Presto: SQL on Everything

Raghav Sethi, Martin Traverso*, Dain Sundstrom*, David Phillips*, Wenlei Xie, Yutian Sun,
Nezih Yigitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte*, Christopher Berner*

Facebook, Inc.

***Abstract*—Presto is an open source distributed query engine that supports much of the SQL analytics workload at Facebook. Presto is designed to be adaptive, flexible, and extensible. It supports a wide variety of use cases with diverse characteristics. These range from user-facing reporting applications with sub-second latency requirements to multi-hour ETL jobs that aggregate or join terabytes of data. Presto's Connector API allows plugins to provide a high performance I/O interface to dozens of data sources, including Hadoop data warehouses, RDBMSs, NoSQL systems, and stream processing systems. In this paper, we outline a selection of use cases that Presto supports at Facebook. We then describe its architecture and implementation, and call out features and performance optimizations that enable it to support these use cases. Finally, we present performance results that demonstrate the impact of our main design decisions.**

***Index Terms*—SQL, query engine, big data, data warehouse**

I. Introduction

The ability to quickly and easily extract insights from large amounts of data is increasingly important to technology-enabled organizations. As it becomes cheaper to collect and store vast amounts of data, it is important that tools to query this data become faster, easier to use, and more flexible. Using a popular query language like SQL can make data analytics accessible to more people within an organization. However, ease-of-use is compromised when organizations are forced to deploy multiple incompatible SQL-like systems to solve different classes of analytics problems.

Presto is an open-source distributed SQL query engine that has run in production at Facebook since 2013 and is used today by several large companies, including Uber, Netflix, Airbnb, Bloomberg, and LinkedIn. Organizations such as Qubole, Treasure Data, and Starburst Data have commercial offerings based on Presto. The Amazon Athena[1] interactive querying service is built on Presto. With over a hundred contributors on GitHub, Presto has a strong open source community.

Presto is designed to be adaptive, flexible, and extensible. It provides an ANSI SQL interface to query data stored in Hadoop environments, open-source and proprietary RDBMSs, NoSQL systems, and stream processing systems such as Kafka. A 'Generic RPC'[2] connector makes adding a SQL interface to proprietary systems as easy as implementing a half dozen RPC endpoints. Presto exposes an open HTTP API, ships with JDBC support, and is compatible with several industry-standard business intelligence (BI) and query authoring tools. The built-in Hive connector can natively read from and write to distributed file systems such as HDFS and Amazon S3; and supports several popular open-source file formats including ORC, Parquet, and Avro.

As of late 2018, Presto is responsible for supporting much of the SQL analytic workload at Facebook, including interactive/BI queries and long-running batch extract-transform-load (ETL) jobs. In addition, Presto powers several end-user facing analytics tools, serves high performance dashboards, provides a SQL interface to multiple internal NoSQL systems, and supports Facebook's A/B testing infrastructure. In aggregate, Presto processes hundreds of petabytes of data and quadrillions of rows per day at Facebook.

Presto has several notable characteristics:

- It is an adaptive multi-tenant system capable of concurrently running hundreds of memory, I/O, and CPU-intensive queries, and scaling to thousands of worker nodes while efficiently utilizing cluster resources.
- Its extensible, federated design allows administrators to set up clusters that can process data from many different data sources even within a single query. This reduces the complexity of integrating multiple systems.
- It is flexible, and can be configured to support a vast variety of use cases with very different constraints and performance characteristics.
- It is built for high performance, with several key related features and optimizations, including code-generation. Multiple running queries share a single long-lived Java Virtual Machine (JVM) process on worker nodes, which reduces response time, but requires integrated scheduling, resource management and isolation.

The primary contribution of this paper is to describe the design of the Presto engine, discussing the specific optimizations and trade-offs required to achieve the characteristics we described above. The secondary contributions are performance results for some key design decisions and optimizations, and a description of lessons learned while developing and maintaining Presto.

Presto was originally developed to enable interactive querying over the Facebook data warehouse. It evolved over time to support several different use cases, a few of which we describe in Section II. Rather than studying this evolution, we describe both the engine and use cases as they exist today, and call out main features and functionality as they relate to these use cases. The rest of the paper is structured as follows. In Section III, we provide an architectural overview, and then dive into system design in Section IV. We then describe some important

*Author was affiliated with Facebook, Inc. during the contribution period.

[1]https://aws.amazon.com/athena

[2]Using Thrift, an interface definition language and RPC protocol used for defining and creating services in multiple languages.

performance optimizations in Section V, present performance results in Section VI, and engineering lessons we learned while developing Presto in Section VII. Finally, we outline key related work in Section VIII, and conclude in Section IX. Presto is under active development, and significant new functionality is added frequently. In this paper, we describe Presto as of version 0.211, released in September 2018.

II. Use Cases

At Facebook, we operate numerous Presto clusters (with sizes up to ∼1000 nodes) and support several different use cases. In this section we select four diverse use cases with large deployments and describe their requirements.

A. Interactive Analytics

Facebook operates a massive multi-tenant data warehouse as an internal service, where several business functions and organizational units share a smaller set of managed clusters. Data is stored in a distributed filesystem and metadata is stored in a separate service. These systems have APIs similar to that of HDFS and the Hive metastore service, respectively. We refer to this as the 'Facebook data warehouse', and use a variant of the Presto 'Hive' connector to read from and write to it.

Facebook engineers and data scientists routinely examine small amounts of data (∼50GB-3TB compressed), test hypotheses, and build visualizations or dashboards. Users often rely on query authoring tools, BI tools, or Jupyter notebooks. Individual clusters are required to support 50-100 concurrent running queries with diverse query shapes, and return results within seconds or minutes. Users are highly sensitive to end-to-end wall clock time, and may not have a good intuition of query resource requirements. While performing exploratory analysis, users may not require that the entire result set be returned. Queries are often canceled after initial results are returned, or use `LIMIT` clauses to restrict the amount of result data the system should produce.

B. Batch ETL

The data warehouse we described above is populated with fresh data at regular intervals using ETL queries. Queries are scheduled by a workflow management system that determines dependencies between tasks and schedules them accordingly. Presto supports users migrating from legacy batch processing systems, and ETL queries now make up a large fraction of the Presto workload at Facebook by CPU. These queries are typically written and optimized by data engineers. They tend to be much more resource intensive than queries in the Interactive Analytics use case, and often involve performing CPU-heavy transformations and memory-intensive (multiple TBs of distributed memory) aggregations or joins with other large tables. Query latency is somewhat less important than resource efficiency and overall cluster throughput.

C. A/B Testing

A/B testing is used at Facebook to evaluate the impact of product changes through statistical hypothesis testing. Much of the A/B test infrastructure at Facebook is built on Presto. Users expect test results be available in hours (rather than days) and that the data be complete and accurate. It is also important for users to be able to perform arbitrary slice and dice on their results at interactive latency (∼5-30s) to gain deeper insights. It is difficult to satisfy this requirement by pre-aggregating data, so results must be computed on the fly. Producing results requires joining multiple large data sets, which include user, device, test, and event attributes. Query shapes are restricted to a small set since queries are programmatically generated.

