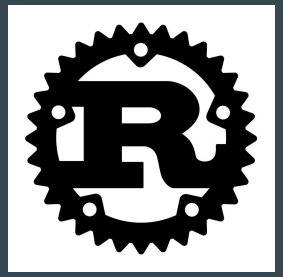


Ballista: Distributed Compute with Rust and Apache Arrow

Andy Grove @ New York Open Statistical Programming Meetup
2/24/2021

About Me

- 30+ years in software
- Started with dBase/Clipper in the late 80's
- C++ developer in the 90's
- Switched to Java in late 90's (Java 1.0)
- Started working with Scala + Spark in 2017
- Started learning Rust seriously in 2018
- Specializing in distributed systems and query engines for ~10 years

Why Ballista?

- Ballista started with my 2018 blog post "Rust is for Big Data"
- I asked *"What if Apache Spark had been implemented in Rust?"*
- I wanted to become more proficient at Rust and learn more about distributed systems and query engines by exploring this possibility.
- This journey has created four projects over the past 3 years:
  - sqlparser-rs (Rust SQL Tokenizer/Parser library)
  - Apache Arrow Rust Implementation
  - Apache Arrow DataFusion query engine
  - Ballista
- All of these projects now have many contributors

Why Arrow?

- Arrow is becoming the de-facto memory and IPC standard for columnar data
- Many popular data projects already use Arrow to some degree
  - Apache Parquet
  - Apache Spark
  - dply
  - Dask
  - Dremio
  - MATLAB
  - Pandas
  - Ray
  - Snowflake
- Building a new platform to be Arrow-native seemed to make a lot of sense
  - No need to invent a new type system and memory format
  - Easier to integrate (efficiently) with other projects in the future

Why Arrow?

- I am not alone in thinking Arrow is an ideal foundation for new data systems
  - Cylon (Multiple universities)
  - Hustle (University of Wisconsin)
  - InfluxDB IOx (InfluxData)
  - NoisePage (CMU)
  - RAPIDS cuDF (NVIDIA)
  - Ursa Computing

Why Rust?

- Pros
  - Similar performance to C++
  - Unique approach to memory management, which guarantees memory safety without the overhead of a garbage collector
    - Smaller memory footprint
    - Faster
    - Predictable / consistent performance (no GC pauses)
  - Excellent tooling
- Cons
  - Steep learning curve for some of the advanced features
  - Language is still maturing and missing some features compared to more mature languages

Apache Arrow

A cross-language development platform for in-memory analytics

Apache Arrow Specification

- Format
  - Defines memory layout for columnar data, for primitive types, lists, maps, and structs
  - Values are stored in contiguous buffers
  - Variable-length types have offset buffers
  - Separate validity bitmap specifies which entries are null or valid
  - Optimized for analytics operations, taking advantage of modern hardware for vectorized processing (e.g. SIMD and GPU)
- IPC
  - IPC format is defined with flatbuffers for exchanging metadata about record batches
  - Data exchange is zero-copy since the memory format is the serialization format
- Flight
  - Flight is a gRPC-based protocol for exchanging queries and Arrow data in distributed systems

Apache Arrow Libraries

- Implementations in many languages
  - C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust
  - Python and R use the C++ library
- Some implementations provide computational “kernels”
  - Including C++, Java, and Rust
- Some implementations are building query engines
  - Including C++, and Rust

Apache Arrow (Rust) Key Data Structures and Concepts

- Data Types
  - Schema, Field, DataType (primitive and complex - structs, maps, lists)
- Arrays & Builders
  - PrimitiveArray, StructArray, DictionaryArray, and others
  - Builders for each type of array
- Compute kernels, including
  - Arithmetic, String, Filter, Aggregate, Cast, Comparison, Boolean
- RecordBatch
  - For representing batches of columnar data with a known schema
- Arrow IO (Readers and Writers)
  - CSV
  - JSON
  - Parquet

Building an array of primitive values

```rust
let mut builder = Int32Array::builder(5);
builder.append_value(0)?;
builder.append_value(2)?;
builder.append_null()?;
builder.append_null()?;
builder.append_value(4)?;
let array = builder.finish();
```

