🙌 Graduated to Top Level Project April 2024

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

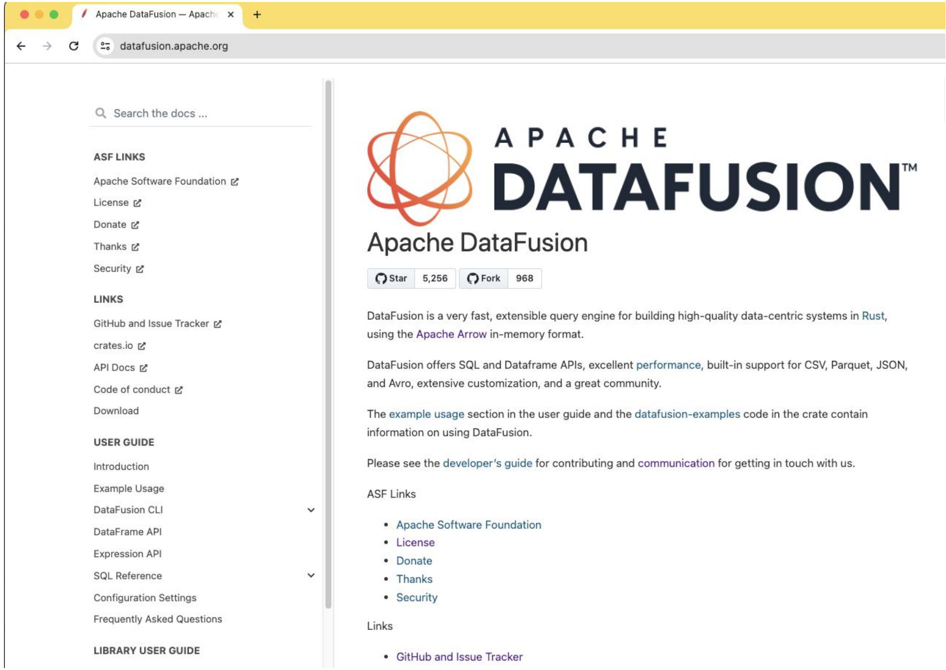
⇒ https://s.apache.org/datafusion-sigmod-2024

TLDR

Thesis: You can build extensible AND high performance query engine

⇒ Most future databases will be built using DataFusion or similar technologies

Goal: Convince you to contribute to DataFusion (have fun and help us make it even better)

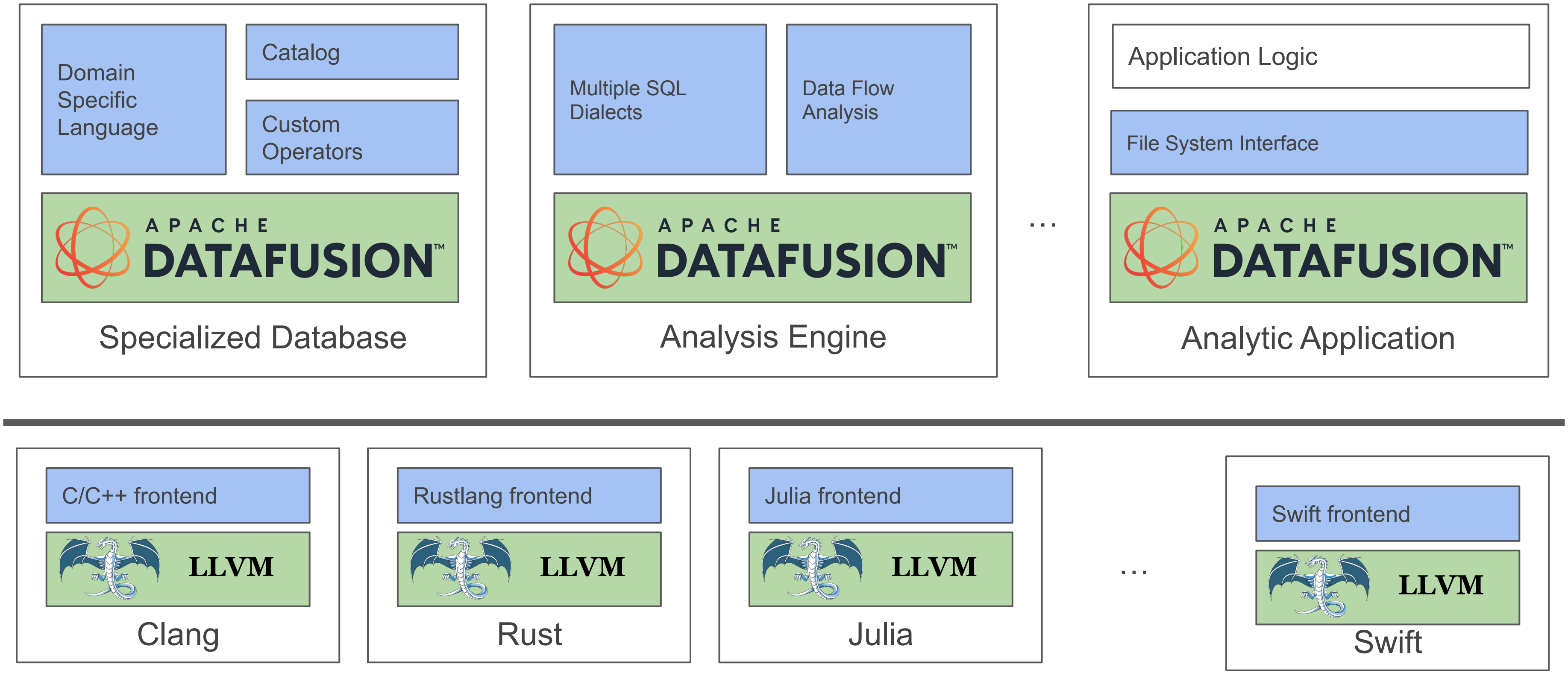


https://datafusion.apache.org/

Outline

- What is DataFusion and why do you need it?
- Feature Highlights
- Performance Analysis

What: DataFusion is LLVM for Databases



Why?

