Velox: Meta's Unified Execution Engine

Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka,
Krishna Pai, Wei He, Biswapesh Chattopadhyay
Meta Platforms Inc.
{pedroerp,oerling,mbasmanova,kevinwilfong,lsakka,kpai,weihe,biswapesh}@fb.com

ABSTRACT

The ad-hoc development of new specialized computation engines targeted to very specific data workloads has created a siloed data landscape. Commonly, these engines share little to nothing with each other and are hard to maintain, evolve, and optimize, and ultimately provide an inconsistent experience to data users. In order to address these issues, Meta has created Velox, a novel open source C++ database acceleration library. Velox provides reusable, extensible, high-performance, and dialect-agnostic data processing components for building execution engines, and enhancing data management systems. The library heavily relies on vectorization and adaptivity, and is designed from the ground up to support efficient computation over complex data types due to their ubiquity in modern workloads. Velox is currently integrated or being integrated with more than a dozen data systems at Meta, including analytical query engines such as Presto and Spark, stream processing platforms, message buses and data warehouse ingestion infrastructure, machine learning systems for feature engineering and data preprocessing (PyTorch), and more. It provides benefits in terms of (a) efficiency wins by democratizing optimizations previously only found in individual engines, (b) increased consistency for data users, and (c) engineering efficiency by promoting reusability.

PVLDB Reference Format:
Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, Biswapesh Chattopadhyay. Velox: Meta's Unified Execution Engine. PVLDB, 15(12): 3372 - 3384, 2022.
doi:10.14778/3554821.3554829

1 INTRODUCTION

The increasing workload diversity in modern data use cases coupled with exponential dataset growth have led to the proliferation of specialized query and computation engines, each targeted to a very specific type of workload. Data processing requirements have grown from simple transaction processing and analytics (both batch and interactive), to ETL and bulk data movement, to realtime stream processing, to log and timeseries processing for monitoring use cases, to more recently, a plethora of artificial intelligence (AI) and machine learning (ML) use cases including data preprocessing and feature engineering.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554829

This evolution has created a siloed data ecosystem composed of dozens of specialized engines that are built using different frameworks and libraries and share little to nothing with each other, are written in different languages, and are maintained by different engineering teams. Moreover, evolving and optimizing these engines as hardware and use cases evolve, is cost prohibitive if done on a per-engine basis. For example, extending every engine to better leverage novel hardware advancements, like cache-coherent accelerators and NVRAM, supporting features like Tensor data types for ML workloads, and leveraging future innovations made by the research community are impractical and invariably lead to engines with disparate sets of optimizations and features. More importantly, this fragmentation ultimately impacts the productivity of data users, who are commonly required to interact with several different engines to finish a particular task. The available data types, functions, and aggregates vary across these systems, and the behavior of those functions, null handling, and casting can be vastly inconsistent across engines. 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.

Although specialized engines, by definition, provide specialized behavior that justifies their existence, the main differences are commonly in the language frontend (SQL, dataframes, and other DSLs), the optimizer, the way tasks are distributed among worker nodes (also referred to as the *runtime*), and the IO layer. The execution engines at the core of these systems are all rather similar. All engines need a type system to represent scalar and complex data types, an in memory representation of these (often columnar) datasets, an expression evaluation system, operators (such as joins, aggregation, and sort), in addition to storage and network serialization, encoding formats, and resource management primitives.

In order to address these issues, Meta has developed Velox, a novel C++ database acceleration library that provides reusable, extensible, and high-performance data processing components which can be used to build, enhance, or replace execution engines in existing data management systems. Velox is designed from the ground up to efficiently support complex types due to their ubiquity in modern workloads, and heavily relies on vectorization [4] and adaptivity. Velox components are language, dialect, and engine-agnostic, and provide many extensibility points where developers can specialize the library behavior and match a particular engine's requirements. In common usage scenarios, Velox takes a fully optimized query plan as input and performs the described computation using the resources available in the local node. As such, Velox does not provide a SQL parser, a dataframe layer, other DSLs, or a global query optimizer, and it is usually not meant to be used directly by data users.

Velox's value proposition is three-fold:

- **Efficiency:** Velox democratizes runtime optimizations previously only implemented in individual engines, such as fully leveraging SIMD, lazy evaluation, adaptive predicate re-ordering and pushdown, common subexpression elimination, execution over encoded data, code generation, and more.
- **Consistency:** by leveraging the same execution library, compute engines can expose the exact same data types and scalar/aggregate function packages, and thus provide a more consistent experience to data users due to their unified behavior.
- **Engineering Efficiency:** all features and runtime optimizations available in Velox are developed and maintained once, thus reducing engineering duplication and promoting reusability.

Velox is under active development and is already integrated or being integrated with more than a dozen data systems at Meta (and beyond), such as Presto, Spark, PyTorch, XStream (stream processing), F3 (feature engineering), FBETL (data ingestion), XSQL (distributed transaction processing), Scribe (message bus infrastructure), Saber (high QPS external serving), and others.

We believe Velox to be a major step towards making data systems more modular and interoperable, with the ultimate goal of providing a more responsible implementation of the *"one size does not fit all"* mantra. Considering the potential impact in the community, Velox is open-source[1] and backed by a fast growing community, including members such as Ahana, ByteDance, Intel, and many other major technology companies and academic partners. Lastly, we also believe that, strategically, Velox will allow Meta to partner with hardware vendors and proactively prepare our data systems for tomorrow's hardware, in addition to streamlining collaborations with researchers and research labs.

In this paper, we make the following contributions:

- We detail the Velox library, its components, extensibility points, and main optimizations.
- We describe how Velox is being integrated with compute engines targeted at very diverse workloads, such as batch and interactive analytics, stream processing, data warehouse ingestion, ML, and more.
- We highlight how Velox is transforming Meta's data landscape, which has traditionally been composed of siloed and specialized engines providing inconsistent semantics for data users.
- We present micro-benchmarks to motivate Velox's main optimizations, in addition to experimental results with Velox's integration with Presto.
- We discuss lessons learned during this journey, future work, and open questions with the hope of motivating further research and fostering collaboration.

2 LIBRARY OVERVIEW

Velox is an open source C++ database acceleration library that provides high-performance, reusable, and extensible data processing components, which can be used to accelerate, extend, and enhance data computation engines. Velox does not provide a language frontend, such as a SQL parser, dataframe layer, or other DSLs; instead, it expects a fully optimized query plan as input describing the computation to be performed, and executes it locally using the resources available in the local host.

Furthermore, Velox does not provide a global query optimizer, but at execution time leverages numerous adaptivity techniques, such as filter and conjunct reordering, dynamic filter pushdown, and adaptive column prefetching. In other words, the components provided by Velox usually sit on the *data-plane*, while individual engines are responsible for providing the *control-plane*. The high-level components provided by Velox are:

- **Type:** a generic type system that allows users to represent scalar, complex, and nested data types, including structs, maps, arrays, tensors, and more.
- **Vector:** an *Arrow-compatible*[2] columnar memory layout module, supporting multiple encodings, such as Flat, Dictionary, Constant, Sequence/RLE, and Bias (frame of reference), in addition to a lazy materialization pattern and support for out-of-order result buffer population.
- **Expression Eval:** a fully vectorized expression evaluation engine built based on Vector-encoded data, leveraging techniques such as common subexpression elimination, constant folding, efficient null propagation, encoding-aware evaluation, and dictionary memoization.
- **Functions:** APIs that can be used by developers to build custom functions, providing a simple (row-by-row) and vectorized (batch-by-batch) interface for scalar functions, and APIs for aggregate functions. Function packages compatible with popular SQL dialects are also provided by the library (currently, for Presto and Spark).
- **Operators:** implementation of common data processing operators such as TableScan, Project, Filter, Aggregation, Exchange/Merge, OrderBy, HashJoin, MergeJoin, Unnest, and more.
- **I/O:** a generic connector interface that allows for pluggable file format encoders/decoders and storage adapters. Support for popular formats such as ORC and Parquet, and S3 and HDFS storage systems are included in the library.
- **Serializers:** a serialization interface targeting network communication where different wire protocols can be implemented, supporting PrestoPage and Spark's UnsafeRow formats.
- **Resource Management:** a collection of primitives for handling computational resources, such as memory arenas and buffer management, tasks, drivers, and thread pools for CPU and thread execution, spilling, and caching.

