Presto: A Decade of SQL Analytics at Meta

YUTIAN "JAMES" 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, and BISWAPESH CHATTOPADHYAY*, Meta Platforms, Inc, USA
BIN FAN, Alluxio, Inc, USA
DEEPAK MAJETI and ADITI PANDIT, Ahana Cloud, Inc, USA

Presto is an open-source distributed SQL query engine that supports analytics workloads involving multiple exabyte-scale data sources. Presto is used for low-latency interactive use cases as well as long-running ETL jobs at Meta. It was originally launched at Meta in 2013 and donated to the Linux Foundation in 2019. Over the last ten years, upholding query latency and scalability with the hyper growth of data volume at Meta as well as new SQL analytics requirements have raised impressive challenges for Presto. A top priority has been ensuring query reliability does not regress with the shift towards smaller, more elastic container allocation, which requires queries to run with substantially smaller memory headroom and can be preempted at any time. Additionally, new demands from machine learning, privacy, and graph analytics have driven Presto maintainers to think beyond traditional data analytics. In this paper, we discuss several successful evolutions in recent years that have improved Presto latency as well as scalability by several orders of magnitude in production at Meta. Some of the notable ones are hierarchical caching, native vectorized execution engines, materialized views, and Presto on Spark. With these new capabilities, we have deprecated or are in the process of deprecating various legacy query engines so that Presto becomes the single piece to serve interactive, ad-hoc, ETL, and graph processing workloads for the entire data warehouse.

*Author was affiliated with Meta during the contribution period.

Authors' addresses: Yutian "James" Sun, jamessun@meta.com; Tim Meehan, tdm@meta.com; Rebecca Schlussel, rschlussel@meta.com; Wenlei Xie, wxie@meta.com; Masha Basmanova, mbasmanova@meta.com; Orri Erling, oerling@meta.com; Andrii Rosa; Shixuan Fan; Rongrong Zhong; Arun Thirupathi, athirupathi@meta.com; Nikhil Collooru, nikhilcollooru@meta.com; Ke Wang, ke1024@meta.com; Sameer Agarwal, sag@meta.com; Arjun Gupta, pgupta6@meta.com; Dionysios Logothetis, dionysios@meta.com; Kostas Xirogiannopoulos, kostasx@meta.com; Amit Dutta, adutta@meta.com; Varun Gajjala, vgajjala@meta.com; Rohit Jain, rohitism@meta.com; Ajay Palakuzhy, ajaygeorge@meta.com; Prithvi Pandian, prithvip@meta.com; Sergey Pershin, spershin@meta.com; Abhisek Saikia, asaikia@meta.com; Pranjal Shankhdhar, pranjalssh@meta.com; Neerad Somanchi, neeradsomanchi@meta.com; Swapnil Tailor, swapniltailor@meta.com; Jialiang Tan, jtan6@meta.com; Sreeni Viswanadha, viswanadha@meta.com; Zac Wen, zacw@meta.com; Biswapesh Chattopadhyay, Meta Platforms, Inc, 1 Hacker Way, Menlo Park, California, USA, 94025; Bin Fan, bin.fan@alluxio.com, Alluxio, Inc, 1825 South Grant Street Suite 600, San Mateo, California, USA, 94402; Deepak Majeti, deepak@ahana.io; Aditi Pandit, aditi@ahana.io, Ahana Cloud, Inc, 238 Castro Street, Mountain View, California, USA, 94041.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/6-ART189 $15.00
https://doi.org/10.1145/3589769

CCS Concepts: • **Information systems → Database query processing**; **Parallel and distributed DBMSs**; **Online analytical processing engines**.

Additional Key Words and Phrases: Data Warehouse, Presto, OLAP, SQL, Distributed Database, Data Analytics, ETL

ACM Reference Format:
Yutian "James" 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

1 INTRODUCTION

Presto [44] is an open-source distributed query engine that has supported production analytical workloads at Meta since 2013. It provides a SQL interface to query data stored on different storage systems, such as distributed file systems. Since its donation to Linux Foundation in 2019, Presto has continued growth in utilization and contribution among US tech industry leaders including Uber, Twitter, Intel, Ahana, etc. After the donation, Meta remains active in Presto contributions with 50% commits coming from Meta. The deployment of the Presto fleet at Meta is also on top of the trunk to ensure every release is battle tested at Meta scale.

Within Meta, Presto is used for interactive, ad-hoc, and extract-transform-load (ETL) workloads at scale. Use cases include dashboarding, A/B testing, ad-hoc analysis, data cleaning, and transformation. With the effort of migrating all SparkSQL [6] workloads to Presto at Meta, Presto will soon provide the only SQL interface to the warehouse in the company.

While Presto was originally designed for exclusively in-memory processing of interactive SQL querying, various trends at Meta challenged its capabilities. Due to its efficiency, employees started to use it for lightweight ETL workloads [44] that run for up to tens of minutes, and, as data grew exponentially, Presto became slower. The transition to a more flexible and elastic resource management model with smaller, ephemeral containers resulted in reduced reliability. Moreover, while there was a growing demand for richer analytics, such as machine learning feature engineering, and graph analytics, they were not well supported. Finally, honoring Meta's data privacy policies required new data abstractions and data storage mechanisms to support privacy enforcement efficiently. The main focus of this paper is to describe how we have improved the architecture of Presto to address these challenges with the following three perspectives.

First, **latency and efficiency**. As data increases, the scan cost of the same query increases leading to longer wait time. As the number of RPC connections of machines in a cluster cannot increase arbitrarily large, adding more machines to a cluster will reach a limit. Also, more machines in use inherently increases the chance of a single machine failure. Other latency improvements are needed to ensure users can still have a low-latency dashboarding experience with large, scalable data scans. Especially for important dashboards, users expect Presto to perform as if the data is already pruned or stored in memory with arbitrary slicing and dicing.

Second, **scalability and reliability**. SQL is preferred for Meta's ETL workloads, which drives Presto's popularity. As Presto does not provide fault tolerance and memory is limited by hardware, new approaches are needed for Presto to support ETL workloads that are orders of magnitude heavier in terms of CPU, memory, and runtime than what Presto currently supports in [44]. In addition, Meta has adjusted the container allocation to be more elastically manageable with a smaller memory footprint. Elasticity allows more flexible capacity to balance peak and off-peak

usage across different types of workloads in the company. However, it casts a complex challenge as machines can go down arbitrarily. With these constraints, new design principles are needed to scale the workload to handle arbitrarily large memory consumption and arbitrarily long runtime with an unstable underlying infrastructure.

Lastly, **requirements going beyond data analytics**. The modern warehouse has become a data lake to allow data usage according to the needs of diverse use cases. A typical use case is machine learning feature engineering. Meta's machine learning related data volume has already exceeded that of analytics. Machine learning engineers leverage analytical engines like Presto or SparkSQL to extract features from raw data for training purposes. Privacy is another important requirement. Facebook, Instagram, and WhatsApp users can decide to opt out of personal data use for content recommendation or any other use cases for the data that has already been collected by Meta. Presto is on the path to ensure the data is properly protected. Moreover, Meta is all about social graphs. We have seen users asking for SQL-like graph analytics through Presto to express complex logic with billions of nodes and edges.

The remainder of the paper is structured as follows. Section 2 provides an overview of Presto's original architecture and the challenges based on the architecture in the past few years at Meta. Sections 3, 4, and 5 introduce the evolution of Presto to improve latency, scalability, and efficiency respectively. Section 6 discusses spaces including machine learning, privacy, and graph analytics to illustrate how users leverage Presto as an engine to manipulate Meta warehouse data for richer analytics. Section 7 demonstrates how these evolutions can help to improve the performance on production data at Meta's scale. Section 8 discusses the related work in this space and Section 9 summarizes remaining challenges and planned work to address them. Our conclusion in Section 10 highlights the improvements discussed in the paper and our deprecation of various engines as Presto becomes the centerpiece of our warehouse.

2 ARCHITECTURE AND CHALLENGES

As of 2022, Meta has 21 data centers [12] worldwide. Each of these data centers is millions of square feet in size. Meta's data warehouse has a non-trivial amount of storage across these data centers. The majority of all Meta employees use Presto daily directly or indirectly through other tools to access those data.

With Meta's warehouse data growing exponentially, Presto has faced various difficulties to guarantee the same latency and scalability experience for users. As dashboards became slower with larger scans, users started to leverage in-memory or collocated-storage compute engines [40, 44] for better performance. On the ETL side, more scalable engines like Spark [57] were preferred as builtin fault tolerance can guarantee long-running jobs finish even with container crashes. The growing trend of using elastic capacity requires allocating and deallocating containers at a much higher frequency. Today, there is no guarantee that a container can be dedicated to a Presto cluster uninterrupted for hours. The original architecture of Presto with disaggregated storage and in-memory processing can only optimally handle queries running between a few seconds to minutes. As the demands of Presto evolved beyond its original requirements, we engineered ways to evolve Presto itself to overcome the challenges presented.