Compute kernel

```
let a = Int32Array::from(vec![5, 6, 7, 8, 9]);
let b = BooleanArray::from(vec![true, false, false, true, false]);
let c = filter(&a, &b)?;
let d = c.as_ref().as_any().downcast_ref::<Int32Array>().expect("downcast failed");
assert_eq!(2, d.len());
assert_eq!(5, d.value(0));
assert_eq!(8, d.value(1));
```

RecordBatch & Downcasting

```rust
#[derive(Clone, Debug)]
pub struct RecordBatch {
    schema: SchemaRef,
    columns: Vec<ArrayRef>,
}
```

```rust
let int_array = array.as_any().downcast_ref::<UInt32Array>()?;
```

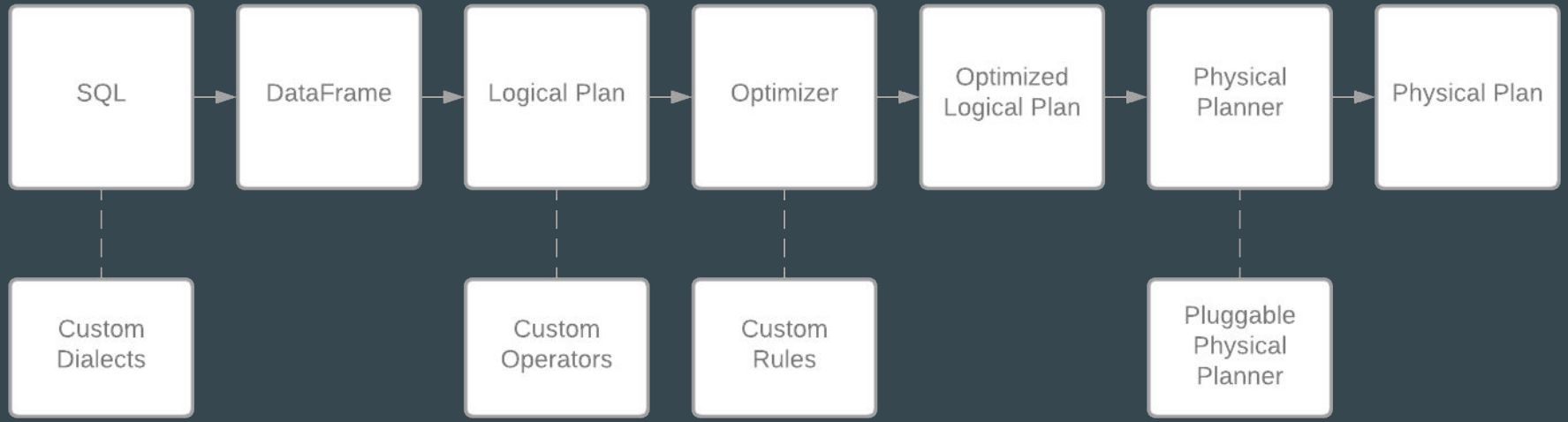
DataFusion

In-memory query engine supporting SQL and DataFrame APIs

What is DataFusion?

- DataFusion is an in-memory query engine that provides both a DataFrame and SQL API for querying CSV, Parquet, and in-memory data.
- DataFusion leverages the Arrow compute kernels for high performance.
- Partitions are processed in parallel using the Tokio threaded runtime

DataFusion Architecture



Supported DataFrame operations

- scan (parquet, csv, json, custom provider)
- select_columns
- select
- filter
- aggregate
- join
- limit
- sort
- collect
- save (parquet, csv, json)
- explain

DataFrame Example

```
// create local execution context
let mut ctx = ExecutionContext::new();

// define the query using the DataFrame trait
let df = ctx
    .read_parquet("/path/to/alltypes_plain.parquet")?
    .select_columns(vec!["id", "bool_col", "timestamp_col"])?
    .filter(col("id").gt(lit(1)))?;

// execute the query
let results = df.collect().await?;
```