D. Developer/Advertiser Analytics

Several custom reporting tools for external developers and advertisers are built on Presto. One example deployment of this use case is Facebook Analytics[3], which offers advanced analytics tools to developers that build applications which use the Facebook platform. These deployments typically expose a web interface that can generate a restricted set of query shapes. Data volumes are large in aggregate, but queries are highly selective, as users can only access data for their own applications or ads. Most query shapes contain joins, aggregations or window functions. Data ingestion latency is in the order of minutes. There are very strict query latency requirements (∼50ms-5s) as the tooling is meant to be interactive. Clusters must have 99.999% availability and support hundreds of concurrent queries given the volume of users.

III. Architecture Overview

A Presto cluster consists of a single *coordinator* node and one or more *worker* nodes. The coordinator is responsible for admitting, parsing, planning and optimizing queries as well as query orchestration. Worker nodes are responsible for query processing. Figure 1 shows a simplified view of Presto architecture.

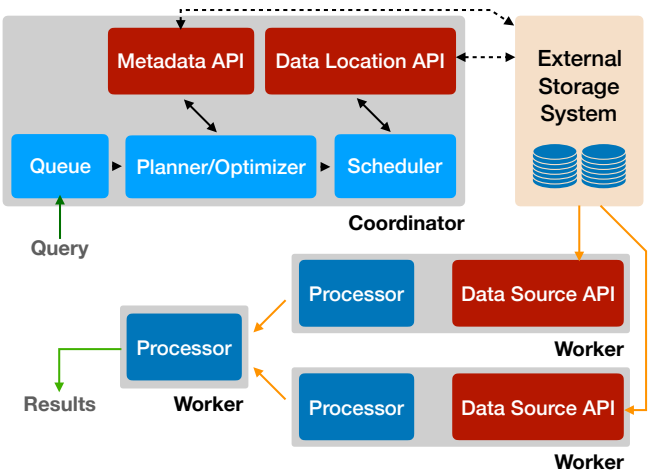

Fig. 1. Presto Architecture

The client sends an HTTP request containing a SQL statement to the coordinator. The coordinator processes the request

[3]https://analytics.facebook.com

by evaluating queue policies, parsing and analyzing the SQL text, creating and optimizing distributed execution plan.

The coordinator distributes this plan to workers, starts execution of *tasks* and then begins to enumerate *splits*, which are opaque handles to an addressable chunk of data in an external storage system. Splits are assigned to the tasks responsible for reading this data.

Worker nodes running these tasks process these splits by fetching data from external systems, or process intermediate results produced by other workers. Workers use co-operative multi-tasking to process tasks from many queries concurrently. Execution is pipelined as much as possible, and data flows between tasks as it becomes available. For certain query shapes, Presto is capable of returning results before all the data is processed. Intermediate data and state is stored in-memory whenever possible. When shuffling data between nodes, buffering is tuned for minimal latency.

Presto is designed to be extensible; and provides a versatile plugin interface. Plugins can provide custom data types, functions, access control implementations, event consumers, queuing policies, and configuration properties. More importantly, plugins also provide *connectors*, which enable Presto to communicate with external data stores through the Connector API, which is composed of four parts: the Metadata API, Data Location API, Data Source API, and Data Sink API. These APIs are designed to allow performant implementations of connectors within the environment of a physically distributed execution engine. Developers have contributed over a dozen connectors to the main Presto repository, and we are aware of several proprietary connectors.

IV. System Design

In this section we describe some of the key design decisions and features of the Presto engine. We describe the SQL dialect that Presto supports, then follow the query lifecycle all the way from client to distributed execution. We also describe some of the resource management mechanisms that enable multi-tenancy in Presto. Finally, we briefly discuss fault tolerance.

A. SQL Dialect

Presto closely follows the ANSI SQL specification [2]. While the engine does not implement every feature described, implemented features conform to the specification as far as possible. We have made a few carefully chosen extensions to the language to improve usability. For example, it is difficult to operate on complex data types, such as maps and arrays, in ANSI SQL. To simplify operating on these common data types, Presto syntax supports anonymous functions (lambda expressions) and built-in higher-order functions (e.g., transform, filter, reduce).

B. Client Interfaces, Parsing, and Planning

1) Client Interfaces: The Presto coordinator primarily exposes a RESTful HTTP interface to clients, and ships with a first-class command line interface. Presto also ships with a JDBC client, which enables compatibility with a wide variety of BI tools, including Tableau and Microstrategy.

2) Parsing: Presto uses an ANTLR-based parser to convert SQL statements into a syntax tree. The analyzer uses this tree to determine types and coercions, resolve functions and scopes, and extracts logical components, such as subqueries, aggregations, and window functions.

3) Logical Planning: The logical planner uses the syntax tree and analysis information to generate an intermediate representation (IR) encoded in the form of a tree of *plan nodes*. Each node represents a physical or logical operation, and the children of a plan node are its inputs. The planner produces nodes that are purely logical, i.e. they do not contain any information about *how* the plan should be executed. Consider a simple query:

```
SELECT
  orders.orderkey, SUM(tax)
FROM orders
LEFT JOIN lineitem
  ON orders.orderkey = lineitem.orderkey
WHERE discount = 0
GROUP BY orders.orderkey
```

The logical plan for this query is outlined in Figure 2.

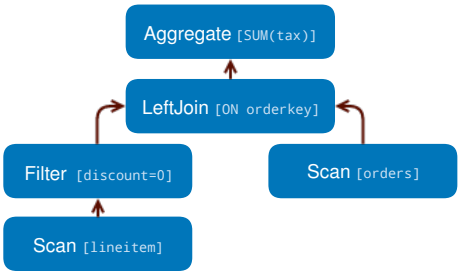

Fig. 2. Logical Plan

C. Query Optimization

The plan optimizer transforms the logical plan into a more physical structure that represents an efficient execution strategy for the query. The process works by evaluating a set of transformation rules greedily until a fixed point is reached. Each rule has a pattern that can match a sub-tree of the query plan and determines whether the transformation should be applied. The result is a logically equivalent sub-plan that replaces the target of the match. Presto contains several rules, including well-known optimizations such as predicate and limit pushdown, column pruning, and decorrelation.

We are in the process of enhancing the optimizer to perform a more comprehensive exploration of the search space using a cost-based evaluation of plans based on the techniques introduced by the Cascades framework [13]. However, Presto already supports two cost-based optimizations that take table and column statistics into account - join strategy selection and join re-ordering. We will discuss only a few features of the optimizer; a detailed treatment is out of the scope of this paper.

1) Data Layouts: The optimizer can take advantage of the physical layout of the data when it is provided by the connector Data Layout API. Connectors report locations and other data properties such as partitioning, sorting, grouping,

and indices. Connectors can return multiple layouts for a single table, each with different properties, and the optimizer can select the most efficient layout for the query [15] [19]. This functionality is used by administrators operating clusters for the Developer/Advertiser Analytics use case; it enables them to optimize new query shapes simply by adding physical layouts. We will see some of the ways the engine can take advantage of these properties in the subsequent sections.

2) Predicate Pushdown: The optimizer can work with connectors to decide when pushing range and equality predicates down through the connector improves filtering efficiency.