Building a high performance query engine is a lot of (well understood) work

Often requirement but not a differentiator for your project

Ideally focus on what makes your project different, and share costs for the underlying infrastructure

Users + Usecases

1. **Tailored database systems** for domain-specific use cases such as time series databases (e.g. InfluxDB 3.0 and Coralogix) and streaming SQL platforms (e.g. Synnada and Arroyo).
2. **Execution run-times for specialized query front-ends**, such as Comet for Apache Spark, Seafowl for PostgreSQL, Vega, and InfluxQL.
3. **SQL analysis tools** such as dask-sql and SDF
4. **Table formats** such as the Rust implementations of Delta Lake, Apache Iceberg and Lance, (predicate-based delete tombstones, compaction, etc)

But… Performance?

“Traditionally, high-performance analytic query engines ..dominated by tightly integrated systems such as Vertica, Spark, and DuckDB”

“Optimizes interfaces between the file format, in-memory layout, and processing engine to achieve peak performance.”

By now understand appropriate module boundaries that don’t give up performance

Longer term trend is “deconstructed databases” / “composable data systems”

Architecture

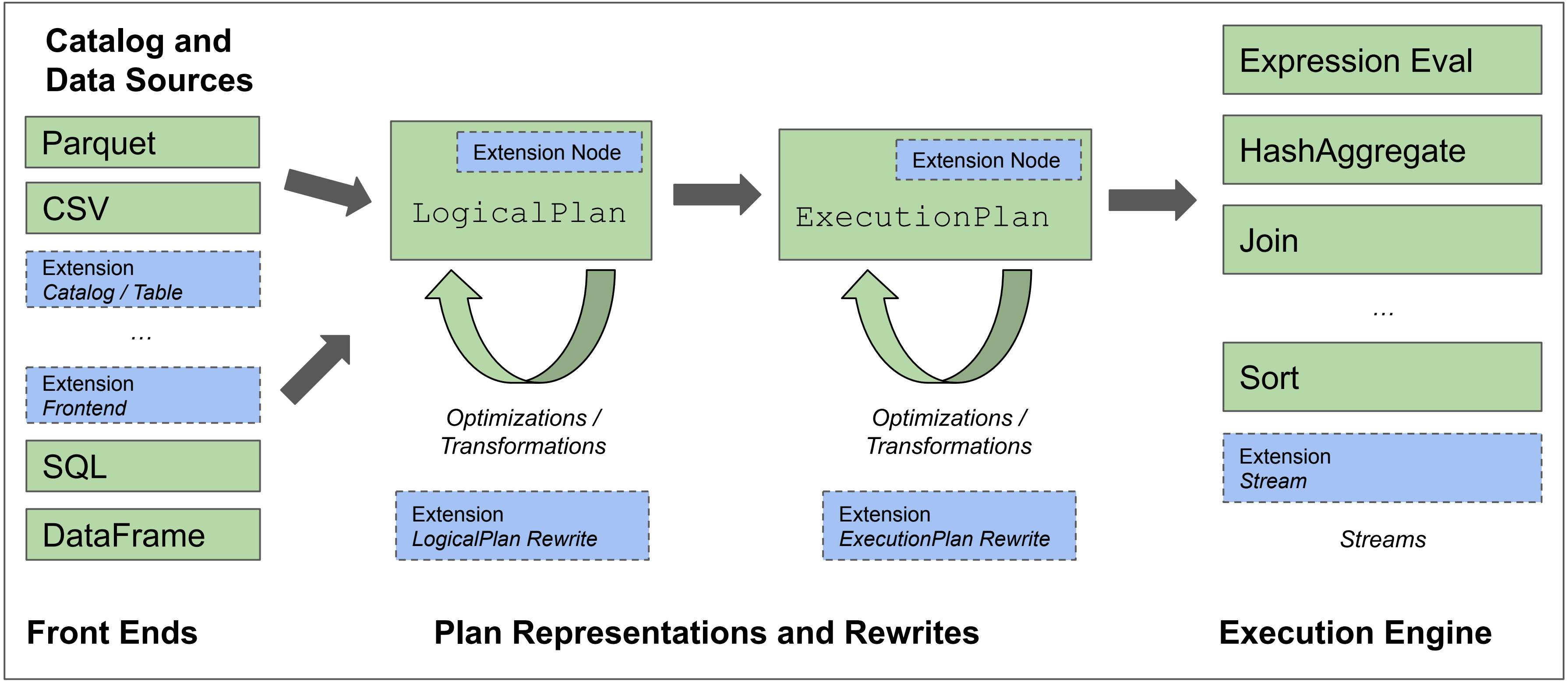
Design Goals: Do the obvious thing

- Work “out of the box”
- Permit customizing everything via API
- Be boring: Follow industrial best practices, informed by research literature, implement well-known patterns

Results

- Users quickly start with a basic, high-performance engine
- Specialize over time to suit their needs and available engineering capacity
- Easy to try out new ideas (operators, rewrites, etc)

Architecture



Feature List

(The point of list is 🤩 – it takes a lot to make a fast query engine)

- LogicalPlans
  - Provided: Expr: Logical expression tree, LogicalPlan: Tree of Relational Operators
  - Provided: APIs to construct, analyze, rewrite, serialize, statistics, boundary analysis
  - Extension: User defined nodes
- Catalog/DataSources
  - Provided: in memory catalog, ListingTable (based on directory of files)
  - Extensions: UserDefined Schemas, Catalogs, TableSources (predicate, projection, limit, pushdown)
- Front Ends
  - Provided: SQL planner ("full" SQL, e.g. can plan TPC-DS)
  - Provided: DataFrame API,
  - Extension: LogicalPlanBuilder to programmatically create plans
- Functions
  - Provided: large library of scalar, aggregate, window functions (strings, datetime, etc).
  - Extension: User defined Scalar, Aggregate, Window and Table Functions (same API as the provided)

Feature List (cont)

