Composable Data Management: An Execution Overview

Pedro Pedreira
Meta Platforms Inc.
Menlo Park, CA
pedroerp@meta.com

Deepak Majeti
IBM
Pittsburgh, PA
deepak.majeti@ibm.com

Orri Erling
Meta Platforms Inc.
Menlo Park, CA
oerling@meta.com

ABSTRACT

The trend of decomposing monolithic data management systems into a stack of reusable components has quickly gained momentum across the industry. Although a series of open-source projects have emerged targeting different layers of the stack, execution engines are of special importance due to the complexity they encapsulate, and the demand to optimize price-performance. In this tutorial, we will survey the space of composability in data management, focusing on the execution layer. We will discuss the main APIs, integration with existing and novel data management systems, and how specialized behavior can be accommodated by using extensibility APIs. With an emphasis on analytics, we will take a deeper dive into performance, discussing modern aspects of vectorization, compressed (encoding-aware) execution, and adaptivity. While the presentation is contextualized using real-world examples and experience while developing the Velox open-source execution engine and integrations with existing systems like Presto (Prestissimo) and Spark (Gluten), the concepts and techniques discussed are generally applicable to other execution engines. Finally, we will discuss future trends and ongoing work regarding novel file formats, compressed execution opportunities, and nascent hardware acceleration efforts, highlighting current challenges and open questions. With a survey of the state-of-the-art in this space, we hope this tutorial will help motivate individuals and organizations to embrace composability and promote collaborations across related projects.

PVLDB Reference Format:
Pedro Pedreira, Deepak Majeti, and Orri Erling. Composable Data Management: An Execution Overview. PVLDB, 17(12): 4249 - 4252, 2024.
doi:10.14778/3685800.3685847

1 INTRODUCTION

Data management system design has evolved over the last decade. The traditional monolithic model of developing vertically integrated systems has resulted in a fragmented landscape, causing inefficiencies such as limited reusability across silos, inconsistent APIs and SQL dialects, and slowed down innovation. A new composable paradigm largely driven by open-source software has emerged [17] where data management systems are decomposed into a modular stack of reusable components with clearly defined responsibilities and based on open standards and APIs. This new architecture is quickly gaining momentum in the industry as it streamlines development, reduces maintenance costs, and ultimately provides a more consistent user experience.

Execution engines are a key component of this architecture. These libraries are responsible for executing fully optimized query plan fragments (or IRs) locally, from a table scan or consumer side of a shuffle, into a table write or producer side of a shuffle. Execution engines encapsulate a high amount of complexity as they contain the implementation of relational operators, efficiently handle I/O, and are the most resource-intensive component of the stack. As execution engines become composable, component boundaries need to be carefully drawn such that each component is independent. The API also plays an important role in performance and productivity; they need to provide configuration options and *extensibility* support to allow developers to extend data types, scalar, aggregate, and window functions, operators, file formats, serialization protocols, connectors, filesystems, and more, enabling its usage in environments that may require specialized behavior [16].

Vectorization [7] is today the main technique used to process large amounts of data. Vectorized execution engines decompose query plan fragments as a sequence of simple and concise operations over batches of data (*tight loops),* which can be efficiently executed by modern CPUs as they provide more predictable memory patterns and minimize CPU stalls. Considering vectorization relies on columnar layout, a myriad of data processing techniques that leverage columnar encodings such as dictionaries, RLEs, and constants are enabled. Today, modern *compressed* or *encoding-aware* execution engines may leverage the input encodings of data batches to choose more efficient kernels (tight loops), and may also generate data that is arbitrarily encoded [9].

Although vectorization and columnar execution have been studied for decades and presented as tutorials in the past [1], recent trends have fundamentally changed requirements and driven the emergence of new techniques. First, composability dictates that modern execution engines should be developed in a more reusable manner, based on open standards and providing adequate extensibility APIs. Second, with compressed execution and more flexible file formats and storage APIs, execution engines can more aggressively leverage the encodings of the input data as read from storage while executing query plans; they may also leverage the generated encodings while writing the data back to storage, or shuffling data across query stages. Third, modern execution engines commonly also rely on *adaptivity*, which is a set of techniques that allow engines to adapt and choose more efficient loops based on stats from processing previous batches from the same query. Balancing configuration options and adaptivity is key for user adoption, as systems with too many configuration options are hard for users to use. Fourth, the composable architecture provides a compelling

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685847

framework where hardware accelerators can more easily be integrated into data management, but also open important research avenues regarding how these can be enabled in a general manner.

Tutorial overview. This tutorial presents a survey of these recent trends, based on the authors' experience creating and developing open-source projects in this space (notably, the Velox open-source execution engine library [15]), and large organizational programs related to composability [8]. Although some of the discussion will be contextualized using real-world examples from Velox and its integrations with systems like Spark [21] and Presto [22], the techniques, concepts, and trade-offs presented are fundamentally equivalent and generally applicable to any other execution engines. During the presentation, related projects such as Apache Datafusion and Comet, in addition to Databricks' Photon will also be discussed.

The tutorial has a 90 min duration, and will be organized into three sections of 30 mins each: (a) we will first present a survey of composability and composable execution, discussing recent industry trends; (b) we will present the performance aspects of modern vectorization and compressed execution; (c) finally, we will discuss ongoing work, challenges, and future directions on this space.

Tutorial audience. This tutorial is intended for engineers and developers of data processing systems, and research groups who want to prototype and evaluate new execution techniques. The audience will learn about composing and reusing various components of a data processing engine, state-of-the-art vectorization techniques for data processing, and skills to improve existing data processing systems by leveraging open-source libraries.

2 TUTORIAL OUTLINE

This tutorial is organized into three main sections. In the first part (30 mins), we present a survey of composability in data management, discussing how it started, the industry trends that led to this inflection point, and the technological and economic benefits of this paradigm. We also present the reference composable data stack, its layers, APIs, responsibilities, principal projects, and open-source technologies on each layer. We focus on the *execution engine* component, highlighting progress and challenges in making them composable and discussing their common extensibility APIs.

In the second part (30 mins), we take a deeper dive into performance. With an emphasis on analytical workloads, we start by discussing the state-of-the-art in columnar layout, vectorized processing, compressed (encoding-aware) execution, I/O, and adaptivity. We take a closer look into the main relational operators - the ones that commonly consume the most resources in production workloads -, discussing in detail how table scans, filter/projection, hash joins, aggregates, shuffles, and table writes work in many real-life systems.

In the third part (30 mins), we discuss ongoing areas of research and future directions. We discuss existing research in improving file formats and data encoding by different groups in the academia, and the additional opportunities this presents for compressed execution and encoding-preserving operators. Lastly, we discuss how composability is paving the way for the adoption of hardware accelerators in data management, and discuss ongoing research avenues and open questions.

2.1 Composability

In the last decade, there has been an inflection on how data management systems are designed [17] [10]. As database vendors increased focus on the delivery of services and price-performance rather than proprietary software, open-source big data technologies and open standards have emerged and become commonplace. The Apache Hadoop project [11] started the trend by disaggregating storage and compute. With projects like Apache Parquet, Iceberg, Ibis, Substrait, and Velox, the data stack has evolved from a vertically integrated monolith into a composable stack of reusable components. A *composable data stack* promotes reusability across systems, reduces maintenance costs, provides a more consistent user experience across engines, and ultimately favors innovation. Recently, not only have new data management systems been created in record time by assembling spare parts and reusing existing components [14] [10], but decade-old battle-tested systems have also gone through major overhauls to make their architecture more composable [5] [8] [16].

The composable data stack is composed of:

(1) A **language frontend** responsible for interpreting user input (such as a SQL string or a dataframe-based API) into an internal format.
(2) An **intermediate representation** (IR) of the query - usually a logical or a physical plan.
(3) A **query optimizer** responsible for taking the query intermediate representation and generating a more efficient representation ready for execution.
(4) An **execution engine** able to locally execute IR fragments.
(5) An **execution runtime** that provides the distributed environment in which query fragments can be executed.

Execution engines. Distributed computations are often decomposed into *query fragments* (or stages) that either start on a table scan or the consumer side of a data shuffle and end on a table write or the producer side of a data shuffle. Execution engines are libraries able to take query fragments and execute them by leveraging the local resources of a host. They provide efficient implementations of relational operators such as table scans, projections, filters, joins, aggregates, shuffles, and table writes.

Extensibility. To be reusable across data management systems which may provide specialized behaviors and/or different SQL semantics, composable execution engines need to be *extensible*. A series of extensibility APIs are commonly exposed to allow developers to specialize the engine behavior, e.g, adding custom scalar, aggregate, and window functions, operators, file formats for storage, network serialization protocols, connector, and filesystem/blob storage APIs [23].

2.2 Performance

There are two predominant paradigms for the efficient processing of relational operators. Execution engines can either remove the cost of query interpretation by generating executable code at runtime (also known as *just-in-time* compilation) [13], or engines can amortize the interpretation overhead by processing batches of data at a time (also known as *vectorization*). Although today most large-scale execution engines are based on vectorization, the suitability of each strategy given a target workload is still a debated topic [12].

The main idea behind a vectorized engine is to decompose larger computations as a sequence of simple and concise operations over a batch of data - *tight loops* - which often provide more predictable memory patterns and minimize CPU stalls caused by cache misses and branch mispredictions. Vectorization enables CPUs to fully leverage out of order execution and SIMD instructions [18], while providing a programming paradigm that is more naturally translatable to highly parallel hardware accelerators like GPUs.

Vectorization relies on columnar layout [4], opening opportunities for more compact representations of the data since values of the same data types are contiguously stored. Although a myriad of encoding and compression techniques have been developed over the last decades [3] [6], cascading (or recursive) encoding techniques such as *dictionaries*, *run-length*, and *constant* have received particular attention due to their *dual* applicability for both compression and processing efficiency. For example, dictionaries can be used not only to compact the data, but also to represent the output of cardinality reducing or increasing operations such as filters, joins, and unnests, in many cases without having to modify the input columnar buffers underneath.

These and other techniques are usually referred to as *compressed* or *encoding-aware execution* [16] [19]. Encoding-aware engines can:

(1) Leverage the *input encodings* of the data for efficient execution. For example, expressions may be evaluated only on distinct values of dictionaries or over a constant.
(2) Use encodings to efficiently represent the *output of operations*. For example, filters may be executed by simply wrapping a dictionary to the input that only selects the rows that survived the filter. Joins, unnests and other cardinality-increasing or reducing operations can be executed using a similar technique.

Despite opening opportunities for numerous optimization techniques, this architecture adds an extra layer of complexity since many versions of the same loop may need to be implemented to handle each type of encoding. It makes the engine's codebase more complicated, harder to write and reason about, and more error-prone [20]. If taken to the extreme, these techniques may also bloat the engine code size and exacerbate compilation times. In the tutorial, we discuss the optimizations and techniques that were found to provide the best benefit and justify the added complexity [20].

Adaptivity. Vectorized engines may also be able to *learn* when applying computations over successive batches of data, in order to more efficiently process incoming batches. For example, engines often keep track of hit rates of filters and conjuncts to optimize their order; engines may also keep track of join key cardinality to more efficiently organize the join execution; or learn about which columns are in fact used at later stages of the operator tree to improve prefetching logic. We discuss some of the main areas where adaptivity is used in modern engines and present ongoing areas of research.

Memory Management. The vectorized engine must efficiently allocate and reserve the available memory among various operators. If an operator requests more memory, then the memory manager must dynamically arbiter the requested amount among other remaining operators. If no memory is available, the operator must spill to disk.

Input/Output. The performance of the vectorized engine also depends on the read/write throughput latency from the file system where the data is stored, and data caching. The data cache can be in memory or on disk/SSD. The in-memory data cache must work with the memory manager for optimal memory utilization. The engine must also be able to coalesce the reads to maximize throughput, and prefetch data to minimize latency. This can further be extended to track the read patterns and adaptively prefetch data streams.

2.3 Ongoing Research and Opportunities

Recently, there are three main areas that have received considerable attention as requirements for data management evolve and center around AI workloads.

File formats. Existing file formats used in most data management systems, such as Apache ORC and Parquet, were developed more than a decade ago when use cases and requirements were substantially different. For instance, the rapid growth of machine learning and proliferation of training tables containing at times tens of thousands of feature streams are inadequately supported in existing formats. Existing formats are also tightly coupled with data encodings, being unable to accommodate recent advances in data encodings and compression or features such as cascading encodings [2]. Lastly, the layout of existing file formats did not take into account that file decoding would eventually need to be efficiently supported in accelerators such as GPUs. The current layouts present major bottlenecks and hinder the adoption of hardware accelerators in data management. In this tutorial, we present the existing research around the topic, highlighting recent advances to the state-of-the-art, and discuss challenges and opportunities.

Encoding preservation. ETL pipelines for offline batch processing are commonly the largest consumer of compute resources in large data lakehouses. The common pattern for ETL queries is to read one or a few input tables, join them, apply transformations, and generate an output table. As file formats evolve and present better encoding ratios, the overhead of decoding the data during this process and re-encoding it before writing to the output table becomes more evident. An open avenue of research is understanding to which extent compressed execution techniques may be applied along with lazy materialization to allow encoded data to traverse the operator tree (potentially through shuffle boundaries and network serialization) to the table writer, without ever getting decoded. In the tutorial, we discuss compelling use cases, the progress made so far, and open other questions around this topic.

Hardware acceleration. The exponential growth of AI workloads has driven an accelerator-first inflection in the data center space. As AI emerges as the main consumer of data management and generation-over-generation gains in accelerators and networks are outpacing CPUs, data management and AI architectures are brought closer together. In the meantime, data management fragmentation has historically hindered the adoption of hardware accelerators, as it is too costly to integrate accelerators into existing systems.

Composable data management systems present a compelling architecture to hardware vendors, as hardware integration could be done in a single composable execution library, and be reused by any data management system integrated with this library. In this tutorial, we discuss the industry landscape of Velox and accelerator integrations, initial results using GPU and FPGA integration in a composable manner, and a sketch of the software framework. We discuss how vectorized execution can be offloaded to accelerators in a general manner, supporting not only multiple data systems but also different types of accelerators. To that end, we generalize the traditional query processing model to extract all latent parallelism and asynchronicity. We then present a new framework under development called Velox Wave, which enables GPU offload of arbitrary Velox query plans. The core open question is under what circumstances does it pay off to pivot to accelerators and what changes in engineering thinking this entails.

Finally, we will highlight the challenges and opportunities in this space and hope to motivate other individuals and organizations to embrace composability as the future of data management architecture.

3 PRESENTERS

Pedro Pedreira is a Software Engineer at Meta. During his 11-year tenure, he has led a variety of projects related to Compute in Meta's Data Infrastructure, including systems like Cubrick, Scuba, and more recently, the Velox open-source execution engine. Pedro has been one of the main proponents behind the composable data system movement and leads many of the collaborations across Meta and the open-source community in this space. Pedro holds a Ph.D, M.Sc, and B.Sc degrees in computer science, and his research interests include query execution, composability, and data system architecture.

Deepak Majeti is a Principal Software Engineer at IBM. He is working towards making Presto a turnkey high-performance analytical engine. Deepak is a Velox maintainer, a PMC member for the Apache ORC project, and a committer for the Apache Arrow project. Deepak is passionate about thinning the line between Big Data and High-Performance computing. Deepak graduated with a Ph.D from Rice University, USA, and an M.Tech from IIT-Kanpur, India in the field of computer science.

Orri Erling co-founded the Velox composable query execution project at Meta. Prior to this, he worked on Google's F1 and before then created OpenLink Virtuoso, a relational/graph store best known for its applications in linked data and knowledge graphs. His research interests include benchmarking and generalizing query processing to fuse with neighboring graph, AI and HPC domains. Orri's mission is to create a line of components from execution to query optimization to distributed computing.

REFERENCES

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1664–1665.

[2] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (may 2023), 2132–2144.

[3] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4, Article 230 (dec 2023), 26 pages.

[4] Apache Arrow. [n.d.]. A cross-language development platform for in-memory analytics. https://arrow.apache.org/. Accessed: 2024-03-21.

[5] 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 *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339.

[6] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2649–2661.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR)*.

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

[9] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query optimization in compressed database systems. *SIGMOD Rec.* 30, 2 (may 2001), 271–282.

[10] Voltron Data. [n.d.]. Theseus, The GPU query engine for petabyte SQL. https://voltrondata.com/theseus.html. Accessed: 2024-03-21.

[11] Apache Hadoop. [n.d.]. Apache Hadoop. https://hadoop.apache.org/docs/stable/. Accessed: 2024-03-21.

[12] 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.

[13] Thomas Neumann. 2021. Evolution of a compiling query engine. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3207–3210.

[14] 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*.

[15] Pedro Pedreira, Masha Basmanova, and Orri Erling. [n.d.]. Introducing Velox: An open source unified execution engine. https://engineering.fb.com/2023/03/09/open-source/velox-open-source-execution-engine/. Accessed: 2024-03-21.

[16] 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.

[17] 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 (jun 2023), 2679–2685.

[18] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508.

[19] 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.

[20] Laith Sakka, Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Wei He, Xiaoxuan Meng, Krishna Pai, and Bikramjeet Vig. 2024. Simple (yet Efficient) Function Authoring for Vectorized Engines. *Proc. VLDB Endow.* 17, 12 (jul 2024).

[21] Apache Spark. [n.d.]. Apache Spark - Unified Engine for large-scale data analytics. https://spark.apache.org/. Accessed: 2024-03-21.

[22] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2, Article 189 (jun 2023), 25 pages.

[23] Velox. [n.d.]. Velox Documentation. https://facebookincubator.github.io/velox/. Accessed: 2024-03-21.