Speeding up query execution in PostgreSQL using LLVM JIT compiler

Dmitry Melnik
dm@ispras.ru

Institute for System Programming
of the Russian Academy of Sciences
(ISP RAS)
September 8, 2016
Speeding up PostgreSQL

- What exactly do we want to accelerate?
  - Complex queries where performance "bottleneck" is CPU rather than disk
    - OLAP, decision-making support, etc.
  - Goal: performance optimization on TPC-H benchmark

- How to achieve speedup?
  - Use LLVM MCJIT for just-in-time compilation of PostgreSQL queries
What if we add LLVM JIT to the PostgreSQL?
Example of query in Postgres

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

"Volcano-style" iterative model

get_next();
get_next() - indirect call
Example of query in Postgres

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

"Volcano-style" iterative model

interpretation: 56% of execution time
Example of query in Postgres

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

"Volcano-style" iterative model

get_next() - indirect call
Profiling TPC-H

TPC-H Q1:

```sql
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 '60 days'
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus;
```
Query Interpretation vs. Query Compilation (1)
1. Reimplementing query interpreter is cumbersome and error-prone.
   a. It is necessary to implement code generation for all operations of all types supported in expressions (about 2000 functions in total).
2. Would need to constantly maintain and keep in sync.
3. Ideally: derive one from the other.
Getting rid of “Volcano-style” iterative model

Goal: get rid of the indirect calls to `get_next()`, use function inlining in LLVM.
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("integer out of range"));
    PG_RETURN_INT64(result);
}
Automatic code generation

int.cpp  LLVM C++ API that generates int.bc

```cpp
Function* define_int8pl(Module *mod) {
    Function* func_int8pl = Function::Create(..., /*Name=*/"int8pl", mod);
    // Block (entry)
    Instruction* ptr_1 = GetElementPtrInst::Create(NULL, fcinfo, 0, entry);
    LoadInst* int64_2 = new LoadInst(ptr_1, "", false, entry);
    Instruction* ptr_3 = GetElementPtrInst::Create(NULL, fcinfo, 1, entry);
    LoadInst* int64_4 = new LoadInst(ptr_4, "", false, entry);
    BinaryOperator* int64_5 = BinaryOperator::Create(Add, int64_2, int64_4, entry);
    BinaryOperator* lobit = BinaryOperator::Create(LShr, int64_2, 63, ".lobit", entry);
    BinaryOperator* lobit1 = BinaryOperator::Create(LShr, int64_4, 63, ".lobit1", entry);
    ICmpInst* int1_6 = new ICmpInst(*entry, ICMP_NE, lobit, lobit1);
    BinaryOperator* lobit2 = BinaryOperator::Create(LShr, int64_5, 63, ".lobit2", entry);
    ICmpInst* int1_7 = new ICmpInst(*entry, ICMP_EQ, lobit2, lobit);
    BinaryOperator* int1_or_cond = BinaryOperator::Create(Or, int1_6, int1_7, ".or.cond", entry);
    BranchInst::Create(ret, overflow, int1_or_cond, entry);
    // Block (overflow)
    CallInst* void_err = CallInst::Create(func_erreport, void, overflow);
    // Block (ret)
    ReturnInst::Create(mod->getContext(), int64_5, ret);
    return func_int8pl;
}
```

int.bc  LLVM IR

```llvm
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
    %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 %.lobit, 63
    %.lobit1 = lshr i64 %.lobit1, 63
    %6 = icmp ne i64 %.lobit, %.lobit1
    %.lobit2 = lshr i64 %.lobit2, %.lobit
    %7 = icmp eq i64 %.lobit2, %.lobit
    %or.cond = or i1 %6, %7
    br i1 %or.cond, label %ret, label %overflow

overflow:
    tail call void @ereport(...)  
ret:
    ret i64 %5
}
```
PostgreSQL backend functions to LLVM IR precompilation

- Needs some adjustment to avoid duplicating global state
- We updated LLVM CPPBackend library for translating LLVM IR to C++ code with the support of LLVM 3.7.1
Semi-automatic implementation in LLVM