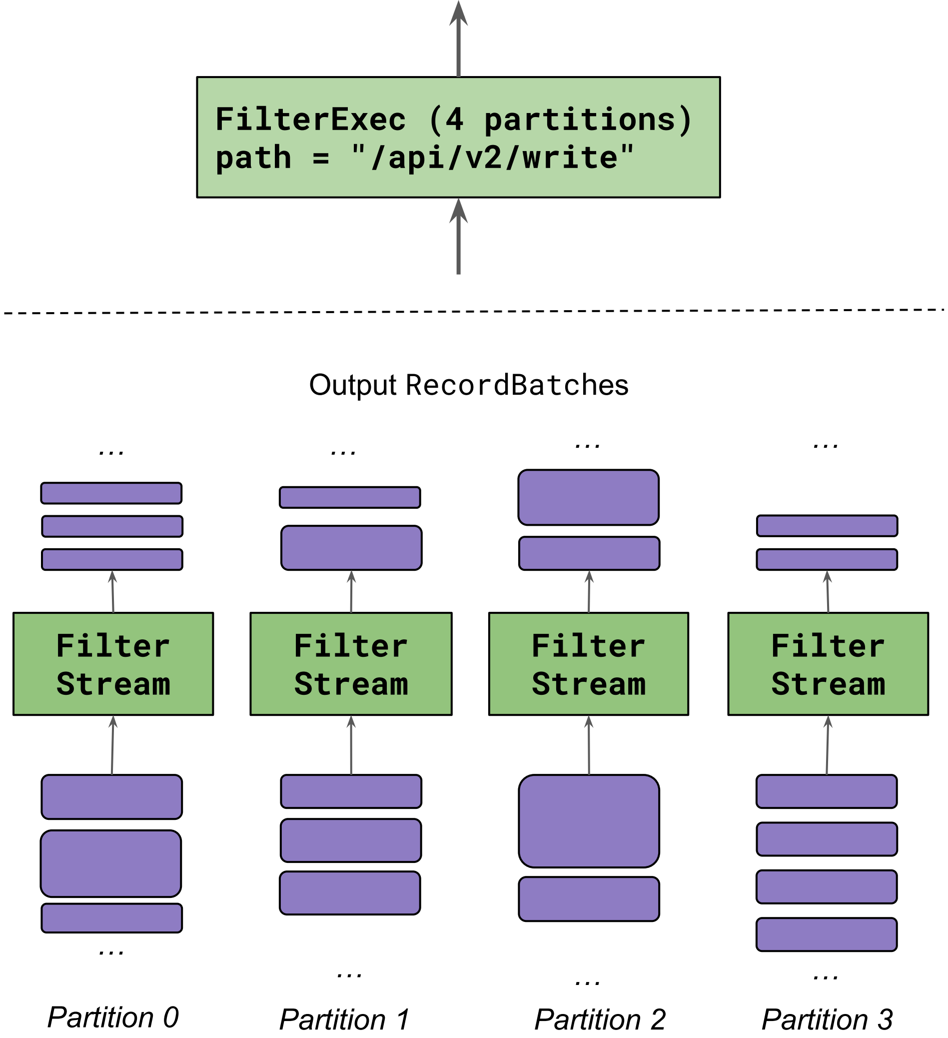
- PhysicalPlans:
  - ExecutionPlan (relational tree, has things like sort, partitioning etc), PhysicalExpr , “Stream” (actual thing that touches data)
  - Extension: User defined ExecutionPlans, exprs and streams
- Operators
  - Provided: all basics like HashJoin, Sort, Merge, Union, Filter, ParquetScan, etc
  - Extension: stream
- Table Sources:
  - Provided: Parquet, Avro, Json, CSV, Arrow (including projection, filter pushdown)
  - Extension: arbitrary formats, indexes, etc.
- Optimizers
  - Provided: 20+ such as type coercion, constant propagation, join flattening, projection/filter/limit pushdown, expression simplification, common subexpression elimination, … , eliminate unnecessary sorts, maximizing parallel execution, …
  - Extension: apply arbitrary plan rewrites

Feature List (cont)

- Execution:
  - **“Volcano Style”:** Pull Based, Exchange
  - **Streaming**: (operate on batches or rows, 8192 by default), except pipeline breakers
  - **Multi-core**: maximum parallelism is set at planning time in exchange operators
  - **Thread scheduling:** Tokio (rust async IO runtime)
  - **Memory Management**: Memory Pools, register large allocations, rest is slop accounted
- Specialized Implementations
  - **Sorting**: spill to temporary disk files when memory is exhausted, specialized implementations for LIMIT (aka "Top K"), normalized Keys (RowFormat)
  - **Grouping**: Two phase parallel vectorized hash aggregations, partial group key sort optimizations
  - **Joins**: Parallel in memory hash join, merge join, Left, Right, Full, LeftSemi, RightSemi, LeftAnti, RightAnti, optimized for equality predicates,
  - **Window Functions**: Sort based window calculations, optimized sliding window implementations (not yet Physical Segment Trees)
  - **Leverage Sort Order**: Use more resource efficient algorithms (like sort aware grouping dump hash table early)
  - **Filter Pushdown / Late Materization**: Filters *during* the scan (e.g. apply filters to ParquetExec Page level statistics), bloom filters, etc

