Architect a High-performance SQL Query Engine in Rust

Jin Mingjian
Outline

- Introduction
- Related works
- Engineering in Rust
- System and Language
- Execution
- Evaluation
- Future insights
Introduction

- Jin Mingjian
- Ph.D., Data nerd
- Fields: bigdata engineering, high performance infrastructure, language and system
  - Cassandra, Spark, Impala, Kudu, ClickHouse...
  - jinmingjian.xyz/resume
Introduction

- For this talk
  - Works and practices in my recent open-source project TensorBase which is building a modern big data warehouse with Rust and its friend C
  - So, all working codes which covered or not covered in this talk are available in the site: tensorbase.io
Introduction

- For audiences of this talk
  - Keep content as simple as possible
    - Indeed both high performance and bigdata system are hard
  - Implementation details are ignored intentionally
    - All are available on the project sites
  - Ask questions, re-read this presentation or join the community
Related works

- Related projects in Rust
  - [DataFusion](https://github.com/apache/arrow/tree/master/rust/datafusion)
    - Use (but not control) Apache Arrow
    - Far-from-performance-oriented architecture: Mixed Arrow and Rust operator impls (in AOT) (traditional design)
    - (opensource) Spark is a bottom baseline from the view of performance
      - The state-of-the-art speed is 100x faster than that of Spark
Related works

- Ruby Y. Tahboub et. al., "How to Architect a Query Compiler, Revisited"
  - Paper only but inspires TensorBase
  - Generative programming v.s. TensorBase's pipeline based IR and literal codegen
  - Generative representations on op-level are verbose and slow to compile
  - (generative) Style is **orthogonal to performance**
Related works

- Shoumik Palkar et. al., "Weld: A Common Runtime for High Performance Data Analytics"
  - Dedicated Weld IR for data vs TensorBase's in-language IR
    - Abstract overhead
  - Deep binding to LLVM apis which is hard to upgrade without LLVM experts

- Problem of academic papers
  - Less engineering oriented
  - Hard to be validated by the public
Engineering in Rust

Source → parse → Parse Tree → semantically transform → HIR → optimize

Rust/C FFI

Rust

C/C++

Kernel (bin) → compile → Result

Kernel (C)

LIR → codegen
Engineering in Rust

● Engineering
  ○ Engineering oriented language design makes development agile and comfortable
  ○ TensorBase benefits from
    ■ Crates ecosystem, cargo, tooling/IDE, modules, macros, language(memory safe), language(zero-overhead C interop), language(ADT)
    ■ TensorBase: one-person project in several months (hope more to join in:)
Engineering in Rust

- General aspects of TensorBase in engineering
  - Performance designed in core
  - Modernization
    - Bleeding edge nightly
    - Good practices
  - KISS
    - Minimization of Dependencies
    - Highly hackable
Engineering in Rust

- Cargo
  - Highly extensible
  - Useful commands (except that most commonly used):
    - **add**: for add latest version of deps
    - **expand**: for proc macro debugging
    - **tree**: for transitive deps checking
Engineering in Rust

- Proc macro
  - Problems
    - Learning curve is high
    - Too many out-of-date materials but less practices
    - Tooling is in primary stage
    - Hard to debug
Engineering in Rust

● Proc macro (cont.)
  ○ Suggestions
    ■ use nightly as possible
    ■ enable feature nightly `proc_macro_diagnostic`, `proc_macro_span` for better debugging output
    ■ Proc_macro2 should improve the testability
Engineering in Rust

● C interoperability
  ○ Zero overhead
  ○ Resource management
    ■ Objects in C is managed manually but not in safe Rust
  ○ Error handling
  ○ Too many “unsafe”s and “as”
    ■ Nice watch PR: [RFC - Safer Transmute](/rfcs/pull/2981)
Engineering in Rust

● Concurrency
  ○ Nice for share-nothing thread safety
  ○ Awkward when memory sharing needed
    ■ Memory sharing - cornerstone of modern multicores
    ■ High frequency copy is a performance disaster
    ■ “Channel” is behind on sharing
  ○ Lack memory model like in Java/C++
Concurrency (cont.)

- E.g. Global (Safe) Sharing/Singleton

```rust
pub static CAT: Lazy<Mutex<Catalog>> = Lazy::new(
    let conf: Conf = Conf::load(load_path: None)
    let schema_dir: String = conf.schema.schema_schema
```

Question: Oneshot change to the global but not at the declaration point?

