From Presto to Prestissimo: A Velox-Powered Modernization Journey

Amit Dutta, Pedro Pedreira, Xiaoxuan Meng, Masha Basmanova, Orri Erling, Sergey Pershin, Jialiang Tan, Ge Gao, Ke Wang, Zac Wen, Krishna Pai, Xiao Du, Natasha Sehgal, Heidi Han, Artem Selishchev, Ke Wang, Sebastiano Peluso, Chandrashekhar Singh, Sreeni Viswanandha, Aditi Pandit[†], Deepak Majeti[†], Tim Meehan[†], Christian Zentgraf[†], Aakash Deep, Rohit Jain, Sridhar Anumandla, Kaushik Ravichandran, and Vivek Gaur

Meta Platforms, Inc. [†]IBM, Inc.

ABSTRACT

Presto is a widely adopted, open-source distributed SQL query engine that has powered Meta's exabyte-scale data analytics for over a decade. Over time, as data volumes and workload diversity consistently increased, the original Java-based Presto worker fleet started facing challenges in reliability, performance, and operational efficiency, despite continuous upgrades and enhancements. To overcome these challenges, we created Prestissimo (open-sourced under Presto), a more modern version of Presto, re-engineered to use the Velox open-source composable library; Prestissimo provides a drop-in C++ worker replacement, delivering substantial improvements in query latency, hardware efficiency, and reliability. In this paper, we outline the phased, in-place migration journey that resulted in a full and seamless transition of hundreds of millions of daily large-scale analytic queries, without service interruption or additional hardware. To ensure correctness and reliability, we developed a comprehensive suite of verification tools and testing frameworks, including randomized fuzzers, end-to-end verifiers, and production traffic shadowing. We discuss the challenges of achieving feature parity at scale across different programming languages, how we handled backward compatibility, how pipeline migration was prioritized, and how operational excellence was achieved in a multi-tenant, business-critical environment. The complete Prestissimo migration took about two years, alongside substantial development in both Prestissimo and Velox. With this journey, we have fully decommissioned the legacy Java worker fleet, achieving broad gains including 2–5x faster queries while at the same time cutting down the hardware footprint significantly, and reducing the number of reliability issues in production by an order of magnitude. Prestissimo is today fully rolled out at Meta, and has replaced Presto Java workers in many other deployments across the industry.

PVLDB Reference Format:
Amit Dutta, Pedro Pedreira, Xiaoxuan Meng, Masha Basmanova, Orri Erling, Sergey Pershin, Jialiang Tan, Ge Gao, Ke Wang, Zac Wen, Krishna Pai, Xiao Du, Natasha Sehgal, Heidi Han, Artem Selishchev, Ke Wang, Sebastiano Peluso, Chandrashekhar Singh, Sreeni Viswanandha, Aditi Pandit, Deepak Majeti, Tim Meehan, Christian Zentgraf, Aakash Deep, Rohit Jain, Sridhar Anumandla, Kaushik Ravichandran, and Vivek Gaur. From Presto to Prestissimo: A Velox-Powered Modernization Journey. PVLDB, 19(12): 4399 - 4412, 2026.
doi:10.14778/3827998.3828041

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/prestodb/presto.

1 INTRODUCTION

Presto [58] is an open-source, distributed SQL query processing engine designed to execute data analytics workloads ranging from a few seconds to several hours over exabyte-scale data sources. Since 2013, Presto has supported a decade's worth of SQL analytics workloads at Meta (and elsewhere), enabling rapid extraction of insights from a diverse array of large, distributed data sources. After its donation to the Linux Foundation in 2019, Presto and its open-source ecosystem continued to evolve with contributions from various technology companies; its ease of use and large set of features make it a suitable choice for a wide range of users and applications, establishing Presto as a foundational technology for many large-scale lakehouses.

At Meta's lakehouse, Presto serves as the main SQL analytics engine, serving hundreds of millions of queries in a few dozen regions distributed across the globe, and scanning many exabytes of data each day, making it a critical component of the entire lakehouse ecosystem. It powers a variety of workloads, including traditional interactive SQL workloads for dashboard generation, data exploration, and a variety of other latency-sensitive products and platforms; it also supports lightweight batch pipelines (ETL), outperforming traditional large-scale batch engines such as Spark [65] in terms of performance and efficiency for queries that take up to a few hours to execute.

Despite continuous upgrades and improvements over the years [60], Presto's Java-based query execution engine started to show signs of obsolescence. While lakehouse workloads continued to grow in terms of data and query volume, operational instabilities in the Presto worker fleet caused by poor memory management and Java Virtual Machine (JVM) garbage collection pauses, resource overhead, and challenges to fine-tune JVMs at web-scale for heterogeneous workloads resulted in Presto struggling to meet service-level objectives (SLOs) and deliver operational excellence.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828041

It became increasingly problematic to support large-scale Java deployments, as infrastructure and its supporting libraries are predominantly written in C++ at Meta. At the same time, rapidly growing AI/ML and analytics workloads, projected to increase query volume by orders of magnitude, required Presto infrastructure to deliver a step-function change in execution efficiency while reducing its capacity footprint.

To address these challenges, we created Velox [52], a novel state-of-the-art vectorized execution engine. To avoid the pitfalls of operating Java deployments at scale, we wrote Velox from the ground up in C++, implementing modern vectorization and compressed execution techniques [54], most of which were not available in Presto Java's counterpart implementation. Because similar headwinds were observed in other internal deployments of large-scale data systems such as Spark, and in internal engines used for stream processing, real-time infrastructure, log analytics, training data loading [66], and a variety of others, we decided to implement the Velox library in a multi-dialect and reusable (or *composable* [31, 53]) manner. Because many other large data organizations were interested in collaborating in similar efforts, we open-sourced Velox in 2022 and observed quick community growth and widespread adoption ever since.

Prestissimo[1] is the name of our next generation of Presto, re-engineered to use Velox. While maintaining Presto's coordinator in its original form, Prestissimo provides a drop-in C++ worker replacement based on the Velox library, completely removing the need for JVM in Presto's worker fleet, which comprises the bulk of the capacity. The goals behind the Prestissimo project were ambitious: we aimed to deliver faster query performance, improved resource efficiency (in fact reducing the hardware footprint), and enhanced reliability by removing the need for JVM and thereby increasing our flexibility in managing resources. In addition to that, we focused on achieving our goals in a composable manner, such that the artifacts could be reused and applied to similar efforts in different data systems, preventing teams from reinventing the wheel and improving overall engineering efficiency.

In this paper, we present our journey of incrementally replacing the Presto Java engine with Prestissimo in one of the largest lakehouses in the world. During the migration, we purposely decided not to support a hybrid architecture, which vastly simplified the system and proved to be the right strategic direction in the long run, but created challenges in the initial migration as only entire queries could be migrated to the new Prestissimo stack. We describe our experience during this journey, discussing the methodology, how the migration of workloads was prioritized, how backward compatibility was handled, lessons learned through this process, and detail the extensive set of tooling and testing, validation, and verification frameworks built to reliably support the process and minimize potential disruptions.

At Meta, we executed the Prestissimo migration over two years alongside substantial development in both Prestissimo and Velox, performing the rollout in-place (without adding new hardware) and progressively scaling down production clusters as Prestissimo enabled higher throughput on fewer machines; by December 2025, we fully decommissioned the legacy Java workers, achieving broad gains including ~2–5x faster queries, lower hardware usage, significantly reduced operating cost, and markedly improved reliability and maintainability — resulting in fewer issues and production interruptions. Prestissimo is fully open source and available as part of the Presto repository [17] under the Apache 2.0 license.

The rest of the paper is organized as follows. Section 2 provides an overview of the entire Presto ecosystem at Meta. Section 3 introduces the Prestissimo architecture, detailing major modifications such as memory management, and integration with Velox. Section 4 discusses the deployment and rollout strategy. Section 5 presents production experience and Section 6 shares lessons learned throughout the migration journey. Section 7 reviews related work, and Section 8 concludes the paper with a discussion of future work and potential extensions for Prestissimo.

2 SYSTEM OVERVIEW

A Presto cluster is set up with a single coordinator and a large pool of worker nodes. The coordinator is responsible for taking a SQL string from the client as input, parsing it into a logical plan, optimizing the logical plan and creating physical plan fragments, and finally assigning fragments to workers and orchestrating the distributed execution. The workers execute plans and shuffle data directly between each other using a streaming shuffle strategy, without persisting the shuffled data or intermediate query results. As a result, many query stages can be under execution at a time, and there is no support for query restartability; if any of the workers fail, the entire query fails and is retried. This architecture optimizes for reasonably short-lived queries that run from a few seconds up to an hour, focusing on low-latency interactive results and small to mid-sized ETL pipelines.

The coordinator is also responsible for scheduling and worker pool management. It tracks worker resource usage to route plan fragments appropriately, typically managing up to 1000 workers per cluster; beyond that, the system scales horizontally via multiple clusters with load-balanced query routing and cross-cluster retry for fault tolerance.

