

O'REILLY®

Learning and Operating Presto

Fast Federated SQL Analytics



**Early
Release**

RAW & UNEDITED

Compliments of

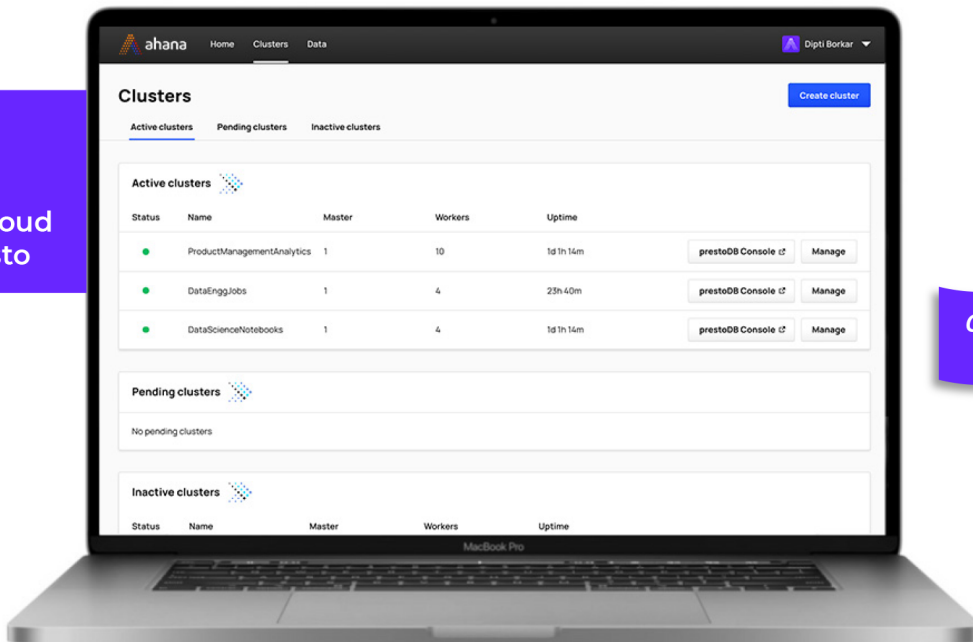


ahana™

Vivek Bharathan,
David E. Simmen
& George Wang

Foreword by Nezh Yigitbasi,

presto 



*0 to Presto in
30 Minutes!*

The Easiest Open Source Presto Experience Ever


Fully Integrated, Cloud-Native Managed Service



Deploy within your AWS account
for complete control of your
Presto clusters and data.

[Get Started](#)



 ahana is proud to be a premier member of the Presto Foundation, hosted under the auspices of The Linux Foundation.

Learning and Operating Presto

Fast Federated SQL Analytics

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Vivek Bharathan, David Simmen, and George Wang

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning and Operating Presto

by Vivek Bharathan, David E. Simmen, and George Wang

Copyright © 2022 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Jonathan Hassell

Development Editor: Gary O'Brien

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media, Inc.

November 2021: First Edition

Revision History for the Early Release

2020-09-16: First Release

2020-11-19: Second Release

2021-01-28: Third Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492095132> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning and Operating Presto*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

Presto and the Presto logo are registered trademarks of *the Linux Foundation*.

Ahana and the Ahana logo are registered trademarks of Ahana Cloud, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly, Ahana, and the Linux Foundation. See our [statement of editorial independence](#).

Table of Contents

Foreword.....	vii
1. Introducing Presto.....	11
Presto Origins	12
What Is Presto?	13
Decoupled Storage and Compute for the Data Analytics Stack	14
Federation of Multiple Backend Data Sources	17
How Presto works	19
Presto query processing explained	22
Presto Operations	24
Presto at Scale	24
Presto in the Cloud	24
Presto as an Analytics Platform	25
Ad hoc querying	25
Reporting and dashboarding	25
ETL using SQL	26
Data lake analytics	26
Real-time analytics with real-time databases	27
Open Source Community	27
Presto Foundation	28
Conclusion	28
2. Operating Presto at Scale.....	31
Common issues when running Presto at scale	31
Presto Coordinator - Single Point of Failure (SPoF)	32
Bad worker state	33
Bad queries	33
Large scale Presto infrastructure in production	34

