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

Andrew Lamb
InfluxData
alamb@influxdata.com

Yijie Shen
Space and Time
yijie.shen@spaceandtime.io

Daniël Heres
Coralogix
daniel.heres@coralogix.com

Jayjeet Chakraborty
UC Santa Cruz
jayjeetc@ucsc.edu

Mehmet Ozan Kabak
Synnada
ozan@synnada.ai

Chao Sun
Apple Inc.
sunchao@apple.com

Liang-Chi Hsieh
Apple Inc.
liangchi@apple.com

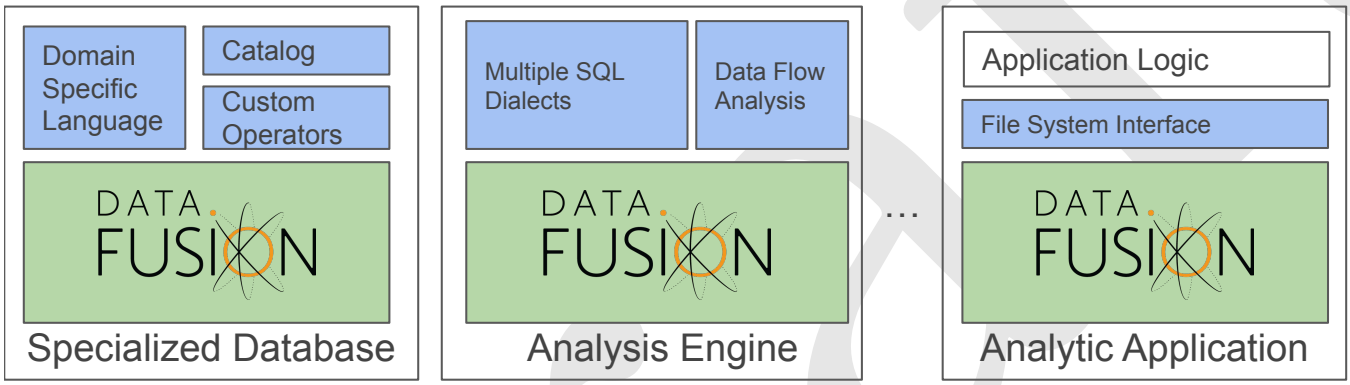

Figure 1. Apache Arrow DataFusion is a fast, embedded query engine that uses Apache Arrow as its in-memory format. System designers implement domain-specific features via extension APIs (blue), rather than re-implementing standard OLAP query engine technology (green). Query performance is similar to best-of-breed, tightly integrated systems.

Abstract

Apache Arrow DataFusion[26] is a fast, embeddable, and extensible query engine written in Rust[71] that uses Apache Arrow[25] as its memory model. Many commercial and open-source databases, machine learning pipelines, and other data-intensive systems are built using DataFusion. DataFusion demonstrates that a rich feature set with state-of-the-art performance is attainable with a modular and extendable design based on open technical standards and Apache Software Foundation[28] governance. We anticipate that the accessibility and versatility of DataFusion, along with its competitive performance, will enable a proliferation of new, high-performance custom data infrastructures tailored to specific needs by composing modular components.

CCS Concepts: • **Information systems → Database management system engines**; **DBMS engine architectures**; **Database query processing**; **Online analytical processing engines**.

Keywords: database systems

ACM Reference Format:
Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Chao Sun, and Liang-Chi Hsieh. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Proceedings of ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

SIGMOD '24, June 09–15, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Traditionally, the realm of high-performance analytic query engines has been dominated by tightly integrated systems such as Vertica[46], Spark[72], and DuckDB[63]. This approach was necessary to optimize the interfaces between the file format, in-memory layout, and processing engine to reach peak performance. However, building such a system is expensive and typically requires substantial commercial and/or research funding, given the extensive software engineering required. As more analytic systems were built over time, academia and industry have learned how best to draw boundaries between subsystems such as the file format, catalog, language front-ends, and execution engine [20, 59].

DataFusion continues this trend and enables the assembly of an end-to-end system from high-quality, reusable, and open components, embracing open standards such as Apache Parquet[27] and Arrow[25] throughout its architecture, and permitting extensions at every level. DataFusion's competitive performance demonstrates that a modern OLAP engine need not have a tight-knit architecture, and its mere existence as a permissively licensed open-source project demonstrates that an open organizational structure, enabled by the Apache governance model[29], is capable of creating and maintaining this level of technology.

This paper makes the following technical contributions:

1. Describes the ecosystem of foundational technologies that power DataFusion and that we believe will power the majority of successful analytic systems over the next decade.
2. Describes the types of systems built with DataFusion, illustrating what is possible with commodity OLAP engines.
3. Describes DataFusion's architecture, feature set, and optimizations, illustrating the breadth of features required of modern analytic engines and quantifying the effort necessary to implement one.
4. Defines DataFusion's extension APIs, outlining key module boundaries in an analytic stack.
5. Evaluates DataFusion's performance, demonstrating that state-of-the-art performance is achievable using modular components and open standards.

The rest of this paper is organized as follows: **Section 2** reviews foundational technologies. **Section 3** describes use cases and examples of real-world adoption. **Section 4** explores the trend towards modular databases. **Section 5** describes DataFusion's architecture, detailing its execution model and key components. **Section 6** enumerates many of the standard query optimizations included in DataFusion. **Section 7** describes the APIs for extending DataFusion. **Section 8** evaluates DataFusion's performance. We describe related work in **Section 9** and conclude in **Section 10**.

2 Foundational Ecosystem

DataFusion is only possible due to the advent of several lower-level transformative technologies: Apache Arrow's in-memory columnar structure and compute kernels, Parquet's efficient columnar storage, and the Rust ecosystem that enables a high-performance, yet comprehensible implementation. Without these technologies, it is unlikely we could have built DataFusion with the relatively modest resources available. Additionally, using these technologies, systems built with DataFusion easily integrate with the broader ecosystem, directly sharing files and in-memory data streams without time-consuming and error-prone format transformations.

2.1 Apache Arrow

At its core, Apache Arrow[25] simply standardizes industrial best practices for representing data in memory using cache-efficient columnar layouts. By standardizing implementation details such as validity/null representations, endianness, variable length byte and character data, lists, and nested structures, systems built with Arrow benefit from well known techniques and easy data interchange between applications. For example, while it is likely not critical for most systems if a `NULL` value is represented by a `0` or `1` in a bit mask, it is critical that all systems agree on the same convention for interoperability.

Originally, Arrow was designed as an in-memory interchange format and added compute-focused features such as `StringView`[22] and high-performance compute kernels over time. Arrow users can thus avoid re-implementing features that are well understood in academia and industry, but time-consuming to implement.

2.2 Apache Parquet

Apache Parquet[27] is an open-source, column-oriented data file format, originally designed for the Hadoop ecosystem and inspired by academic work on columnar storage[69]. It provides efficient data compression and encoding schemes, along with support for structured types via record shredding [53], embedded schema description, zone-map[54] like index structures and Bloom filters for fast data access.

Unlike Arrow, which is designed for fast random access and efficient in-memory processing, Parquet is optimized to store large amounts of data in a space-efficient manner. Like all formats, Parquet is not perfect, but it has become the de-facto standard for data storage and interchange in the analytic ecosystem. Its combination of an open format, excellent compression across real-world data sets, broad ecosystem and library support, and embedded self-describing schema makes it a compelling choice for storing and exchanging compressed data. In addition to compression and compatibility, the file structure allows query engines to apply advanced projection and filter push-down techniques, such as

late materialization, directly on files, yielding competitive performance compared with specialized formats[3].