Presto has two major deployment modes at Meta: interactive and batch. The common practice is to deploy clusters that are built for one of these use cases, and route user queries accordingly. In both flavors, the same Presto software is deployed with different configurations. Presto interactive optimizes for latency, for workloads where a user is waiting for the query to complete and return results, e.g., when generating dashboards or supporting interactive data exploration. The average runtime for interactive use cases is from a few seconds to a few minutes. Presto interactive is usually deployed in hosts with local SSDs which are used for local caching (along with additional memory caching), and query fragments are assigned to worker hosts with soft affinity to ensure adequate cache hit rates.

Presto batch clusters run periodic workloads scheduled by data pipeline management infrastructure. Presto batch queries can typically span from minutes up to a few hours. Batch workloads scan and process massive datasets, and are substantial resource consumers in lakehouse deployments. Batch workloads are generally retryable upon failure, up to a certain number of times, and customers are presented with the final failure if all tries are exhausted.

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

Apart from general-purpose batch and interactive clusters, it is also common for organizations to deploy special clusters dedicated to a particular use case or customer. Such use cases normally come from a specific platform that generates predictable SQL input, making queries fixed-shape and cache-friendly.

While Presto batch deployments can support mid-sized batch queries, longer queries have a higher chance of being impacted by hardware failures, unscheduled maintenance, or other non-deterministic production issues. Additionally, queries requiring processing of hundreds of TBs or even PBs can't be handled by batch clusters due to the limited number of workers and the streaming nature of Presto batch requiring data to be held in workers' memory. To address this, Presto-on-Spark [21] executes the same Presto queries on Spark's runtime, mapping query fragments to RDD operations with persisted shuffles that enable task-level restartability. This supports queries from minutes up to 72 hours using the same Presto software with different configurations.

2.1 Production Deployments

Meta's data infrastructure spans a few dozen regions across the globe [16]. Each lakehouse region includes Meta's distributed storage, called *Tectonic* [50], a metadata service instance, multiple Presto clusters for load balancing and high availability, and a variety of supporting services. Each Presto cluster typically comprises a few hundred workers. For accounting purposes, storage is segmented into logical partitions (called namespaces), which can be spread across multiple regions.

Presto clusters are always co-located with storage. That is leveraged by an internal regional load & data balancing system, which ensures that compute and data are placed as close as possible, with the goal of minimizing cross-region traffic and maximizing compute utilization. All clusters are registered to a Presto gateway service, which is responsible for routing queries from a client to a specific Presto cluster. A Presto gateway service receives queries from clients, determines the target region by resolving partition locations via an I/O router, and performs cluster-level load balancing to forward queries to a suitable cluster.

3 PRESTISSIMO

Prestissimo is a re-engineered version of Presto that uses Velox for query execution. Prestissimo's architecture (shown in Figure 1) maintains the same unchanged Presto coordinator (written in Java) but transparently replaces Java workers with native C++ workers powered by the Velox library. C++ was chosen for three reasons: (1) Meta's infrastructure stack is predominantly C++ and includes widely adopted open-source libraries such as Proxygen [59] (HTTP framework), fbthrift [7] (RPC and serialization), and Folly [8] (foundational C++ utilities), enabling Prestissimo to build on battle-tested foundations rather than reimplementing them; (2) a vectorized execution engine demands fine-grained control over memory layout and allocation to maximize cache locality, which C++ provides; and (3) Velox was designed as a composable library for multiple existing C++ data systems at Meta [31] (stream processing, real-time analytics, training data loading), making native C++ integration essential.

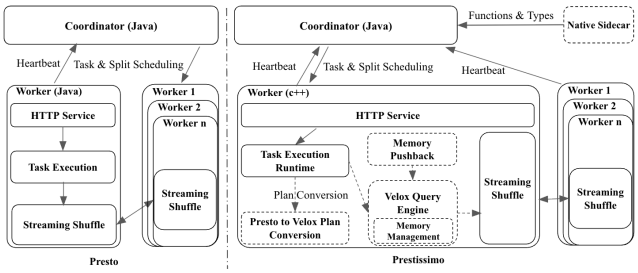

Figure 1: Presto (left) vs. Prestissimo (right) architecture. The coordinator remains unchanged; the worker is fully replaced with a C++ implementation powered by Velox. New components are shown with dotted lines.

To achieve a true *drop-in* replacement, Prestissimo strictly adheres to Presto's existing network protocols. Prestissimo uses Proxygen to implement the coordinator and worker HTTP/REST interface and internally delegates physical plan execution to Velox, enabling the underlying execution engine to be swapped entirely without altering the control plane. During multi-stage query execution, data is shuffled between workers using Presto's columnar serialization protocol, called PrestoPage [38], which is also implemented and available in Velox. This compatibility extends to data communication with the Java coordinator, as the last stage of execution is performed in the coordinator itself before data can be returned to the client. For wide shuffles, Prestissimo uses *CompactRow* [23], a row-oriented format that reduces memory and network overhead for C++ worker-to-worker communication.

Furthermore, the Prestissimo architecture provides a set of features that enable it to be seamlessly integrated into the Presto ecosystem. First, to allow for full query coverage, Prestissimo reimplements Presto's extensive library containing hundreds of scalar, aggregate, and window built-in functions [56], rigorously testing the C++ implementations against their Java counterparts using a comprehensive suite of data-driven tests (described in Section 4). Second, Prestissimo reimplements in C++ the connectors that are most frequently used (such as the Hive and Iceberg connectors), and provides a remote connector adapter based on Arrow Flight to support the long tail of existing Java connectors. Third, a C++ sidecar component [37] runs alongside the Java coordinator to support query-time operations (function discovery, constant folding) that require C++ execution code.

Lastly, because Velox is designed as a composable, engine-agnostic execution library that is entirely decoupled from Presto's internal details and data structures, Prestissimo needs to provide a software layer to bridge the two software stacks. When a C++ worker receives a Presto query plan, Prestissimo converts it into a Velox query plan. This translation is not a strict one-to-one node mapping; a single Presto plan node is sometimes expanded into multiple Velox physical operators, or vice versa. This clean architectural boundary ensures Velox remains a versatile, standalone library while Prestissimo encapsulates all Presto-specific domain adaptations.

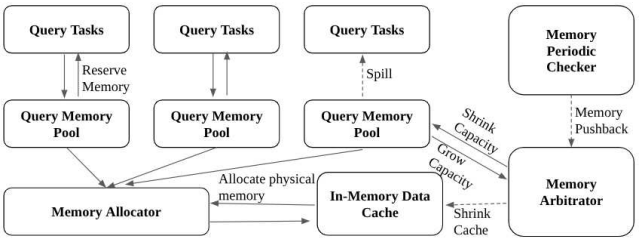

Figure 2: Memory Management

3.1 Memory Management

Prestissimo relies on Velox's hierarchical memory management [43] and spilling [44] subsystem for predictable, high-performance execution. Figure 2 illustrates different components of the memory management system.

The memory management system allocates a root memory pool for the worker node, and hierarchically spawns child pools for individual queries, tasks, and operators. This tree-like structure allows the system to enforce granular memory limits and track usage at every level of execution. When memory limits are reached, Prestissimo utilizes a global *memory arbitrator*; instead of abruptly failing a query (OOM), the arbitrator dynamically pauses tasks, and either reclaims memory from other queries (based on priorities or some other policy), or forces memory-intensive operators (like hash joins and aggregations) to spill their intermediate states to disk, either locally or remotely. This localized spilling and dynamic arbitration ensure high query success rates even under heavy concurrency or resource contention.

Beyond query-level arbitration, Prestissimo employs a memory pushback mechanism to prevent system-level out-of-memory failures. A background thread periodically monitors the worker's overall memory usage, including pressure from non-Velox components such as the HTTP server and storage clients, and triggers corrective actions when a configurable threshold is exceeded. Depending on the workload type, pushback may shrink the in-memory data cache, abort lower-priority queries to reclaim their memory, or force the allocator to release fragmented, unused pages back to the OS. This reactive, system-wide safety valve complements the query-level arbitrator: while arbitration redistributes memory among queries, pushback defends against aggregate pressure that no single query controls, substantially reducing out-of-memory crashes in production. Both mechanisms were co-designed with Prestissimo's requirements (high concurrency, unpredictable memory profiles from user-defined functions, and mixed workloads sharing the same cluster), then generalized within Velox for reuse across other engines (e.g., Spark via Gluten [10]), avoiding duplication of memory management logic.

Prestissimo also provides an advanced local caching layer, which is only enabled for interactive workload clusters. Velox's SSD cache is utilized to store frequently accessed file segments (such as Parquet/ORC/Nimble footers or row groups/stripes) directly on the worker nodes; recently used blocks are also cached in memory, filling up any spare memory capacity not currently used by queries or operators.