Figure 1 illustrates the original architecture of a Presto cluster [44]. It consists of a single *coordinator* and a number of *workers* that can scale to thousands. The coordinator is responsible for queueing and parsing a query string, then turning it into a *plan*. Optimizations will be applied to the plan and later fragmented into *plan fragments* or simply *fragments* based on *shuffle* boundaries. These fragments will be scheduled to workers in parallel. Workers are responsible for query processing with all data in memory and data shuffling through streamed RPCs over the network. Each worker will launch *tasks* to process the data based on the fragments received. The processed

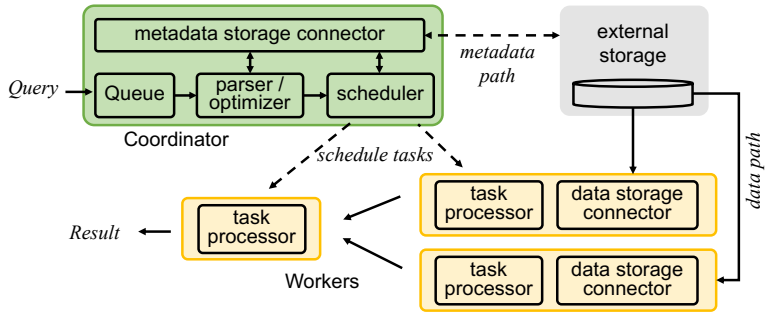


Fig. 1. Original Presto architecture

data will be shuffled into different buffers in memory waiting for the different downstream tasks to fetch. A cluster can run multiple queries and their tasks concurrently with full multi-tenancy sharing memory, IO, network, and CPU. Presto also supports *storage connectors* to allow scanning heterogeneous data sources for the same query.

As we can tell from the original architecture, the latency can be bottlenecked by IO due to external storage being disaggregated from compute engines. Moreover, all workers are currently written in Java, which is programmatically slower than native code without fine control over memory allocation. The storage connectors have also become a double-edged sword: to support low-latency dashboards, systems like Raptor [44] were built to dump warehouse data to local memory or disk with specialized types of machines so dashboards can load faster. The collocated storage not only introduced data management overhead but also diminished the benefit of independent scaling of storage and compute.

From the scalability perspective, the coordinator being a single point of failure and the lack of fault tolerance of workers have been magnified with the data growth as well as the introduction of elastic capacity. The in-memory processing design also defines an upper bound of how much data the system can hold. The network-based shuffle cannot scale beyond thousands of workers due to connection limitations.

In addition to the latency and scalability challenges, the growing trend of machine learning-focused and privacy-focused requirements has gradually shifted the traditional analytics-focused data warehouse into a more open and flexible "data lake" setup. Analytical data is no longer immutable. Meta needs the ability to remove user data in response to users' privacy choices. Columns can also be added with high flexibility to try out different candidate features during machine learning feature engineering.

In the remainder of this paper, we discuss several Presto evolutions that have been rolled out successfully at Meta to solve the above challenges. Some of the improvements have more in-depth discussion in our prestodb blogs [1, 8, 17, 29, 37, 54, 55]. Figure 2 illustrates the high-level idea of the new Presto architecture. When a query is sent to the Presto fleet, it can be run on either (1) the original Presto architecture but with multiple coordinators to avoid single point failure, native vectorized execution to boost performance, data cache on flash to avoid IO bottleneck, and many other improvements that will be discussed later in the paper or (2) Presto on Spark that leverage Spark as the runtime and Presto as the evaluation library for scalability. In both setups, we provide materialized views to improve query performance and data mutability for machine learning feature engineering and privacy use cases. Moreover, both setups can spill the in-memory data to temporary storage to overcome memory limitations. The original Presto architecture now is also enhanced with recoverability to materialize intermediate data. The Presto on Spark setup,

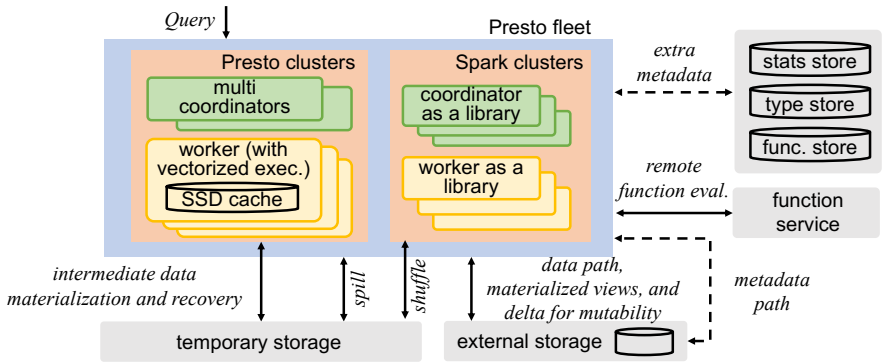


Fig. 2. New Presto architecture

on the other hand, leverages the temporary storage for shuffle. Extra metadata is also introduced. Type store is used for supporting user-defined types, function store is for supporting SQL function authoring and evaluation, and statistics store is used for better optimization decisions. Remote functions are built for running user-defined functions.

3 LATENCY IMPROVEMENTS

As data grows, the query latency will naturally suffer from degradation. This section introduces several enhancements to Presto to improve latency from CPU, IO, and memory perspectives.

3.1 Caching

Disaggregated storage enables scaling and independent computation. However, the disaggregation introduces new challenges for query latency since scanning huge amounts of data or even metadata over the wire can be IO bound when the network is saturated. To solve this problem, we introduced caches at various levels. In the remainder of this paper, we use the concept of *files* that represent slices of data that are physically stored in the remote storage.

Raw Data Cache: Data cache on local flash devices on workers can help to reduce IO time from remote storage nodes. A Presto worker caches remote data in its original form (compressed and possibly encrypted) on local flash upon read. If in the future, there is a read request covering the range that can be found on the local flash, the request will return the result directly. The caching units are with the aligned sizes to avoid fragmentation. For example, if a read request covers range $[2.3MB, 4.5MB)$, Presto will issue a remote read of range $[2MB, 5MB)$ and caches as well as indexes for the blocks of $[2MB, 3MB)$, $[3MB, 4MB)$, and $[4MB, 5MB)$. For any future read that overlaps the range of $[2MB, 5MB)$, the overlapped part will be fetched from the local disk directly. The eviction policy on these caching units is LRU (least recently used).

Fragment Result Cache: Moreover, a task that is running a leaf stage, which is a task that is responsible for pulling data from remote storage, can decide to cache the partially computed results on local flash. This is to prevent duplicated computation upon multiple queries. A typical approach is to cache the plan fragment results on leaf stages with one level of scan, filter, projection, and/or aggregation. For example, users may decide to query the aggregated result of a reporting over the past 1 day. Later on, they could adjust the dashboard to see the aggregated result of the past 3 days. Then for the second query, we could prevent the duplicated computation of the first 1 day by caching the fragment results from the previous query. Only the remaining 2 days' data need scanning and partial aggregation.

Note that the fragment result is based on the leaf query fragment, which could be highly variable as users can adjust filters or projections. To maximize the cache hit rate even when users change the filters or projections frequently, we rely on statistics-based *canonicalization*. The canonicalization first performs an isomorphic mapping from different variable names into fixed ones so that queries with different aliases with the same meaning end up with the same plan. Then, it sorts the expressions so that expressions like $a > b$ and $b < a$ will have the same format. Finally, it prunes predicates in filters. Given a filter ϕ in the form of a conjunction of predicates, *predicate pruning* generates a new filter by removing all satisfied predicates in ϕ. Note that the approach is not limited to conjunctions, other general representations like disjunctions are also applicable. Because each worker only reads part of the data, it can prune more predicates of a filter at runtime than the coordinator at planning time. For a file read by a worker, the worker takes the statistics of the file (usually minima and maxima) to check if the statistics ranges satisfy some of the predicates or not. The worker will remove fully satisfied predicates in the filter or evaluate the entire filter to `False` if any predicate not satisfied.

Metadata Cache and Catalog Servers: Various metadata-level caches are also introduced on coordinators and workers. Hot data like file indexes (which are also called "footers" or "headers" in other contexts) are cached in memory. Mutable metadata like table schemas or file paths is cached with versioning in the coordinators. There is also an option to host the metadata cache in *catalog servers* to further scale the cache. Catalog servers can be a standalone deployment in addition to the coordinator or could be collocated with the coordinator. However, at Meta, we do not use standalone catalog servers to avoid deployment fragmentation.

Cache locality: To maximize the cache hit rate on workers (in either memory or local flash), the coordinator needs to schedule the read requests of the same file to the same worker with a hash function. To avoid hotspot workers, the scheduler will fall back to its secondary picked worker for caching or just skip the cache when necessary. Various hashing policies like simple modular hashing or consistent hashing are available. The same logic is also applied to query routing. As Presto is deployed globally across several data centers, the router will redirect the query to a cluster that has the cached data with hotspot prevention as a fallback.