2.3 Rust

Rust[71] is a relatively new system programming language, featuring a low-level, yet safe memory management approach and C-like performance. It incorporates an innovative memory ownership model that mitigates many of the worst memory and thread safety challenges prevalent in traditional C/C++ programming. Rust programs are easy to embed in other systems as they do not require a language run-time and have C ABI compatibility. Rust's strong emphasis on zero-cost abstractions and its rich ecosystem of performance-centric libraries, along with developer-friendly documentation and diagnostic tools, make it a compelling choice for implementing high-performance applications with a relatively lower engineering investment.

Unlike many C/C++ build systems, which can require substantial effort to just configure on a particular environment, Rust's built-in Cargo Package Manager[14] and crate ecosystem makes adding DataFusion to most projects as simple as adding a single line to a configuration file.

3 Use Cases

A wide variety of commercial products and open source projects use DataFusion due to its combination of extensibility, feature set, fast query performance, and ecosystem compatibility. Projects leveraging DataFusion typically spend most of their time innovating value-adding features rather than replicating existing analytic engine technologies. While still early in adoption, DataFusion is already used in:

1. **Tailored database systems** for domain-specific uses such as time-series databases (e.g. InfluxDB 3.0[40] and Coralogix[15]), as well as **streaming SQL platforms** (e.g. Synnada[70] and Arroyo[7]).
2. **Execution run-times** for diverse query front-ends, such as Apache Spark (Section 3.1), the Vega visualization language[52], and the InfluxQL[38] time series query language.
3. **SQL analysis tools** such as dask-sql[60] and SDF[65], which leverage DataFusion's SQL parser, planner, and plan representation to analyze SQL queries.
4. **Table formats** such as the Rust implementations of Delta Lake[6] and Lance[47], which use DataFusion expressions and query plans to fetch and decode remote data, implement predicate-based delete tombstones, push predicates to specialized secondary indexes, and compact files while retaining sort orders.

All these systems inherit the Arrow-native aspects of Datafusion, and easily integrate with the Python ecosystem via `pyarrow`[30]. For example, Lance has many APIs where users write Python functions that operate on `RecordBatches`, which operate directly on the data without any conversion.

3.1 Accelerating Apache Spark

Apache Spark[72] is an open-source analytic engine for large-scale data processing, widely adopted as a standard tool for data engineering, data science, and machine learning. Implemented primarily in JVM languages Scala and Java, its performance suffers from well-known JVM overheads.

With its high adoption and easy-to-use APIs, Spark will likely remain a major data infrastructure component in the near term. Fortunately, Spark's design allows replacing *just* the execution engine with a specialized implementation like Velox[58] (open-source) or Photon[9] (proprietary).

DataFusion is used by several Spark native runtimes, including Blaze[10] and at least one project that is not yet open-source. In these projects, Spark's query front-ends and its parsing, analysis, and optimization steps are used as is, while its execution plans are converted to DataFusion `ExecutionPlans` (Section 5.5) that execute through JNI interfaces where zero-copy data exchange is facilitated by Apache Arrow. In scenarios where Spark's semantics differ from those offered by DataFusion, the latter's extensible design (Section 7) permits these projects to override and implement Spark specific expressions and operators (e.g., decimal related operations where Spark semantics deviate from ANSI SQL).

4 Deconstructed Databases

The rise of DataFusion and similar systems, such as Apache Calcite[8] and Velox[58], is part of a longer-term trend away from monolithic "one size fits all" general-purpose systems to "fit for purpose" specialized systems[68]. Given the expense of building the underlying technology, widespread proliferation of such specialized systems is only feasible when they can be assembled from reusable high-quality components, a trend which has been called the *Deconstructed Database*[43][59].

The database systems literature offers a vast array of advanced and thoroughly studied techniques for most operations. However, due to economic and architectural constraints, these techniques have historically been confined to tightly integrated, often proprietary databases or analogous analytic systems. This tight integration limits reuse, leading to numerous costly re-implementations.

One classic example of a re-implementation is data science analysis tools, such as `pandas`[57]. The data science community innovated new APIs (DataFrame vs. SQL) and preferred a different deployment model (local files vs. networked servers), distinct from most contemporary database offerings. However, these tools initially performed poorly and did not incorporate many well-known techniques from database systems, such as query planning/optimization and parallel vectorized execution. In fact, Apache Arrow was initially born out of a desire to bring such well-studied database systems techniques to the data science ecosystem.

Another example of a missed opportunity for reuse was the emergence of MapReduce[21] and its open-source implementation, Hadoop, for parallel distributed processing. While database researchers pointed out several ways it was technically inferior[24], the lack of open, standard, and reusable components inevitably led to the re-implementation of very similar low-level analytical techniques.

4.1 Parallel with `LLVM`

The transition from a few monolithic implementations to a large number of specialized systems that share an underlying open-source foundation has happened before in system software stacks. Modern programming language tooling underwent a similar transformation, enabled by LLVM[48]:

1. **Tightly integrated** designs where compilers with hardware-specific code generation and operating system-specific libraries were distributed as an integral part of systems such as IBM System/390, Solaris, AIX, and HP-UX. Similarly, traditional integrated monolithic database systems such as Oracle[18], SQL Server [17], and DB2[16] directly manage storage hardware, client connections, SQL language functions, query execution, and on-disk/in-memory formats.
2. **Open source, internally integrated** compilers such as `gcc` work across multiple hardware platforms and operating systems and thus were much more widely adopted. Similarly, cross-platform, open source, but internally integrated database systems such as MySQL and PostgreSQL were widely adopted.
3. **Open source and modular** compilers for programming languages like Rust[71], Swift[5], Zig[32], and Julia[11] share the same high quality backend, `LLVM`. This is similar to how InfluxDB 3.0, GreptimeDB, and Coralogix are built using the same high-quality reusable query engine, DataFusion.

Just as LLVM's modular and reusable compiler technology catalyzed the new development of advanced, industrial-strength systems programming languages, DataFusion catalyzes the development of new data systems. For programming languages built on LLVM, authors now concentrate on enhancing language-specific features and reusing the same shared code for critical, yet commonplace, low-level details such as intermediate representations, standard optimizations (e.g. loop unrolling), architecture-specific code generation, and auto-vectorization. For databases built on DataFusion, designers create value-added, domain-specific features instead of re-implementing elements like SQL front-ends, plan representations, optimizations, storage formats, and execution operators.

5 DataFusion Features

5.1 Engine Overview

DataFusion is designed to work "out of the box" with very little effort while also providing extensive customization APIs, which we describe in Section 7. This architecture, shown in Figure 2, allows users to quickly start with a basic, high-performance engine and specialize the implementation over time to suit their needs and engineering capacity. The implementation follows industrial best practices informed by database research literature, with a focus on efficient and extensible implementations of well-known patterns.

5.2 Catalog and Data Sources

5.2.1 Catalog. To plan and execute queries, DataFusion needs a *Catalog* to provide metadata such as which tables exist along with their columns and data types, statistical information, and storage details. DataFusion includes a simple in-memory catalog and a Apache Hive[12]-like partitioned file/directory-based catalog. Catalog management, being a key design element for most data systems, it is unlikely any general-purpose catalog implementation will work well for all use cases. Therefore, most implementations use DataFusion's APIs (Section 7.2) to supply catalog information, e.g., directly from a Hive metastore.