Prestissimo heavily relies on buffer reuse. Vectors and memory blocks are recycled across operator invocations rather than being continually allocated and freed. Velox (and consequently Prestissimo) also makes extensive use of memory arenas, particularly for operations involving millions of small, variable-length allocations, such as hash table construction for joins and aggregations. These allocators request large, contiguous memory blocks from the system and sub-allocate them for individual strings or structures, bypassing system-level allocation overhead, and allowing for O(1) memory reclamation when operators are destroyed. Lastly, for the long tail of small, short-lived, and untracked allocations used for auxiliary and ephemeral containers, Prestissimo relies on the jemalloc [13] allocator to leverage thread-caching and size-class segregation, dramatically reducing fragmentation and overhead.

4 DEPLOYMENT AND ROLLOUT

The parallel development and rollout of Prestissimo and Velox at Meta was a major engineering effort. This process required numerous ad-hoc, sophisticated, and creative techniques to meet the goal of a full transition to Prestissimo, while maintaining the integrity and reliability of Meta's analytics infrastructure. The risks associated with data corruption, privacy and security vulnerabilities, or execution inefficiencies were substantial, as any such issues could result in company-wide critical incidents. Furthermore, the transition was constrained by the requirement to avoid duplicating infrastructure capacity, maintain existing service guarantees, and deliver improved performance and cost efficiency. In the remainder of this section, we outline the key restrictions and challenges encountered in this process, and the following sections detail these challenges and the strategies used to address them.

No Hybrid Execution. Prestissimo replaces Java workers entirely with C++ workers, preventing mixed Java/C++ query execution; reaching feature parity with the mature Java engine therefore required substantial time and effort.

In-Place Rollout. The migration proceeded without transitional hardware, prioritizing early efficiency gains via a phased rollout starting with pilot workloads.

Correctness. Replacing Meta's long-standing authoritative analytics engine at massive daily query volume demanded extensive, sophisticated testing to avoid pipeline breakages or data corruption.

Backward Compatibility. While result parity with Presto Java was a core goal, strict compatibility was not always feasible; selective, intentional divergences were accepted to preserve performance and architectural simplicity.

Reliability and Stability. Prestissimo needed to meet or exceed the proven reliability of Presto Java at Meta's scale.

Team/People Management. The shift to a Velox-based C++ worker engine required deliberate knowledge-transfer and ownership measures to ensure the Presto team retained agency to evolve the full stack.

4.1 No Hybrid Execution

The Prestissimo worker was developed as a drop-in C++-based replacement for the Presto Java worker, with no Java components included in its architecture. During the design phase, a hybrid approach where queries could be partially executed in Java and partially in C++ was considered. That could have enabled a more incremental rollout, as queries could be partially migrated across engines, one feature or operator at a time. However, it would have complicated the architecture in a few different ways. First, it would have introduced operational complexities due to JNI (or some other form of inter-process communication), making memory management and, more generally, resource management more challenging, as it would need to be coordinated across the JNI boundary, or between processes. Second, it would have required our team to continue maintaining a JVM process, which had historically been a source of reliability and operational issues due to unpredictable garbage collection runs, and maintenance of an additional set of binaries and supporting libraries. Third, we also considered that a fallback approach could prove too convenient, and fail to provide the forcing function our team needed in the long run to move to a cleaner architecture; i.e., we were concerned that the hybrid architecture could have unintentionally become the final architecture, as people and priorities oscillate over time.

A second architecture we considered was extending Presto's flat worker model into a two-tiered architecture, where a set of C++ hosts would be able to execute simple (but common) operations like table scans and filters, and the remaining operations (the long tail) would still rely on a separate tier of hosts running the legacy Java stack. We could have then incrementally shifted capacity from one tier to the other as more features were available in C++, until eventually decommissioning the Java tier. However, this would have required us to extend the query optimizer to split queries into additional stages, adding more complexity and potential bugs into the system. It would also make provisioning capacity more challenging as workers would run mutually exclusive worker tasks. We were also concerned that the additional query stage (and required shuffles) could add overhead to queries, negating the performance benefits obtained. After design deliberations, we decided to stay away from this design to avoid adding more complexity, even if temporary, into the system.

We eventually decided to pursue a full worker replacement architecture, where no JVM would be available in Prestissimo worker hosts. While this provided a cleaner and simpler architecture in the long run, it made the deployment process and rollouts more challenging, since only queries that were fully supported by the new stack could be executed in Prestissimo. Presto supports a broad array of features, including hundreds of scalar user-defined functions (UDFs), dozens of user-defined aggregate functions (UDAFs), connectors to various storage systems, geospatial capabilities, diverse SQL constructs, custom data types, and a series of other Meta-specific extensions. Efficiently supporting this large array of features in the new C++ stack would have taken years, so we focused on designing a deployment plan that could enable us to add value incrementally to specific workloads, while still resulting in a final clean architecture. We describe the rollout strategy in the next section.

4.2 In-Place Rollout

Given the scale of Meta's Presto deployment, it was not feasible to simply provision the parallel capacity required by new Prestissimo clusters. At the same time, Java worker capacity could not be decommissioned as it actively supported production workloads. With these hard restrictions, we had to design a rollout process that could be executed in-place, or cluster by cluster, without relying on additional resources. We designed a rollout strategy that was structured into multiple phases, each targeting an incremental portion of production workloads. During the migration, Prestissimo C++ clusters did not initially support the full range of features, so we also had to design intelligent routing mechanisms that could forward user queries to either C++ or Java clusters, depending on a series of factors. We started with very restrictive sets of rules, ensuring that workloads and their required features were validated beforehand to guarantee that they produced correct results, and incrementally relaxing restriction and routing rules as the C++ stack became more mature. As we gained more confidence in the quality and reliability of the new stack, we started incrementally shifting capacity from Java to C++ clusters. Throughout the process, we expected the overall worker pool footprint to decrease, to demonstrate the superior efficiency of the Prestissimo stack. At the end, the freed capacity would either be returned to other capacity pools or used to support organic growth. The rollout strategy was designed in the following phases: (1) Fixed-Shape Workload, (2) General Purpose (verified-only), (3) General Purpose (with block-list), and (4) General Availability, discussed in the next subsections.

4.2.1 Fixed-Shape Workload. Fixed-shape workloads were query patterns automatically generated and issued by platforms built on top of Presto. These workloads were mostly static, or only contained minor changes over time, so they were good candidates for initial Prestissimo rollouts as not every Presto feature had to be implemented to be able to support them. Nevertheless, they were still important production workloads that required a solid and stable foundation for the small set of query features required. As described in Section 2, there are multiple performance-intensive fixed-shape workload runs in Presto at Meta. Particularly important examples are the A/B testing and experimentation platforms that use Presto to perform various complex statistical analyses and reporting in an interactive manner. The workload issued by these platforms included only a subset of query features, like subselects, inner joins, and statistical aggregates, so it enabled the team to initially focus on stabilizing only the required features. Most platform teams are limited by the amount of Presto capacity available, so moving to a more efficient Presto cluster was attractive to enable A/B testing platforms to scale their workload and support larger and more numerous experiments. Because of their importance and query shape predictability, they became our first Prestissimo migration target.

To deliver on the project promises, we had to ensure not only that queries executed without correctness issues—matching the exact results returned by existing Java clusters—but also that they did so with better efficiency and reliability. Since the query shapes were fixed, it was possible to verify every query shape for a correctness match—more details on query verification strategies are presented in Section 4.3. The migration happened incrementally, one cluster and one region at a time, slowly reducing the hardware footprint as

the Prestissimo clusters supported the same SLOs, but with fewer hosts. Eventually, the extra hardware freed was utilized to support platform organic growth. Because during the migration both Java and C++ clusters were available for queries, it was crucial to ensure query results were identical, and a substantial amount of tooling and testing frameworks were built to validate queries across C++ and Java stacks.

4.2.2 General Purpose (verified-only). As a next step, we focused on more general query shapes, specifically targeting ETL pipelines from Presto batch clusters. Besides increased workload diversity, batch pipelines have the common characteristic of outputting results to a destination table (instead of returning them to humans and dashboards), which makes them more challenging to verify. This was the first step to make Prestissimo available for general rollout, so we were extra careful about data correctness, reliability, and performance. We conducted several experiments with shadow tooling (discussed in Section 4.5) to ensure a Prestissimo cluster could operate with lower capacity in worker pool than a Java cluster, while still providing better performance. Along with performance enhancements, we also added critical features like TableWriters, type system extensions such as timestamps and timezone variations, date/time-specific functions, and a series of Meta-internal connectors to other data sources, which helped narrow the feature gap between Prestissimo and Presto Java.

To ensure a safe rollout, we initially selected a subset of production pipelines and verified that they produced correct results in the new Prestissimo clusters. The methodology was to ensure that the pipelines shadowed in Prestissimo clusters would produce output tables identical to those of their production counterparts running on the existing Java stack. Naturally, this was only incrementally done for a subset of pipelines, as verifying every query would have required test clusters to be provisioned with capacity equivalent to that of the actual production clusters, which was impractical.

