Apache DataFusion: Design Choices when Building Modern Analytic Systems

Boston University
Data System Seminar: October 28, 2024

Andrew Lamb
Staff Engineer @ InfluxData
Apache {DataFusion, Arrow} PMC

influxdata®

Andrew Lamb

Staff Engineer
InfluxData

> ~~20~~ 21 😱years in enterprise software development

Oracle: Database (2 years)

DataPower: XSLT compiler (2 years)

Vertica: DB / Query Optimizer (6 years)

Nutonian/DataRobot: ML Startups (7 years)

InfluxData: InfluxDB 3.0, Arrow, DataFusion (4 years)

Goal

Content: Case Study of Choices in Apache DataFusion Query Engine

- Exposure to important aspects of analytic query engines
- Introduction to currently important technologies

Convince you to build with / contribute to the building blocks:

- Rust
- Arrow
- Parquet
- DataFusion!
- …

Outline

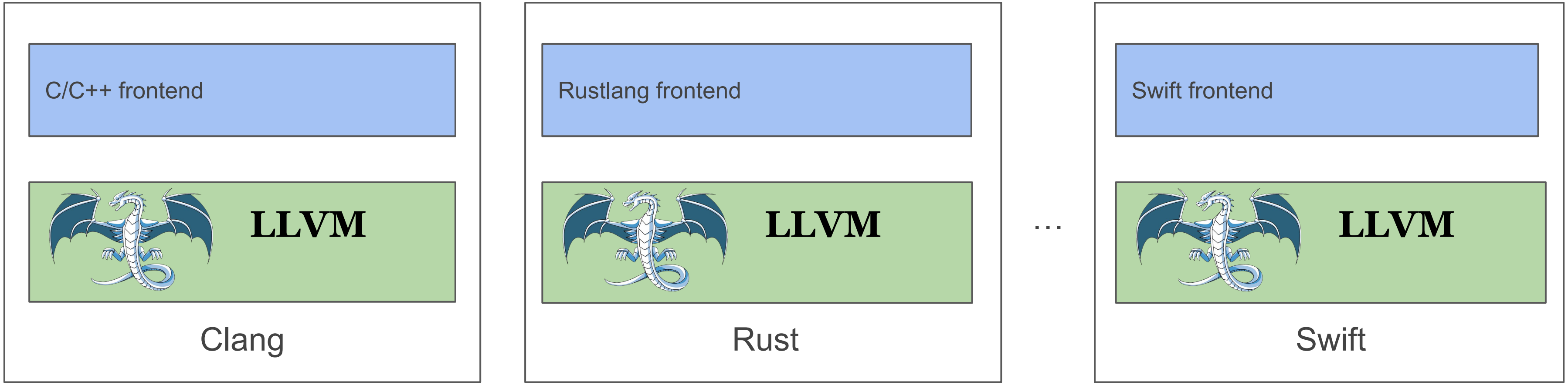
Brief Intro to DataFusion

Design Decisions

- Options
- What we chose and why
- Technical Overview
- Lessons learned

Intro to DataFusion

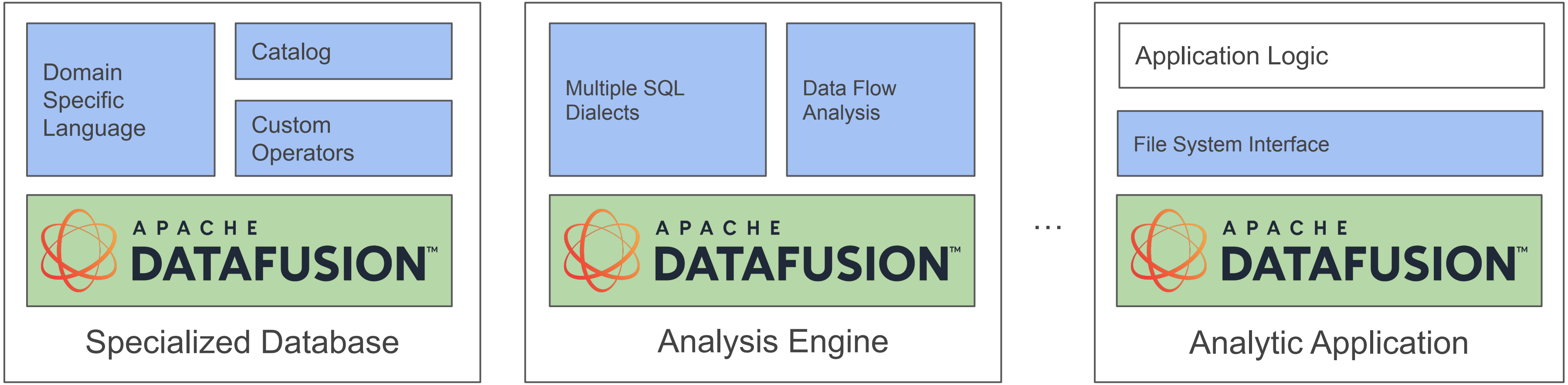
Analogy: DataFusion is LLVM for Databases



LLVM enabled innovation in programming languages:

- High quality reusable optimizer, code generator, debugger, lsp integration, etc.
- Focus on language design, ecosystem, libraries, etc

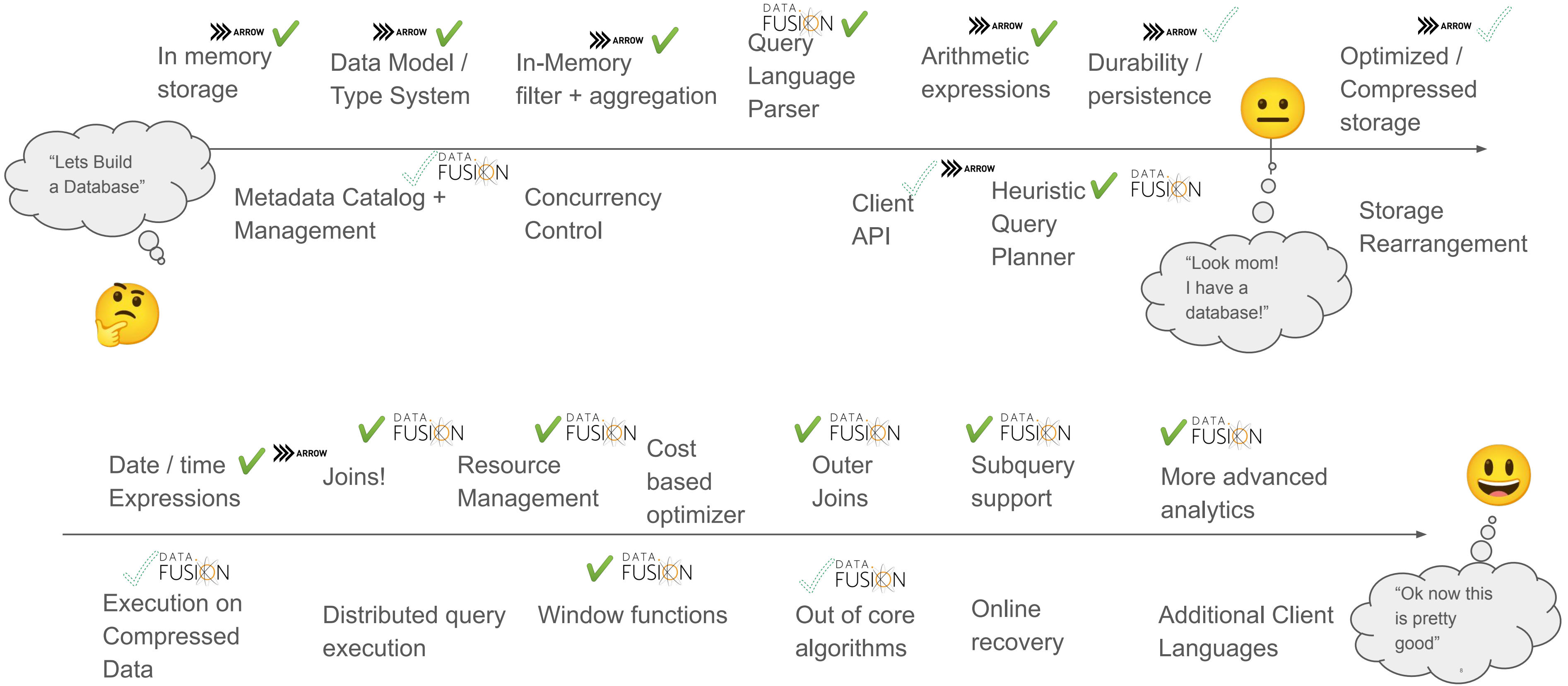
Analogy: DataFusion is LLVM for Databases



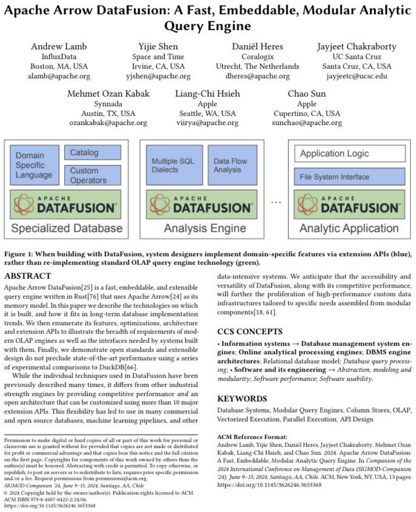
DataFusion enables innovation in data intensive systems

- High quality reusable SQL planner, optimizer, function library, vectorized operators, etc
- Focus on language design, data management, use case specific features

Implementation timeline for a new Database system



More Reading / Viewing / Background



Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine

Andrew Lamb, InfluxData, Boston, MA, USA, alamb@apache.org
Yijie Shen, Space and Time, Irvine, CA, USA, yjshen@apache.org
Daniël Heres, Coralogix, Utrecht, The Netherlands, dheres@apache.org
Jayjeet Chakraborty, UC Santa Cruz, Santa Cruz, CA, USA, jayjeetc@ucsc.edu
Mehmet Ozan Kabak, Synnada, Austin, TX, USA, ozankabak@apache.org
Liang-Chi Hsieh, Apple, Seattle, WA, USA, viirya@apache.org
Chao Sun, Apple, Cupertino, CA, USA, sunchao@apache.org

Figure 1: When building with DataFusion, system designers implement domain-specific features via extension APIs (blue), rather than re-implementing standard OLAP query engine technology (green).