Using Presto Gateway to improve infrastructure availability	34
Workload Manager - Presto Resource Groups for better resource utilization	36
Query Event Listener framework for better observability	38
Query protection layer to protect the infrastructure from bad actors	41
Choosing node type and JVM settings in production	42
Conclusion	28
Acknowledgement	44
3. Real-time Analytics for Real-time Business Insights: Presto & Apache Pinot	45
Introducing Apache Pinot	46
A closer look at Pinot	46
Why Use Pinot via Presto?	49
Setting up Presto+Pinot	51
Connecting a Pinot cluster to Presto	51
Exposing Pinot tables as Presto tables	52
How Presto queries Pinot	53
Presto-Pinot querying in action	55
Date/Time handling in Pinot vs. Presto	58
Troubleshooting Common Issues	60
Summary	62

Foreword

Back in 2014, I was part of the Big Data Platform team at Netflix and we were faced with the challenge of building a large-scale interactive data analytics platform for the whole company. Netflix was one of the early pioneers to bring a data-driven culture to video entertainment; the various product teams used a data-driven approach to making product decisions. Therefore, my team played a significant part in helping those teams make sense of their multi-petabytes of data in order to gain product and consumer insights—recommendations, user experiences, and audience segmentation.

As an early user of public cloud providers, namely Amazon Web Services (AWS), we had moved our big data platform to an architecture which separated the compute and storage layers. This allowed for a separation of concerns: running compute clusters for the analytic jobs and storing data at the lowest cost per terabyte.

While at that time most of our users were either running Pig jobs, which could take hours, or running interactive jobs on a much smaller subset of data on 3rd party products, their desire for better data-driven insights created the need for low-latency interactive data exploration on our much broader set of data on Amazon S3.

We have evaluated multiple systems at that time with our production workloads in terms of performance and reliability, and Presto, which was initially released to open source around the end of 2013, was the clear winner. To productionize Presto we have worked on improving Presto's AWS & Parquet file format integration in addition to helping make Presto much more reliable, scalable, and also elastic to make it a first class citizen in Netflix's data infrastructure. That epitomizes what is great about open source, when you have an itch, you can scratch it! Since Presto supports ANSI SQL, it was very easy for our analysts and developers to utilize it. The amount of jobs grew quickly, and before we knew it Presto became an integral part of the Netflix data ecosystem. We were one of the first major users outside of Facebook to leverage Presto at scale.

Why This Book Is Important

Since those early days, companies of all sizes and types have sought out ways to become more data driven in their own business. The days of “build it, sell it, and ship it” no longer apply in today’s competitive environment. As the saying goes, “innovate or die” has become a mantra within most companies, causing them to explore larger amounts of data at shorter intervals than ever before. The companies that do this well are considered to be part of a new Data Economy; that is, they are companies that derive significant additional value from their data.

Starting with Facebook, the Presto community has expanded at an astonishing pace. Beyond Netflix, other internet-scale companies like Uber, Twitter, LinkedIn, and JD.com leverage Presto for a wide variety of use cases. In addition, major cloud providers and vendors also have Presto offerings.

Presto is one of the fastest growing open source projects in data analytics today because it fits well with that data-driven paradigm shift. I believe that’s due to three primary reasons: 1) Presto is based on ANSI SQL so it’s easy for people to get running with it, 2) the Presto connector architecture enables the federated access of almost any data source, whether a database, data lake, or other data system, and 3) Presto can start from one node and scale to thousands. As such, there is now an energized community around Presto, having started with the Facebook Presto team, and having grown to hundreds of contributors around the world.

While Presto adoption has been nothing but amazing, there’s still many more people who want to get started with Presto. Presto is a complex distributed system that runs on many machines to process diverse workloads from multiple users, and in a typical deployment it interacts with multiple storage systems. To add to this complexity Presto has plenty of configuration knobs to change its behavior and sometimes to get the best performance, queries or the system itself needs to be tuned carefully. Therefore, it can be challenging even for experienced engineers to ramp up with the Presto architecture and its operations.