With all of the above mechanisms implemented, we were able to completely deprecate collocated storage connectors like Raptor [44] and in-memory database like Cubrick [40] at Meta [14] by providing the same or faster query latency. More details, including TPC-H benchmarks, can be found in our blogs [1, 29].

3.2 Native vectorized execution

Presto is written in Java. Not only does this prevent precise memory management but also renders us unable to leverage modern vectorized CPU execution like SIMD. Velox [41] is a project originally incubated from Presto at Meta to support C++ vectorized execution. It later became a general-purpose vectorized execution library that could benefit use cases like machine learning acceleration.

Presto has tight integration with Velox to leverage vectorized execution. To host the C++ library, native C++ workers are built to directly communicate with the coordinator. The shuffle and IO are in native Velox formats so no extra copy is paid to transform into Presto formats. When a query starts, the coordinator will schedule the query plan fragments to C++ workers. The workers receive the plan fragments and translate them into Velox plans. A native thread is spawned on receiving a Velox plan directly inside the C++ workers to fully leverage the memory fungibility.

Within an execution thread of Velox, functions, expressions, and IO are executed in a vectorized fashion. Simple expressions are evaluated once for multiple values through SIMD. Velox has compatible type and function semantics with Presto so that the same function signature can produce the same result on both Java and C++ executions.

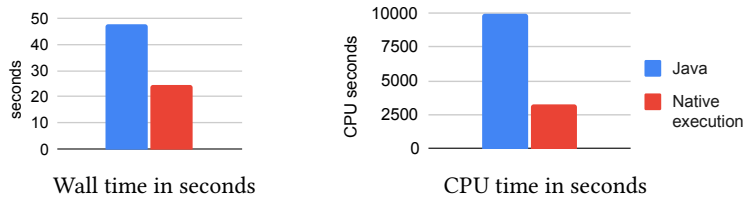


Fig. 3. Native execution acceleration on TPC-H queries

Figure 3 illustrates the average query latency and CPU time on all TPC-H queries with a scale factor of 1000 for both Java and native execution. The benchmark is done on the same cluster with the same number of cores and memory configuration. The overall improvements in latency and CPU are at around 2 - 3X. The more detailed comparison of production data is discussed in Section 7.

3.3 Adaptive filtering

Efficient pruning is important as users can slice and dice the dimensions arbitrarily. This section introduces filtering and pruning techniques newly built in Presto over the last few years.

Subfield Pruning: Complex types like maps, arrays, and structs are widely used in modern data warehouses. Machine learning workloads, for instance, often produce large maps with thousands of embedded features that are stored within table columns. A *subfield* of a complex type instance, denoted as τ, refers to a nested element within τ. As an illustration, if τ is an array type instance, $\tau[2]$ refers to the second subfield of τ. It is important to note that subfields can be recursively nested, based on the types involved. Extracting the subfields effectively without reading the entire complex object is required for CPU efficiency. Presto supports *subfield pruning* by signaling to the reader the needed indices or keys of the complex objects. The reader will skip subfields based on the columnar format like ORC [38] or Parquet [39] to avoid reading the unused subfields. In the previous example of array type instance τ, only $\tau[2]$ is read from the disk; all other indices of τ are skipped. The pruning is recursive to support arbitrary levels of nesting.

Filter reordering: In addition to subfield pruning, filter pushdown is a common strategy to reduce the scan size by applying filtering while scanning so that some of the columns or rows do not have to be materialized even if they are explicitly required in the query plan. In various cases, some filters are more effective than others; they drop more rows in fewer CPU cycles. During runtime, Presto automatically reorders the filters so that the more selective filters are evaluated before the less selective ones. Prior to reading any data, each function within the filter is initialized with (1) a "CPU cycle estimation", which is calculated based on the function's arity and input types, and (2) a fixed selectivity. As the reader begins to scan and filter data, the selectivity of each function is profiled, and the CPU cycle estimation is adjusted to reflect actual CPU cycles. At runtime, the order of functions within the filter is dynamically reordered based on the product of their selectivity and average CPU cycles. As the data changes during the scan, the selectivity and CPU cycles are constantly adjusted to adaptively reorder the filter.

Filter-based lazy materialization: While applying a set of filters in some order for a batch of rows, Presto keeps track of the rows that have satisfied the filter predicates. For the rows that have failed the early filters in that batch, there is no need to evaluate or even materialize the rows of the columns that are needed for the other filters in the same batch. For example, if we are to apply filter "`col1 > 10 AND col2 = 5`" on columns `col1` and `col2`, the scan will first evaluate `col1 > 10` against all the rows in `col1`, which must be materialized. However, only the rows

that pass `col1 > 10` in `col2` need to be materialized for evaluating `col2 = 5`. This is a technique implemented in most modern databases. However, it was not introduced in [44]. The gain of the overall filtering improvement in production is detailed in Section 7.

Dynamic join filtering: In Presto, the filter pushdown can be further enhanced to work with "dynamic join filtering". For an inner join, the build side can provide a "digest" in the format of bloom filters, ranges, or distinct values to serve as a filter for the probe side. The digest can be pushed down through the above framework as an extra filter during the scan, so that the probe side reader will not materialize the data that is not matching the join key. The format of the digest is dependent on the number of distinct values of the build side so the size of the digest should be small and relatively effective on filtering but not "overfitting".

3.4 Materialized views and near real-time data

Data warehouses typically write data for tables with columnar formats in an incremental manner hourly or daily. The written data becomes immutable after the time increment passes. Historically, Presto could only read immutable data. Recently, we have extended the capability to read in-flight data ingested into the warehouse to provide near real-time (NRT) support. At Meta, NRT support is available with a tens of seconds delay from the time data is created.

With the NRT support, More NRT dashboards are being built to reflect more frequent metrics changes. Presto powers the majority of the dashboards at Meta. Rarely do users build dashboards against the raw data which is in general too large to provide a low-latency experience. Pre-computed tables are preferred to reduce cardinality ahead of time. However, such methods do not apply to NRT use cases as data is coming continuously. To satisfy both low-latency requirements as well as data freshness, materialized view functionality is built into Presto.

A *materialized view* is a view represented by a query whose result is stored. When a materialized view is created by Presto, an automatic job will be created to materialize the data for the view. As long as some units (hours or days usually) of the base tables become immutable, the automatic job will run the view query to materialize the view data. The continuous incoming NRT data, on the other hand, will not be materialized for the view until it becomes immutable. When a user queries the materialized view, Presto identifies which part of the view has been materialized and which has not. Presto then breaks the query into a `UNION ALL` query to combine the materialized data as well as the non-materialized fresh data from the base table. This allows the query to provide both freshness as well as low latency due to reduced data size.

Another use case of the materialized view is *subquery optimization*. Given a query, Presto retrieves all the materialized views associated with the queried tables. Presto attempts to match if a materialized view is a subquery of the received one. If there is a match, the received query will be rewritten to leverage the materialized view instead of fetching data from the base tables. The current supported query pattern only allows scan, filter, project, and aggregation. A handful of aggregation functions are supported like `SUM`, `MIN`, `MAX`, `AVG`, `COUNT`, etc.

As materialized view support is only rolled out for pilot users at Meta without general availability, we only showcase the wins from early users. Figure 4 illustrates the improvements with materialized views on one of the largest single-table interactive workloads at Meta. The workload on this table includes all simple aggregation queries on an NRT table, which contains hundreds of billions of rows with half PB compressed size. Five materialized views were created for this table as a result of the most frequently used common subqueries of the entire workload. There is no user-side change as the subquery optimization happens automatically on the engine side. With the materialized views, there are more than 2X reductions in CPU, scanned rows, and latency as on the 90th percentile.

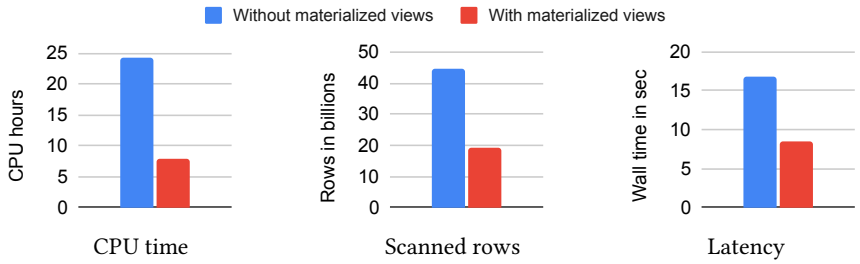


Fig. 4. Subquery optimization with materialized views

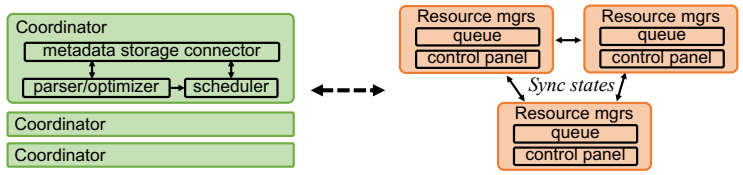


Fig. 5. Multiple coordinators

4 SCALABILITY IMPROVEMENTS

Presto has been leveraged more and more to support heavy ETL jobs. When entering the realm of hours of runtime and PB-size scans, the original Presto architecture will not adequately scale. Various improvements as well as re-architecturing have been integrated into Presto to handle single-point failures, worker crashes, data skews, and memory limitation.