Engines integrating with Velox can choose which components to use based on the functionality required. For instance, engines with simple data representation and serialization requirements can leverage Type, Vector, and Serializer only, while a complete SQL analytical query engine would require the full extent of operators and resource management primitives available.

[1] Velox is available at https://github.com/facebookincubator/velox

[2] The format differences between Velox Vectors and Apache Arrow are discussed in Section 4.2.1.

In addition to being modular, Velox also provides extensibility APIs that can be used to customize the library. Developers can use these APIs to add plugins to support custom data types, scalar and aggregate functions, engine-specific operators, new serialization formats, file encodings, and storage adapters. The decision about whether a particular plugin will be provided as part of the main library or not is based on its genericity: if it is used by multiple engines (e.g. Parquet and ORC file encoders, and common operators such as Aggregate, OrderBy, and HashJoin), they are included in the main library; otherwise, they are provided as part of the client's engine codebase (e.g. ML-specific functions and stream processing operators).

3 USE CASES

This Section describes the main use cases for Velox at Meta, and how different specialized engines are leveraging Velox to accelerate, unify, and consolidate user workloads. We start by describing interactive SQL analytics use cases in Presto [17] (Subsection 3.1), and large batch/ETL SQL analytics workloads in Spark [18] (Subsection 3.2). Next, we outline the realtime data infrastructure integration by describing how Velox is being used in a stream processing platform called XStream (Subsection 3.3.1), in a messaging bus called Scribe (Subsection 3.3.2), and in a data ingestion system called FBETL (Subsection 3.3.3). Finally, we describe Velox integrations with ML platforms focused on data preprocessing and feature engineering in Subsection 3.4.

3.1 Presto

Presto is an open-source distributed query engine created by Meta circa 2013, which allows users to run SQL queries over data stored in Hive and other environments. It currently powers most of the interactive (low-latency) SQL analytic workloads at Meta and part of the batch workloads, though most of the heavy-lifting ETL processing is done in Spark. Presto is organized in a two-tiered architecture composed of a coordinator node, responsible for receiving user queries, SQL parsing, metadata resolution, global query optimization, and resource management; and worker nodes, which are responsible for the actual execution of a query given a query plan fragment. Both coordinator and worker processes share the same Java codebase and communicate via a HTTP REST interface. Considering that all data processing and shuffling happens within or between worker nodes, there is usually a 100-1000 to 1 ratio between the number of workers and coordinators, and thus the vast majority of cpu time is spent on worker nodes.

Prestissimo[3] is the codename of the project aimed to replace Java workers by a C++ process based on Velox, targeting efficiency gains. Prestissimo provides a C++ implementation of Presto's HTTP REST interface, including worker-to-worker exchange wire protocol and coordinator-to-worker orchestration, and status reporting endpoints, thereby providing a drop-in replacement for Presto workers. The main query workflow consists in receiving a Presto plan fragment from a Java coordinator, translating it into a Velox query plan, and handing it off to Velox for execution. In this manner, no Java processes, JVM, or expensive garbage collection procedures are needed on worker nodes, which used to be a source of operational issues.

Unification. Prestissimo leverages the full extent of the Velox library: types, Vectors, expression eval, functions, operators, serializers, I/O, and resource management primitives. Being the first Velox implementation to provide end-to-end functionality, many of the components implemented to support Prestissimo, such as Presto wire protocol and Presto function/aggregate packages, today constitute the core of Velox and are, in fact, reused in other engines. Examples are the use of Presto wire protocol in the realtime data infrastructure (described in Subsection 3.3), and use of Presto function packages in Stream Processing and ML platforms (discussed in Subsection 3.4).

3.2 Spark

Spark is an open-source unified computation engine for large-scale data processing, which manages and coordinates the execution of tasks on data across a cluster of servers. Spark applications consist of a driver process and a set of executor processes: the driver is responsible for task planning, scheduling, and communicating with an external resource manager, while executors are responsible for performing the actual computation and communication with the remote storage system. At Meta, Spark is commonly used for the execution of batch and ETL SQL queries, expressed using SparkSQL or the Dataframe API, due to Spark's superior fault-tolerance characteristics for long-running queries/applications.

Spruce is the codename for the Velox implementation for Spark. Spruce leverages a pre-existing interface that allows users to execute arbitrary binaries in Spark, called *Spark script transform*, to offload execution to an external C++ process in which Velox is executed. At query time, a Spark executor receives a query plan fragment, serializes it, and forwards it to the external C++ process using the transform interface. The external process (called SparkCpp), deserializes the plan, converts it to a Velox plan, and executes it using Velox.

The SparkCpp process uses Velox's extensibility APIs to add operators, scalar, and aggregate functions that make the new C++ code fully compatible with the existing Spark Scala execution engine. SparkCpp also adds an UnsafeRow serializer to Velox, which is the format used for data shuffling in Spark and to return data back to clients. Even though Velox is being customized differently when integrating with Presto and Spark in order to maintain backwards compatibility for existing queries, having a shared execution engine paves the way for achieving semantic equivalence between engines in the future, along with providing immediate efficiency gains.

3.3 Realtime Data Infrastructure

Velox is also being used by Meta's realtime data infrastructure in three different, but related, use cases: (a) stream processing, (b) distributed messaging infrastructure, and (c) data ingestion. These three use cases are described in the next subsections.

3.3.1 Stream Processing. XStream is Meta's stream processing platform which allows users to create stream processing applications, expressed either using SQL or a dataframe-like fluent API. XStream applications commonly read data continuously from Meta's messaging infrastructure, a system called Scribe, and after applying

[3]From music theory: "the fastest possible tempo; faster than Presto".

business logic, write results back to Scribe or other data sinks, such as log analytics platform (Scuba) or key-value store systems for online serving. Although the abstraction provided by stream processing is to operate over a single row at a time, in practice reads and writes are batched to optimize IO, and thus benefit from Velox's vectorized execution model. In production deployments, data is usually batched in chunks of up to 500kb, buffered over a window of at most 20 seconds.

Most data processing operations available in XStream map directly to Velox operators and hence are directly reused, e.g, projections, filters, and, as ongoing work, lookup joins. The use of Velox also allows XStream to expose the same function package used in Presto, increasing consistency and reducing friction for users who are already familiar with Presto SQL.

Because stream processing aggregations require engine-specific logic related to temporal windows, namely tumbling, hopping and session windows, aggregations are implemented as an extension to Velox in XStream. Even though these special aggregations are most commonly required in stream processing applications, there are plans to provide this operation as part of the core Velox library and expose it to Presto and Spark users as temporal extensions. This work is only possible without substantial duplication of effort due to the unified execution engine provided by Velox.

3.3.2 Messaging Bus. Scribe is a distributed messaging system for collecting, aggregating, and delivering high volumes of data with low latency, serving as the main entry point to data ingestion pipelines at Meta. In common usage scenarios, data is generated in the web tier and written to Scribe, and subsequently read by stream processing applications (which might write the processed data back to a new Scribe pipe), or delivered to a system called FBETL for data warehouse ingestion.