For example, the Developer/Advertiser Analytics use case leverages a proprietary connector built on top of sharded MySQL. The connector divides data into shards that are stored in individual MySQL instances, and can push range or point predicates all the way down to individual shards, ensuring that only matching data is ever read from MySQL. If multiple layouts are present, the engine selects a layout that is indexed on the predicate columns. Efficient index based filtering is very important for the highly selective filters used in the Developer/Advertiser Analytics tools. For the Interactive Analytics and Batch ETL use cases, Presto leverages the partition pruning and file-format features (Section V-C) in the Hive connector to improve performance in a similar fashion.

3) Inter-node Parallelism: Part of the optimization process involves identifying parts of the plan that can be executed in parallel across workers. These parts are known as 'stages', and every stage is distributed to one or more tasks, each of which execute the same computation on different sets of input data. The engine inserts buffered in-memory data transfers (shuffles) between stages to enable data exchange. Shuffles add latency, use up buffer memory, and have high CPU overhead. Therefore, the optimizer must reason carefully about the total number of shuffles introduced into the plan. Figure 3 shows how a naïve implementation would partition a plan into stages and connect them using shuffles.

Data Layout Properties: The physical data layout can be used by the optimizer to minimize the number of shuffles in the plan. This is very useful in the A/B Testing use case, where almost every query requires a large join to produce experiment details or population information. The engine takes advantage of the fact that both tables participating in the join are partitioned on the same column, and uses a co-located join strategy to eliminate a resource-intensive shuffle.

If connectors expose a data layout in which join columns are marked as indices, the optimizer is able to determine if using an index nested loop join would be an appropriate strategy. This can make it extremely efficient to operate on normalized data stored in a data warehouse by joining against production data stores (key-value or otherwise). This is a commonly used feature in the Interactive Analytics use case.

Node Properties: Like connectors, nodes in the plan tree can express properties of their outputs (i.e. the partitioning, sorting, bucketing, and grouping characteristics of the data) [24]. These nodes have the ability to also express *required* and *preferred* properties, which are taken into account when introducing shuffles. Redundant shuffles are simply elided, but in other cases the properties of the shuffle can be changed to reduce the number of shuffles required. Presto greedily selects partitioning that will satisfy as many required properties as possible to reduce shuffles. This means that the optimizer may choose to partition on fewer columns, which in some cases can result in greater partition skew. As an example, this optimization applied to the plan in Figure 3 causes it to collapse to a single data processing stage.

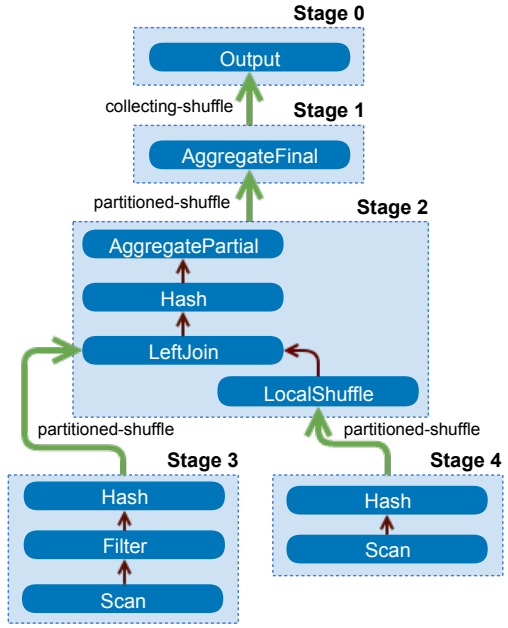

Fig. 3. Distributed plan for Figure 2. The connector has not exposed any data layout properties, and shuffle reduction optimizations have not been applied. Four shuffles are required to execute the query.

4) Intra-node Parallelism: The optimizer uses a similar mechanism to identify sections within plan stages that can benefit from being parallelized across threads on a single node. Parallelizing within a node is much more efficient than inter-node parallelism, since there is little latency overhead, and state (e.g., hash-tables and dictionaries) can be efficiently shared between threads. Adding intra-node parallelism can lead to significant speedups, especially for query shapes where concurrency constrains throughput at downstream stages:

- The Interactive Analytics involves running many short one-off queries, and users do not typically spend time trying to optimize these. As a result, partition skew is common, either due to inherent properties of the data, or as a result of common query patterns (e.g., grouping by user country while also filtering to a small set of countries). This typically manifests as a large volume of data being hash-partitioned on to a small number of nodes.
- Batch ETL jobs often transform large data sets with little or no filtering. In these scenarios, the smaller number of nodes involved in the higher levels of the tree may be insufficient to quickly process the volume of data generated by the leaf stage. Task scheduling is discussed in Section IV-D2.

In both of these scenarios, multiple threads per worker per-

forming the computation can alleviate this concurrency bottleneck to some degree. The engine can run a single sequence of operators (or *pipeline*) in multiple threads. Figure 4 shows how the optimizer is able to parallelize one section of a join.

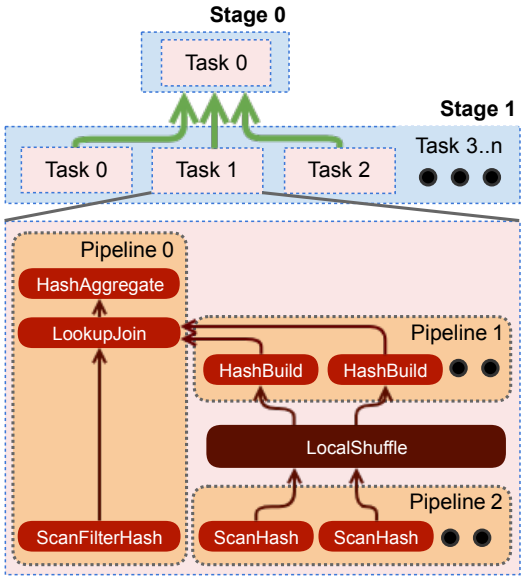

Fig. 4. Materialized and optimized plan corresponding to Figure 3, showing tasks, pipelines, and operators. Pipeline 1 and 2 are parallelized across multiple threads to speed up the build side of a hash-join.

D. Scheduling

The coordinator distributes plan stages to workers in the form of executable tasks, which can be thought of as single processing units. Then, the coordinator links tasks in one stage to tasks in other stages, forming a tree of processors linked to one another by shuffles. Data streams from stage to stage as soon as it is available.

A task may have multiple pipelines within it. A pipeline consists of a chain of *operators*, each of which performs a single, well-defined computation on the data. For example, a task performing a hash-join must contain at least two pipelines; one to build the hash table (build pipeline), and one to stream data from the probe side and perform the join (probe pipeline). When the optimizer determines that part of a pipeline would benefit from increased local parallelism, it can split up the pipeline and parallelize that part independently. Figure 4 shows how the build pipeline has been split up into two pipelines, one to scan data, and the other to build partitions of the hash table. Pipelines are joined together by a local in-memory shuffle.

To execute a query, the engine makes two sets of scheduling decisions. The first determines the order in which stages are scheduled, and the second determines how many tasks should be scheduled, and which nodes they should be placed on.

1) Stage Scheduling: Presto supports two scheduling policies for stages: *all-at-once* and *phased*. All-at-once minimizes wall clock time by scheduling all stages of execution concurrently; data is processed as soon as it is available. This scheduling strategy benefits latency-sensitive use cases such as Interactive Analytics, Developer/Advertiser Analytics, and A/B Testing. Phased execution identifies all the strongly connected components of the directed data flow graph that must be started at the same time to avoid deadlocks and executes those in topological order. For example, if a hash-join is executed in phased mode, the tasks to schedule streaming of the left side will not be scheduled until the hash table is built. This greatly improves memory efficiency for the Batch Analytics use case.