That’s where this book comes in. *Learning and Operating Presto* is an approachable on-ramp to using Presto and getting Presto into a production deployment. It is intended to make Presto more accessible to anyone who cares about their organization becoming data driven. Regardless of whether you’re a current user of Presto, new to Presto, or going to provide Presto to your users, this book does an excellent job of explaining what you need to know.

I encourage you to supplement what you learn in this book by participating in Presto’s growing community, which is filled with experienced users, developers, and data architects. Join the PrestoDB Slack channel and attend the meetups and events. Use this book as your entry point into the world of Presto.

Comment on the Authors

Lastly, I'd like to share a few thoughts about the authors, all of whom are passionate about open source and driving innovation for distributed analytics systems. I am thankful to them for taking the time to write this book for the community. In addition to deep expertise in database internals, Vivek Bharathan is a Presto contributor with hands-on experience running large Presto clusters at Uber that processed 35PB of data per day. George Wang previously built an advanced Presto-based cloud service while at Alibaba and is a Presto contributor. And with Dave Simmen's depth in federated databases from his years of pioneering work at Splunk, Teradata Aster and IBM Research, I believe the authors are more than well-equipped to guide your Presto journey.

— *Nezih Yigitbasi, Founding Presto Foundation Technical Steering Committee Chairperson*

Prospective Table of Contents

- Chapter 1: Introducing Presto
- Chapter 2: Getting Started
- Chapter 3: Security
- Chapter 4: Administration
- Chapter 5: Syntax
- Chapter 6: Connectors
- Chapter 7: Top 15 Key Configuration Parameters
- Chapter 8: Clusters
- Chapter 9: Tuning
- Chapter 10: Operating Presto at Scale
- Chapter 11: Troubleshooting: Logs, Error Messages and More
- Chapter 12: Real-time Analytics for Real-time Business Insights: Presto + Apache Pinot
- Chapter 13: Extending Presto

Introducing Presto

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

Every company, whether it’s a startup or a large established business, depends on more data than ever before. As a valuable asset in the company, we collect it, produce it, combine it, and analyze it. We’re now past the big data era. There’s no longer a need for adjectives--it’s just a lot of data. In addition to the growing amount, data is more varied than ever before, with different data types ranging from structured tables to unstructured objects and files. With cloud and edge computing becoming the norm, data is being created in more places. Along with needing to store large amounts of data in many systems comes the need for scalable tools that can query data faster and with a common language, like Structured Query Language (SQL).

Along with having a lot of data in a myriad of forms from a broad spectrum of locations, we are now expected to use it to become data-driven. By exploring and making sense of all the data with SQL, you can extract valuable insights to make better decisions. Backed by data, companies can innovate new ways of interacting with their customers or create new products to better meet their customer needs. As a reader of this book, you’ve probably heard that Presto is a fast, flexible distributed SQL engine

created and used by Facebook at scale. In this book, you'll learn about Presto and how to operate Presto to solve your needs for fast insights on data of any type and any size. In Chapter 1, you'll learn what Presto is, where it came from, and how it is different from other data warehousing solutions. You'll hear about who uses Presto and the ways they solve data challenges with Presto.

Presto Origins

When you were younger perhaps, “Presto” is what you said when you amazed your family and friends with a magic trick. Today, if you're a developer, data platform engineer, data analyst or scientist, Presto is better known as the powerful query engine that helps derive insights from data.

In 2012, Facebook needed a way to provide end users access to enormous data sets to perform ad hoc analysis. At the time, Facebook was using Apache Hive¹ to perform this kind of analysis. As Facebook's data sets grew, Hive was found to not be as interactive (read: too slow) as desired. This is largely because the foundation of Hive is MapReduce, which, at the time, required intermediate data sets to be persisted to disk. This requires a lot of I/O to disk for data for transient, intermediate result sets. So, Facebook developed Presto, a new distributed SQL query engine, designed as an in-memory engine without the need to persist intermediate result sets for a single query. This approach led to a query engine that processed the same query orders of magnitude faster, with many queries completing with less-than-a-second latency. End users such as engineers, product managers, and data analysts found they could interactively query fractions of large data sets to test hypotheses and create visualizations.