4.1 Multiple coordinators

The coordinator has been a single point of failure for Presto. This is especially a challenge for long-running queries, thousands of which could be queued in the coordinator during peak hours. A crash of the coordinator means all queries will fail. From a scalability aspect, horizontally scaling a coordinator would reach a limit with more queries running in parallel since query scheduling takes a non-trivial amount of memory and CPU. Moreover, Meta infrastructure design is trending toward containers with less memory, currently all query queuing, query scheduling, and cluster management cannot be achieved with smaller memory.

Presto solves this problem by separating the life cycles of queries and clusters. The coordinators only control the life cycles of queries and the newly introduced *resource managers* are in charge of the queueing and resource utilization monitoring of a cluster. Figure 5 demonstrates the topology of multiple coordinator and multiple resource manager architecture, which all originally resided in a single coordinator. A query will first be sent to an arbitrary coordinator. The coordinators are independent of each other without communication among them. The query will then optionally be sent to the resource managers for queueing. The resource managers are highly available. The queued queries and cluster control panel information are replicated across all instances. Consensus protocol like Raft [36] is used to guarantee a crash of a resource manager does not result in any loss of queued queries. Coordinators fetch the queueing information from the resource managers periodically to decide what queries to execute. Using periodical information fetching, if a coordinator finds there is no query queued in the resource manager or if the queries in the queue are low priority, it can decide to execute a newly submitted query to avoid enqueue overhead or network hop latency.

partition 1

col1	col2
1	a
4	b
1	a
7	d
7	b

partition 2

col1	col2
5	b
2	b
2	b
5	e
2	a

partition 3

col1	col2
6	a
6	a
3	c
3	d
3	c

Fig. 6. Example of modular hash partitions

The introduction of multiple coordinators not only eliminates the single point of failure, but also overcomes issues around the elastic capacity and Meta's infrastructure pushing for smaller containers. Now coordinators or resource managers can be deallocated more frequently without having to keep the queueing states for hours. More details can be found in our blog [17].

4.2 Recoverable grouped execution

The Presto architecture with streaming RPC shuffle and in-memory data processing is optimized for latency. However, when it comes to running ETL queries with PB-size scans or hours of runtime, it is neither scalable in memory limitation nor reliable in some guarantee that no worker would crash. To support arbitrarily large queries, we developed *recoverable grouped execution*. (More details can be found in our blog [8]).

In a warehouse, data is usually *partitioned*. For example, data can land by day, and thus "day" is a natural partition. This can also be extended to have other types of partitions like modular hash or z-ordering. Rows with identical *partition keys* (which are represented by table columns) belong to the same partition. Figure 6 shows an example of hash partitions where the table is partitioned on column `col1` with hash function `mod(3)` resulting in 3 partitions.

In Presto, a query can be executed in a "grouped" fashion if the first aggregation, join, or window function key after the table scan is a superset of the data partition key. In such a case, the engine will not scan the entire data set and shuffle based on the aggregation, join, or window function key. It will only scan partition by partition as the keys will be disjoint across partitions. If executing the entire query requires more memory than the cluster can provide, a grouped execution is preferred to lower peak memory consumption. Continuing with the example in Figure 6, suppose a user has a query "`SELECT COUNT() from table1 GROUP BY col1`". A normal scan will read all 3 partitions in parallel and shuffle them based on the aggregation key `col1`. Then, the aggregation stage will receive all 7 distinct values in memory before emitting the final aggregated result. On the contrary, grouped execution will scan one partition at a time. Because the partition key `col1` is the same as the aggregation key `col1` in the query, it will first scan everything in `partition 1` and build the hash table with only 3 distinct values (1, 4, and 7) in memory and emit the final results for the 3 values. Then it will continue the processing with only two values for `partition 2` and 3 each. In such a case, peak memory usage would be smaller than scanning everything in parallel.

The grouped execution can be extended beyond the first shuffle or when the data is not partitioned by the aggregation, join, or window function key. The way to achieve this is by injecting a shuffle phase to materialize the source data in a partitioned way based on the downstream keys. The benefit is to allow grouped execution to apply to arbitrary queries with arbitrary source data. The downside is the overhead of intermediate data materialization.

With the intermediate data materialized, we further built support for failure recovery of grouped execution at the boundary of shuffle points. If a worker crashes, the scheduler will rerun the failed execution directly from the materialized intermediate data instead of from the source. From an architectural perspective, it is also possible to support more fine-grain recovery prior to shuffle

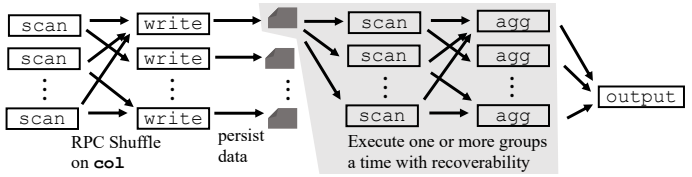


Fig. 7. Recoverable grouped execution

with a fault-tolerant local disk-based or distributed disaggregated shuffle service (for example, Cosco [25] integration).

Figure 7 shows an example of the recoverable grouped execution for query "`SELECT COUNT() FROM table1 GROUP BY col1`" where `table1` contains trillions of distinct values for `col1` that cannot fit into the memory of a whole cluster. To overcome the memory limitation, the first shuffle will be based on `col1`. Instead of directly pipelining the shuffled key into `COUNT` aggregation, writers will persist the data. Then the aggregation phase can have a grouped execution (shown in the gray box) on the shuffled data to lower peak memory consumption. Each grouped execution is recoverable as the immediate data on `col1` has persisted.

4.3 Presto on Spark

Recoverable grouped execution enables Presto to overcome the memory limitation with the support of failure recovery. While the failure recovery boundary is at the shuffle point, which could be too coarse. There are several mature general-purpose data compute engines that have builtin failure recovery mechanisms with finer granularity. Spark [57] is one of them. Spark provides *resilient distributed dataset (RDD)*, which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs automatically recover from container or task failures. *Presto on Spark* is a new architecture that completely gets rid of the existing Presto cluster topology with multi-tenancy. It leverages Presto as a library and runs on top of the Spark RDD interface to provide scalability and reliability for no additional cost.

The Presto on Spark architecture replaces the Presto builtin scheduler, shuffle, resource management, and task execution with Spark ones demonstrated in Figure 8. To start a Presto on Spark query, Spark first launches a simplified Presto coordinator as a library inside its process to parse and optimize the query. The simplified coordinator will then discover all necessary tasks and compile them together with the optimized physical plan into RDD tasks that are sent to Spark for scheduling. An RDD task instance carries the original Presto plan fragment. Once scheduled, the RDD execution thread will run on a simplified Presto worker as a library based on the Presto plan fragment. External shuffle execution will need to be implemented on the worker to leverage the external shuffle service. An external shuffle service can avoid the shortage of RPC shuffle in terms of connection limitation and failure boundary. An RDD thread will be automatically retried by the Spark cluster manager if the container crashes. Note that the original Presto services like coordinator and worker all serve as libraries. These libraries do not communicate with each other or manage memory, threading, or network. All these aspects are removed from the library for simplification and delegated to the Spark clusters.

Note that we only leverage Spark with its RDD level and below. The SparkSQL [6] is not used in this scenario as we need to guarantee the language syntax and semantics consistency of Presto. At Meta, originally both Presto and SparkSQL with Meta internal syntax variation[1] were used to

[1]Hive [47], another well-known ETL engine used at Meta in the 2010s has been completely deprecated by SparkSQL. But Hive's syntax has been kept instead of using the SparkSQL syntax.

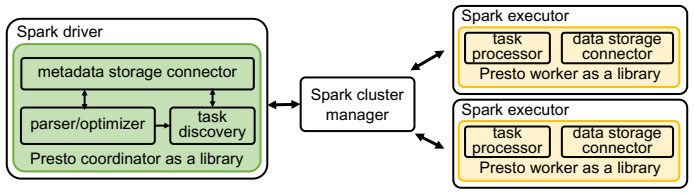


Fig. 8. Presto on Spark architecture

run ETL jobs. However, the language differences between the two led to high user friction. The Presto on Spark project came to unify the stacks with the language semantics from Presto as well as scalability and reliability from Spark.

Both Presto on Spark and recoverable grouped execution aimed to solve scalability and reliability challenges. The recoverable grouped execution still leverages the multi-tenancy mode with more fungibility of memory and CPU. Presto on Spark on the other hand is with container-level isolation that can provide better scalability and reliability. In Meta production, due to the uncertainty of elastic capacity, containers can be drained frequently to balance peak and off-peak usage. Thus, recoverability is a strong requirement for long-running jobs to handle containers offline. More details can be found in our blog [54].

