ADAPTIVE EXECUTION OF
COMPILED QUERIES

by
Andre Kohn, Viktor Leis & Thomas Neumann

Presented by
Anders Hallem Iversen & Sveinung Øverland
INTRODUCTION
INTRODUCTION

● Traditional Volcano-style execution engines
  ○ Made for the IO/disk bottleneck
  ○ Row by row interpretation
  ○ Lots of virtual function calls
  ○ Many unnecessary instructions
  ○ Many cache misses
  ○ Unfriendly to code- and data-locality

● Compiling queries in runtime = performance
  ○ Lots of query specific information in runtime
  ○ Code-generation - query in one single function
  ○ Less function overhead, less number of instructions, cache-friendly...etc
INTRODUCTION

- Compiling queries to machine code
  - Hekaton, MemSQL, Impala, Hyper, LegoBase
- Advantages:
  - Efficiency
  - Avoids interpretation overhead of traditional execution engines (Volcano)
- Disadvantages:
  - Takes time to generate machine code
  - Just a waste of time for quick queries
INTRODUCTION

Example query with Hyper:

```sql
SELECT c.oid, c.relname, n.nspname
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhparent
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE i.inhrelid = 16490 ORDER BY inhseqno
```

- Hyper: 1ms execution time
- Compilation with LLVM: **54ms**
- **98%** wasted on compilation
GOAL

A query compiling execution framework with:

LOW LATENCY
for small queries

HIGH THROUGHPUT
for long-running queries
SOLUTION: ADAPTIVE QUERY EXECUTION

Combing interpretation and compilation

Recipe:

01 A fast bytecode interpreter specialized for database queries.

02 A method for accurately tracking query progress

03 A way to dynamically switch between interpretation and compilation
Query Execution Via Compilation
COMPILATION ENGINES

- Translation of **query plan** into low-level **machine independent code**

- This paper:
  - Based on Hyper
  - Compiles query plan to **LLVM IR**
  - Lets the LLVM run optimization passes
  - Compiles to **machine code** with the LLVM
LATENCY VS. THROUGHPUT TRADEOFF

- Experiment on TPC-H Q1; Scale 1:
  - Tradeoff between **latency** and **throughput**
  - Interpreters = *Low latency*
    - Sacrifices throughput
  - Long-running queries = compilation
  - Quick queries = interpretation
  - Unoptimized code = between
    - Optimizations are **time-consuming**
  - Proposal:
    - Switch between interpretation and compilation based on heuristics
    - Compilation = single threaded
    - Compile on **one core**, interpret on the rest
The Adaptive Framework: Execution
**THREE EXECUTION MODES**

“Compilation-based engines should support:”

01. **Bytecode interpretation**
   - Low latency for quick queries

02. **Compilation unoptimized**
   - Good tradeoff for medium-sized queries

03. **Compilation optimized**
   - Achieves peak for throughput for long-running queries
THE EXECUTION MODES

- Dynamic execution mode
  - Always start execute bytecode interpreter on all available threads
  - Monitor the execution progress
  - Is compiling with or without optimization beneficial?
  - If so, start compiling on a background thread
  - Once finished, switch all threads to the compiled machine-code
- No work is lost when switching
- Does not compile entire query
  - Compiles specific pipelines

Fig. 3. Execution modes and their compilation times.
Fig. 4. Illustration of query plan translation to pseudo code. `queryStart` is the main function. Each of the three query pipelines is translated into a `worker` function. The lower left corner shows that the work of each pipeline is split into small morsels that are dynamically scheduled onto threads.
The execution consists of **workers**
- Consist of a *state* and a *morsel* (range)
- Multiple works on the same pipeline, different *morsel*
- Work stealing

**Tracks query progress**
- Track timing information
  - How much estimated time is left?
- Number of processed morsels
  - Per pipeline?
CHOOSING BETWEEN EXECUTION MODES

- Always start a query with bytecode **interpretation**
- **Continuously** evaluate for each pipeline:
  - Should proceed? Should compile w/o optimizations?
  - Estimated time left? Expected compilation time? Expecting speedup?
- Performs **extrapolation** based on empirical data
  - Compilation time; correlated with # of instructions
  - Computed continuously by a single thread based on tracked query information

```plaintext
// f: worker function
// n: remaining tuples
// w: active worker threads
evaluatePipelineDurations(f, n, w):
  r0 = avg(rate in threadRates)
  r1 = r0 * speedup1(f); c1 = ctime1(f)
  r2 = r0 * speedup2(f); c2 = ctime2(f)
  t0 = n / r0 / w
  t1 = c1 + max(n - (w-1)*r0*c1, 0) / r1 / w
  t2 = c2 + max(n - (w-1)*r0*c2, 0) / r2 / w
  switch min(t0, t1, t2):
    case t0: return DoNothing
    case t1: return Unoptimized
    case t2: return Optimized
```