When the scheduler determines that a stage should be scheduled according to the policy, it begins to assign tasks for that stage to worker nodes.

2) Task Scheduling: The task scheduler examines the plan tree and classifies stages into *leaf* and *intermediate* stages. Leaf stages read data from connectors; while intermediate stages only process intermediate results from other stages.

Leaf Stages: For leaf stages, the task scheduler takes into account the constraints imposed by the network and connectors when assigning tasks to worker nodes. For example, shared-nothing deployments require that workers be co-located with storage nodes. The scheduler uses the Connector Data Layout API to decide task placement under these circumstances. The A/B Testing use case requires predictable high-throughput, low-latency data reads, which are satisfied by the Raptor connector. Raptor is a storage engine optimized for Presto with a shared-nothing architecture that stores ORC files on flash disks and metadata in MySQL.

Profiling shows that a majority of CPU time across our production clusters is spent decompressing, decoding, filtering and applying transformations to data read from connectors. This work is highly parallelizable, and running these stages on as many nodes as possible usually yields the shortest wall time. Therefore, if there are no constraints, and the data can be divided up into enough splits, a leaf stage task is scheduled on every worker node in the cluster. For the Facebook data warehouse deployments that run in shared-storage mode (i.e. all data is remote), every node in a cluster is usually involved in processing the leaf stage. This execution strategy can be network intensive.

The scheduler can also reason about network topology to optimize reads using a plugin-provided hierarchy. Network-constrained deployments at Facebook can use this mechanism to express to the engine a preference for rack-local reads over rack-remote reads.

Intermediate Stages: Tasks for intermediate stages can be placed on any worker node. However, the engine still needs to decide how many tasks should be scheduled for each stage. This decision is based on the connector configuration, the properties of the plan, the required data layout, and other deployment configuration. In some cases, the engine can dynamically change the number of tasks during execution. Section IV-E3 describes one such scenario.

3) Split Scheduling: When a task in a leaf stage begins execution on a worker node, the node makes itself available to receive one or more splits (described in Section III). The information that a split contains varies by connector. When reading from a distributed file system, a split might consist of

a file path and offsets to a region of the file. For the Redis key-value store, a split consists of table information, a key and value format, and a list of hosts to query, among other things.

Every task in a leaf stage must be assigned one or more splits to become eligible to run. Tasks in intermediate stages are always eligible to run, and finish only when they are aborted or all their upstream tasks are completed.

Split Assignment: As tasks are set up on worker nodes, the coordinator starts to assign splits to these tasks. Presto asks connectors to enumerate small batches of splits, and assigns them to tasks lazily. This is a an important feature of Presto and provides several benefits:

- Decouples query response time from the time it takes the connector to enumerate a large number of splits. For example, it can take minutes for the Hive connector to enumerate partitions and list files in each partition directory.
- Queries that can start producing results without processing all the data (e.g., simply selecting data with a filter) are frequently canceled quickly or complete early when a `LIMIT` clause is satisfied. In the Interactive Analytics use case, it is common for queries to finish before all the splits have even been enumerated.
- Workers maintain a queue of splits they are assigned to process. The coordinator simply assigns new splits to tasks with the shortest queue. Keeping these queues small allows the system to adapt to variance in CPU cost of processing different splits and performance differences among workers.
- Allows queries to execute without having to hold all their metadata in memory. This is important for the Hive connector, where queries may access millions of splits and can easily consume all available coordinator memory.

These features are particularly useful for the Interactive Analytics and Batch ETL use cases, which run on the Facebook Hive-compatible data warehouse. It's worth noting that lazy split enumeration can make it difficult to accurately estimate and report query progress.

E. Query Execution

1) Local Data Flow: Once a split is assigned to a thread, it is executed by the driver loop. The Presto driver loop is more complex than the popular Volcano (pull) model of recursive iterators [1], but provides important functionality. It is much more amenable to cooperative multi-tasking, since operators can be quickly brought to a known state before yielding the thread instead of blocking indefinitely. In addition, the driver can maximize work performed in every quanta by moving data between operators that can make progress without additional input (e.g., resuming computation of resource-intensive or explosive transformations). Every iteration of the loop moves data between all pairs of operators that can make progress.

The unit of data that the driver loop operates on is called a *page*, which is a columnar encoding of a sequence of rows. The Connector Data Source API returns pages when it is passed a split, and operators typically consume input pages, perform computation, and produce output pages. Figure 5 shows the structure of a page in memory. The driver loop continuously moves pages between operators until the scheduling quanta is complete (discussed in Section IV-F1), or until operators cannot make progress.

2) Shuffles: Presto is designed to minimize end-to-end latency while maximizing resource utilization, and our inter-node data flow mechanism reflects this design choice. Presto uses in-memory buffered shuffles over HTTP to exchange intermediate results. Data produced by tasks is stored in buffers for consumption by other workers. Workers request intermediate results from other workers using HTTP long-polling. The server retains data until the client requests the next segment using a token sent in the previous response. This makes the acknowledgement implicit in the transfer protocol. The long-polling mechanism minimizes response time, especially when transferring small amounts of data. This mechanism offers much lower latency than other systems that persist shuffle data to disk [4], [21] and allows Presto to support latency-sensitive use cases such as Developer/Advertiser Analytics.

The engine tunes parallelism to maintain target utilization rates for output and input buffers. Full output buffers cause split execution to stall and use up valuable memory, while underutilized input buffers add unnecessary processing overhead.

The engine continuously monitors the output buffer utilization. When utilization is consistently high, it lowers effective concurrency by reducing the number of splits eligible to be run. This has the effect of increasing fairness in sharing of network resources. It is also an important efficiency optimization when dealing with clients (either end-users or other workers) that are unable to consume data at the rate it is being produced. Without this functionality, slow clients running complex multi-stage queries could hold tens of gigabytes worth of buffer memory for long periods of time. This scenario is common even when a small amount of result data ($\sim$10-50MB) is being downloaded by a BI or query authoring tool over slow connections in the Interactive Analytics use case.

On the receiver side, the engine monitors the moving average of data transferred per request to compute a target HTTP request concurrency that keeps the input buffers populated while not exceeding their capacity. This backpressure causes upstream tasks to slow down as their buffers fill up.

3) Writes: ETL jobs generally produce data that must be written to other tables. An important driver of write performance in a remote-storage environment is the concurrency with which the write is performed (i.e. the aggregate number of threads writing data through the Connector Data Sink API).

Consider the example of a Hive connector configured to use Amazon S3 for storage. Every concurrent write to S3 creates a new file, and hundreds of writes of a small aggregate amount of data are likely to create small files. Unless these small units of data can be later coalesced, they are likely to create unacceptably high overheads while reading (many slow metadata operations, and latency-bound read performance). However, using too little concurrency can decrease aggregate write throughput to unacceptable levels. Presto takes an adaptive approach again, dynamically increasing writer

concurrency by adding tasks on more worker nodes when the engine determines that the stage producing data for the write exceeds a buffer utilization threshold (and a configurable per-writer data written threshold). This is an important efficiency optimization for the write-heavy Batch ETL use case.

