памяти при помощи компиляции под встроенную в LLVM псевдоплатформу CPPBackend.

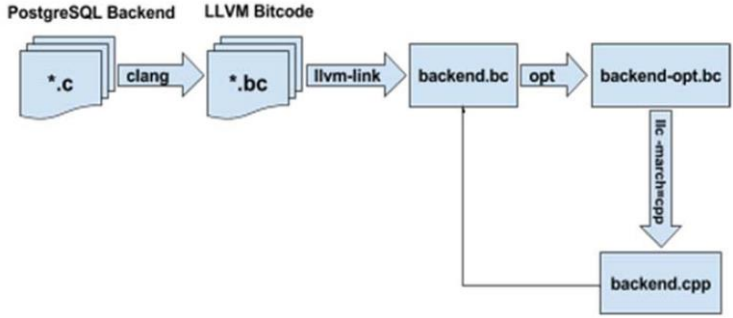


Рис. 5. Метод предкомпиляции выражений
Fig. 5. Expression precompilation method

В [6] также описана оптимизация доступа к атрибутам кортежа, основанная на предподсчёте разницы между смещениями извлекаемых атрибутов с использованием доступной в заголовке таблицы информации о типах атрибутов и установленных флагах `attnotnull` и `attlen`, определяющих, имеет ли атрибут фиксированный размер в памяти или нет. Последовательности неиспользуемых в выражении атрибутов фиксированной длины в скомпилированном коде пропускаются. Оптимизация позволяет уменьшить число извлекаемых атрибутов до $\{n \leq m(N) \vee n \in N \vee \neg attfixed(n)\} \vee$, где N — множество используемых атрибутов, по сравнению с обобщённой реализацией в интерпретаторе, которая требует извлечения всех $m(N)$ атрибутов с номерами в промежутке до максимального номера атрибута, используемого в выражении.

Представленный метод реализован в виде расширения к PostgreSQL и показывает ускорение до 4.7 раз на некоторых запросах.

2.3.4 Greenplum (2016)

В [23] описывается использование динамической компиляции запросов с использованием LLVM в СУБД Greenplum.

Исследователи выделяют два способа применения динамической компиляции в СУБД: компиляция выражений и компиляция запросов — и разделяют некоторые существующие СУБД в соответствии с этим на два класса. Динамическая компиляция в Greenplum используется на уровне выражений и горячих функций, что позволяет расширять множество компилируемых функций инкрементально в рамках существующей СУБД.

Результат профилирования на запросах TPC-H [12] выделил три множества функций-кандидатов для компиляции: функции доступа к атрибутам, функции интерпретатора выражений и агрегатные функции.

Процесс замены каждой отдельной функции состоит из следующих шагов (рис. 6):

1. Замена непосредственных вызовов функции вызовами по указателю, помещённому во внутренние структуры данных СУБД.
2. Компиляция функции-кандидата в LLVM IR и в машинный код во время инициализации запроса.
3. Замена указателей в структурах данных на указатели в результирующий машинный код.
4. Освобождение скомпилированного кода и сбрасывание указателей во время финализации запроса.

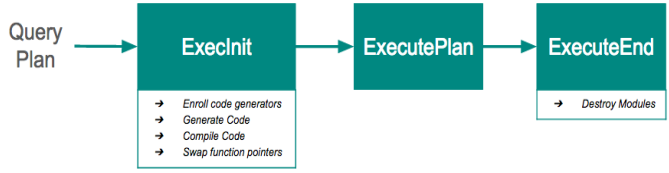


Рис. 6. Компиляция горячих функций в Greenplum
Fig. 6. Hotspot compilation in Greenplum

Метод позволяет компилировать только те обобщённые функции, выполнение которых связано с существенными накладными расходами в профиле работы СУБД, а также вводить определённые ограничения при компиляции каждой конкретной функции, при нарушении которых прерывать компиляцию до шага 3 (замены указателей), что приведёт к использованию при выполнении запроса обобщённой версии соответствующей функции. Недостатком является необходимость сохранения интерфейсов между обобщённым и компилируемым кодом.

На запросе TPC-H Q1 применение динамической компиляции позволило

получить двукратное ускорение, которое складывается в основном из ускорения функций интерпретатора выражений (1.25 раза) и агрегатных функций (1.35 раза).

3. Компиляция запросов

Классической моделью выполнения, используемой в интерпретаторах запросов, является модель Volcano [32], предложенной в одноимённой системе выполнения запросов.

В модели Volcano результатом выполнения каждого оператора в плане выполнения запроса является итератор, предоставляющий доступ к последовательности возвращаемых кортежей. Функцией каждого оператора является формирование последовательности возвращаемых кортежей из последовательностей кортежей, принимаемых на вход от дочерних операторов. Например, оператор агрегации, применённый к отсортированной последовательности входных кортежей, осуществляет группировку кортежей по значению одного или нескольких атрибутов и вычисление аргегатных функций, при этом результирующая последовательность состоит из одного кортежа на каждую группу кортежей во входной последовательности.

Интерфейс итератора состоит из трёх методов:

- метод `open` вызывается для инициализации внутреннего состояния оператора;
- метод `next` вызывается для вычисления и возврата *одного* следующего кортежа в результирующей последовательности оператора, при достижении конца результирующей последовательности `next` возвращает специальный символ конца последовательности;
- метод `close` вызывается для освобождения ресурсов и сброса внутреннего состояния оператора (при этом конец последовательности мог и не быть достигнут).

Модель Volcano упрощает разработку операторов за счёт применения принципа декомпозиции, позволяет распределять операторы по различным вычислительным узлам и естественным образом представлять бесконечные последовательности, поскольку вычисление каждого конкретного члена результирующей последовательности откладывается до того момента, как значения его атрибутов потребуются для вычисления очередного кортежа в родительском операторе.

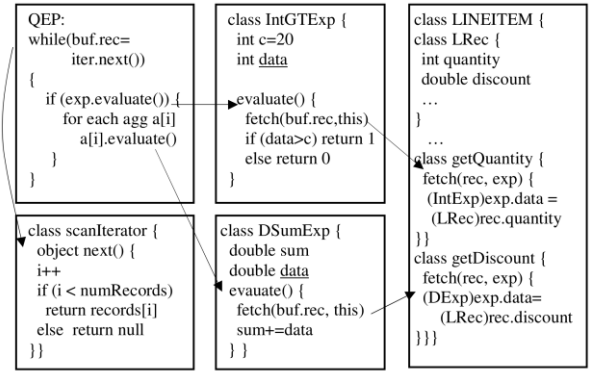
В типичном случае, однако, модель Volcano крайне неэффективна, потому что для получения каждого следующего кортежа требуется многократные сохранение и загрузка состояний операторов в соответствующем поддерева плана выполнения запроса.

Альтернативой является модель явных циклов (push-based модель [33]), в которой дочерний оператор возвращает результирующие кортежи родительскому оператору посредством вызова соответствующего

обработчика, принимающего очередной кортеж и продолжающего его обработку. При этом загрузка состояния дочернего оператора не требуется.
Push-based модель используется в компиляторах запросов, обзору которых посвящён этот раздел, при этом компиляция декомпозируется за счёт введения соответствующих абстракций времени компиляции.

3.1 JamDB (2006)

JamDB [34] — это реляционная СУБД в основной памяти, реализованная на Java. JamDB поддерживает построчное хранение кортежей (N-ary Storage Model) как Java-объектов в куче JVM, первичные и внешние ключи, трёхзначную логику, индексирование.
Для JamDB реализовано два движка запросов: интерпретатор и компилятор. Интерпретатор следует классической модели Volcano и состоит из трёх основных компонент: итераторы, соответствующие вершинам дерева плана, интерпретатор арифметических и логических выражений и интерпретатор агрегаций. Интерпретатор реализован обобщённо с использованием указателей на функции и данные типа `void*`, значения которых становятся известны только во время выполнения, потому что код интерпретатора используется для вычисления многих запросов, не известных во время сборки интерпретатора, и поэтому для каждого конкретного запроса реализация, предоставляемая интерпретатором, далека от оптимальной.



(a) Interpreted Plan

```
class Q1 {  class Q1Agg { double sum1; }
            execute(){
               Lrecords = (LRecord[])records;
               while (i < numRecords) {
                  rec = Lrecords[i];
                  if (rec.quantity > 20) q1agg.sum1+=rec.discount;
                  i++ ;
            }}}
```

(b) Compiled Plan

Рис. 7. Сравнение интерпретируемого и компилируемого планов JamDB
Fig. 7. Interpreted and compiled plans in JamDB

Компилятор запросов использует push-модель выполнения и реализован на основе виртуальной машины JVM. Код генерируется во время обхода дерева плана, начиная с листовых вершин. По завершении обхода фрагменты кода объединяются в одну функцию и компилируются и загружаются в программу как Java-класс. Использование JVM позволяет существенно упростить реализацию компилятора, в частности, генерацию кода и спекулятивную оптимизацию горячих участков, которая позволяет получить производительность, сравнимую с аналогичным кодом, реализованным на языке C. Динамическая компиляция позволяет специализировать код под конкретный SQL-запрос, провести межоператорные оптимизации и девиртуализировать и встроить вызовы, например, арифметических функций в выражениях запроса.

Сравнение интерпретатора запросов с кодом, генерируемым компилятором, приведено на рис. 7.

Исследователи отмечают, что в системах длительного хранения накладные расходы на интерпретацию проявляются не так чётко, как в системах в основной памяти из-за того, что затраты на доступ к данным часто доминируют над затратами на вычисления.

В [34] также отмечается реализация длительного хранения для JamDB, что

потребует структуры данных, позволяющей минимизировать затраты на загрузку и запись данных на диск за счёт разделения данных на страницы или буферы, и соответствующей системы управление буферами в памяти.

Исследователи также отмечают возможность упростить реализацию компилятора за счёт использования технологии Java Emitter Templates (JET) [35].

Экспериментальные результаты на бенчмарке TPC-H показывают, что производительность компилятора запросов, реализованного JamDB, в 2 раза превышает производительность интерпретатора, которая, в свою очередь, значительно превышает производительность коммерческой СУБД DB2 [36]. При этом производительность компилятора запросов превышает производительность планов, реализованных вручную на C/C++.