To make sure we continually increased coverage in the pipelines validated in the new stack, we built verification and routing mechanisms that allowed us to control whether to validate (or re-validate) pipelines that had been tested before. For each Prestissimo build, we collected verification results and logged the MD5 hash of successful queries to a database. Then, during query dispatching (shown in Figure 3), in the client, we computed the MD5 of the incoming query and searched for a match in the database. If a match was found, the *"NATIVE_ELIGIBLE"* tag (a piece of metadata) was added to the query. When the Presto gateway detected the tag, it would prefer routing it to a Prestissimo cluster over a Java cluster; note that queries that were tagged this way could still be forwarded to a Java cluster, depending on the load and amount of capacity available for Prestissimo clusters. This gave us fine-grained levers to control workloads and their destination, in addition to providing a kill switch mechanism that allowed our team to revert traffic (on a per-query basis) to Java clusters in case production issues were found. Before computing the MD5 hash, we replaced parameter values of the queries using constants to ensure that different instances of the same query were still eligible. For example, a verified query such as *"select ds from table where ds = '2026-01-12'"*, would have been modified to *"select ds from table where ds = '<DATEID>'"*. So if during the production run the same query is found with a different

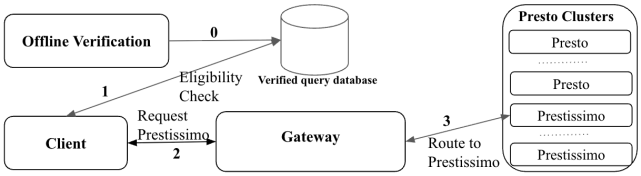

Figure 3: General Purpose (verified-only)

parameter value, say *'2026-01-13'*, the routing system would still mark it as eligible for Prestissimo. Note that, in addition to storing the MD5 hash, the system would also store the build version and commit where the query was verified, such that queries were only eligible to that build. This was done initially as a safety measure to reduce chances of data corruption, due to the massive amount of development happening at the time from both internal teams and the external open-source community. The strategy of verifying each build proved to be successful and ensured a smooth rollout for a small number of clusters, allowing Prestissimo to run equivalent workloads compared to its Java counterpart, and establishing the first general-purpose production clusters.

Verifying queries for each build enabled the replacement of a few production clusters (with lower capacity). However, this process was not scalable as the same queries were repeatedly verified for different builds, resulting in a limited set of verified queries and constraining the expansion of Prestissimo to only a small fraction of total queries. With increased confidence in Prestissimo and enhancements to testing frameworks (discussed in Section 4.3), we developed a historical verified query database independent of build versions. This database has grown steadily through daily testing with new builds and queries, significantly increasing the proportion of production queries executed by Prestissimo.

4.2.3 General Purpose (with block-list). Historical verification provided a starting point that enabled the rollout of general-purpose clusters, but it wasn't a scalable approach due to the massive volume of queries. At this phase, as we gained confidence in the Prestissimo stack, we switched the rollout strategy from verifying each pipeline individually, to enabling them by default but actively blocking specific features. To achieve this, we created and maintained a block-list based on function names, connectors, plan nodes, or any other characteristics that could be extracted from the optimized query plan. We ensured that only query plans that passed the block-list could run in Prestissimo, and aimed to slowly remove block-list rules as features became stable, in order to completely replace all Presto Java clusters. Therefore, a more robust routing mechanism was necessary to remove the dependency on client-side changes, as such changes are harder to roll out in a controlled manner, and client-side logic is hard to replicate across all clients written in different languages such as C++, Python, and PHP.

A query plan validation mechanism was introduced in the Presto gateway, leveraging the capabilities supported by Prestissimo. At this stage of migration, the objective was to establish the following invariant: Prestissimo should be able to execute all queries, except for a defined subset of features that are either unimplemented or currently under development. Once these features reached a stable

state and got removed from the block-list, Prestissimo would have been mature enough to fully replace all existing Java-based clusters.

Figure 4 provides an overview of this routing mechanism. In the previous routing approach, the addition of the "NATIVE_ELIGIBLE" tag was performed on the client side. This limited the functionality to a small number of clients capable of consulting a verified query database. To enable a broader rollout, this process was enhanced such that the gateway itself was now responsible for appending the tag and making routing decisions. The revised procedure was as follows:

Query Submission and Plan Validation. The client submits a query to the Presto gateway. The gateway communicates with explain clusters, which are equipped with Prestissimo plan checkers provided by Native Sidecar. These checkers generate a logical plan from the query text and consult a configuration file to determine if the plan contains any block-listed elements, such as specific function names, connectors, or catalogs, which were known to be either not implemented or still not thoroughly verified. If the query utilized any unsupported feature as specified in the configuration, the plan checker returned a response to the gateway containing a list of unsupported items it found in the plan.

Routing Decision. The gateway evaluated the results from the plan checker and consulted its own routing configuration to determine whether to append the *"NATIVE_ELIGIBLE"* tag. The gateway routing configuration also served as a secondary blocking mechanism. In certain scenarios, users or the support team may require that a query always be executed on either Presto Java or Prestissimo to address urgent operational needs, as without the override an eligible query could be routed to any cluster. For example, we have used this mechanism to ensure that queries using geospatial [9] capabilities were always forwarded to Java clusters, since these were only implemented later in the Velox stack. The config also supported queries where customers (specifically early adopters) can force their traffic to Prestissimo for performance improvements.

Dynamic Configuration. Both the Prestissimo capabilities configuration and the gateway routing configuration are hot-reloadable, enabling the Presto team to enable or disable features dynamically. The Prestissimo routing configuration was initially constructed to reflect capabilities not yet supported in Velox. As new capabilities are introduced in each release, the configuration is updated accordingly to enable routing of additional queries to Prestissimo. This approach ensured a robust and flexible migration path, allowing for incremental adoption of Prestissimo while maintaining operational stability and configurability. Traffic was slowly enabled in Prestissimo, as features became available and well-tested, allowing the team to incrementally shift worker pool capacity from the legacy stack, by decommissioning existing Java clusters, to the new C++ stack, by adding new clusters to Prestissimo pool using that capacity.

4.3 Correctness

Verifying that Prestissimo produced identical results to Presto Java was crucial to ensure correctness and continuity of production data pipelines. A wide array of features and complex SQL queries needed to be validated against the existing Java implementation. In many

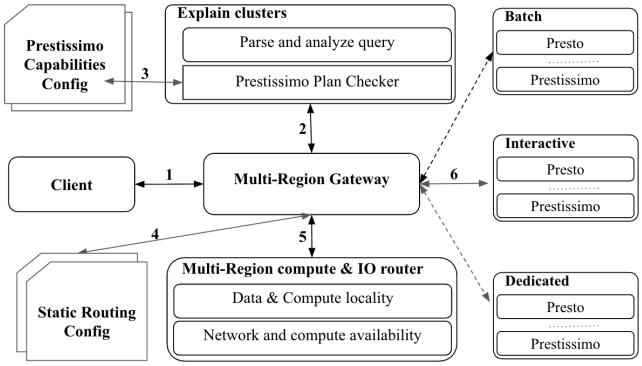

Figure 4: General Purpose (with block-list)

cases, Velox used more modern and efficient libraries for faster computation (for example, for regular expression, JSON parsing, or date/time implementation), and these at times did not match their existing Java counterparts. Meanwhile, running every single pipeline to ensure consistency between both stacks wasn't an option, as the volume and shapes of queries evolve daily, and testing capacity is limited. To further complicate the problem, almost half of the queries produced non-deterministic results—making simple checksum-based comparison (as discussed in Section 4.3.2) an impractical approach. Innovative solutions were necessary to ensure Prestissimo could effectively and efficiently verify query results, enabling rollout of new features and pipelines.

To address verification needs, we built testing frameworks at a variety of levels. These tests covered the entire development lifecycle: CI, code-land time, continuous builds, and release verification done prior to deployments. The tests covered granular Velox components such as scalar and aggregate functions, operators, memory management, shuffles, and many more, to end-to-end SQL queries using the entire Presto stack. Naturally, each component had its own traditional unit tests, which covered a fixed number of inputs. However, to make the Prestissimo migration successful, comprehensive testing infrastructure was required at each component level to minimize the risk of integration issues, which unit tests in isolation would not provide.