ABSTRACT

Apache Arrow DataFusion[25] is a fast, embeddable, and extensible query engine written in Rust[76] that uses Apache Arrow[24] as its memory model. In this paper we describe the technologies on which it is built, and how it fits in long-term database implementation trends. We then enumerate its features, optimizations, architecture and extension APIs to illustrate the breadth of requirements of modern OLAP engines as well as the interfaces needed by systems built with them. Finally, we demonstrate open standards and extensible design do not preclude state-of-the-art performance using a series of experimental comparisons to DuckDB[66].

While the individual techniques used in DataFusion have been previously described many times, it differs from other industrial strength engines by providing competitive performance *and* an open architecture that can be customized using more than 10 major extension APIs. This flexibility has led to use in many commercial and open source databases, machine learning pipelines, and other data-intensive systems. We anticipate that the accessibility and versatility of DataFusion, along with its competitive performance, will further the proliferation of high-performance custom data infrastructures tailored to specific needs assembled from modular components[18, 61].

CCS CONCEPTS

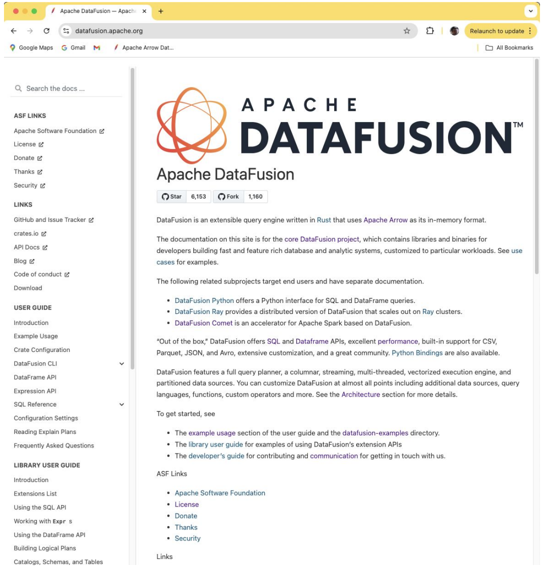
• **Information systems** → **Database management system engines**; **Online analytical processing engines**; **DBMS engine architectures**; Relational database model; *Database query processing*; • **Software and its engineering** → *Abstraction, modeling and modularity; Software performance; Software usability.*

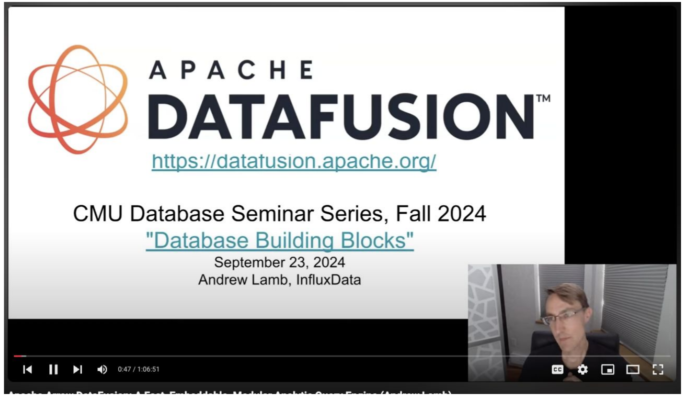
KEYWORDS

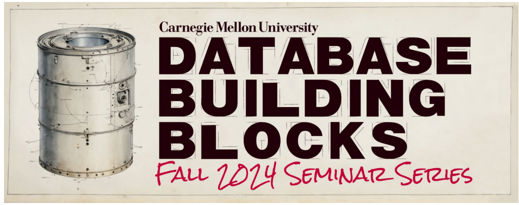
Database Systems, Modular Query Engines, Column Stores, OLAP, Vectorized Execution, Parallel Execution, API Design

ACM Reference Format:
Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653368

ACM ISBN 979-8-4007-0422-2/24/06.
https://doi.org/10.1145/3626246.3653368







Santiago
Chile

SIGMOD
PODS
2024

In [SIGMOD 2024]

[https://datafusion.apache.org/]

[https://db.cs.cmu.edu/seminar2024/]

Design Decisions

Programming Language?

Programming Language

Option 1: C/C++

Pros: Well understood, significant track record in databases

Cons: Hard to write correct code. The build system! Macros! etc.

Option 2: Rust

Pros: Memory and Thread safety, Hip langage (attracts new developers), modern tooling (e.g. Cargo)

Cons: Not battle tested for Database implementations when choosing

Programming Language

What we did: Rust

Why:

- Initially because Andy Grove believed in Rust*

* See https://andygrove.io/2018/01/rust-is-for-big-data/

Lamb Theory on Evolution of Systems Languages

time

2000s: C/C++ **Productivity** **Speed**

Personal C/C++ productivity anecdotes. Hours spent:

- Chasing build problems: “symbol not found”, what `-W`, `-l`… incantation needed 🤔
- Memory stomps/races: “the program segfaults under extreme load, intermittently”

2010s: Java **Productivity** **Speed**

2020s: Rust **Productivity** **Speed**

Thanks to https://x.com/KurtFehlhauer [(twitter)]() for helping with this slide

Quiz: does this program have undefined behavior?

```cpp
std::vector<int> v { 10, 11 };
int *vptr = &v[1];
v.push_back(12);
std::cout << *vptr;
```

```rust
let mut v = vec![10, 11];
let vptr = &mut v[1];
v.push(12);
println!("{}", *vptr);
```

Points *into* v.

Use after free (maybe)

Compiler error

Source: Communications of the ACM: [Safe Systems Programming in Rust] codapi

C++ playground

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> v { 10, 11 };
  int *vptr = &v[1];
  v.push_back(12);
  std::cout << *vptr;
}
```

Run Share Fullscreen

```
11
```

C++ is a general-purpose programming language that began as an extension of C, but later acquired object-oriented, generic, and functional features.

homepage • αω tutorial • community

CodeAPI Sandbox

Example: Implement specialized group storage in Rust

```rust
fn insert_if_new_inner<MP, OP, B>(
   &mut self,
   values: &ArrayRef,
   mut make_payload_fn: MP,
   mut observe_payload_fn: OP,
) where
   MP: FnMut(Option<&[u8]>) -> V,
   OP: FnMut(V),
   B: ByteArrayType,
{
   // step 1: compute hashes
   let batch_hashes = &mut self.hashes_buffer;
   batch_hashes.clear();
   batch_hashes.resize(values.len(), 0);
   create_hashes(&[values.clone()], &self.random_state, batch_hashes)
       // hash is supported for all types and create_hashes only
       // returns errors for unsupported types
       .unwrap();

   // step 2: insert each value into the set, if not already present
   let values = values.as_bytes::<B>();

   // Ensure lengths are equivalent
   assert_eq!(values.len(), batch_hashes.len());
…
```

binary_map.rs from DataFusion

```rust
    for (value, &hash) in values.iter().zip(batch_hashes.iter()) {
        // handle null value
        let Some(value) = value else {
            ...(handle nulls here)
            };
            observe_payload_fn(payload);
            continue;
        };

        // get the value as bytes
        let value: &[u8] = value.as_ref();
        let value_len = O::usize_as(value.len());

        // value is "small"
        let payload = if value.len() <= SHORT_VALUE_LEN {
            let inline = value.iter().fold(0usize, |acc, &x| {
                acc << 8 | x as usize
            });

            // is value is already present in the set?
            let entry = self.map.get_mut(hash, |header| {
                // compare value if hashes match
                if header.len != value_len {
                    return false;
                }
                // value is stored inline so no need to consult buffer
                // (this is the "small string optimization")
                inline == header.offset_or_inline
            });

            if let Some(entry) = entry {
                entry.payload
            }
```

```rust
            if let Some(entry) = entry {
                entry.payload
            }
            // if no existing entry, make a new one
            else {
                // Put the small values into buffer
                self.buffer.append_slice(value);
                self.offsets.push(
                    O::usize_as(self.buffer.len())
                );
                let payload = make_payload_fn(Some(value));
                let new_header = Entry {
                    hash,
                    len: value_len,
                    offset_or_inline: inline,
                    payload,
                };
                self.map.insert_accounted(
                    new_header,
                    |header| header.hash,
                    &mut self.map_size,
                );
                payload
            }
        }
        // value is not "small"
        else {
...
    }
    // Check for overflow in offsets
```

binary_map.rs from DataFusion

