The Composable Data Management System Manifesto

Pedro Pedreira
Meta Platforms Inc.
pedroerp@meta.com

Orri Erling
Meta Platforms Inc.
oerling@meta.com

Konstantinos Karanasos
Meta Platforms Inc.
kkaranasos@meta.com

Scott Schneider
Meta Platforms Inc.
scottas@meta.com

Wes McKinney
Voltron Data
wes@voltrondata.com

Satya R Valluri
Databricks Inc.
satya.valluri@databricks.com

Mohamed Zait
Databricks Inc.
mohamed.zait@databricks.com

Jacques Nadeau
Sundeck
jacques@sundeck.io

ABSTRACT

The requirement for specialization in data management systems has evolved faster than our software development practices. After decades of organic growth, this situation has created a siloed landscape composed of hundreds of products developed and maintained as monoliths, with limited reuse between systems. This fragmentation has resulted in developers often reinventing the wheel, increased maintenance costs, and slowed down innovation. It has also affected the end users, who are often required to learn the idiosyncrasies of dozens of incompatible SQL and non-SQL API dialects, and settle for systems with incomplete functionality and inconsistent semantics. In this vision paper, considering the recent popularity of open source projects aimed at standardizing different aspects of the data stack, we advocate for a paradigm shift in how data management systems are designed. We believe that by decomposing these into a modular stack of reusable components, development can be streamlined while creating a more consistent experience for users. Towards that goal, we describe the state-of-the-art, principal open source technologies, and highlight open questions and areas where additional research is needed. We hope this work will foster collaboration, motivate further research, and promote a more composable future for data management.

PVLDB Reference Format:
Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. The Composable Data Management System Manifesto. PVLDB, 16(10): 2679 - 2685, 2023. doi:10.14778/3603581.3603604

1 INTRODUCTION

The increasing workload diversity in modern data use cases has led to the proliferation of specialized data management systems, each targeted to a somewhat narrow class of workloads. Based on the *"one size does not fit all"* engine specialization tenet, hundreds of database system offerings were developed in the last few decades and are today available in the industry. While workloads, requirements, and environmental trends have dramatically evolved since the first databases were developed, our software development practices have not; data management systems continue to be, by and large, developed and distributed as vertically integrated monoliths.

While modern specialized data systems may seem distinct at first, at the core, they are all composed of a similar set of logical components: (a) a **language frontend**, responsible for interpreting user input into an internal format; (b) an **intermediate representation** (IR), usually in the form of a logical and/or physical query plan; (c) a **query optimizer**, responsible for transforming the IR into a more efficient IR ready for execution; (d) an **execution engine**, able to locally execute query fragments (also sometimes referred to as the *eval engine*); and (e) an **execution runtime**, responsible for providing the (often distributed) environment in which query fragments can be executed. Beyond having the same logical components, the data structures and algorithms used to implement these layers are also largely consistent across systems. For example, there is nothing *fundamentally different* between the SQL frontend of an operational database system and that of a data warehouse; or between the expression evaluation engines of a traditional columnar DBMS and that of a stream processing engine; or between the string, date, array, or json manipulation functions across database systems.

However, this fragmentation and consequent lack of reuse across systems has slowed us down. It has forced developers to reinvent the wheel, duplicating work and hurting our ability to quickly adapt systems as requirements evolve. Our development model still leads to siloed systems, high maintenance costs, and wasted engineering cycles, suggesting we can be more efficient as an engineering community. More importantly, the byproducts of this fragmentation — incompatible SQL and non-SQL APIs, disparate functionality, distinct function packages, and inconsistent semantics across the board — impact the productivity of end users who are commonly required to interact with multiple distinct data systems to finish a particular task, each with their own quirks.

We believe it is time for a paradigm shift. We envision that by decomposing data management systems into a more modular stack of reusable components, the development of new engines can be streamlined, while reducing maintenance costs and ultimately providing a more consistent user experience. By clearly outlining APIs and encapsulating responsibilities, data management software could more easily be adapted, for example, to leverage novel devices and accelerators, as the underlying hardware evolves. By relying on a modular stack that reuses execution engine and language frontend, data systems code could provide a more consistent experience and

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603604

semantics to users, from transactional to analytic systems, from stream processing to machine learning workloads.