In 2013, Facebook open-sourced Presto, opening up the Github repository “prestodb” under the permissive Apache 2.0 license. Early adopters and collaborators included large San Francisco Bay Area companies like Airbnb, Uber, and Netflix. In 2015, Netflix showed that Presto was, in fact, 10 times faster than Hive—and even faster in some cases. Their cluster supported many dozens of concurrent running queries with varying resource requirements. Plotting their real-world production queries against wall clock time revealed that half of their queries ran in less than four seconds. This low-latency trend kept up, with 85% of their queries running in less than a minute. This enabled end users to run ad hoc queries and get answers fast, with iterative exploratory data analysis.

Beyond the early users, hundreds of companies have since presented publicly about using Presto in production. As you'll read further about Presto's flexibility and extensibility, Presto can support a variety of SQL use cases. Organizations typically start with a Presto cluster for their interactive, ad hoc queries and then add other diverse

¹ Hive was also originally developed and open sourced by Facebook.

use cases which we'll cover later in this chapter. For example, at Facebook, Presto started with interactive ad hoc analytics, with latencies of less than one second for many hundreds of concurrent queries. Then they added another use case they call A-B testing. After that, they saw that Presto was so efficient that they could apply it to run batch and Extract-Transform-Load (ETL) workloads.

What Is Presto?

Presto is a distributed SQL query engine written in Java. It takes any query written in SQL, analyzes the query, creates and schedules a query plan on a cluster of worker machines which are connected to data sources, and then returns the query results. The query plan may have a number of execution stages depending on the query. For example, if your query is joining together many large tables, it may need multiple stages to execute, aggregating tables together. After each execution stage there may be intermediate data sets. You can think of those intermediate answers like your scratch-pad for a long calculus problem.

In the past, distributed query engines like Hive were designed to persist intermediate results to disk. As [Figure 1-1](#) illustrates, Presto saves time by executing the queries in the memory of the worker machines, including performing operations on intermediate datasets there, instead of persisting them to disk. The data can reside in HDFS or any database or any data lake, and Presto performs the executions in-memory across your workers, shuffling data between workers as needed. Avoiding the need for writing and reading from disk between stages ultimately speeds up the query execution time.

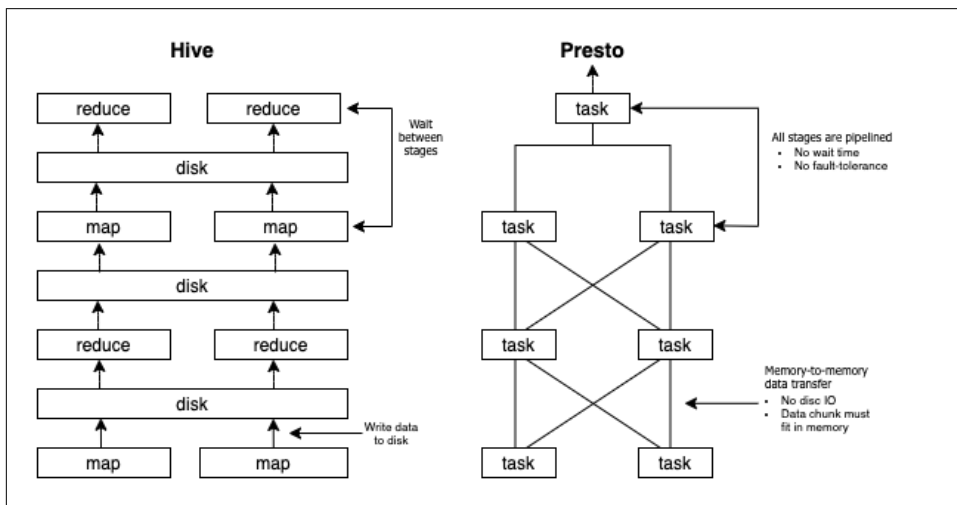


Figure 1-1. Hive intermediate data sets are persisted to disk. Presto executes tasks in-memory.