Starting in early 2022, Meta began migrating all SparkSQL workloads to Presto on Spark to unify the SQL interface. The parser, analyzer, optimizer, and operator execution layers of SparkSQL stack will be completely deprecated in light of Presto. Only the Spark RDD interface remains as it serves as a major component for Presto on Spark. We are also working to replace the recoverable grouped execution after running the stack in production for years because it is not as scalable as Presto on Spark. More learning is available in Section 7.

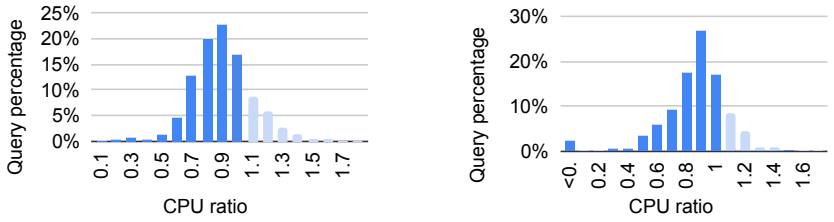
4.4 Spilling

Though Presto has the previous two scalable options to overcome the cluster-wide memory limitation, data skew can still happen, causing a single worker to go beyond the local worker memory limitation. This becomes particularly severe as Meta is moving towards smaller memory-size containers for better elasticity. Spilling is implemented in Presto to materialize the in-memory hash tables for aggregation, join, window function, and topN operators to disk. Having application-level spilling instead of relying on operating systems to swap memory pages to disk helps to have finer control over the query execution. At Meta, interactive and ad-hoc workloads spill data to local flash for latency, and ETL workload spill data to remote storage for scalability.

Once the memory limit is hit when building the hash tables for a query, each hash table will be sorted based on the hash key and serialized to disk. Then the query will continue processing as if the hash table is empty. Once the hash table again grows to the limit, the same process will repeat until all data has been processed. Then, an external merge of these sorted hash tables will be performed to limit the memory usage when emitting the results. Note that the techniques of in-memory hashing and overflow resolution are well known in industry [22, 45].

5 EFFICIENCY IMPROVEMENTS

In addition to the latency and scalability improvements, efficiency is also important to query performance. This section illustrates several enhancements we have made to improve efficiency.



(a) cost-based over default (b) hist-based over cost-based

Fig. 9. CPU ratio of different optimization setup

5.1 Cost-Based Optimizer

Optimizers are essential to query engines. A proper plan can leverage the best use of the resources in a cluster. Presto has a *cost-based optimizer* to assign costs to CPU, IO, and memory to balance these factors to generate an optimized plan. In detail, cost-based optimization is used to make decisions upon (1) join type selection including broadcast join and redistributed join and (2) join reordering to minimize the overall memory usage. To fully leverage the memory yet provide CPU efficiency without exceeding the memory limit is desired. However, for broadcast join, it can also provide lower latency and fewer CPU cycles. So the tradeoff is to minimize the memory usage to a limit to provide the optimized CPU performance. The use case of filter reordering is not covered by the cost-based optimizer as it is determined at runtime discussed in Section 3.3.

To make the right decision, external information is needed to estimate the cost. At Meta, statistics are stored for each table partition to describe the data distribution; here a partition is the definition laid out in Section 4.2. All services, including Presto, that write data into the warehouse are responsible for calculating and publishing the partition statistics to the metadata store. These statistics are dropped with the deletion of their corresponding partition. The common statistics include histogram, total value count, distinct value count, null count, minima, maxima, etc. These statistics can help to estimate the filter selectivity to estimate the cardinalities of the input tables after filters. It also helps to estimate the join table sizes for memory estimation. During planning time, the cost-based optimizer will take the statistics of the input tables and populate the cost estimation from the leaves of the plan to the root and adjust the plan accordingly to generate the minimum cost. Simple heuristics are applied for filter or join selectivity to estimate the cardinalities and sizes of data in the upper part of the plan.

Figure 9 (a) shows the CPU reduction of a production cluster's ETL queries with joins after applying the cost-based optimization. 60% of such queries change plans and reduce their CPU usage. The column chart demonstrates the CPU ratio of the queries having cost-based optimization enabled over the same set of queries having the optimization disabled. The majority of the queries have improved CPU efficiency (indicated by the area with ratio ≤ 1). Though there are queries running with more CPU with cost-based optimization, it does not necessarily mean a regression. For those queries with increased CPU utilization, 83% of them have lowered memory usage.

5.2 History-Based Optimizer

Table statistics in most cases can provide enough information for plan cost estimation. However, the estimation could be off. Also, the filter or join selectivity is unknown ahead of time so the estimation could be increasingly imprecise with more filters embedded in a query. Therefore, Presto also has support for *history-based optimizer*. As Presto is heavily leveraged for ETL jobs at Meta, the queries are highly repetitive and predictable. The idea of the history-based optimizer is to

leverage the precise execution statistics from the previously finished repeated queries to guide the planning of future repeated queries. In addition to the two join strategies mentioned in Section 5.1, the history-based optimizer also has more fine-grained control over the plan including (1) adjusting the shuffle fanout sizes and (2) partial or intermediate aggregation elimination.

When a query plan is generated, the same canonicalization approach mentioned in Section 3.1 is applied to the query plan. (Note that the predicate pruning in Section 3.1 is at file level on workers. For optimizers, only table-level statistics are available). Then, the constants of the plan are replaced with symbols. The "symbolic plan" will serve as the key for external statistics storage together with the value to be the actual execution statistics after the query is finished. When a query with the same structure but different constants is scheduled, the cost estimation will be directly fetched from the external statistics storage with the same symbolic plan. Because ETL queries only change "date" constant day to day, the statistics provided by the symbolic plan generated previously can be precise for the latest ETL processing.

Figure 9 (b) demonstrates the CPU reduction of the ETL queries of a production cluster similar to the one in Figure 9 (a). We compare the queries having history-based optimization enabled over the same set of queries having only cost-based optimization enabled. The queries are arbitrary ones beyond join queries as history-based optimization provides improvement for generic queries. 25% of the queries have plans changed with an overall 10% CPU improvement.

5.3 Adaptive execution

Statistics are helpful for planners to make decisions. Presto's optimizer strives to statically select the best plan using data statistics as discussed in the previous sections. However, incomplete statistics, assumptions about the data (uniformity assumption, lack of information about data correlations and skew), and complex queries (for example, complex functions or multi-way joins) lead to suboptimal plans. Therefore, *adaptive execution* is needed to dynamically adjust the query plan if during runtime the plan is not optimal.

Adaptive execution leverages the finished tasks to report the statistics back to the coordinator so that the coordinator can use them to re-optimize the plan for downstream tasks. The types of optimization are a superset of the ones supported by the history-based optimizer in Section 5.2; adaptive execution also provides skew handling for join and aggregation. This is mainly because detecting the skewed keys at runtime does not require any external knowledge as many metadata store do not have the proper support of providing the skewed value for tables or columns.

To leverage the runtime statistics, the scheduler schedules tasks in a phased manner from the scan tasks all the way up to the root. Once the upstream tasks are finished, the optimizer will have a rerun based on the newly collected statistics and the downstream tasks will be scheduled based on the new plan. As the original Presto architecture shuffles data in a streaming fashion, adaptive execution is only available for Presto on Spark mode when phased execution and disaggregated shuffle are supported.

6 ENABLING RICHER ANALYTICS

In addition to the latency, scalability, and efficiency improvements for analytical workload, there is a growing trend at Meta of emphasizing machine learning feature engineering use cases, increasing support for privacy requirements, and graph analytics. This section discusses the support for various such use cases.

6.1 Handling mutability

Data warehouses historically only support immutable data. In recent years, we have seen an increasing trend of mutable data support with versioning. Examples are Delta Lake [5], Iceberg [28],

and Hudi [27]. Presto integrates with all these table formats. However, they are not sufficient for use cases within Meta.

There are two major use cases for mutability at Meta: (1) machine learning feature engineering and (2) row-level deletion for privacy. For (1), feature engineering is the process of using domain knowledge to extract useful information in the form of *features* that can be consumed by machine learning algorithms. At Meta, such processes can be done through analytical engines like Presto or streaming engines with declarative languages (with SQL as an example) by generating features from raw data. Machine learning engineers will keep exploring the data to find the proper features to improve the machine learning models. Before a feature is selected for a model, a candidate feature is logged and associated with the main table. Based on the training result, the candidate feature could be merged into the main table or dropped. There could be hundreds of exploratory candidate features being developed at the same time. Frequent change in the main table schema is not ideal. Thus, a more flexible way of mutating the columns is needed. For (2), Meta users (including Facebook, Instagram, and WhatsApp) can choose not to have their personal data collected for content recommendation or other uses. Meta needs the ability to remove user data in response to users' decisions. Warehouse tables are at EB scales. It is not feasible to rewrite these tables over and over again at high frequency. Thus, a mutable solution for these immutable data is needed.