Why now? In the last decade, the ubiquity of clouds and disaggregation of computation from storage has caused a major inflection in the design of data management systems. As vendors increased the emphasis on the delivery of services rather than on proprietary software, open source big data technologies and open standards such as Apache Arrow, Orc, Parquet, Hudi, and Iceberg emerged, and were rapidly popularized. Recently, projects like Velox, Substrait, and Ibis quickly gained adoption across the industry, allowing modern data stacks to be assembled by only reusing available parts. Given these trends, we foresee that composability is soon to cause another major disruption to how data management systems are designed. We foresee that monolithic systems will become obsolete, and give space to a new composable era for data management.

In this vision paper, based on the authors' experience as founders and developers of many of the aforementioned projects [6] [27] [29] and on componentizing some of the largest data systems in the world [4], we make the following contributions:

- We highlight the importance of composability in data management systems and argue that now is the right moment for a paradigm shift.
- We summarize previous work that describe individual projects in the composability space, their significance so far, and the investments required going forward.
- We extend the state-of-the-art by presenting a novel reference composable architecture, and discussing the parts of this stack that have received less attention but are key to component composability and reusability, highlighting open questions and areas that require additional research.

2 COMPOSABILITY IN DATA MANAGEMENT

Decomposing software complexity into smaller subsets of relatively independent components is a well-known software design technique. The state-of-the-art dictates that components should be deep, encapsulating as much complexity as possible, and that their APIs should be narrow, minimizing dependencies across components and preventing implementation details from leaking through the API. We believe that increasing the degree of composability in data management systems, by developing and adopting reusable components, provides the benefits discussed below.

Engineering efficiency. By reducing the duplication of work, more engineers could work on fewer systems and components. This saves us from re-inventing the wheel, consolidates domain-specific knowledge into fewer specialized teams, and ultimately enables engineering organizations to be more efficient and move faster.

Faster innovation. For large organizations, having fewer codebases reduces operational burden, and allows engineering teams to focus on new features, optimizations, and other enhancements. For small companies, it decreases their time-to-market, considering that many components do not need to be re-developed [17], leveling the field and allowing them to compete with larger database vendors.

Coevolution. Data system diversity disincentivizes hardware vendors from investing in hardware support for data processing. Unifying the physical execution layer, for example, could enable a more synergistic relationship with the semiconductor industry through the manufacture of custom chips that are more efficient for database workloads. This would enable database software and hardware to evolve closer together.

Better user experience. By reusing the same components, from language to execution, users can expect consistent semantics across data systems, in addition to a more even set of available features. This reduces the user's learning curve and cognitive burden, increasing their productivity.

Despite the benefits, developing components in a way that they can be shared and reused across different projects is a challenging task. Based on the authors' own experience and observations, some of the frequent reasons for lack of reuse are:

Learning curve. Libraries and frameworks are complicated and take time to learn. Even when developed using the same programming language, many libraries use inconsistent nomenclature, coding standards, tooling, build systems, processes, and provide inconsistent APIs which are not interoperable with each other.

Developer bias. Developers are often passionate about writing their own code, and commonly find reading documentation and reviewing someone else's code to be a tedious process.

Time-to-market fallacy. It is also common for developers to believe that a quick prototype containing a subset of functionality decreases the time-to-market for their products. In cases where this holds true, it frequently understates the high cost of stabilizing (hardening) the software, and the long-tail of features required to turn the prototype into a real product. Ultimately, time-to-market does not simply depend on writing the code, but on stabilizing it against a real workload. This usually results in products with incomplete and inconsistent features, hard to maintain (once the engineers who wrote the prototype move to a different project), and generalized tech debt.

Close fit. In many cases, a component that provides the required functionality exists, but is written in the wrong programming language, has too many dependencies, is too hard to use, or is distributed under an incompatible license. In other situations, the third-party component would have to be modified, but the project maintainers are not receptive to the changes. In summary, a component is reusable only if it does not have to be modified in any significant way outside of using its extensibility APIs.