5.2.2 Data Sources: Parquet, Avro, JSON, CSV, Arrow. DataFusion includes five built-in `TableProviders` for commonly used file formats: Apache Parquet, Apache Avro, JSON, CSV, and Apache Arrow IPC files – all implemented via the same API any other custom source would use. The Parquet reader leverages the Arrow Rust implementation and features advanced predicate pushdown and late materialization (Section 6.8), bloom filters, and nested types. The CSV and JSON readers automatically infer schema from source files, and the JSON reader fully supports structs and lists.

5.3 Front Ends

5.3.1 Data Types. DataFusion directly uses the Apache Arrow type system and inherits its broad range of supported types, including integral and floating-point numerics of various byte widths, fixed precision decimals, variable length character and binary strings, dates, times, timestamps, intervals, duration types, nested structs and lists. During execution, operators exchange data as either Arrow Arrays or scalar values.

5.3.2 SQL Planner. DataFusion uses the `sqlparser-rs` [67] library to parse SQL and generates a `LogicalPlan` from the parsed query representation. While it is likely that no SQL implementation should ever claim to be "complete" given the amorphous, ever-expanding SQL specification[41]; DataFusion supports a large subset of SQL features including `WHERE`, `GROUP BY`, `ORDER BY`, `LIMIT`, `DISTINCT`, `WINDOW / OVER`, `UNION / INTERSECT`, `GROUPING SETS`, `FULL / INNER`

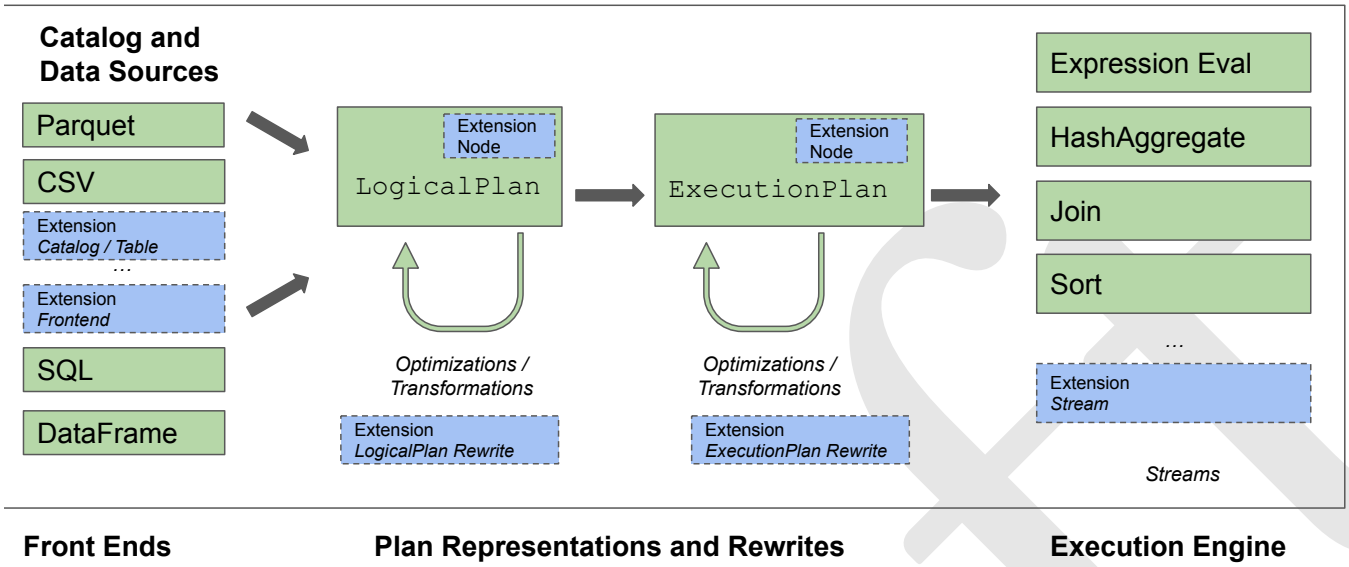

Figure 2. DataFusion's Architecture. First, a front-end uses catalog and data source information to create a `LogicalPlan`, a tree of relational operators. Optimizer passes rewrites the `LogicalPlan` to a more optimal form, which is then lowered to an `ExecutionPlan` that includes additional characteristics of the intermediate results and specific algorithm selections. A second set of passes rewrites the `ExecutionPlan` to best match available physical resources and data layout. Finally, the `ExecutionPlan` is executed by creating one or more specialized `Streams` ("operators") that produce results incrementally. At each stage, DataFusion can be extended and customized.

`/ OUTER JOIN`. It also supports more advanced functionality such `ROWS` / `VALUES PRECEDING`, `FOLLOWING`, `UNBOUNDED` window bounds and, `GROUP BY` with per-aggregate `FILTER` and `ORDER BY`.

5.3.3 DataFrame and `LogicalPlanBuilder` APIs.

In addition to SQL, DataFusion also offers a DataFrame API, modeled after `pandas`[57], for expressing queries in a procedural style. The DataFrame API generates the same underlying `LogicalPlan` representation (Section 5.4.1) as the SQL API, which is optimized and executed the same way. For more advanced uses, such as custom query languages, the `LogicalPlanBuilder` API offers a Rust builder-style interface for constructing plans directly.

5.4 Plan Representations and Rewrites

5.4.1 Plans and Expressions.

DataFusion's API includes: (1) A full range of structures to represent and evaluate trees of expressions and relational operators, both at logical (`Expr` and `LogicalPlan`) and physical (`PhysicalExpr` and `ExecutionPlan`) levels, along with routines to create and manipulate them ergonomically; (2) Libraries to (de)serialize these structures from/to bytes suitable for network transport, both using Protocol Buffers as well as Substrait[23]; (3) Structures to describe statistics that may be known at planning time, such as row counts and minimum/maximum values.

5.4.2 Expression Analysis.

In addition to basic expression evaluation, DataFusion provides libraries for simplification, interval analysis[55], and range propagation. Combined with statistics, these libraries provide predicate cardinality and selectivity estimates, and plan-time partition elimination (e.g. Parquet row group pruning, described in Section 6.8). These features are both usable directly by client systems and used to implement DataFusion's built-in optimizations.

5.4.3 Function Library.

DataFusion features a large library [31] of built-in scalar, window, and aggregate functions, including string operations, timestamp/date/time manipulations, interval arithmetic, and list/struct/map operations. These functions are implemented using the same API as user-defined functions by manipulating Arrow Arrays and can be invoked via both SQL or DataFrame APIs.

5.4.4 Rewrites.

DataFusion includes an extensible plan rewriting framework, implemented as a series of `LogicalPlan` and `ExecutionPlan` transformations. These passes handle details such as automatically coercing types to match available operator and function signatures, and introducing necessary sort and redistribution operations. The same framework is used for optimizations as well (Section 6.1).

5.5 Execution Engine

DataFusion uses a pull-based streaming execution model and distributes work across multiple cores using Volcano-style [34] exchange operators (viz. `RepartitionExec`).

5.5.1 Streaming Execution. Whenever possible, all operators produce output incrementally (Figure 3) as Arrow Arrays grouped into `RecordBatches`, with a default size of 8192 rows. For pipeline-breaking operations such as a full sort, final aggregation, or a hash join, the operators buffer (and spill to disk) tuples as necessary. Data flows through operators such as Arrow Arrays, which allows for seamless integration of user-defined operators (Section 7.7). *Within* each operator, non-Arrow representations, such as the *Row Format* (Section 6.6) are used when necessary for performance.

```
impl Stream for MyOperator {
  ...
  // Pull next input (may yield at await)
  while let Some(batch) = stream.next().await {
    // Calculate, check if output is ready
    if Some(output) = self.process(&batch)? {
      // "Return" RecordBatch to output
      tx.send(batch).await
    }
  }
  ...
}
```

Figure 3. Streaming Execution. Each `Stream` (operator) implements the Rust `Stream`[33] trait, incrementally producing Apache Arrow `RecordBatches` that flow through the plan. Control flow is managed using Rust's built-in `await` continuation generation, automatically marshaling the necessary state before yielding control. Each `Stream` attempts to output `RecordBatches` with a target number of tuples.