F. Resource Management

One of the key features that makes Presto a good fit for multi-tenant deployments is that it contains a fully-integrated fine-grained resource management system. A single cluster can execute hundreds of queries concurrently, and maximize the use of CPU, IO, and memory resources.

1) CPU Scheduling: Presto primarily optimizes for overall cluster throughput, i.e. aggregate CPU utilized for processing data. The local (node-level) scheduler additionally optimizes for low turnaround time for computationally inexpensive queries, and the fair sharing of CPU resources amongst queries with similar CPU requirements. A task's resource usage is the aggregate thread CPU time given to each of its splits. To minimize coordination overhead, Presto tracks CPU resource usage at the task level and makes scheduling decisions locally.

Presto schedules many concurrent tasks on every worker node to achieve multi-tenancy and uses a cooperative multi-tasking model. Any given split is only allowed to run on a thread for a maximum *quanta* of one second, after which it must relinquish the thread and return to the queue. When output buffers are full (downstream stages cannot consume data fast enough), input buffers are empty (upstream stages cannot produce data fast enough), or the system is out of memory, the local scheduler simply switches to processing another task even before the quanta is complete. This frees up threads for runnable splits, helps Presto maximize CPU usage, and is highly adaptive to different query shapes. All of our use cases benefit from this granular resource efficiency.

When a split relinquishes a thread, the engine needs to decide which task (associated with one or more splits) to run next. Rather than predict the resources required to complete a new query ahead of time, Presto simply uses a task's aggregate CPU time to classify it into the five levels of a multi-level feedback queue [8]. As tasks accumulate more CPU time, they move to higher levels. Each level is assigned a configurable fraction of the available CPU time. In practice, it is challenging to accomplish fair cooperative multi-tasking with arbitrary workloads. The I/O and CPU characteristics for splits vary wildly (sometimes even within the same task), and complex functions (e.g., regular expressions) can consume excessive amounts of thread time relative to other splits. Some connectors do not provide asynchronous APIs, and worker threads can be held for several minutes.

The scheduler must be adaptive when dealing with these constraints. The system provides a low-cost yield signal, so that long running computations can be stopped within an operator. If an operator exceeds the quanta, the scheduler 'charges' actual thread time to the task, and temporarily reduces future execution occurrences. This adaptive behavior allows us to handle the diversity of query shapes in the Interactive Analytics and Batch ETL use cases, where Presto gives higher priority to queries with lowest resource consumption. This choice reflects the understanding that users expect inexpensive queries to complete quickly, and are less concerned about the turnaround time of larger, computationally-expensive jobs. Running more queries concurrently, even at the expense of more context-switching, results in lower aggregate queue time, since shorter queries exit the system quickly.

2) Memory Management: Memory poses one of the main resource management challenges in a multi-tenant system like Presto. In this section we describe the mechanism by which the engine controls memory allocations across the cluster.

Memory Pools: All non-trivial memory allocations in Presto must be classified as user or system memory, and reserve memory in the corresponding memory pool. *User memory* is memory usage that is possible for users to reason about given only basic knowledge of the system or input data (e.g., the memory usage of an aggregation is proportional to its cardinality). On the other hand, *system memory* is memory usage that is largely a byproduct of implementation decisions (e.g., shuffle buffers) and may be uncorrelated with query shape and input data volume.

The engine imposes separate restrictions on user and total (user + system) memory; queries that exceed a global limit (aggregated across workers) or per-node limit are killed. When a node runs out of memory, query memory reservations are blocked by halting processing for tasks. The total memory limit is usually set to be much higher than the user limit, and only a few queries exceed the total limit in production.

The per-node and global user memory limits on queries are usually distinct; this enables a maximum level of permissible usage skew. Consider a 500 node cluster with 100GB of query memory available per node and a requirement that individual queries can use up to 5TB globally. In this case, 10 queries can concurrently allocate up to that amount of total memory. However, if we want to allow for a 2:1 skew (i.e. one partition of the query consumes 2x the median memory), the per-node query memory limit would have to be set to 20GB. This means that only 5 queries are guaranteed to be able to run without exhausting the available node memory.

It is important that we be able to run more than 5 queries concurrently on a 500-node Interactive Analytics or Batch ETL cluster. Given that queries in these clusters vary wildly in their memory characteristics (skew, allocation rate, and allocation temporal locality), it is unlikely that all five queries allocate up to their limit on the same worker node at any given point in time. Therefore, it is generally safe to overcommit the memory of the cluster as long as mechanisms exist to keep the cluster healthy when nodes are low on memory. There are two such mechanisms in Presto – spilling, and reserved pools.

Spilling: When a node runs out of memory, the engine invokes the memory revocation procedure on eligible tasks in ascending order of their execution time, and stops when enough memory is available to satisfy the last request. Revocation is

processed by spilling state to disk. Presto supports spilling for hash joins and aggregations. However, we do not configure any of the Facebook deployments to spill. Cluster sizes are typically large enough to support several TBs of distributed memory, users appreciate the predictable latency of fully in-memory execution, and local disks would increase hardware costs (especially in Facebook's shared-storage deployments).

Reserved Pool: If a node runs out of memory and the cluster is not configured to spill, or there is no revocable memory remaining, the reserved memory mechanism is used to unblock the cluster. The query memory pool on every node is further sub divided into two pools: general and reserved. When the general pool is exhausted on a worker node, the query using the most memory on that worker gets 'promoted' to the reserved pool on all worker nodes. In this state, the memory allocated to that query is counted towards the reserved pool rather than the general pool. To prevent deadlock (where different workers stall different queries) only a single query can enter the reserved pool across the entire cluster. If the general pool on a node is exhausted *while* the reserved pool is occupied, all memory requests from other tasks on that node are stalled. The query runs in the reserved pool until it completes, at which point the cluster unblocks all outstanding requests for memory. This is somewhat wasteful, as the reserved pool on every node must be sized to fit queries running up against the local memory limits. Clusters can be configured to instead kill the query that unblocks most nodes.

G. Fault Tolerance

Presto is able to recover from many transient errors using low-level retries. However, as of late 2018, Presto does not have any meaningful built-in fault tolerance for coordinator or worker node crash failures. Coordinator failures cause the cluster to become unavailable, and a worker node crash failure causes all queries running on that node to fail. Presto relies on clients to automatically retry failed queries.

In production at Facebook, we use external orchestration mechanisms to run clusters in different availability modes depending on the use case. The Interactive Analytics and Batch ETL use cases run standby coordinators, while A/B Testing and Developer/Advertiser Analytics run multiple active clusters. External monitoring systems identify nodes that cause an unusual number of failures and remove them from clusters, and nodes that are remediated automatically re-join the cluster. Each of these mechanisms reduce the duration of unavailability to varying degrees, but cannot hide failures entirely.

Standard checkpointing or partial-recovery techniques are computationally expensive, and difficult to implement in a system designed to stream results back to clients as soon as they are available. Replication-based fault tolerance mechanisms [6] also consume significant resources. Given the cost, the expected value of such techniques is unclear, especially when taking into account the node mean-time-to-failure, cluster sizes of $\sim$1000 nodes and telemetry data showing that most queries complete within a few hours, including Batch ETL. Other researchers have come to similar conclusions [17].