4.3.1 Velox Fuzzers. In a query engine, traditional unit testing, while necessary, cannot adequately cover the combinatorial explosion of operator and expression compositions, input data encodings, and execution paths that are found in production workloads. Velox addresses this challenge through a comprehensive randomized testing infrastructure (referred to as "fuzzers"), which combines differential testing, oracle-based validation, and metamorphic testing techniques. Because verifying each Presto query is not feasible, Velox fuzzers provide a resource-efficient solution to test against synthetically generated input, covering a wide range of expressions and corner cases; most importantly, they can do so using only modest computational resources. The major components of the Velox fuzzer suite include:

Expression Fuzzer. The Expression Fuzzer increases confidence in Velox expression and scalar-function correctness amid complex

interacting optimizations (such as constant folding, common subexpression evaluation, lazy evaluation, encoding preservation) [6]. It uses differential testing: the same randomly generated expression is evaluated along an optimized path and a simplified reference path (with encodings flattened and optimizations disabled), and the results must match. Inputs are random expression trees built from registered function signatures, with optional bias toward specific functions under development.

Aggregation Fuzzer. The Aggregation Fuzzer validates the correctness of Velox aggregate functions via differential testing against an external reference engine (typically legacy Presto Java), since aggregates lack a sufficiently simplified "source-of-truth" implementation. It has uncovered bugs in areas such as null handling, arithmetic overflow, timestamp precision, timezone behavior, and decimal arithmetic. Using Presto Java as the oracle is feasible because both systems target identical Presto SQL aggregation semantics; other database engines could not serve reliably due to semantic differences.

Join Fuzzer. The Join Fuzzer is used to validate different join strategies over the same input data. The fuzzer generates probe and build inputs with controlled overlap (typically 10% matching keys), and executes a sequence of logically equivalent join physical plans including hash, merge, and nested loop joins, ensuring they in fact produce the same results.

Window Fuzzer. The Window Fuzzer extends the Aggregation Fuzzer for window aggregation functions. The fuzzer generates different frame types and bounds to validate that window functions produce consistent results, using Presto Java as the source of truth.

Memory Spilling and Arbitration Fuzzer. Unlike other fuzzers that validate result correctness, this fuzzer tests the memory management subsystem under concurrent query execution with constrained memory. This fuzzer runs multiple concurrent queries competing for limited memory, triggering spilling across different operators simultaneously, testing query cancellation and resource cleanup under pressure, and validating that memory is properly reclaimed after query completion. No reference query runner is used; the fuzzer focuses on stability, using internal consistency checks and absence of memory leaks or deadlocks as validation criteria.

4.3.2 Presto Verifier. The Presto Verifier is an end-to-end SQL testing strategy that uses a known Presto release "golden version" as the source of truth. During Prestissimo development, Presto Java acted as the golden version for result comparison. This test methodology is more computationally demanding than fuzzers, as it requires a Presto cluster setup and capacity allocation to run queries end to end, but provides additional coverage against integration issues. Presto Verifier runs are automatically scheduled during release verification, but can also be triggered by developers who would like to validate a particular feature.

Presto Verifier works in the following way: For any given query, it will write temporary tables using both the control and test versions, run checksums over the temporary table data, and compare them. The process consists of the following steps:

(1) Input query, e.g.: *"SELECT column1, column2 FROM table;"*
(2) Temporary table and data generation query: *"CREATE TABLE temp_table AS SELECT column1, column2 FROM table;"* (runs in control and test version).
(3) Checksum query: *"SELECT count(*), checksum(column1) as column1_checksum, checksum(column2) as column2 _checksum from temp_table"* (runs in control and test).

For insert queries, a similar approach is used:

(1) Input query: *"INSERT INTO table (column1, column2) SELECT 1, 2"*
(2) Temporary table generation query: *"CREATE TABLE temp _table LIKE table"* (runs in control and test).
(3) Insert data in temporary table: *"INSERT INTO temp_table (column1, column2) SELECT 1,2"* (runs in control and test).
(4) Checksum queries identical as above.

Presto Verifier takes a list of queries (the verifier test suite) to validate, often sampled from production workloads, and in case of discrepancies reports "row mismatch", "column mismatch", or both. This validation tool has extensively helped provide a high-signal correctness test during the Prestissimo rollout, as it replicates real workloads found in production.

During this process, we have also enhanced the Presto Verifier to tackle non-deterministic queries using function rewrites. As discussed above, queries using functions like random(), approx_distinct(), and approx_percentile() provide similar but likely different results between the C++ and Java stacks. To ensure we were able to deterministically validate such queries, we used query rewrites to replace functions known to be non-deterministic with a deterministic one. Besides the actual non-deterministic function implementation, this method allowed us to validate other aspects of the query (e.g., joins or subselects), leaving the validation of the non-deterministic function implementation itself to be done using Velox fuzzers. This combination served us well during the migration, enabling our team to vastly increase the coverage of our test suite as a large fraction of queries were non-deterministic in nature.

Besides non-deterministic query coverage, a series of extensions were made to our test frameworks to increase coverage. For example, to prevent a bug found in the new TableWriter operator that caused data to be misaligned across partitions being written, or within partition buckets (and hence couldn't be caught by simply checksumming entire partitions), we enabled more granular checks. For instance, in a query such as *"INSERT INTO new_table (ds, column1) SELECT ds, column1 from table1"*, for a table partitioned by ds, our framework validates that new_table will also have the same number of partitions, each with an individual checksum. A similar approach is used for bucketed tables.

4.3.3 Query Banks. While Presto Verifier provides high-confidence test signals by replaying actual production workloads, the tests executed may not be entirely reproducible as workloads and data change over time. For example, a production pipeline being phased out by a pipeline owner may result in a feature losing coverage on the next Presto Verifier run. To avoid test coverage loss and improve reproducibility, we have created query banks. Query banks [40] are a pre-defined set of queries that are executed over datasets generated synthetically (and deterministically), using LLM-based

techniques that are able to replicate production data patterns, in a privacy-safe manner. Queries in query banks are collected from various historical incidents, to prevent bugs that managed to escape all other types of tests from re-appearing in production. Query banks are continuously improved as new features are developed, serving as a high-signal regression testing framework. Similar to Verifiers, Query bank queries are either run during release verification, or when explicitly triggered by developers.

4.4 Backward Compatibility

During the migration period, because queries could be routed to run in either a Java or a C++ cluster (as shown in Figure 4), ensuring strict backward compatibility was desirable to minimize potential customer impact. However, full backward compatibility could not always be maintained, due to latent bugs in the Presto Java codebase, semantic differences in C++ libraries used by Velox, or stricter invariants applied by the Velox library. Where feasible and correct, we focused on reducing divergence by applying fixes to either or both of the stacks. In rare cases where we intentionally decided to break backward compatibility, we supported affected users through documentation and targeted "*white-glove*" onboarding, but kept it to a minimum to reduce overhead.

Fixes to the stacks allowed us to reach feature parity in ~99% of queries; the remaining ~1%, spanning thousands of distinct query patterns, required us to closely work with customers. Below, we provide examples of subtle bugs and semantic differences found in the process:

NaN semantics. Presto Java exhibited subtly inconsistent handling of NaN (not-a-number) values in ordering and comparison operations, with different functions and operators producing contradictory results (e.g., *"SELECT NAN() > INFINITY()"* yields False, while *"SELECT ARRAY_SORT(a) FROM (VALUES ARRAY[NAN(), INFINITY()]) AS T(a)"* produces *["Infinity", "NaN"]*) [47]. Rather than replicating this inconsistency, we reviewed and comprehensively fixed Presto's NaN semantics [57].

Timezone Conversions. The Presto Java version used the old Joda [14] library (version 2.12.15), which used an embedded and outdated IANA timezone database [51]. This caused inconsistencies in conversions involving timezones that contained recent daylight saving modifications. We updated Java's timezone database to resolve it.

Like semantics. In Presto Java, LIKE expressions were not looking past new lines when there was no wildcard [49]. As a result, certain expressions were incorrectly returning matches when, after the newline, the input did not match. For example: *"SELECT 'foo<newline>bar' LIKE 'foo'"*. Previously that query would mistakenly return "true".

Fixing these and a series of other gaps enabled us to verify and roll out a larger number of queries as they increased result parity. Next, we present some of the situations where we decided to keep Presto Java and C++ incompatible [35], and worked with customers to adapt queries as needed.

Floating point to String conversion. There are subtle differences in how floating point values (real and double) are formatted as strings. Presto Java relies on Java's native implementation for converting floating points to strings, while Velox uses the Folly C++ library. For example, for *"SELECT cast(3.8027738365156062E14 as varchar)"*, Java would return "3.802773836515606E14" and Prestissimo would return "3.8027738365156062E14" (extra '2' before 'E').

Regular Expression Library. Presto Java uses the Joni [15] library, and Velox uses RE2 [18] for regular expression evaluation. While most of the semantics are supported by both libraries, RE2, unlike Joni, does not support all PCRE functionality and all look-around assertions [19]. However, RE2 guarantees linear-time execution and a fixed stack footprint, which is a critical requirement for multi-tenant systems like Presto. In other libraries (such as boost::regex [4] and std::regex [20]), look-around assertions are implemented with recursive backtracking solutions that are time- and memory-intensive, and caused production incidents in various services (including Presto) at Meta.