5.5.2 Multi-Core Execution. Each `ExecutionPlan` generates one or more `Streams` (i.e. operators) that run in parallel. Most `Streams` coordinate only with their input(s), but some must coordinate with sibling `Streams`, such as a `HashJoinExec` when building a shared hash table or a `RepartitionExec` when redistributing data to different streams. The number of `Streams` created for each `ExecutionPlan` is called its *partitioning*, which is determined at plan time (Figure 4).

5.5.3 Thread Scheduling. DataFusion `Streams` are implemented as Rust `async` functions and run within a `Tokio`[64] runtime leveraging a thread pool. `Tokio` is one of the most widely used libraries in the Rust ecosystem and was initially designed for asynchronous network I/O. However, its combination of an efficient, work-stealing scheduler, first-class compiler support for automatic continuation generation, and exceptional performance makes it a compelling choice for CPU-intensive applications as well[45]. While some recent work[49] describes challenges with the Volcano model on NUMA architectures, in practice, DataFusion achieves similar scalability as systems that use alternate designs (Section 8.2).

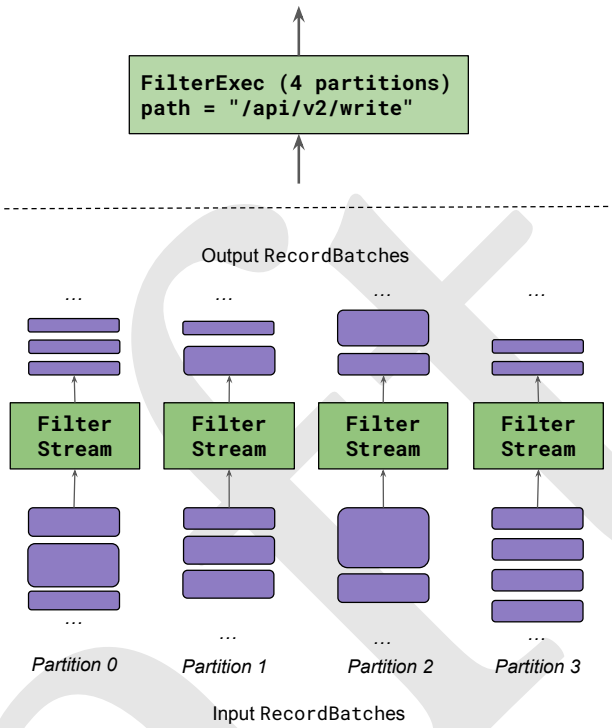

Figure 4. Partitioned Execution: Each `ExecutionPlan` is annotated with a number of partitions chosen by the planner, and a `Stream` (operator) is created for each partition. The `Streams` run independently on multiple threads. In this figure, the `FilterExec` `ExecutionPlan` (top) has 4 partitions. Thus, 4 distinct `FilterStream` operators are created during execution, and they run in parallel without coordination.

5.5.4 Memory Management. DataFusion manages memory using a `MemoryPool`, which is shared across one or more concurrently running queries. `Streams` cooperatively record their memory usage with the `MemoryPool` consumer APIs. `Stream` implementations take a pragmatic approach to memory management, accurately tracking the largest memory consumers (e.g., contents of the hash table in a hash aggregate), but not small ephemeral allocations (e.g., memory for the current output batch).

DataFusion has two built-in memory pool implementations. The first is `GreedyPool`, which sets per-process memory limits but does not attempt to distribute resources fairly across `Streams` in a query. The second is a basic `FairPool`, designed to distribute resources evenly among all pipeline-breaking operators. Systems built on DataFusion typically implement domain-specific policies using the same API.

6 Optimizations

Query engines allow users to express their desired results, and the engine handles the many details necessary to compute them efficiently. This section enumerates some of the techniques used by DataFusion to efficiently execute queries.

The techniques are not novel. Each has been extensively studied and documented in research literature and implemented many times in commercial systems. DataFusion's well-tested implementations and extensibility allow new systems to avoid the cost of re-implementing them (yet) again.

6.1 Query Rewrites

DataFusion includes a variety of query rewrites for both `LogicalPlans` and `ExecutionPlans`. `LogicalPlan` rewrites include projection pushdown, filter pushdown, limit pushdown, expression simplification, common subexpression elimination, join predicate extraction, correlated subquery flattening, and outer-to-inner join conversion. `ExecutionPlan` rewrites include eliminating unnecessary sorts, maximizing parallel execution, and determining specific algorithms such as Hash or Merge joins.

6.2 Sorting

Sorting, along with grouping and joining, is one of the most expensive operations in an analytic system and is well-studied in the literature. Most commercial analytic systems include heavily optimized multi-column sorting implementations, and DataFusion is no exception. Broadly based on the techniques described in [35], it incorporates a tree-of-losers, a RowFormat (Section 6.6), the ability to spill to temporary disk files when memory is exhausted, and specialized implementations for `LIMIT` (aka "Top K").

6.3 Grouping and Aggregation

Similarly to sorting, grouped aggregations are a core part of any analytic tool, as they create understandable summaries of large data volumes and are both well-studied and highly optimized in industrial systems. DataFusion contains a two-phase parallel partitioned hash grouping implementation[2], featuring vectorized execution, the ability to spill to disk when memory is exhausted, and special handling for no group keys, partially ordered and fully ordered group keys.

6.4 Joins

When joining multiple relations, DataFusion automatically identifies equality (equi-join) predicates, heuristically reorders joins based on statistics, pushes predicates through joins (subject to OUTER join restrictions), introduces transitive join predicates, and picks the optimal physical join algorithm. It includes parallel in-memory hash join, merge join, symmetric hash join, nested loops join and cross join implementations which each support Inner, Left, Right, Full, LeftSemi, RightSemi, LeftAnti, RightAnti joins, and are optimized for equality predicates. The in-memory hash join is implemented using vectorized hashing and collision checking similar to that described in [36]. While not implemented at the time of writing, we are working on additional join performance such as dynamically applying join filters during scans[1] (a form of sideways information passing[66]).

6.5 Window Functions

DataFusion supports SQL Window Functions (e.g. functions that have an `OVER` clause). Like most optimized window function implementations, DataFusion minimizes resorting by reusing existing sort orders, sorting only if necessary based on the `PARTITION BY` and `ORDER BY` columns. It evaluates window functions incrementally[56], producing output once the required input window is present. We have not yet found the need to implement newer, more sophisticated (and complex) schemes such as Physical Segment Trees[50] as the processing time of queries with window functions is typically dominated by other operations such as sorting.

6.6 Normalized Sort Keys / RowFormat

Columnar engines like DataFusion perform well on operations that naturally vectorize. However, query processing also requires efficient fundamentally row-based operations such as multi-column sorting and multi-column equality comparisons for grouping and joins, where the per row overhead can not be amortized by vectorization[42]. Within such operators, DataFusion uses a *RowFormat*[4], a form of normalized key[35] which 1) permits byte-wise comparisons using `memcmp` and 2) offers predictable memory access patterns. The RowFormat is densely packed, one column after another, with specialized encoding schemes for each data type, optionally adjusted for SQL sort options, such as `ASC` or `DESC` order and `NULL` placement. For example, unsigned and signed integers are encoded using their big-endian representation, whereas floating-point numbers are converted to a signed integer representation that incorporates the sign bit.