```rust
// value is not "small"
else {
  // Check if the value is already present in the set
  let entry = self.map.get_mut(hash, |header| {
      // compare value if hashes match
      if header.len != value_len {
          return false;
      }
      // Need to compare the bytes in the buffer
      // SAFETY: buffer is only appended to, and we correctly inserted values and offsets
      let existing_value =
          unsafe { self.buffer.as_slice().get_unchecked(header.range()) };
      value == existing_value
  });
```

binary_map.rs from DataFusion

Rust: Lessons Learned

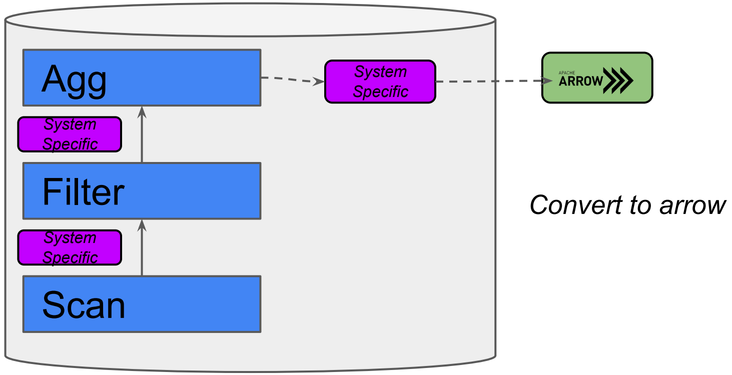
- Rust lived up to the hype
  - In 4 years, we had ~1 memory issue, and no multi-threaded bugs / race conditions
- Learning curve is quite steep
  - Be prepared to curse the compiler for a while
- Ecosystem / package manager is amazingly productive
  - `cargo new my_project`
  - `cd my_project`
  - `cargo add datafusion`

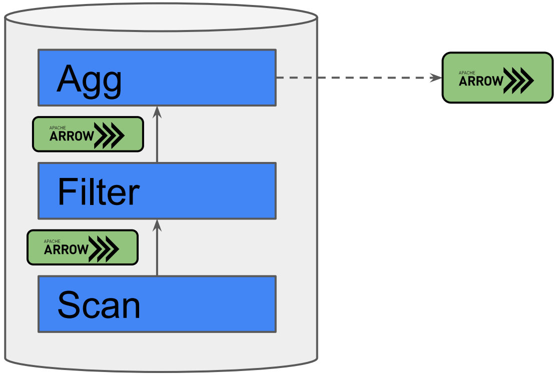
Rust: Lessons Learned

- Improves Open Source Project Velocity
  - Compiler enforces memory safety rather than relying on code reviews
  - Reviewer bandwidth is the most limited resource we have
- Improved Quality of Contributions
  - Non trivial dedication to learn Rust → filtering effect increases contribution quality

Memory Format?

Memory Format





Option 1: Use specialized structures internally, convert to Arrow at edges (Spark, Velox, DuckDB, …)

Pros: Can use specialized structures

Cons: Maintain specialized code

Option 2: Use Arrow Internally (pola.rs, Acero)

Pros: Fast interchange, reuse Arrow libraries, UDF* become trivial

Cons: Constrained(*) to Arrow

Memory Format

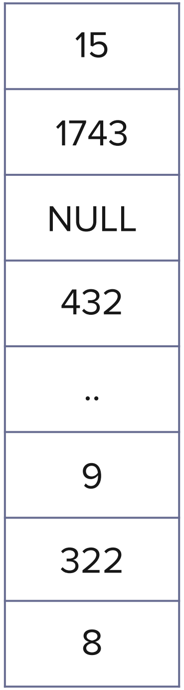
What we did:

- Used Apache Arrow

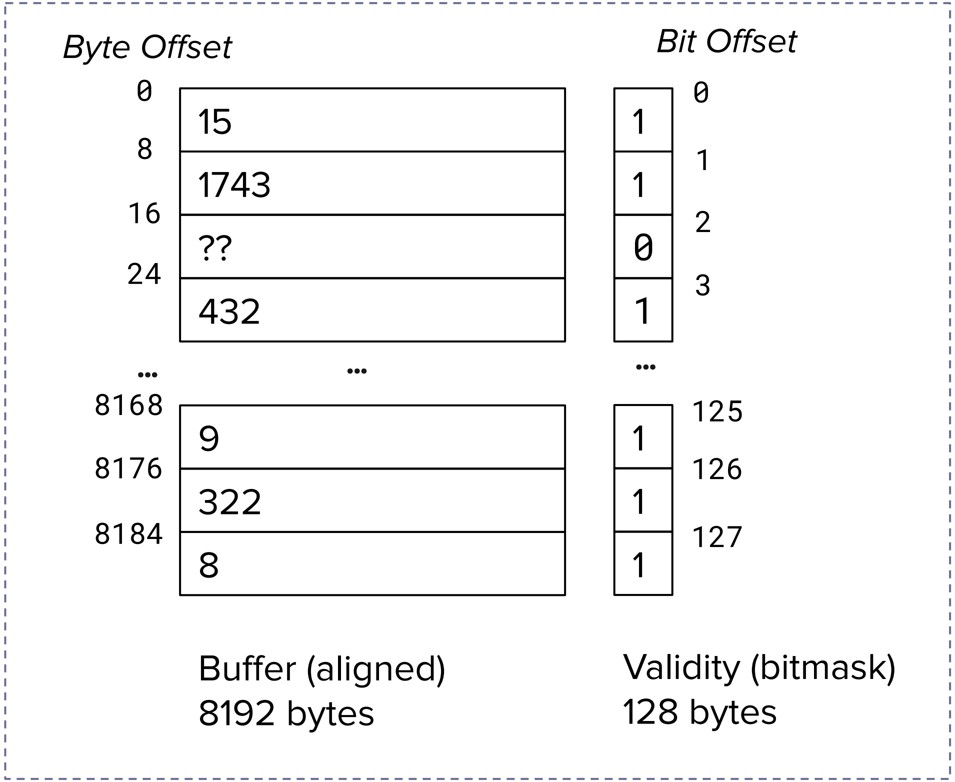
Why:

- Theory: Using Arrow is “good enough” compared to specialized structures

Arrow Array: Int64Array



Logically 1024
8 byte integers



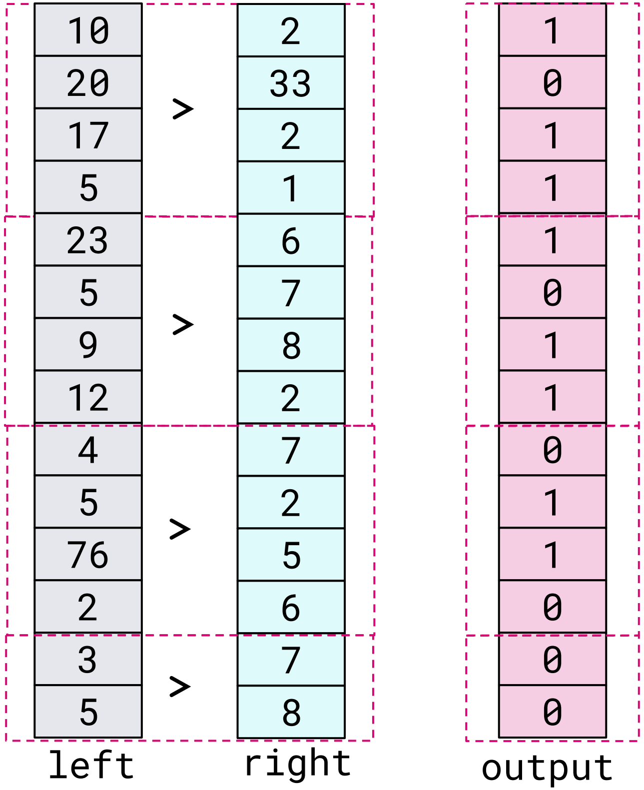
Arrow Array

Pretty much what you will find in every vectorized column store engine

 |

influxdata®

Compute Kernels




