Making an Embedded DBMS JIT-friendly

Carl Friedrich Bolz  Darya Kurilova (CMU)  Laurence Tratt

Software Development Team
ECOOP
2016-07-20
def select():
    iterator = conn.execute("""select quantity, extendedprice, discount
                        from lineitem"""
                        )
    sum_qty = 0
    sum_base_price = 0
    sum_disc_price = 0
    for quantity, extendedprice, discount in iterator:
        sum_qty += quantity
        sum_base_price += extendedprice
        sum_disc_price += extendedprice * (1 - discount)
    return sum_qty, sum_base_price, sum_disc_price
- SQLite is an embedded database
- Commonly combined with a (dynamic) language, e.g. Python
- Getting the data across the boundary is slow
- Can we improve it by adding a JIT to SQLite?
- SQLite is an embedded database
- Commonly combined with a (dynamic) language, e.g. Python
- Getting the data across the boundary is slow
- Can we improve it by adding a JIT to SQLite?
- We call this combined Python/SQLite JIT “SQPyte”
Small embedded SQL database
The most used database
used a bit everywhere (Mac OS, Android, ........)
dynamically typed
PyPy and RPython

- reimplementaton of Python in RPython
- good JIT via the RPython JIT framework
- which adds a tracing JIT to PyPy semi-automatically
PyPy
Python code
RPy interp.
Python data
def select():
    iterator = conn.execute("select quantity, extendedprice, discount from lineitem")
    sum_qty = 0
    sum_base_price = 0
    sum_disc_price = 0
    for quantity, extendedprice, discount in iterator:
        sum_qty += quantity
        sum_base_price += extendedprice
        sum_disc_price += extendedprice * (1 - discount)
    return sum_qty, sum_base_price, sum_disc_price
PyPy
Python code
RPy interp.
Python data

SQLite
SQL
???
C interp.
SQL Values

8/16 HTTP://SOFT-DEV.ORG/
case OP_Return: {
    pIn1 = &aMem[pOp->p1];
    assert(pIn1->flags == MEM_Int);
    pc = (int)pIn1->u.i;
    pIn1->flags = MEM_Undefined;
    break;
}
case OP_Return: {
    pIn1 = &aMem[pOp->p1];
    assert(pIn1->flags == MEM_Int);
    pc = (int)pIn1->u.i;
    pIn1->flags = MEM_Undefined;
    break;
}

def python_OP_Return(hlquery, op):
    pIn1 = op.mem_of_p(1)
    assert pIn1.get_flags() == CConfig.MEM_Int
    pc = pIn1.get_u_i()
    pIn1.set_flags(CConfig.MEM_Undefined)
    return pc
Optimizations

- inline across database/language boundary
- type conversion optimization
- dynamic typing in SQLite
Evaluation: Hypotheses

**H1** Optimisations which cross the barrier between a programming language and embedded DBMS significantly reduce the execution time of queries.
Evaluation: Hypotheses

H1 Optimisations which cross the barrier between a programming language and embedded DBMS significantly reduce the execution time of queries.

H2 Replacing the query execution engine of a DBMS with a JIT reduces execution time of standalone SQL queries.
Microbenchmarks

Factor (lower is better)

SQPyte
SQLite

select  innerjoin  pythonjoin  pyfunction  pyaggregate  filltable  geomean

1 1 1 1 1 1 1
4.38 1.58 2.39 6.66 4.72 2.8 3.37

SQPyte
SQLite

0 1 2 3 4 5 6 7

13/16 HTTP://SOFT-DEV.ORG/
TPC-H

Factor (lower is better)

SQPyte
SQLite

HTTP://SOFT-DEv.ORG/
Where do the speedups come from?

- Optimizing the type checks of the dynamically typed SQLite DB?
- Inlining across the languages?
### Microbenchmarks Analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Factor (lower is better)</th>
<th>SQPyte</th>
<th>SQPyte-no-flags</th>
<th>SQPyte-no-inline</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>1.03</td>
<td>4.38</td>
<td>2.52</td>
<td>1.58</td>
<td>2.39</td>
</tr>
<tr>
<td>innerjoin</td>
<td>1.28</td>
<td>1.58</td>
<td>1.27</td>
<td>1.58</td>
<td>1.27</td>
</tr>
<tr>
<td>pythonjoin</td>
<td>1.02</td>
<td>2.39</td>
<td>2.12</td>
<td>2.39</td>
<td>2.39</td>
</tr>
<tr>
<td>pyfunction</td>
<td>1.04</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>pyaggregate</td>
<td>0.98</td>
<td>6.66</td>
<td>6.66</td>
<td>6.66</td>
<td>6.66</td>
</tr>
<tr>
<td>filltable</td>
<td>1.03</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td>geomean</td>
<td>1.02</td>
<td>3.37</td>
<td>2.39</td>
<td>2.39</td>
<td>2.39</td>
</tr>
</tbody>
</table>
TPC-H Analysis

Factor (lower is better)

SQPyte
SQPyte-no-flags
SQPyte-no-inline

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 3 5 7 9 11 13 15 17 19 21 geomean

0
0.5
1
1.5
2
Optimizing across the language/database boundary with a JIT can give good performance improvements.

Could reuse significant parts of the SQLite codebase.
Summary

- Optimizing across the language/database boundary with a JIT can give good performance improvements
- Could reuse significant parts of the SQLite codebase

Future Work

- Are there less intrusive ways to get some of the performance improvements?
- How much further can the approach be pushed?
- Interaction with an ORM
- Try with “real” DB?