3.2 DBToaster (2009)

DBToaster [37] — это система потоковой обработки данных, основанная на вычислении и актуализации стационарных запросов и представлений к потоку изменений, добавлений и удалений кортежей в таблицах базы данных.

Исследователи отмечают, что традиционные интерпретаторы и компиляторы запросов не справляются с задачами потоковой обработки данных, поскольку вынуждены вычислять запросы заново при любом изменении данных и не могут использовать результаты предыдущих вычислений.

Подход, предлагаемый в DBToaster, основан на вычислении при помощи алгебраических преобразований для каждого стационарного запроса специального набора разностных запросов на каждую таблицу базы данных и на каждый тип изменения этой таблицы (*update, insert, delete*), которые будут срабатывать на соответствующих типах изменений и выполнять актуализацию результата стационарного запроса к изменению данных, что позволяет избежать повторных вычислений.

Обновление одного стационарного запроса может потребовать введения новых вспомогательных стационарных запросов, возможно параметризованных. В DBToaster автоматически вычисляется результирующее множество стационарных запросов, которые требуется вычислять и обновлять при изменениях таблиц, и множество разностных запросов, нужных для эффективного вычисления, после чего разностные запросы компилируются в машинный код при помощи JIT-компилятора LLVM.

В статье рассматриваются применения DBToaster в контексте алгоритмической торговли и аналитических запросов: потоковая обработка данных и инкрементальное вычисление запросов позволяет ускорить этап загрузки данных, необходимый в классических СУБД.

3.3 Компиляция запросов для LINQ

LINQ (Language Intergrated Query) [38] — это расширение языков платформы .NET, позволяющее встраивать выражения на декларативном языке запросов в программы на хостовом языке. Синтаксис LINQ-запросов во многом позаимствован из SQL, семантика определяется пользователем и реализуется на хостовом языке: источник данных реализует соответствующий интерфейс, в вызовы функций которого транслируются запросы LINQ. Интеграция с хостовым языком позволяет в LINQ-запросах использовать выражения (которые передаются источнику LINQ или в виде функций-объектов CLR, или в виде синтаксических деревьев) и систему типов хостового языка.

Источники данных реализуются в модели Volcano: реализуемый ими интерфейс `IEnumerable` состоит из метода `GetEnumerator`, который возвращает объект-итератор с методами `Current` для доступа к текущему объекту, на который указывает итератор, и `MoveNext` для продвижения итератора к следующему объекту.

LINQ позволяет использовать одни и те же декларативные запросы как для работы с реляционными данными или XML — соответствующие источники встроены в .NET — так и с распределёнными данными [39].

3.3.1 DryadLINQ (2008)

DryadLINQ [39] — это реализация LINQ на основе системы распределённой обработки данных Dryad [40]. Использование декларативного языка запросов упрощает распределённую и параллельную обработку данных и делает возможным применение высокоуровневых оптимизаций.

В DryadLINQ по входному LINQ-запросу строится граф выполнения (Execution Plan Graph) — неориентированный ациклический граф, вершинами которого являются операторы, а дугами — входы и выходы операторов. К графу выполнения применяются статические оптимизации, такие как конвейеризация операторов, удаление лишних вычислений, вставка операторов агрегации и оптимизация ввода–вывода — после чего вершины графа распределяются по вычислительным узлам и для каждого узла генерируются специализированный код сериализации данных для отправки между узлами и LINQ-выражение, осуществляющее обработку данных на самом узле, при этом вычисления также распределяются по доступным ядрам процессора при помощи библиотеки PLINQ [41]. Результирующий код обработки данных и сериализации компилируется во внутреннее представление .NET и передаётся по сети вместе с библиотеками, необходимыми для его работы.

В процессе вычисления запроса также применяются динамические оптимизации для перестроения графа выполнения во время работы исходя из статистической информации об обрабатываемых данных.

3.3.2 Steno (2011)

Steno [42] — это компилятор запросов для LINQ.

Исследователи отмечают, что накладные расходы на интерпретацию и поддержку абстракции времени выполнения (модели итераторов) замедляют (последовательное) выполнение LINQ-запросов в несколько раз по сравнению с эквивалентной реализацией вручную на императивном языке. Накладные расходы представлены в основном виртуальными вызовами функций, необходимыми как для передачи управления коду, реализующему вершины плана выполнения запроса, так и для вызова функций-параметров запроса: предикатов и проекций; и необходимостью сохранения и загрузки состояния операторов для эмуляции сопроцедур, соответствующих операторам запроса, на императивном языке реализации LINQ. Накладные расходы увеличиваются с увеличением уровня вложенности запросов.

Steno решает эту проблему за счёт динамической компиляции запросов в специализированный и оптимизированный императивный код в модели явных циклов.

Компилятор Steno реализован в виде отдельного источника данных, который транслирует абстрактное синтаксическое дерево (АСД) запроса в код на промежуточном представлении QUIL (Query Intermediate Language), на котором проводится оптимизация слияния итераторов (iterator fusion) и трансляция вложенных запросов во вложенные циклы. По коду на промежуточном представлении затем строится АСД языка C#, соответствующее классу с единственным методом, реализующим конкретный запрос, которое затем компилируется в динамическую библиотеку при помощи компилятора C# и загружается в программу. Объект загруженного класса инстанциируется при помощи механизма рефлексии и инициализируется ссылками на используемые в запросе объекты.

Промежуточное представление QUIL служит для сведения большого числа операторов LINQ к шести основным:

- `Src` представляет исходную коллекцию объектов.
- `Trans` выполняет поэлементное преобразование последовательности, параметризуется функцией преобразования.
- `Pred` выполняет фильтрацию последовательности, параметризуется функцией-предикатом.
- `Sink` выполняет материализацию последовательности во временной коллекции в памяти, параметризуется функциями `() → IEnumerable<U>` для инициализации коллекции и `IEnumerable<U> × T → IEnumerable<U>` для обновления коллекции.
- `Agg` выполняет агрегацию коллекции в скалярное значение, параметризуется функциями `() → U` для инициализации скалярного значения и `U × T → U` для обновления скалярного значения.

- `Ret` означает конец выполнения запроса.

Для запроса без подзапросов, корректные программы на QUIL распознаются конечным автоматов на рис. 8.

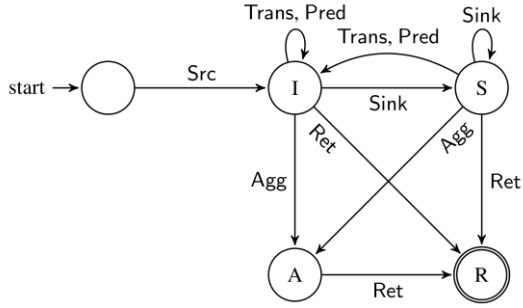


Рис. 8. Конечный автомат операторов QUIL в Steno
Fig. 8. Finite state machine of QUIL operators in Steno

Генерация кода в Steno также следует этому автомату. Каждый переход на рис. 8 связан с определённой процедурой генерации кода — зависящей, таким образом, и от оператора, и от состояния — при этом в каждом состоянии поддерживаются три точки вставки (см. рис. 9):

- пролог цикла (loop prelude),
- тело цикла (loop body),
- эпилог цикла (loop postlude).

Наличие подзапросов — вложенных пар операторов `Src`–`Ret` — делает язык QUIL контекстно-зависимым, поэтому к конечном автомату на рис. 8 добавляется стек (делая его автоматом с магазинной памятью). В стек добавляются тройки точек вставки (пролог, тело цикла, эпилог), соответствующие одному уровню вложенности запроса.

Интереса заслуживает переход из состояния *I* (Iterating) в состояние *R* (Returning) по оператору `Ret`. На самом внешнем уровне вложенности в этом случае генерируется оператор `yield return` языка C#, что является встроенным в язык способом определения итераторов. Если же этот переход является частью вложенного подзапроса, то вместо этого две тройки $(\alpha_i, \mu_i, \omega_i)$ и $(\alpha_{i+1}, \mu_{i+1}, \omega_{i+1})$ на вершине стека заменяются на совмещённую тройку $(\alpha_i, \mu_{i+1}, \omega_i)$ — таким образом происходит совмещение внешнего и вложенного циклов.

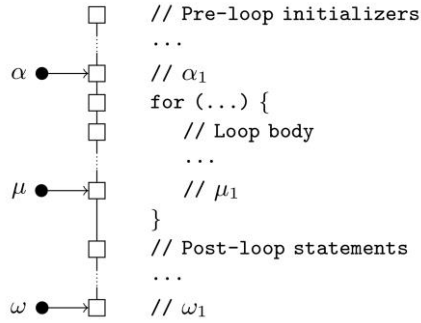


Рис. 9.Точки вставки кода при компиляции запроса в Steno
Fig. 9. Code insertion points in Steno

Метод имеет константные, но довольно большие накладные расходы, связанные с использованием механизмов динамической загрузки кода и рефлексии, которые могут быть сокращены при помощи кеширования загруженных объектов CLR, соответствующих скомпилированным LINQ-запросам, для повторного использования. Накладные расходы также могут быть сокращены при помощи статической компиляции, в частности за счёт совмещения с этапом кодогенерации DryadLINQ [39].

В статье также рассматривается вопрос распределения вычислений на несколько вычислительных узлов и интеграции с DryadLINQ. Метод основан на выделении последовательностей гомоморфных операторов — операторов, которые могут быть применены к элементам последовательности независимо. Гомоморфными операторами QUIL являются операторы `Trans`, `Pred`, а также вложенные запросы (`Src` и `Ret`).

На нескольких синтетических тестах Steno позволяет получить ускорение от 3.32 до 14.1 раз по сравнению со встроенной в .NET реализацией LINQ для массивов в памяти, при этом накладные расходы по сравнению с императивным кодом, оптимизированным вручную, составляют до 53%, но в среднем не превышают 3%. При тестировании с DryadLINQ исследователи получили ускорение до 1.9 раз, но отмечают существенную зависимость от особенностей задачи.

3.4 HIQUE (2010)

В HIQUE [43] рассматривается динамическая компиляция SQL-запросов на основе шаблонов кода.