```
let output = gt(
  &left,
  &right
);
```

The `gt` *(greater than) kernel computes an output* `BooleanArray` *where each element is* `left > right`

Kernels handle nulls (validity masks), optimizations for different data sizes, etc.

~50 different kernels, full list: [docs.rs page]()

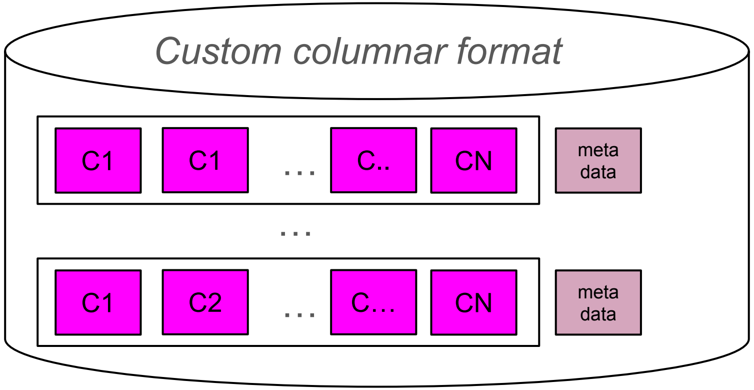
Memory Format - Lessons Learned

- Upstream wasn't quite ready ⇒ needed lots of help
- arrow-rs optimized kernels were as important as layout
- Missing Features: * Selection Vectors / "String Views" / RLE encoding
- Single constant value (ScalarValue) should have been in Rust Arrow
- Awkward that Arrow `DataTypes` both logical and physical (`DictionaryArray`)

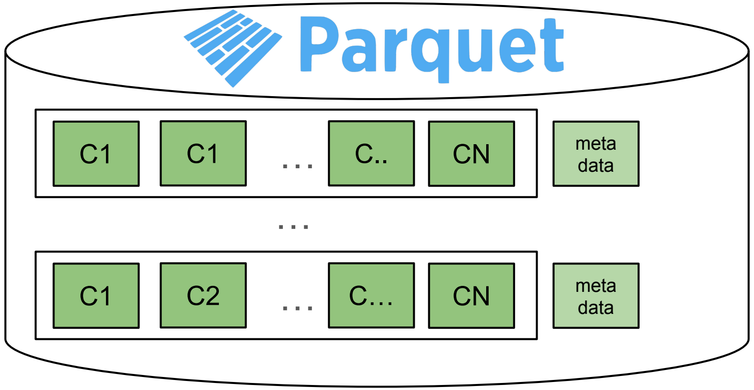
* RLE + StringView were added later

Storage/File Format?

File Format



PAX style, encodings, statistics in Zone Maps, etc



Option 1: Custom format (e.g. DuckDB, Snowflake, Vertica, ...)

Pros: Can use specialized structures, encodings, control the format

Cons: Maintain specialized code, pay to copy data in / out of this format

Option 2: Existing Format (e.g. Parquet)

Pros: Well understood, ecosystem interoperability

Cons: Constrained(*) to formats + existing implementations

File Format

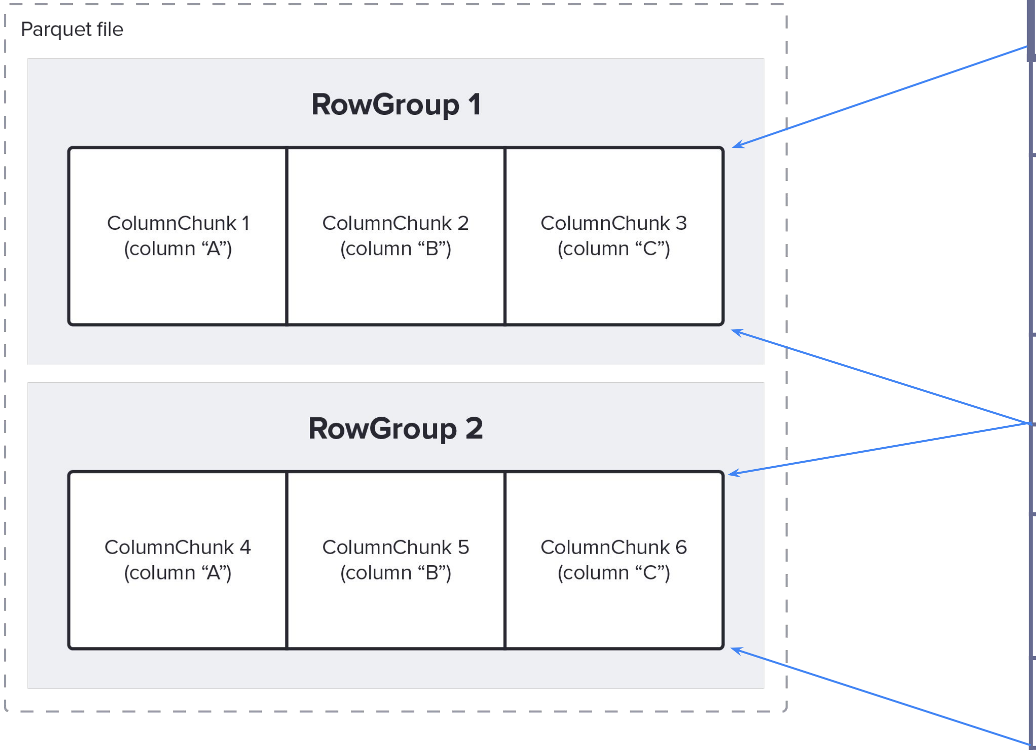
What we did:

- Use Parquet, Avro, Json, CSV, Arrow
- Extension APIs for others

Why?

- Parquet has enough (Pax, Bloom Filters, Zone Maps)
- Huge amount of Parquet already out there (table stakes!)

Parquet Organization

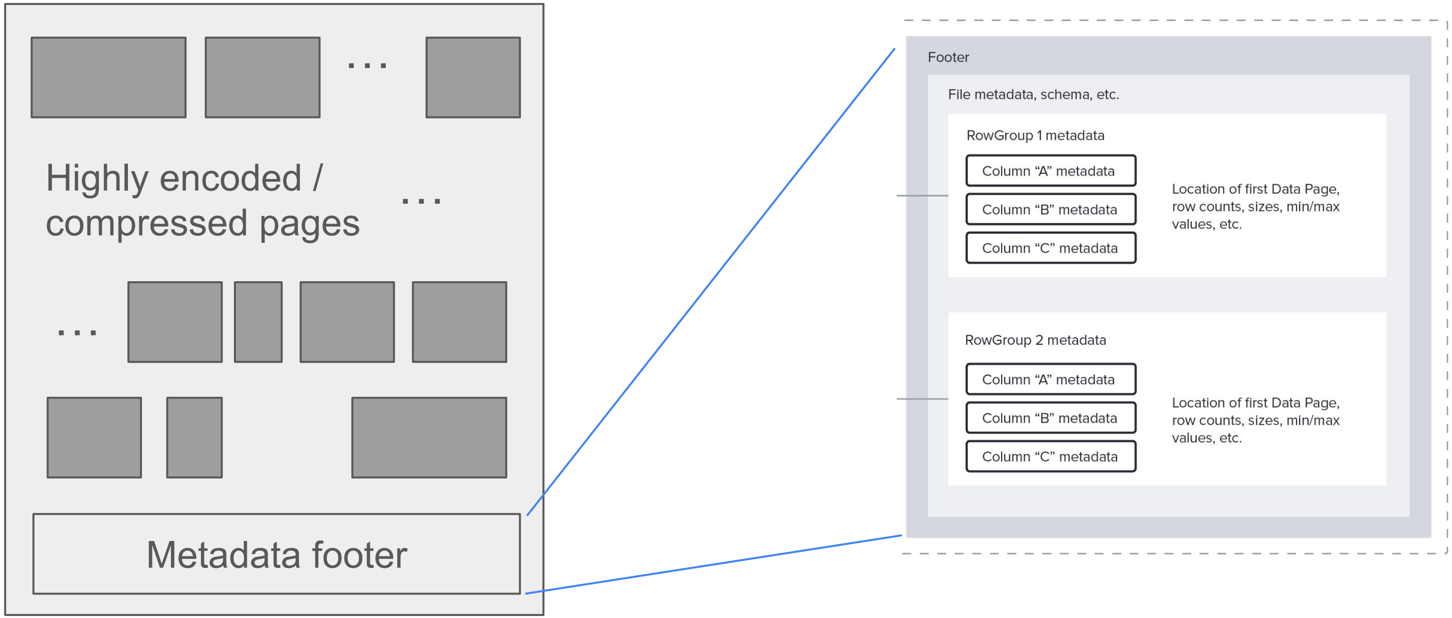


A	B	C
15	Foo	1/1/2023
	•••	
11	Bar	..1/5/2023
50	Baz	1/1/2023
	•••	
32	Blarg	1/6/2023

Source: https://www.influxdata.com/blog/querying-parquet-millisecond-latency/

(“PAX” in DB literature)

Parquet Structure + Metadata



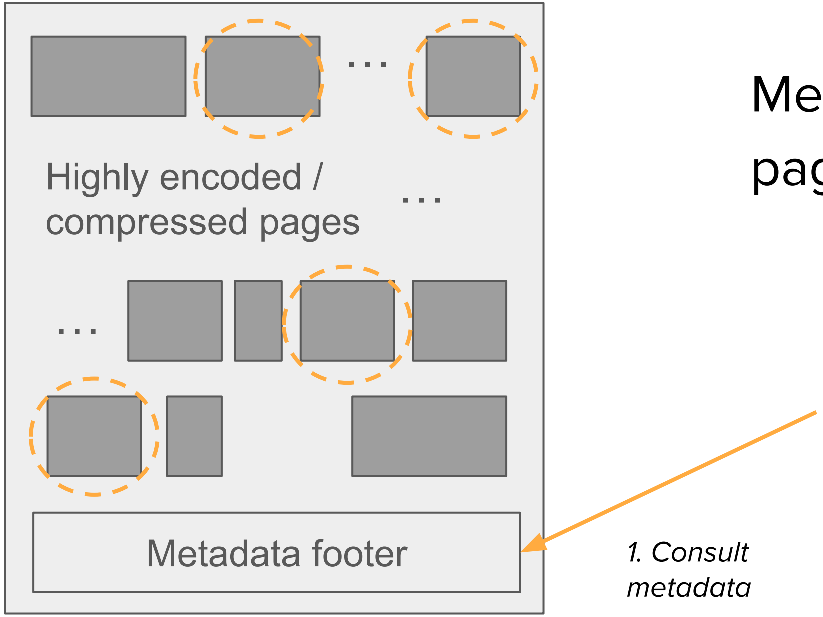
Footer contains location of pages, and statistics such as min/max/count/nullcount.

(“Zone Maps”, “Small Materialized Aggregates” in DB literature)

Source:
https://www.influxdata.com/blog/querying-parquet-millisecond-latency/

Parquet Projection + Filter Pushdown

2. Only read/decode necessary pages



1. Consult metadata

Metadata + query to prune (skip) pages that aren’t needed

```
SELECT A
...
WHERE C > 25
```

Source:
https://www.influxdata.com/blog/querying-parquet-millisecond-latency/

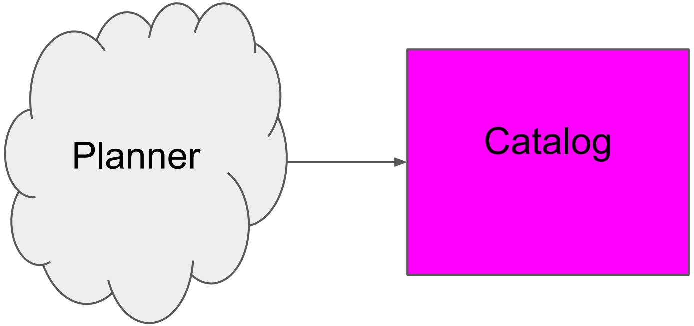
File Format - Lessons Learned

- Standard formats ⇒ large community interested, able and willing to help
- Parquet has many optimization opportunities
  - Rust Parquet implementation is now really good
  - Statistics Pruning (file, row group, page index)
  - Filter pushdown / Late Materialization / IO Interleaving
- Leveraging existing format and invest heavily in the software implementation
  - Work with community to evolve rather than replace Parquet

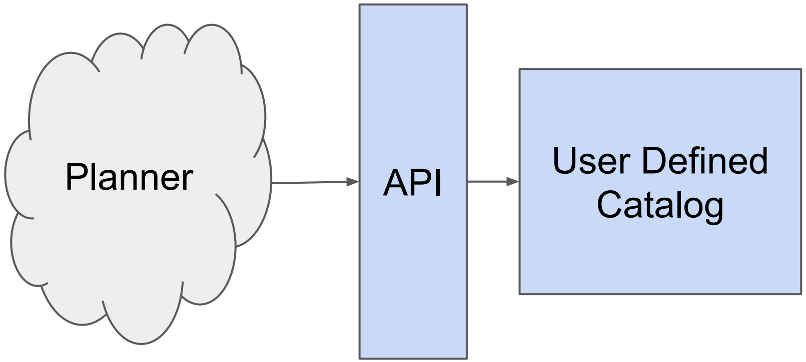
See blog: [Querying Parquet with Millisecond Latency] for more details

Catalog?

Catalog Format



Concrete catalog implementation



User provides the catalog implementation

Option 1: Provide catalog (e.g. `.sqlite`, `.duckdb`, Iceberg, etc)

Pros: Fast to start using

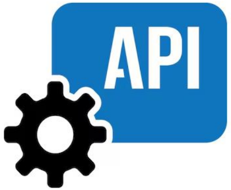
Cons: catalog implementation bound to usecase (e.g. local files vs remote service), planner may be more coupled to catalog

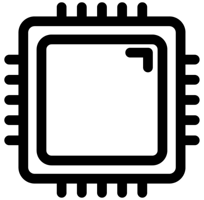
Option 2: Provide an API (e.g. Calcite)

Pros: can tailor the catalog to needs, planner not coupled to catalog

Cons: higher startup cost (have to implement catalog)

Catalog Format





What we did:

- 2 simple built in catalogs (memory + file based / Hive-style partitioning)
- APIs for others

Why:

- Catalog format is very usecase / system dependent
- Nothing we could have built in would likely work well

Listing Table Catalog