To solve the above problems, *delta* is built into Presto. Delta is a solution inside Meta that allows mutation of tables with the flexibility of adding or moving columns or rows. Delta associates one or more "delta files" to a single main file. The delta files serve as a change log to the main file to indicate either that there are new columns added or removed or new rows added or removed from the main file. Both main files and delta files are aligned with the same logical row count to recover the logical data from the physical representation. When Presto reads the main file, it will launch extra readers to merge these delta files to reflect the changes. The association and order of the delta files are kept in the metadata store with versioning. Delta files enable logical deletes for the warehouse to satisfy privacy requirements. These delta files are compacted into the main file on a regular basis to avoid read overhead. This process ensures all corresponding physical bits are removed.

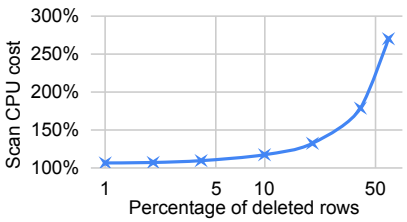


Fig. 10. Deletion overhead

In such a case, machine learning candidate features can be modeled as extra delta columns and user data removal can be modeled as delta rows to be deleted. Any new candidate features added or some users removing personal data will result in new delta files to be associated with the main file in sequence. Note that given personal data removal activities happen frequently, batching of these row removals is needed to avoid creating too many delta files.

Delta file merge for column addition or removal does not affect scan performance as the file formats are columnar. However, for row deletions, performance will be impacted. Figure 10 shows the performance impact of scan CPU when merging delta files during read in production. The x-axis shows the percentage of row counts deleted and the y-axis represents the CPU cost compared to the one without deletion. When only 1% of the rows need to be deleted, an additional CPU cost of 6% is incurred. However, if 60% of the rows need to be deleted, the cost can increase significantly to 170%.

6.2 User-defined types

User-defined types are allowed in Presto to enrich the semantics. Types can be defined in hierarchies with inheritance. For example, a `ProfileId` type can be defined based on `Long` type with both `UserId` and `PageId` types to be its subtypes. The user-defined type definitions are stored in the remote metadata store. In addition to the type definitions themselves stored, extra information can be associated with user-defined types. Examples are constraints expressed by SQL expressions. This allows data quality checks at runtime. For example, one would not expect a `UserId` to be a negative integer or exceed a certain length. Another example is the policy specification, which relates to the growing requirements for privacy. There are common requirements in recent years around user data protection, anonymization, and deletion. To achieve this goal, a prerequisite is to identify the user data in the warehouse. User-defined types allow business domain experts to model their data to reflect the user data in the tables as well as associate the privacy policy with them. For example, a table owner can define an `Email` type that should be anonymized immediately when landed and deleted after 7 days. The warehouse can apply these policies in the background to comply with privacy requirements.

6.3 User-defined functions

User-defined functions (UDFs) allow embedding customized logic into SQL. There are various ways that UDFs are supported in Presto.

In-process UDF: The basic support is the in-process UDFs. Functions are authored and published in the forms of libraries. Presto loads the library at runtime and executes them in the same process as the main evaluation engine. This mode can be efficient as there is no context switch. However, it is only supported by Presto on Spark as the function libraries contain arbitrary code that is not safe to run in multi-tenancy mode.

UDF service: To support UDF in multi-tenancy mode or in different programming languages, Presto has built *UDF servers*. The functions are invoked in remote servers with RPCs from Presto clusters. The UDF servers update functions frequently (in minutes to hours) so the function release cadence can be much faster than the Presto engines. Because an expression can contain both local executable functions as well as remote UDFs, during the compile time, an expression will be decomposed into local executable and remote executable with different projection phases in the plan. The local executable expressions are compiled into bytecode for fast execution; while the remote ones are executed in the UDF servers.

SQL functions: Though UDFs provide flexibility, it is necessary for auditing and privacy purposes that a query should be able to be "reasoned" about without a blackbox of execution. To balance between expressiveness and reasonability, *SQL functions* are introduced. When a function logic can be expressed by SQL, we allow users to define SQL functions to simplify the query logic by avoiding writing long and hard-to-read SQL statements. A SQL function is a piece of SQL code with input and output types well defined. SQL function definitions are also stored in the remote metadata store. SQL functions will be automatically compiled and optionally inlined during execution. A detailed breakdown of how SQL functions work has been published on our blog [50].

6.4 Graph extensions

Graph datasets arise naturally in several use cases at Meta, ranging from social networks to lineage graphs representing how data flows through systems. While users have leveraged specialized systems for graph querying, like graph databases [2, 9, 35, 48] and graph analytics engines [15, 19, 56], we could leverage Presto for many such workloads allowing us to consolidate these specialized

engines on top of Presto. This consolidation has resulted in multiple benefits, like providing a common frontend for users and allowing us to run graph workloads on shared infrastructure.

Supporting graph workloads on Presto is challenging for two main reasons. First, expressing graph queries using vanilla SQL means performing graph traversals via joins, which is unintuitive, error-prone, and often impractical due to the complexity. Second, graph traversal queries are iterative and stateful in nature (for example, the vertex to visit next depends on the vertices already visited), typically resulting in queries with many, large joins that challenge Presto's ability to optimize execution and scale to large graphs.

To address these challenges, we extended Presto SQL with graph querying *language constructs*, inspired by existing graph query languages [21, 26, 31, 51]. These language constructs open up graph querying to more people by providing a declarative interface familiar to SQL users as opposed to making users learn graph specific programming frameworks. Moreover, we built a graph query planner that incorporates graph-specific optimizations to execute iterative queries on the Presto runtime efficiently.

```
SELECT vertices(path) FROM GRAPH g
MATCH (src:Vertex)-/ path:Edge{1,5}/ -> (dst:Vertex)
WHERE g.date = '2022-09-22' AND src.id IN (1,2,3)
AND all_match(edges(path), e -> e.property = TRUE)
```

Listing 1. Example query with graph extension

The example query in Listing 1 covers a few of the features that we have incorporated into the language. First, the `FROM GRAPH` clause does not reference a table, but rather references a "graph". This is a new metadata artifact that we introduced in the warehouse at Meta, which contains a mapping from the schema of a graph (the vertex or edge types, as well as the names and types of their properties), to underlying tables where the graph is stored. We omit details about the way users specify and store graph artifacts, as this is outside the scope of this paper.

In most cases, queries on graph artifacts aim to compute a set of paths in a graph. We use the `MATCH` syntax for specifying a visual pattern which provides a template for the paths we want to query. Parentheses like "`(src:Vertex)`" are used to specify vertices and "`->`" arrows with labels like "`/:Edge/`" to specify edges and their direction. The above example computes paths from vertices `src` to vertices `dst` with a `path` of length at least 1 and at most 5. The output of graph queries is a table where each row is a path. The `WHERE` clause inherits the standard SQL predicate semantics for filtering the computed paths. We use graph-specific functions, alongside existing Presto functions to reference complex predicates on top of the `path` array with expressions like "`all_match(edges(path), e -> e.property = TRUE)`". In the same example, `vertices(path)` in the `SELECT` clause returns an array containing all of the vertex objects in the path in the order they are found.

The high-level expressivity enabled by these language extensions provides the opportunity for graph-specific optimizations. Under the hood, a graph query is parsed into a special *graph logical plan* that is then optimized leveraging the semantics of the graph query. Eventually, the optimized graph logical plan is translated to a relational plan that is executed like with any Presto query. Below, we describe some of these optimizations.

Multi-step execution: A naive implementation of a query like Listing 1 translates to a relational query with as many joins as the maximum length of the path. Such queries may reach the memory limitations of Presto, especially when there are too many paths to compute. To address this, we have implemented an optimization that translates a graph query plan into a series of smaller Presto query plans. Each smaller query plan computes paths up to a certain length and stores them into a

temporary intermediate table, which is used to then continue extending the paths. This keeps each iteration within the memory limitations.

Efficient path extension: Taking Listing 1 again, a naive plan would compute paths of length 1, 2, and so on, and conducts a `UNION ALL` over them. This results in redundant computation. Computing paths of length N requires the same work as computing paths of length $N-1$, plus the work to extend them to paths of length N. However, it is not straightforward for the Presto optimizer to eliminate the redundant work in a plan like this in a general way. Instead, in the query plan we produce, once we have computed paths of length $N-1$, we generate two copies of each path. We then extend one of the copies to paths of length N, keeping the other copy around, effectively reusing computation.

Efficient subgraph computation: Given a set of vertices V, we define a *subgraph* as a subset of the graph consisting of only edges that are reachable from any of the vertices in V. Computing paths versus subgraphs have different requirements. For example, when computing a subgraph, there is no need to track paths and extend them by joining the edge table. We just need to track the edges that have been visited. This allows the subgraph computation plan to scan the edge table from storage once, and then operate on it by marking edges as visited if they can be extended, minimizing IO.

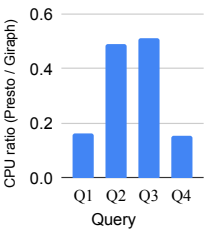


Fig. 11

Complex filter pushdown: Users can specify filters on the paths with functions like `all_match`, which allows for specifying arbitrary predicates that apply to all elements of the input paths. For example, Listing 1 only queries paths where all edges have `property=TRUE`. This predicate is hard for the current general-purpose Presto optimizer to push down. Instead, graph semantic information allows us to directly push these filters down after every join, each of which computes the next hop, thus minimizing the number of intermediate paths that are computed.