Execution

Streaming, partitioned execution



* yes I know about push style morsel driven parallelism,

* no you don't need it to get good multi-core performance – see results

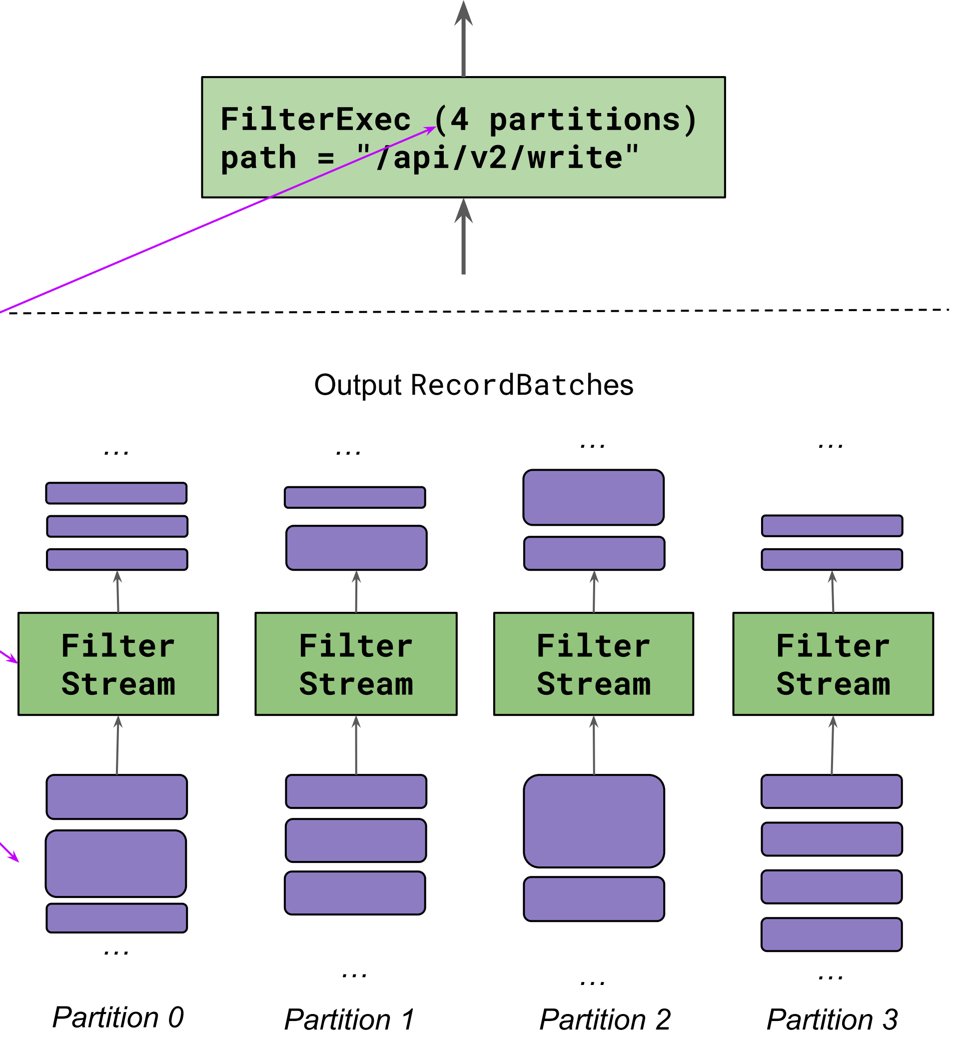
Execution

Streaming, partitioned execution

- ExecutionPlan annotated with a number of partitions by the planner
- A Stream /operator (green) is created for each partition.
- Record Batches (purple) are processed independently on multiple threads by the different operators

* yes I know about push style morsel driven parallelism,

* no you don't need it to get good multi-core performance – see results



Execution: Pull Style in Rust

```rust
impl Stream for MyOperator {
  ...
  // Pull next input (may yield at .await)
  while let Some(batch) = input.next().await {
    // Calculate, do we have output?
    if Some(output) = self.process(batch)? {
      // "Return" RecordBatch to caller
      return Poll::Ready(Ok(Some(output))
    }
  }
  ...
}
```

Execution: Pull Style in Rust

```
impl Stream for MyOperator {
  ...
  // Pull next input (may yield at .await)
  while let Some(batch) = input.next().await {
    // Calculate, do we have output?
    if Some(output) = self.process(batch)? {
      // "Return" RecordBatch to caller
      return Poll::Ready(Ok(Some(output))
    }
  }
  ...
}
```

Cooperative multi-threading

Rust compiler generates continuations

So, how well does this work?

TLDR: As well as tightly integrated approaches

- Compared against DuckDB
- Conclusion: Determining factor is engineering effort rather than architecture