Lack of incentives. Developers are usually not compelled to write reusable components because there are few incentives to do so. From an individual developer's perspective, it takes more effort to develop a modular system than to develop a monolith, and it is more difficult to build a business model for a data management system component than it is for an end-to-end system. In the short-term, it is usually easier for a particular group to develop their own system and internal components (*local optimum*), than to share with other groups, reuse, and collaborate (*global optimum*).

Although providing an ultimate solution for the issues described above is outside of the scope for this work, we believe the path forward is to focus on the following principles: (a) define and agree on a standard set of logical components across data management systems, (b) define stable (yet extensible) APIs for communication between these components, (c) provide canonical implementations for these components and APIs which are efficient and consistent, and (d) provide extensibility APIs in every layer of the stack to allow developers to implement specialized behavior.

3 A MODULAR DATA STACK

First and foremost, the new modular data stack emerging in the open source community provides a stronger separation between language and execution, in such a way that the execution is language-independent, and takes a well-defined and system-agnostic intermediate representation (IR) as input. IRs are generated by a language component, which is responsible for parsing and analyzing user queries, and serve as input for a query optimizer. Query optimizers are built using a universal (but extensible) framework, and ultimately generate IR fragments which are ready for execution. Figure 1 illustrates the outline of this data stack.

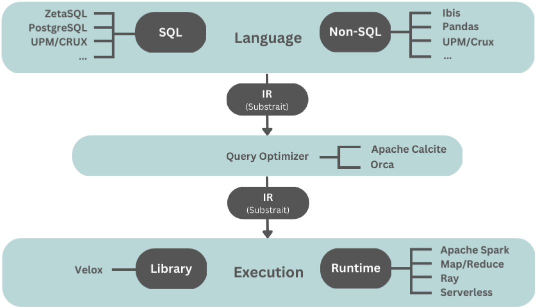

Figure 1: Open source modular data stack outline.

Fragments are executed using an execution engine that provides general and efficient storage primitives, including data layout formats (columnar and row-wise) and data access methods. Lastly, the execution of these self-contained fragments is orchestrated by an execution runtime, responsible for deployment, stats collections, monitoring, failure handling, resource management, and distributed orchestration of the entire computation.

Based on this outline, we make two claims:

(1) This model is *general enough*, and allows every existing data management system to be mapped to it, from OLTP to OLAP systems, stream processing, log analytics, ML preprocessing and more. In a few real-life systems some of these components might be missing, like ML preprocessing libraries commonly lacking an optimizer, or single-node systems lacking a full-fledged execution runtime, but the premise still holds.

(2) These components are *predominantly consistent* across specialized data management systems, and the areas where they specialize/diverge are the exception rather than the norm. For instance, stream processing systems require specialized logic to handle checkpoints/barriers and watermarks, but the rest of the engine is equivalent to an OLAP execution engine.

In the remainder of this Section, we detail each of these components (language, IR, optimizer, execution, and runtime), explore the state-of-the-art, describe existing leading projects and how they fit into this framework, and raise open questions hoping to motivate further research in this topic.

3.1 Language Frontend

Despite the current fragmentation, language frontend is the most straightforward layer in the stack to be made composable due to its simple API: translate user input into an IR. In fact, Google's GoogleSQL (open sourced as ZetaSQL [30]) and more recently Meta's CoreSQL [4] are two successful projects aimed at consolidating language frontends across multiple data systems in large organizations, through the use of a unified C++ parser and analyzer library. PostgresSQL's parser can also be used as a standalone library, and is already leveraged by modern systems like DuckDB [22].