If this distributed in-memory model sounds familiar, that's because Apache Spark uses the same basic concept to effectively replace MapReduce-based technologies. However, Spark and Presto manage stages differently. In Spark, data needs to be fully processed before passing to the next stage. Presto uses a pipeline processing approach and doesn't need to wait for an entire stage to finish.

Presto was developed with the following design considerations:

- High performance with in-memory execution
- High scalability from 1 to 1000s of workers
- Flexibility to support a wide range of SQL use cases
- Highly pluggable architecture that makes it easy to extend Presto with custom integrations for security, event listeners, etc.
- Federation of data sources via Presto connectors
- Seamless integration with existing SQL systems by adhering to the ANSI SQL standard

"SQL is old, but is not going anywhere. Learn it."

An aphorism amongst people who work with data, SQL has become the lingua franca of data systems. In 1986, the American National Standards Institute (ANSI) adopted SQL as a standard. This is known as the ANSI SQL standard. Even after more than four decades, SQL has stood the test of time and is gaining in popularity. SQL is the most mainstream way to work with any database. It is easy to learn and provides users with a broad interoperability for the majority of databases. Adherence and support of the ANSI SQL standard has been an important characteristic for a federated system like Presto. While there are variants and extensions for SQL, to be compliant with ANSI SQL means that the major, commonly-used commands, like SELECT, UPDATE, DELETE, INSERT, and WHERE, all operate as you'd expect. This means that SQL code that works on other databases will likely work on Presto, out of the box, without any changes to the SQL. This also means that dozens of popular Business Intelligence (BI) tools work with the Presto engine without any additional integration. As most users already know how to write SQL, Presto is easily accessible, and doesn't require any further learning. Presto's SQL compliance immediately enables a large number of use cases.

Decoupled Storage and Compute for the Data Analytics Stack

A traditional data warehouse has multiple layers required for data processing vertically integrated into it. At a high level, this can be broken down into compute and

storage. Data warehouses also control how data is written, where that data resides, and how it is read, as shown in [Figure 1-2](#).