All scripts: https://github.com/JayjeetAtGithub/datafusion-duckdb-benchmark

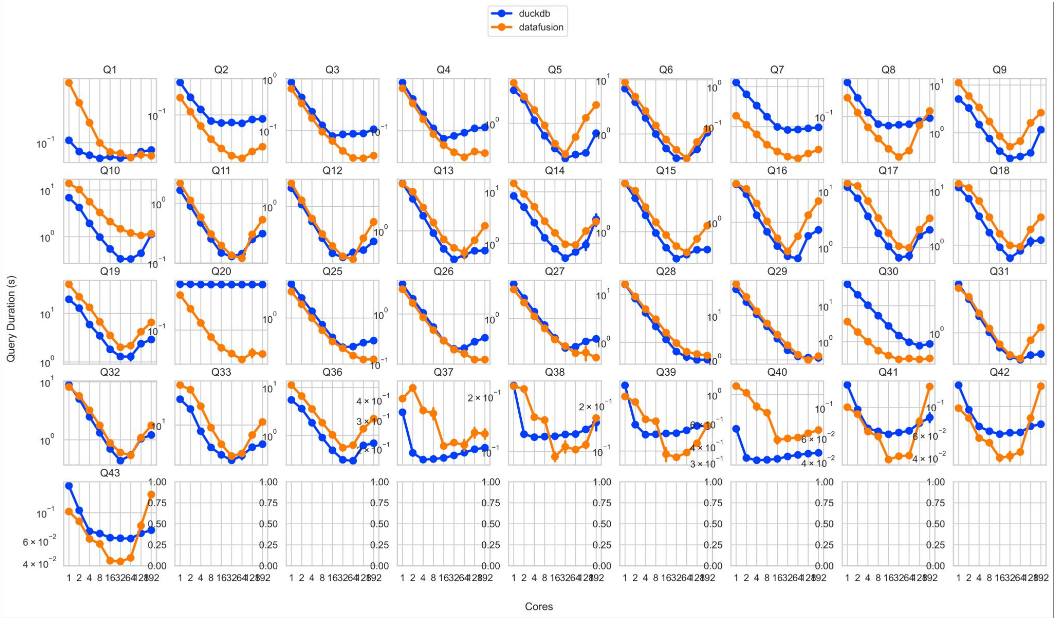
Scaling: ClickBench

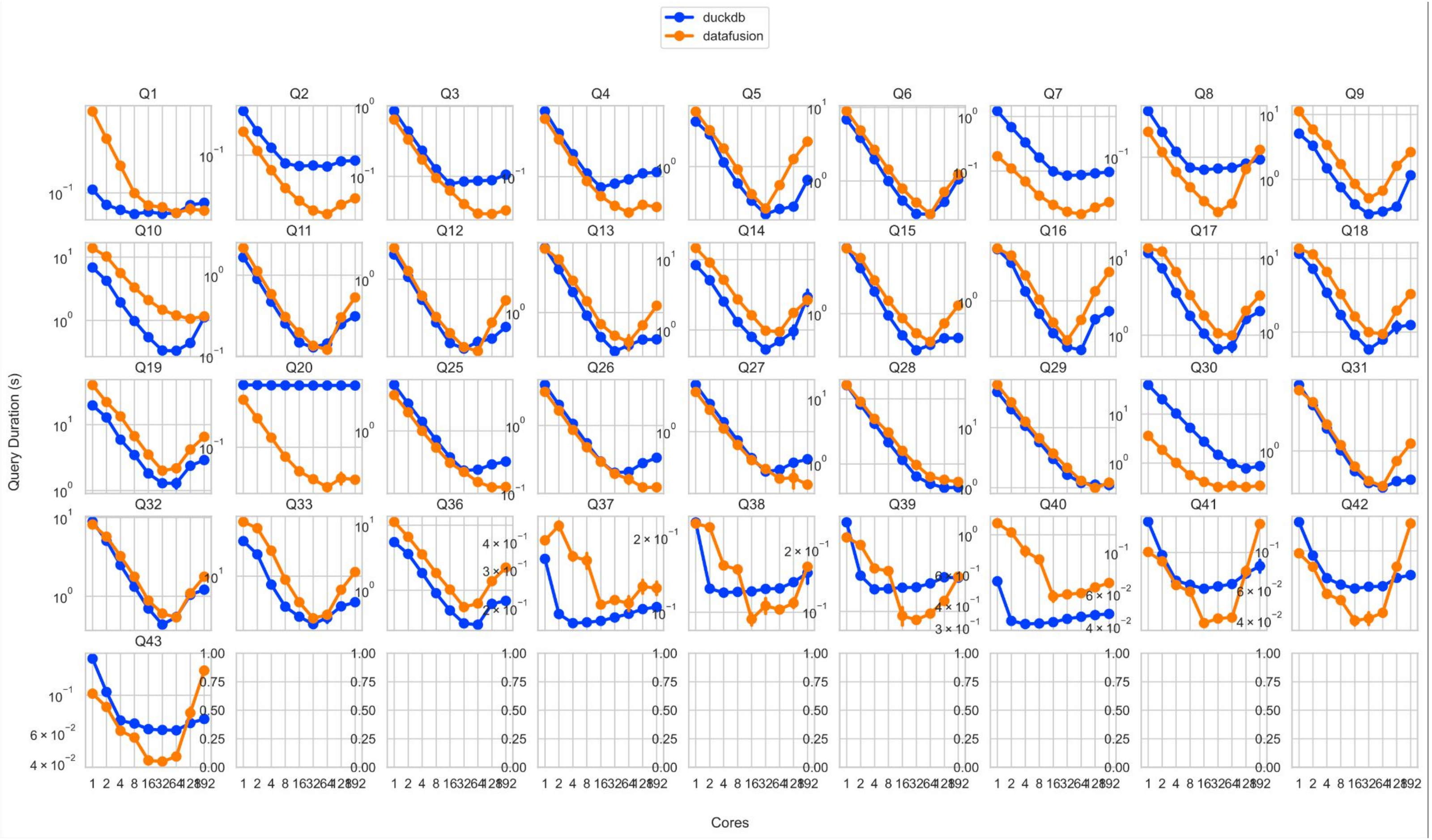
Experimental Setup:

- Google Cloud Platform VM: `c3-highcpu-176`
- Intel Sapphire Rapids micro-architecture, 176 virtual CPUs (cores), and 352 GB RAM
- Ubuntu 22.04 Linux kernel version 6.2.0-1016-gcp

Otherwise the same

Varied cores between 1 and 176



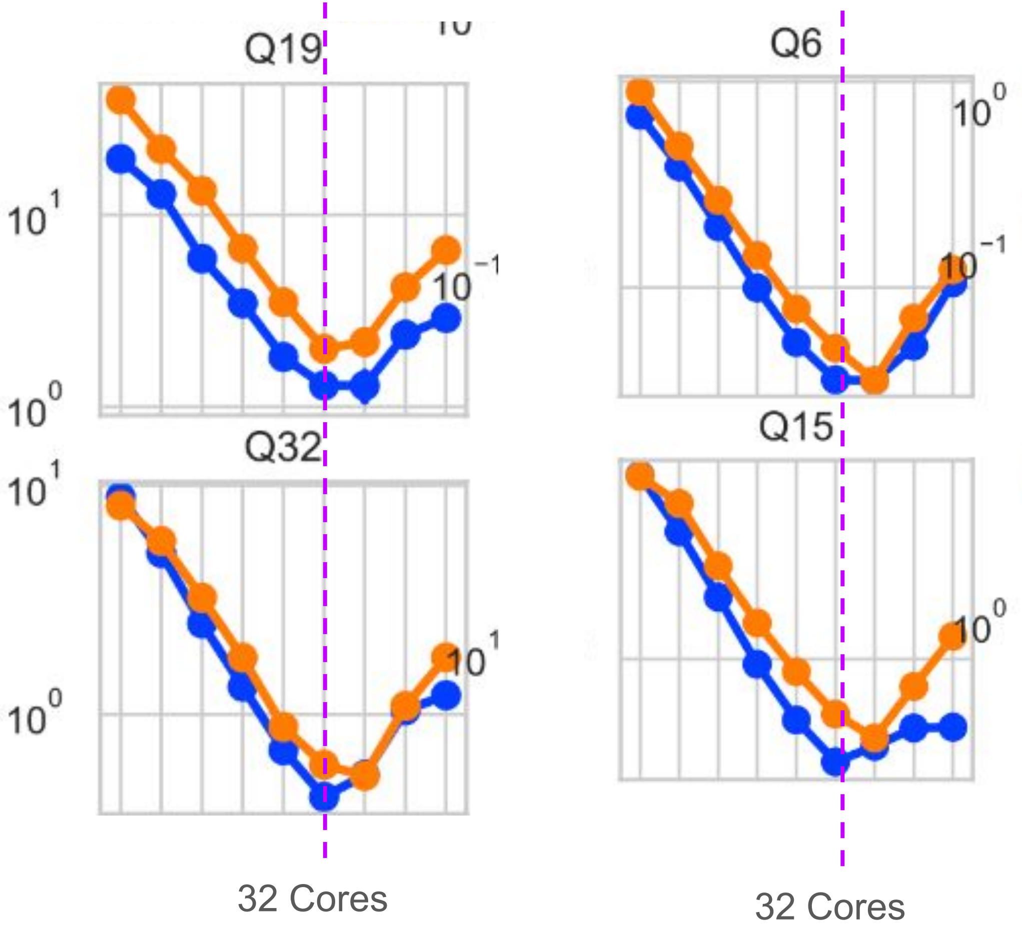


Note: Curves are similar in shape, pull + exchange is ~ the same in practice as morsel driven parallelism