Алгоритмы, используемые в традиционных СУБД, разработаны в первую очередь для оптимизации использования подсистемы ввода-вывода, что в современных условиях и при современных объёмах доступной памяти недостаточно. В случаях, когда вся база данных или значительная её часть

умещается в оперативной памяти, узким местом производительности является процессор. Исследователи отмечают, что изменение подсистемы хранения для упрощения обработки данных — одно из предложенных решений — слишком радикально меняет архитектуру существующих СУБД. Более ортогональным решением является компиляция запросов в машинный код, которая позволяет значительно сократить накладные расходы на исполнение плана запроса по сравнению с интерпретаторами, реализующими Volcano-модель и основанными на абстракции итератора.

Предлагаемый в статье подход *целостного выполнения запросов* (*holistic query evaluation*) основан на динамической компиляции плана выполнения запроса в программу на некотором языке программирования на основе шаблонов кода, разработанных для каждого оператора. Полученная программа оптимизируется как одно целое, при этом применяются как платформо-зависимые, так и межоператорные оптимизации, недоступные в интерпретаторе из-за существующих границ абстракции между различными операторами. Динамическая компиляция позволяет убрать эти границы абстракции, уменьшить число вызовов функций и увеличить локальность данных, в том числе за счёт более эффективного использования регистров процессора.

HIQUE использует построчное хранение кортежей (N-ary Storage Model). Архитектура подсистемы обработки запросов представлена на рис. 10.

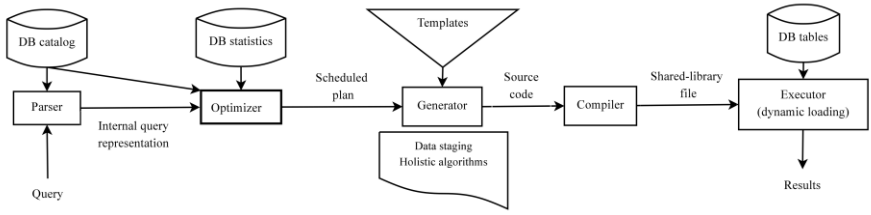


Рис. 10. Архитектура обработки запросов в HIQUE
Fig. 10. Query processing architecture of HIQUE

Результатом работы оптимизатора является топологически отсортированная последовательность операторов, в которой на вход оператор o_i принимает или выход оператора o_j, где $j < i$, или первичную таблицу в базе данных. Компиляция каждого оператора состоит из этапа предобработки, который заключается в определении используемых атрибутов и вычислении смещений, применении предикатов сканирования и вставки дополнительных операторов сортировки и секционирования, где необходимо, и следующего за ним этапа генерации кода, который заключается в инстанциировании соответствующего шаблона кода — в статье приведены примеры шаблонов для некоторых операторов. Промежуточные результаты работы каждого оператора сохраняются во временные таблицы: между каждой парой операторов неявно

вставляется точка материализации.

Результатом компиляции является функция на языке C, которая затем транслируется в объектный код и подгружается в основную программу для выполнения.

Экспериментальная оценка производительности на бенчмарке TPC-H показывает прирост производительности от 4 раз (в сравнении с колоночной СУБД MonetDB [44]) до 167 раз (в сравнении с PostgreSQL).

3.5 HyPer (2011)

HyPer [33, 45] — СУБД в основной памяти, основанная на модели явных циклов и динамической компиляции запросов в машинный код с использованием инфраструктуры LLVM.

Исследователи утверждают, что с ростом объёмов основной памяти производительность СУБД больше определяется эффективностью использования процессора и классическая Volcano-модель, несмотря на свою простоту и гибкость, не позволяет использовать процессор достаточно эффективно из-за нелокального доступа к памяти и ошибок в прогнозировании переходов.

Предлагаемый и реализованный в HyPer подход состоит в определении *границ конвейеризации* (*pipeline boundaries*) — операторов плана выполнения запроса, которые приводят к материализации кортежей, см. рис. 11 — и компиляции операторов в пределах границ конвейеризации в один цикл обработки данных (см. рис. 12), оставляя таким образом границы между операторами только там, где необходимо: материализация кортежей происходит только на границах между циклами, которые определяются планом выполнения запроса и используемыми в нём алгоритмами обработки данных, а на каждой итерации одного цикла происходит применение нескольких операторов к одному кортежу, что позволяет максимально эффективно использовать для хранения атрибутов кортежа регистры процессора.

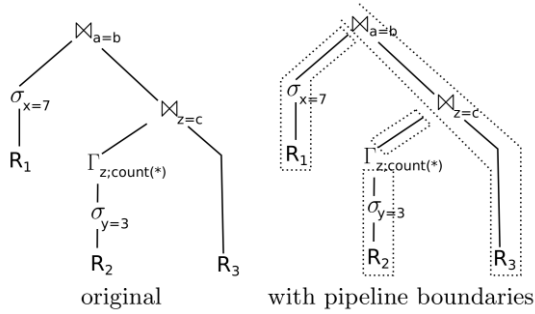


Рис. 11. Определение границ конвейеризации в HyPer
Fig. 11. Pipeline boundaries in HyPer

```
initialize memory of ⋈_{a=b}, ⋈_{c=z}, and Γ_z
for each tuple t in R_1
    if t.x = 7
        materialize t in hash table of ⋈_{a=b}
for each tuple t in R_2
    if t.y = 3
        aggregate t in hash table of Γ_z
for each tuple t in Γ_z
    materialize t in hash table of ⋈_{z=c}
for each tuple t_3 in R_3
    for each match t_2 in ⋈_{z=c}[t_3.c]
        for each match t_1 in ⋈_{a=b}[t_3.b]
            output t_1 ∘ t_2 ∘ t_3
```

Рис. 12. Результат компиляции границ конвейеризации в HyPer (псевдокод)
Fig. 12. Pipeline boundaries compiled to pseudocode

Генерация кода выполняется при помощи методов `produce` и `consume`, сопоставленных операторам плана выполнения запроса. Метод `produce` определён для всех операторов и служит для построения кода, состоящего из одного или нескольких (в количестве листовых вершин) циклов и реализующего функциональность соответствующего поддерева плана выполнения запроса. `produce` рекурсивно вызывается для потомков внутренних операторов плана выполнения запроса. Когда выполнение доходит до листового оператора, генерируется заголовок цикла и вызывается метод `consume` родительского оператора, который служит для генерации кода, обрабатывающего приём очередного кортежа от дочернего оператора. Методы `consume` определены только для внутренних операторов плана в количестве, равном общему числу дуг в дереве плана выполнения запроса. Генерация кода для всего запроса производится посредством вызова метода `produce` на самом внешнем операторе плана. Вызовы `produce` и `consume`, таким образом, производят обход дерева плана в глубину, по завершении которого самый внешний вызов `produce` возвращает код, реализующий в модели явных циклов весь исходный запрос.

Интерфейс, представленный функциями `produce` и `consume`, подобен интерфейсу итераторов в Volcano-модели и упрощает независимую разработку реляционных операторов, но существует только во время компиляции запроса — результат компиляции не содержит вызовов этих функций и представлен всего несколькими императивными циклами (см. рис. 12).

Для генерации машинного кода в HyPer используется инфраструктура LLVM [11] и промежуточное представление LLVM IR. Исследователи отмечают следующие преимущества над альтернативными технологиями:

- приемлемое время компиляции (по сравнению с языками высокого уровня);

- достаточную низкоуровневость и контроль над результирующим машинным кодом (например, в части использования флагов арифметического переполнения);
- наличие неограниченного множества виртуальных регистров;
- платформо-независимость;
- типобезопасность;
- наличие большого числа встроенных оптимизаций.

Для максимизации продуктивности, упрощения поддержки и уменьшения времени компиляции существенная часть внутренней логики операторов реализована на C++ и вызывается из сгенерированного кода. Тем не менее, для обеспечения максимальной производительности необходимо, чтобы наиболее горячие участки кода, в особенности части, ответственные за обработку кортежей в циклах в пределах границ конвейеризации, были реализованы на LLVM IR во избежание накладных расходов на вызовы функций на C++, в частности на сохранение и загрузку регистров согласно используемому соглашению о вызовах.

В [33] исследователи отмечают некоторые особенности генерации кода для оптимизации производительности, в частности:

- загрузку атрибутов из памяти немного раньше, чем необходимо, для сокрытия задержек работы с памятью;
- использование циклов с постусловием вместо циклов с предусловием для улучшения предсказания переходов.

В статье также описывается обработка в цикле нескольких кортежей одновременно с использованием SIMD-инструкций, доступных в современных процессорах.

[45] дополнительно описывает представление SQL-значений и операций в коде на LLVM IR, при этом как правило одному SQL-значению соответствует несколько LLVM-значений (например, число или указатель и флаг *null* или длина строки) и одной операции — несколько элементарных операций LLVM (например, сложение чисел и проверка переполнения); абстракцию времени компиляции для генерации графа потока управления (условий и циклов) и абстракцию времени компиляции для представления кортежей (в сжатом или дематериализованном виде). Использование абстракций времени компиляции существенно упрощает разработку и поддержку динамического компилятора и при этом не приводит к дополнительным затратам времени выполнения.

На синтетических запросах динамическая компиляция в HyPer позволяет получить ускорение до 2–8 раз в сравнении с интерпретацией в зависимости от запроса, в некоторых случаях до 3000 раз, а на бенчмарке TPC-H — до 3.7 раз в сравнении с колоночной СУБД VectorWise [46].

3.6 Hekaton (2013)

Hekaton [47, 48] — это расширение SQL Server [49], включающее в себя

таблицы и индексы для данных в основной памяти, неблокирующие структуры данных для эффективного многопоточного выполнения и MVCC и компилятор запросов.

Исследователи отмечают, что производительность СУБД при выполнении OLTP-запросов зависит от трёх основных параметров: количества выполняемых инструкций, количества циклов на инструкцию и коэффициента масштабируемости — причём последние два параметра суммарно могут дать только прирост производительности в 3–4 раза. Для ускорения СУБД в 10–100 раз необходимо, таким образом, существенно сократить количество выполняемых инструкций.

Компилятор запросов Hekaton принимает на вход структуры результат работы оптимизатора и генерирует код на промежуточном представлении PIT (Pure Imperative Tree), на основе которого после серии преобразований генерируется код на языке C, компилируемый и загружаемый в процесс SQL Server для выполнения.