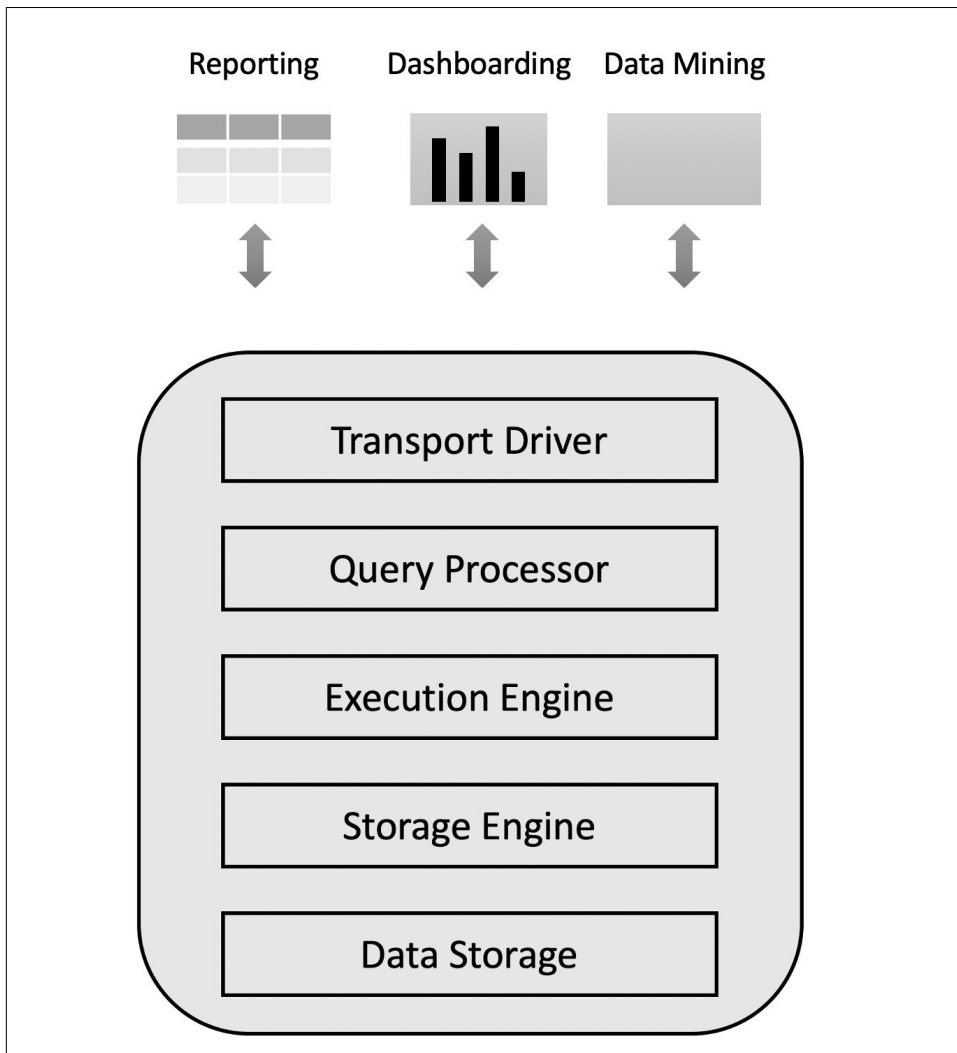


Figure 1-2. Vertically integrated database / data warehouse architecture with compute and storage in one system.

Operational databases and other external sources of data are ingested into the data warehouse server's storage layer, making a second copy which is usually transformed and written in a optimized, but proprietary, format for more efficient analytical processing by the middle tier. Some issues with the vertically integrated data warehouse are the non-linear increase of costs associated with the growing amounts of ingested

data and analytical processing. In addition to needing to ingest all data into one system, the data warehouse may not be designed to handle all analytical processing requirements. In contrast, today's disaggregated systems are designed to scale to handle much larger amounts of data and analytic computation. Presto is one of most popular distributed computing engines. Your data platform team can decouple data storage from data processing, as illustrated in [Figure 1-3](#).