Fig. 7. Extrapolation of the pipeline durations.
FAST BYTECODE INTERPRETATION

- Generating machine-code takes a non-trivial amount of time
- LLVM => machine code, but also **interpretation**
  - Is extremely slow
  - Was designed for a versatile and generic format
  - Pointer-based in-memory representation => cache unfriendly
- Proposed solution - **custom bytecode interpreter**:
  1. Translate native LLVM IR to optimized bytecode
  2. Efficient translation from LLVM IR to bytecode
  3. Fast Virtual Machine behaving 100% identical to machine code
THE VIRTUAL MACHINE

**Register** machine
- Fixed length
- Statically typed
- Mimics LLVM IR instruction set
- Fixed length operands
- Specific operands for functions with specific return type and parameters

```c
define i32 @add(i32, i32) {
  %3 = add i32 %1, %0
  ret i32 %3
}
```

will be translated into a very similar VM fragment:

```c
add_i32 24 16 20
return_i32 24
```

```c
while (true) {
  switch ((++ip)->op) {
    case Op::add_i32: (*((int32_t*)(regs + ip->a1))) = (*((int32_t*)(regs + ip->a2))) + (*((int32_t*)(regs + ip->a3))); break;
    case Op::add_i64: (*((int64_t*)(regs + ip->a1))) = (*((int64_t*)(regs + ip->a2))) + (*((int64_t*)(regs + ip->a3))); break;
    case Op::call_void_i32: (void*)((int32_t*)(ip->lit))(*((int32_t*)(regs + ip->a1))); break;
    ...
  } // around 500 more instructions
```
TRANSLATING LLVM IR INTO VM BYTECODE

- LLVM IR => Bytecode
- Register allocation is hard
  - Fitting a program in a finite amount of registers
  - LLVM IR is Single Static Assignment (SSA)
- Requires **liveness analysis**
  - When a value comes “alive”, and when it “dies”

```c
define i32 @add(i32, i32) {
  %3 = add i32 %1, %0
  ret i32 %3
}
```

Will be translated into a very similar VM fragment:
```
add_i32 24 16 20
return_i32 24
```

What are variables live at line 1?
REGISTER ALLOCATION

- Mapping LLVM values to register slots
- Values should be in L1 cache
- Requires liveness analysis
- Problem:
  - Liveness analysis is super-linear runtime based on # of basic blocks
  - Spilling to memory is not an option
- Solution:
  - New specific linear-time register algorithm
  - Utilizes loop structures to approximate optimal register allocation
1. Interpret bytecode in VM

2. Should we compile based on the metrics?

3. If yes, compile, and switch execution mode
Experiments

8 core ryzen 7 1700x (+ equivalent intel cpu), 32gb ram, LLVM 3.8, Linux 4.11

All 22 TPC-H benchmark queries on scale factors 0.01-30 (time shown as geometric mean)
TPC-H query 11 examination

Scale factor 1, 4 core

Dynamic execution trace, precise timing for morsels being processed

We observe uneven pipeline execution time, most of the time is spent on two pipelines

Ca 1 ms before adaptive decides to compile

Adaptive 10% better than unoptimized, 40% better than bytecode and 80% better than optimized compilation
Planning and compilation time

Compared to PostgreSQL 9.6 (Volcano-style) and MonetDB 1.7 (Column-at-a-time)

TPC-H query 1 through 5, (ms)

Plan includes parsing, semantic analysis and query optimization

MonetDB and PostgreSQL can execute directly from plan but HyPer needs to generate LLVM IR (cdg stage)

Scale factor 1
Query compilation time

Adaptive uses a linear cost function based on nr instructions in the pipeline, derived from TPC-H and TPC-DS benchmarks (not particularly large queries)

Machine generated SQL can quickly become very large (MBs of SQL)

Fast translation to bytecode becomes very important

Single table scan + increasing number of aggregate expressions (10 -> 1900)

Gives query plans with between 1 000 - 160 000 LLVM instructions
Summary & Further work

Interpretation and compilation are important building blocks

Proposed an adaptive execution framework, decisions made at a pipeline granularity based on runtime feedback

Proposed a bytecode interpreter that features a linear-time translation of LLVM IR

Together able to outperform all statically chosen execution modes for TCP-H queries

Improve VM by analyzing frequently occurring instruction sequences and make bytecode instructions for them, for example NULL checks
REFERENCES