```sql
SELECT …
FROM table2
WHERE date='2024=06-02'
```

SQL Query

```
table1
├── file1.parquet
└── file2.parquet

table2
├── date=2024-06-01
│   ├── file3.parquet
│   └── file4.parquet
└── date=2024-06-02
    └── file5.parquet
```

Filesystem directory structure

Tables:

- Directories of files
- “Standard” hive-style directory partitioning

Catalog Format - Lessons Learned

- Providing basic “get started” implementation and extension APIs worked great
  - Let people start quickly, but customize as needed
- ListingTable (directory Filesystem):
  - More complicated than expected
  - Should have had a cleaner separation from the start

Planning?

SQL Dialects

```sql
SELECT
    age,
    sum(civility) AS total_civility
FROM star_wars_universe
GROUP BY ALL
ORDER BY ALL;
```

[*Friendlier SQL with DuckDB*]

Option 1: Implement your own

Pros: have exactly the semantics you want, friendlier language

Cons: It is a lot of work (implementation *and* education)

```sql
SELECT
    age,
    sum(civility) AS
total_civility
FROM star_wars_universe
GROUP BY age
ORDER BY age, total_civility;
```

"Standard" SQL

Option 2: Use existing dialect

Pros: Avoid having to define semantics

Cons: Bug for bug compatible. Have to pick one dialect

SQL Dialects

What we did:

- Emulate postgres semantics default
- Extension APIs

Why:

- Well understood
- It is time consuming / expensive to invent semantics

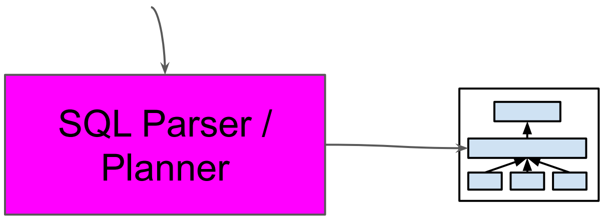
SQL Dialects - Lessons Learned

- Dialect syntax only part of the story. Others:
    - Function Library and semantics (e.g `null` or error on invalid args?)
    - DataTypes (`VARCHAR2`? `CHAR` / `VARCHAR`?)
    - Type Coercion Rules
- No one ideal choice: Spark Dialect vs Postgres (A: UDFs!)
- Postgres + Arrow timestamp representation impedance mismatch
- On the whole this was still a good idea

SQL Planner

```sql
SELECT status, COUNT(1)
FROM http_api_requests_total
WHERE ...
```

SQL Text



LogicalPlan

```sql
SELECT status, COUNT(1)
FROM http_api_requests_total
WHERE ...
```

SQL Text

LogicalPlan

Option 1: Implement sql parser / planner

Pros: Minimize dependencies, native integration into plan structures / exprs

Cons: Much more work

Option 2: Calcite

Pros: Mature

Cons: Java (dependencies), bridge plan representation to internal representation

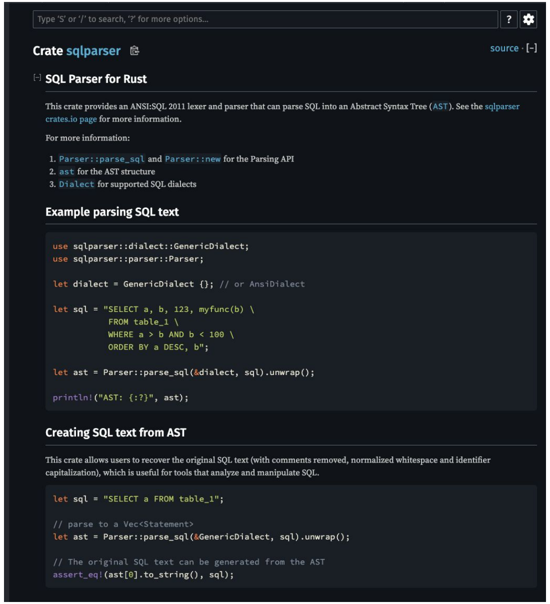
SQL Planner

What we did

- Implemented own sqlparser-rs and planner in Rust
- + extensions

Why:

- Avoid Java dependencies / have a pure Rust stack



Type 'S' or '/' to search, '?' for more options...

Crate sqlparser

source · [−]

SQL Parser for Rust

This crate provides an ANSI:SQL 2011 lexer and parser that can parse SQL into an Abstract Syntax Tree (AST). See the sqlparser crates.io page for more information.

For more information:

1. Parser::parse_sql and Parser::new for the Parsing API
2. ast for the AST structure
3. Dialect for supported SQL dialects

Example parsing SQL text

```
use sqlparser::dialect::GenericDialect;
use sqlparser::parser::Parser;

let dialect = GenericDialect {}; // or AnsiDialect

let sql = "SELECT a, b, 123, myfunc(b) \
           FROM table_1 \
           WHERE a > b AND b < 100 \
           ORDER BY a DESC, b";

let ast = Parser::parse_sql(&dialect, sql).unwrap();

println!("AST: {:?}", ast);
```

Creating SQL text from AST

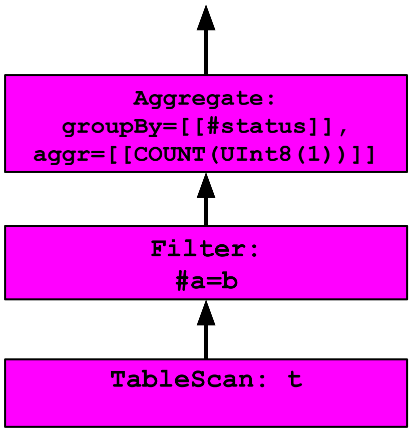
This crate allows users to recover the original SQL text (with comments removed, normalized whitespace and identifier capitalization), which is useful for tools that analyze and manipulate SQL.

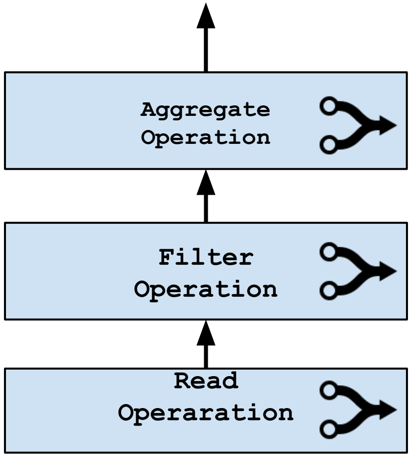
```
let sql = "SELECT a FROM table_1";

// parse to a Vec<Statement>
let ast = Parser::parse_sql(&GenericDialect, sql).unwrap();

// The original SQL text can be generated from the AST
assert_eq!(ast[0].to_string(), sql);
```

SQL Planner - Lessons Learned

- Implementing a SQL planner is a LOT of work
  - SQL is a crazy creole language
- Modularity helped
  - sqlparser-rs has a clean split from DataFusion
  - Means it is used by many projects, and thus benefits from larger community
- Would recommend avoiding this if you can

Plan Representation / API





Option 1: Custom Structures

Pros: Native APIs, make it ergnonomic

Cons: (very) large API surface area + code to maintain, have to define semantics precisely

Option 2: Use existing library

Pros: Code is simple, planning is predictable

Cons: Limited to whatever is available

Plan Representation / API

What we did:

- Custom structures and API
- + Extension APIs

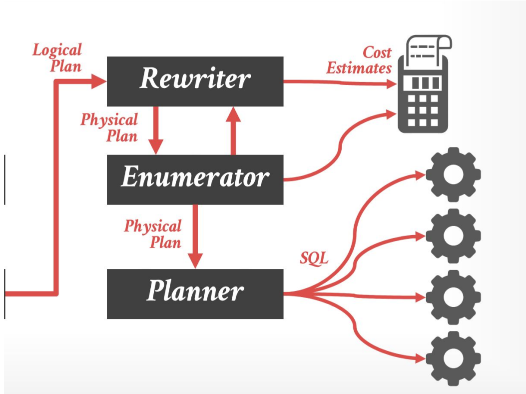
Why:

- No compelling alternative

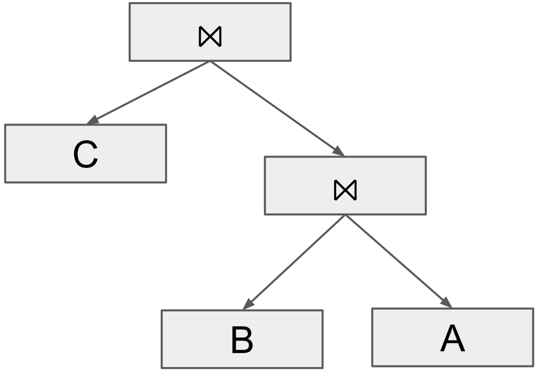
Plan Representation / API - Lessons Learned

- It is a lot of code and API design
- TreeNode API is quite cool (unified Expr/Plan walking)
- Custom serialization takes takes lots of time
- Would / should have used [substrate]() if it was ready
- Should have used rewrite in place APIs for performance reasons

Cost Based / Stratified / Unified / Join Order Optimizer