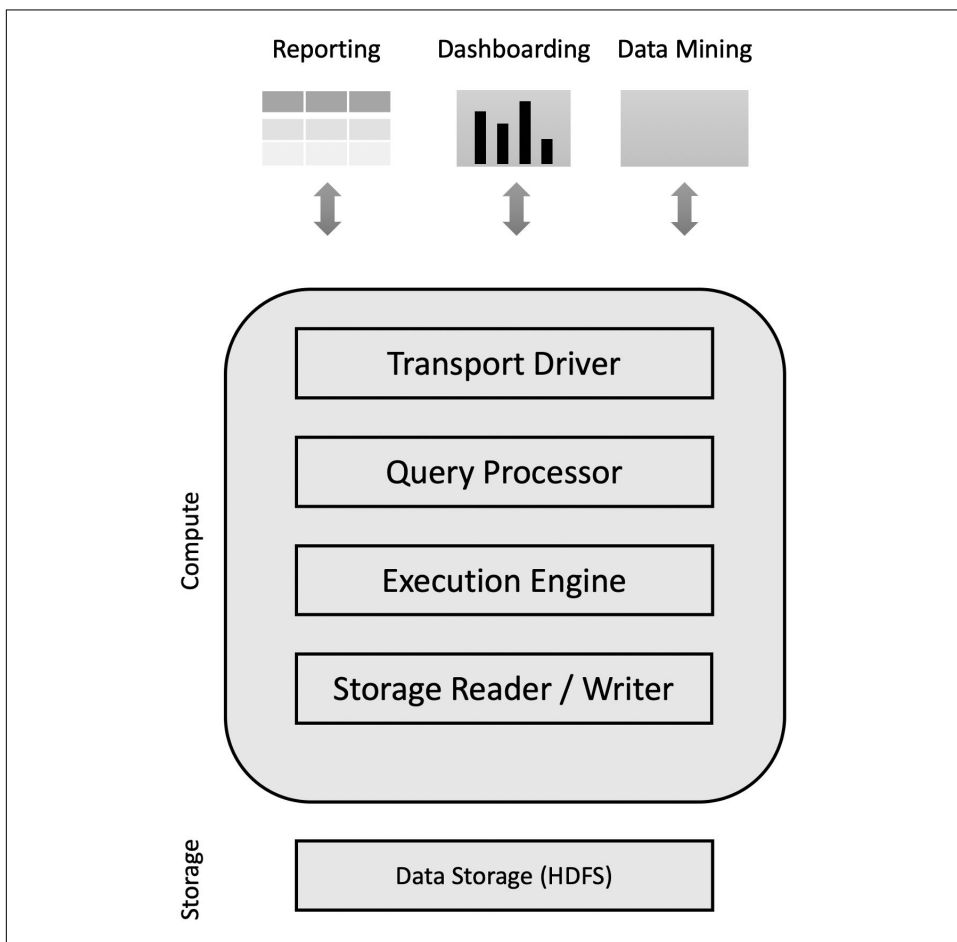


Figure 1-3. Fig. 1-3: Data Storage using HDFS separated from compute for systems like Apache Hive.

A full deployment of Presto has a coordinator and multiple workers. Queries are submitted to the coordinator by a client like the command line interface (CLI), a BI tool, or a notebook that supports SQL. The coordinator parses, analyzes and creates the optimal query execution plan using metadata and data distribution information. That plan is then distributed to the workers for processing.

The advantage of this decoupled storage model is that Presto is able to provide a single view of all of your data that has been aggregated into the data storage tier like Hadoop Distributed File System (HDFS).

Federation of Multiple Backend Data Sources

Federated database architectures are a concept from the late 1980's and early 1990's. This approach is seeing a resurgence due to the greater amount of data, the proliferation of data serialization formats, data lakes and specialized databases (i.e. NoSQL, time-series, metrics, etc.) and end users wanting to access this data on-demand, without necessarily ingesting it all into a common warehouse. Federated databases provide a virtual database abstraction over connected heterogeneous data sources, which can be queried for business insights.



The term federation is a mouthful, but it's increasingly important in our data technology world, in which we deal with many different databases, data lakes, and a vast ecosystem of relational and non-relational open source data systems available either on-premises and/or in the cloud. The adjective “federated” can be added in front of a “query engine” or a “query” but mean slightly different things:

Federated query engine: A query engine that is mapped to multiple data sources which enables unified access to those systems for either queries to a single data source at a time or for federated queries with multiple data sources (including with JOINS across tables from different data sources). Most users today use Presto to unify access to multiple data sources but query them one at a time. Why? JOINS across sources require the underlying data to be correlated. And those linkages may or may not be available. However, correlation across data sources is the ultimate prize of federation to find insights and patterns on data across multiple sources, which otherwise would not be possible.

Federated query: A single query which stores or retrieves data from multiple different data sources, instead of a single data source.

Presto is a federated query engine that supports pluggable connectors to access data from external data sources and write data to those external data sources -- no matter where they reside. Many data sources are available for integration with Presto.