6.7 Leveraging Sort Order

DataFusion's Optimizer is aware of, and takes advantage of, any order that pre-exists in the input or the intermediate results that flow from `Stream` to `Stream`. DataFusion 1) tracks multiple sort orders[2] and 2) includes `Streams` optimized for sorted or partially sorted input, such as Merge Join and partially ordered (streaming) Hash Aggregation.

Leveraging sort-order is important for at least two reasons:

1. **Physical Clustering**: Secondary indexes are often too expensive to build and maintain at high ingest rates,

[1] https://github.com/apache/arrow-datafusion/issues/7955

[2] E.g. data is sorted by (`A`, `B`) *and* (`A`, `C`) via an order preserving join on `B=C`

and thus, the sort order of primary storage is the only available physical optimization to cluster data.
2. **Memory Usage and Streaming Execution**: The sort order defines how the data that flows through `Streams` is partitioned in time, defining where values may change and thus where intermediate results can be emitted.

6.8 Pushdown and Late Materialization

DataFusion pushes several operations down (towards the data sources): 1) projection (column selection) which elides unnecessary columns from intermediate results 2) `LIMIT` and `OFFSET`, which permits the plan to stop early when results are no longer needed and 3) predicates which moves filtering closer (or *in*) to data sources, minimizing the amount of data processed by the rest of the plan.

Pushing filters into data source enables implementations to apply filters *during* the scan, potentially avoiding significant work during execution. For example, DataFusions's Parquet reader uses pushed-down predicates to 1) prune (skip) entire Row Groups and Data Pages based on metadata and Bloom filters and 2) apply predicates after decoding only a subset of column values, a form of late materialization[1] which can avoid the effort required to decode values in other columns that will be filtered out.

To illustrate, consider a query with the condition `A > 35 AND B = "F"`. DataFusion's Parquet reader:

1. Prunes (skips) all Row Groups such that A_{max} `<= 35` or B_{max} `< 'F'` or B_{min} `> 'F'` using Row Group metadata.
2. Decodes column `B`, and evaluates `B = "F"`, capturing all rows which pass as a `RowSelection` (e.g. row indexes `[100-244]`)
3. Decodes only pages that contain the relevant rows from Column `A`, using the Page Index, and evaluates $A > 35$ further refining the `RowSelection` (e.g. to row indexes `[100-150]`)
4. Decodes the pages containing the remaining `RowSelection` for any other selected columns (e.g. `C`)

Together, these techniques are very effective when predicate columns are cluster together such as when they appear early in the sort order of a sorted file[3].

7 Extensibilities

This section describes the extension points for DataFusion, which are sufficiently flexible to support a wide variety of use cases (Section 3). We believe this list of extension APIs offers a blueprint for future modular query engines as well as internal boundaries of more tightly integrated systems.

All extension APIs represent data using Arrow Arrays. Because DataFusion uses Arrow internally, extensions have equal performance as built-in functions and can use the same wide range range of existing libraries, knowledge, and tools (e.g. well-documented and optimized computation kernels).

7.1 Scalar, Aggregate, and Window Functions

Systems built on DataFusion often add use case specific functions that don't belong in a general function library. Examples systems have added include window functions that compute derivatives, calendar bucketing for timeseries, and custom binary manipulation for cryptography functions.

Users can register several types of functions with DataFusion dynamically at runtime, which receive Arrow Arrays as input arguments and produce Arrow Arrays as output:

1. **Scalar**: a single output row for each input row
2. **Aggregate**: a single output row for many input rows
3. **Window**: a single output row for each input row, but the calculation has access to values in a surrounding window frame.

DataFusion is not, of course, the first engine to offer user-defined function APIs. However, the utility of such APIs in other systems is often limited because the performance and functionality are worse than built-in functions. Even when similarly performant APIs do exist, they must be tightly bound to the specifics of how the engine represents and operates on its data. This is especially true for column-oriented engines, which are often more challenging to implement than one-row-at-a-time interfaces due to vectorization[42] .

7.2 Catalog

Using a combination of the Catalog API and expression evaluation (Section 5.4.1), Catalogs built with DataFusion use file metadata (such as minimum and maximum values) to avoid reading entire files or parts of files (e.g. Row Groups). For example, the Rust implementation of the Delta Lake table format uses DataFusion to skip reading Parquet files based on the query predicates.

The Catalog API consists of 1) `TableProvider` for individual tables (Section 7.3), 2) `SchemaProvider`, a collection of `TableProviders`, and 2) `CatalogProvider`, a collection of `SchemaProviders`, a concept sometimes referred to as a "catalog" or "database" in other systems. These APIs are `async` Rust functions, which makes it straightforward to implement remote catalogs.

7.3 Data Sources

Using the DataFusion DataSource API, systems can query in-memory buffers of Arrow Arrays, stream data from remote servers (perhaps via Arrow Flight), or read from custom file formats, including optimizations such as filtering and projection.

DataFusion's built-in providers (Section 5.2.2) are implemented with the same API exposed to users, the `TableProvider` trait, and produces the same Rust `async Stream` of Arrow

Arrays as `ExecutionPlans`. The `TableProvider` API additionally supports 1) partitioned inputs, 2) pushdown of projection, filter, and limit, 3) parallel concurrent reads, and 4) communicating pre-existing sort orders.

Similarly to user-defined functions, in tightly integrated engines it is typically challenging to create user-defined datasources that perform as well as built-in formats. Not only must the implementation produce data in the engine's native format, it must also interact with expression representation to implement predicate pushdown, and interface with asynchronous network I/O to implement incremental (streaming) output.

7.4 Execution Environment

Execution environments vary widely from system to system. For example, if fast local NVMe storage is present, caching metadata in memory might make less sense than it does in environments where persistent local disk is less available, such as Kubernetes. Likewise, some systems run multiple queries concurrently, optimistically sharing resources between them, and others run a mix of queries with predefined resource budgets. DataFusion can be customized for these different environments using the *MemoryPool* trait to control memory allocations, the *DiskManager* trait for managing temporary files (if any), and a *CacheManager* for caching information such as directory contents and per-file metadata.

7.5 New Query / Language Frontends

Users extend the SQL supported by DataFusion by rewriting the AST prior to calling the DataFusion SQL planner (Section 5.3.2). For more substantial extensions or entirely different languages such as PromQL or Vega, users implement their own parser and/or planners that create `LogicalPlans` using the structures described in Section 5.4.1.

7.6 Query Rewrites / Optimizer Passes

DataFusion users have added domain-specific optimizations such as input reordering and macro expansions by implementing *OptimizerRule*s and *PhysicalOptimizerRule*s, which rewrite `LogicalPlan` and `ExecutionPlan` trees, respectively, with the same APIs as the built-in rewrites (Section 6.1). Users can also specify the order in which rewrites are applied, both provided as well as their own.

7.7 Relational Operators

Domain-specific systems often require relational operations not found in SQL-only systems. For example, InfluxDB IOx [39] has specialized operators for timeseries gap filling, schema pivoting operations, and insert order resolution.