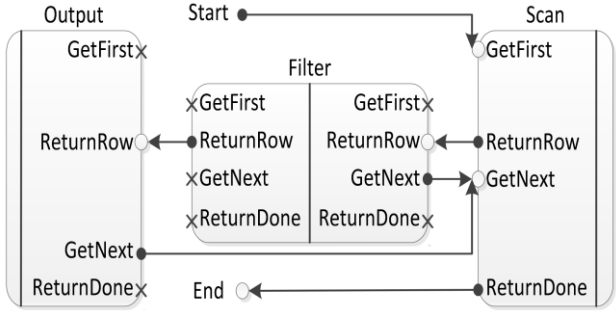


Рис. 13. Межоператорный поток управления в Hekaton
Fig. 13. Interoperator control flow in Hekaton

Операторы в Hekaton реализуют интерфейс, состоящий из функций `GetFirst`, `GetNext`, `ReturnRow` и `ReturnDone`, что позволяет комбинировать операторы произвольным образом согласно плану выполнения запроса.

В Hekaton реализована модель явных циклов при помощи объединения кода операторов в одну функцию и трансляции переходов между ними — вызовов функций `GetFirst`, `GetNext` и т. д. — в безусловные переходы между соответствующими операторам базовыми блоками (см. рис. 13).

Результирующий код (рис. 14) из соображений безопасности не содержит идентификаторов исходного запроса, что ещё больше затрудняет экспертный анализ, но проведённое исследование показало, что генерация кода в одну функцию позволяет минимизировать как число выполняемых инструкций, так и размер бинарного кода.

Сравнение с интерпретатором запросов показало сокращение количества выполняемых инструкций до 10–15 раз.

```
    /*Seek*/
l_17:; /*seek.GetFirst*/
    hr = (HkCursorHashGetFirst(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->Transaction),
        0, 0, 1,
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
l_20:; /*seek.1*/
    if ((hr == 0))
    {
        goto l_14 /*filter.child.ReturnRow*/ ;
    }
    else
    {
        goto l_12 /*query.ReturnDone*/ ;
    }
l_21:; /*seek.GetNext*/
    hr = (HkCursorHashGetNext(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->ErrorObject),
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
    goto l_20 /*seek.1*/ ;
    /*Filter*/
l_14:; /*filter.child.ReturnRow*/
    result_22 = ((rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ->hkc_1 /*[Id]*/ ) ==
        ((long)((valueArray[1 /*@id*/ ]).SignedIntData)));
    if (result_22)
    {
        goto l_13 /*output.child.ReturnRow*/ ;
    }
    else
    {
        goto l_21 /*seek.GetNext*/ ;
    }
```

Рис. 14. Пример результирующего кода в Hekaton
Fig. 14. Compiled code in Hekaton

3.7 MemSQL (2016)

MemSQL [50] — это СУБД в основной памяти, в которой реализована компиляция запросов в машинный код при помощи инфраструктуры LLVM.
Исследователи выделяют три направления оптимизации производительности программных систем:

- оптимизация подсистемы ввода–вывода: планировка (scheduling), перемещение данных, распределение нагрузки;
- оптимизация потребления памяти вычислительных узлов;
- оптимизация использования ЦПУ для вычислений — в первую очередь сокращение количества исполняемых инструкций.

В дисковых СУБД оптимизация ввода–вывода имеет гораздо более важное значение, потому что ввод–вывод в дисковых СУБД является самым «узким местом». В СУБД в основной памяти, напротив, оптимизации памяти и

вычислений выходят на первый план и компиляция запросов в эффективный машинный код может дать существенный прирост производительности.

В MemSQL для компиляции используются высокоуровневое промежуточное представление MPL (MemSQL Plan Language) и промежуточное представление среднего уровня MBC (MemSQL Bit Code), которое затем транслируется в низкоуровневое промежуточное представление LLVM IR.

Исследователи отмечают, что компиляция запросов даёт как количественные, так и качественные преимущества для конечного пользователя: например, возможность использования систем визуализации и мониторинга реального времени.

3.8 Компиляция запросов в компиляторе, разрабатываемом в ИСП РАН (2016)

В ИСП РАН разрабатывается расширение [51, 52] к СУБД PostgreSQL, реализующее динамическую компиляцию запросов на основе инфраструктуры LLVM.

Модель итераторов, используемая в PostgreSQL, сопряжена с существенными накладными расходами, связанными с неявными вызовами функций `next` и сопутствующими ошибками предсказания переходов и необходимостью сохранения состояния операторов между вызовами функций `next`.

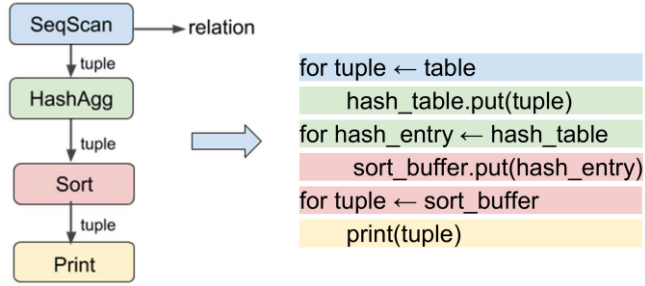


Рис. 15. Переход к модели явных циклов
Fig. 15. Push-based execution model

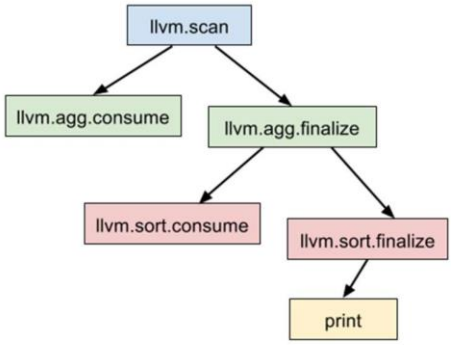
В статье описывается переход от модели итераторов к модели явных циклов (см. рис. 15) и алгоритм генерации кода в этой модели (см. рис. 16), основанный на сопоставленных каждому оператору функциях `consume` и `finalize`. Во время генерации кода совершается обход плана выполнения запроса, при котором каждая вершина 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`. Процесс генерации начинается с передачи генератору самого внешнего оператора плана выполнения запроса функции `consume`, осуществляющей передачу очередного кортежа результата, и пустой функции `finalize`.

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

Рис. 16. Генерация кода в модели явных циклов
Fig. 16. Code generation for push-based execution model

На рис. 17 показан результат компиляции. После встраивания получается код как на рис. 15 справа.




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

Рис. 17. Результирующий код в модели явных циклов
Fig. 17. Compiled code in push-based model

На бенчмарке TPC-H метод позволяет получить ускорение до 5.5 раз.

4. Методы, основанные на специализации кода

4.1 ToasterBooster (2013)

ToasterBooster [53] — это расширение DBToaster [37] для динамической компиляции запросов при помощи технологии многоуровневой компиляции, реализованной во фреймворке LMS (Lightweight Modular Staging) [54] для языка Scala.

Многоуровневая компиляция в LMS реализуется посредством

параметризованных типов Scala, LMS предоставляет параметризованный тип `Rep`. Если значения типа `T` относятся к уровню компиляции `N`, то значения типа `Rep[T]` относятся к уровню компиляции `N+1`. Например, отличие между типами `Rep[T] -> Rep[T]` и `Rep[T -> T]` состоит в том, что значениями первого типа являются функции на множестве объектов промежуточного представления, в то время как значениями второго типа являются функции *на языке* промежуточного представления. Во время выполнения операций над значениями типа `Rep[T]` осуществляется генерация и оптимизация кода на языке промежуточного представления, реализующего соответствующие операции над значениями типа `T`.

Явное разделение уровней выполнения позволяет провести автоматическую специализацию кода и данных следующего уровня к коду и данным текущего уровня: встраивание функций, разворачивание циклов, подстановка константных значений.

Архитектура системы представлена на рис. 18. После оптимизации на языках промежуточного представления M3 и K3 запрос попадает в генератор кода на языке Scala, в котором проводятся дополнительные оптимизации на языке промежуточного представления LMS и генерируется код на Scala или C++, реализующий разностные функции исходного запроса.

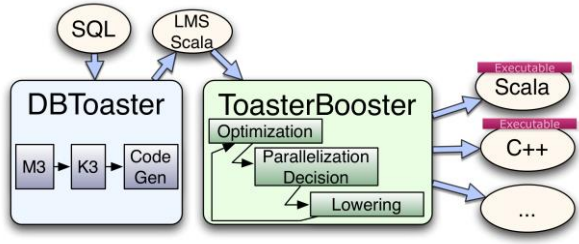


Рис. 18. Архитектура ToasterBooster
Fig. 18. ToasterBooster architecture

В статье также рассматривается реализация оптимизации дефорестации в LMS, которая заключается в удалении границ абстракции между коллекциями, используемыми в генерируемом коде, при помощи генераторов, предоставляющих интерфейс коллекции времени компиляции (что выражается в замене типов `Rep[Collection[T]]` типами `Generator[T]`) и позволяющих совместить различные операции над коллекцией и получить на выходе эффективный императивный код.

Экспериментальная оценка на синтетических запросах показывает ускорение до 5 раз относительно компилятора запросов, встроенного в DBToaster.

4.2 LegoBase (2014)

LegoBase [55] — компилятор запросов, реализованный на языке Scala.

Для оптимизации производительности в нём используется технология многоуровневой компиляции и фреймворк многоуровневой компиляции LMS [54].

В статье рассматриваются оптимизации, реализованные на уровне промежуточного представления LMS специально для LegoBase, такие как:

- оптимизация структур данных — частичное вычисление адресов и параметров обращений, замена ассоциативных структур данных (хеш-таблиц) линейными (массивами);
- изменение модели выполнение с Volcano-модели (pull-based) на push-based — исследователи отмечают, что реализация автоматического изменения модели выполнения оказалась возможна только благодаря достаточно высокому уровню абстракции реализации операторов на языке Scala;
- удаление лишней материализации кортежей на границе операторов;
- изменение схемы расположения данных таблиц: замена построчного хранения поколоночным.

В результате компиляции и оптимизации LMS генерирует код на языке Scala. Для достижения максимальной производительности в LegoBase реализована дополнительная трансляция результирующего кода в код на языке C. Исследователи отмечают, что трансляция в C потребовала решения двух проблем:

- Трансляции вызовов библиотечных функций и структур данных. Для решения этой проблемы в LegoBase используется библиотека GLib [56].
- Трансляции модели управления памятью. В языке Scala используется сборщик мусора, поэтому во время трансляции кода со Scala в C необходимо обеспечить своевременное освобождение неиспользуемой памяти во избежание утечек памяти. Для решения этой проблемы в LegoBase используется ручное управление памятью.

