ИСП РАН
Институт системного программирования
Российской академии наук

Ускорение выполнения SQL-запросов в СУБД PostgreSQL с использованием динамической компиляции

Д.М. Мельник, Р.А. Жуйков

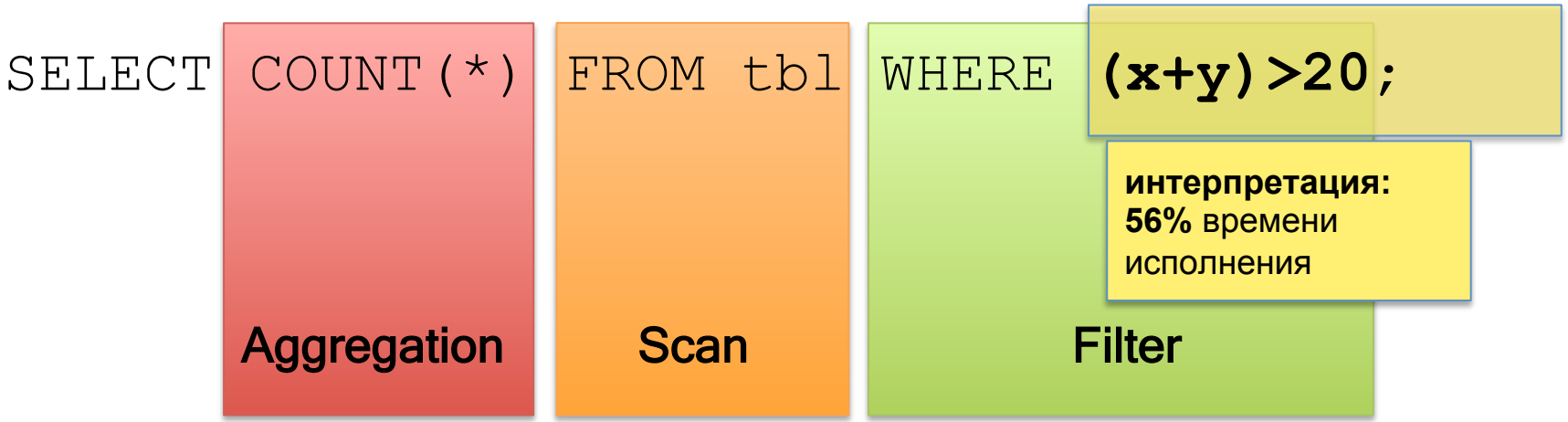
ИСП РАН

30.11.2016

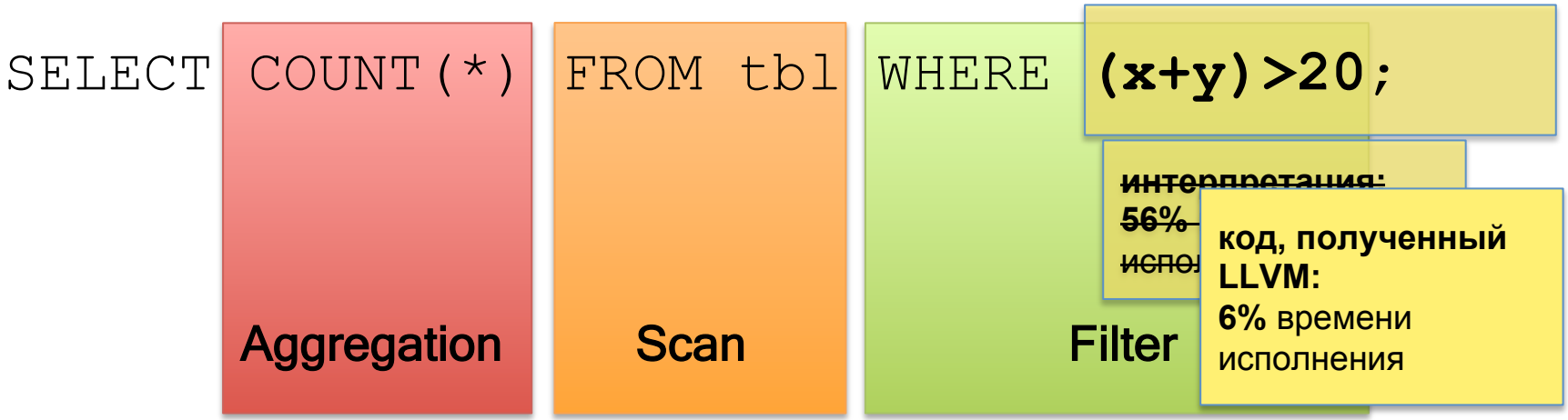
PostgreSQL

- Свободно распространяемая объектно-реляционная система управления базами данных (СУБД).
- Производительность и надёжность.
- Поддержка Unix-подобных и Windows систем.
- Расширяемость (процедурные языки, типы данных, функции, индексы); богатая экосистема пользовательских расширений http://pgxn.org.
- Активное сообщество.
- Разрешительная BSD-like лицензия.
- 20+ лет разработки.

Пример оптимизации запроса



Пример оптимизации запроса



=> Ускорение выполнения запроса в 2 раза

Цель работы

- Увеличение производительности PostgreSQL на вычислительно сложных SQL-запросах.
- Что именно мы хотим ускорить?
  - Сложные запросы, узким местом в производительности которых является процессор, а не дисковые операции (OLAP).
  - Оптимизация производительности на наборе тестов TPC-H.
- Как ускорить?
  - Динамически компилировать запросы в машинный код.

Profiling TPC-H

ISP RAS

TPC-H Q1:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
FROM
    lineitem
WHERE
    l_shipdate <=
    date '1998-12-01' -
    interval '90' day
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22