Users extend DataFusion by implementing the *ExecutionPlan* trait, exactly the same as the nodes provided with DataFusion, such as join, filter, group by, and windowing. DataFusion does not distinguish between user-defined and built-in plans while optimizing and running plans. While other systems offer similar functionality in the form of user defined table functions, those APIs both restrict the syntax and placement of those operators in plans and are often unable to perform as well as built-in operators.

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.

8 Performance Evaluation

To quantify the performance penalty of using open standards and a modular architecture, rather than a tightly integrated design, we compared DataFusion's performance to

DuckDB[63], a system we think exemplifies a state of the art, tightly integrated query engine. DataFusion performs similarly over a variety of real world usecases. While we acknowledge the challenges of benchmarking[62], different target usecases, and the rate of change in both engines, these results show there is nothing fundamental about an open design that requires performance sacrifices.

The most important measurement for a query engine is the end-to-end query performance that users experience, so we used a set of standard benchmarks that reflects commonly encountered data sizes and query patterns:

1. **ClickBench**[37] models large scale web analytic processing, with queries that filter and aggregate a large denormalized dataset. We used the unmodified 14 GB `athena_partitioned` dataset, which consists of 100 Parquet files, each approximately 140 MB in size.
2. **TPC-H**[19] models classic data warehouse analytics with 22 queries that join several tables in summary reports. We used the standard TPC-H data generator with Scale Factor=10 and converted each of the resulting 8 CSV files to a single parquet file, limiting row groups to 1M records, for a total file size of 2.5 GB.
3. **H2O-G**[51], models operations commonly found in data science workloads. We run the `groupby` task queries on the `G1_1e7_1e2_5_0.csv` dataset, a single 488 MB Comma Separated Value (CSV) file with 10M records.

We run all benchmarks directly on the raw source data files. While transforming and loading into specialized per-database formats is typical for previous generations of systems, we believe it is becoming increasingly impractical as data flows become more fluid and dynamic. For our target systems, data is most commonly read and written using open formats using a diverse ecosystem of tools.

We measured performance of DataFusion 32.0.0 and DuckDB 0.9.1, the most recently released versions as of the time of this writing, using their respective Python bindings. Our evaluation scripts are available online [3].

To evaluate end-to-end performance, we measured single raw per-core efficiency in Section 8.1 and multi-threaded scalability in Section 8.2. We limited the cores used by DataFusion by setting `target_partitions` and for DuckDB we set the `threads` `PRAGMA`.

8.1 Single Core Efficiency

To measure how efficiently each engine uses CPU to compute query results, we ran each query limited to a single core, on an `e2-standard-8` instance on Google Cloud Platform. This instance used the Intel Broadwell microarchitecture, had 32 GB of RAM and 8 virtual cores. We ran `Ubuntu 22.04.3 LTS` and Linux kernel version `6.2.0-1013-gcp`.

ClickBench: Table 1 shows query execution time for the ClickBench queries. DataFusion performs better on queries that have highly selective predicates such as Q2, Q8, and Q20 likely due to its ability to push predicates into the parquet scan to skip entire row groups. DataFusion also does well for queries with a single group such Q4 and Q7 and Q30, likely due to its vectorized aggregate updates. For queries with medium selectivity and medium group cardinally, such as Q15, Q31, Q32, Q41 and Q42 the engines are similar in performance. For queries that have high group cardinally (10M groups or more) such as Q18, Q19, Q36, DuckDB performs better, likely due to its highly optimized parallel group by aggregation[44].

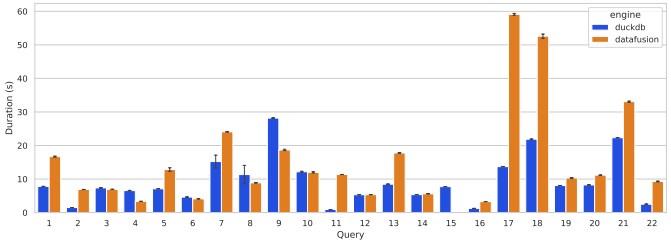

Figure 5. TPC-H SF=10 performance on a single core, one parquet file per table.

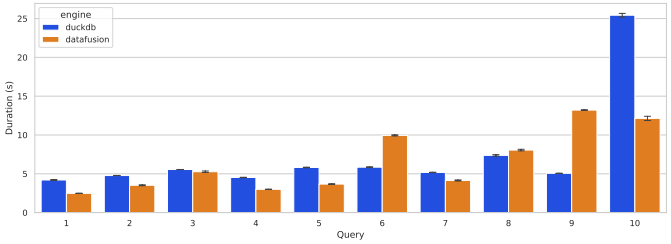

Figure 6. H2O-G (grouping) performance on single core with a single 488MB CSV file.

TPC-H: Figure 5 shows query execution time for TPC-H queries. Unlike ClickBench, most queries in TPC-H join several tables. DataFusion is faster for some queries such as Q4 and Q9, with highly selective predicates. There are some queries where performance is roughly equal such as Q3, Q6 and Q14. There are also several queries where DataFusion is well over 2x slower, such as Q11, Q17, Q18, and Q21. Much of this largest differences is due to a suboptimal join order[4], and when we manually force a better join order, the performance of the two systems is similar.

H2O-G Figure 6 shows query execution time for the H20-G queries. DataFusion has slightly better performance for most queries, though is significantly worse for Q9, due to an inefficient implementation of the `corr` aggregate function. The performance of all queries is largely dominated by the time spent parsing the CSV file, and DataFusion benefits from the highly optimized CSV parser included in the Rust implementation of Apache Arrow. Limiting to a single core may also unfairly penalize DuckDB in this case, which seems

[3] https://github.com/JayjeetAtGithub/datafusion-duckdb-benchmark

[4] https://github.com/apache/arrow-datafusion/issues/7949

to optimize multi-threaded parsing[5], while a similar trade off doesn't exist for DataFusion.

Discussion Both engines perform similarly using a single core, with different strengths and weaknesses depending on attributes of the particular query. We conclude there is nothing about using open standards that fundamentally limits DataFusion's performance. Our intuition and experience implementing industrial systems is that the determining factor is instead available engineering investment. DataFusion's community already has projects underway to improve performance for query patterns where it lags DuckDB in these benchmarks, such as join ordering[6] and high cardinally grouping[7] and Likewise, we expect that DuckDB's performance will improve in areas where it lags DataFusion such as low cardinally grouping, parquet predicate pushdown, and CSV parsing with additional investment.

8.2 Scalability

DataFusion is often used as a single-node engine, or the embedded engine in distributed systems, so its ability to scale "up" and use the resources of multiple cores is important. Figure 7 plots how performance varies with increasing core count. We ran each ClickBench query 5 times, varying the number of cores from 1 to 192, plotting the final 3 runs to remove any caching or warm-up effects. We ran this experiment on the highest end CPU available to us on Google Cloud Platform, a `c3-highcpu-176` instance with the Intel Sapphire Rapids micro-architecture, 176 virtual CPUs (cores), and 352 GB of memory. We ran all experiments using Ubuntu 22.04 with Linux kernel version `6.2.0-1016-gcp`.

Relative performance The absolute value of the y-axis is important. Some queries like Q10 take seconds to execute while other queries like Q1 take less than a second. Thus even while the relative performance difference between the two engines may appear substantial in some queries, such as Q1-Q4 or Q37-Q42, the absolute difference is 100s of milliseconds, while the absolute difference in queries such as Q19, Q32 and Q33 is an order of magnitude higher.

1, 2, 3, 8, 16, 34 Cores Up to 32 cores, both DataFusion and DuckDB show excellent, near-linear decrease in execution times as the core counts increase.