Array Sort Lambda. Presto has support for the *array_sort()* function, which takes a custom comparator lambda expression to sort the given array. In a vectorized engine like Velox, this requires a vectorized expression evaluation call for each comparison made by the sorting algorithm, adding substantial overhead. While Java can more efficiently support this type of construct due to JVM's JIT compilation, we decided to remove support since it was only sporadically used [26].

JSON Functions. Presto Java uses the Jackson library [12] for JSON processing; Velox adopted simdjson [22] for vectorized efficiency. While Velox's simdjson-based implementation could support almost all JSON paths supported by Jackson, there were subtle differences. For example:

- Using functions inside a JSON path is not supported. For example, queries like *"CAST(JSON_EXTRACT(config, '$.properties.keys()') AS ARRAY(ARRAY(VARCHAR)))"* needed to be rewritten as *map_keys(JSON_EXTRACT(config, '$.properties'))*.
- Expressions inside the JSON path aren't supported. For example: queries like *JSON_EXTRACT(config, '$.store.book[?(@.price > 10)]')* needed to be rewritten where the filter involving price happened after the json_extract. Velox's vectorized execution makes this faster in practice rather than resorting to un-vectorized JSON library implementation.
- Presto Java can parse some invalid JSON, but not all. Velox keeps the behavior deterministic and always fails on bad JSON [48]. Presto Java can produce non-canonicalized JSON output while Velox always produces canonicalized form [63].

Date/Time Differences. Currently the maximum date range supported for *from_unixtime()* and similar functions is between (292 million BCE, 292 million CE). The exact values corresponding to this are [292,275,055-05-16 08:54:06.192 BCE, +292,278,994-08-17 00:12:55.807 CE], and this corresponds to a unix time between [-9223372036854775, 9223372036854775]. Presto Java also supports the same range, however it silently truncates (hence queries used to succeed), whereas Prestissimo throws an error if the values exceed

this range. Once again, due to Hyrum's law [11], we found customer queries depending on this specific unintentional behavior.

URL mismatches. Presto and Prestissimo implement different URL function specs which can lead to some URL function mismatches [36]. Prestissimo implements RFC-3986 [29] whereas Presto implements RFC-2396 [28].

4.5 Reliability and Stability

Prestissimo aimed not only to match but to exceed the decade-hardened reliability of the Presto Java stack, while improving efficiency on less hardware. Because new codebases inevitably introduce new failure modes, we invested heavily in production-traffic shadowing: a dedicated service that replays real workload traffic on new Prestissimo builds without deploying them to production. This infrastructure was designed to surface server-health and stability issues that emerge only under real conditions—e.g., peak load, resource contention, and other multi-tenant cluster dynamics. The shadow framework enabled the following testing signals:

4.5.1 Performance Comparisons. The shadow testing tool enabled direct performance comparisons between production and test clusters by mirroring production queries in near real time. This approach allowed for precise measurement of key metrics such as execution time, CPU time, and memory usage, providing immediate feedback for evaluating test builds against production standards.

4.5.2 Stress Testing. The shadow tool also supported stress testing by directing workloads from multiple clusters to a single test cluster, artificially inflating workloads and sampling from different regions and traffic sources. This process was critical to assessing system robustness, and ensuring Prestissimo gracefully handled high load conditions.

4.5.3 Stability Testing. Effective server health monitoring is essential for maintaining reliability in a multi-tenant environment. The shadowing tool enabled the early identification of critical issues such as stuck tasks and threads, deadlocks, starvation, and segmentation faults in test servers. Early exposure to these issues enabled developers to reproduce, diagnose, and validate fixes prior to production deployment, thereby significantly enhancing the reliability and stability of Prestissimo.

4.5.4 Shadow Monitoring and Observability. The Presto shadow service operated in real time over extended periods. It generated dashboards to expose a range of metrics, including execution time, queue time, operator performance, and segmentation faults, enabling early detection and remediation of issues. This comprehensive observability framework further supported the proactive maintenance and continuous improvement of Prestissimo deployments.

4.6 Team/People Management

The migration to a composable Prestissimo/Velox architecture reshaped team ownership: the Presto team moved from end-to-end control of the stack to depending on Velox, a general-purpose execution library maintained by a separate team and shared across multiple systems. To manage this shift, we formed a joint Prestissimo group staffed by both Presto and Velox engineers, with deliberately porous boundaries across the library, service, and tooling, and a shared oncall to keep development and operations tightly coupled.

As rollout matured and Velox contributors shifted toward other engine integrations, we mitigated the risk of knowledge loss and reduced agency within the Presto team by deliberately broadening Velox expertise across Presto engineers and ensuring continued Presto participation in Velox development and governance.

5 PRODUCTION EXPERIENCE AND BENCHMARKS

Below, we present production metrics illustrating improvements over the migration timeline, performance comparison of Prestissimo against vectorized engines, and an overview of the cost model reductions achieved with Prestissimo.

5.1 Performance, Stability and Maintenance

Production data demonstrates that the decision to avoid a hybrid solution for Prestissimo is justified. Figure 5 demonstrates average ~2–5x execution-time improvements in batch, and ~6–7x improvements in interactive workloads. These gains are delivered by the Velox execution engine through vectorized processing, SIMD-optimized operators, and hybrid SSD/memory caching, while Prestissimo itself acts as a thin integration layer bridging Presto's protocol and plan representation to Velox without contributing execution logic of its own. Figure 6 also shows a histogram of relative speedup provided by Prestissimo (for interactive use cases) where 0x means Presto Java is faster; 10x means Prestissimo is 10 times or more faster. We have observed that many queries experienced more than an order of magnitude speedup due to Velox's hybrid caching techniques.

Prestissimo exhibits superior reliability under stress, achieving a daily success rate of ~99.8%, compared to ~99% for Presto Java. As shown in Figure 7, this decomposes into distinct contributions: the elimination of JVM pause errors is a direct consequence of removing the JVM entirely, which was the migration's core architectural decision; the reduction in out-of-memory errors stems from Velox's memory arbitration with automatic spilling and Prestissimo's system-level pushback mechanism (Section 3.1); and the decrease in inter-worker communication errors reflects the reliability of Proxygen's battle-tested C++ HTTP framework replacing Jetty's Java-based implementation, which was more susceptible to failures under garbage collection pressure. Routine maintenance tasks such as restarting clusters to manage high heap usage are no longer necessary. Note that these improvements compound only at full migration. Retaining even a fraction of Java clusters would have preserved the legacy hardware footprint and doubled the operational surface area, negating much of the efficiency and simplification achieved.

The release cadence for Prestissimo has improved to a weekly schedule, a longstanding goal for the Presto team. In contrast, Presto Java releases were delayed by the absence of robust fuzzing frameworks, requiring extensive end-to-end verification and resulting in a 2–3 week (or longer) release cycle. With advancements in Velox fuzzing, Prestissimo maintains correctness despite a tenfold increase in weekly code changes, enabling faster and more reliable releases. The quality of releases has also improved, with rollbacks decreasing from twice per month to twice in six months.

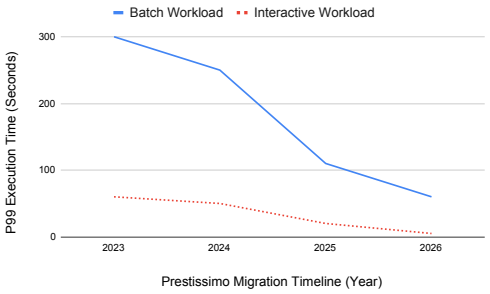

Figure 5: Prestissimo execution time improvement over Presto Java

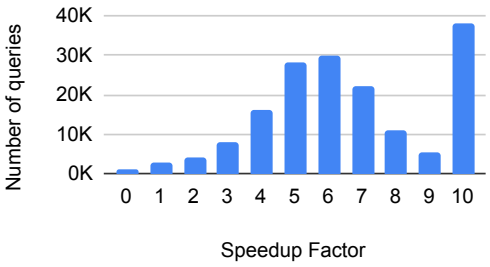

Figure 6: Distribution of per-query speedup of Prestissimo over Presto Java for interactive queries. A value of 0x means Presto Java is faster; 10x means Prestissimo is ≥10x faster.