We demonstrate the practicality and efficiency of running graph workloads on top of Presto with a benchmark . We compare Presto against Apache Giraph [15], an engine designed specifically for batch graph analytics workloads at Meta. Note that part of Giraph functionality is also undergoing deprecation in light of these Presto graph extensions. Figure 11 showcases the efficiency of the two engines in terms of CPU ratio running on Presto over Giraph. We have 4 graph queries executed to illustrate the CPU gains of using Presto. Queries Q1-3 compute a set of paths starting from a specific vertex. The connectivity of the vertex is high resulting in an exponential increase in the number of paths. Q1 computes all paths up to hop 10, Q2 up to hop 15, and Q3 up to hop 20. In total, after 20 hops we compute 1.2 billion paths. The majority of these paths are found between hops 10 and 16. Q4 is a query that computes a "subgraph" (set of reachable edges downstream of a given vertex) over a larger graph. Regarding the observed performance, Giraph allows a high degree of customizability in the implementation of each algorithm, which can be a double-edged sword. Even though it enables high levels of job-specific optimization, this also makes it more prone to inefficiencies introduced by custom code. Presto, on the other hand, has a declarative SQL interface, trading off expressibility with always using highly optimized implementations of each operator (scan, join, aggregation, etc). These declarative graph constructs enable users to express graph analytics logic that we can transparently translate into the SQL required to efficiently execute them.

7 PERFORMANCE IN PRODUCTION

This section illustrates the Meta production workload performance and learnings during the iterative developments introduced in this paper. [12] provides a rough idea of the scale at Meta.

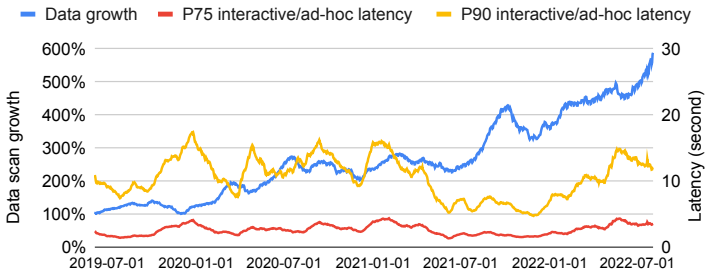


Fig. 12. Interactive/ad-hoc latency with data growth

Given we cannot disclose the detailed numbers, we demonstrate to a point that the effort of this paper has led us to overcome the hyper growth in data volume in the past few years.

7.1 Interactive and ad-hoc workload scalability

Even with increasing data, Presto's pruning, filtering, and caching allow the latency to be consistent to provide the same user experience year after year. Figure 12 illustrates the P75 (75th percentile) and P90 interactive and ad-hoc workload latency as well as the data growth over the past 3 years. The red and yellow series are the P75 and P90 latencies respectively that are relatively stable over the past 3 years. However, if we use the data scan volume in mid of 2019 as the baseline, the scanned data has been growing close to 600% in 3 years leading to a 5X growth. Within the same period, the number of cores added to the interactive cluster fleet is only 82%. Note that Figure 13 demonstrates the latency for both interactive and ad-hoc workload mixed; in general, ad-hoc workload latency is higher and more fluctuated than interactive ones due to their exploratory nature.

7.2 Interactive workload latency

In this section, we compare the interactive workloads at Meta. The entire fleet of such workload has been fully migrated out of the original Presto architecture discussed in [44]. Any collocated in-memory or on-disk storage connectors have also been deprecated in light of the new architecture.

To illustrate the improvement despite the complete deprecation of the original architecture and connectors, we manually set up the cluster with the same cores, threading, and memory as the one in production to shadow production traffic. The four settings on the such cluster to compare are: (1) the original architecture with disaggregated storage [44], (2) caching improvements discussed in Section 3.1 on top of (1), (3) adaptive filtering improvements discussed in Section 3.3 on top of (2), and (4) native vectorized execution integration discussed in Section 3.2 on top of (3).

Figure 13 illustrates the latency comparison of the four settings against the Meta production workload at P75 (75th percentile), P90, and P95. The Y-axis is the latency in seconds. In general, across all percentiles of the execution latency, caching provides about 60% improvement compared to the original architecture. Adaptive filtering adds another 10 - 20%. Another major improvement of over 50% comes from native vectorized execution.

7.3 ETL workload scalability

Similar to the one discussed in Section 7.1, Figure 14 illustrates the data scan footprint for ETL workload in the past 3 years. We also use the data scan volume in mid of 2019 as the baseline. The scanned data has been growing to 450% leading to 3.5X growth. The figure also shows the launch of recoverable grouped execution in mid of 2020 and the launch of Presto on Spark in mid of 2021.

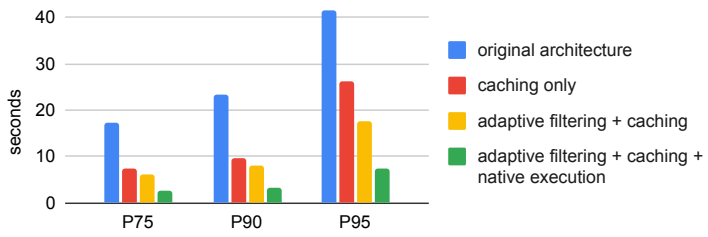


Fig. 13. Interactive workload latency comparison

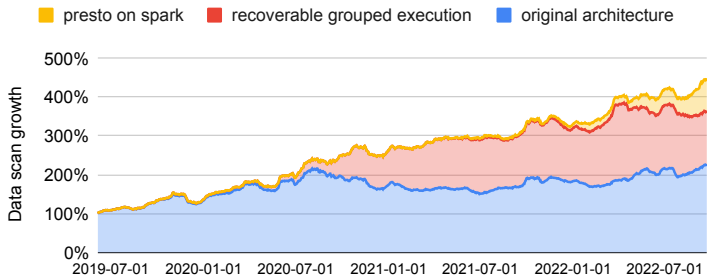


Fig. 14. ETL scan footprint with data growth

After launch, both these two new architectures started to have rapid growth to handle heavier workloads. As mentioned in Section 4, we initially developed recoverable grouped execution as a means of supporting large-scale ETL queries. However, we found that this approach was not as scalable as we had hoped, mainly due to its emphasis on reducing memory consumption and its inadequate fault tolerance for worker crashes. Consequently, with the launch of Presto on Spark, we have seen a decrease in the usage of recoverable grouped execution, as depicted in Figure 14. Users have been rapidly migrating to Presto on Spark, prompting us to begin the deprecation process for recoverable grouped execution.

8 RELATED WORK

Interactive and ad-hoc analytical engines are offered widely by cloud providers. Representative ones include BigQuery powered by Dremel [33, 34], Snowflake [18], and Redshift [7]. Various internal ones are Procella [13] and F1 [43]. Similar techniques like disaggregate storage and caching are also used in these systems.

Regarding analytical SQL batch engines, SparkSQL [6] is a popular open-source engine supporting long-running ETL jobs. SparkSQL, as a SQL evaluation engine, is built on top of Spark [57] which is the general-purpose compute engine. Presto in this paper started directly with a SQL evaluation engine and gradually evolved with fault-tolerance support on top of Spark. F1 [43] is another example of leveraging the interactive engine as a library and running on MapReduce framework [20] to support fault tolerance.

Vectorized engines are an industry trend to boost query performance. Notable ones are DuckDB [42], Photon [10], ClickHouse [16], and Alibaba’s Hologres [30].

Mutability, versioning, and time traveling are supported in various open-source solutions including Delta Lake [5], Iceberg [28], and Hudi [27]. Presto has integration with all these table formats yet still only relies on Meta's solution called "delta" to support more flexible data mutation.

Giraph [15] is an open-source solution to do graph analytics. Part of its functionality has been replaced and migrated to the Presto graph extension. GraphX [56] and GraphFrames [19] are alternative open-source solutions built on top of Spark [57].

There have been many iterations of graph query language syntax solutions over the years, one popular being Cypher [26] used by Neo4j [35]. PGQL [51] is Oracle's vision of such a syntax, and GCORE [3] attempts to formalize the core concepts around building property graph query languages. TigerGraph [48] is another language developed with different syntax. There has been progress made on an ISO standard for property graph query languages called GQL [31], the same way SQL is a standard. Recently, SQL/PGQ [21] has been proposed which will ultimately merge into GQL, all of which are still under active development. Gremlin [49] is an API for querying graphs that follows more of a dataflow structure and differs from declarative SQL-like languages.

9 FUTURE WORK

The techniques mentioned in this paper are our initial exploration of handling more complex workloads. Recoverable grouped execution in Section 4.2 is one example of our early exploration and was later replaced by a more general solution (Presto on Spark in Section Section 4.3). A list of some of the remaining challenges and our most recent attempts at engineering solutions follows.