Data is written to Scribe on a row-by-row basis (log production) and was traditionally read in the same manner. Today, Scribe Read Service (the service responsible for serving read requests from Scribe) is able to leverage the full extent of wire serialization formats available in Velox, which are more efficient due to column-oriented encoding, and can be easily deserialized to Velox Vectors by data consumers. In addition, Velox usage in Scribe Read Service allows data consumers to pushdown operations such as projections (read a subset of columns) and filtering closer to the storage, e.g. for stream processing applications, reducing the amount of data read from Scribe, and in many cases resulting in cross data center traffic reduction. Furthermore, the filters and projections pushed down to Scribe provide the same semantics as in other compute engines, promoting a more consistent data user experience.

3.3.3 Data Ingestion. FBETL is Meta's data ingestion engine, responsible for two main use cases: data warehouse ingestion, and database ingestion. Data warehouse ingestion is the process of converting data read from Scribe pipes into warehouse files (commonly encoded using the ORC format, or an internal variant called DWRF) for longer retention and further processing. Other than reusing the same ORC file encoder codebase as other SQL engines that generate ORC files (such as Spark and Presto), thus promoting consistency, using Velox in FBETL allows users to specify data transformations (projections) including expressions, UDFs, and filtering applied to the data at ingestion time. This frees users from having to create a full stream processing application to achieve the same result, which would incur the overhead of writing to a new Scribe pipe, and reading it again for ingestion. Once again, Velox provides a consistent user experience by exposing the same set of functions and semantics available in other engines; e.g, an end-user could reuse any function available in Presto to specify the transformation applied to the data at ingestion time, in addition to reducing code duplication.

Another ingestion use case is database ingestion, which is the process of scraping operational database logs and saving snapshots to the data warehouse. Besides the benefits aforementioned, Velox also aids the implementation of snapshotting, which is a standalone periodic process that consists of reading the previous snapshot from a warehouse table partition, applying the modifications read from database redo logs (an operation similar to a merge-join), and writing the results back to a new table partition.

3.4 Machine Learning

Almost every machine learning (ML) pipeline contains a pre-processing step that wrangles the data into the right form before it can be fed into the models. These transformations usually sit between offline processing done by Data Analytics systems, such as large scale joins, aggregations and filtering, and neural networks, which are operations on tensors like matrix multiplications and convolutions, powered by machine learning frameworks such as PyTorch. This process, also referred to as data preprocessing, is usually limited to row-wise transformations such as normalization, embedding lookups, and image cropping, and can usually be expressed using expression evaluation and user-defined functions, as provided by traditional computation engines.

Despite the undeniable similarities, Data Analytics and ML infrastructure have largely evolved independently at Meta. The exponential growth in the demand for ML systems along with the lack of proper reusable components, resulted in a highly fragmented space composed of data preprocessing libraries providing incomplete data type support, incompatible memory representations, and inconsistent function packages. An internal survey discovered about 14 libraries being used for data preprocessing at Meta, providing disparate functionality, suboptimal efficiency, and inconsistent user experience. Moreover, although these transformations only encompass the processing needed before the training process starts, it has been estimated that preprocessing computation can consume up to 50% of the resources used for ML workloads [20].

3.4.1 Data Preprocessing. TorchArrow is a new project from PyTorch aimed at unifying and providing first-class structured data preprocessing capabilities for ML users. It provides a Python dataframe layer similar to Pandas, deeply integrated into the PyTorch ecosystem. TorchArrow internally translates the dataframe representation into a Velox plan and delegates it to Velox for execution; this integration, besides helping converge the fragmented space of ML data preprocessing libraries, also allows Meta to consolidate execution engine code between Data Analytics and ML infrastructure - an initiative loosely referred to as "DI for AI" (Data Infrastructure for Artificial Intelligence). It also provides a more consistent experience for ML end users, who are commonly required to interact with different computation engines to complete a particular task, such

as large scale SQL queries for data preparation, interactive SQL analytics for debugging, and stream processing, by exposing the same functions and ensuring consistent behavior across engines.

3.4.2 Feature Engineering. Another use case for such transformations are feature engineering workflows, which are the process of using domain knowledge to extract useful information in the form of features that can be consumed by ML algorithms. Meta's feature engineering framework, a system called F3[4], allows users to programmatically create features by defining a F3 DSL file that specifies the transformations required to generate a particular feature, e.g, transforming birthdate into a numeric age value. Based on the DSL definition, F3 manages both offline and realtime data generation by integrating the feature generation transformations into Spark (for batch processing) and XStream (for realtime datasets). Lastly, the same DSL definition is also used during online serving to generate the feature values fed into the models during inference, providing consistency between training and serving/inference.

F3 is currently in the process of unifying its execution engine with Velox. Considering that both Spark and XStream already use Velox for execution, both offline and realtime F3 data generation pipelines can natively run in these computation engines. However, the integration with F3's online serving path is a work in progress and poses an interesting challenge: considering that many of these transformations are called from applications in the user serving path, with very high QPS and low latency requirements, and executed against small batches of data (often a single record), Velox's vectorized execution engine is not the optimal fit due to the interpretation overhead. For this use case, considering that the DAG itself is mostly static, the team is investing in adding codegen based execution to Velox - more about codegen is discussed in Subsection 4.3.3.

4 DEEP DIVE

The following subsections detail the main components provided by Velox, and present experimental micro-benchmark results.

4.1 Type System

At its core, Velox provides a type system that allows users to represent primitive types, including integers and floats of different precision, strings (both varchar and varbinary types), dates, timestamps, and functions (lambdas). It also supports complex types such as arrays, fixed-size arrays (used to implement ML tensors), maps, and rows/structs; all these types can be arbitrarily nested and provide serialization/deserialization methods. Finally, Velox provides an opaque data type that developers can use to easily wrap arbitrary C++ data structures.

The type system is extensible to allow developers to add engine-specific types without having to modify the main library. Examples are Presto's HyperLogLog[5] type for cardinality estimation, and other Presto date/time specific data types such as *timestamp with timezone*. The types added through type extensibility can then be used when building custom scalar and aggregate functions.

[4]Historically, **F**acebook **F**eature **F**ramework - F3.

[5]Although the current HLL implementation is Presto-specific, we have plans to generalize and provide it as part of the core library.

4.2 Vectors

Velox Vectors allow developers to represent columnar datasets in memory leveraging a variety of encoding formats, and are used as input and output to most other components, The basic memory layout extends the Apache Arrow format [2], and is composed of a *size* variable (denoting the number of rows represented in the Vector), the data type (as described in the previous subsection), and an optional nullability bitmap to represent null values. The base Vector class also provides a collection of methods to help users copy, resize, hash, compare, and print Vectors.

Vectors can represent fixed-size (e.g. primitive types like integers and floats) or variable-size elements (e.g. strings, arrays, maps, and structs/rows). Vectors can also be nested in arbitrary ways (e.g. arrays of arrays of structs containing strings and other primitive types), and can leverage different encoding formats such as flat, dictionary, constant, sequence/RLE, and bias (frame of reference), though the system or component generating a particular vector is responsible for choosing the appropriate encoding. All Vector data is stored using Velox Buffers, which are contiguous pieces of memory allocated from a memory pool, and that can be subclassed to support different ownership modes (e.g. owned and buffer view). All Vectors and Buffers are reference counted, and a single Buffer can be referenced by multiple Vectors; naturally, only singly-referenced data is mutable, but any Vector and Buffer can be made writable via copy-on-write.

Moreover, Velox provides the concept of *Lazy Vectors*, which are Vectors that only get populated upon first use. Lazy Vectors are useful in cardinality reduction operations such as joins and conditionals in projections, where, depending on the operation's selectivity, one can entirely avoid materialization, or scope it to a few surviving rows. This feature is particularly useful when reading Vector data from remote storage (such as S3 or HDFS), as it can optimize away entire IO operations for sparsely accessed columns. Lazy vectors also provide support for running a callback over the loaded data, which can be used to pushdown computation (such as aggregations) without having to materialize an intermediate Vector.