Safe Rust: Lazy lock

Comment: Locks are heavy. It could be safe if have change(write) “happens before” use(read) (common high-perf pattern in Java)
Engineering in Rust

- Concurrency (cont.)
  - Async-await
    - Style is **orthogonal to performance**
    - Great concept but too deep boxing in kinds of implementations
      - Hard to debug when something wrong
    - Not used in TensorBase now
Engineering in Rust

- Lifetime
  - Engineering excellence but make codes complex
    - Always recommends: to dance with, rather than to evade
    - Compiler enforces more than that needed when not smart
  - Alternatives
    - Arena allocator: TensorBase IR
    - Unsafe into C: TensorBase kernel algorithms in C
Language

- Input
  - Query: Plain SQL
  - Not necessarily SQL

- Parse Tree
  - PEG, based on [Pest](https://github.com/pest-parser)
  - Lexical validation
  - (free-style) AST
Language

- IR (Intermediate Representation)
  - Layered
  - No reinventing
    - Reuse modern low-level compilation infra as possible
    - Many optimizations in popular engines not needed any more
    - e.g. CSE (Common Subexpression Elimination)
    - select (a+123) as c1, (a+123)*2 as c2 from ...
Language

- HIR (High-level IR)
  - Data related optimizations which can be not handled by low levels compilers
    - Semantic validation
    - Relational Algebra
      - e.g. predicate pushdown
    - Some RAs can be optimized by low-level compiler
Language

- HIR (High-level IR) (cont.)
  - Unified RA operators
    - Core: 4 ops
    - ->(map), +(union/agg), *(join), <> (sort/top)
    - Inspired by J. Kepner's associative array
    - In Rust...
Language

Prettyprinted Dump of HIR

```sql
select c1, c2*c2 pc2, avg(c3) from tab
where c3 > 1
group by c1, pc2
order by c1, pc2
limit 5
```

```
. (c1,pc2,avg(c3)),
← => [c1,pc2,avg(c3)) [key=(c1,pc2), kind=asc, limit=5]
← + [c1,pc2,c3) [agg_key=(c1,pc2), avg(c3)=(c3,avg)]
← → [c1,pc2,c3) [c3>1]
← → [c1,c2,c3) [pc2=c2*c2]
←→ {(t,c1),(t,c2),(t,c3)}
```
“Sea of pipes”

- Unify data and control-flow dependencies in graph of "pipes"

Pipe (a.k.a. Pipeline) (note: back to the dump)

- Operator-fused computing/data unit (being extended to more)
- Operator-level volcano model is systematically low inefficient
- Unified dual view - Data and Op
- No more pull or push style, no more data center or control center
Language

- LIR (Low-level IR)
  - Low level optimizations which can be not handled by low level compilers and not conveniently handled by high level logics
  - Platform related semantics
    - e.g. multi-cores + codegen
    - Boundaries between the high and low are not fixed
    - Scheduling will join later
Language

- LIR (Low-level IR) (cont.)
  - Parallelization representation for multi-cores
    - Generic DAG scheduling has abstract overhead
    - TensorBase: enable parallelism patterns
      - Map/reduce, fork-join...
  - Linearization representation for codegen
    - Proc macro template: provide human readable linear mapping from LIR to C primitives
Language

- LIR (Low-level IR) (cont.)
  - Right C template enables a simple map reduce pattern
  - Free style human oriented string interpolation
  - Runnable and debuggable in IDE with some setups
Execution

- Core
  - Decentralized, self-scheduling, JIT compilation based kernel
  - Decentralized != embarrassingly parallel
    - v.s. Popularly centralized scheduling
  - Why JIT compilation
    - Top performance
    - Arch agnostic
Execution

● Engine
  ○ Modified Clang based C JIT compiler
  ○ vs. popular LLVM IR based JIT engine (ClickHouse, Apache Impala...)

● Why C
  ○ Run on (almost) everything (CPU/GPU/FPGA/Accel)
  ○ Human debuggable
  ○ Fast enough compilation for OLAP
Execution

```c
int64_t kernel()
{
    char *blk_raw_c0 = NULL;
    char fpath[64];
    sprintf(fpath, "/data/n3/data/%d", 0); //TEMP
    ker_scan(&blk_raw_c0, fpath);

    struct Args s[48];
    pthread_t ths[48];
    for (size_t i = 0; i < 48; i++)
    {
        s[i].id = i;
        s[i].part_raw_c0 = blk_raw_c0;
        pthread_create(&ths[i], NULL, reduce, &s[i]);
    }
    for (size_t i = 0; i < 48; i++)
    {
        // ... parse and load data elided ...
        char* x96 = x8;
        char* x53 = x13*x42;
        x96[x76] = x53;
    }
}
```

V.S.

```c
// main function
int main(int x0, char** x1) {
    long x2 = DEFAULT_INPUT_SIZE;
    long x3 = x2;
    long x4 = 0L;
    char** x7 = (char**)malloc(x2 * sizeof(char*));
    char** x8 = x7;
    int* x9 = (int*)malloc(x2 * sizeof(int));
    int* x10 = x9;
    long x14 = 0L;
    bool x15 = false;
    int x11 = open("Emp.tbl",0);
    long x12 = lsize(x11);
    char* x13 = mmap(0, x12, PROT_READ,
                     MAP_FILE | MAP_SHARED, x11, 0);
    bool x101 = !true;
    for (;;)
    {
        // ... parse and load data elided ...
        char* x96 = x8;
        char* x53 = x13*x42;
        x96[x76] = x53;
    }
}
```

