Динамическая компиляция SQL-запросов в PostgreSQL с использованием LLVM JIT

Д.М. Мельник, Р.А. Бучацкий, Р.А. Жуйков, Е.Ю. Шарыгин
Институт системного программирования Российской академии наук (ИСП РАН)
17 марта 2017

Dynamic Compilation of SQL Queries in PostgreSQL Using LLVM JIT

Dmitry Melnik, Ruben Buchatskiy, Roman Zhuykov, Eugene Sharygin
Institute for System Programming of the Russian Academy of Sciences (ISP RAS)
March 17, 2017
Motivational Example

SELECT COUNT(*) FROM tbl WHERE (x+y) > 20;

- Aggregation
- Scan
- Filter

interpreter: 56% of execution time
Motivational Example

```
SELECT COUNT(*) FROM tbl WHERE (x+y) > 20;
```

- **Aggregation**: 56% of execution time
- **Scan**: 56% of execution time
- **Filter**: LLVM-generated code: 6% of execution time

=> Speedup query execution 2 times
Project Goals

• Speed up PostgreSQL for computationally intensive SQL-queries
• What exactly we want to speed up?
  – Complex queries where performance "bottleneck" is CPU rather than disk (primarily analytics, but not limited to)
  – Optimize performance for TPC-H benchmark
• How to achieve speedup?
  – Dynamically compile queries to native code using LLVM JIT
Related Work

- Vitesse DB:
  - Proprietary database based on PostgreSQL
  - JIT compiling expressions as well as execution plan
  - Speedup up to 8x on TPC-H Q1
  - JIT compiling expressions for Filter and Aggregation
  - Speedup up to 37% on TPC-H
- New expression interpreter for Postgres (WIP by Andres Freund):
  - Changes tree walker based interpreter to more effective one
  - Also adds LLVM JIT for expressions (~20% speedup on TPC-H Q1)
LLVM (Low Level Virtual Machine) – compiler infrastructure designed for program compilation and optimization

- Platform-independent internal representation (LLVM IR)
- Wide set of compiler optimizations
- Code generation for popular platforms (x86/x86_64, ARM32/64, MIPS, ...)
- Well suited for building a JIT-compiler: the dynamic library with an API for generating LLVM IR, optimizing and compiling into machine code via just-in-time compilation
- License: UIUC (permissive BSD-like)
- Relatively simple code, easy to understand
Using LLVM JIT is a popular trend

- 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), recently replaced by B3 (Apple’s custom JIT)
  - LLV8 – LLVM added to Google V8 (open source project by ISP RAS)

- DBMS:
  - MemSQL, Impala
  - … and now PostgreSQL
Adding LLVM JIT to PostgreSQL?
JIT-compiling Expressions

\[ X + Y < 1 \]

\[
\text{define i1 @ExecQual()} \{
\%
x = \text{load} &X\text{.attr}
\%
y = \text{load} &Y\text{.attr}
\%
pl = \text{add} \%x, \%y
\%
lt = \text{icmp} \text{lt} %pl, 1
\text{ret} \%lt
\}
\]
Precompiling Postgres Backend Functions

Build time:

- *.c → clang → *.bc → llvm-link → backend.bc → opt → backend-opt.bc

PostgreSQL Backend (src/backend/*.c)

LLVM Bitcode

Server start up

Load .bc

Memory Buffer

Parse

Query execution:

- SQL Query → Generate code for specific SQL query → LLVM Module → Optimize & Compile → Native code → Execute
Pre-compiling backend functions

Datum

```c
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);
}
```

Clang

```cpp
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 %5, 63
  %.lobit1 = lshr i64 %4, 63
  %.lobit2 = lshr i64 %5, 31
  %7 = icmp eq i64 %.lobit2, %.lobit
  %or.cond = or i1 %7, %6
  br i1 %or.cond, label %ret, label %overflow

overflow:
  call void @ereport(...) 
ret:
  ret i64 %5
}
```
JIT Compilation at Different Levels