Frequently, developers have no control about how a particular Vector was created, e.g. when implementing a scalar function or operator, and therefore need to deal with input data that can be arbitrarily encoded. While on the one hand this gives developers the flexibility to leverage the input data encoding for efficient processing (e.g. only running a particular operation over the distinct values on dictionary-encoded input), on the other hand this adds complexity and increases the cognitive burden on developers. To address this issue, Velox also provides the Decoded Vector abstraction, which transforms an arbitrarily-encoded Vector into a flat vector and a set of indices for all or parts of its elements, and exposes a logically consistent API. Decoded Vectors are zero-copy for flat, constant, and single-level dictionary encoded inputs (the most common cases), but need to materialize a new array of dictionary indices to cover nestings of multiple dictionaries/run length encodings.

4.2.1 Arrow Comparison. Although Velox Vectors are based and compatible with the Apache Arrow format, we deliberately decided to extend the standard to accelerate data processing operations

commonly found in Velox. The three areas where the Velox Vectors and Apache Arrow formats diverge are discussed below.

1. Strings. While Arrow represents strings using the traditional layout for variable-sized elements, consisting of one buffer containing the string contents, and either one *lengths* buffer denoting string sizes or one *offsets* buffer marking where strings start, Velox follows the StringView representation described in [11]. In this layout, string vectors are also composed of two buffers, one for metadata, containing 16 bytes per string element, and one storing the strings' data. The string metadata class, called StringView, is defined as:

```
struct StringView {
  uint32_t size_;
  char prefix_[4];
  union {
    char inlined[8];
    const char* data;
  } value_;
}
```

StringViews always store a small (4 bytes) prefix inline, focusing on short-circuiting failed comparisons to speed up operations such as filtering and ordering. In addition, small strings up to 12 bytes are fully inlined, and do not require access to the secondary buffer. This layout also allows certain string operations, such as $trim()$ and $substr()$, to be executed zero-copy by only updating the metadata pointers.

2. Out-of-order Write Support. In order to efficiently support execution of conditionals, such as IF and SWITCH operations, Velox extends the Apache Arrow format to support out-of-order writes. In these transformations, the condition is first evaluated to generate a bitmask describing which branch to take for each row. Subsequently, based on the generated bitmask, each branch is processed individually in a vectorized manner, writing the calculated values to the single output vector. Primitive types can always be written out-of-order since the element size is constant. Moreover, using the representation described above, strings can also be written out-of-order because the string metadata objects have a constant size (16 bytes). In order to support out-of-order writes for the remaining variable-sized types (such as arrays and maps), Velox maintains *both* lengths and offsets buffers. Besides speeding up the execution of conditionals, this layout gives the engine more flexibility to slice and rearrange elements without copying, since the lengths and offset of each array/map can be updated independently, other than allowing arrays/maps with overlapping elements.

3. More Encodings. Velox Vectors also add two other encoding formats commonly found in data warehouse workloads: run-length encoding (RLE), and constant encoding. The latter is used to represent that all values in a column are the same, for instance, to represent literals and partition keys.

Despite the divergence, Velox provides a conversion API for engines that need to be interoperable with Apache Arrow, performing zero-copy format conversion when possible, and re-arranging the data when needed. Lastly, these optimizations have recently been proposed to the Apache Arrow community [10]. Although the community was receptive to the idea, the possibility of incorporating these ideas to the Apache Arrow format is still under discussion.

4.3 Expression Eval

Velox provides a vectorized expression evaluation engine that can be used in a few situations: first, it is used by the FilterProject operator, to evaluate filter and projection expressions; second, it is used by TableScan and IO connectors to consistently evaluate predicate pushdown; and third, it can be used as a standalone component for engines that only require expression evaluation capabilities, such as realtime infrastructure and most data preprocessing operations for ML use cases.

Expression evaluation takes expression trees as input. Each node in the tree represents one of the following: (a) a reference to an input column, (b) a constant (or literal), (c) a function call, represented by a function name and a list of input expressions, (d) a CAST expression, or (e) a lambda function. In addition to traditional functions, function call nodes are also used to express conjunctions (AND/OR), conditionals (IF/SWITCH), and try expressions. Tree nodes also contain metadata regarding determinism (whether the subexpression deterministically produces the same results for the same inputs), and null propagation (if a null value in any of the input columns causes this expression to always return null). Expression evaluation is broken down into two steps, compilation and evaluation, which are detailed in the next subsections.

4.3.1 Compilation. The compilation step takes a list of one or more input expression trees, and produces a compiled (executable) expression. The main runtime optimizations applied during this process are described below.

Common Subexpression Elimination. The expression compilation process is responsible for identifying common subexpressions, which are optimized and calculated only once during evaluation. For example, consider the following expression expression: *strpos(upper(a), 'FOO') > 0 OR strpos(upper(a), 'BAR') > 0.* In this example, $upper(a)$ is a common subexpression and therefore will be calculated only once. FilterProject operator also benefits from this capability by creating a single compiled expression object for all of the filter and project expressions, allowing subexpressions to be shared between projection and filter expressions.

Constant Folding. The compilation step is also responsible for applying constant folding, which is the process of evaluating deterministic subexpressions that do not depend on any input columns, and replacing it by a constant/literal expression node. For example, the expression $upper(a) = upper(`Foo`)$ would be transformed into $upper(a) = `FOO`$ during compilation.

Adaptive Conjunct Reordering. Lastly, when evaluating AND or OR expressions, the engine dynamically tracks the performance of individual conjunts and chooses to evaluate the most effective conjunct first, i.e. the one which drops the most values in least time, as per $time/(1 + n\_in - n\_out)$; the least score is the best. In order to maximize the effect of adaptive conjunct reordering during execution, expression compilation also flattens adjacent AND/OR expressions. For instance, the input expression AND(AND(AND(a, b), c), AND(d, e)) is flattened to a single AND(a, b, c, d, e) node during compilation.

4.3.2 Evaluation. The evaluation process takes a compiled expression and an input dataset (represented using Velox Vectors), and after calculating the results returns an output dataset. The process

consists of a recursive descent of the expression tree, passing down a row mask identifying the active (non-null and not masked out by conditionals) elements. On each step, evaluation can be avoided in two cases (a) if the current node is a common subexpression and the result was already calculated, or (b) if the expression is marked as propagating nulls, and any of its inputs are null. The latter step can efficiently be implemented by simply combining the nullability bitmask of all inputs and updating the active rows mask, using SIMD operations.

Peeling. When inputs are dictionary-encoded, deterministic expressions can be efficiently computed by only considering distinct values. This is achieved by first validating that all input columns share the same dictionary wrappings, and if so, peeling off these wrappings to extract the set of inner vectors (the distinct values), evaluating the expression on these inner vectors, and wrapping the results back into dictionary vectors using the original wrappings. For example, consider a dictionary-encoded vector representing a 1k row dataset for a *color* column, encoded using a dictionary consisting of 3 values: 0 - red, 1 - green, 2 - blue. The memory layout consists of an indices buffer of 1k values in the range of [0, 2], and an inner vector of size 3, containing the following values: [red, green, blue]. When evaluating an expression such as $upper(color)$, for instance, after peeling off the dictionary wrappings, the upper function is only applied to the 3 distinct values - [red, green, blue] - to produce another vector of size 3: [RED, GREEN, BLUE]. As the last step, the result is wrapped in a dictionary vector using the original indices, producing a dictionary-encoded vector that represents 1k color values in upper case.

Memoization. The evaluation step can be repeated as needed to process multiple batches of data reusing the same compiled expression object. When multiple batches of data are read from the TableScan operator, for instance, it is common for batches to be dictionary-encoded and reference the same base vector. In the example described above, a color column may have millions of rows that refer to the same base set of distinct values, [red, green, blue], represented by dictionary-encoded vectors having the same base vector, but different indices buffers. The evaluation engine leverages this property and remembers the calculated expression evaluation results over the underlying inner vector, reusing these results for subsequent batches. For each new batch, it just wraps the existing calculated results using the indices buffer of the input vector.