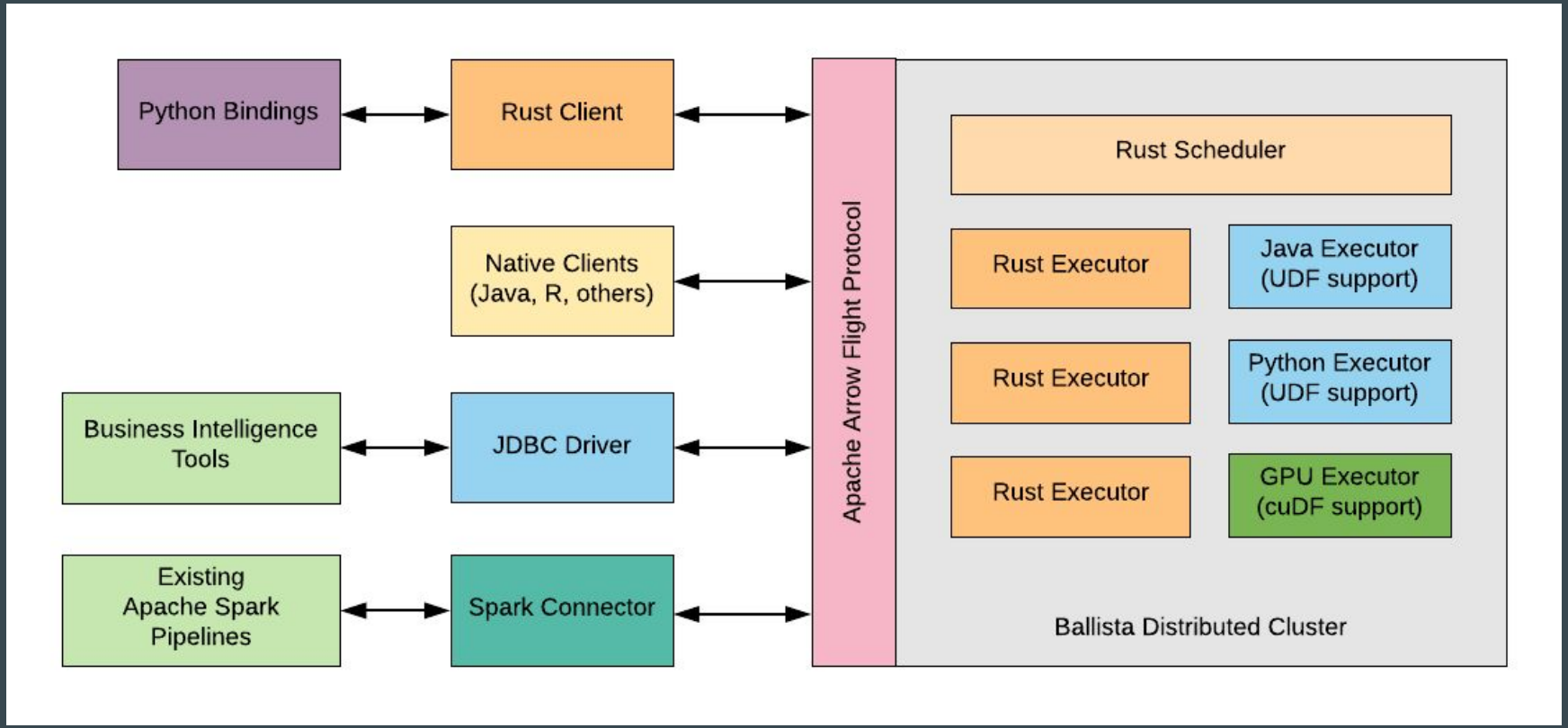
SQL Example

```
let mut ctx = ExecutionContext::new();

// register parquet file with the execution context
ctx.register_parquet(
    "alltypes_plain",
    "/path/to/alltypes_plain.parquet",
)?;

// execute the query
let df = ctx.sql(
    "SELECT int_col, double_col, CAST(date_string_col as VARCHAR) \
    FROM alltypes_plain \
    WHERE id > 1 AND tinyint_col < double_col",
)?;
let results = df.collect().await?;
```