- Based on Postgres 9.6.1
- TPC-H Q1 speedup is 20%
- Expressions JIT is published as open source and available at github.com/ispras/postgres

Expressions JIT

backend functions


- supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, ...):
  \[ a^2 + b^2 \leq r^2 \]

- + supports built-in functions:
  \[ \sqrt{\text{pow}(a, 2) + \text{pow}(b, 2)} \leq r \]
# Profiling TPC-H

## TPC-H Q1:

```sql
SELECT
    l_returnflag, l_linenstatus,
    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_linenstatus
ORDER BY l_returnflag, l_linenstatus;
```

<table>
<thead>
<tr>
<th>Function</th>
<th>TPC-H Q1</th>
<th>TPC-H Q2</th>
<th>TPC-H Q3</th>
<th>TPC-H Q6</th>
<th>TPC-H Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecQual</td>
<td>6%</td>
<td>14%</td>
<td>32%</td>
<td>3%</td>
<td>72%</td>
</tr>
<tr>
<td>ExecAgg</td>
<td>75%</td>
<td>-</td>
<td>1%</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>SeqNext</td>
<td>6%</td>
<td>1%</td>
<td>33%</td>
<td>-</td>
<td>13%</td>
</tr>
<tr>
<td>IndexNext</td>
<td>-</td>
<td>57%</td>
<td>-</td>
<td>-</td>
<td>13%</td>
</tr>
<tr>
<td>BitmapHeapNext</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>85%</td>
</tr>
</tbody>
</table>
Executor JIT Overview

1. **Use** `ExecutorRun_hook` *(before execution start)*
2. Check whether all nodes, functions and expressions used in query plan are supported by JIT
   - If the query unsupported, call standard PostgreSQL interpreter
3. Generate LLVM IR for query
4. Compile to native code using LLVM JIT
5. Run code
Plan Execution: Volcano model

- Each operator is represented as a sequence of tuples, which is accessed by calling method `next()` (ExecProcNode in Postgres)
- Indirect function call: branch misprediction, inlining is impossible
- Need to store/load internal state between calls to `next()`

```sql
select a, sum(b) from tbl group by a order by a;
```
Plan Execution: push-based model

- Query execution is controlled by the leaf node of Plan tree
- The query is represented as number of loops
Changing Execution Model

**Before:** Pull-based (Volcano) model

**After:** Push-based model

```sql
select <columns> from <table> where <condition>
group by <column>
order by <column>;
```

Diagram:
- **SeqScan**
  - Next() → Tuple
  - Next() → Tuple
  - Next() → Tuple
  - Table
- **HashAgg**
  - Next() → Tuple
  - Next() → Tuple
  - Next() → Tuple
  - Hash Table
- **Sort**
  - Next() → Tuple
  - Next() → Tuple
  - Table
- **Print**
  - Next() → Tuple
  - Next() → Tuple
  - Next() → Tuple
  - Table

**Before:**
- Hash Table.put(Tuple)
- Sort Buffer.put(Hash Entry)
- Print(Tuple)

**After:**
- Hash Table.put(Tuple)
- Sort Buffer.put(Hash Entry)
- Print(Tuple)
Plan Execution: push-based model

- Generator functions for LLVM C API:

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

- Generated functions contain calls to `consume()` for each output tuple and `finalize()` after processing all tuples

  ```c
  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)
  ```
Plan Execution: push-based model

• No indirect calls
• No need to store internal state

```cpp
void main() {
    for (tuple ← table
        hash_table.put(tuple)
    )
    for (hash_entry ← hash_table
        sort_buffer.put(hash_entry)
    )
    for (tuple ← sort_buffer
        print(tuple)
    )
}
```
JIT Compilation at Different Levels

Expressions JIT

- backend functions

Executor JIT

Supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, ...): 
\[ a^2 + b^2 \leq r^2 \]

- supports built-in functions:
  \[ \sqrt{a^2 + b^2} \leq r \]

- compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements Push model

TPC-H Q1 speedup ~
- 17%
- 20%
- 3%
JIT-compiling attribute access

- slot_deform_tuple
- Optimize out:
  - attribute number
  - nullability
  - attribute lengths
  - unused attributes

```c
for (attnum = 0; attnum < natts; attnum++) {
    Form_pg_attribute thisatt = att[attnum];

    if (att_isnull(attnum, bp)) {
        values[attnum] = (Datum) 0;
        isnull[attnum] = true;
        continue;
    }

    isnull[attnum] = false;
    off = att_align_nominal(off, thisatt->attalign);
    values[attnum] = fetchatt(thisatt, tp + off);
    off = att_addlength_pointer(off, thisatt->attlen, tp + off);
}

isnull[0] = false;
values[0] = *(int32 *)(tp);
isnull[2] = false;
values[2] = *(int32 *)(tp + 8);
...```
Expressions JIT + optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes
 Executor JIT + compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements Push model

- Implemented as an extension for Postgres 9.6.2
- TPC-H Q1 speedup is 5.5x times
- Continuing work on Executor JIT
- Compilation time is sufficient for short-running queries

TPC-H Q1 speedup ~ 5.5x times

17% 20% 3%
Results for Executor JIT

- Measured on PostgreSQL 9.6.1, extension with LLVM 4.0 ORC JIT
- Database: 75GB (on RamDisk storage, data folder size ~200GB), shared_buffers = 32GB
- CPU: Intel Xeon E5-2699 v3.

<table>
<thead>
<tr>
<th>TPC-H-like workload</th>
<th>Q1</th>
<th>Q3</th>
<th>Q9</th>
<th>Q13</th>
<th>Q17</th>
<th>Q19</th>
<th>Q22</th>
<th>Q6</th>
<th>Q14</th>
<th>Q15</th>
<th>Q18</th>
<th>Q12</th>
<th>Q10</th>
<th>Q4</th>
<th>Q2</th>
<th>Q20</th>
<th>Q7</th>
<th>Q11</th>
<th>Q5</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL (sec)</td>
<td>283.27</td>
<td>107.38</td>
<td>197.27</td>
<td>134.92</td>
<td>8.62</td>
<td>6.10</td>
<td>12.65</td>
<td>72.94</td>
<td>77.48</td>
<td>139.56</td>
<td>130.48</td>
<td>136.29</td>
<td>98.59</td>
<td>45.02</td>
<td>17.64</td>
<td>79.93</td>
<td>153.69</td>
<td>6.13</td>
<td>342.66</td>
<td>43.08</td>
</tr>
<tr>
<td>+with JIT (sec)</td>
<td>52.11</td>
<td>60.26</td>
<td>164.52</td>
<td>95.69</td>
<td>5.99</td>
<td>4.83</td>
<td>9.74</td>
<td>25.83</td>
<td>31.52</td>
<td>63.20</td>
<td>61.74</td>
<td>72.74</td>
<td>56.10</td>
<td>34.93</td>
<td>15.57</td>
<td>75.63</td>
<td>147.44</td>
<td>5.95</td>
<td>338.60</td>
<td>42.89</td>
</tr>
<tr>
<td>Compilation</td>
<td>0.81</td>
<td>1.05</td>
<td>1.51</td>
<td>0.93</td>
<td>0.78</td>
<td>0.99</td>
<td>1.12</td>
<td>0.40</td>
<td>0.72</td>
<td>1.04</td>
<td>1.03</td>
<td>0.89</td>
<td>1.25</td>
<td>0.71</td>
<td>2.20</td>
<td>0.96</td>
<td>1.45</td>
<td>1.14</td>
<td>1.24</td>
<td>1.62</td>
</tr>
<tr>
<td>Speedup, (times)</td>
<td>5.44</td>
<td>1.78</td>
<td>1.20</td>
<td>1.41</td>
<td>1.44</td>
<td>1.26</td>
<td>1.30</td>
<td>2.82</td>
<td>2.46</td>
<td>2.21</td>
<td>2.11</td>
<td>1.87</td>
<td>1.76</td>
<td>1.29</td>
<td>1.13</td>
<td>1.06</td>
<td>1.04</td>
<td>1.03</td>
<td>1.01</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Type DECIMAL changed to DOUBLE PRECISION; CHAR(1) to ENUM.
- Bitmap Heap Scan, Material, Merge Join turned off for queries, marked with **yellow**;
Q16 и Q21 are not yet supported.
Saving optimized native code for PREPARED queries

- When using *generic* plan for prepared statements it’s possible to save generated code for the plan and then reuse it, eliminating compilation overhead for OLTP workload
- Slower than our regular JITted code because it can’t contain immediate values and absolute addresses for structures allocated at the stage of query initialization (e.g. in ExecutorStart)
- Increased memory footprint because of storing native code along with the plan for PREPARED query
Prototype implementation for saving code

Modified code generation to add needed indirection for simple query:

```sql
set enable_indexscan = 0; set enable_bitmapscan = 0;
prepare qz as select * from lineitem where l_quantity < 1;
```

<table>
<thead>
<tr>
<th>Size = 1 GB</th>
<th>Postgres, no JIT</th>
<th>JIT extension</th>
<th>JIT and saving code (first time)</th>
<th>Execute loaded code (next times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory, bytes</td>
<td>6736</td>
<td>8480</td>
<td>8304</td>
<td></td>
</tr>
<tr>
<td>Compile time, ms</td>
<td>-</td>
<td>172</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>Exec time, ms</td>
<td>525</td>
<td>156</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td>Total time, ms</td>
<td>525</td>
<td>328</td>
<td>344</td>
<td>170</td>
</tr>
</tbody>
</table>

Can save ~170ms on compilation, but the query runs ~15ms (9%) slower and takes extra ~1.5Kb memory (in addition to ~6.5Kb for saved plan context)
Using llvm.patchpoint

• Now working on more general approach for saving native code for prepared queries using llvm.patchpoint to reuse the code with new addresses for structures and possibly new parameter values.

• For more complex queries like those in TPC-H native code takes up to 2-5 times more memory than saved generic plan. To store both plan and native code it takes 3-6 times more memory, and on average 4.5 times more (110 Kb instead of 25 Kb for single query).

• Possible to save memory by storing only selected (“hottest”) queries, or don’t compile Postgres built-in functions with LLVM, calling them from Postgres binary instead, but this will be slower (up to ~30%).
JIT Compilation at Different Levels

Expressions JIT
- supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, ...):
  \( a^2 + b^2 \leq r^2 \)
- supports built-in functions:
  \( \sqrt{\text{pow}(a, 2) + \text{pow}(b, 2)} \leq r \)
- optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes

Executor JIT
- compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model
- Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)

Caching JITted code for PREPARED statements

+ supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, ...):
  \( a^2 + b^2 \leq r^2 \)
+ supports built-in functions:
  \( \sqrt{\text{pow}(a, 2) + \text{pow}(b, 2)} \leq r \)
+ optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes
+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model
+ Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)

TPC-H Q1 speedup
- 17%
- 20%
- 3%
- 5.5x times
+ eliminates compilation overhead
JIT-compiling Index Creation

- `comparetup_index_btree()` takes ~25-30% of total index creation time
- JIT-compiling `comparetup_index_btree()` and comparators for different types
- Comparators are inlined into `comparetup_index_btree()` during JIT-compilation
- Below are results for creating indexes for 2GB TPC-H-like database

<table>
<thead>
<tr>
<th>Type</th>
<th>Speedup</th>
<th>Original time, sec</th>
<th>JIT time, sec</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>(INT, DOUBLE)</td>
<td>7.21%</td>
<td>7.9</td>
<td>7.3</td>
<td>CREATE INDEX i_l_orderkey_quantity ON lineitem (l_orderkey, l_quantity);</td>
</tr>
<tr>
<td>INT</td>
<td>5.29%</td>
<td>9.7</td>
<td>9.2</td>
<td>CREATE INDEX i_l_suppkey ON lineitem (l_suppkey);</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>15.23%</td>
<td>13.6</td>
<td>11.5</td>
<td>CREATE INDEX i_l_quantity ON lineitem (l_quantity);</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>8.95%</td>
<td>16.9</td>
<td>15.4</td>
<td>CREATE INDEX i_l_comment ON lineitem (l_comment);</td>