Non-SQL API: GraphSQL in Section 6.4 is a SQL extension only working for graph-related use cases. We are exploring a general-purpose non-SQL API in Python similar to Snowpark [4] or PySpark [23] to allow execution of control flow on the coordinator and data flow with SQL-equivalent semantics on the workers. The new non-SQL API aims to provide a procedural-like programming experience with richer semantics that could cover graph processing.

Distributed Caching: The caching strategy in Section 3.1 relies on machines having local flash. This is a strong assumption at Meta as compute machines are mostly without disks. We are exploring a remote flash cache strategy directly embedded into Meta's distributed file system. In such a design, caching responsibility can be hidden from Presto. It also provides opportunities for other services using distributed caching beyond a data warehouse.

Unified Container Scheduling: Presto on Spark relies on schedulers to allocate containers for isolation. The current scheduler is a home-built one similar to the open-source offering Yarn [52]. In addition, Meta's streaming engine also relies on its own home-built scheduler [32]. Both of these schedulers have overlapping functionalities with Meta's container solution Tupperware [46] similar to Kubernetes [11]. We are currently prototyping a lightweight model with Tupperware to support fast and frequent container allocation. The new architecture aims to consolidate the scheduling strategy for Presto on Spark, streaming engines, and other general-purpose cluster management.

Unified UDFs: The UDFs in Section 6.3 only support Presto. They cannot be used by machine learning services like training or inference. This has caused friction for users to write multiple versions of UDFs for the same purpose deployed to different services. Machine learning services and Presto are migrating to the Velox execution library mentioned in Section 3.2. We are in the process of extending the UDF offering so that functions will only need to be authored once. This will further defragment the various UDF authoring platforms at Meta.

More Privacy Challenges: In addition to data mutation discussed in Section 6.1, we also face additional privacy-related challenges. One major challenge is query rewrite, which allows users to obtain insight into the data from the warehouse without exposing sensitive data. For example, it is allowed to show the approximate distribution of Facebook's user ages; yet, it is not allowed to show the exact distribution or not to say the individual user's age. This is commonly known as

differential privacy [24]. Unfortunately, various query rewrite techniques to achieve differential privacy mentioned in works like [53] could generate overly complex SQL leading to higher CPU or memory usage. Several explorations have been done at Meta without a successful rollout. Another main challenge is lineage. To understand sensitive data usage, a perfect lineage graph is needed to track how sensitive data is flowed into the warehouse and how it is used. However, customized UDFs, complicated SQL logic, or downloading data out of the warehouse can make tracking hard to achieve. Today, we rely on users to tell the lineage service how data is used and flowed, which is error-prone.

10 CONCLUSIONS

Presto has continued its evolution to handle fast-growing data volume with better latency for interactive workload and better scalability for ETL workload. Various evolutions took place to improve these two. The design principle of supporting both low-latency and long-running queries has considered future data growth instead of doing incremental improvements. Various techniques discussed including caching strategies, vectorized execution, or compiling execution libraries on MapReduce-like framework are well known in the industry. However, to our knowledge, it is the first time a company can illustrate concrete impact by implementing these techniques and open sourcing them with battle-tested quality at Meta scale for community use. Through these efforts, we have successfully consolidated our data warehouse design by centralizing the traditional ETL workload (previously handled by SparkSQL), ad-hoc analysis (previously handled by Presto), interactive serving (previously handled by Raptor or Cubrick), and graph processing (previously handled by Giraph) on Presto. This has eliminated the need for multiple query engines and simplified our data warehouse design. Any new requirements (for example, security or privacy asks) coming to the data warehouse do not need to be implemented in previously fragmented engines. Going forward, one single change in Presto covers all entry points.

ACKNOWLEDGMENTS

We would like to thank Jim Apple, Philip Bell, Leiqing Cai, Naveen Cherukuri, Steve Chuck, Serge Druzkin, Victoria Dudin, Ge Gao, Shrinidhi Joshi, Konstantinos Karanasos, Shaloo Kshetrapal, Jiexi Lin, Eric Liu, Lin Liu, Ryan Lim, Mengdi Lin, Ruslan Mardugalliamov, Guy Moore, Sara Narayan, Daniel Ohayon, Sourav Pal, Pedro Eugenio Rocha Pedreira, Harsha Rastogi, Michael Shang, Chandrashekhar Kumar Singh, Ying Su, Ariel Weisberg, Zhan Yuan, and many others who are or used to work at Meta for their contributions to this paper and Presto. We are very grateful for the contributions from the Presto open-source community. These engineering accomplishments would not have been possible without contributions from Ahana, Alluxio, Uber, Twitter, and many others.

REFERENCES

[1] RaptorX: Building a 10X Faster Presto. 2021. https://prestodb.io/blog/2021/02/04/raptorx.
[2] Oracle Labs PGX: Parallel Graph AnalytiX. 2022. https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html.
[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.
[4] Snowpark API. 2022. https://docs.snowflake.com/en/developer-guide/snowpark/index.html.
[5] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[6] 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, Melbourne, Victoria, Australia*. 1383–1394.

[7] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, 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 *SIGMOD '22: International Conference on Management of Data*. ACM, 2205–2217.

[8] Presto Unlimited: MPP SQL Engine at Scale. 2019. https://prestodb.io/blog/2019/08/05/presto-unlimited-mpp-database-at-scale.

[9] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. 2018. Amazon Neptune: Graph Data Management in the Cloud.. In *ISWC (P&D/Industry/BlueSky)*.

[10] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data*. ACM, 2326–2339.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[12] Meta Data Centers. 2022. https://datacenters.fb.com/.

[13] 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. *Proc. VLDB Endow.* 12, 12 (2019), 2022–2034.

[14] Biswapesh Chattopadhyay, Pedro Eugenio Rocha Pedreira, Sundaram Narayanan, Sameer Agarwal, Yutian Sun, Peng Li, Suketu Vakharia, and Weiran Liu. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *13th Conference on Innovative Data Systems Research, CIDR*.

[15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (2015), 1804–1815.

[16] ClickHouse. 2016. https://clickhouse.com/.

[17] Disaggregated Coordinator. 2022. https://prestodb.io/blog/2022/04/15/disggregated-coordinator.

[18] Benoît 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, SIGMOD Conference 2016*. ACM, 215–226.

[19] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: an integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 2.

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)*. 137–150.

[21] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[22] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. ACM Press, 1–8.

[23] Tomasz Drabas and Denny Lee. 2017. *Learning PySpark*. Packt Publishing Ltd.

[24] Cynthia Dwork. 2006. Differential privacy. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33*. Springer, 1–12.

[25] Cosco: An efficient facebook-scale shuffle service. 2020. https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service.

[26] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445.

[27] Apache Hudi. 2017. https://hudi.apache.org.
[28] Apache Iceberg. 2018. https://iceberg.apache.org.
[29] Avoid Data Silos in Presto in Meta: the journey from Raptor to RaptorX. 2022. https://prestodb.io/blog/2022/01/28/avoid-data-silos-in-presto-in-meta.
[30] 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.
[31] GQL: One Property Query Language. 2022. https://gql.today/.
[32] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1591–1602.
[33] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339.
[34] 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.
[35] Neo4j. 2022. https://neo4j.com/.
[36] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14*. 305–319.
[37] Common Sub-Expression optimization. 2021. https://prestodb.io/blog/2021/11/22/common-sub-expression-optimization.
[38] Apache ORC. 2013. https://orc.apache.org/.
[39] Apache Parquet. 2013. https://parquet.apache.org/.
[40] Pedro Pedreira, Chris Croswhite, and Luis Carlos Erpen De Bona. 2016. Cubrick: Indexing Millions of Records per Second for Interactive Analytics. *Proc. VLDB Endow.* 9, 13 (2016), 1305–1316.
[41] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12, 3372–3384.
[42] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference*. ACM, 1981–1984.
[43] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12 (2018), 1835–1848.
[44] 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 *35th IEEE International Conference on Data Engineering, ICDE*. IEEE, 1802–1813.
[45] Leonard D. Shapiro. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.* 11, 3 (1986), 239–264.
[46] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 787–803.
[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE*. 996–1005.
[48] TigerGraph. 2022. https://www.tigergraph.com/.
[49] Apache Tinkerpop. 2022. https://tinkerpop.apache.org/.
[50] Tutorial: How to Define SQL Functions With Presto Across All Connectors. 2021. https://dzone.com/articles/tutorial-how-to-define-sql-functions-with-presto-a.
[51] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

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

[53] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially private SQL with bounded user contribution. *Proceedings on privacy enhancing technologies* 2020, 2 (2020), 230–250.

[54] Scaling with Presto on Spark. 2021. https://prestodb.io/blog/2021/10/26/Scaling-with-Presto-on-Spark.

[55] Getting Started with PrestoDB and Aria Scan Optimizations. 2020. https://prestodb.io/blog/2020/08/14/getting-started-and-aria.

[56] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES, co-located with SIGMOD/PODS*. CWI/ACM, 2.

[57] 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*.

Received November 2022; revised February 2023; accepted March 2023