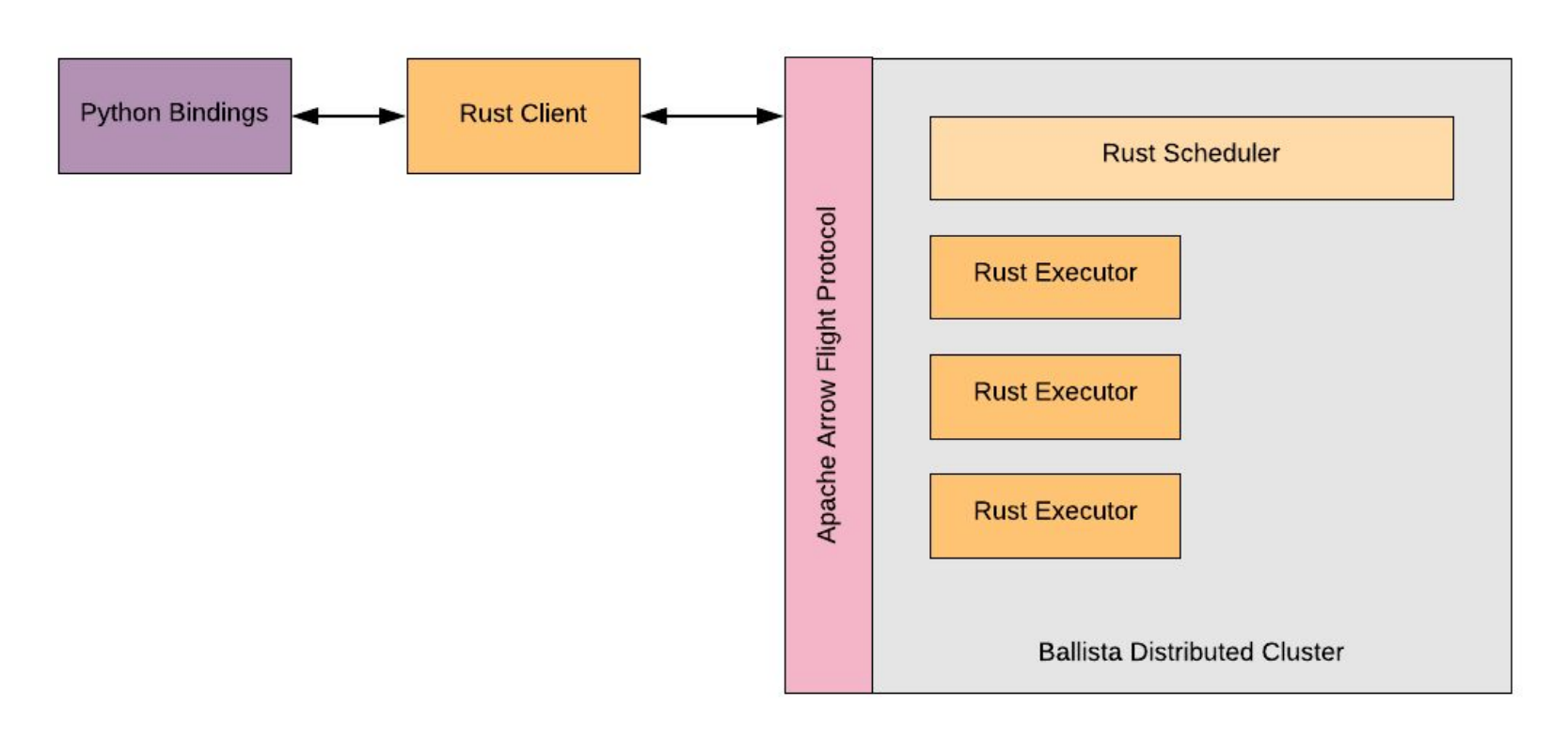
Ballista

Modern distributed compute platform implemented in Rust, and powered by Apache Arrow

Architecture (eventually....)