Полученный код на языке C компилируется при помощи Clang [31] и исполняется.

В статье также рассматривается возможность перекомпиляции кода во время выполнения при изменении параметров СУБД. Например, при изменении параметра, отвечающего за логирование этапов выполнения запроса, требуется перекомпилировать функции, в которых производится проверка этого параметра, с целью добавления или удаления соответствующих вызовов. Подход, используемый в LegoBase, позволяет реализовывать компиляторы языков запросов на языке высокого уровня без потерь в производительности. Многоуровневая компиляция позволяет автоматически удалять из результирующего кода на промежуточном представлении используемые при разработке абстракции. Исследователи отмечают следующие преимущества по сравнению с другими методами компиляции запросов:

- типобезопасность (обеспечивается использованием высокоуровневого языка);
- относительная простота реализации и поддержки;
- поддержка межоператорных оптимизаций (по сравнению с методами, основанными на шаблонах);
- поддержка высокоуровневых оптимизаций (обеспечивается более высоким уровнем промежуточного представления по сравнению с, например, промежуточным представлением LLVM).

Эксперименты с использованием бенчмарка TPC-H показывают, что LegoBase достигает большей производительности, чем СУБД HyPer [33], в которой для компиляции запросов используется LLVM [11]. Сравнение с HyPer указывает на упущенные оптимизационные возможности, вызванные слишком низким уровнем абстракции, предоставляемым LLVM IR, и реализованные при помощи использования более высокоуровневого промежуточного представления LMS.

4.3 DexterDB (2015)

В [57] представлен метод компиляции запросов на основе специализации исходного кода СУБД на уровне промежуточного представления LLVM, реализованный для DexterDB [58].

Исследователи выделяют два основных подхода к вычислению запросов в СУБД: классический подход с использованием интерпретатора запросов и динамическая компиляция на основе шаблонов. Проблемой классического подхода является производительность, а проблемой второго — сложность разработки, расширения и поддержки. Подход на основе специализации кода представляет собой расширение классического подхода и совмещает и простоту разработки, свойственную интерпретаторам запросов, и высокую степень специализации и оптимизации, свойственную динамическим компиляторам.

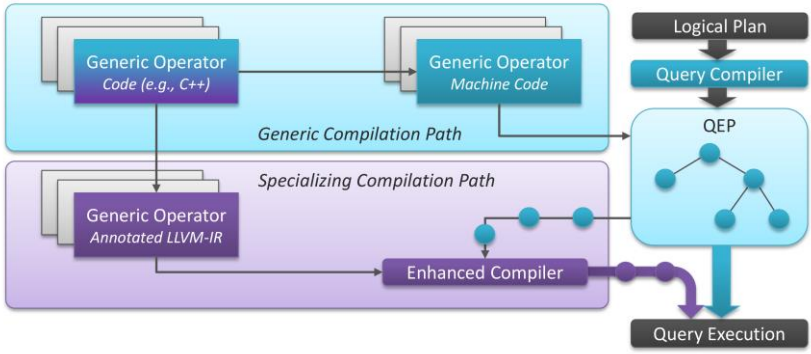


Рис. 19. Специализация запросов в DexterDB

Fig. 19. Generic and specialized compilation paths in DexterDB

В предлагаемом подходе операторы плана выполнения реализуются на высокоуровневом языке и могут быть использованы как непосредственно (см. *Generic Compilation Path* на рис. 19) — этот случай эквивалентен классическому подходу — так и с использованием специализатора. В последнем случае (*Specializing Compilation Path*) исходный код СУБД, являющийся обобщённой реализацией оператора, предварительно компилируется в LLVM IR (см. рис. 20) и во время выполнения автоматически специализируется к параметрам оператора, соответствующим конкретному запросу.

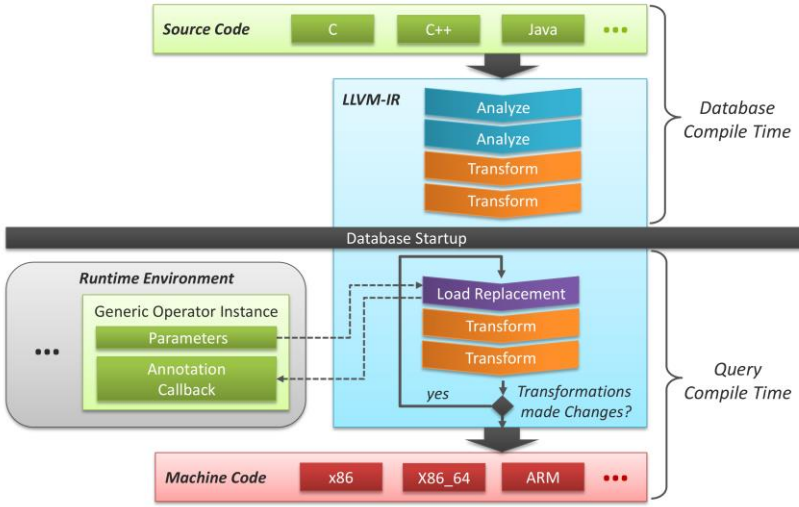


Рис. 20. Компиляция кода операторов в DexterDB
Fig. 20. Compilation pipeline in DexterDB

Специализация основана на частичном вычислении инструкций загрузки констант из памяти (Load Replacement), которое выполняется в цикле вместе со встроенными оптимизациями LLVM — в статье приведён перечень и последовательность применения встроенных оптимизаций — до достижения неподвижной точки.

Load replacement состоит в замене инструкций загрузки из памяти (`load`), соответствующих параметрам оператора, непосредственными значениями этих параметров. Особенности реализации операторов в DexterDB позволяют сократить поиск потенциальных инструкций для замены: в DexterDB каждый оператор моделируется классом, а его параметры — полями соответствующих объектов. Таким образом, доступ к параметрам оператора всегда происходит по смещениям относительно указателя `this` на объект класса, который в методах оператора является неявным параметром, — что в LLVM IR

соответствует определённым цепочкам инструкций `load`, `bitcast` и `getelementptr`. На каждой итерации во время специализации обнаруживаются все такие цепочки с известным смещением относительно `this`, соответствующие указателям на константные параметры, и заменяются на значения соответствующих параметров — значения по соответствующим указателям.

Определение того, соответствует ли адрес относительно значения `this` константному параметру, происходит в DexterDB также во время выполнения посредством вызова метода `bool memoryIsConstant(void *mem)`, который с байтовой точностью определяет множество виртуальных адресов констант в памяти процесса СУБД.

Оценка производительности на бенчмарке SSB [59] показывает ускорение в 1.2–1.6 раза по сравнению с классической схемой (*Generic Compilation Path*).

4.4 LB2-Spatial (2016)

LB2-Spatial [60] — это СУБД с поддержкой геопространственных данных на основе СУБД LB2 [61], которая, в свою очередь, является форком LegoBase [55].

Поддержка геопространственных данных требует специализированных вычислительно сложных алгоритмов и пространственных индексов, таких как R-деревья, k-d деревья и quad-деревья. В LB2-Spatial эти алгоритмы реализованы с использованием фреймворка многоуровневой компиляции LMS [54] для Scala, что позволяет для каждого запроса автоматически генерировать специализированные версии этих алгоритмов.

В статье также затрагивается вопрос расширения СУБД PostgreSQL и Spark / SparkSQL компиляцией геопространственных функций, а также распределения вычислений на несколько узлов.

На тестовом запросе LB2-Spatial позволяет получить ускорение до 14 раз относительно геопространственного расширения PostGIS [62] к PostgreSQL.

4.5 Flare (2017)

Flare [63] — это компилятор запросов для Apache Spark, совместимый на уровне интерфейса со Spark SQL [14], от встроенного компилятора которого он отличается в следующем:

- Во Flare запросы компилируются не в байткод JVM, а в машинный код, что позволяет избежать связанных с JVM накладных расходов.
- Во Flare запросы компилируются целиком, а не на уровне отдельных операторов или групп операторов, как в Spark SQL, что позволяет повысить эффективность использования вычислительных узлов за счёт сокращения накладных расходов на поддержку абстракции и передачи данных между узлами. Исследователи отмечают, что на практике вертикальное масштабирование в пределах нескольких

сильных узлов позволяет решить задачи проще и дешевле, чем горизонтальное масштабирование кластера на много слабых узлов.

- Flare поддерживает компиляцию и встраивание пользовательских функций на нескольких предметно-ориентированных языках, непрозрачных для оптимизатора и компилятора Spark SQL.

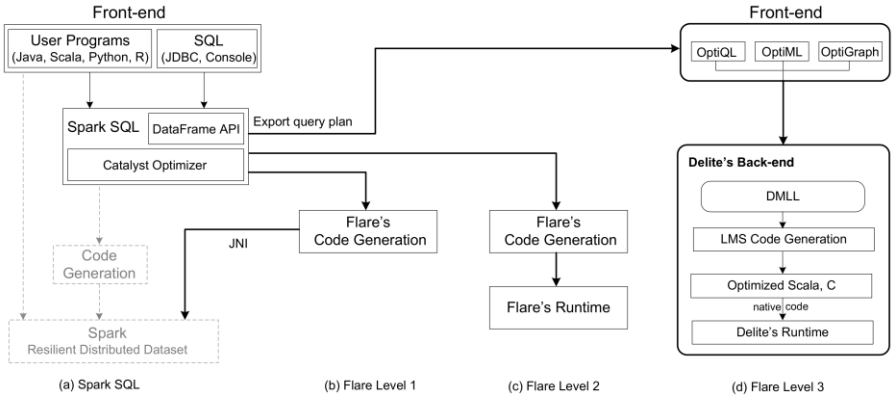


Рис. 21. Архитектура Flare
Fig. 21. Flare architecture

Архитектура Flare (рис. 21) состоит из нескольких уровней. Первый уровень заменяет генератор байткода JVM для операторов запроса на генератор, реализованный с использованием фреймворка LMS [54], промежуточное представление которого транслируется в C, компилируется в машинный код и подгружается в адресное пространство процесса Spark. Для вызова к среде выполнения Spark из результирующего кода используются механизм JNI (Java Native Interface).