```
SELECT ...
FROM
   A
   JOIN B ON ...
   JOIN C ON ...
```



Option 1: CBO + Heuristics

Pros: Well understood pattern in DBs

Cons: Known hard problem: cost estimates, cardinality estimates, correlations, performance cliffs, etc

Option 2: “Syntactic” optimizer (whatever order user tells you)

Pros: Code is simple, planning is predictable

Cons: Complex join orders ⇒ 😦

Cost Based / Stratified / Unified / Join Order Optimizer

What we did:

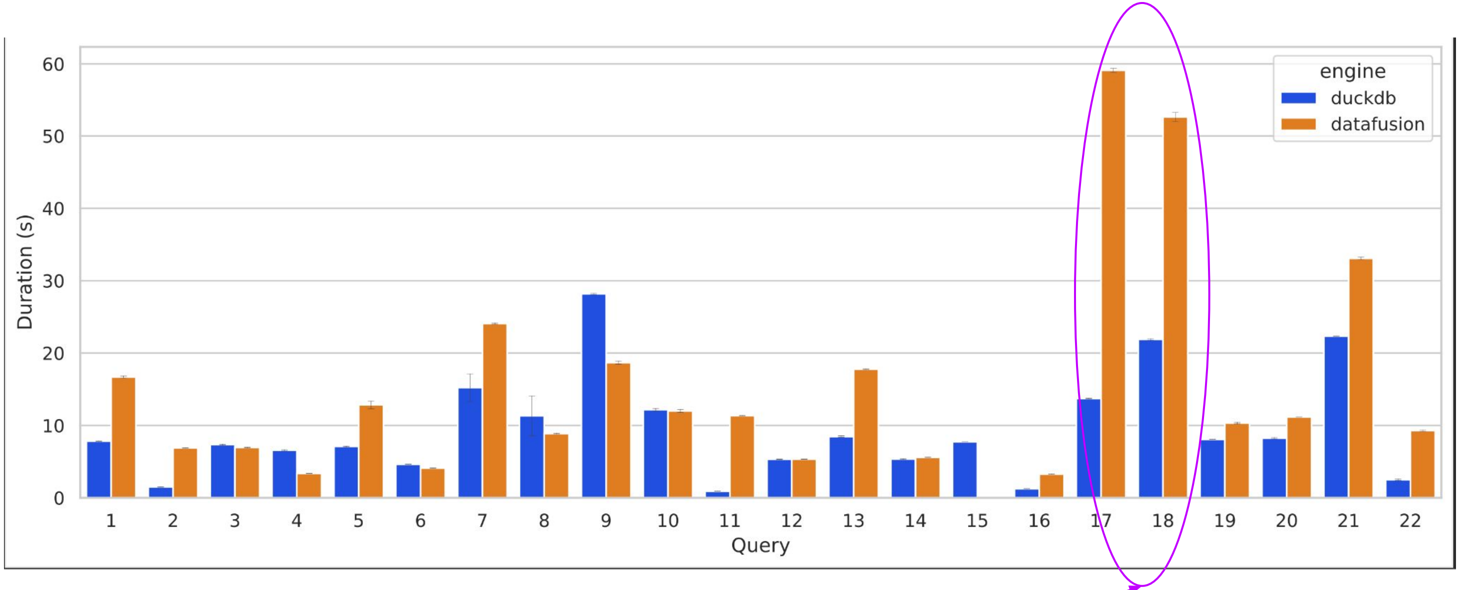
- Syntactic optimizer
- + Just enough to avoid TPCH disasters

Why:

- Handling complex join reordering is hard (both theoretically and practically): Depends a lot on cost model + accurate statistics (also hard)
- Denormalized tables very common in olap workloads (so join order relatively less important)
- Users can implement more sophisticated strategies as rewrites

Cost Based Optimizer: Lessons Learned

- Worked OK (Some TPCH embarrassments)



Join order disaster (subquery cardinality estimation, fixed in #7949)

Cost Based / Stratified / Unified / Join Order Optimizer

- Example of implement Join Ordering as user defined rule

“Currently, optd is integrated into Apache Arrow Datafusion as a physical optimizer. It receives the logical plan from Datafusion, implements various physical optimizations (e.g., determining the join order), and subsequently converts it back into the Datafusion physical plan for execution.”

https://github.com/cmu-db/optd

Conclusion 😅

Conclusion and Takeaways

- Analytic Systems take a lot of work
- Rust and Apache {Arrow, Parquet, DataFusion} are awesome
- Reusing open building blocks saved lots of effort
  - Not free: contributed a lot back to help make them better
- Basic implementations + Extension APIs: kept core “simple”
  - Same APIs for Built in and User Defined
  - Forces the API to be complete / no special casing (e.g. UDFs)

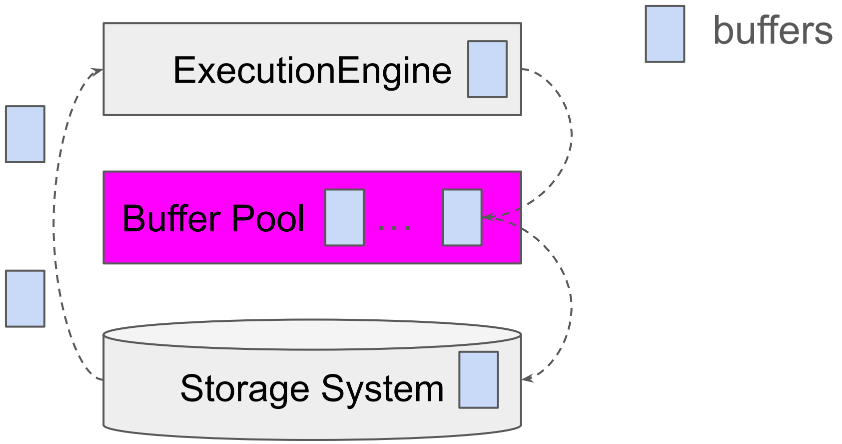
Thank you

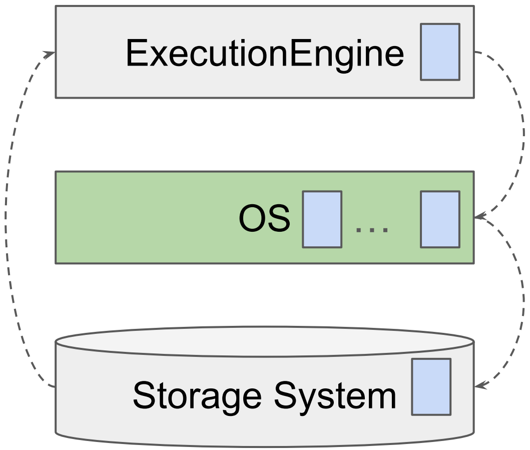
APACHE DATAFUSION™

Come join us!

https://datafusion.apache.org/

Buffer Pool





Option 1: Custom Bufferpool

Pros: Well understood pattern in DBs

Cons: Significant system complexity, have to manage memory distribution between I/O, execution, etc. Tune pool to workload

Option 2: OS Alloc + Page Cache

Pros: No code to manage

Cons: Beholden to OS, Potential MMAP 💩 situations

Buffer Pool

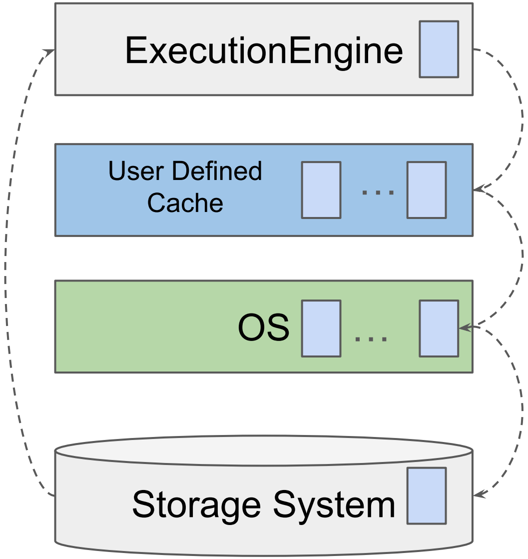
What we did: Use OS + User defined cache

Why: Simple

- Optimal Caching strategy almost always highly dependent on system / environment
- Can implement caching strategies (aka buffer pool) via extensions

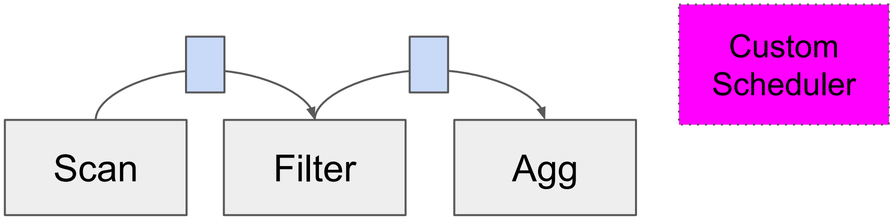
Lessons Learned:

- This has worked very well – basic implementation is easy to understand and very predictable

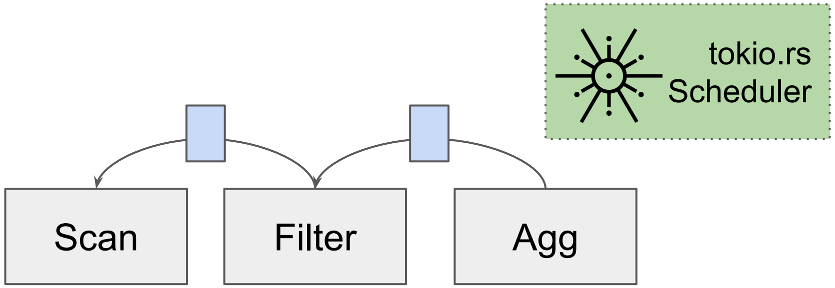


Backup Content

Execution Engine Scheduler



Scheduler "pushes" blocks from scan through plan



Scheduler "pulls" blocks from scan through plan

Option 1: Write own (push based) scheduler

Pros: Tight control over behavior, prioritization, etc

Cons: Very hard to write correctly and tune well, especially under load, network backpressure, etc

Option 2: Use tokio scheduler (pull)

Pros: Someone else writes scheduler and tools, integrated IO + CPU (tokio) patterns, already present in many rust apps

Cons: Less control

Execution Engine Scheduler

What we did: Used Tokio + Futures + Rust async continuations

Why:

- Super well tested, built in compiler language support and tools
- Didn’t really have budget to make our own

Lessons Learned:

- I would do it again (though I will admit others are not convinced)
- Performance turned out to be no different than custom scheduler
- TUM papers are convincing (esp with DuckDB’s story)
- Easy to run IO and CPU on the same pool, so care is warranted (though I would argue it was good we didn’t need to worry about this until it actually mattered for our scale)

More details in https://thenewstack.io/using-rustlangs-async-tokio-runtime-for-cpu-bound-tasks/

Execution Engine Scheduler: Morsels?

We actually tried to implement a push based (morsel driven) scheduler

“the rayon-based [push based] scheduler in its current incomplete incarnation provides minimal benefits over the current tokio-based approach, and is a non-trivial amount of fairly complex code.” - removal PR

More details in the Epic

Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis* Peter Boncz† Alfons Kemper* Thomas Neumann*

* Technische Universität München † CWI

* {leis,kemper,neumann}@in.tum.de † p.boncz@cwi.nl

ABSTRACT

With modern computer architecture evolving, two problems conspire against the state-of-the-art approaches in parallel query execution: (i) to take advantage of many-cores, all query work must be distributed evenly among (soon) hundreds of threads in order to achieve good speedup, yet (ii) dividing the work evenly is difficult even with accurate data statistics due to the complexity of modern out-of-order cores. As a result, the existing approaches for “plan-driven” parallelism run into load balancing and context-switching bottlenecks, and therefore no longer scale. A third problem faced by many-core architectures is the decentralization of memory controllers, which leads to Non-Uniform Memory Access (NUMA).