ExecQual	6%	14%	32%	3%	72%
ExecAgg	75%	-	1%	1%	2%
SeqNext	6%	1%	33%	-	13%
IndexNext	-	57%	-	-	19%
BitmapHeapNext	-	-	-	85%	-

Related Work

- Neumann T., Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, Vol. 4, No. 9, 2011.
- Butterstein D., Grust T., Precision Performance Surgery for PostgreSQL – LLVM-based Expression Compilation, Just in Time. VLDB 2016.
  - Динамическая компиляция выражений для PostgreSQL.
  - Ускорение до **37%** на TPC-H.
- Vitesse DB:
  - Проприетарная СУБД на основе PostgreSQL.
  - Ускорение до **8** раз на TPC-H Q1.

LLVM

- Компиляторная инфраструктура для компиляции и оптимизации программ.
- Платформо-независимое внутреннее представление LLVM IR.
- Широкий набор оптимизаций и инструментов для анализа и трансформации программ.
- Поддержка множества платформ: x86, x86_64, ARM, MIPS…
- Встроенные средства для JIT-компиляции (MCJIT, ORC JIT).
- Разрешительная BSD-like лицензия.

Где используется LLVM JIT

ISP RAS

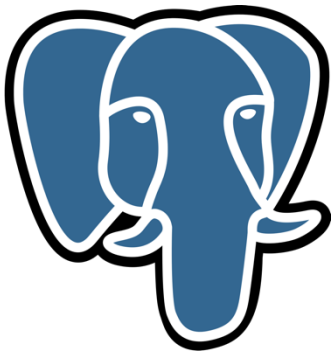
- Pyston (Python, Dropbox)
- HHVM (PHP & Hack, Facebook)
- LLILC (MSIL, .NET Foundation)
- Julia (Julia, community)

- JavaScript:
  - JavaScriptCore in WebKit (Apple) – Fourth Tier LLVM JIT (FTL JIT), недавно заменен на B3
  - LLV8 – LLVM добавлен в Google V8 в качестве дополнительного уровня JIT (ИСП РАН => open source)

- СУБД:
  - MemSQL, Impala
  - … и теперь PostgreSQL

Что если добавить LLVM JIT в PostgreSQL?