Scaling: ClickBench

At 1, 2, 3, 8, 16, 32 cores

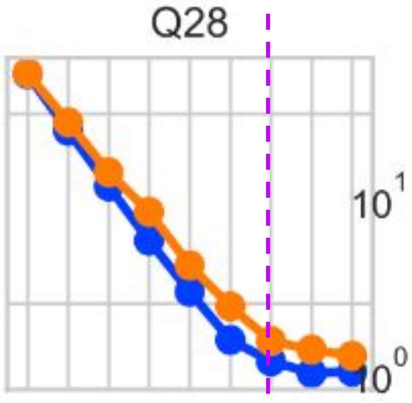
Both systems show near linear scaling



Scaling: ClickBench

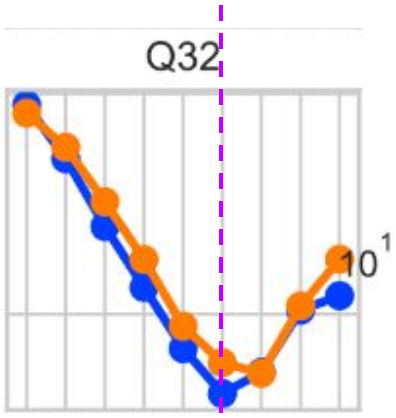
64, 128, 176 cores

Mix of better/worse



Near ideal scaling: Low cardinality aggregation (6000 groups) with expensive (LIKE) predicates

32 Cores



Pronounced increase at higher core counts: 14M Groups

32 Cores

Single Core Efficiency: ClickBench

Experimental setup:

- Google Cloud Platform VM: `e2-standard-8`
- Intel Broadwell CPU, 32 GB of RAM 8 virtual cores
- Ubuntu 22.04.3 LTS / Linux Kernel 6.2.0-1013-gcp

Limited to a single core:

- DataFusion by setting `target_partitions` to 1
- DuckDB by setting the `threads` PRAGMA to 1.

Workload

- Input: 14 GB athena_partitioned dataset (100M Rows)
- 100 Parquet files @ ~140 MB
- Queries: DataFusion Dialect, DuckDB Dialect

Versions

- DataFusion: 32.0.0
- DuckDB: 0.9.1

Query	**DataFusion**	**DuckDB**	Delta
1	1.22	0.18	6.74x slower
2	0.36	0.81	2.25x faster
3	1.11	1.78	1.6x faster
4	1.09	1.5	1.38x faster
5	20.74	8.34	2.49x slower
6	17.81	11.98	1.49x slower
7	0.3	2.08	6.91x faster
8	0.37	0.83	2.24x faster
9	27.91	10.83	2.58x slower
10	25.84	14.11	1.83x slower
11	4.29	3.22	1.33x slower
12	4.67	8.69	1.86x faster
13	11.38	10.27	1.11x slower
14	26.96	14.61	1.84x slower
15	12.7	11.15	1.14x slower
16	13.31	9.12	1.46x slower
17	29.6	21.97	1.35x slower
18	29.09	21.23	1.37x slower
19	92.31	39.1	2.36x slower
20	0.8	1.33	1.65x faster
25	6.01	8.44	1.4x faster
26	5.02	6.11	1.22x faster
27	6.59	8.4	1.28x faster
28	23.62	23.85	1.01x faster
29	107.41	62.99	1.71x slower
30	5.91	69.08	11.7x faster
31	12.59	12.95	1.03x faster
32	14.85	15.93	1.07x faster
33	92.17	57.2	1.61x slower
36	27.89	11.48	2.43x slower
37	0.67	0.52	1.31x slower
38	0.34	0.38	1.12x faster
39	0.34	0.42	1.24x faster
40	2.05	0.83	2.46x slower
41	0.2	0.25	1.28x faster
42	0.17	0.24	1.43x faster
43	0.19	0.27	1.44x faster

Table 1: ClickBench performance on a single core, in seconds, processing a 14GB dataset partitioned into 100 parquet files.

Single Core Efficiency: ClickBench

Highly selective* predicates

```
SELECT COUNT(*) FROM hits WHERE "AdvEngineID" <>
0;

SELECT "AdvEngineID", COUNT(*) FROM hits WHERE
"AdvEngineID" <> 0 GROUP BY "AdvEngineID" ORDER BY
COUNT(*) DESC;

SELECT "UserID" FROM hits WHERE "UserID" =
435090932899640449;
```

* Q2+Q8: Selectivity: 0.006 (630k/100M) , Q20: 4/100M

Query	**DataFusion**	**DuckDB**	Delta
1	1.22	0.18	6.74x slower
2	0.36	0.81	2.25x faster
3	1.11	1.78	1.6x faster
4	1.09	1.5	1.38x faster
5	20.74	8.34	2.49x slower
6	17.81	11.98	1.49x slower
7	0.3	2.08	6.91x faster
8	0.37	0.83	2.24x faster
9	27.91	10.83	2.58x slower
10	25.84	14.11	1.83x slower
11	4.29	3.22	1.33x slower
12	4.67	8.69	1.86x faster
13	11.38	10.27	1.11x slower
14	26.96	14.61	1.84x slower
15	12.7	11.15	1.14x slower
16	13.31	9.12	1.46x slower
17	29.6	21.97	1.35x slower
18	29.09	21.23	1.37x slower
19	92.31	39.1	2.36x slower
20	0.8	1.33	1.65x faster
25	6.01	8.44	1.4x faster
26	5.02	6.11	1.22x faster
27	6.59	8.4	1.28x faster
28	23.62	23.85	1.01x faster
29	107.41	62.99	1.71x slower
30	5.91	69.08	11.7x faster
31	12.59	12.95	1.03x faster
32	14.85	15.93	1.07x faster
33	92.17	57.2	1.61x slower
36	27.89	11.48	2.43x slower
37	0.67	0.52	1.31x slower
38	0.34	0.38	1.12x faster
39	0.34	0.42	1.24x faster
40	2.05	0.83	2.46x slower
41	0.2	0.25	1.28x faster
42	0.17	0.24	1.43x faster
43	0.19	0.27	1.44x faster

Table 1: ClickBench performance on a single core, in seconds, processing a 14GB dataset partitioned into 100 parquet files.