However, we are actively working on improved fault tolerance for long running queries. We are evaluating adding optional checkpointing and limiting restarts to sub-trees of a plan that cannot be run in a pipelined fashion.

V. Query Processing Optimizations

In this section, we describe a few important query processing optimizations that benefit most use cases.

A. Working with the JVM

Presto is implemented in Java and runs on the Hotspot Java Virtual Machine (JVM). Extracting the best possible performance out of the implementation requires playing to the strengths and limitations of the underlying platform. Performance-sensitive code such as data compression or checksum algorithms can benefit from specific optimizations or CPU instructions. While there is no application-level mechanism to control how the JVM Just-In-Time (JIT) compiler generates machine code, it is possible to structure the code so that it can take advantage of optimizations provided by the JIT compiler, such as method inlining, loop unrolling, and intrinsics. We are exploring the use of Graal [22] in scenarios where the JVM is unable to generate optimal machine code, such as 128-bit math operations.

The choice of garbage collection (GC) algorithm can have dramatic effects on application performance and can even influence application implementation choices. Presto uses the G1 collector, which deals poorly with objects larger than a certain size. To limit the number of these objects, Presto avoids allocating objects or buffers bigger than the 'humongous' threshold and uses segmented arrays if necessary. Large and highly linked object graphs can also be problematic due to maintenance of *remembered set* structures in G1 [10]. Data structures in the critical path of query execution are implemented over flat memory arrays to reduce reference and object counts and make the job of the GC easier. For example, the `HISTOGRAM` aggregation stores the bucket keys and counts for all groups in a set of flat arrays and hash tables instead of maintaining independent objects for each histogram.

B. Code Generation

One of the main performance features of the engine is code generation, which targets JVM bytecode. This takes two forms:

1) Expression Evaluation: The performance of a query engine is determined in part by the speed at which it can evaluate complex expressions. Presto contains an expression interpreter that can evaluate arbitrarily complex expressions that we use for tests, but is much too slow for production use evaluating billions of rows. To speed this up, Presto generates bytecode to natively deal with constants, function calls, references to variables, and lazy or short-circuiting operations.

2) Targeting JIT Optimizer Heuristics: Presto generates bytecode for several key operators and operator combinations. The generator takes advantage of the engine's superior knowledge of the semantics of the computation to produce bytecode that is more amenable to JIT optimization than that of a

generic processing loop. There are three main behaviors that the generator targets:

- Since the engine switches between different splits from distinct task pipelines every quanta (Section IV-F1), the JIT would fail to optimize a common loop based implementation since the collected profiling information for the tight processing loop would be polluted by other tasks or queries.
- Even within the processing loop for a single task pipeline, the engine is aware of the types involved in each computation and can generate unrolled loops over columns. Eliminating target type variance in the loop body causes the profiler to conclude that call sites are monomorphic, allowing it to inline virtual methods.
- As the bytecode generated for every task is compiled into a separate Java class, each can be profiled independently by the JIT optimizer. In effect, the JIT optimizer further adapts a custom program generated for the query to the data actually processed. This profiling happens independently at each task, which improves performance in environments where each task processes a different partition of the data. Furthermore, the performance profile can change over the lifetime of the task as the data changes (e.g., time-series data or logs), causing the generated code to be updated.

Generated bytecode also benefits from the second order effects of inlining. The JVM is able to broaden the scope of optimizations, auto-vectorize larger parts of the computation, and can take advantage of frequency-based basic block layout to minimize branches. CPU branch prediction also becomes far more effective [7]. Bytecode generation improves the engine's ability to store intermediate results in registers or caches rather than in memory [16].

C. File Format Features

Scan operators invoke the Connector API with leaf split information and receive columnar data in the form of `Pages`. A page consists of a list of `Blocks`, each of which is a column with a flat in-memory representation. Using flat memory data structures is important for performance, especially for complex types. Pointer chasing, unboxing, and virtual method calls add significant overhead to tight loops.

Connectors such Hive and Raptor take advantage of specific file format features where possible [20]. Presto ships with custom readers for file formats that can efficiently skip data sections by using statistics in file headers/footers (e.g., min-max range headers and Bloom filters). The readers can convert certain forms of compressed data directly into blocks, which can be efficiently operated upon by the engine (Section V-E).

Figure 5 shows the layout of a page with compressed encoding schemes for each column. Dictionary-encoded blocks are very effective at compressing low-cardinality sections of data and run-length encoded (RLE) blocks compress repeated data. Several pages may share a dictionary, which greatly improves memory efficiency. A column in an ORC file can use a single dictionary for an entire 'stripe' (up to millions of rows).

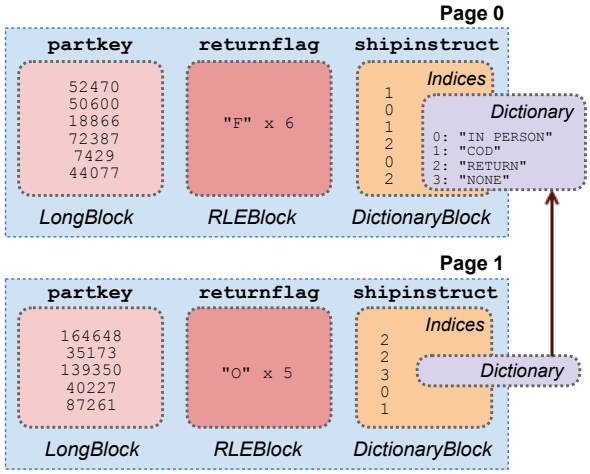

Fig. 5. Different block types within a page

D. Lazy Data Loading

Presto supports lazy materialization of data. This functionality can leverage the columnar, compressed nature of file formats such as ORC, Parquet, and RCFile. Connectors can generate *lazy blocks*, which read, decompress, and decode data only when cells are actually accessed. Given that a large fraction of CPU time is spent decompressing and decoding and that it is common for filters to be highly selective, this optimization is highly effective when columns are infrequently accessed. Tests on a sample of production workload from the Batch ETL use case show that lazy loading reduces data fetched by 78%, cells loaded by 22% and total CPU time by 14%.

E. Operating on Compressed Data

Presto operates on compressed data (i.e. dictionary and run-length-encoded blocks) sourced from the connector wherever possible. Figure 5 shows how these blocks are structured within a page. When a page processor evaluating a transformation or filter encounters a dictionary block, it processes all of the values in the dictionary (or the single value in a run-length-encoded block). This allows the engine to process the entire dictionary in a fast unconditional loop. In some cases, there are more values present in the dictionary than rows in the block. In this scenario the page processor speculates that the un-referenced values will be used in subsequent blocks. The page processor keeps track of the number of real rows produced and the size of the dictionary, which helps measure the effectiveness of processing the dictionary as compared to processing all the indices. If the number of rows is larger than the size of the dictionary it is likely more efficient to process the dictionary instead. When the page processor encounters a new dictionary in the sequence of blocks, it uses this heuristic to determine whether to continue speculating.

Presto also leverages dictionary block structure when building hash tables (e.g., joins or aggregations). As the indices are processed, the operator records hash table locations for every dictionary entry in an array. If the entry is repeated for a subsequent index, it simply re-uses the location rather than re-computing it. When successive blocks share the same

dictionary, the page processor retains the array to further reduce the necessary computation.