Второй уровень содержит компилятор запросов в модели явных циклов и заменяет части среды выполнения Spark. Выполнение в несколько потоков поддерживается при помощи OpenMP [64]. Выполнение на нескольких вычислительных узлах не поддерживается и требует расширения Spark механизмами перехвата управления, которые позволили бы обеспечить запуск и координацию Flare на нескольких узлах. Flare также компилирует код загрузки и дематериализации данных таблиц с использованием информации о схеме таблиц и используемого алгоритма кодирования.

Третий уровень добавляет поддержку пользовательских функций на предметно-ориентированных языках OptiQL [65], OptiML [66], OptiGraph [65], OptiMesh [65], реализованных на Scala, что позволяет комбинировать различные парадигмы программирования в одном запросе. Для этого используется фреймворк Delite [67] и внутреннее представление DMLL [68], в которое генерируется и план выполнения, полученный от встроенного в Spark SQL оптимизатора запросов Catalyst, и

пользовательские функции, используемые в запросе.
Экспериментальная оценка на бенчмарке TPC-H показывает время работы, сравнимое с HyPer [33] и ускорение относительно Spark SQL до 89 раз.

5. Заключение

В статье рассмотрены работы в области динамической компиляции запросов, как для распределённых [1], так и для нераспределённых систем (см. табл. 1).
Можно заметить, что во всех рассмотренных динамических компиляторах запросов для распределённых систем, за исключением Flare и DryadLINQ, и некоторых компиляторах для PostgreSQL компиляция реализуется на уровне выражений и горячих функций. Здесь показателен подход Greenplum, основанный на замене указателей в структурах данных, изначально указывающих на обобщённый код, во время подготовки запроса к выполнению, и немного более общий подход микроспециализации и горячей замены [27], основанный на динамическом изменении указателей в коде программы в зависимости от значений определённых переменных.
Компилятор запросов Flare разработан для распределённой системы Spark, но поддерживает выполнение одного запроса строго на одном узле, поэтому принципиально не отличается от других рассмотренных компиляторов запросов для нераспределённых систем (в частности LegoBase).
DryadLINQ является единственным из рассмотренных компиляторов запросов, который транслирует один запрос в код для нескольких вычислительных узлов. Для этого после серии статических оптимизизаций запрос разделяется на части (LINQ-подвыражения), каждая из которых компилируется в промежуточное представление .NET и отправляется на множество узлов, которые во время выполнения будут отвечать за соответствующую часть запроса. Интересно, что само отображение подвыражений на вычислительные узлы в DryadLINQ также может меняться во время выполнения.
Можно выделить два основных подхода к реализации компиляторов запросов: реализация генератора на основе модели явных циклов (JamDB, HyPer, Hekaton, MemSQL, Steno, компилятор ИСП РАН) и реализация двухуровневого компилятора на основе фреймворка LMS (LegoBase, Flare, ToasterBooster, LB2-Spatial).
Преобразование в модель явных циклов (push-модель) явно сформулировано в [33]. Оно состоит в генерации по плану выполнения запроса императивного кода со вложенными циклами, в котором одна из листовых вершин плана выполнения (сканирование таблицы или индекса) является самым внешним циклом, а из самого внутреннего цикла вызывается код, отвечающий за приём кортежей самой внешней (корневой) вершиной плана. В разных компиляторах можно найти разные способы реализации этой модели (HyPer, Hekaton, Steno, компилятор ИСП РАН).
В HIQUE реализована более простая модель, основанная на генерации кода

операторов в топологическом порядке с материализацией во временные таблицы на каждом шаге. Тем не менее, даже такая модель позволила получить существенное ускорение относительно коммерческих СУБД.

Компиляторы, основанные на специализации кода, к разновидности которой можно отнести многоуровневую компиляцию в том виде, в котором она реализована в LMS, позволяют существенно упростить разработку и поддержку компиляторов за счёт того, что код ядра компилятора является по сути кодом обобщённого интерпретатора плана выполнения, при этом накладные расходы, связанные с интерпретацией, сокращаются используемым при компиляции фреймворком (LegoBase, Flare, ToasterBooster, LB2-Spatial) или набором оптимизаций (DexterDB). Можно отметить некоторое сходство этих подходов недавней линии исследований в области реализации виртуальных машин [69, 70].

Табл. 1. Сводная таблица компиляторов запросов
Table 1. Summary of just-in-time query compilers

Система обработки данных	**Распределённая**	**Компилятор запросов**	**Единица компиляции**
Impala	да	[7, 8]	Выражения
Greenplum	да	[23]	Выражения
Spark	да	Spark SQL [14]	Выражения
		Flare [60]	Запрос (LMS)
Dryad	да	DryadLINQ [38]	Распределённый запрос
PostgreSQL	нет	Микроспециализация [25–27]	Выражения
		Butterstein, Grust [30]	Выражения
		Компилятор ИСП РАН [6, 49, 50]	Запрос (push-модель)
JamDB	нет	[34]	Запрос (push-модель)
HyPer	нет	[33, 43]	Запрос (push-модель)
SQL Server	нет	Hekaton [45, 46]	Запрос (push-модель)
MemSQL	нет	[48]	Запрос

Система обработки данных	Распределённая	Компилятор запросов	Единица компиляции
			(push-модель)
LINQ-совместимые / .NET	—	Steno [41]	Запрос (push-модель)
HIQUE	нет	[42]	Запрос (шаблоны)
DBToaster	нет	[36]	Разностные запросы
		ToasterBooster [51]	Запрос (LMS)
DBX[1]	нет	LegoBase [53]	Запрос (LMS)
LB2	нет	LB2-Spatial [58]	Запрос (LMS)
DexterDB	нет	[55]	Запрос (специализация)

Список литературы

[1]. Кузнецов, С. Основы современных баз данных. http://citforum.ru/database/osbd/contents.shtml (дата обращения 18.05.2017).
[2]. Chamberlin, D.D., Astrahan, M.M., et al. 1981. A history and evaluation of System R. *Commun. ACM*. 24, 10 (1981), 632–646.
[3]. Wade, B.W. 2012. Compiling SQL into System/370 machine language. *IEEE Annals of the History of Computing*. 34, 4 (2012), 49–50.
[4]. Greer, R. 1999. Daytona and the fourth-generation language Cymbal. *SIGMOD 1999, proceedings ACM SIGMOD international conference on management of data* (Philadelphia, Pennsylvania, USA, 1999), 525–526.
[5]. Copeland, G.P., Khoshafian, S. 1985. A decomposition storage model. *Proceedings of the 1985 ACM SIGMOD international conference on management of data* (Austin, Texas, USA, 1985), 268–279.
[6]. Шарыгин, Е., Бучацкий, Р., Скворцов, Л., Жуйков, Р., Мельник, Д. 2016. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL. *Труды ИСП РАН*. 28, 4 (2016), 217–240.
[7]. Kornacker, M., Behm, A., et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. *CIDR 2015, seventh biennial conference on innovative data systems research* (Asilomar, CA, USA, 2015).
[8]. Wanderman-Milne, S., Li, N. 2014. Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.
[9]. Apache Hadoop, open-source software for reliable, scalable, distributed computing. The

[1] Неназванная коммерческая СУБД

Apache Software Foundation; http://hadoop.apache.org (дата обращения 19.06.2017).
[10]. Apache HBase, the Hadoop database, a distributed, scalable, big data store. The Apache Software Foundation; https://hbase.apache.org (дата обращения 19.06.2017).
[11]. Lattner, C., Adve, V.S. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM international symposium on code generation and optimization (CGO 2004)* (San Jose, CA, USA, 2004), 75–88.
[12]. TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpch (дата обращения 25.05.2017).
[13]. Apache Spark, a fast and general engine for large-scale data processing. The Apache Software Foundation; https://spark.apache.org (дата обращения 19.06.2017).
[14]. Armbrust, M., Xin, R.S., et al. 2015. Spark SQL: Relational data processing in Spark. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1383–1394.
[15]. PostgreSQL, an open source object-relational database system. The PostgreSQL Global Development Group; https://www.postgresql.org (дата обращения 16.06.2017).
[16]. PostgreSQL derived databases. PostgreSQL wiki; https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases (дата обращения 20.06.2017).
[17]. ToroDB Stampede, a database bridging NoSQL and SQL. 8Kdata; https://www.torodb.com (дата обращения 19.06.2017).
[18]. Vertica, a “shared nothing” distributed analytical database. Hewlett Packard Enterprise Development; https://www.vertica.com (дата обращения 19.06.2017).
[19]. AgensGraph, a highly optimized, multi-model graph database for the modern, complex connected data environment. Bitnine Global; http://www.agensgraph.com (дата обращения 19.06.2017).
[20]. Tan, C. 2015. Vitesse DB: 100% Postgres, 100X faster for analytics. Presented at the 2nd South Bay PostgreSQL Meetup; https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub (дата обращения 20.06.2017).
[21]. ParAccel 2010. *The ParAccel analytic database: A technical overview*. ParAccel, Inc. https://marketplace.informatica.com/mpresources/docs/ParAccel-Technical-Overview-White-Paper%202011.pdf (дата обращения 20.06.2017).
[22]. Gupta, A., Agarwal, D., et al. 2015. Amazon Redshift and the case for simpler data warehouses. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1917–1923.
[23]. Armenatzoglou, N., Rajaraman, K.J., et al. 2016. Improving query execution speed via code generation. *Pivotal Engineering Journal; http://engineering.pivotal.io/post/codegen-gpdb-qx* (дата обращения 20.06.2017). (2016).
[24]. DeepgreenDB, a scalable MPP data warehouse solution derived from the open source Greenplum database project. Vitesse Data; http://vitessedata.com/deepgreen-db (дата обращения 19.06.2017).
[25]. Zhang, R., Debray, S., Snodgrass, R.T. 2012. Micro-specialization: dynamic code specialization of database management systems. *10th annual IEEE/ACM international symposium on code generation and optimization, CGO 2012* (San Jose, CA, USA, 2012), 63–73.
[26]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Micro-specialization in DBMSes. *IEEE 28th international conference on data engineering (ICDE 2012)* (Washington, DC, USA (Arlington, Virginia), 2012), 690–701.