Single Core Efficiency: ClickBench

Single Group Aggregates (no GROUP BY)

```
SELECT AVG("UserID") FROM hits;

SELECT MIN("EventDate"::INT::DATE),
MAX("EventDate"::INT::DATE) FROM hits;

SELECT SUM("ResolutionWidth"),
SUM("ResolutionWidth" + 1), SUM("ResolutionWidth"
+ 2), … SUM("ResolutionWidth" + 88),
SUM("ResolutionWidth" + 89) FROM hits;
```

Query	**DataFusion**	**DuckDB**	Delta
1	1.22	0.18	6.74x slower
2	0.36	0.81	2.25x faster
3	1.11	1.78	1.6x faster
4	1.09	1.5	1.38x faster
5	20.74	8.34	2.49x slower
6	17.81	11.98	1.49x slower
7	0.3	2.08	6.91x faster
8	0.37	0.83	2.24x faster
9	27.91	10.83	2.58x slower
10	25.84	14.11	1.83x slower
11	4.29	3.22	1.33x slower
12	4.67	8.69	1.86x faster
13	11.38	10.27	1.11x slower
14	26.96	14.61	1.84x slower
15	12.7	11.15	1.14x slower
16	13.31	9.12	1.46x slower
17	29.6	21.97	1.35x slower
18	29.09	21.23	1.37x slower
19	92.31	39.1	2.36x slower
20	0.8	1.33	1.65x faster
25	6.01	8.44	1.4x faster
26	5.02	6.11	1.22x faster
27	6.59	8.4	1.28x faster
28	23.62	23.85	1.01x faster
29	107.41	62.99	1.71x slower
30	5.91	69.08	11.7x faster
31	12.59	12.95	1.03x faster
32	14.85	15.93	1.07x faster
33	92.17	57.2	1.61x slower
36	27.89	11.48	2.43x slower
37	0.67	0.52	1.31x slower
38	0.34	0.38	1.12x faster
39	0.34	0.42	1.24x faster
40	2.05	0.83	2.46x slower
41	0.2	0.25	1.28x faster
42	0.17	0.24	1.43x faster
43	0.19	0.27	1.44x faster

Table 1: ClickBench performance on a single core, in seconds, processing a 14GB dataset partitioned into 100 parquet files.

Single Core Efficiency: ClickBench

Medium Selectivity* / Medium Group Cardinality* Aggregates

```
SELECT "SearchEngineID", "SearchPhrase", COUNT(*)
AS c FROM hits WHERE "SearchPhrase" <> '' GROUP BY
"SearchEngineID", "SearchPhrase" ORDER BY c DESC
LIMIT 10;
```

```
SELECT "SearchEngineID", "ClientIP", COUNT(*) AS
c, SUM("IsRefresh"), AVG("ResolutionWidth") FROM
hits WHERE "SearchPhrase" <> '' GROUP BY
"SearchEngineID", "ClientIP" ORDER BY c DESC LIMIT
10;
```

* Selectivity: 0.13 (13M/100M), 6.5M Groups

Query	**DataFusion**	**DuckDB**	Delta
1	1.22	0.18	6.74x slower
2	0.36	0.81	2.25x faster
3	1.11	1.78	1.6x faster
4	1.09	1.5	1.38x faster
5	20.74	8.34	2.49x slower
6	17.81	11.98	1.49x slower
7	0.3	2.08	6.91x faster
8	0.37	0.83	2.24x faster
9	27.91	10.83	2.58x slower
10	25.84	14.11	1.83x slower
11	4.29	3.22	1.33x slower
12	4.67	8.69	1.86x faster
13	11.38	10.27	1.11x slower
14	26.96	14.61	1.84x slower
15	12.7	11.15	1.14x slower
16	13.31	9.12	1.46x slower
17	29.6	21.97	1.35x slower
18	29.09	21.23	1.37x slower
19	92.31	39.1	2.36x slower
20	0.8	1.33	1.65x faster
25	6.01	8.44	1.4x faster
26	5.02	6.11	1.22x faster
27	6.59	8.4	1.28x faster
28	23.62	23.85	1.01x faster
29	107.41	62.99	1.71x slower
30	5.91	69.08	11.7x faster
31	12.59	12.95	1.03x faster
32	14.85	15.93	1.07x faster
33	92.17	57.2	1.61x slower
36	27.89	11.48	2.43x slower
37	0.67	0.52	1.31x slower
38	0.34	0.38	1.12x faster
39	0.34	0.42	1.24x faster
40	2.05	0.83	2.46x slower
41	0.2	0.25	1.28x faster
42	0.17	0.24	1.43x faster
43	0.19	0.27	1.44x faster

Table 1: ClickBench performance on a single core, in seconds, processing a 14GB dataset partitioned into 100 parquet files.

Single Core Efficiency: ClickBench

High Cardinality* Groups

```
SELECT "UserID", "SearchPhrase", COUNT(*) FROM
hits GROUP BY "UserID", "SearchPhrase" LIMIT 10;
```

```
SELECT "ClientIP", "ClientIP" - 1, "ClientIP" - 2,
"ClientIP" - 3, COUNT(*) AS c FROM hits GROUP BY
"ClientIP", "ClientIP" - 1, "ClientIP" - 2,
"ClientIP" - 3 ORDER BY c DESC LIMIT 10;
```