Presto also *produces* intermediate compressed results during execution. The join processor, for example, produces dictionary or run-length-encoded blocks when it is more efficient to do so. For a hash join, when the probe side of the join looks up keys in the hash table, it records value indices into an array rather than copying the actual data. The operator simply produces a dictionary block where the index list is that array, and the dictionary is a reference to the block in the hash table.

VI. Performance

In this section, we present performance results that demonstrate the impact of some of the main design decisions described in this paper.

A. Adaptivity

Within Facebook, we run several different connectors in production to allow users to process data stored in various internal systems. Table 1 outlines the connectors and deployments that are used to support the use cases outlined in Section II.

To demonstrate how Presto adapts to connector characteristics, we compare runtimes for queries from the TPC-DS benchmark at scale factor 30TB. Presto is capable of running all TPC-DS queries, but for this experiment we select a low-memory subset that does not require spilling.

We use Presto version 0.211 with internal variants of the Hive/HDFS and Raptor connectors. Raptor is a shared-nothing storage engine designed for Presto. It uses MySQL for metadata and stores data on local flash disks in ORC format. Raptor supports complex data organization (sorting, bucketing, and temporal columns), but for this experiment our data is randomly partitioned. The Hive connector uses an internal service similar to the Hive Metastore and accesses files encoded in an ORC-like format on a remote distributed filesystem that is functionally similar to HDFS (i.e., a shared-storage architecture). Performance characteristics of these connector variants are similar to deployments on public cloud providers.

Every query is run with three settings on a 100-node test cluster: (1) Data stored in Raptor with table shards randomly distributed between nodes. (2) Data stored in Hive/HDFS with no statistics. (3) Data stored in Hive/HDFS along with table and column statistics. Presto's optimizer can make cost-based decisions about join order and join strategy when these statistics are available. Every node is configured with a 28-core Intel™ Xeon™ E5-2680 v4 CPU running at 2.40GHz, 1.6TB of flash storage and 256GB of DDR4 RAM.

Figure 6 shows that Presto query runtime is greatly impacted by the characteristics of connectors. With no change to the query or cluster configuration, Presto is able to adapt to the connector by taking advantage of its characteristics, including throughput, latency, and the availability of statistics. It also demonstrates how a single Presto cluster can serve both as a traditional enterprise data warehouse (that data must be ingested into) and also a query engine over a Hadoop data warehouse. Data engineers at Facebook frequently use Presto to perform exploratory analysis over the Hadoop warehouse and then load aggregate results or frequently-accessed data into Raptor for faster analysis and low-latency dashboards.

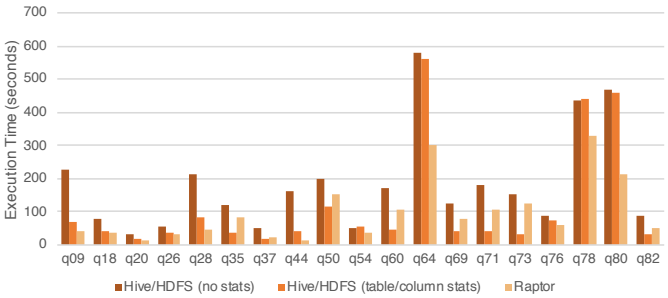

Fig. 6. Query runtimes for a subset of TPC-DS

B. Flexibility

Presto's flexibility is in large part due to its low-latency data shuffle mechanism in conjunction with a Connector API that supports performant processing of large volumes of data. Figure 7 shows a distribution of query runtimes from production deployments of the selected use cases. We include only queries that are successful and actually read data from storage. The results demonstrate that Presto can be configured to effectively serve web use cases with strict latency requirements (20-100ms) as well as programmatically scheduled ETL jobs that run for several hours.

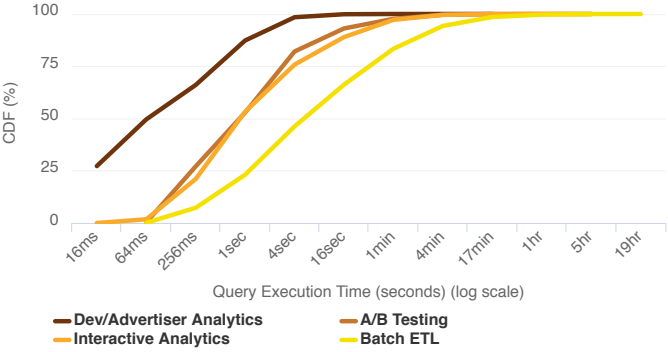

Fig. 7. Query runtime distribution for selected use cases

C. Resource Management

Presto's integrated fine-grained resource management system allows it to quickly move CPU and memory resources between queries to maximize resource efficiency in multi-tenant clusters. Figure 8 shows a four hour trace of CPU and concurrency metrics from one of our Interactive Analytics clusters. Even as demand drops from a peak of 44 queries to a low of 8 queries, Presto continues to utilize an average of ~90% CPU across worker nodes. It is also worth noting that the scheduler prioritizes new and inexpensive workloads as they arrive to maintain responsiveness (Section IV-F1). It does this by allocating large fractions of cluster-wide CPU to new queries within milliseconds of them being admitted.

VII. Engineering Lessons

Presto has been developed and operated as a service by a small team at Facebook since 2013. We observed that some engineering philosophies had an outsize impact on Presto's design through feedback loops in a rapidly evolving environment:

Use Case	Query Duration	Workload Shape	Cluster Size	Concurrency	Connector
Developer/Advertiser Analytics	50 ms - 5 sec	Joins, aggregations and window functions	10s of nodes	100s of queries	Sharded MySQL
A/B Testing	1 sec - 25 sec	Transform, filter and join billions of rows	100s of nodes	10s of queries	Raptor
Interactive Analytics	10 sec - 30 min	Exploratory analysis on up to ~3TB of data	100s of nodes	50-100 queries	Hive/HDFS
Batch ETL	20 min - 5 hr	Transform, filter, and join or aggregate 1-100+TB of input data	Upto 1000 nodes	10s of queries	Hive/HDFS

TABLE I
PRESTO DEPLOYMENTS TO SUPPORT SELECTED USE CASES

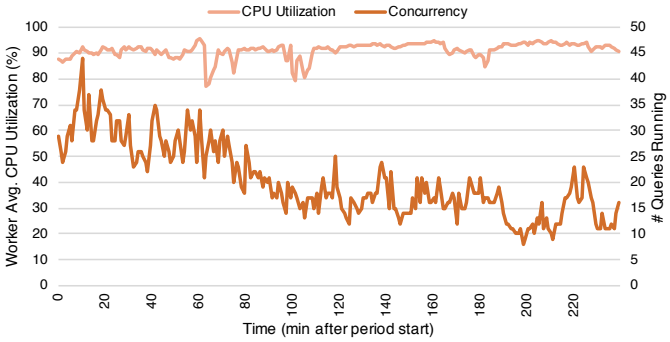

Fig. 8. Cluster avg. CPU utilization and concurrency over a 4-hour period

Adaptiveness over configurability: As a complex multi-tenant query engine that executes arbitrary user defined computation, Presto must be adaptive not only to different query characteristics, but also *combinations* of characteristics. For example, until Presto had end-to-end adaptive backpressure (Section IV-E2), large amounts of memory and CPU was utilized by a small number of jobs with slow clients, which adversely affected latency-sensitive jobs that were running concurrently. Without adaptiveness, it would be necessary to narrowly partition workloads and tune configuration for each workload independently. That approach would not scale to the wide variety of query shapes that we see in production.