[27]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Application of micro-specialization to query evaluation operators. *Workshops proceedings of the IEEE 28th international conference on data engineering, ICDE 2012* (Arlington, VA, USA, 2012), 315–321.
[28]. *Callgrind: A call-graph generating cache and branch prediction profiler*. Valgrind Developers; http://valgrind.org/docs/manual/cl-manual.html (дата обращения 8.06.2017).
[29]. TPC-C, an on-line transaction processing benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpcc (дата обращения 8.06.2017).
[30]. Butterstein, D., Grust, T. 2016. Precision performance surgery for PostgreSQL: LLVM-based expression compilation, just in time. *PVLDB*. 9, 13 (2016), 1517–1520.
[31]. Clang: A C language family frontend for LLVM. The LLVM Foundation; https://clang.llvm.org/ (дата обращения 1.06.2017).
[32]. Graefe, G. 1994. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.
[33]. Neumann, T. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[34]. Rao, J., Pirahesh, H., Mohan, C., Lohman, G. 2006. Compiled query execution engine using jVM. *Proceedings of the 22Nd international conference on data engineering* (Washington, DC, USA, 2006), 23.
[35]. Java Emitter Templates, part of Eclipse Modeling Framework. Eclipse Foundation; http://www.eclipse.org/modeling/m2t/?project=jet (дата обращения 7.06.2017).
[36]. DB2, a relational database. IBM Corporation;
https://www.ibm.com/analytics/us/en/technology/db2 (дата обращения 21.06.2017).
[37]. Ahmad, Y., Koch, C. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*. 2, 2 (2009), 1566–1569.
[38]. Box, D., Hejlsberg, A. 2007. LINQ: .NET language-integrated query. *Microsoft Developer Network; https://msdn.microsoft.com/en-us/library/bb308959.aspx (дата обращения 8.06.2017)*. (2007).
[39]. Yu, Y., Isard, M., et al. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *8th USENIX symposium on operating systems design and implementation, OSDI 2008, proceedings* (San Diego, California, USA, 2008), 1–14.
[40]. Dryad data-parallel processing framework. Microsoft; https://www.microsoft.com/en-us/research/project/dryad (дата обращения 19.06.2017).
[41]. Duffy, J. 2007. A query language for data parallel programming: Invited talk. *Proceedings of the 2007 workshop on declarative aspects of multicore programming* (New York, NY, USA, 2007), 50.
[42]. Murray, D.G., Isard, M., Yu, Y. 2011. Steno: Automatic optimization of declarative queries. *Proceedings of the 32Nd aCM sIGPLAN conference on programming language design and implementation* (New York, NY, USA, 2011), 121–131.
[43]. Krikellas, K., Viglas, S., Cintra, M. 2010. Generating code for holistic query evaluation. *Proceedings of the 26th international conference on data engineering, ICDE 2010* (Long Beach, California, USA, 2010), 613–624.
[44]. MonetDB, an open source column-oriented database. MonetDB B.V. https://www.monetdb.org (дата обращения 21.06.2017).
[45]. Neumann, T., Leis, V. 2014. Compiling database queries into machine code. *IEEE Data Eng. Bull.* 37, 1 (2014), 3–11.
[46]. Actian Vector (former VectorWise), a relational vectorized columnar analytic database. Actian Corporation; https://www.actian.com/analytic-database/vector-smp-analytic-database

(дата обращения 19.06.2017).
[47]. Diaconu, C., Freedman, C., et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013* (New York, NY, USA, 2013), 1243–1254.
[48]. Freedman, C., Ismert, E., Larson, P. 2014. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull*. 37, 1 (2014), 22–30.
[49]. SQLServer, a relational database. Microsoft; https://www.microsoft.com/en-us/sql-server (дата обращения 19.06.2017).
[50]. Paroski, D. 2016. Code generation: The inner sanctum of database performance. *High Scalability; http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html* (дата обращения 19.06.2017). (2016).
[51]. Бучацкий, Р., Шарыгин, Е., Скворцов, Л., Жуйков, Р., Мельник, Д., Баев, Р. 2016. Динамическая компиляция SQL-запросов для СУБД PostgreSQL. *Труды ИСП РАН*. 28, 6 (2016), 37–48.
[52]. Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E. 2017. JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017; https://www.pgcon.org/2017/schedule/attachments/467_PGCon%202017-05-26%2015-00%20ISPRAS%20Dynamic%20Compilation%20of%20SQL%20Queries%20in%20PostgreSQL%20Using%20LLVM%20JIT.pdf (дата обращения 19.06.2017).
[53]. Dashti, M., Abadi, R. 2013. Database query optimization using compilation techniques. (2013).
[54]. Rompf, T., Odersky, M. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Generative programming and component engineering, proceedings of the ninth international conference on generative programming and component engineering, GPCE 2010* (Eindhoven, The Netherlands, 2010), 127–136.
[55]. Klonatos, Y., Koch, C., Rompf, T., Chafi, H. 2014. Building efficient query engines in a high-level language. *PVLDB*. 7, 10 (2014), 853–864.
[56]. GLib, a general-purpose utility library. The GNOME Foundation; https://developer.gnome.org/glib/ (дата обращения 2.06.2017).
[57]. Hänsch, C., Kissinger, T., Habich, D., Lehner, W. 2015. Plan operator specialization using reflective compiler techniques. *Datenbanksysteme für business, technologie und web (BTW), 16. Fachtagung des GI-fachbereichs «datenbanken und informationssysteme» (DBIS), 4.-6.3.2015, proceedings* (Hamburg, Germany, 2015), 363–382.
[58]. Dexter: Dresden index for transactional access on emerging technologies. Dresden Database Systems Group; http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/ (дата обращения 17.01.2013).
[59]. O'Neil, P., O'Neil, B., Chen, X. 2009. Star Schema Benchmark. (2009).
[60]. Tahboub, R.Y., Rompf, T. 2016. On supporting compilation in spatial query engines: (Vision paper). *Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS 2016* (Burlingame, California, USA, 2016), 9:1–9:4.
[61]. Rompf, T. LB2, a fork of LegoBase. https://github.com/TiarkRompf/legobase-micro (дата обращения 16.06.2017).
[62]. PostGIS, a spatial database extender for PostgreSQL. PostGIS Project Steering Committee; http://postgis.net (дата обращения 21.06.2017).
[63]. Essertel, G.M., Tahboub, R.Y., Decker, J.M., Brown, K.J., Olukotun, K., Rompf, T. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *CoRR*. abs/1703.08219, (2017).

[64]. OpenMP, an API specification for parallel programming. OpenMP Architecture Review Board; http://www.openmp.org (дата обращения 19.06.2017).
[65]. Sujeeth, A.K., Rompf, T., et al. 2013. Composition and reuse with compiled domain-specific languages. *ECOOP 2013 - object-oriented programming - 27th european conference, proceedings* (Montpellier, France, 2013), 52–78.
[66]. Sujeeth, A.K., Lee, H., et al. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. *Proceedings of the 28th international conference on machine learning, ICML 2011* (Bellevue, Washington, USA, 2011), 609–616.
[67]. Brown, K.J., Sujeeth, A.K., et al. 2011. A heterogeneous parallel framework for domain-specific languages. *2011 international conference on parallel architectures and compilation techniques, PACT 2011* (Galveston, TX, USA, 2011), 89–100.
[68]. Brown, K.J., Lee, H., et al. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. *Proceedings of the 2016 international symposium on code generation and optimization, CGO 2016* (Barcelona, Spain, 2016), 194–205.
[69]. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th symposium on dynamic languages, DLS '12* (Tucson, AZ, USA, 2012), 73–82.
[70]. Würthinger, T., Wimmer, C., et al. 2013. One VM to rule them all. *ACM symposium on new ideas in programming and reflections on software, onward! 2013, part of SPLASH '13* (Indianapolis, IN, USA, 2013), 187–204.

Survey of Just-in-Time Query Compilation Methods

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

Abstract. Data processing systems have been traditionally optimized for I/O, mainly because, until pretty recently, disk storage has been the most affordable type of storage and the most prevalent one. This is not necessarily the case today, particularly in the world of big data analytics. As the problems posed by data analytics become more commonplace, efficient CPU utilization becomes the new bottleneck. Just-in-time query compilation is a promising solution to this challenge that is currently being applied both in academic studies and across the industry. This paper is a survey of just-in-time query compilation methods sampled from the literature available on the subject. All methods are broadly categorized into expression compilation and hotspot methods, whole-query compilation methods, and specialization-based methods. A number of query processors are identified within confines of each category, various methods, architectures, and significant results are described. Finally, we conclude with an overview of most general approaches to query compilation that we identified.

Keywords: just-in-time compilation; query engines; query languages; expression compilation; hotspot compilation; holistic compilation; push-model; code specialization.

DOI: 10.15514/ISPRAS-2017-29(3)-11

For citation: Sharygin E. Y., Buchatskiy R. A. Survey of Just-in-Time Query Compilation Methods. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue. 3, 2017, pp. 179-224 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-11

References

[1]. Kuztetsov, S. Foundations of Modern Database Systems. http://citforum.ru/database/osbd/contents.shtml, accessed 18.05.2017 (in Russian).
[2]. Chamberlin, D.D., Astrahan, M.M., et al. 1981. A history and evaluation of System R. *Commun. ACM*. 24, 10 (1981), 632–646.
[3]. Wade, B.W. 2012. Compiling SQL into System/370 machine language. *IEEE Annals of the History of Computing*. 34, 4 (2012), 49–50.
[4]. Greer, R. 1999. Daytona and the fourth-generation language Cymbal. *SIGMOD 1999, proceedings ACM SIGMOD international conference on management of data* (Philadelphia, Pennsylvania, USA, 1999), 525–526.
[5]. Copeland, G.P., Khoshafian, S. 1985. A decomposition storage model. *Proceedings of the 1985 ACM SIGMOD international conference on management of data* (Austin, Texas, USA, 1985), 268–279.
[6]. 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
[7]. Kornacker, M., Behm, A., et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. *CIDR 2015, seventh biennial conference on innovative data systems research* (Asilomar, CA, USA, 2015).
[8]. Wanderman-Milne, S., Li, N. 2014. Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull*. 37, 1 (2014), 31–37.
[9]. Apache Hadoop, open-source software for reliable, scalable, distributed computing. The Apache Software Foundation; http://hadoop.apache.org (accessed 19.06.2017).
[10]. Apache HBase, the Hadoop database, a distributed, scalable, big data store. The Apache Software Foundation; https://hbase.apache.org (accessed 19.06.2017).
[11]. Lattner, C., Adve, V.S. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM international symposium on code generation and optimization (CGO 2004)* (San Jose, CA, USA, 2004), 75–88.
[12]. TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpch (accessed 25.05.2017).
[13]. Apache Spark, a fast and general engine for large-scale data processing. The Apache Software Foundation; https://spark.apache.org (accessed 19.06.2017).
[14]. Armbrust, M., Xin, R.S., et al. 2015. Spark SQL: Relational data processing in Spark. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1383–1394.
[15]. PostgreSQL, an open source object-relational database system. The PostgreSQL Global Development Group; https://www.postgresql.org (accessed 16.06.2017).
[16]. PostgreSQL derived databases. PostgreSQL wiki; https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases (accessed 20.06.2017).
[17]. ToroDB Stampede, a database bridging NoSQL and SQL. 8Kdata; https://www.torodb.com (accessed 19.06.2017).