In response, we present the “morsel-driven” query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data (“morsels”) and schedules these to worker threads that run entire operator pipelines until the next pipeline breaker. The degree of parallelism is not baked into the plan but can *elastically* change during query execution, so the dispatcher can react to execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Further, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of executions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute performance and an average speedup of over 30 with 32 cores.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

Morsel-driven parallelism; NUMA-awareness

1. INTRODUCTION

The main impetus of hardware performance improvement nowadays comes from increasing multi-core parallelism rather than from speeding up single-threaded performance [2]. By SIGMOD 2014

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
ACM 978-1-4503-2376-5/14/06.
http://dx.doi.org/10.1145/2588555.2610507 .

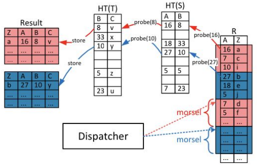


Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

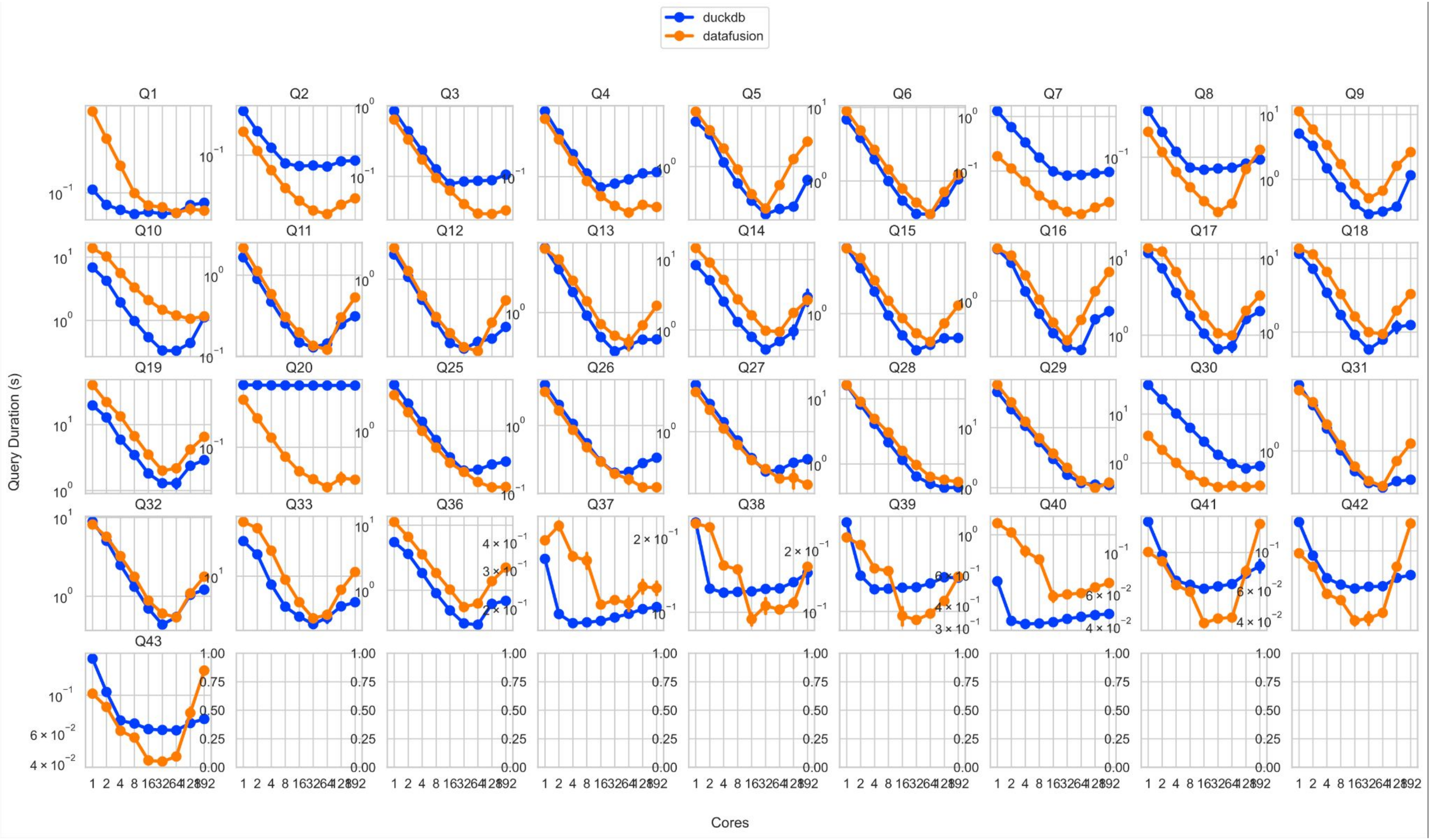
Intel’s forthcoming mainstream server Ivy Bridge EX, which can run 120 concurrent threads, will be available. We use the term *many-core* for such architectures with tens or hundreds of cores.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of memory access, which was needed to scale throughput to huge memories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the accessing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA division of the RAM has to be considered carefully to ensure that threads work (mostly) on NUMA-local data.

Abundant research in the 1990s into parallel processing led the majority of database systems to adopt a form of parallelism inspired by the Volcano [12] model, where operators are kept largely unaware of parallelism. Parallelism is encapsulated by so-called “exchange” operators that route tuple streams between multiple threads each executing identical pipelined segments of the query plan. Such implementations of the Volcano model can be called *plan-driven*: the optimizer statically determines at query compile-time how many threads should run, instantiates one query operator plan for each thread, and connects these with exchange operators.

In this paper we present the adaptive *morsel-driven* query execution framework, which we designed for our main-memory database system HyPer [16]. Our approach is sketched in Figure 1 for the three-way-join query $R \bowtie_A S \bowtie_B T$. Parallelism is achieved

Paper: Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age



ClickBench Scaling of DataFusion (orange) vs DuckDB (blue) 1 - 172 cores ⇒ shapes are very similar

More details in our SIGMOD paper: https://s.apache.org/datafusion-sigmod-2024

Governance

THE APACHE SOFTWARE FOUNDATION

OpenJS Foundation

THE LINUX FOUNDATION

CLOUD NATIVE COMPUTING FOUNDATION

"Benevolent Dictator for Life", "Lone Maintainer", etc

Foundation

Option 1: Custom Governance

Pros: Very flexible, custom tailored doesn't slow down

Cons: Can be non trivial to setup, unclear governance long term sustanaibility

Option 2: Open, existing system

Pros: Predictable well understood governance

Cons: Often slower / not ideal process, may be hard to join

Governance

What we did: Open Governance via Apache Software Foundation

Why:

- Allows contributors from many companies
- Makes governance clear and predictable, especially growing capacity

Lessons Learned

- Totally open / public community takes getting used to
- Works shockingly well
- Not the fastest way of making decisions

* Full Disclosure: I am an ASF foundation member (via work on Arrow + Datafusion)

Architecture / Roadmap

Cathedral

Option 1: Tight Control

Pros: Unified architecture, coherent APIs, decided by a small group of people

Cons: scaling up, limits contributors

Bazzar

Option 2: Accept anything

Pros: Can draw from many people

Cons: Can end up with frankenstein APIs, half finished features

Eric Raymond: https://en.wikipedia.org/wiki/The_Cathedral_and_the_Bazaar

Architecture / Roadmap

What we did: default accept features

- Datafusion-contrib: outlet for people to contribute without as many constraints / still have community

Why:

- Encourages growth and contribution as people feel control over direction

Lessons Learned

- Shockingly effective
- Insist on tests for all PRs: likely more important than the actual code
- Code quality can vary (which is ok to clean it up over time, with tests)
- Need to invest in tracking follow on work with tickets

Architecture / Roadmap

Example contributor driven features

- Predicate pushdown into TableProvider - Thanks @returnString
- Window functions – Thanks @jimexist
- COPY statement, parallelized parquet writer – Thanks @devinjdangelo
- Parquet bloom filter / data page pruning – Thanks @Ted-Jiang

Examples of half finished things

- Recursive CTEs (then @jonahgao picked it up and brought it over the line)
- Predicate selectivity estimation

But quality?

The New York Times

Boaty McBoatface: What You Get When You Let the Internet Decide

Share full article

A computer image of the research vessel, which is still being designed and is scheduled to set sail in 2019. The Natural Environment Research Council

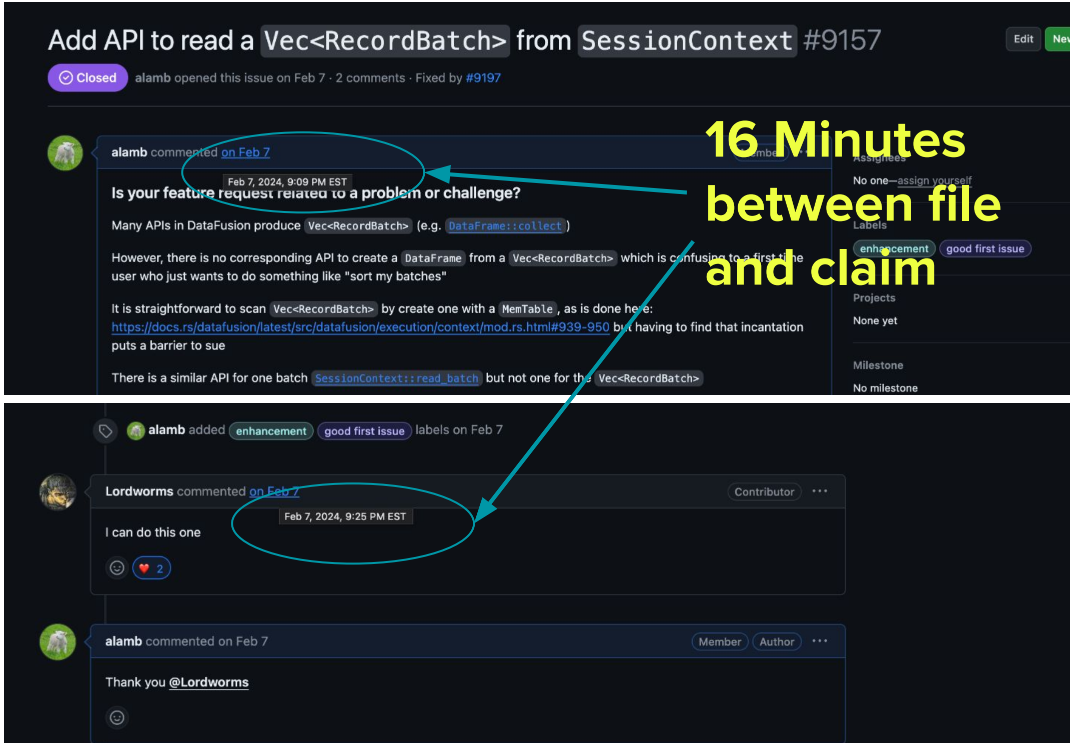
“Originally suggested by former BBC radio presenter James Hand, by the end of the poll on 16 April Boaty McBoatface had garnered 124,109 votes and 33% of the total vote.”

⇒ “Science Minister Jo Johnson said there were "more suitable" names.”

But quality?

- Quality is quite high