</tr>
<tr>
<td>DATE</td>
<td>7.73%</td>
<td>9.6</td>
<td>8.9</td>
<td>CREATE INDEX i_l_shipdate ON lineitem (l_shipdate);</td>
</tr>
<tr>
<td>CHAR</td>
<td>18.59%</td>
<td>21.8</td>
<td>17.7</td>
<td>CREATE INDEX i_l_shipinstruct ON lineitem (l_shipinstruct);</td>
</tr>
</tbody>
</table>
JIT Compilation Everywhere

Expressions JIT
- supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):
  \[ a^2 + b^2 \leq r^2 \]

Executor JIT
- + supports built-in functions:
  \[ \sqrt{\text{pow}(a, 2) + \text{pow}(b, 2)} \leq r \]
- + optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes
- + compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements Push model

Index Creation JIT
- + Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)
- + Compiling creation of btree index

Caching JITted code for PREPARED statements
- + eliminates compilation overhead

TPC-H Q1 speedup ~
- 20%
- 17%
- 3%
- 5.5x times

slot_deform_tuple()
Automatic JIT generation?

Challenges:
- Same semantics implemented twice
- Maintainability: new features should be added to both interpreter and JIT, tested separately, etc.

Automatic code specialization could be the answer
Run time code specialization

Program:
int foo(int x, int y) {
    return x * y;
}

Data:
x = 1

Program specialized for data:
int foo_x1(int y) {
    return 1 * y;
}

- For a given SQL query in Postgres:
  - static data (Plan, PlanState, EState) — depend only from SQL query itself, and are invariant during query execution
  - dynamic data (HeapTuple, ItemPointer) — depend both from SQL query and data
Run-time specialization

Data available at run-time specialization is marked in **RED**

- Can unroll loops and pre-compute offsets
- This is actually done in LLVM IR (C shown for readability)
Run time code specialization

- **Postgres Executor code** precompiled into LLVM IR (.bc)
- **SQL Query** (this can be considered as a constant data for query execution code)
- **LLVM-based code specializer and optimizer**
- **Postgres statically compiled functions**

Specialize code for specific SQL query

Native code optimized for given SQL query
Run time code specialization
JIT Compilation Everywhere

**Expressions JIT**
- JITted code for PREPARED statements
- supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):
  \[ a^2 + b^2 \leq r^2 \]
- + supports built-in functions:
  \[ \sqrt{\text{pow}(a, 2) + \text{pow}(b, 2)} \leq r \]
- + optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes

**Executor JIT**
- + compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model
- + Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)
- + Compiling creation of *btree* index
- + eliminates compilation overhead

**Index Creation JIT**
- + Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)

**Caching JITted code for PREPARED statements**

TPC-H Q1 speedup ~ 17% 20% 5.5x times

5-19%
Conclusions

• Expression JIT
  – Open source: github.com/ispras/postgres
  – Speedup up to 20% on TPC-H

• PostgreSQL Extension JIT (still developing)
  – Speedup up to 5.5 times on TPC-H
  – Caching native code for PREPARED statements to reduce compilation time

• Index creation JIT
  – Up to 19% speedup

• Developing automatic code specialization
  – Developing Push-model Executor in Postgres (filed as GSoC project)

• Feedback is needed!
  – If you have a workload that you think can benefit from JIT, we’ll be happy to test and tune for it
  – We’re open for collaboration!
Thank you!

Questions, comments, feedback:
dm@ispraras.ru