64, 128, 192 cores At higher core counts, both engines show a mix of better and worse performance. In Q28 and Q29, performance continues to improve as the core count increases, close to the ideal curve. These queries contain low (6000) and medium (3M) cardinality grouping operations and require significant CPU effort to evaluate `LIKE` string matching predicates. In queries such as Q11, Q14, and Q32 both engines show a pronounced *increase* in query duration (they slow down) with more cores, likely because as the work done by each core decreases, the relative overhead of coordinating between the cores increases. In queries such as Q41, Q42 and Q43, the slowdown at high core count is more pronounced for DataFusion[8], and in some queries such as Q25 and Q26 it is more pronounced for DuckDB.

Discussion DataFusion and DuckDB exhibit similar scaling behavior, and thus we conclude DataFusion's modular design and pull based scheduler do not preclude state of the art multi-core performance. The curves in Figure 7 for both engines are similar in shape, suggesting performance differences are largely due to implementation details rather than any fundamental differences in design.

9 Related Work

The theme of more modular and composable architectures was observed at least as early as 2000[13], the term "Deconstructed Database" was initially popularized[43] in 2018, and there are recent calls to accelerate modular design[59].

The Velox[58] and the Apache Calcite[8] projects both provide components for assembling new databases and analytic systems. However, building a working end-to-end system requires substantial integration (e.g. bridging JVM and Native code and build systems), while using DataFusion requires a single configuration line. Modular designs allow swapping components based on use case, and the Photon[9] and Gluten[61] (based on Velox) projects replace just one module, the execution engine, of Apache Spark with a faster native implementation.

Similarly to DataFusion, DuckDB[63] is an open-source SQL system that does not require a separate server. DuckDB is targeted at users who run SQL, while DataFusion is targeted at people building new systems (that may run SQL as well as other types of processing). DuckDB has a more limited extension API and its own custom in-memory representation, storage format, Parquet implementation, and thread scheduler.

9.1 Future Research

We believe there is a need for modular systems like DataFusion to accelerate other areas of database implementation, such as transaction processing and distributed key/value stores. First class support, either as bindings to DataFusion or separate implementations, for other systems languages like C/C++ and Swift, are also needed.

10 Conclusion

Since the introduction of LLVM, the necessity to build compilers from scratch has significantly diminished. With the emergence of technologies like DataFusion, the need to construct

[5] https://github.com/duckdb/duckdb/issues/9136
[6] https://github.com/apache/arrow-datafusion/issues/7949
[7] https://github.com/apache/arrow-datafusion/issues/5546

[8] Some of the slowdown in DataFusion is due to a poorly tuned hash table flushing strategy for high cardinalities https://github.com/apache/arrow-datafusion/issues/6937

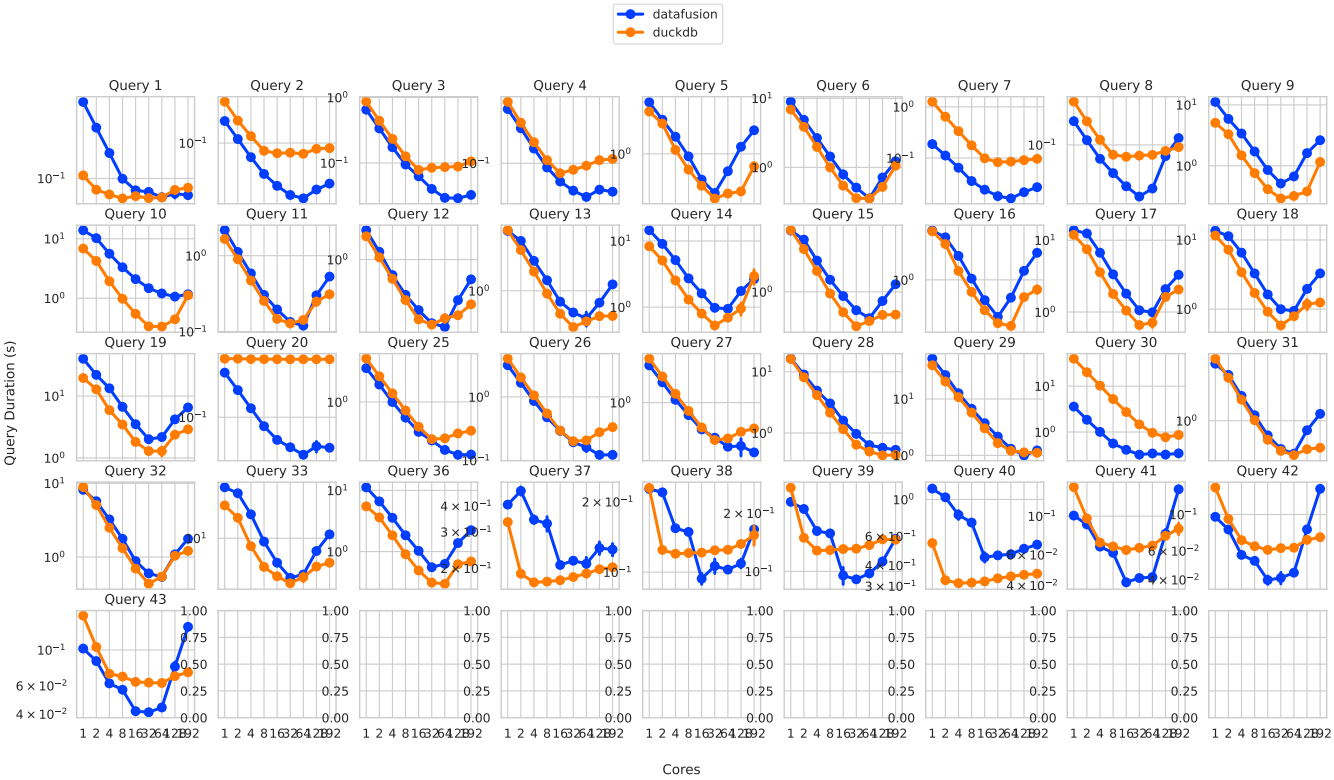

Figure 7. Query duration for ClickBench queries using 1, 2, 4, 8, 16, 32, 64, 128, or 192 threads, respectively.

database systems from the ground up should become similarly rare. Of course, with sufficient engineering investment, a tightly integrated engine can theoretically outperform a modular one. However, as the effort to reach state-of-the-art functionality and performance increases ever more, we believe that widely used, modular engines such as DataFusion can attract mass investment from open-source communities to offer a richer feature set and better performance than all but the most well-resourced tightly integrated designs.

Modular designs are by no means the only strategy for building systems, and we continue to see new tightly integrated systems emerge. However, as awareness of systems such as DataFusion increases, we predict adoption will accelerate and an explosion of new analytic systems will emerge that would previously not have been possible.

11 Acknowledgments

DataFusion is a community-driven project encompassing a diverse array of individuals over a considerable span of time. At the time of writing, DataFusion had over 4600 proposed contributions (Pull Requests) from over 300 distinct members. The authors thank all community members, who contributed the ideas, reviews, bug reports, code, and tests over the years, and made DataFusion possible.