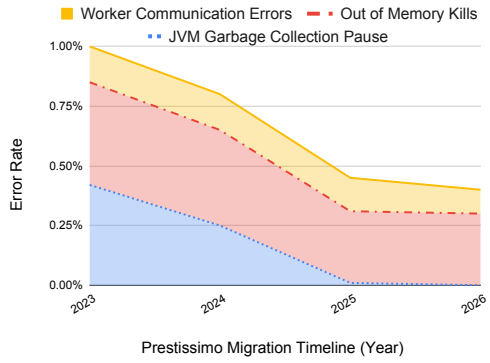

Figure 7: Prestissimo error rate improvement over Presto Java

5.2 Vectorized Engine Comparison

Meta's production workload relies heavily on nested types, JSON, regular expressions, geospatial functions, custom UDFs, and complex join patterns that extend beyond what standard benchmarks like TPC-H [62] or TPC-DS [61] are designed to measure. Nonetheless, to provide a standardized comparison, we benchmarked a local Prestissimo deployment against DuckDB [55] on TPC-H. We co-located the Prestissimo coordinator and worker on a single host (158 cores, 1024 GB RAM) to approximate DuckDB's single-process

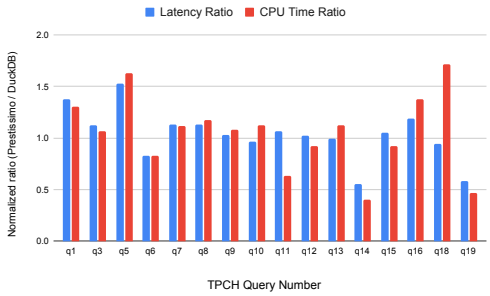

Figure 8: Prestissimo vs. DuckDB on TPC-H (SF-1000). Bars show the ratio of Prestissimo to DuckDB for latency and CPU time; values above 1 indicate Prestissimo is slower.

model. For a fair execution-focused comparison, we excluded six queries where Presto's optimizer produced suboptimal plans due to correlated subqueries, retaining only queries with equivalent plans across both engines. Figure 8 shows results normalized to DuckDB's latency and CPU time (values >1 indicate Prestissimo is slower). On aggregate, the two engines deliver comparable end-to-end latency (within 1%), with Prestissimo consuming ~9% more CPU time, competitive despite architectural differences.

5.3 Pricing Model and Demonstrating Efficiency

A core requirement of the project was demonstrating to customers that they directly benefit from Prestissimo's efficiency. At Meta, Presto attributes compute consumption using a virtual resource type (as required by Meta Capacity Planning [1]), the Presto Compute Unit (*PCU*), which serves as both a consumption metric and a quota enforcement mechanism. Since hardware is heterogeneous, each host type w is assigned a normalized weight ρ_w that represents its contribution to the fleet, making capacity comparable across configurations. A higher ρ_w means fewer hosts of that type are needed to serve a given *PCU* budget, so ρ_w directly governs the hardware footprint. The *PCU* charge for a query q is:

$$PCU_q = \sum_w f_{q,w}(\psi_{q,w}, \phi_{q,w}) \times \rho_w$$

where $f_{q,w}$ captures the fraction of worker w's resources consumed by q, based on a CPU component $\psi_{q,w}$ and a memory component $\phi_{q,w}$. Since most queries are either CPU-bound or memory-bound, the charging function reflects the dominant resource dimension for each worker. Because Presto operates on shared, multi-tenant infrastructure, per-query consumption may exhibit minor fluctuations due to co-located workloads, but remains sufficiently stable for capacity planning.

Shadow testing validated that Prestissimo substantially reduces $f_{q,w}$ for the same queries, due to Velox, efficient memory management, and the elimination of JVM overhead. The challenge was surfacing this efficiency gain to customers through the *PCU* model. During the migration, both Presto and Prestissimo clusters were active, and a given query could be routed to either engine type, causing per-query consumption to vary unpredictably. To maintain stability, we applied an artificial inflation multiplier to queries running

on Prestissimo clusters, normalizing their *PCU* charge to match the Java baseline. Once all Java clusters were decommissioned, we removed the multiplier and increased ρ_w to reflect Prestissimo's true efficiency while preserving PCU_q. The same workload now requires significantly fewer compute resources, translating into a substantial reduction in the overall hardware footprint.

6 LESSONS LEARNED

The development of Velox from the ground up and subsequent integration into existing massive-scale infrastructure systems like Prestissimo was an ambitious, but successful and well-rewarded bet. Below, we highlight key lessons learned.

6.1 *Soft* Backward Compatibility

Migrating a lakehouse query engine serving hundreds of millions of daily queries without disrupting pipelines is difficult: beyond matching documented SQL semantics, it must reproduce undocumented edge cases embedded in the legacy codebase. Under Hyrum's law [11], even legacy bugs can become de facto API contracts, so fixing them in the new C++ stack sometimes broke existing workloads. We initially prioritized backward compatibility, but quickly encountered diminishing returns where underlying libraries (e.g., regex, JSON, date/time parsing) diverged and porting Java behavior to C++ offered limited migration value while adding long-term debt. Instead, we handled incompatibilities case-by-case—identifying impacted pipelines, communicating broadly, and partnering with owners to update queries (notably for regex). Although this imposed upfront overhead, it ultimately produced a more robust and internally consistent stack with clearer semantics.

6.2 *Oversized* Testing Infrastructure Investment

Shipping bugs in a production lakehouse query engine can cause data corruption and expensive rollbacks, repairs, and backfills, while also eroding organizational confidence. To reduce this risk, we invested early in comprehensive testing, validation, and verification infrastructure (Section 4.3), continually expanding it as new issues emerged. Although this slowed initial rollout, it enabled a more reliable and reproducible Prestissimo deployment and produced reusable tooling now leveraged in other Velox-based integrations (e.g., Spark) and internal systems.

6.3 The *Fixed* Cost of Composability

Prestissimo was Meta's first large-scale deployment of a composable library architecture [31], but because Velox was built and hardened in parallel, the immediate reuse benefits were limited. Designing Velox as a general, extensible component rather than a Presto-specific module likely increased the initial timeline, but has since accelerated integrations with other engines by enabling reuse of battle-tested features with primarily integration-level validation.

6.4 Know *Your* Traffic

Despite the heterogeneity of Presto workloads at Meta, the systematic analysis of monitoring logs and usage statistics has been invaluable for migration planning. During the early stages of the transition, this data-driven methodology enabled the team to identify and prioritize the most frequently utilized features, thereby focusing development efforts on high-impact functionalities. As a result, fixed-shape queries and the most commonly employed capabilities were addressed first, proving the investment in Prestissimo worthwhile by showing cost and performance benefits early.

7 RELATED WORK

The migration from Google Datastore to Firestore [34] shares foundational principles with Prestissimo: zero downtime, no data (or, in Prestissimo's context, no queries) left behind, automated migration, and active customer engagement throughout the transition.

Disaggregated storage-compute architectures are well established, with cloud analytics engines such as Redshift [25], Snowflake [33], Dremel [41, 42]/BigQuery, and Procella [30]. Spark [65] is widely adopted for batch ETL, while SparkSQL [24] adds SQL and DataFrame APIs.

Hybrid Java/C++ architectures are common. Photon [27] accelerates Spark by offloading portions of the query plan to a native engine via JNI, while retaining the Spark runtime for unsupported operators—unlike Prestissimo's fully native worker replacement. Similarly, Intel's OAP [46] uses Gazelle [45], a SIMD-accelerated plugin with Apache Arrow and JNI-based communication.

Vectorized execution engines such as ClickHouse [32], Hologres [39], and DuckDB [55] are increasingly prevalent. In the Spark ecosystem, Apache Gluten [10] offloads execution to native engines (e.g., Velox) via Spark's plugin interfaces. Apache DataFusion [3] provides a Rust-based, Arrow-native [2] query engine, and Apache Comet [5] accelerates Spark SQL through a similar columnar/native execution path, reducing JVM overhead.

8 CONCLUSIONS

The migration from Presto Java to Prestissimo, powered by Velox, represents a landmark achievement in the evolution of large-scale distributed SQL analytics at Meta. Through a carefully orchestrated, in-place rollout, Prestissimo has successfully replaced a decade's worth of production workloads, delivering substantial improvements in query latency, hardware efficiency, and reliability. The journey underscored the importance of investing in comprehensive testing infrastructure, including randomized fuzzers, end-to-end verifiers, and production traffic shadowing, to ensure correctness and operational excellence at scale. Prestissimo's success demonstrates the viability and long-term benefits of composable, native execution engines in modern data infrastructure. As Prestissimo paves the way for further integrations (e.g., Vector Search [64]) across Meta's data ecosystem, it sets a new standard for efficiency, reliability, and adaptability in exabyte-scale analytics platforms.

ACKNOWLEDGMENTS

We thank Andrii Rosa, Naveen Cherukuri, Swapnil Tailor, Pradeep Vaka, Han Yan, Kevin Wilfong, Jimmy Lu, Bikramjeet Vig, Wei He, Neerad Somanchi, Steve Chuck, Rebecca Schlussel, Konjac Huang, Prashant Golash, Jiayin Yan, Shelton Cai, Nikhil Collooru, Shang Ma, Zhichen Xu, Peter Enescu, Pratik Pugalia, Eric Liu, Yihao Zhou, Slava Andreykiv, Kevin Tang, Arjun Gupta, Shruthi Adappa, Matt David, and many others at Meta for their contributions to Prestissimo. We equally thank the vibrant Presto open-source community for their contributions to Prestissimo's production excellence.