- Semi-automatic implementation of Postgres nodes in LLVM C API
  - When traversing the plan tree, operation in a node isn't executed, instead, it generates a corresponding LLVM IR.
- Getting rid of “Volcano-style” iterative model
- Calls to precompiled backend functions
- Getting rid of the indirect calls to get_next(), use function inlining in LLVM
- Achieve up to $5x$ speedup on TPC-H Q1 compared to original PostgreSQL interpreter.
1. Ideally, Query Compiler is derived from Query Interpreter fully automatically.
2. Can use precompilation technique and specialize Query Interpreter source code for the query at hand.
Query Compiler Generation (2)

- Optimize functions called with constant arguments.

```c
declare void @ExecutePlan(%struct.PlanState* %planstate)
```

```c
define void @ExecutePlan.1()
{
  call void @ExecutePlan(i64 3735927486 to %struct.PlanState*)
  ret void
}
```

- Need a recursive function pass, with some kind of constant folding / SCCP.

- Challenges:
  - Need support for tracking memory in order to replace loads from Query Execution Plan with data being loaded.
  - Need support for CFG restructuring, such as unrolling compile-time loops (very common in Query Interpreter code).
Example (1)

Function CheckQual called with constant arguments:

```c
bool CheckQual (Tuple *tuple, List *qual)
{
    for (; qual; qual = qual->next) {
        if (getattr(tuple, qual->index) == qual->value) {
            return true;
        }
    }
    return false;
}
```

Function CFG:
Semi-Automatic vs. Automatic Approach

1. Precise control over execution model & optimizations.
2. Need to reimplement PostgreSQL execution engine. Huge development & maintenance costs.

1. Limited to Volcano execution model employed by the current PostgreSQL executor.
2. Everything is supported by default. No feature lag.
3. Opportunities to tweak & hand-optimize parts of query executor without having to rewrite it all.
4. Modest development & maintenance effort.
Results (for semi-automatic method)

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

Semi-automatic

<table>
<thead>
<tr>
<th>TPC-H</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>partial</td>
<td>yes</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PG, sec</td>
<td>431.81</td>
<td>22.90</td>
<td>212.06</td>
<td>45.05</td>
<td>255.74</td>
<td>112.52</td>
<td>98.41</td>
<td>41.36</td>
<td>180.78</td>
<td>173.71</td>
<td>11.46</td>
<td>228.55</td>
<td>252.1</td>
<td>127.36</td>
<td>249.93</td>
<td>163.56</td>
<td>9.03</td>
<td>39.2</td>
<td>16.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JIT, sec</td>
<td>100.52</td>
<td>25.35</td>
<td>103.38</td>
<td>30.01</td>
<td>224.4</td>
<td>36.71</td>
<td>71.39</td>
<td>41.49</td>
<td>152.18</td>
<td>92.97</td>
<td>11.08</td>
<td>131.25</td>
<td>175.9</td>
<td>44.43</td>
<td>161.82</td>
<td>100.4</td>
<td>7.07</td>
<td>37.01</td>
<td>15.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X times</td>
<td>4.30</td>
<td>0.90</td>
<td>2.05</td>
<td>1.50</td>
<td>1.14</td>
<td>3.07</td>
<td>1.38</td>
<td>1.00</td>
<td>1.19</td>
<td>1.87</td>
<td>1.03</td>
<td>1.74</td>
<td>1.43</td>
<td>2.87</td>
<td>1.54</td>
<td>1.63</td>
<td>1.28</td>
<td>1.06</td>
<td>1.08</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- DECIMAL types in all tables changed to DOUBLE PRECISION and CHAR(1) to ENUM
- Partial means successful run with disabled BITMAPHEAPSCAN, MATERIAL, MERGE JOIN
- Not yet supported - Q16, Q18, Q21
Conclusion

○ Developed PostgreSQL extension for dynamic compilation of SQL-queries using LLVM JIT.
○ Developed a tool for automatic compilation of PostgreSQL backend files into C++ code that uses LLVM C++ API and allows to generate LLVM bitcode of backend functions while processing queries.

○ Results:
○ Semi-automatic:
  • Speedup by ~7 times on simple synthetic tests
  • Speedup by ~5 times on TPC-H Q1
○ Automatic:
  • Currently, speedup by 10% on simple synthetic tests
Future work

○ Implement on LLVM all types of nodes.
○ Testing on TPC-* and other benchmarks, profiling, search of places to optimize.
○ Parallelism:
  ○ Parallel compilation.
○ More code to JIT (extensions, access methods, etc.)
○ Preparing for release in Open Source, interaction with the PostgreSQL Community.
Thank you!

Questions, comments, feedback:
dm@isprasad.ru