In conclusion, many of the techniques described might not present considerable improvements for simple arithmetic operations over primitive types, but they do provide substantial speed up for complex expressions such as string operations, regular expressions, array/map manipulation, and other operations over nested data types. Velox makes a conscious decision of optimizing for these workloads, considering empirical data indicating these operations as being top cpu time consumers, while still providing fast paths for the base cases.

4.3.3 Code Generation. Velox also provides experimental support for expression evaluation through code generation (codegen). When enabled, at execution time the entire expression tree is rewritten as a C++ function's source code, which is written to a source file and compiled into a shared library using a regular compiler such as gcc or clang. The shared library is then dynamically linked to the main process and used for evaluation instead of the vectorized interpreted path. Considering that the codegen process involves a full compiler invocation, compilation times are usually high (up to 10s in some cases), and are not meant to be used in short lived queries or interactive workloads. Rather, our initial evaluation is focused on large ETL queries (many hours to days of execution), and usages where the expression tree is fixed, such as the feature engineering use case described in Section 3.4.2.

As of today, codegen support in Velox is still experimental. Use cases where codegen provides clear benefits, outweighting compilation delays, decreased developer productivity, and debuggability are under investigation. Furthermore, a full evaluation of the trade-offs between the codegen approach versus traditional JIT LLVM-based compilation, in addition to the exploration of runtime adaptivity between the vectorized and codegen-based evaluation paths, building on the work described in [9], are open questions and avenues for future research.

4.4 Functions

Velox provides APIs that allows developers to build custom scalar and aggregate functions. These APIs are described in the next subsections.

4.4.1 Scalar Functions. Scalar functions are functions that take values from a single row as parameters, and produce a single output row. As a vectorized engine, Velox scalar function API is also vectorized and provides input parameters as Vectors (batch-by-batch), along with their nullability buffers, and a bitmap describing the set of active rows. In many cases, vectorized scalar functions can leverage the columnar data layout to produce results in constant time. For example, $is\_null()$ can be implemented in constant time by just returning the internal nullability buffer, $cardinality()$ can leverage the internal *lengths* buffer that represents the size of each array in a vector, and $map\_keys()$/$map\_values()$ can return either the keys or values internal buffer of the input MapVector.

However, for the remaining functions that cannot leverage the columnar format, requiring developers to manually iterate over each input row, correctly handle the nullability buffers, different input (and output) encoding formats, complex nested types, and allocating or reusing output buffers, turned out to be too cumbersome (and error-prone), particularly considering that the number of developers implementing these functions is constantly growing. Moreover, due to the diversity in modern workloads' needs, such as advanced string and json processing, date and time conversions, array/map/struct manipulation, regular expressions, mathematical functions for data science, and more, scalar functions quickly became the largest portion of Velox's codebase.

Simple Functions. With the developer productivity (and reliability) issue described above in mind, Velox also provides a *simple scalar function API* that strives for simplicity, ease-of-use, and hides as many details of the underlying engine and data layout as possible, while still providing the same level of performance as vectorized functions. The simple scalar function API allows developers to express their business logic by providing a C++ function that takes a single row of values at a time (as opposed to full Vectors), for example:

```
class MultiplyFunction {
  void call(
      int64_t& result,
      const int64_t& a,
      const int64_t& b) {
    result = a * b;
  }
};
registerFunction<
    MultiplyFunction,
    int64_t,
    int64_t,
    int64_t>({"multiply"});
```

The class implementing the scalar function is required to provide a *call*() method, where the first parameter is the function's return value, passed as a reference, followed by the function parameters, taken as const references. The method might return bool, denoting the nullability of the return value (true means not-null), or void, signaling the function never produces nulls. By default, it is assumed the function presents *default null behavior*, whereby any nulls detected on inputs automatically produce a null output without calling the function above. Functions which require a different null behavior can provide a *callNullable*() method that takes the input parameters as a C++ pointer instead of a reference.

The simple function framework hides the input data encoding format from developers by leveraging the DecodedVectors abstraction described in Subsection 4.2, and leverages C++ template metaprogramming to efficiently apply the provided method against batches of rows (Vectors), without incurring dispatching costs per row. In fact, the framework is optimized and provides hints to the C++ compiler to ensure that in most cases all logic within the execution loop is inlined (thus preventing function calls and cache misses in the hot loop), and allowing compilers to apply auto-vectorization. For example, major compilers such as clang and gcc are able to automatically generate SIMD code for arithmetic functions solely based on the function definition above, when compiling Velox's engine.

The simple function framework directly maps all primitive types to their corresponding C++ types. Non-primitive types, such as strings, arrays, maps, and rows/structs, are implemented by using proxy objects to prevent the overhead of materializing and copying data into temporary objects, such as *std::string* or *std::vector*. The proxy objects present a similar API to their std counterparts (for instance ArrayReader/ArrayWriter's API is similar to *std::vector*), but operate directly over the underlying data represented using Velox Vectors, and do not incur in extra allocation or copies.

Figure 1 illustrates the performance comparison of three different functions that were initially implemented using the vectorized API by a developer integrating with Velox, versus its implementation using the simple API. In the first function, *plus()*, we show that operations like arithmetics over primitive types can be implemented using the simple framework at no cost, despite the substantial developer productivity gains. For the remaining functions using complex types, we have observed a curious fact: the simple function implementation, besides being considerably easier to read and write, was also more efficient. After investigation, we discovered that the performance gap was due to missed optimization opportunities in the implementation of these vectorized functions, such as flat

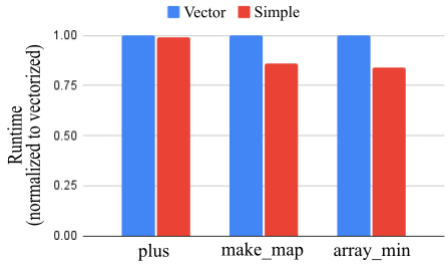

Figure 1: Comparison of three different functions implemented using the vectorized and simple APIs.

encoding and null-free fast-paths, which are automatically applied by the simple function framework. Although this gap can be easily overcome by optimizing the vectorized function's implementation, the framework encapsulates this complexity and automatically takes the burden off of developers.

Furthermore, functions implemented in this framework can also specify their *determinism* and *null behavior*, to allow or disallow some of the optimizations described in Subsection 4.3 from being applied by the expression evaluation engine. Most of the optimizations only apply to deterministic functions (always produces the same results for the same inputs) with default null behavior (always produces null if any of the inputs are null). Fortunately, the vast majority of functions present these behaviors, and thus are eligible for the full extent of optimizations described in Subsection 4.3, with very few exceptions, such as *rand*() and *shuffle()*.

Advanced String Processing. Most string manipulation functions need to correctly handle UTF-8 characters, posing unnecessary overhead when inputs happen to be composed of ASCII characters only. To address this issue, the simple function framework allows developers to provide a specialized version of the *call*() function, *callAscii*(), which is automatically called when string inputs are ASCII-only. This feature is based on the observation that the vast majority of strings in Meta's data warehouse tables are composed of ASCII characters only. Furthermore, simple functions can also declare their *ASCII behavior*, i.e, whether the evaluation engine is allowed to assume that string outputs generated by this function are ASCII-only if all string inputs were ASCII-only. This flag allows the expression evaluation engine to skip ASCII detection routines on data generated by these functions. Figure 2 compares results of common string manipulation functions over ASCII-only inputs, with and without the optimization enabled.

Many string operations, such as *substr*(), *trim*(), and other string tokenization functions, can produce zero-copy results by referencing input strings in the generated output strings. In order to achieve this, the function developer needs to set a flag in the function class, advising the engine to carry a reference to a specific input string buffer on the generated output string Vector. The same feature is also applicable for functions generating arrays of strings, such as *split*(). Figure 3 shows micro-benchmarks comparing three different implementations of *substr*(): no assumptions over input encoding and no buffer reuse (NoOpts), ASCII-only, and ASCII-only with buffer reuse.