[18]. Vertica, a “shared nothing” distributed analytical database. Hewlett Packard Enterprise Development; https://www.vertica.com (accessed 19.06.2017).
[19]. AgensGraph, a highly optimized, multi-model graph database for the modern, complex connected data environment. Bitnine Global; http://www.agensgraph.com (accessed 19.06.2017).
[20]. Tan, C. 2015. Vitesse DB: 100% Postgres, 100X faster for analytics. Presented at the 2nd South Bay PostgreSQL Meetup; https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub (accessed 20.06.2017).
[21]. ParAccel 2010. *The ParAccel analytic database: A technical overview*. ParAccel, Inc. https://marketplace.informatica.com/mpresources/docs/ParAccel-Technical-Overview-White-Paper%202011.pdf (accessed 20.06.2017).
[22]. Gupta, A., Agarwal, D., et al. 2015. Amazon Redshift and the case for simpler data warehouses. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1917–1923.
[23]. Armenatzoglou, N., Rajaraman, K.J., et al. 2016. Improving query execution speed via code generation. *Pivotal Engineering Journal; http://engineering.pivotal.io/post/codegen-gpdb-qx* (accessed 20.06.2017). (2016).
[24]. DeepgreenDB, a scalable MPP data warehouse solution derived from the open source Greenplum database project. Vitesse Data; http://vitessedata.com/deepgreen-db (accessed 19.06.2017).
[25]. Zhang, R., Debray, S., Snodgrass, R.T. 2012. Micro-specialization: dynamic code specialization of database management systems. *10th annual IEEE/ACM international symposium on code generation and optimization, CGO 2012* (San Jose, CA, USA, 2012), 63–73.
[26]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Micro-specialization in DBMSes. *IEEE 28th international conference on data engineering (ICDE 2012)* (Washington, DC, USA (Arlington, Virginia), 2012), 690–701.
[27]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Application of micro-specialization to query evaluation operators. *Workshops proceedings of the IEEE 28th international conference on data engineering, ICDE 2012* (Arlington, VA, USA, 2012), 315–321.
[28]. *Callgrind: A call-graph generating cache and branch prediction profiler*. Valgrind Developers; http://valgrind.org/docs/manual/cl-manual.html (accessed 8.06.2017).
[29]. TPC-C, an on-line transaction processing benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpcc (accessed 8.06.2017).
[30]. Butterstein, D., Grust, T. 2016. Precision performance surgery for PostgreSQL: LLVM-based expression compilation, just in time. *PVLDB*. 9, 13 (2016), 1517–1520.
[31]. Clang: A C language family frontend for LLVM. The LLVM Foundation; https://clang.llvm.org/ (accessed 1.06.2017).
[32]. Graefe, G. 1994. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.
[33]. Neumann, T. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[34]. Rao, J., Pirahesh, H., Mohan, C., Lohman, G. 2006. Compiled query execution engine using jVM. *Proceedings of the 22Nd international conference on data engineering* (Washington, DC, USA, 2006), 23.
[35]. Java Emitter Templates, part of Eclipse Modeling Framework. Eclipse Foundation; http://www.eclipse.org/modeling/m2t/?project=jet (accessed 7.06.2017).

[36]. DB2, a relational database. IBM Corporation; https://www.ibm.com/analytics/us/en/technology/db2 (accessed 21.06.2017).
[37]. Ahmad, Y., Koch, C. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*. 2, 2 (2009), 1566–1569.
[38]. Box, D., Hejlsberg, A. 2007. LINQ: .NET language-integrated query. *Microsoft Developer Network; https://msdn.microsoft.com/en-us/library/bb308959.aspx* (accessed 8.06.2017). (2007).
[38]. Yu, Y., Isard, M., et al. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *8th USENIX symposium on operating systems design and implementation, OSDI 2008, proceedings* (San Diego, California, USA, 2008), 1–14.
[40]. Dryad data-parallel processing framework. Microsoft; https://www.microsoft.com/en-us/research/project/dryad (accessed 19.06.2017).
[41]. Duffy, J. 2007. A query language for data parallel programming: Invited talk. *Proceedings of the 2007 workshop on declarative aspects of multicore programming* (New York, NY, USA, 2007), 50.
[42]. Murray, D.G., Isard, M., Yu, Y. 2011. Steno: Automatic optimization of declarative queries. *Proceedings of the 32Nd aCM sIGPLAN conference on programming language design and implementation* (New York, NY, USA, 2011), 121–131.
[43]. Krikellas, K., Viglas, S., Cintra, M. 2010. Generating code for holistic query evaluation. *Proceedings of the 26th international conference on data engineering, ICDE 2010* (Long Beach, California, USA, 2010), 613–624.
[44]. MonetDB, an open source column-oriented database. MonetDB B.V. https://www.monetdb.org (accessed 21.06.2017).
[45]. Neumann, T., Leis, V. 2014. Compiling database queries into machine code. *IEEE Data Eng. Bull.* 37, 1 (2014), 3–11.
[46]. Actian Vector (former VectorWise), a relational vectorized columnar analytic database. Actian Corporation; https://www.actian.com/analytic-database/vector-smp-analytic-database (accessed 19.06.2017).
[47]. Diaconu, C., Freedman, C., et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013* (New York, NY, USA, 2013), 1243–1254.
[48]. Freedman, C., Ismert, E., Larson, P. 2014. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.
[49]. SQLServer, a relational database. Microsoft; https://www.microsoft.com/en-us/sql-server (accessed 19.06.2017).
[50]. Paroski, D. 2016. Code generation: The inner sanctum of database performance. *High Scalability; http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html* (accessed 19.06.2017). (2016).
[51]. 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.
[52]. Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E. 2017. JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017; https://www.pgcon.org/2017/schedule/attachments/467_PGCon%202017-05-26%2015-00%20ISPRAS%20Dynamic%20Compilation%20of%20SQL%20Queries%20in%20PostgreSQL%20Using%20LLVM%20JIT.pdf (accessed 19.06.2017).
[53]. Dashti, M., Abadi, R. 2013. Database query optimization using compilation techniques.

(2013).
[54]. Rompf, T., Odersky, M. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Generative programming and component engineering, proceedings of the ninth international conference on generative programming and component engineering, GPCE 2010* (Eindhoven, The Netherlands, 2010), 127–136.
[55]. Klonatos, Y., Koch, C., Rompf, T., Chafi, H. 2014. Building efficient query engines in a high-level language. *PVLDB*. 7, 10 (2014), 853–864.
[56]. GLib, a general-purpose utility library. The GNOME Foundation; https://developer.gnome.org/glib/ (accessed 2.06.2017).
[57]. Hänsch, C., Kissinger, T., Habich, D., Lehner, W. 2015. Plan operator specialization using reflective compiler techniques. *Datenbanksysteme für business, technologie und web (BTW), 16. Fachtagung des GI-fachbereichs «datenbanken und informationssysteme» (DBIS), 4.-6.3.2015, proceedings* (Hamburg, Germany, 2015), 363–382.
[58]. Dexter: Dresden index for transactional access on emerging technologies. Dresden Database Systems Group; http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/ (accessed 17.01.2013).
[59]. O'Neil, P., O'Neil, B., Chen, X. 2009. Star Schema Benchmark. (2009).
[60]. Tahboub, R.Y., Rompf, T. 2016. On supporting compilation in spatial query engines: (Vision paper). *Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS 2016* (Burlingame, California, USA, 2016), 9:1–9:4.
[61]. Rompf, T. LB2, a fork of LegoBase. https://github.com/TiarkRompf/legobase-micro (accessed 16.06.2017).
[62]. PostGIS, a spatial database extender for PostgreSQL. PostGIS Project Steering Committee; http://postgis.net (accessed 21.06.2017).
[63]. Essertel, G.M., Tahboub, R.Y., Decker, J.M., Brown, K.J., Olukotun, K., Rompf, T. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *CoRR*. abs/1703.08219, (2017).
[64]. OpenMP, an API specification for parallel programming. OpenMP Architecture Review Board; http://www.openmp.org (accessed 19.06.2017).
[65]. Sujeeth, A.K., Rompf, T., et al. 2013. Composition and reuse with compiled domain-specific languages. *ECOOP 2013 - object-oriented programming - 27th european conference, proceedings* (Montpellier, France, 2013), 52–78.
[66]. Sujeeth, A.K., Lee, H., et al. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. *Proceedings of the 28th international conference on machine learning, ICML 2011* (Bellevue, Washington, USA, 2011), 609–616.
[67]. Brown, K.J., Sujeeth, A.K., et al. 2011. A heterogeneous parallel framework for domain-specific languages. *2011 international conference on parallel architectures and compilation techniques, PACT 2011* (Galveston, TX, USA, 2011), 89–100.
[68]. Brown, K.J., Lee, H., et al. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. *Proceedings of the 2016 international symposium on code generation and optimization, CGO 2016* (Barcelona, Spain, 2016), 194–205.
[69]. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th symposium on dynamic languages, DLS '12* (Tucson, AZ, USA, 2012), 73–82.
[70]. Würthinger, T., Wimmer, C., et al. 2013. One VM to rule them all. *ACM symposium on new ideas in programming and reflections on software, onward! 2013, part of SPLASH '13*

(Indianapolis, IN, USA, 2013), 187–204.