At the core, these libraries are all similar: they provide a C/C++ parser and analyzer library that is able to translate a SQL statement (or other DSLs) into an IR, usually doing so by leveraging grammar and tokenization libraries such as Flex, Bison or Antlr. This process commonly requires APIs for table, column, and function signature resolution (for type binding) to be implemented. In addition to simple tokenization, parsing, and type binding capabilities, we believe a state-of-art language frontend library should also provide support for the features below, which are increasingly relevant:

- **Semantic Types:** Modern language frontends should allow users to provide additional meaning to their data through the use of richer *semantic types*. These user-defined types do not affect how data is stored or processed (i.e., are invisible to the IR or execution layer), but improve the capabilities of static type-checking. For example, comparisons of two integers representing different concepts (*UserID* and *DeviceID*) can be statically avoided during type checking. Similarly, the handling of data in different units (e.g, *TimestampMs* and *TimestampSec*) can also be transparently prevented or normalized.
- **Type Checked Macros:** SQL queries that are hundreds of lines long are common in modern data warehouses. SQL macros provide a static way to both improve the abstraction level of large queries and enable code reuse. Stored procedures and prepared statements partially fill this role, but defining and using them requires both special syntax and execution layer support.
- **IDE Interoperability:** When the frontend is a standalone library, it can provide additional APIs to allow IDE developers to build a richer set of SQL authoring features, such as improved syntax highlighting, token predictions, static type checking, and autocompletion support, all in a consistent manner by leveraging the exact same parsing/analyzer library used by the data management system underneath.

Non-SQL APIs. While SQL's declarative nature is convenient for humans, it can be cumbersome to imperative programs. This need gave birth to a series of dataframe-like APIs and other DSLs, offering a more programmatic way to express the same type of computation without requiring error-prone concatenations of pieces of SQL statements. This diversity resulted in a fragmented language landscape, where some systems provide dataframe APIs [16] [20] [9] (common in data science applications), and many provide SQL APIs (traditional DBMSs) [8] [7]. In fact, some open source projects such as Ibis [21] were created to bridge this very gap and translate dataframe APIs into SQL for execution in traditional data management systems. We believe cross-language translation to be a poor architecture to address language fragmentation (see "challenges" in Section 3.2).

Extending on the model popularized by Spark [25], where both SQL (SparkSQL) and non-SQL (DataFrames, Datasets, and PySpark) APIs generate the same IR, we believe that, more generally, language will meet execution through a unified IR. Different language frontend libraries can be provided to interface with specialized DSLs, but should ultimately generate an IR that can be universally and consistently executed. For instance, these two inputs should provide an equivalent IR and be indistinguishable from an execution standpoint:

```
SELECT a, b, c FROM table WHERE a = 1;
Table.where("a=1").select("a,b,c");
```

As non-SQL APIs find increased adoption, we also believe that higher-level APIs, platforms, and frameworks used by database application developers (such as ORM) will evolve, due to two main reasons. First, traditional frameworks were originally designed assuming SQL to be the only API provided by data management systems, while modern non-SQL imperative languages are more flexible and offer a wider range of capabilities. Second, as language gets componentized and decoupled from execution, application frameworks could evolve into an entirely new language component. Similar to other language libraries, it could communicate directly with the underlying database system via a structured IR, bypassing any (potentially narrower) intermediate language APIs. However, the details of this evolution is still an open question.

Language Unification. We believe language frontend modularization to also pave the way for language unification, or supporting a single unified SQL dialect, and a single unified dataframe dialect across data management systems. Language unification removes the cognitive burden inherent to switching between incompatible language dialects, decreases vendor lock-in, and enables applications to be made more portable. Although language unification can only be realized with proper IR and execution support (discussed in Sections 3.2 and 3.3), efforts like GoogleSQL at Google [30] and CoreSQL at Meta [4] have set a feasibility precedent.

3.2 Intermediate Representation

Intermediate representation (IR) is a term commonly used in the field of compilers to describe any structured representation of a program that carries enough information to allow it to be accurately executed, usually serving as the common interface between its components [5]. In data management systems, IRs provide a structured representation of a query and serve as the bridge between language and execution.

Data management systems have historically defined their own internal and individual IRs (logical and/or physical plans), because decoupling language and execution was never a goal. Though IRs in current data systems are tightly coupled with their internals, in practice they are only different representations of the same data processing primitives. They all represent expression trees, containing function calls, table references, and literals, and traditional SQL operations such as filtering, projections, ordering, joining, aggregations, windowing, shuffling/repartitioning, unnesting, and more, with little variation beyond that.