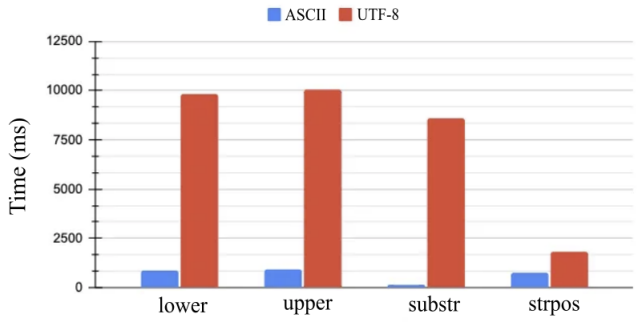

Figure 2: Effect of ASCII fast path optimization in different functions.

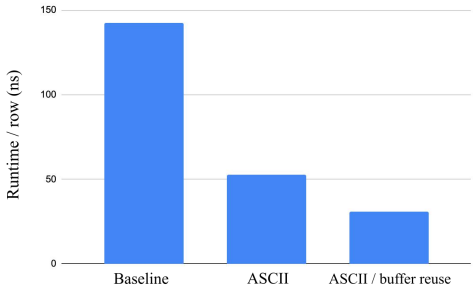

Figure 3: Different implementations of substr(): baseline, ASCII-only, and ASCII-only and buffer reuse.

4.4.2 Aggregate Functions. Aggregates are functions that summarize multiple rows from a particular group into a single output row. Aggregate functions in Velox are typically calculated in two steps: (a) *partial aggregation* takes raw input data and produces intermediate results, and (b) *final aggregation* takes intermediate results and produces the final result. Velox also allows developers to specify two additional steps: (c) *single aggregation*, used when data is already partitioned on the grouping keys, and therefore no shuffle or intermediate results are necessary, and (d) *intermediate aggregation*, which are used to combine the results of partial aggregations, e.g, when computed by multiple threads in parallel to reduce the amount of data sent to the final aggregation stage.

Aggregate functions can be classified based on the characteristics of their intermediate results (also called *accumulators*), into fixed-size and variable-size. Functions like *count*(), *sum*(), *avg*(), *min*(), and *max*() use fixed-size accumulators, while functions such as *distinct*(), *pct*() and their approximate counterparts require variable-sized accumulators. Considering that during aggregation data is stored in rows, where each row corresponds to a single group or unique combination of grouping key values, fixed-size accumulators are stored inline within the row itself, and variable-sized accumulators are stored in a separate buffer, and a pointer is stored in the row. More details about the hashing adaptivity and hash table layout are discussed in Subsection 4.5.2.

4.5 Operators

Velox query plans are composed of a tree of PlanNodes, such as Filter, Project, TableScan, Aggregation, HashJoin, Exchange, etc, that describe the computation to be performed. To execute a query plan, plan nodes are first converted into Operators. The conversion is mostly one-to-one, with a few exceptions, such as (a) Filter node followed by Project node is merged into a single FilterProject Operator, and (b) plan nodes with two or more children are converted into multiple Operators, e.g, HashJoin node is converted into a pair of operators, HashProbe and HashBuild.

The top level Velox execution concept is the *Task*, which is the unit of function shipping in distributed execution and corresponds to a query plan fragment along with its Operator tree. A Task starts with a TableScan or an Exchange (shuffle) source as input, and ends in another Exchange. The Operator tree of a Task is decomposed into one or more linear sub-trees called *Pipelines*: for instance, HashProbe and HashBuild are mapped to one Pipeline each. Each Pipeline has one or more threads of execution, called Drivers, each with its own state. Drivers can be running on a thread or not, depending on whether they have work to perform. A Driver can go off-thread for many reasons, for example, because its consumer has not consumed data yet, its source exchange hasn't produced data, or a scan is waiting for files to scan. This model is more convenient for going on or off-thread than the traditional Volcano iterator tree model [8], since the state is resumable without having to construct a control flow on the stack. Lastly, Tasks can be canceled or paused by other Velox actors at any time. Being able to pause tasks is convenient in situations such as enforcing priorities, checkpointing state, forcing another Task to spill, or other coordination activities.

All operators implement the same base API, consisting of methods such as adding a batch of vectors as input, getting a batch of vectors as output, checking if the operator is ready to accept more input data, and notifying that no more data will be added; the latter can be used, for instance, to inform a blocking sort or aggregation to flush its internal state and start producing output. Although Velox already provides an extensive set of commonly used Operators, the library also allows engine developers to add custom operators containing engine-specific business logic, such as stream-based aggregations for stream processing. Some important characteristics and optimizations present in common operators are described in the next subsections.

4.5.1 Table Scans, Filter, and Project. Table scans are done column-by-column with filter pushdown support. Columns containing filters are processed first, producing a row number for hits plus optionally the hitting value. Filters are adaptively ordered at run time, so that the filter with the minimum time to drop a value is evaluated first. The score is defined as *time / (1 + values-in - values-out)*, in such a way that the best filter drops the most values in the least amount of time. This is the same principle as in reordering conjuncts in AND/OR expressions, described in Section 4.3.

Simple filters are evaluated multiple values at a time using SIMD, which allows Velox to process roughly an integer hit per CPU clock using AVX2. Filter results for dictionary-encoded data are cached (as described in Section 4.3), and SIMD is used again to check cache hits using a *gather + compare + mask lookup + permute* to write out the passing rows, processing more than one hit per CPU clock, on

average. Velox also provides an efficient implementation for large IN filters, used for hash join pushdown, which allows it to trigger 4 cache misses at a time.

In addition, the FilterProject operator uses a single expression evaluation context for all of the filter and project expressions. For each batch of input data, the operator first evaluates the filter expression on all input rows, and only executes the project expressions on the subset of rows that passed the filter. If no rows passed the filter, the evaluation of project expressions is completely skipped.

4.5.2 Aggregate and Hash Joins. Hash joins and aggregations are the backbone of analytical data processing. Velox provides a carefully designed hash table implementation optimized for both of these use cases, which, other than promoting reusability, unifies the adaptivity in both scenarios. Hashing keys are processed in a columnar manner using an abstraction called *VectorHasher*, which recognizes the key ranges and cardinality, and where applicable translates keys to a smaller integer domain. If all keys map to a handful of integers, they are directly mapped to a flat array. If there are multiple keys, they are mapped to a single 64 bit normalized key if possible, and either used to index a flat array or used as a single hash table key depending on the range of this generated key. An inefficient multipart hash key is only used if none of the optimizations described above were possible. Moreover, the optimal hashing layout is decided adaptively, and is subject to change as new batches of data are processed. Considering that VectorHashers form a kind of digest of distinct values per key, these objects can also be pushed down to TableScans and used as efficient IN filters, in cases where the table scan and hash joins are colocated.

The hash table layout is similar to Meta's F14 [5]. Memory accesses between lookups of different keys are interleaved, with the purpose of maximizing the number of cache misses in-flight at a time, and incurring fewer and shorter pipeline stalls due to data dependency. The hash table values are stored row-wise in order to minimize cache misses, since commonly all dependent data is accessed in hash joins and aggregations.

4.6 Memory Management

Velox Tasks track memory usage via memory pools. Small objects like query plans, expression trees, and other control structures are allocated directly from the C++ heap, but larger objects such as data cache entries, hash tables for aggregates and hash joins, and other assorted buffers are allocated using a custom allocator offering zero fragmentation for large objects (using mmap and madvise), similar to work described in [11]. All memory allocations made through memory pools are tracked in a hierarchical fashion, and are subject to limit enforcement policies. Velox memory consumers can also reserve memory in order to have a guaranteed budget for completing a specific operation, such as processing a batch of group by keys.