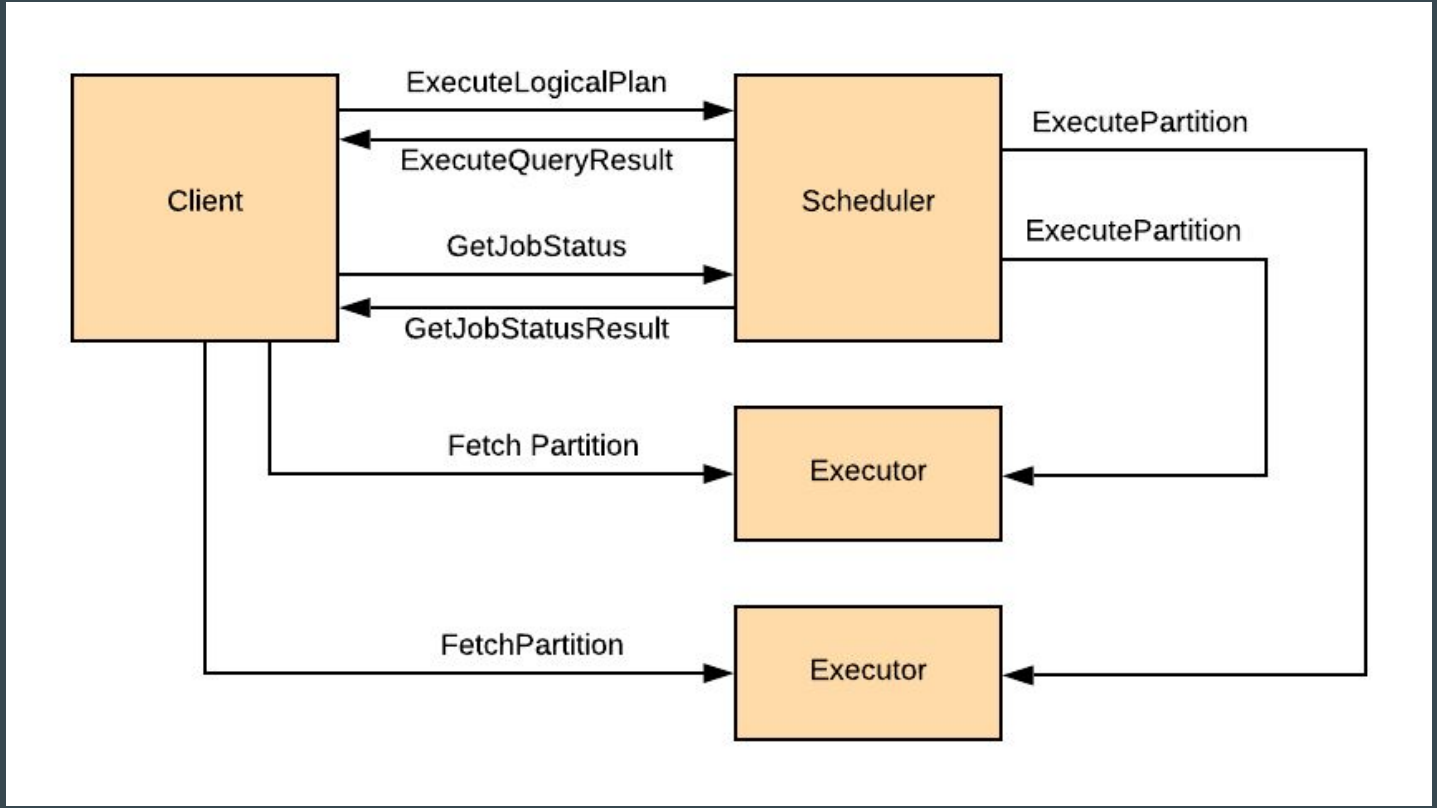
Architecture Today (Ballista 0.4.0)



Distributed Query Planner

- Inspired by Adaptive Query Execution feature in Apache Spark
- Query is broken into stages, where each stage consists of a chain of operators with the same partitioning
  - Query stage execution is “embarrassingly parallel”
- Distributed plan becomes a directionally-acyclic graph (DAG) of query stages
- Statistics are available from each completed query stage
  - This allows reoptimization of the remaining query stages based on these statistics
- Data needs to be “shuffled” around between query stages
  - Data is streamed between executors in Arrow IPC format, with optimizations to read directly from disk when executors are co-located or where shared storage is used