References

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. 466–475. https://doi.org/10.1109/ICDE.2007.367892

[2] Daniël Heres Andrew Lamb, Raphael Taylor-Davies. 2023. *Aggregating Millions of Groups Fast in Apache Arrow DataFusion.* https://www.influxdata.com/blog/aggregating-millions-groups-fast-apache-arrow-datafusion

[3] Raphael Taylor-Davies Andrew Lamb. 2022. *Querying Parquet with Millisecond Latency.* https://arrow.apache.org/blog/2022/12/26/querying-parquet-with-millisecond-latency/

[4] Raphael Taylor-Davies Andrew Lamb. 2023. *Fast and Memory Efficient Multi-Column Sorts in Apache Arrow Rust.* https://arrow.apache.org/blog/2022/11/07/multi-column-sorts-in-arrow-rust-part-1/

[5] Inc Apple. 2023. *The Swift programming language.* https://developer.apple.com/swift/

[6] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[7] Arroyo. 2023. *Arroyo - Serverless Stream Processing.* https://www.arroyo.dev/

[8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of*

Data (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[9] Alexander Behm and Shoumik Palkar et al. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[10] The Blaze. 2023. *The Blaze accelerator for Apache Spark*. https://github.com/blaze-init/blaze

[11] Tyler A. Cabutto, Sean P. Heeney, Shaun V. Ault, Guifen Mao, and Jin Wang. 2018. An Overview of the Julia Programming Language. In *Proceedings of the 2018 International Conference on Computing and Big Data* (Charleston, SC, USA) *(ICCBD '18)*. Association for Computing Machinery, New York, NY, USA, 87–91. https://doi.org/10.1145/3277104.3277119

[12] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1773–1786. https://doi.org/10.1145/3299869.3314045

[13] Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1–10.

[14] The Rust community. 2023. *Cargo: Rust's built-in package manager*. https://crates.io/

[15] Coralogix. 2023. *Coralogix - Full-Stack Observability Platform with In-Stream Data Analytics*. https://coralogix.com

[16] IBM Corporation. 2023. *IBM DB2*. https://www.ibm.com/products/db2

[17] Microsoft Corporation. 2023. *Microsoft SQL Server*. https://www.microsoft.com/en-us/sql-server

[18] Oracle Corporation. 2023. *The Oracle Database Server*. https://www.oracle.com/database/

[19] The Transaction Processing Council. 2023. *The TPC-H Benchmark*. https://www.tpc.org/tpch/

[20] Voltron Data. 2023. *The Composable Codex*. https://voltrondata.com/codex

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[22] Arrow developers. 2023. *Mailing list: [DISCUSS][Format] Starting the draft implementation of the ArrayView array format*. https://lists.apache.org/thread/r28rw5n39jwtvn08oljl09d4q2c1ysvb

[23] Substrait Developers. 2023. *Substrait: Cross-Language Serialization for Relational Algebra*. https://substrait.io/

[24] David J. DeWitt and Michael Stonebraker. 2008. *MapReduce: A major step backwards*. https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

[25] Apache Software Foundation. 2023. *Apache Arrow*. https://arrow.apache.org

[26] Apache Software Foundation. 2023. *Apache Arrow DataFusion*. https://arrow.apache.org/datafusion/

[27] Apache Software Foundation. 2023. *Apache Parquet*. https://parquet.apache.org

[28] Apache Software Foundation. 2023. *How the ASF Works*. https://www.apache.org/foundation/how-it-works/

[29] Apache Software Foundation. 2023. *A Primer on ASF Governance*. https://www.apache.org/foundation/governance/

[30] Apache Software Foundation. 2023. *PyArrow - Apache Arrow Python bindings*. https://arrow.apache.org/docs/python/index.html

[31] The Apache Software Foundation. 2023. *Apache DataFusion SQL reference*. https://arrow.apache.org/datafusion/user-guide/sql/index.html

[32] The Zig Software Foundation. 2023. *The Zig programming language*. https://ziglang.org/

[33] Rust futures crate. 2023. *Stream trait*. https://docs.rs/futures/0.3.28/futures/prelude/stream/trait.Stream.html

[34] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, USA) *(SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 102–111. https://doi.org/10.1145/93597.98720

[35] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3 (sep 2006), 10–es. https://doi.org/10.1145/1132960.1132964

[36] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (01 2012).

[37] ClickHouse Inc. 2023. *ClickBench — a Benchmark For Analytical DBMS*. https://benchmark.clickhouse.com/

[38] InfluxData Inc. 2023. *The Influx Query Language Specification*. https://github.com/influxdata/influxql

[39] Inc. InfluxData. 2023. *Announcing InfluxDB IOx - The Future Core of InfluxDB Built with Rust and Arrow*. https://www.influxdata.com/blog/announcing-influxdb-iox/

[40] Inc. InfluxData. 2023. *InfluxDB — open source time series, metrics, and analytics database*. https://influxdata.com/

[41] ISO/IEC 9075:2023 2023. *Information technology - Database languages - SQL*. Standard. International Organization for Standardization.

[42] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (sep 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

[43] Amandeep Khurana and Julien Le Dem. 2018. The Modern Data Architecture: The Deconstructed Database. *login Usenix Mag.* 43, 4 (2018). https://www.usenix.org/publications/login/winter-2018-vol-43-no-4/khurana

[44] DuckDB Labs. 2023. *Parallel Grouped Aggregation in DuckDB*. https://duckdb.org/2022/03/07/aggregate-hashtable.html

[45] Andrew Lamb. 2022. *Using Rustlang's Async Tokio Runtime for CPU-Bound Tasks*. https://thenewstack.io/using-rustlangs-async-tokio-runtime-for-cpu-bound-tasks/

[46] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[47] Lance. 2023. *Lance: modern columnar data format for ML*. https://lancedb.github.io/lance/

[48] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[49] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[50] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proc. VLDB Endow.* 8, 10 (jun 2015), 1058–1069. https://doi.org/10.14778/2794367.2794375

[51] Database like ops benchmark. 2023. *H2O.ai.* https://h2oai.github.io/db-benchmark/

[52] Jon Mease. 2023. *VegaFusion: serverside scaling for the Vega visualization library.* https://vegafusion.io/

[53] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[54] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[55] Ramon E Moore. 1966. *Interval analysis.* Vol. 4. Prentice-Hall Englewood Cliffs.

[56] Mehmet Ozan Kabak Mustafa Akur. 2023. *Running Windowing Queries in Stream Processing.* https://www.synnada.ai/blog/running-window-query-in-stream-processing

[57] The pandas development team. 2020. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134

[58] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[59] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (aug 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[60] The Dask project. 2023. *The dask-sql project.* https://dask-sql.readthedocs.io/en/latest/

[61] The OAP project. 2023. *Gluten: Plugin to Double SparkSQL's Performance.* https://h2oai.github.io/db-benchmark/

[62] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3209950.3209955

[63] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[64] Tokio rs Developers. 2023. *Tokio: As asynchronous Rust runtime.* https://tokio.rs/

[65] SDF. 2023. *SDF.* https://www.sdf.com/engine

[66] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. 2013. Materialization strategies in the Vertica analytic database: Lessons learned. In *2013 29th IEEE International Conference on Data Engineering (ICDE 2013)*. IEEE Computer Society, Los Alamitos, CA, USA, 1196–1207. https://doi.org/10.1109/ICDE.2013.6544909

[67] The sqlparser-rs authors. 2023. *sqlparser-rs: Extensible SQL Lexer and Parser for Rust.* https://github.com/sqlparser-rs/sqlparser-rs

[68] Michael Stonebraker. 2008. Technical perspective - One size fits all: an idea whose time has come and gone. *Commun. ACM* 51, 12 (2008), 76. https://doi.org/10.1145/1409360.1409379

[69] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[70] Synnada. 2023. *Synnada realtime data platform.* https://www.synnada.ai/

[71] The Rust team. 2023. *The Rust programming language.* https://www.rust-lang.org/

[72] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664