Substrait [27] is a recent pioneering effort at providing a unified and cross-language IR specification. Substrait provides a standardized IR for data management systems, with the purpose of creating a *lingua franca* to describe computation. The standard describes common functionality found in data management systems, establishing a clear delineation between specifications that must be respected and the ones that can be optionally ignored, hence accommodating systems with different physical capabilities. Substrait also allows systems to privately or publicly extend the representation to support custom/specialized operations.

Challenges. IR unification (through Substrait or otherwise) is a leap forward in data management systems architecture, as it enables language and execution to be fully decoupled, componentized, and made interoperable. However, a few challenges arise as we work towards making this unification practical in real-life data systems.

First, in order to allow data systems to execute IRs generated by foreign components, IRs will need to become part of the system's external API. This results in less flexibility while evolving the IR representation, since any IR changes need to be backwards compatible and properly versioned. But more importantly: taking a full-fledged IR as input will, by definition, increase the input domain of a data system. The cross-language character of an IR also implies that the computations an IR can represent go beyond computations which are expressible through SQL APIs - and might result in revealing dormant bugs or other limitations. Some recent efforts try to avoid this situation by serializing IRs into a SQL statement (described in Section 3.1). We believe this to result in a poor architecture, as it loses the additional expressiveness, flexibility, and preciseness provided by an IR by funneling it through a narrower API (SQL), which is subject to dialect peculiarities.

Second, current IRs are not yet descriptive enough to ensure *runtime semantic equivalence*. For instance, a system could silently ignore integer overflows while others might throw exceptions; or one can provide 0-index array semantics while others could be 1-index based [23] [26]. While these system characteristics are captured in a Substrait IR, we believe representing every nuance of existing systems will prove to be challenging - and yet, necessary to ensure correct and equivalent execution across systems.

Lastly, the set of functions available today in different systems are fully disparate. In cases where systems provide functions that look similar, they rarely provide the exact same, *bug-by-bug*, semantic. In order to ensure correct execution, functions need to be globally identifiable by a URI that controls not only their name, but also the base implementation they refer to, and some notion of versioning (e.g. $map\_entries()$ from SparkSQL, version 124). Executing an IR referencing this function in a system other than SparkSQL, for instance, should fail. In practice, this prevents different execution engines from being interoperable at an IR level. We believe the path forward to be, in the future, to bypass dialect and function packages incompatibilities through language and execution unification.

3.3 Query Optimization

Query optimization is a very diverse and well-studied field [14]. While the majority of the industry query optimizers are tailored to the target database system, there has been a significant body of work related to building extensible and composable query optimizers. The goal of extensibility in query optimization is to provide abstractions and integration hooks to allow replacing or adding new features in order to support new use-cases. On the other hand, the goal of composability is to make it easier to port the optimizer into a different system than the one where it was initially built for.

Orca [24] and Apache Calcite [1] are the most known attempts in the area of composable and reusable optimizers. Orca provides a clear separation between the optimizer and execution engine by using an XML-based language to exchange information between the two. While Orca is designed to be modular and extensible, it was reportedly non-trivial to integrate it with non-PostgreSQL systems [12]. Apache Calcite has been successfully integrated into several open source projects like Apache Hive and Apache Phoenix, stream processing engines like Apache Flink and Apache Samza, and commercial systems like Qubole. In addition to the optimizer, Apache Calcite provides a full language frontend and IR, in such a way that users can either provide a SQL statement or Calcite's own structured IR as input. The overhead incurred by translating between IRs (from the underlying target system to Calcite and vice-versa) and the fact that it is written in Java, makes it challenging to embed Calcite into non-Java systems where support for efficient short running queries is important. Bridging programming language gaps into a lighter-weigh library and basing it on a unified IR, such as Substrait, would satisfy most of the criterion of a composable query optimizer, and potentially further increase its adoption.

3.4 Execution Engine

Execution is the layer responsible for taking a query fragment as input (represented by an IR), and executing it leveraging the local resources provided by the execution runtime (see Section 3.5). Execution for a particular query fragment usually starts with a table/index scan or exchange (shuffle) as input, and after processing the incoming data, ends with another exchange or table write. Common data processing primitives range from expression evaluation, filtering, ordering, joining, unnesting, and other operators required to implement SQL semantics.

Though the vast majority of operations applied to the data in current data management systems follow a few simple basic SQL operations (apart from possible engine-specific extensions), today, no two systems share the same execution codebase. Execution is a highly fragmented domain, posing challenges to large organizations which are required to individually maintain dozens of siloed and specialized codebases due to user workload diversity [4]. It also impacts users, considering that the set of SQL functions available are often engine/dialect-specific, and that the code fragmentation leads to subtle semantic differences across systems. For instance, an informal survey conducted at Meta identified at least 12 different implementations of the simple string manipulation function $substr()$, presenting different parameter semantics (0- vs. 1-based indices), null handling, and exception behavior [18].

In theory, a composable execution engine merely needs to provide ways to locally execute an IR, providing extensibility APIs where different data access methods and storage formats can be plugged in, in addition to APIs to allow for exchange boundary specialization. To become dialect-agnostic, this library also needs to provide extensibility APIs to allow for engine-specific and user-defined data types, functions (scalar, aggregates, window, and tabular), and operators. Even though composing at the IR level gives developers the highest degree of flexibility when implementing execution primitives, it still leads to the duplication of a substantial amount of components. For example, all execution engines need to define the dataset memory layout (based on Apache Arrow or otherwise), local resource management (memory pools and arenas, SSD and memory caching, CPU and thread pool allocations), and encoding/decoding of popular file formats.

Therefore, we expect composability in execution to happen on a more granular level, where a unified library will provide the common execution framework (a common local execution bus), and allow developers to extend and customize it. While the exact places where APIs and extensibility points should be created are open questions, we believe two overarching principles should hold. First, the composable and monolithic versions of a system should be equivalent in terms of performance. As described in [18], this can be achieved by drawing API boundaries outside of hot code paths (for example, making API calls per query or per batch, but not per record), so that the cost associated with crossing component boundaries is amortized and made negligible. And second, APIs need to be designed (and evolved) in a way such that innovation is not hindered.

A motivating factor for such an architecture is the rise of hardware accelerators as a way to circumvent the end of Moore's law. As GPUs, FPGAs, TPUs, and other ASIC accelerators become commonplace, extending different execution engines to accommodate every available accelerator becomes impractical, duplicates efforts, and ultimately wastes engineering resources. Similarly, building full-fledged execution engines based on specific accelerators perpetuates fragmentation [3] [15] [11]. We believe a unified execution engine will pave the way for accelerators to become pervasive in data management through operator specialization. For example, using this model, one could build a general GPU-accelerated execution engine by simply specializing the most frequently used operators for a workload (say, table scan and hash joins), and reuse the remaining CPU-based components provided by the unified library. Furthermore, having such a framework integrated into major data systems in the industry provides a compelling platform for hardware vendors, since a single integration would provide access to a number of different workloads/markets. We expect this model to both increase collaboration between hardware and data management practitioners, and provide incentives to further the development of database-specific hardware primitives.

Velox [18] is one project aimed at filling this gap. Velox is the first large-scale open source project aimed at providing a unified execution engine for data management systems. It provides reusable, extensible, and dialect-agnostic data processing components, and is currently integrated with more than a dozen data management system within Meta and elsewhere, including analytical query engines like Presto and Spark, stream processing platforms, data warehouse ingestion infrastructure, machine learning systems for data preprocessing and feature engineering, and many more. Velox demonstrates that it is possible not just to componentize execution, but also to unify it across stacks. While hardware acceleration experiments in Velox are still in early stages, the rapidly growing interest from both hardware vendors and data management system developers provides a validation of this model.

Furthermore, Velox's dialect-agnostic architecture means that it is possible to extend/customize it to implement different SQL dialects. While it allows for a drop-in execution engine replacement (by following the same semantic as the system being replaced), it also allows for SQL dialects to be unified across data management systems. Alongside language and IR unification, Velox allowed Meta to unify SQL dialects (CoreSQL) across analytical systems for interactive and batch execution, stream processing, and data preprocessing, reducing the burden on users and providing a more consistent experience across specialized data systems.

3.5 Execution Runtime

The *execution runtime* provides the environment needed by the execution engine to perform the computation. It is responsible for (a) the scheduling and allocation of resources, and (b) the containerization and proper isolation between tasks (due to parallel execution and/or multi-tenancy). In distributed settings, it also provides the distributed computation model and inter-node communication (e.g., shuffles). Below, we focus on distributed environments as they are more challenging and provide more opportunities for composability and reuse.

The distributed computation model has significantly evolved over the last few decades, ranging from MapReduce (and its Hadoop open source implementation) to frameworks that allow the computation of DAGs, such as Apache Spark/RDDs and Apache Tez. Recently, frameworks like Ray and Dask push the computation flexibility even further and allow arbitrary functions to be executed at every worker, offering tighter integration with the Python ecosystem and targeting data science and machine learning workloads. At the same time, systems like Apache Flink and Spark Streaming have been widely employed to support streaming applications.

Although only a few of these systems have become truly widely used, there is currently no standardization to the level of abstraction that strikes the best balance between ease of programming and tight control of execution. Nonetheless, we have witnessed an interesting development towards composability in recent years: many of these systems have decoupled their execution runtime from their original execution engine (typically written in Java) and are using high-performance vectorized execution engines (typically in C++). Examples include Databrick's Photon [2] and Apache Gluten [10], which combine Spark's execution runtime with Velox or Databrick's proprietary C++ engine.

Schedulers responsible for negotiating cluster resources, on the other hand, have a somewhat narrower API. In fact, Apache YARN [28], the resource manager created and then decoupled from Hadoop, can today be used with most runtimes, including Spark and Tez. Though there is no standardization related to task isolation and inter-task communication, there are usually two design choices: (a) tasks are either isolated (using cgroups or other types of containers) or allowed to share the same process space, and (b) shuffles are either stream-based or persisted to a local or remote filesystem. We believe the advent of serverless computing and the worker-as-a-service paradigm will present further opportunities, though their applicability to data management is currently being assessed [19].

To sum up, it is still an open question if runtimes could converge and provide more configurable modes for data management, but we believe one principle should hold true: runtimes should not be coupled with data management systems, and preferably be interchangeable through a standard API.

4 CALL FOR ACTION

With this work, we hope to incentivize developers to follow a different mindset when developing data management systems. First, we encourage developers to consider the described logical stack, and ask themselves: *which parts of this stack am I planning to specialize?* We foresee that in the future, as components become higher quality and more extensible, the answer, in many cases, will converge to **none**. In these cases, a full data management system could be built by merely assembling parts. Despite sounding idealistic, a reasonably functional stack can be built today by solely leveraging open source projects like Ibis (language), Substrait (IR), Calcite (optimizer), Velox (execution), and a distributed runtime such as Spark, Ray, or a serverless architecture.

Second, in cases where some form of specialized behavior is required, we encourage developers to ask themselves: *could this new functionality be implemented through a well-defined extensibility API?* For instance, if a new system is to provide specialized geospatial capabilities, it could be implemented using Velox's operator extensibility API. Alternatively, if the hypothetical proposition is a better query optimizer, one could fully replace the optimizer layer by a custom implementation, as long as the APIs are maintained, and keep the remaining layers of the stack intact.

Finally, for cases where the specialization cannot be implemented using the current stack and extensibility APIs, we encourage developers and researchers alike to ask themselves: *how could we improve the current APIs to make them more general?* As a recent example, Velox had specialized its columnar layout to allow for more efficient manipulation of strings and complex types in addition to more flexible encoding types [18], and later collaborated with the Arrow community to extend the columnar standard (the API) [13].

5 CONCLUSION

The rapid evolution of user workloads has driven the development of hundreds of specialized data management systems. Despite sharing many of the same architectural decisions, data structures, and internal data processing techniques, today, the degree of reuse between these systems is unsettlingly limited. This leads to duplication of work, high maintenance costs, decelerates innovation, and ultimately affects users who are required to interact with numerous systems with incompatible SQL and non-SQL APIs, incomplete feature sets, and generally inconsistent semantics.

In this vision paper, we advocated for a paradigm shift on how these systems are designed and developed. We presented the different components of a reference architecture, their APIs, and responsibilities, and discussed how recent popular open source projects fit this model. We believe that by componentizing data management systems, the pace of innovation can be accelerated. We believe composable is the future of data management, and hope more individuals and organizations will join us in this effort.

REFERENCES

[1] 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.

[2] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 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*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[3] BlazingSQL. [n.d.]. A lightweight, GPU accelerated, SQL engine built on the RAPIDS.ai ecosystem. https://github.com/BlazingDB/blazingsql.

[4] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian James Sun, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. *Conference on Innovative Data Systems Research (CIDR)* (2023).

[5] David Chisnall. 2013. The Challenge of Cross-Language Interoperability. 56, 12 (2013).

[6] The Apache Software Foundation. 2023. Apache Arrow: A cross-language development platform for in-memory analytics. Retrieved March 1, 2023 from https://arrow.apache.org/

[7] The PostgreSQL Global Development Group. [n.d.]. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[8] Oracle Inc. [n.d.]. MySQL. https://www.mysql.com/.

[9] Snowflake Inc. [n.d.]. Snowpark for Python, Java, and Scala. https://www.snowflake.com/en/data-cloud/snowpark/.

[10] Intel. 2023. Gluten: A Spark Plugin to Offload SQL Engine to Native Library. Retrieved March 1, 2023 from https://github.com/oap-project/gluten

[11] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013.

[12] Arunprasad P. Marathe, Shu Lin, Weidong Yu, Kareem El Gebaly, Per-Åke Larson, and Calvin Sun. 2022. Integrating the Orca Optimizer into MySQL. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. OpenProceedings.org, 2:511–2:523.

[13] Wes McKinney. [n.d.]. Adding new columnar memory layouts to Arrow. https://lists.apache.org/thread/49qzofswg1r5z7zh39pjvd1m2ggz2kdq.

[14] Guido Moerkotte. 2023. Building Query Compilers. Retrieved March 1, 2023 from https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[15] OmniSci. [n.d.]. OmniSci. https://docs.omnisci.com/.

[16] Pandas. [n.d.]. Pandas - Python Data Analysis Library. https://pandas.pydata.org/.

[17] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zait (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper1_MoshaPasumansky.pdf

[18] 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), 13.

[19] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. New York, NY, USA, 131–141.

[20] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046. http://www.vldb.org/pvldb/vol13/p2033-petersohn.pdf

[21] The Ibis Project. [n.d.]. The flexibility of Python analytics with the scale and performance of modern SQL. https://ibis-project.org/.

[22] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[23] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813.

[24] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. 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, 337–348. https://doi.org/10.1145/2588555.2595637

[25] Apache Spark. [n.d.]. Apache Spark - Unified Engine for large-scale data analytics. https://spark.apache.org/.

[26] Apache Spark. [n.d.]. Spark SQL And DataFrames. https://spark.apache.org/sql/.

[27] Substrait. 2023. Substrait: Cross-Language Serialization for Relational Algebra. Retrieved March 1, 2023 from https://substrait.io/

[28] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin C. Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, Guy M. Lohman (Ed.). ACM, 5:1–5:16. https://doi.org/10.1145/2523616.2523633

[29] Velox. 2023. Velox - Github Repository. Retrieved March 1, 2023 from https://github.com/facebookincubator/velox

[30] David Wilhite. 2022. GoogleSQL: A SQL Language as a Component. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zait (Eds.). https://cdmsworkshop.github.io/2022/Slides/Fri_C2.5_DavidWilhite.pptx