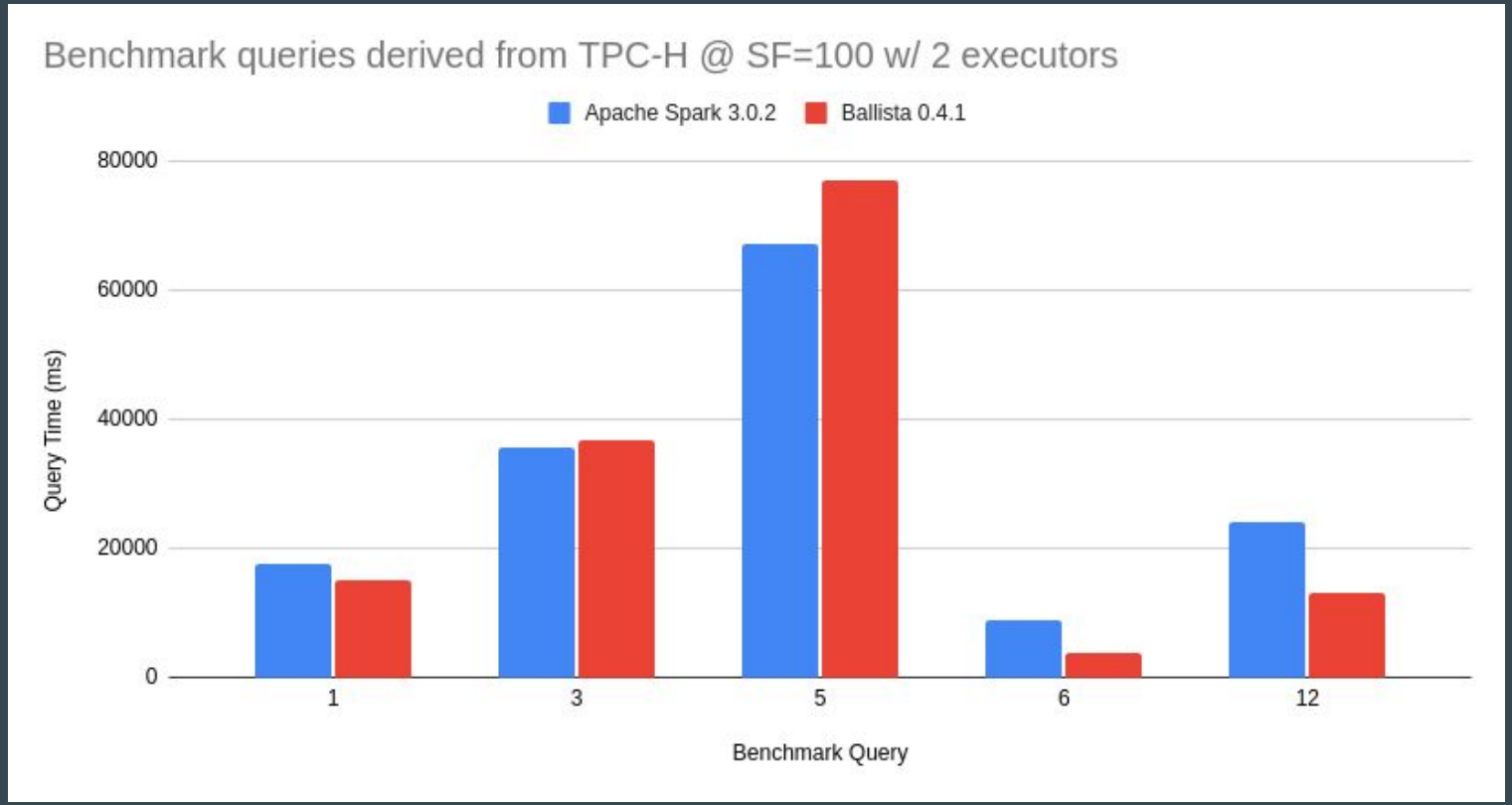
Query Execution Flow



Ballista 0.4.0 (released 2/20/21)