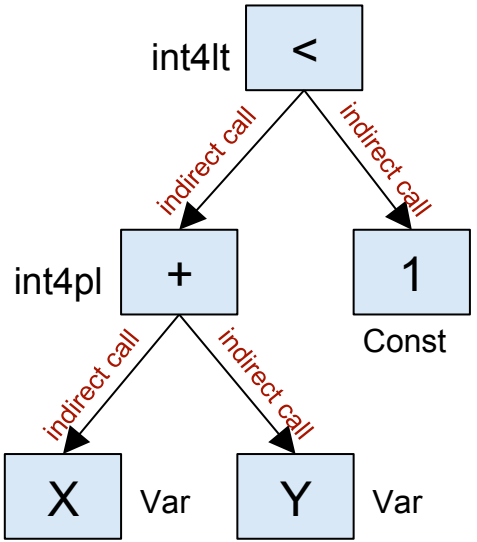
ISP RAS



=

Динамическая компиляция выражений

X+Y < 1

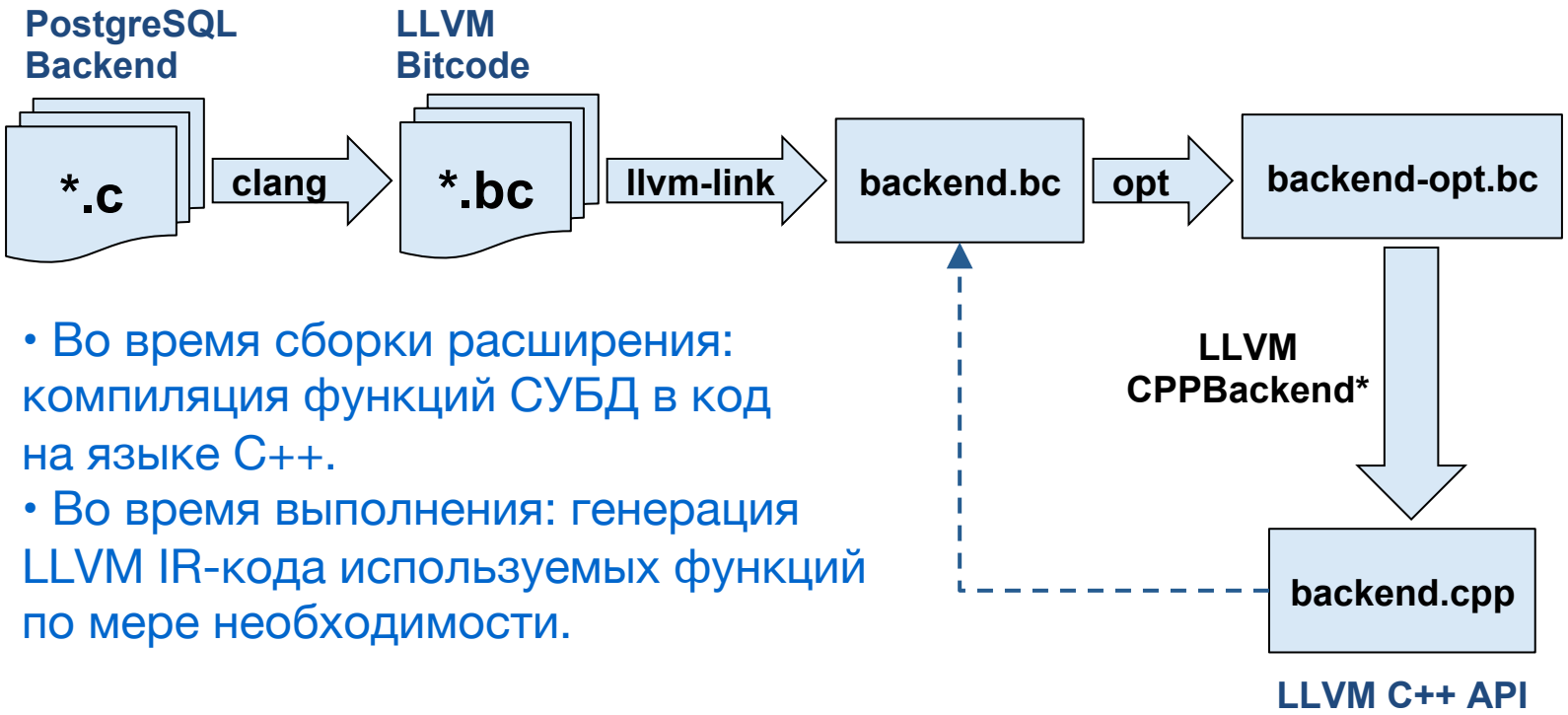



```
define i1 @ExecQual() {
  %x = load &X.attr
  %y = load &Y.attr

  %pl = add %x, %y
  %lt = icmp lt %pl, 1

  ret %lt
}
```

Метод предкомпиляции функций СУБД



- Во время сборки расширения: компиляция функций СУБД в код на языке C++.
- Во время выполнения: генерация LLVM IR-кода используемых функций по мере необходимости.