- Bugs and regressions do get introduced
- But fixed almost as soon as they are filed (often by the original author)
- I believe quality is better than many of the enterprise software systems I have worked on

Example: Shocking effectiveness of tickets



🤯: File ticket and often there is a PR up within 24 hours to solve it

⇒ “file clear requests” and people want to help

It does take non-coding effort

https://github.com/apache/arrow-datafusion/issues/9157

Make the engine Distributed Engine

What we did instead: Focused on single core, features to build distributed engine

Plan Serialization + use arrow flight

Why:

Distribution strategy varies greatly on usecase

All can be built with a single core

Had other projects (like Ballista) that added distribution

Lessons Learned:

Optimized Comparisons

Why: optimize multi column sort

Add RowFormat to Arrow (see Blog)

Used in merge, multi-column grouping and join comparisons

Lessons Learned:

- Dictionary interning was a bad idea (memory exploded)
-

Manually vectorized kernels

What we did instead: Use autovectoruixation in Rust

Why: Partly Rust's SIMD intrincs story wasn't ideal, but we found we could often get better results with careful rust code than with custom kernels

Lessons Learned:

-

Optimized Hash Aggregates

Why: aggreagation key

What: Two phase hash aggregate (todo blog post link)

Special column based key storage, type specialized, etc

Lessons Learned:

- Need special case code for different types
- Rust auto-vectorization is quite nice

Function Library

Why: Turns out a lot of SQL functionality

What: Massive library (TODO quantify) of functions, all use the same API as user functions

Lessons Learned:

- Had a split initially between built in functions and udfs
- Should have had easier UDFs earlier on
- That way behavior can be customized more easily rather than the core ever growing

Pruning Predicate

Why: Key building block for many opimtizations, inclding fast parquet reader

What: given expression and min/max/nullcounts (and bloom filter info) tell “can this possibly have rows that evaluate to true)?

Lessons Learned:

- API needs to be vectorized (quickly prune large numbers of files)
- Pruning is crazy tricky (because you are trying to prove the null) – testing

Listing Table: Hive style partitioned data

Why: lots of data is in this format (predates table formats)

What: TODO show example directory

Lessons Learned:

- Should have had this outside the core earlier on (as people almost immediately wanted to plug in other file types in there)

Foundations: Apache Parquet

- Open column-oriented data file format,
- Provides efficient data compression and encoding schemes, along with support for structured types via record shredding [56], embedded schema descriptions, zone-map [57] like index structures and Bloom filters for fast data access.

Differences

- Arrow: fast random access and efficient in-memory processing
- Parquet: store large amounts of data in a space-efficient manner.

Why Parquet and not a specialized format?

- De-facto standard for data storage and interchange in the analytic ecosystem.
- Open format, excellent compression across real-world data sets, broad ecosystem and library support, and embedded self-describing schema
- Structure allows query engines (like DataFusion) to apply projection and filter pushdown techniques, such as late materialization, directly on files, yielding competitive performance compared with specialized formats [48].

Foundations: Apache Arrow

Apache Arrow

- Standardizes industrial best practices to represent data in memory using cache-efficient columnar layouts.
- Users avoid re-implementing features that are well understood in academia and industry, but time-consuming to implement.

Specifies

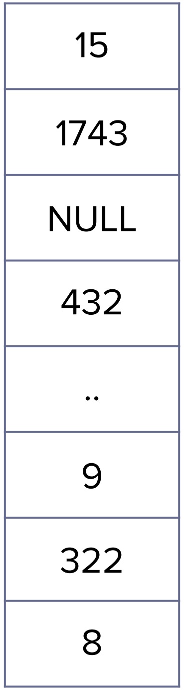
- Validity/null representations; Endianness
- Variable length byte and character data; lists, and nested structures,

Benefits

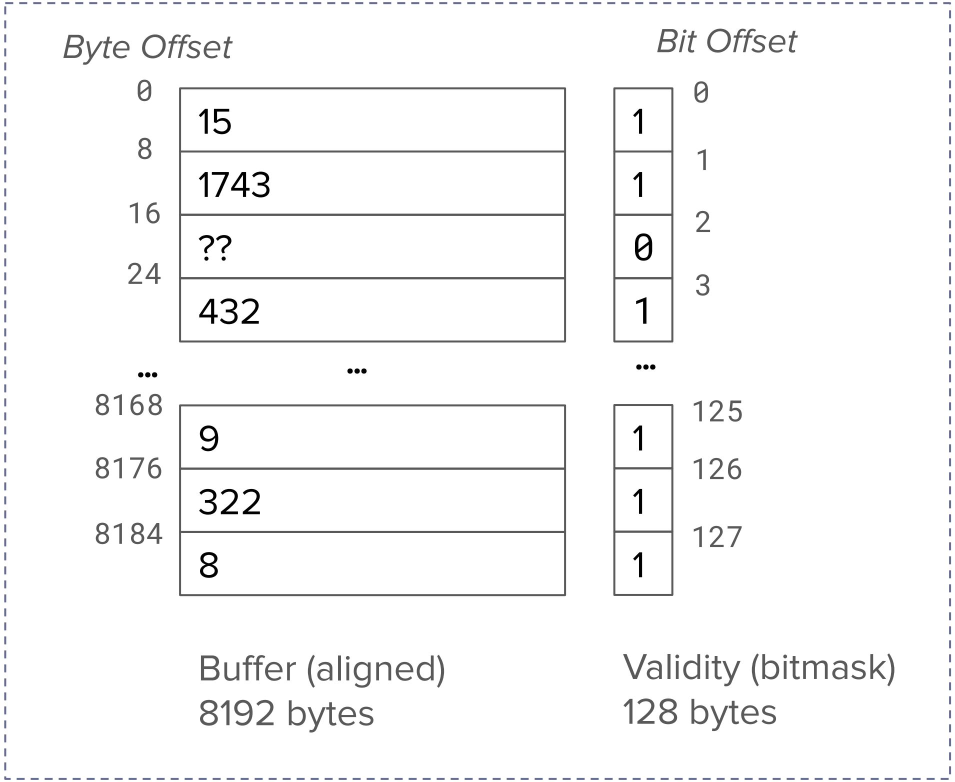
- Well-known techniques (e.g. vectorized compute kernels, special case nulls/no-nulls)
- Easy + zero cost data interchange (e.g. is a NULL value is represented by a 0 or 1 in a bit mask?).
- Arrow evolves over time (e.g. StringView [21] and high-performance compute kernels)

APACHE
ARROW

Arrow Array: Int64Array



Logically 1024
8 byte integers



Arrow Array

APACHE ARROW

Pretty much what you will find in every vectorized column store engine

Why Arrow Internally (and not just at interface)?

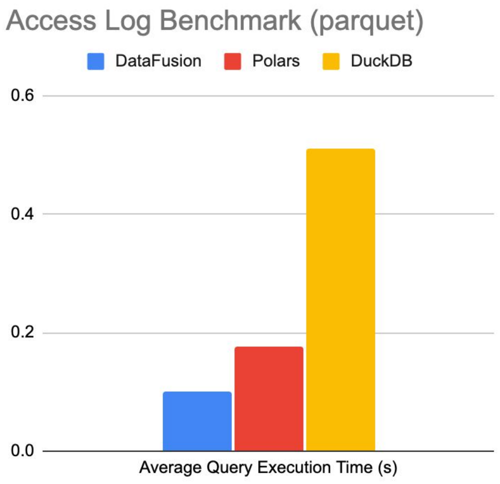
Theory: Using Arrow is "good enough" compared to specialized structures

Pooled open source development → invest heavily in optimized parquet reader

So far results are encouraging

Good: Sorting, Filtering, Projection, Parquet

Could Improve: Grouping, Joining



https://github.com/tustvold/access-log-bench

Foundations: Rust and its Ecosystem

Rust [76]: new system programming language (LLVM based)

1. **Excellent Performance**: similar to C/C++
2. **Easy to Embed**: no language run-time and have C ABI compatibility.
3. **Resource Efficient:** Low level (but safe!) memory management
4. **Productive Ecosystem:** Cargo Package Manager[12] and crate ecosystem make adding DataFusion to most projects as simple as adding a single line to a configuration file.

Physical Segment Trees

What we did instead: Classic sort based optimziations (sorted input by PARTITION BY / ORDER BY clause)

Why: Simple, well understood

Lessons Learned: