Shared Foundations: Modernizing Meta's Data Lakehouse

Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian "James" Sun,
Suketu Vakharia, Peng Li, Weiran Liu, Sundaram Narayanan
{biswapesh,pedroerp,sag,jamessun,suketukv,plifb,weiranliu,sunnar}@fb.com
Meta Platforms Inc.
Menlo Park, CA, USA

ABSTRACT

Data processing systems have evolved significantly over the last decade, driven by large trends in hardware and software, the exponential growth of data, and new and changing use cases. At Meta (and elsewhere), the various data systems composing the data lakehouse had historically evolved organically and independently, leading to data stack fragmentation, and resulting in work duplication, subpar system performance, and inconsistent user experience. This paper describes how we transformed the legacy data lakehouse stack at Meta to adapt to the new realities through a large cross-organizational effort called Shared Foundations. This program promotes a compositional approach based on the principles of reusable components, deduplicated systems, and common and consistent APIs. The Shared Foundations effort has resulted in a more modern data architecture at Meta – one that offers better performance, richer features, higher engineering velocity, and a more consistent user experience, setting up the data lakehouse stack at Meta for faster innovation in the future.

1 INTRODUCTION

The requirements for large-scale data lakehouses [25] have evolved in the last decade. Apart from the exponential data growth fueled by new products, increasingly complex requirements, and sheer organic growth, the demand for fresher data and faster queries, essential to lowering the time to insight, has also increased. Data models have become more complicated, resulting in most tables containing complex data types such as structs, maps, and arrays. In parallel, query complexity has also grown; it is increasingly common to find queries with a large number of stages, iterative graph queries, time series analysis, and complex business logic, especially in data pipelines.

Other trends in hardware and software have also impacted the architecture of modern data lakehouses. The death of Moore's law for CPUs has raised the importance of horizontal scalability and elastic computing, making resource fungibility across compute engines a prerequisite. Because the power to drive these CPUs is scarce, power efficiency became fundamental and led to numerous efforts such as native code optimizations leveraging SIMD instructions, GPUs, and other special purpose hardware co-optimized with software. With faster networks and larger storage units, storage disaggregation became commonplace and the architecture of choice of modern data warehouses [4] [16]. Lastly, the recent emergence of machine learning workloads has developed a new set of trends in terms of data volume, complexity, and unusual access patterns [26].

Meanwhile, Meta's data stack had only evolved incrementally over the last decade. This has resulted in a fragmented stack which was difficult to maintain and evolve, composed of almost a dozen SQL dialects, multiple engines targeting similar workloads (each with their own quirks), and numerous copies of the same data in different locations and formats. The lack of standardization and reusable components not only increased the operational burden on engineering teams, but ultimately slowed down innovation. It also impacted our users, who had to interact with engines exposing different SQL dialects, inconsistent semantics, and presenting suboptimal performance.

To address these challenges, we started a cross-organizational effort involving dozens of engineering teams called Shared Foundations, promoting a compositional approach based on the principles of reusable components, consolidated engines, common APIs, and consistent standards. By consolidating compute engines, converging storage libraries and metadata, and unifying SQL dialects and execution engines, the Shared Foundations effort has created a more modern data architecture; one that offers better performance, richer features, higher engineering velocity and a more consistent user experience. Ultimately, it made Meta's data lakehouse stack more adaptable to current and future trends, promoting faster innovation.

In this paper, we make the following contributions:

- We describe how new usage patterns and trends in hardware and software throughout the last decade have driven organic changes to our exabyte-scale data lakehouse, resulting in a data stack that was difficult to enhance and maintain.
- We present the historical context and main design principles behind one of the largest data lakehouses in the world and characterize the systems comprising it.
- We motivate the importance of a horizontal large-scale effort across different engines, focusing on reusability, unification, and consistency, hoping to further motivate the need for research on composability of data management systems.
- We detail the Shared Foundations program and the different efforts under its umbrella. We also highlight how real-world hyperscalar compute engines, storage libraries, SQL dialects, and execution engines were consolidated and made consistent to users, which in turn made our engineering organization more efficient and effective.

2 BACKGROUND

Meta's data stack has been incrementally built over the last two decades. Starting with Hive, a system created and open sourced

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2023. 12th Annual Conference on Innovative Data Systems Research (CIDR '23). January 9-12, 2023, Amsterdam, Netherlands.

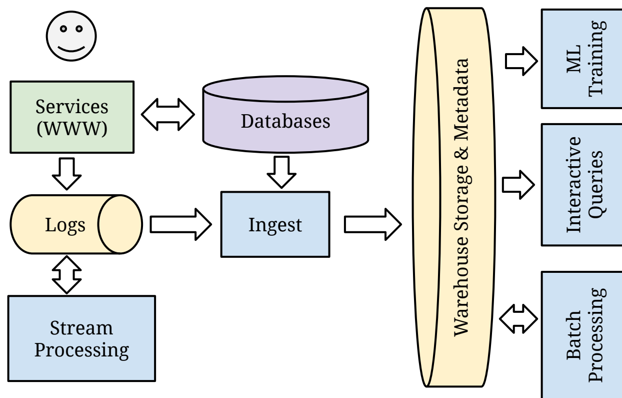

Figure 1: High-level data flow: Meta's data lakehouse stack.

by Meta in 2010, the data warehouse quickly grew from tens to hundreds of petabytes [14], and more recently to multiple exabytes. The high level schematic of the stack is shown in Figure 1. Hive was built on several principles which are still mostly relevant today:

- Data, metadata and compute are disaggregated, allowing each to scale independently.
- Data is stored on a distributed file system, HDFS, which allows data to scale horizontally independent of compute, and provides increased resilience. Meta has in recent years replaced HDFS with its own file system, Tectonic [19], which provides several enhancements such as greater scalability, better isolation, smarter encodings and better support for SSDs.
- Metadata is stored in MySQL. Hive Metastore provides partitioning mechanisms to optimize data organization. Over time, Meta's metadata size has grown exponentially, requiring us to add support for sharded MySQL for storage. Traditionally, sharding is done on a per namespace basis.
- Data format is columnar. The original columnar format, RCFile, was enhanced to create ORC [11]. ORC has several advanced features like multiple encodings, NULL support, flexible compression levels and support for complex types such as arrays and maps. Meta uses an internal variant of ORC, named DWRF [10], which has additional features such as better support for large maps and encryption.
- Multiple compute engines can run on the same data, since compute and storage are disaggregated. This has allowed us to evolve the Compute space independently, and many engines can run on the Hive Warehouse, such as Spark [7] (which replaced the original Hive engine), Presto [23] and DiGraph (Meta's deployment of Giraph [8]).

However, limitations of the Hive architecture led to the development of many other systems over time. Some of these limitation were:

- **No support for stream processing.** This resulted in various streaming systems being built over the years, notably Puma [5]. These systems typically were not well integrated with Hive.
- **No support for real time data ingestion into Hive.** This resulted in systems like Scuba [1], which was originally built for log analytics at scale and hence lacks support for accurate results or complex queries, to be (ab)used for analytics. Scuba is written in C++, has its own SQL dialect, and uses its own file format and local storage instead of Hive.
- **Programming language divergence.** Most of the DI stack (Hive, the engines running on it, Puma, etc) was written in Java; however, the majority of Meta uses C++, e.g, Machine Learning systems, Tectonic, RocksDB, and many others. Java is also not a primarily supported language inside Meta [13]. This resulted in new engines such as Scuba and Cubrick [20] being required to rebuild many core components in C++, such as the execution primitives (e.g, functions and operators), codecs for reading and writing data formats such as ORC, and even developing their own file formats. The lack of reusable components and programing language convergence resulted in duplicate implementations, and increased maintenance burden.
- **Poor query latency.** The Hive engine was too slow for interactive analytics. This resulted in multiple new engines such as Presto, Scuba, Raptor and Cubrick trying to address the interactive analytics space. Some were written in Java and some in C++, resulting in fragmentation. Further, these engines, even when written in the same language (e.g. Scuba and Cubrick, or Presto and Spark), did not share any code or components, because of various historical reasons. As a particularly egregious example, Spark and Presto both read and wrote the same ORC format stored on Hive, but used completely different libraries for interacting with file system and Metastore, encoding and decoding ORC files, and even decompressing the data! Similar patterns were also found in the streaming and batch processing worlds.
- **Inefficient I/O Usage.** Hive data is traditionally stored in hard disks in HDFS (later, Tectonic) storage nodes. Fetching data from HDD over the network is too slow for many interactive analytics scenarios. Thus, many interactive engines (Raptor, Cubrick, Scuba) were designed with co-located compute and storage, where data is required to be pre-loaded into local SSD or memory for querying. This resulted in further storage fragmentation and data duplication, and stranded resources.

In that landscape, the widespread fragmentation made the Meta data lakehouse stack difficult to maintain and evolve. There were half a dozen different SQL dialects, three implementations of Metastore client and ORC codecs, about a dozen different engines targeting similar workloads, and many copies of the same data in different locations and formats. The lack of standardization and reusable components not only increased the operational burden in the engineering teams, but ultimately slowed down innovation. It also impacted our users, who had to interact with engines exposing different SQL dialects, inconsistent semantics and suboptimal latency.

3 SHARED FOUNDATIONS

To address the challenges described above, we started a cross-organizational effort called Shared Foundations, with the purpose of re-architecting our data lakehouse stack. Shared Foundations is a

multi-year program involving dozens of engineering teams within and outside of the Data Infrastructure organization, and hundreds of engineers throughout Meta. It is based on the following core principles:

- **Fewer systems:** If different systems targeting the same set of use cases with overlapping functionality are available, they should be consolidated into a single system. For example, directionally, there should exist a single compute engine targeted to each of the following spaces: interactive, batch, streaming and machine learning. Each of these systems should be best-of-breed and provide a superset of the functionality available with the fragmented systems pre-consolidation. During this process, any residual data or duplicated pipelines should also be removed.
- **Shared components:** To avoid the one-size-fits-all issue, if use cases and requirements are indeed distinct (e.g, batch and interactive query processing), different compute engines could still be provided. For these cases, there is a strong focus on composability and on reusing as many components as possible across the different layers of the stack. For example, there is no reason for the storage encodings or formats to be different for interactive and batch engines.
- **Consistent APIs:** Users often have to interact with different engines to get their job done. If engines expose consistent APIs, it lowers the learning curve for users, making them more productive. Similarly, consistent APIs make integration of components easier, thus promoting modularization and reusability.

The advantages of these principles were three-fold:

- **Engineering efficiency:** More engineers could work on each of these (smaller number) of systems and components. This reduces duplication, prevents us from re-inventing the wheel, consolidates domain-specific knowledge in fewer specialized teams, and ultimately enables our engineering organization to be more efficient and move faster.
- **Faster innovation:** Having fewer systems to maintain reduces operational burden, and allows engineering teams to focus on new features, optimizations and other enhancements, favoring innovation.
- **Better user experience:** Users can now expect consistent syntax, semantics and features across these systems, reducing their learning curve and increasing productivity. Users can also reap the benefits from faster innovation, such as more features and better performance.

The Shared Foundations effort is organized in a few major convergence areas, namely: storage, formats, metadata, execution, language and engine. Figure 2 illustrates the mapping of the layers to the scope, challenges, desired end state and projects, which are detailed in the following subsections.

3.1 Compute Engine Convergence

The heavily fragmented compute engine ecosystem at Meta, resulting from decades of organic development, was one of the earlier challenges faced in our convergence effort. Multiple engines aimed at very similar spaces were available, such as Presto, Raptor, Cubrick and Scuba for interactive SQL querying; Presto and Spark for batch SQL execution; and Puma, Stylus, XStream, and MRT for stream processing. The subsections below detail the convergence efforts for interactive (3.1.1), batch (3.1.2) and streaming (3.1.3) engines.

Layer	Scope	Challenges	End state	Projects
Language	SQL dialects, functions, entity & type metadata	SQL dialect fragmentation, lack of expressibility	Single SQL dialect	CoreSQL, Sapphire, XStream
Distribution	Distributed execution, shuffle, resource management	Scalability, Efficiency, Fragmentation	Engine consolidation	RaptorX, Sapphire, XStream
Execution	Evaluation at node, caching	Latency, efficiency, Java / C++, dialect fragmentation	Unified execution library	Velox
Data Access	Formats, storage, disaggregation	Library fragmentation, not disaggregated, limited encodings	Disaggregated storage, unified optimized format	RaptorX, Alpha, Metastore-X, NRT

Figure 2: Stack layers, challenges, end state & projects.

3.1.1 *Interactive Engine Convergence.*

The ideal interactive engine would have the best-of-breed features from Presto, Raptor, Cubrick and Scuba. Essentially, the converged engine should (a) provide full SQL support, including complex queries and data models, (b) be able to operate directly on the lakehouse data without additional copies, (c) provide low query latency, driven by having the majority of the data in memory or local SSD, and (d) have support for real time data.

The convergence effort was based on Presto, as the system that provided most of the properties above, and the performance gap between Presto and other existing systems was bridged through affinity and local caching. By introducing smart hierarchical caching, and therefore keeping the most frequently used data and metadata in the local memory and SSD of the workers and the coordinator, we achieved an order of magnitude speedup for most common query patterns [17]. This speedup met or exceeded the performance of existing systems, on less hardware.

Migration Process: Many interactive analytic use cases are built around general purpose visualization and dashboarding tools. For these cases, which were targeted for migration first, merely adding the capability of generating SQL according to the target system was sufficient. For the long tail of remaining use cases (other custom built tools), the superior performance and set of capabilities were often sufficient to motivate users to migrate their queries and tools to Presto. Moreover, Data Infrastructure engineers also engaged with users to motivate and support the migration based on ROI: larger use cases that allowed a larger portion of the old system to be deprecated were targeted first.

When migrating queries to a new engine and dialect, not only syntactic incompatibilities were addressed, but also the different function packages (by mapping to functions available in Presto) and system capabilities. Fortunately, Presto has proven to be a more flexible engine and allowed all user queries to be mapped to a supported Presto query. Lastly, considering that the data previously being loaded into Raptor, Cubrick, and Scuba analytic tables was already generated in the lakehouse (and only then moved to these systems), data migration was not an issue; in fact, it allowed many ETL pipelines to be simplified and be made more reliable.

The entire migration effort took over two years. At the end of the migration, Raptor and Cubrick were completely deprecated, eliminating two large systems and several hundred thousand lines of code. Several analytic use cases running on Scuba were also migrated to Presto, allowing Scuba to focus on the intended log analytics and monitoring use cases. Beyond reducing the operational load of maintaining three additional systems, several thousand machines were also saved in the process.

Near Real Time Data: Another important aspect of storage, particularly for interactive analytics use cases, is the availability of near real time data. While real time ingestion is a common feature in most data lakehouses, such as BigQuery [16], Databricks [25] and Snowflake [6], the Meta data lakehouse had been restricted to batch data because of the lack of disaggregated storage in some interactive analytic engines, and necessary features in Metastore.

With Presto supporting interactive queries directly on Hive data, it became possible to expose the real time data for querying as it was imported into the lakehouse. FBETL (Meta's ingestion system), already supported continuous ingestion of log data. However, query engines did not have access to this data since partitions were only registered into Metastore once all the data for that partition was available, which tended to be based on hourly or daily boundaries.

To address this issue, an additional partition state was introduced in Metastore ("open"), and partitions are now registered as soon as data starts arriving. Presto is now able to query data immediately upon landing for ad-hoc analysis and dashboarding, though batch pipelines continue to rely on the "closed" partition signal to ensure they always run on complete datasets. This design allows the same datasets to be consumed by real-time queries as well as batch pipelines, avoiding cumbersome migrations. Finally, we also strengthened the ingestion commit protocol to ensure exactly-once data delivery for real-time data.

The near real time ingestion feature has already been enabled in numerous datasets in the lakehouse, providing data freshness in the order of a few minutes - from logging to being queryable.

Metadata Improvements: An important aspect of evolving metadata for Presto was about performance and improving cacheability of Metadata. Hive Metastore already leveraged memcache to reduce latency of MySQL accesses, but this metadata was not cacheable on the client side without relaxing consistency guarantees in the event of updates. We rebuilt the metadata layer and APIs to introduce the concept of *versions* for each piece of metadata that mutates in the event of any update. Presto, in its turn, introduced metadata caching leveraging this versioning information, vastly improving metadata latency for repeated queries.

3.1.2 Batch Engine Convergence. At Meta, a large number of offline analytics use-cases discretize (or batch) the data into static data sets, commonly using hourly, daily, or monthly intervals, and process them in a time-agnostic manner. These queries, typically referred to as batch queries, are often automatically scheduled once their data and upstream pipeline dependencies are ready (e.g., input partitions or parent jobs), and start execution as soon as idle resources are available in the shared resource pool. While users do not always expect a batch job to finish immediately (in contrast to interactive queries), there are implicit expectations around when the output tables should land. As a result, batch query processing engines often need to strike the right balance between throughput and latency.

Meta created the Hive engine for all batch processing in late 2000s. Hive's SQL dialect (HiveQL) was primarily motivated by extensibility; it supported user-defined functions (UDFs, UDAFs, and UDTFs) implemented in Java, and allowed users to embed custom map-reduce scripts in C++, Python, and PHP using a row-based streaming interface. These extensions made the HiveQL dialect very flexible and it quickly gained user adoption. However, as a result of organic evolution, these extensions were ad-hoc and not standards compliant.

Subsequently, when Presto was built, we decided to build a cleaner standards compliant SQL dialect (henceforth referred to as PrestoSQL). Farther down the road, when Hive was replaced by Spark SQL [3], we decided to keep the HiveQL-compliant dialect to accelerate the migration process. However, given Presto's higher performance, many batch pipelines were also migrated to Presto. But at the same time, Presto's streaming architecture proved to be insufficiently resilient to machine failures and most of the larger longer running pipelines gravitated towards Spark due to its superior resilience characteristics, including a far more scalable shuffle implementation [9]. Thus, Meta ended up in a situation where we had two engines with two different/incompatible dialects, and no one engine had the required set of capabilities to run all workloads. To further complicate matters, because Spark and Presto have different resource models, there was no machine fungibility, and we had to provision batch capacity separately for Presto and Spark.

We decided to solve this problem through the principles of componentization. Specifically, we decided to marry the scalability of the Spark engine with the standards compliant SQL dialect from Presto. The result was *Presto on Spark* [12]. Presto on Spark achieves this by refactoring the Presto front-end (parser, analyzer, optimizer, planner) and backend (evaluation and I/O) libraries and embedding these in the Spark driver and worker respectively. By running the exact same code on the front-end and back-end, Presto on Spark guarantees 100% compatibility with PrestoSQL, allowing the user to seamlessly move from interactive to ad-hoc to batch use cases without needing to rewrite their queries. At the same time, because it runs on the Spark RDD runtime and uses Meta's scalable Cosco shuffle infrastructure, the scalability and resilience benefits of Spark are leveraged (notably, fine-grained task retries), allowing Presto on Spark to run large-scale and long-running pipelines using the same resource pools as Spark. A surprising side benefit of Presto on Spark has been latency wins; as it turns out, many pipelines can actually run faster on it if compared to traditional Presto, mainly due to Presto on Spark's ability to assign an increased number of shuffle partitions to a given query.

Presto on Spark is currently in production and running thousands of pipelines every day. The current production version of Presto on Spark uses the Presto Java backend to quickly achieve 100% compatibility with Presto, but we have started work on replacing it with Velox, our unified C++ execution engine described in Section 3.4. Early results indicate large efficiency benefits. We are also building tooling to translate existing HQL queries to PrestoSQL to accelerate the migration process. Presto on Spark is fully open source and available as part of the PrestoDB repository.

3.1.3 Streaming Engine Convergence. Stream processing at Meta has also evolved organically due to a variety of reasons, resulting in a fragmented ecosystem [5]. The two main themes have been the programming language (C++ vs. Java vs. PHP) and abstraction level (low-level procedural vs. high-level SQL-like declarative API). The legacy data lakehouse stack was composed of multiple streaming engines: Puma (Java, declarative), Stylus (C++, low level), and others which had different combinations of abstraction levels (declarative, procedural) and implementation languages (C++, Java, PHP).

This fragmentation led to multiple challenges. For end users, they needed to deal with inconsistent language designs and implementation details, e.g. some solutions were running under at-least once semantics and some were at-most once. For developers, there was a significant burden in maintaining multiple stacks, and any new features like data management (e.g. lineage, schematization, and privacy enforcement) needed to be implemented multiple times. To get rid of the existing technical debt, and continue supporting emerging needs like real-time machine learning, we built the next generation of stream processing platform, called XStream [15].

While developing XStream, we made a number of design choices along the way. At the beginning, the design was to provide a single dataframe based declarative language, and underneath, use generated C++ code for expression evaluation and gluing predefined template code to perform common transformations like joining and window aggregation. While the dataframe language was highly expressive, it was a non-trivial learning curve to end users and SQL support was constantly asked. Compiling the generated code with C++ templates was time-consuming, and compilation errors were hard for users to understand.

We revised the design choices and decided to integrate with CoreSQL (described in subsection 3.2), add stream processing extensions, and promote SQL as the main language experience. On the other hand, we replaced code generation with Velox-based [21] interpretive execution, which not only allowed us to leverage a unified library for execution, but also provided performance benefits. XStream today supports a wide variety of use cases from SQL, machine learning, function as a service, and low-level system coordination. It allowed us to deprecate Puma and therefore another SQL dialect (PQL); further consolidation efforts are still ongoing.

3.2 SQL Dialect Convergence

One of the important decisions we had to make as we consolidated our data stack was around SQL dialects. We had half a dozen variants of SQL being actively used at Meta: Presto SQL, HiveQL (in Spark), PQL (Puma), Scuba SQL, Cubrick SQL and MySQL. This made for a very steep learning curve for our users, and wasted a lot of engineering effort while adding features to the various dialects. We decided to whittle it down to two, MySQL and Presto SQL:

- MySQL is the only dialect for OLTP applications and is ubiquitous at Meta, so moving off of it was impractical. However, it has limitations regarding its type system and extensibility, making it unsuitable for analytical applications.
- Among the analytical dialects, Presto's SQL dialect was a superior choice since it already had widespread adoption, a clean standard-compliant design, and support for complex types, rich features and extensibility. We adopted this for all analytical applications, extended it for streaming, graph and other use cases, and internally named it CoreSQL.

However, the difficulty then was around how to actually achieve compatibility across the different engines. We looked at the industry - Google had achieved this with ZetaSQL and this provided a good framework. At a high level, we needed two components:

1. A SQL parser and analyzer (the front-end), responsible for parsing and analyzing queries, in addition to creating and validating query plans. For this, we already had a Java implementation (Presto), and a Python implementation (used by developer tools). We decided to rewrite the Python implementation in C++ for better performance and better integration with the C++ engines. Work is underway to further simplify this into a single C++ library with Java bindings.
2. A library of functions and operators (the backend) that provided a canonical implementation of the language. Again, we already had the Java implementation from Presto, which we decided to reuse to get us off the ground, but we also started an ambitious effort to rewrite the execution engines as a library in C++ from the ground up (described in subsection 3.4), for maximum performance and portability across engines.

With both front-end and backend available as libraries, it became easier for engines to adopt CoreSQL as the standard dialect across engines. Thus, as we executed on the engine consolidation strategy, the engines we consolidated on all converged on CoreSQL as the only supported SQL dialect. Specifically, the new converged engines for interactive analytics, batch (Presto on Spark) and Streaming (XStream) all support CoreSQL as the only SQL dialect.

3.3 Storage Convergence

3.3.1 Converging codecs for ORC. Meta has traditionally used ORC as the columnar format for the lakehouse. The internal ORC variant, named DWRF [10], has additional features such as better support for large maps, and finer grained encryption. Though the new DWRF fork allowed us to evolve the format faster, the organic growth of compute engines within Meta using this format resulted in a fragmented space for codec libraries. Two Java implementations of these codecs existed, one for Spark and DiGraph, and one for Presto, and one in C++ named DWIO, primarily used by ML applications. Although these three libraries followed the DWRF standard, they each had their own limitations.

While we wanted to move to a pure C++ execution mode for data in the long run, we still needed to support Java for a few years. We therefore executed on a two-pronged strategy. First, we converged the Java codecs into one. We chose the Presto codec as the base because of its higher performance and the fact that it was already open sourced, then gradually incorporated any missing features into it. We then switched over Spark, DiGraph and other systems to use the new codec. Second, we refactored the DWIO library into Velox, and added all the features and optimizations that had gone into Java. This is now available in the open source as part of Velox.

3.3.2 A new ML optimized file format. While storing analytical and ML tables together in the data lakehouse is beneficial from a management and integration perspective (e.g. ML tables can be

analyzed and processed using standard analytical query engines such as Presto and Spark), it also imposes some unique challenges. For example, it is increasingly common for ML tables to outgrow analytical tables by up to an order of magnitude. ML tables are also typically much wider, and tend to have tens of thousands of features usually stored as large maps.

As we executed on our codec convergence strategy for ORC, it gradually exposed significant weaknesses in the ORC format itself, especially for ML use cases. The most pressing issue with the DWRF format was metadata overhead; our ML use cases needed a very large number of features (typically stored as giant maps), and the DWRF map format, albeit optimized, had too much metadata overhead. Apart from this, DWRF had several other limitations related to encodings and stripe structure, which were very difficult to fix in a backward-compatible way. Therefore, we decided to build a new columnar file format that addresses the needs of the next generation data stack; specifically, one that is targeted from the onset towards ML use cases, but without sacrificing any of the analytical needs.

The result was a new format we call *Alpha*. Alpha has several notable characteristics that make it particularly suitable for mixed Analytical nd ML training use cases. It has a custom serialization format for metadata that is significantly faster to decode, especially for very wide tables and deep maps, in addition to more modern compression algorithms. It also provides a richer set of encodings and an adaptive encoding algorithm that can smartly pick the best encoding based on historical data patterns, through an encoding history loopback database. Alpha requires fewer streams per column for many common data types, making read coalescing much easier and saving I/Os, especially for HDDs. Alpha was written in modern C++ from scratch in a way that allows it to be extended easily in the future.

Alpha is being deployed in production today for several important ML training applications and showing 2-3x better performance than ORC on decoding, with comparable encoding performance and file size.

3.4 Execution Engine Convergence

For any engine, the largest, most complex and the most performance sensitive part is usually the execution engine library that implements all data-intensive operations. The organic evolution of Meta's data lakehouse had created a fragmented ecosystem for execution engines. It resulted in dozens of specialized implementations that shared little to nothing with each other, were written in different programming languages, were maintained by different engineering teams, and largely provided inconsistent semantics to users. For example, an informal internal survey 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.

In order to address these challenges, we created Velox [21], a novel state-of-art C++ database acceleration library that provides high-performance data processing components with the purpose of unifying execution engines across different compute engines. In the common usage scenario, Velox takes fully optimized query plans as input, and performs the described computation using the resources available in the local host. Velox democratizes optimizations that were previously found only in individual engines, providing a framework in which consistent semantics across engines can be implemented. This reduces work duplication, promotes reusability, and improves overall efficiency and consistency.

Velox is under active development, but it is already in various stages of integration with more than a dozen data systems at Meta, including Presto, Presto on Spark, XStream, FBETL (our system for data ingestion into the Warehouse), Scribe, as well other internal ML systems for feature engineering and data preprocessing, and even transactional systems. In addition to consistent semantics and reusability benefits, Velox provides an implementation of the CoreSQL dialect, and presents performance improvements of up to an order of magnitude [21]. Although only recently open sourced [22], Velox is already backed by a fast-growing community composed of hundreds of companies and individual contributors.

4 FUTURE WORK

While many of the efforts in the sections above have landed in production, significant work still remains to fully reap the benefits of the new modernized architecture based on shared foundations. Many of the initiatives have been fully completed (such as interactive engine and ORC codec consolidation); others have landed in production but need more work to reach full completion (e.g. Velox, Presto on Spark, Alpha and XStream). We are also discussing the feasibility of open sourcing more components of this architecture.

Furthermore, we continue to explore new areas where we can apply the principles of Shared Foundations. One active area of investigation is on providing unified support for user-defined functions, since UDF support APIs across engines are vastly incompatible and provide inconsistent user experience. Extending the language consolidation effort described for SQL, we are also exploring if other non-sql APIs (dataframe interfaces and other DSLs, like TorchArrow [18], Spark Dataset API and Pandas) could be unified, as they become commonplace with the popularization of ML and data science.

We believe componentization and composability to be the future of data management. Further down the stack, we have started researching whether query optimizers could be consolidated. While the state-of-the-art suggests that optimizers are tightly intertwined with an engine's runtime and physical capabilities, we believe they could (at least) share an underlying framework. Projects like Apache Calcite [2] and Orca [24] give us precedent, but the extent in which these components can be consolidated is still an open question. Lastly, as Velox becomes the standard for execution within and outside of Meta, we are exploring how Velox could be enhanced to take advantage of hardware accelerators, allowing us to adapt all of our engines at once as hardware evolves.

5 CONCLUSIONS

Over the last three years, we have implemented a generational leap in the data infrastructure landscape at Meta through the Shared Foundations effort. The result has been a more modern, composable and consistent stack, with fewer components, richer features, consistent interfaces, and better performance for the users of our stack,

particularly, machine learning and analytics. We have deprecated several large systems and removed hundreds of thousands of lines of code, improving engineering velocity and decreasing operational burden. We have open sourced several of the major components of the new architecture, such as Velox, DWIO, Presto on Spark, Presto with hierarchical caching, and TorchArrow, and are working closely with the open source community to donate several enhancements to existing open source projects, such as Arrow and ORC. This journey is 1% finished.

REFERENCES

[1] L. Abraham et al. Scuba: Diving into Data at Facebook. *Proc. of the International Conference on Very Large Data Bases.*, 6(11):1057–1067, aug 2013.

[2] Apache Software Foundation. Apache Calcite - The foundation for your next high-performance database. https://calcite.apache.org/.

[3] M. Armbrust et al. Spark SQL: Relational Data Processing in Spark. page 1383–1394, 2015.

[4] M. Armbrust et al. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. of the International Conference on Very Large Data Bases.*, 13(12):3411–3424, aug 2020.

[5] G. Chen et al. Realtime Data Processing at Facebook. In *SIGMOD 2016*, 2016.

[6] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Databricks Inc. Apache Spark. https://spark.apache.org/.

[8] Facebook Inc. Apache Giraph. https://giraph.apache.org/.

[9] Facebook Inc. Cosco: An efficient facebook-scale shuffle service. https://www.databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service.

[10] Facebook Inc. The DWRF file format. https://github.com/facebookarchive/hive-dwrf.

[11] Facebook Inc. The ORC file format. https://orc.apache.org/.

[12] Facebook Inc. Presto on apache spark: A tale of two computation engines. https://www.databricks.com/session_na20/presto-on-apache-spark-a-tale-of-two-computation-engines.

[13] Facebook Inc. Programming languages endorsed for server-side use at Meta. https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/.

[14] Facebook Inc. Scaling the Facebook data warehouse to 300 PB. https://engineering.fb.com/2014/04/10/core-data/scaling-the-facebook-data-warehouse-to-300-pb/.

[15] Facebook Inc. Xstream: Stream processing platform at facebook. https://www.slideshare.net/aniketmokashi/xstream-stream-processing-platform-at-facebook.

[16] Google Inc. BigQuery explained: An overview of BigQuery's architecture. https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview.

[17] Meta Inc. Raptorx: Building a 10x faster presto. https://prestodb.io/blog/2021/02/04/raptorx.

[18] Meta Platforms Inc. TorchArrow: a data processing library for PyTorch. https://github.com/pytorch/torcharrow.

[19] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, P. ShivaShankar, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. R. Tipton, E. Katz-Bassett, and W. Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *FAST*, 2021.

[20] P. Pedreira, C. Croswhite, and L. Bona. Cubrick: Indexing Millions of Records per Second for Interactive Analytics. *Proc. of the International Conference on Very Large Data Bases.*, 9(13):1305–1316, sep 2016.

[21] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay. Velox: Meta's unified execution engine. *Proc. of the International Conference on Very Large Data Bases.*, aug 2022.

[22] Pedro Pedreira, Masha Basmanova, Orri Erling. Introducing Velox: An Open Source Unified Execution Engine. https://engineering.fb.com/2022/08/31/open-source/velox/.

[23] R. Sethi et al. Presto: SQL on Everything. In *ICDE 2019*, pages 1802–1813, 2019.

[24] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A modular query optimizer architecture for big data. SIGMOD '14, page 337–348, New York, NY, USA, 2014. Association for Computing Machinery.

[25] M. Zaharia et al. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR 2021*. www.cidrdb.org, 2021.

[26] M. Zhao et al. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *ISCA '22*. ACM, 2022.