* потребовалось обновление до версии LLVM 3.7

Предкомпиляция: пример (1)

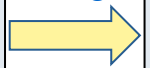
```c
Datum
int8pl(FunctionCallInfo fcinfo)
{
  int64       arg1 = fcinfo->arg[0];
  int64       arg2 = fcinfo->arg[1];
  int64       result;

  result = arg1 + arg2;

  /*
   * Overflow check.
   */
  if (SAMESIGN(arg1, arg2) && !SAMESIGN(result, arg1))
      ereport(ERROR,
         (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),
          errmsg("bigint out of range")));
  PG_RETURN_INT64(result);
}
```

PostgreSQL int8.c

Clang



```
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
  %1 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %4 = load i64, i64* %3
  %5 = add nsw i64 %4, %2
  %.lobit = lshr i64 %2, 63
  %.lobit1 = lshr i64 %4, 63
  %6 = icmp ne i64 %.lobit, %.lobit1
  %.lobit2 = lshr i64 %5, 31
  %7 = icmp eq i64 %.lobit2, %.lobit
  %or.cond = or i1 %6, %7
  br i1 %or.cond, label %ret, label %overflow

overflow:
  call void @ereport(...)
ret:
  ret i64 %5
}
```

LLVM IR int8.bc

Предкомпиляция: пример (2)

```
Function* define_int8pl(Module *mod) {
  Function* func_int8pl = Function::Create("int8pl");

  // Block (entry)
  ptr_1 = GetElementPtrInst::Create(fcinfo, 0);
  int64_2 = new LoadInst(ptr_1);
  ptr_3 = GetElementPtrInst::Create(fcinfo, 1);
  int64_4 = new LoadInst(ptr_3);
  int64_5 = BinaryOperator::Create(Add, int64_2, int64_4);
  lobit = BinaryOperator::Create(LShr, int64_2, 63);
  lobit1 = BinaryOperator::Create(LShr, int64_4, 63);
  int1_6 = new ICmpInst(ICMP_NE, lobit, lobit1);
  lobit2 = BinaryOperator::Create(LShr, int64_5, 63);
  int1_7 = new ICmpInst(ICMP_EQ, lobit2, lobit);
  int1_or_cond = BinaryOperator::Create(Or, int1_6, int1_7);
  BranchInst::Create(ret, overflow, int1_or_cond);

  // Block (overflow)
  CallInst::Create(func_erreport);

  // Block (ret)
  ReturnInst::Create(mod->getContext(), int64_5, ret);

  return func_int8pl;
}
```

LLVM API

int8.cpp

CPPBackend

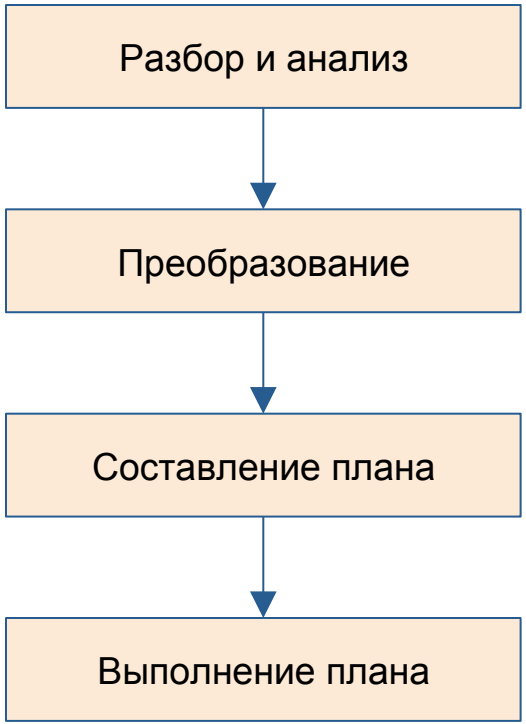
```
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
  %1 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %4 = load i64, i64* %3
  %5 = add nsw i64 %4, %2
  %.lobit = lshr i64 %2, 63
  %.lobit1 = lshr i64 %4, 63
  %6 = icmp ne i64 %.lobit, %.lobit1
  %.lobit2 = lshr i64 %5, 31
  %7 = icmp eq i64 %.lobit2, %.lobit
  %or.cond = or i1 %6, %7
  br i1 %or.cond, label %ret, label %overflow

overflow:
  call void @ereport(...)
ret:
  ret i64 %5
}
```

LLVM IR

int8.bc

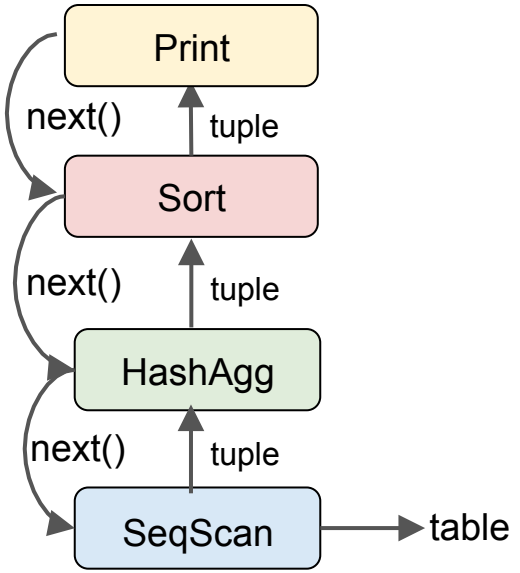
Обработка запросов в PostgreSQL



- Лексический и синтаксический анализ и построение дерева разбора.
- Семантический анализ, необходимый для вычисления используемых таблиц, функций и операторов.
- Составление и оптимизация плана выполнения запроса.
- Интерпретация плана выполнения.

Выполнение плана: модель Volcano

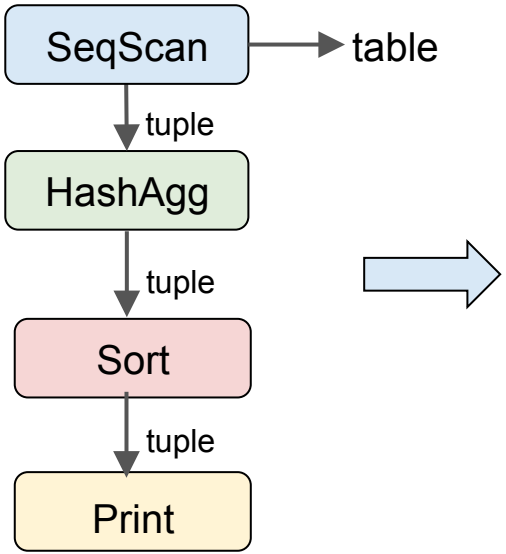
- Graefe G., Volcano— An Extensible and Parallel Query Evaluation System. IEEE TKDE 6 (1), 120-135, 1994.
- Каждый оператор представляется последовательностью кортежей, доступ к элементам которой осуществляется посредством вызова метода next().
- Неявный вызов функции: branch misprediction, невозможен inlining.
- Необходимость сохранения состояния между вызовами next().




```
select a, sum(b) from tbl
group by a order by a;
```

Выполнение плана: push-based модель (1)

- Выполнение запроса управляется одним из листовых узлов.
- Запрос представляется в виде нескольких циклов.




```
for tuple ← table
    hash_table.put(tuple)
for hash_entry ← hash_table
    sort_buffer.put(hash_entry)
for tuple ← sort_buffer
    print(tuple)
```

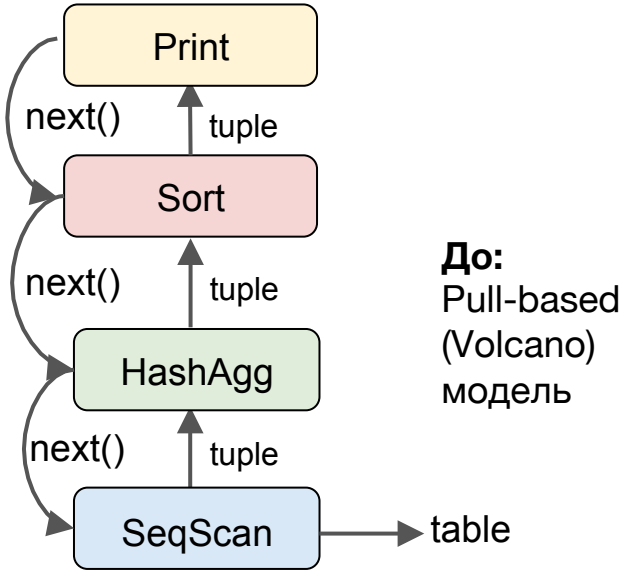
Изменение модели выполнения

```
select <columns>                    from <table> where <condition>
```

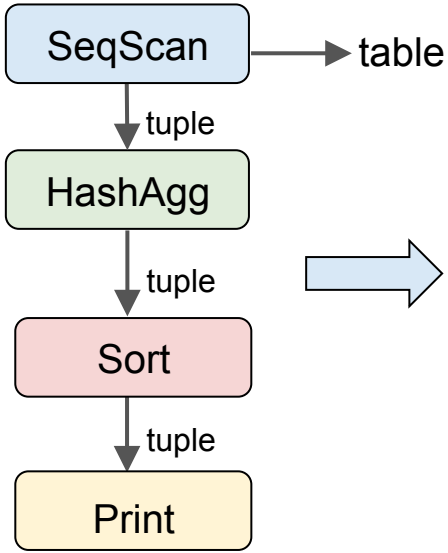
HashAgg | **SeqScan** **Filter**

```
group by <column>
order by <column>;
```

Sort



До:
Pull-based (Volcano) модель




```
for tuple ← table
        hash_table.put(tuple)
for hash_entry ← hash_table
        sort_buffer.put(hash_entry)
for tuple ← sort_buffer
        print(tuple)
```

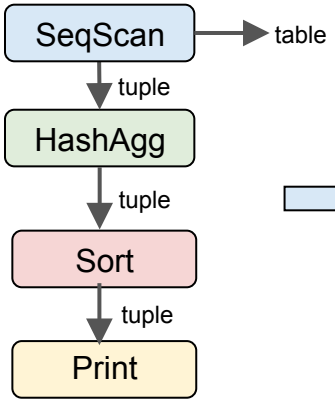
После:
Push-based модель

Выполнение плана: push-based модель (2)

- Функции-генераторы на LLVM C API:

```
LLVMFunction HashAgg(LLVMFunction consume, LLVMFunction finalize)
```

- Возвращаемые функции содержат вызовы consume для каждого выходного кортежа и finalize после обработки всех кортежей.

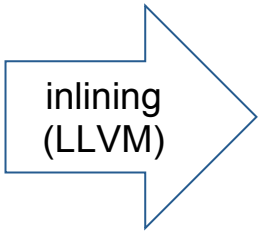



```
SeqScan(c, f)
HashAgg(c, f) = SeqScan(HashAgg.consume(),
                        HashAgg.finalize(c, f))
Sort(c, f) = HashAgg(Sort.consume(),
                     Sort.finalize(c, f))

Print() = Sort(print, null)
```

Выполнение плана: push-based модель (3)

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

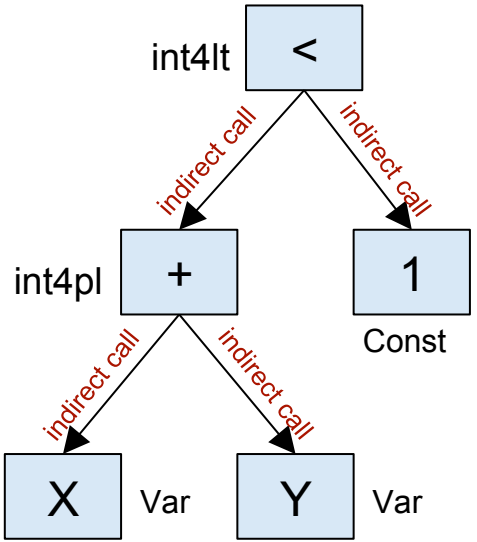



```
main() {
    for tuple ← table
        hash_table.put(tuple)
    for hash_entry ← hash_table
        sort_buffer.put(hash_entry)
    for tuple ← sort_buffer
        print(tuple)
}
```

- Отсутствие неявных вызовов.
- Отсутствие необходимости сохранения состояния.

Динамическая компиляция выражений

X+Y < 1




```
define i1 @ExecQual() {
  %x = load &X.attr
  %y = load &Y.attr

  %pl = add %x, %y
  %lt = icmp lt %pl, 1

  ret %lt
}
```

Общая схема метода

1. Перехват управления перед этапом выполнения плана.
2. Проверка операторов, функций и выражений, используемых в запросе.
   - Если запрос не поддерживается, возврат управления интерпретатору PostgreSQL.
3. Генерация кода на языке LLVM IR.
4. Компиляция в машинный код при помощи LLVM MCJIT.
5. Выполнение.

ISP RAS

Результаты

- PostgreSQL 9.6 beta2.
- Database: 100GB (on RamDisk storage).
- CPU: Intel Xeon.

TPC-H-like workload	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q17	Q19	Q20	Q22
PostgreSQL, (sec)	431,81	22,90	212,06	45,05	255,74	112,52	98,41	41,36	180,78	173,71	11,46	228,55	252,1	127,36	249,93	163,56	9,03	39,2	16,47
+with JIT (sec)	100,52	25,35	103,38	30,01	224,4	36,71	71,39	41,49	152,18	92,97	11,08	131,25	175,9	44,43	161,82	100,4	7,07	37,01	15,29
Ускорение, (раз)	**4,30**	**0,90**	**2,05**	1,50	1,14	**3,07**	1,38	1,00	1,19	1,87	1,03	1,74	1,43	**2,87**	1,54	1,63	1,28	1,06	1,08

- Тип DECIMAL изменён на DOUBLE PRECISION, CHAR(1) на ENUM.
- Выключены Bitmap Heap Scan, Material, Merge Join.
- Пока не поддерживаются: Q16, Q18, Q21.

JIT Компилятор для PostgreSQL: 2 версии

ISP RAS

- **JIT для выражений**
  - Ускорение до **20%** на TPC-H
  - Небольшие изменения в коде исполнения запросов
  - Open source: github.com/ispras/postgres
- **PostgreSQL Extension**
  - Ускорение до **5 раз** на TPC-H
  - Реализованы методы Scan / Aggregation / Join / Sort вручную с использованием LLVM API
  - Модель выполнения изменена с pull (Volcano) на push-based
  - Включает в себя JIT для выражений
  - JIT-компиляция процедуры сканирования кортежа *slot_deform_tuple()*
- Обе версии:
  - Основаны на PostgreSQL 9.6.1, поддерживают параллельное выполнение запросов
  - Автоматическая трансляция встроенных функций бэкэнда PostgreSQL в LLVM IR

Будущие Работы

- OLTP:
  - Сохранение исполняемого кода для PREPARED запросов
- JIT-компиляция построения индексов
- Реализация на LLVM всех операторов PostgreSQL
- Тестирование на реальных задачах, на других тестовых наборах
- Профилирование, дальнейшие оптимизации
- Ускорение JIT-компиляции
  - Использование встроенных оценок времени выполнения запроса в PostgreSQL
  - Параллельная компиляция

Заключение

- JIT для выражений
  - open source: github.com/ispras/postgres
  - Ускорение до **20%** на TPC-H (с небольшими изменениями в коде исполнения запросов)
- PostgreSQL Extension (завершается разработка)
  - Ускорение до **5 раз** на TPC-H
- Внедрение: ищем партнеров в индустрии
  - Будем рады сотрудничеству – получить обратную связь, протестировать на реальных задачах; что еще нужно сделать, чтобы лучше работало на различных данных

ISP RAS

Спасибо!

Questions, comments, feedback:
dm@ispras.ru