TensorBase generated kernel

Ruby Y. Tahboub et. al., Sigmod18
## Evaluation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Language (TensorBase M0)</th>
<th>Time</th>
<th>Ref Time (from public)</th>
<th>Language (Ref system)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing (TPC-DS)</td>
<td>Rust</td>
<td>130 us**</td>
<td>~50 us*</td>
<td>C++</td>
</tr>
<tr>
<td>Parsing/IR/codegen (one column sum)</td>
<td>Rust</td>
<td>130 us**</td>
<td>~1 ms*</td>
<td>C++</td>
</tr>
<tr>
<td>C Kernel JIT Compilation</td>
<td>C/C++</td>
<td><strong>13 ms</strong> (boost) - 20 ms (no boost) (Q#1 like, -O2)**</td>
<td>59 ms* (TPC-H Q#1, opt.)</td>
<td>C++</td>
</tr>
<tr>
<td>End-to-end Query Time (one column sum with 1.47B row NYC taxi dataset)</td>
<td>Almost in Rust (mixed)</td>
<td>~60 ms (compilation cached) - ~100 ms (compilation uncached)**</td>
<td>642 ms** In ClickHouse 20.05 (compilation cached)</td>
<td>C++</td>
</tr>
</tbody>
</table>

* André Kohn et. al., DOI: 10.1109/TKDE.2019.2905235

** TensorBase M0 benchmark: https://tensorbase.io/
Evaluation

Points (part 1)

- Rust is lighting fast even untuned (but not Rust compilation)
- C based JIT Compilation is lighting fast even untuned (on par with LLVM IR)
- C based JIT Compilation is much faster than C++ (and Rust) even untuned
- C based JIT Compilation is quite enough for OLAP even untuned
Evaluation

- Points (part 2)
  - Saturates memory bandwidth in core of such runs
    - **Can’t not be faster** in single 6-channel xeon sp with 100GB/s memory bandwidth(measured by vtune) for memory bound applications
    - Napkin math: $1.47 \times 4 / 100 = \sim 0.06$ sec (= 60msec)
    - Simple sum aggregation query
      => Tight loop in kernel and finally vectorized by compiler

```c
size_t s = 0;
for (size_t i = 0; i < blk_len_c0; i++)
{
    int32_t c0 = blk_c0[i];
    s += c0;
}
```
Evaluation

- Points (part 3)
  - Partial compilation
    - Makes compilation time of TensorBase is correlated to the size of hot kernel (rather than the total size of execution codes)
  - Lower bound of LLVM compilation overhead is high
    - -O0: ~9ms for helloworld (untuned)
    - Too many passes
    - Speed does not come for free
Future insights

- Storage layer
  - Popular storage and compute separation is genetically less efficient

- Optimizer
  - Makes the queries which can not be optimized fastest
  - Data driven, low entropy inference (e.g. category theory)

- Execution engine
Future insights

○ Tiered C compilation
  ■ OLTP style wants faster codegen
  ■ C compilation/interpretation possibly done in microseconds or less

○ Alternative JIT compilation choices
  ■ e.g. Rust + Cranelift codegen (JIT) backend

○ Scheduling
Future insights

- Modern hardware oblivious
- Distributed
  - Techniques on single node does *NOT* mean they only works for single node
  - Consistency as plug-in
- Engineering
  - Contracts and formal verifications
  - Rust - C - Rust chaining and maximized engineering Rust
Future insights

- TensorBase 2020.11 (WIP)
  - Main operators on on single table
  - Storage Layer v1
  - Compatibility and hybrid deployment with ClickHouse
    - Compatible to ClickHouse Native Protocol
    - Partially compatible to ClickHouse on-disk data storage (read-only)
Future insights

- TensorBase 2020.11 (WIP) (cont.)
  - Superb in complex aggregations (e.g. group by)
  - Early results (compared to ClickHouse)
    - 6x faster in several “group by” queries on top of ClickHouse MergeTree storage with a real-world billion level dataset
    - 10x faster if do some simple optimizations
    - Faster results expected when TensorBase own storage coming
Recap

- **Abstract overhead** everywhere
  - Carefully make trade-offs

- **High-performance programming paradigm** in Rust
  - Safe(almost) + unsafe (bounded but more freedom)

- **Top performance** OLAP is firstly achieved with **engineering Rust**
  - All shown can be picked up from the open source project (tensorbase.io)
Thanks