Memory consumers may provide recovery mechanisms such as spilling for cases when memory allocations fail. To support memory recovery strategies, consumers may be asynchronously paused; when a pause is requested, the consumer will acknowledge by going off-thread and returning a continuation future that allows the engine to resume the consumer's execution at a later point in time. While in the paused state, a task may be instructed to spill to secondary storage, or be canceled in order to make space for other tasks, depending on prioritization policies.

The default action for exceeding memory limit is to invoke a process wide memory arbiter. This arbiter has visibility over all running tasks, their memory usage, and the amount of reclaimable memory, which is how much memory the Task could release in case it was instructed to spill. However, the logic for deciding which Task will be requested to spill or be canceled is pluggable, and can be provided by developers to implement engine-specific behavior.

In order to support spilling, operators need to implement an interface that communicates how much memory could be released by spilling, and the actual spilling method. If these methods are not implemented by an operator, when an allocation fails, the operator has no choice but to either continue execution without the additional allocation, or fail. Lastly, operators can choose to also monitor overall memory usage, and react based on different memory pressure scenarios; for example, the Exchange operator could decide to reduce its buffer size if memory is becoming scarce (or if some of its allocations fail).

4.6.1 Caching. For data computation systems leveraging a disaggregated storage architecture, Velox provides support for both memory and SSD caching in order to alleviate the impact of remote IO stalls to query latency. Memory caching acts as a special memory user and is allowed to consume all memory not allocated otherwise. All IO buffers are allocated directly from the memory cache and can have arbitrary sizes, according to the underlying columnar dataset's layout, unlike in Operating Systems where caches are allocated in fixed-size chunks (pages). Arbitrary allocation sizes are mixed without fragmentation by leveraging mmap/madvise [11] (as mentioned above).

Cached columns are first read from disaggregated storage systems, such as S3 or HDFS, stored in RAM for the time of first use, and eventually persisted to local SSD. Furthermore, IO reads for nearby columns are typically coalesced (merged) if the gap between them is small enough (currently about 20K for SSD and 500K for disaggregated storage), aiming to serve neighboring reads in as few IO reads as possible. Naturally, this leverages the effect of temporal locality which makes correlated columns to be cached together on SSD.

Considering that all remote columnar formats have similar access patterns, consisting of first reading file metadata to identify the buffer boundaries, followed by read of parts of these buffers, IO reads can be scheduled in advance (prefetched) in order to interleave IO stalls and CPU processing. Velox tracks access frequencies of columns on a per-query basis, and adaptively schedules prefetches for hot columns. The combination of memory caching and smart pre-fetching logic makes many SQL interactive analytical workloads, which are commonly built based on small to mid-sized tables, to be effectively served from memory, since IO stalls are taken off of the critical path and do not contribute to query latency.

Table 1 illustrates the read throughput (read latency, plus decoding and decompression) from different layers in the storage hierarchy. According to empirical data from Meta's hardware and workloads, RAM cache hits are roughly 3x faster than reads from local SSD, which are about 4x faster than remote reads from Meta's

disaggregated storage system. The data corresponds to queries executing a simple filter or aggregation on scalar columns, based on a 26-core server with 64GB of memory and 2x2TB SSD devices.

Table 1: Data rate for reading and decompressing data.

	RAM	SSD	Disaggregated
Read rate	8GB/s	2-3GB/s	700MB/s

5 EXPERIMENTAL RESULTS

In this Section, we present experimental results obtained from end-to-end tests with Prestissimo, the Velox integration with Presto described in Section 3.1, comparing the new C++ Velox-based execution engine with the current Presto Java implementation. The test platform is a cluster composed of 80 nodes with 64G RAM and 2x2TB SSD devices. Both systems have local caching enabled and run from a warm cache. The dataset is a 3TB TPC-H in ORC format with no zstd compression, and lineitem and orders co-partitioned. The query formulations are hand-written to have the right join tree shape with all the selective joins gathered on the build side, and joins are hash joins. Table 2 presents CPU and wall times for selected CPU-bound queries (Q1 and Q6), and shuffle/IO heavy queries (Q13 and Q19).

Table 2: TPC-H results comparing Prestissimo (Velox's C++ engine) vs. Presto Java engine.

	Wall time (sec)			CPU time (sec)		
	C++	Java	Speedup	C++	Java	Speedup
Q1	5	42	8.4x	2211	14435	6.5x
Q6	1	9	9x	538	2018	3.7x
Q13	15	31	2x	5647	12322	2.1x
Q19	6	13	2.1x	1362	3483	2.5x

For CPU-bound queries, Q1 and Q6, Prestissimo provides a speedup close to an order of magnitude, and is now bottlenecked on the coordinator's speed to dispatch work. For the queries that shuffle data, Q13 and Q19, the new bottleneck is shuffle latency. The possible optimizations are better metadata handling on the coordinator, better timing and message sizes on the shuffle and possibly some very lightweight encoding to cut down on the shuffle data volume.

While TPC-H is still a valid data point for system comparison, it does not provide a comprehensive representation of modern workloads. In order to evaluate Velox's performance under real workloads, we have conducted an experiment where we replay production traffic generated by a variety of interactive analytical tools found at Meta to two clusters with identical hardware characteristics (one running Prestissimo and one running Presto Java). Figure 4 shows a histogram of the relative speedup provided by Prestissimo over Presto Java - 0x means Presto Java is faster; 10x means Prestissimo is 10 or more times faster. The average speedup is about 6-7x, but many queries observe a speedup larger than an order of magnitude.

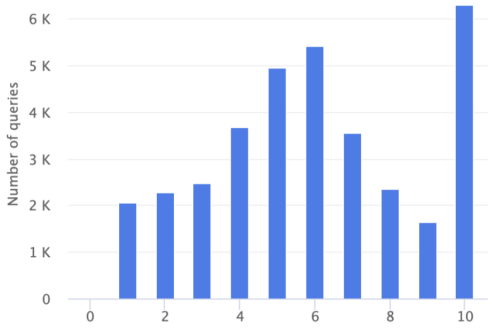

Figure 4: Prestissimo speedup over Presto Java under real interactive analytical workloads. The bars represent how many times Prestissimo is faster than Presto Java.

Lastly, in addition to the initial question of how much CPU can be saved by the new C++-based stack, a natural follow up question when dealing with hyperscale system deployments is the capacity impact of this new stack in terms of number of servers, which ultimately translates to datacenter power. In order to conduct this experiment, we created two clusters (one Prestissimo and one Presto Java) shadowing the exact same production workloads, and slowly decreased the number of servers in the Prestissimo cluster. We observed that with the Velox based stack, Prestissimo was able to support the same workloads with equal or better user-perceived performance, with **3x** fewer servers (60 vs 20).

6 FUTURE DIRECTIONS

In the last decade, the ubiquity of clouds and disaggregation of compute from storage caused a major inflection in the design of data management systems. We believe that today we are experiencing two new ongoing trends that have a similar disruptive potential: (a) the rise of AI as the principal consumer of data management, and (b) componentization and specialization of compute resources, as exemplified by GPUs, FPGAs, tensor accelerators, and cache-coherent interconnects like CXL [6].

Up until now, data computation engines were developed as monoliths containing their own language frontend, execution engine, and storage. In the future, we expect specialized processing kernels to be plugged into some kind of data management function bus, such as Velox, which can then execute query plans from multiple frontends with different execution characteristics, and fully leverage the available underlying hardware. This platform trend is already reflected in software with the advent and standardization of in memory data formats like Apache Arrow, and more recently, Substrait [19] for interoperable plan representation. We believe Velox to be an important step towards this direction.