REFERENCES

[1] 2024. *Meta Capacity Planning for AI Training Hardware.* Retrieved June 28, 2026 from https://atscaleconference.com/meta-capacity-planning-for-ai-training-hardware/

[2] 2026. *Apache Arrow.* Retrieved June 28, 2026 from https://arrow.apache.org/

[3] 2026. *Apache DataFusion.* Retrieved June 28, 2026 from https://datafusion.apache.org/

[4] 2026. *Boost.Regex.* Retrieved June 28, 2026 from https://www.boost.org/doc/libs/latest/libs/regex/doc/html/index.html

[5] 2026. *Comet Accelerator for Apache Spark and Apache Iceberg.* Retrieved June 28, 2026 from https://datafusion.apache.org/comet/

[6] 2026. *Expression Evaluation.* Retrieved June 28, 2026 from https://facebookincubator.github.io/velox/develop/expression-evaluation.html#common-subexpression-detection

[7] 2026. *Facebook Thrift.* Retrieved June 28, 2026 from https://github.com/facebook/fbthrift

[8] 2026. *Folly: Facebook Open-source Library.* Retrieved June 28, 2026 from https://github.com/facebook/folly

[9] 2026. *Geospatial Functions.* Retrieved June 28, 2026 from https://prestodb.io/docs/current/functions/geospatial.html

[10] 2026. *Gluten.* Retrieved June 28, 2026 from https://gluten.apache.org/

[11] 2026. *Hyrum's Law.* Retrieved June 28, 2026 from https://www.hyrumslaw.com/

[12] 2026. *Jackson.* Retrieved June 28, 2026 from https://github.com/FasterXML/jackson

[13] 2026. *jemalloc memory allocator.* Retrieved June 28, 2026 from https://jemalloc.net/

[14] 2026. *Joda-Time.* Retrieved June 28, 2026 from https://www.joda.org/joda-time/

[15] 2026. *joni regular expression library.* Retrieved June 28, 2026 from https://github.com/jruby/joni

[16] 2026. *Meta Global Data Center Fleet.* Retrieved June 28, 2026 from https://datacenters.atmeta.com/all-locations/

[17] 2026. *Prestissimo (Presto Native Execution).* Retrieved June 28, 2026 from https://github.com/prestodb/presto/tree/master/presto-native-execution

[18] 2026. *RE2, a regular expression library.* Retrieved June 28, 2026 from https://github.com/google/re2

[19] 2026. *RE2 Syntax.* Retrieved June 28, 2026 from https://github.com/google/re2/wiki/syntax

[20] 2026. *Regular expressions library (since C++11).* Retrieved June 28, 2026 from https://en.cppreference.com/w/cpp/regex.html

[21] 2026. *Scaling with Presto on Spark.* Retrieved June 28, 2026 from https://prestodb.io/blog/2021/10/26/scaling-with-presto-on-spark/

[22] 2026. *simdjson.* Retrieved June 28, 2026 from https://github.com/simdjson/simdjson

[23] 2026. *Velox CompactRow.* Retrieved June 28, 2026 from https://facebookincubator.github.io/velox/develop/serde/compactrow.html

[24] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[25] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[26] Masha Basmanova. 2026. *array_sort lambda function.* Retrieved June 28, 2026 from https://velox-lib.io/blog/array-sort/

[27] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.

[28] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 1998. *Uniform Resource Identifier (URI): Generic Syntax.* Retrieved June 28, 2026 from https://datatracker.ietf.org/doc/html/rfc2396

[29] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 2005. *Uniform Resource Identifier (URI): Generic Syntax.* Retrieved June 28, 2026 from https://datatracker.ietf.org/doc/html/rfc3986

[30] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-Ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying Serving and Analytical Data at YouTube. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2022–2034.

[31] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian Sun, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *CIDR*.

[32] ClickHouse. 2026. *ClickHouse.* Retrieved Feb 20, 2026 from https://clickhouse.com/

[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[34] Ed Davisson, Tilo Dickopp, David Gay, Eric Karasuda, Ram Kesavan, and Vadim Yushprakh. 2024. Transparent migration from Datastore to Firestore. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3960–3972.

[35] Amit Dutta. 2026. *Presto vs Prestissimo – Known differences and workarounds.* Retrieved June 28, 2026 from https://prestodb.io/blog/2026/01/22/presto-vs-prestissimo-known-differences-and-workarounds/

[36] Peter Enescu. 2026. *Known URL mismatches.* Retrieved June 28, 2026 from https://github.com/facebookincubator/velox/issues/14204

[37] The Presto Foundation. 2026. *Presto Native Sidecar.* Retrieved June 28, 2026 from https://prestodb.io/docs/current/plugin/native-sidecar-plugin.html

[38] The Presto Foundation. 2026. *SerializedPage Wire Format.* Retrieved June 28, 2026 from https://prestodb.io/docs/current/develop/serialized-page.html

[39] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. 2020. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. VLDB Endow.* 13, 12 (2020), 3272–3284.

[40] Yu Liu, Jiangnan Cheng, Steve Chuck, Lyublena Antova, Yurgis Baykshtis, Matt David, Ge Gao, Mehrdad Honarkhah, Kuan-Sung Huang, Chen-Kuei Lee, Usman Muhammad, Shihao Peng, Andrii Rosa, Rebecca Schlussel, Michael Shang, Kelvin Silva, Brandon Vo, Zac Wen, and Yihao Zhou. 2024. Compute Engine Testing with Privacy-Compliant Production-Like Synthetic Data. In *Proceedings of the Workshops of the 50th International Conference on Very Large Data Bases (VLDB 2024)*.

[41] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1 (2010), 330–339.

[42] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[43] Xiaoxuan Meng. 2026. *Velox Memory Management.* Retrieved June 28, 2026 from https://facebookincubator.github.io/velox/develop/memory.html

[44] Xiaoxuan Meng. 2026. *Velox Spilling.* Retrieved June 28, 2026 from https://facebookincubator.github.io/velox/develop/spilling.html

[45] OAP. [n.d.]. *Gazelle Plugin - A Native Engine for Spark SQL with vectorized SIMD optimizations.* Retrieved Feb 20, 2026 from https://oap-project.github.io/gazelle_plugin/latest/

[46] OAP. [n.d.]. *Optimized Analytics Package.* Retrieved Feb 20, 2026 from https://oap-project.github.io/latest/

[47] Krishna Pai. 2026. *Clarifications on comparisons with NaN.* Retrieved June 28, 2026 from https://github.com/prestodb/presto/issues/21936

[48] Krishna Pai. 2026. *Parsing bad json.* Retrieved June 28, 2026 from https://github.com/prestodb/presto/issues/24090

[49] Krishna Pai. 2026. *Presto Like pattern matching issue.* Retrieved June 28, 2026 from https://github.com/prestodb/presto/issues/23281

[50] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[51] Pedro Pedreira. 2026. *Presto Timezone issue.* Retrieved June 28, 2026 from https://github.com/prestodb/presto/issues/22981

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

[53] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zaït, and Jacques Nadeau. 2023. The composable data management system manifesto. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685.

[54] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4249–4252.

[55] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of*

data. 1981–1984.

[56] Laith Sakka, Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Wei He, Xiaoxuan Meng, Krishna Pai, and Bikramjeet Vig. 2024. Simple (yet Efficient) Function Authoring for Vectorized Engines. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4187–4199.

[57] Rebecca Schlussel. 2026. *Presto NAN RFC*. Retrieved June 28, 2026 from https://github.com/prestodb/rfcs/blob/main/RFC-0001-nan-definition.md

[58] 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)*. IEEE, 1802–1813.

[59] Daniel Sommermann and Alan Frindell. 2014. *Introducing Proxygen, Facebook's C++ HTTP framework*. Retrieved June 28, 2026 from https://engineering.fb.com/2014/11/05/production-engineering/introducing-proxygen-facebook-s-c-http-framework/

[60] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2, Article 189 (June 2023), 25 pages. https://doi.org/10.1145/3589769

[61] Transaction Processing Performance Council. [n.d.]. *TPC-DS Benchmark Specification*. Retrieved May 7, 2026 from https://www.tpc.org/tpcds

[62] Transaction Processing Performance Council. [n.d.]. *TPC-H Benchmark Specification*. Retrieved May 7, 2026 from https://www.tpc.org/tpch

[63] Bikramjeet Vig. 2026. *Inconsistent behavior in json functions*. Retrieved June 28, 2026 from https://github.com/prestodb/presto/issues/24563

[64] Zhichen Xu, Ge Gao, Ke Wang, Aakash Deep, Junjie Qi, Jort Gemmeke, Amit Dutta, Xiaoxuan Meng, Trang Nguyen, Jiayin Yan, Xiao Du, Vivek Gaur, Kaushik Ravichandran, and Vishal Gandhi. 2026. SQL-Native Vector Search at Billion Scale in Presto. *Proceedings of the VLDB Endowment* 19 (2026).

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*.

[66] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra 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: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*. 1042–1057.