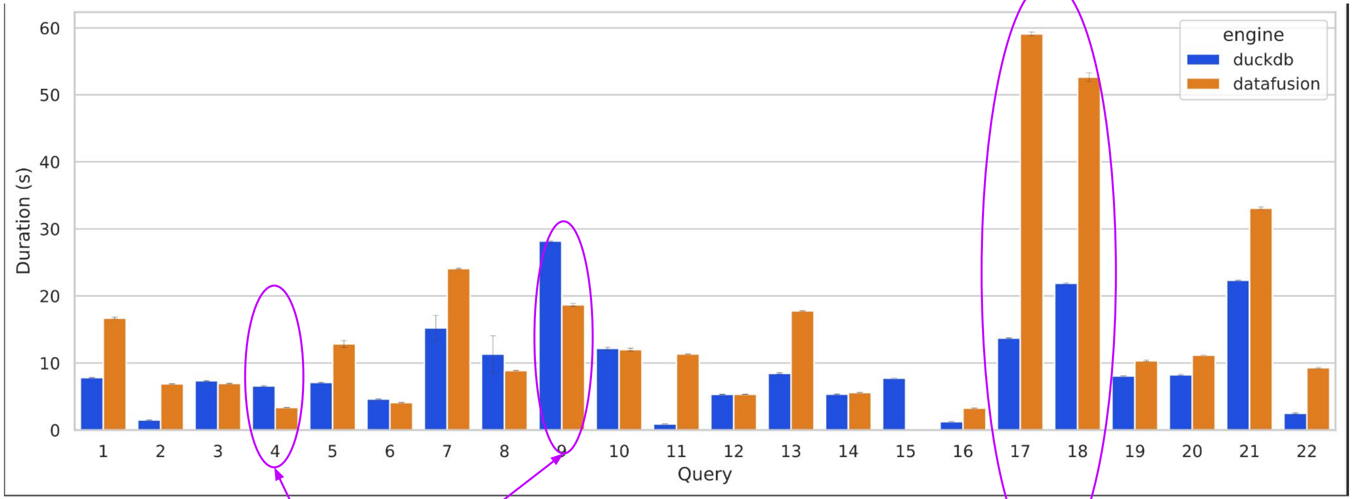
* Q18/Q19: 24M Groups, Q36: 10M Groups

Query	**DataFusion**	**DuckDB**	Delta
1	1.22	0.18	6.74x slower
2	0.36	0.81	2.25x faster
3	1.11	1.78	1.6x faster
4	1.09	1.5	1.38x faster
5	20.74	8.34	2.49x slower
6	17.81	11.98	1.49x slower
7	0.3	2.08	6.91x faster
8	0.37	0.83	2.24x faster
9	27.91	10.83	2.58x slower
10	25.84	14.11	1.83x slower
11	4.29	3.22	1.33x slower
12	4.67	8.69	1.86x faster
13	11.38	10.27	1.11x slower
14	26.96	14.61	1.84x slower
15	12.7	11.15	1.14x slower
16	13.31	9.12	1.46x slower
17	29.6	21.97	1.35x slower
18	29.09	21.23	1.37x slower
19	92.31	39.1	2.36x slower
20	0.8	1.33	1.65x faster
25	6.01	8.44	1.4x faster
26	5.02	6.11	1.22x faster
27	6.59	8.4	1.28x faster
28	23.62	23.85	1.01x faster
29	107.41	62.99	1.71x slower
30	5.91	69.08	11.7x faster
31	12.59	12.95	1.03x faster
32	14.85	15.93	1.07x faster
33	92.17	57.2	1.61x slower
36	27.89	11.48	2.43x slower
37	0.67	0.52	1.31x slower
38	0.34	0.38	1.12x faster
39	0.34	0.42	1.24x faster
40	2.05	0.83	2.46x slower
41	0.2	0.25	1.28x faster
42	0.17	0.24	1.43x faster
43	0.19	0.27	1.44x faster

Table 1: ClickBench performance on a single core, in seconds, processing a 14GB dataset partitioned into 100 parquet files.

Single Core Efficiency: TPCH (SF10, parquet files)

TPCH: exercises low + medium cardinality aggregations, joins

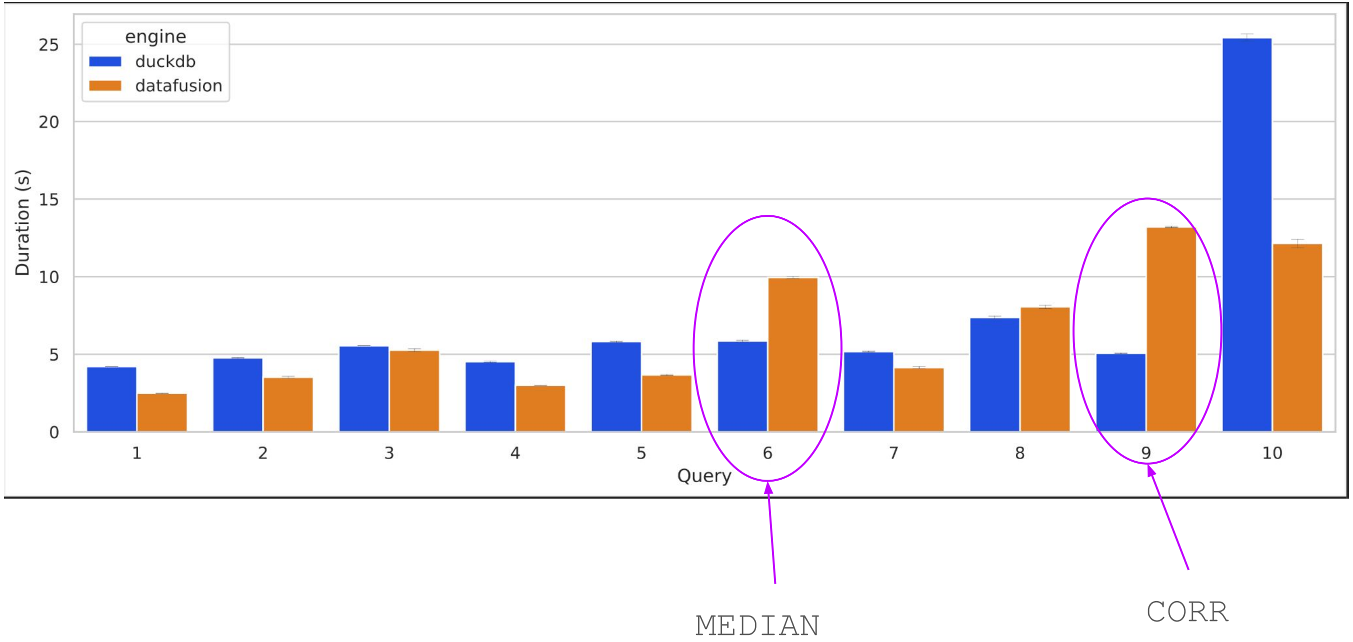


Selective Predicates

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

Single Core Efficiency: H2O Grouping

Group on CSV
Mostly a test of CSV parser performance



Notes:

1. DuckDB, may have optimized multi-threaded parsing over single core (we filed ticket), while a similar trade off doesn't exist for DataFusion.
2. DataFusion's MEDIAN has been improved since these measurements

Conclusion

- Relative performance reflects engineering effort, not architecture
- Volcano exchange style parallelism scales to at least 172 cores
- With additional engineering both engines can (and will) get faster
- ⇒ an open design does not require performance sacrifices

APACHE

DATAFUSION™

https://datafusion.apache.org/

Come help us build this amazing technology

All open source, Apache 2.0 Licensed, Governed by the Apache Software Foundation

Find 100s of like-minded people hacking on Databases!