As next steps, we expect to continue blurring the boundaries between AI and traditional data management systems by investing in stack unification and consolidation of language frontends and functions packages. We will also continue integrating Velox with

other data computation stacks within Meta, for instance, monitoring and observability systems, graphs, and operational workloads. The latter still poses substantial challenges related to vectorization, small batch sizes, and low latency requirements, which are still open questions.

Lastly, we envision a componentized world that relies less and less on a forever omniscient query optimizer, but that rather emphasizes local intelligence and adaptivity at all levels of the stack. Similarly, manual configuration of parameters such as resource allocation between different users becomes more and more laborious, error-prone, and opaque, and therefore computation systems need to be autonomous, auto-configurable, and self-driven [14]. We believe this to be the path towards a unified computation stack suited for different specialized workloads such as batch/ETL, interactive analytics, stream processing, transactional, AI/ML, and more, with one extensible open source engine.

7 RELATED WORK

DuckDB [15] is an embeddable analytical RDBMS developed as a C++ library, focused on providing fast SQL processing capabilities in a lightweight and portable way. DuckDB provides extensive data management features and is deeply integrated into the Python and R ecosystems, in addition to providing language bindings for Java, C, and C++. Although providing a vectorized engine that shares many of the same design decisions as Velox, DuckDB focuses on providing a full-stack RDBMS implementation, having a single SQL dialect as the main API with its users. Contrarily, Velox focuses on providing modular, extensible, language-agnostic, and high-performance building blocks to be integrated into existing large scale computation engines, including stream processing and realtime data infrastructure, ML platforms, and more.

The Apache Arrow project [3] provides a module containing analytical functions that process Arrow columnar data, known as "Arrow Compute" [1]. These functions (or kernels) represent computation operations over inputs of possibly varying types, and are intended for use inside query engines and data frame libraries. Despite the fact that Arrow's function API, composed of scalar, vectorized, and aggregate functions, is similar in principle to Velox's function APIs, Arrow Compute has a considerably narrower scope and does not provide other SQL operators or resource management primitives available in Velox. The Apache Arrow library also provides Gandiva [7], an LLVM-based execution environment for analytical kernels over Arrow encoded data. Despite the design differences between Arrow Compute and Gandiva (interpreted vectorized vs. just-in-time compiled), the projects are similar in terms of scope, being restricted to execution of functions/kernels.

Photon [16] is a proprietary C++ vectorized execution engine developed by Databricks which is deeply (and transparently) integrated into the Spark ecosystem and targets speedup of Spark queries. When Photon is enabled, the Spark runtime takes another pass over the optimized query plan, determining which parts of that plan can be run in Photon. The Photon library is then loaded into the JVM to execute these fragments, leveraging JNI for communication and exchange of off-heap data pointers. Although sharing similar design decisions and optimizations with Velox, Photon is solely focused on accelerating Spark workloads, in addition to being proprietary. Velox, in its turn, is engine- and dialect-agnostic and developed in partnership with the open source community.

Lastly, Optimized Analytics Package [13] (OAP) is an open source project driven by Intel, also aimed at optimizing Spark. OAP contains a plugin called Gazelle [12], focused on providing SIMD-optimized execution kernels and an LLVM-based expression engine to accelerate Spark queries. Similarly to Photon, JNI is used to communicate and exchange data using the Apache Arrow layout, leveraging Arrow Compute, Gandiva, and custom operators for vectorized kernel execution. Despite following similar design decisions, Gazelle is focused on Spark workloads and not targeted to engine-agnostic usage in other domains.

8 CONCLUSION

The fast proliferation of specialized data computation engines targeted to very specific types of workloads has created a siloed data ecosystem. These engines usually share little to nothing with each other and are hard to maintain, evolve, and optimize, and ultimately provide an inconsistent experience to data users. In this paper we presented Velox, a novel open source C++ database acceleration library which provides reusable, extensible, high-performance, and dialect-agnostic data processing components that are being used to unify existing computation engines at Meta.

Velox demonstrates that it is possible to converge existing computation engines into a best-of-breed query execution component, considering it is being integrated with more than a dozen data systems at Meta, including not only analytical engines such as Presto and Spark, but also stream processing platforms, message buses and data warehouse ingestion infrastructure, ML systems for data preprocessing and feature engineering, and more. Velox provides benefits in terms of (a) efficiency wins, by democratizing optimizations previously only found in individual engines, (b) increased consistency for data users, and (c) engineering efficiency by promoting reusability and eliminating duplicated efforts.

As developers of data infrastructure technology, we see Velox as the crystallization of the experience from some of the +20 systems the authors have worked on. As future steps, we are exploring unifying operational and analytical systems through Velox, integration with graph and monitoring engines, as well as further convergence with ML platforms. We are also exploring new areas where adaptivity can be leveraged, in addition to further componentization and hardware specialization via efficient and reusable kernels. Finally, we hope Velox can be used as an open compute platform where developers and users alike can experiment and direct advances, providing an open laboratory for the next generation.

ACKNOWLEDGMENTS

The work presented in this paper was only possible due to the extensive contributions from the Velox community. A special thank you to Deepak Majeti, Aditi Pandit, and Ying Su from Ahana, Frank Hu from ByteDance, and other Meta contributors such as Sagar Mittal, Sergey Pershin, Jialiang Tan, Zhenyuan Zhao, Behnam Robatmili, Jimmy Lu, Wenlei Xu, and many others, in addition to the support and sponsorship from Naveen Cherukuri, Sridhar Anumandla, Jiju John, Victoria Dudin, and Jana van Greunen.

REFERENCES

[1] Apache Arrow. [n.d.]. Apache Arrow C++ Compute Functions. https://github.com/apache/arrow/tree/master/cpp/src/arrow/compute. Accessed: 2022-02-23.

[2] Apache Arrow. [n.d.]. Arrow Columnar Format. https://arrow.apache.org/docs/format/Columnar.html. Accessed: 2022-02-23.

[3] Apache Arrow. [n.d.]. A cross-language development platform for in-memory analytics. https://arrow.apache.org/. Accessed: 2022-02-23.

[4] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *Conference on Innovative Data Systems Research, CIDR*.

[5] Nathan Bronson and Xiao Shi. [n.d.]. Open-sourcing F14 for faster, more memory-efficient hash tables. https://engineering.fb.com/2019/04/25/developer-tools/f14/. Accessed: 2022-02-23.

[6] The CXL Consortium. [n.d.]. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/. Accessed: 2022-02-23.

[7] Dremio. [n.d.]. Introducing the Gandiva Initiative for Apache Arrow. https://www.dremio.com/announcing-gandiva-initiative-for-apache-arrow. Accessed: 2022-02-23.

[8] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

[9] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*.

[10] Wes McKinney. [n.d.]. Adding new columnar memory layouts to Arrow. https://lists.apache.org/thread/49qzofswg1r5z7zh39pjvd1m2ggz2kdq. Accessed: 2022-02-23.

[11] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020*. www.cidrdb.org.

[12] OAP. [n.d.]. Gazelle Plugin - A Native Engine for Spark SQL with vectorized SIMD optimizations. https://oap-project.github.io/gazelle_plugin/latest/. Accessed: 2022-02-23.

[13] OAP. [n.d.]. Optimized Analytics Package. https://oap-project.github.io/latest/. Accessed: 2022-02-23.

[14] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[15] 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.

[16] Greg Rahn, Alexander Behm, and Ala Luszczak. [n.d.]. Photon: The next-generation query engine for the lakehouse. https://databricks.com/product/photon. Accessed: 2022-02-23.

[17] 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.

[18] Apache Spark. [n.d.]. Apache Spark - Unified Engine for large-scale data analytics. https://spark.apache.org/. Accessed: 2022-02-23.

[19] Substrait. [n.d.]. Cross-Language Serialization for Relational Algebra. https://substrait.io/. Accessed: 2022-02-23.

[20] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training. arXiv:2108.09373 [cs.DC]