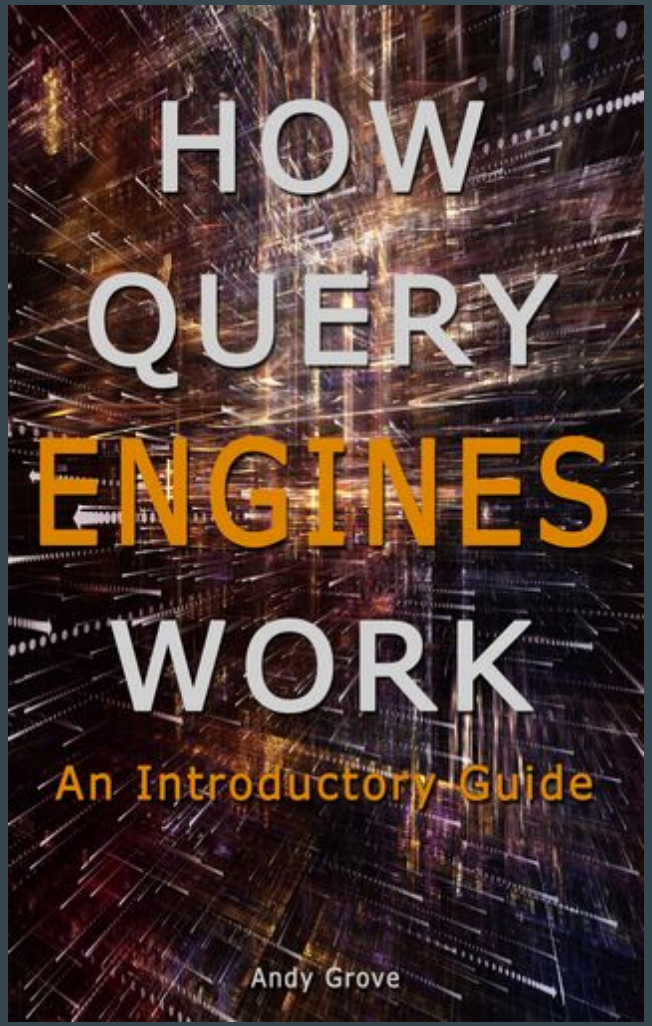
- First truly usable release
- Uses latest Apache Arrow / DataFusion from GitHub main branch
- Supports Kubernetes, Docker Compose and Standalone clusters
- Capable of executing distributed queries using SQL and DataFrame API
- Supports roughly the same operators and expressions as DataFusion
- Capable of running TPC-H queries 1, 3, 5, 6, 10, and 12 so far
- Some queries are 2x the speed of Apache Spark, others are considerably slower
- Current focus is on implementing additional operators and optimizations to make all queries perform well at scale
    - E.g. ShuffleHashJoin, SortMergeJoin, dynamically changing join strategies based on statistics

Relatively fair benchmarks*



Ballista Roadmap

- 0.4.x
  - Incremental improvements to performance, deployment, documentation
- 0.5.0
  - Performance and scalability at least as good as Apache Spark for all supported benchmark queries, at scale factors >= 1 TB
  - Use published version of Apache Arrow
  - JDBC Driver
  - GPU Support (leveraging RAPIDS cuDF)
- 1.0.0 (all the things!)
  - Support multi-language UDFs / UDAFs (Rust, JVM, Python, etc)
  - Web UI in scheduler for monitoring queries and cluster state
  - Apache Spark integrations
  - Notebooks / REPL
  - Support more data sources (S3, HDFS)
  - Metrics, profiling, failover, etc



Have you ever wondered how query engines work?

Have you ever wanted to build your own query engine?

This book provides an introduction to the topic suitable for beginners and walks through all aspects of building a fully working SQL query engine in Kotlin.

https://leanpub.com/how-query-engines-work

Thanks for listening!

- Apache Arrow
  - https://arrow.apache.org/
  - https://github.com/apache/arrow/
- Ballista
  - https://github.com/ballista-compute/ballista
  - @BallistaCompute
- Me
  - @andygrove73
  - https://andygrove.io/
  - https://www.linkedin.com/in/andygrove/