Effortless instrumentation: Presto exposes fine-grained performance statistics at the query and node level. We maintain our own libraries for efficient statistics collection which use flat-memory for approximate data structures. It is important to encourage observable system design and allow engineers to instrument and understand the performance of their code. Our libraries make adding statistics as easy as annotating a method. As a consequence, the median Presto worker node exports ~10,000 real-time performance counters, and we collect and store operator level statistics (and merge up to task and stage level) for every query. Our investment in telemetry tooling allows us to be data-driven when optimizing the system.

Static configuration: Operational issues in a complex system like Presto are difficult to root cause and mitigate quickly. Configuration properties can affect system performance in ways that are hard to reason about, and we prioritize being able to understand the state of the cluster over the ability to change configuration quickly. Unlike several other systems at Facebook, Presto uses static rather than dynamic configuration wherever possible. We developed our own configuration library, which is designed to fail 'loudly' by crashing at startup if there are any warnings; this includes unused, duplicated, or conflicting properties. This model poses its own set of challenges. However, with a large number of clusters and configuration sets, it is more efficient to shift complexity from operational investigations to the deployment process/tooling.

Vertical integration: Like other engineering teams, we design custom libraries for components where performance and efficiency are important. For example, custom file-format readers allow us to use Presto-native data structures end-to-end and avoid conversion overhead. However, we observed that the ability to easily debug and control library behaviors is equally important when operating a highly multi-threaded system that performs arbitrary computation in a long-lived process.

Consider an example of a recent production issue. Presto uses the Java built-in `gzip` library. While debugging a sequence of process crashes, we found that interactions between `glibc` and the `gzip` library (which invokes native code) caused memory fragmentation. For specific workload combinations, this caused large native memory leaks. To address this, we changed the way we use the library to influence the right cache flushing behavior, but in other cases we have gone as far as writing our own libraries for compression formats.

Custom libraries can also improve developer efficiency – reducing the surface area for bugs by only implementing necessary features, unifying configuration management, and supporting detailed instrumentation to match our use case.

VIII. RELATED WORK

Systems that run SQL against large data sets have become popular over the past decade. Each of these systems present a unique set of tradeoffs. A comprehensive examination of the space is outside the scope of this paper. Instead, we focus on some of the more notable work in the area.

Apache Hive [21] was originally developed at Facebook to provide a SQL-like interface over data stored in HDFS, and executes queries by compiling them into MapReduce [9] or Tez [18] jobs. Spark SQL [4] is a more modern system built on the popular Spark engine [23], which addresses many of the limitations of MapReduce. Spark SQL can run large queries over multiple distributed data stores, and can operate on intermediate results in memory. However, these systems do not support end-to-end pipelining, and usually persist data to a filesystem during inter-stage shuffles. Although this improves fault tolerance, the additional latency causes such systems to be a poor fit for interactive or low-latency use cases.

Products like Vertica [15], Teradata, Redshift, and Oracle Exadata can read external data to varying degrees. However, they are built around an internal data store and achieve

peak performance when operating on data loaded into the system. Some systems take the hybrid approach of integrating RDBMS-style and MapReduce execution, such as Microsoft SQL Server Polybase [11] (for unstructured data) and Hadapt [5] (for performance). Apache Impala [14] can provide interactive latency, but operates within the Hadoop ecosystem. In contrast, Presto is data source agnostic. Administrators can deploy Presto with a vertically-integrated data store like Raptor, but can also configure Presto to query data from a variety of systems (including relational/NoSQL databases, proprietary internal services and stream processing systems) with low overhead, even within a single Presto cluster.

Presto builds on a rich history of innovative techniques developed by the systems and database community. It uses techniques similar to those described by Neumann [16] and Diaconu et al. [12] on compiling query plans to significantly speed up query processing. It operates on compressed data where possible, using techniques from Abadi et al. [3], and generates compressed intermediate results. It can select the most optimal layout from multiple projections a là Vertica and C-Store [19] and uses strategies similar to Zhou et al. [24] to minimize shuffles by reasoning about plan properties.

IX. Conclusion

In this paper, we presented Presto, an open-source MPP SQL query engine developed at Facebook to quickly process large data sets. Presto is designed to be *flexible*; it can be configured for high-performance SQL processing in a variety of use cases. Its rich plugin interface and Connector API make it *extensible*, allowing it to integrate with various data sources and be effective in many environments. The engine is also designed to be *adaptive*; it can take advantage of connector features to speed up execution, and can automatically tune read and write parallelism, network I/O, operator heuristics, and scheduling to the characteristics of the queries running in the system. Presto's architecture enables it to service workloads that require very low latency and also process expensive, long-running queries efficiently.

Presto allows organizations like Facebook to deploy a single SQL system to deal with multiple common analytic use cases and easily query multiple storage systems while also scaling up to $\sim$1000 nodes. Its architecture and design have found a niche within the crowded SQL-on-Big-Data space. Adoption at Facebook and in the industry is growing quickly, and our open-source community continues to remain engaged.

Acknowledgement

We would like to thank Vaughn Washington, Jay Tang, Ravi Murthy, Ahmed El Zein, Greg Leclercq, Varun Gajjala, Ying Su, Andrii Rosa, Rebecca Schlussel, German Gil, Jiexi Lin, Masha Basmanova, Rongrong Zhong, Shixuan Fan, Elon Azoulay, Timothy Meehan, and many others at Facebook for their contributions to this paper and to Presto. We're thankful to David DeWitt, Nathan Bronson, Mayank Pundir, and Pedro Pedreira for their feedback on drafts of this paper.

We are very grateful for the contributions and continued support of Piotr Findeisen, Grzegorz Kokosiński, Łukasz Osipiuk, Karol Sobczak, Piotr Nowojski, and the rest of the Presto open-source community.

References

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

[2] SQL – Part 1: Framework (SQL/Framework). ISO/IEC 9075-1:2016, International Organization for Standardization, 2016.

[3] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.

[5] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *SIGMOD*, 2011.

[6] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD*, 2005.

[7] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing A SQL Implementation On The MapReduce Framework. In *PVLDB*, volume 4, pages 1318–1327, 2011.

[8] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An Experimental Time-Sharing System. In *Proceedings of the Spring Joint Computer Conference*, 1962.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first Garbage Collection. In *ISMM*, 2004.

[11] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *SIGMOD*, 2013.

[12] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[13] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[14] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[15] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

[16] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[17] A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat, et al. Themis: An I/O-Efficient MapReduce. In *SoCC*, 2012.

[18] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *SIGMOD*, 2015.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[20] D. Sundstrom. Even faster: Data at the speed of Presto ORC, 2015. https://code.facebook.com/posts/370832626374903/even-faster-data-at-the-speed-of-presto-orc/.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.

[22] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *ACM Onward! 2013*, pages 187–204. ACM, 2013.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[24] J. Zhou, P. A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, 2010.