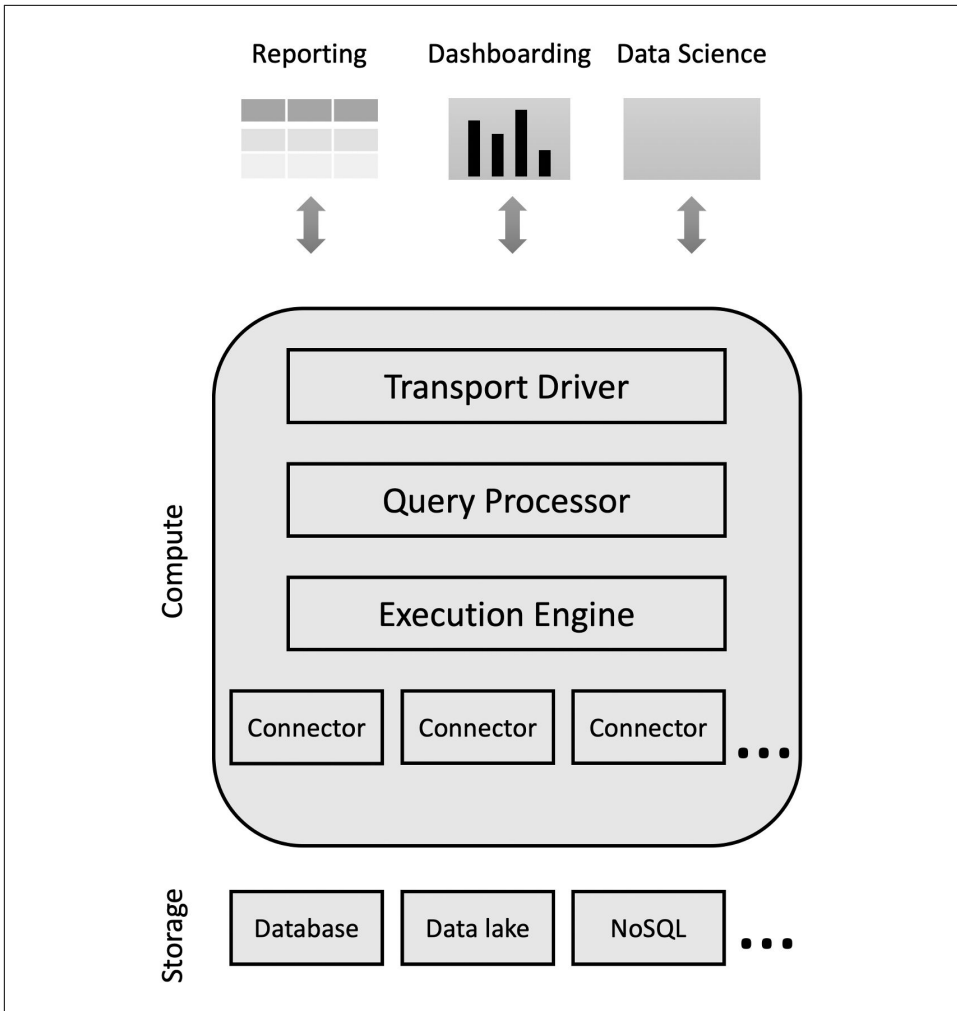


Figure 1-4. The Presto federated query engine with multiple connectors processing data in a disaggregated stack from multiple data sources

Presto can connect to each data source and ship some of the query processing down to it, with a filtered set of data pulled back into the workers for further processing. This could even include correlating multiple data sources together with SQL Joins. With this approach, SQL can be used to not only query traditional relational data sources but also non-relational data sources like NoSQL databases (like MongoDB, Elasticsearch) and data lakes (like HDFS, Amazon S3). The data is flattened and made into a tabular form when it is read from non-relational sources and once pulled into Presto, can be then be correlated with other structured data.



These connections are possible because Presto understands SQL, specifically the ANSI SQL standard. Using one standard language helps the project be consistent over time and avoids the need to rewrite queries when you change a backend data source.

As of this writing, there are over two dozen connectors available and this number is growing as the community writes more. These connectors are for databases, object stores, data lakes, streaming data, almost any data system. If you have a data source that currently doesn't have a connector, you could write one yourself and contribute it back too.



Figure 1-5. Examples of external data sources you can query with Presto

How Presto works

We've covered what Presto is, but how does it work? Presto is written in Java, and therefore requires a JDK or JRE to be able to start. It is deployed as two main services, a single coordinator and many workers. The coordinator service is effectively the brain of the operation, receiving query requests from clients, parsing the query, building an execution plan, and then scheduling work to be done across many worker services. Each worker processes a part of the overall query in parallel, and you can add worker services to your Presto deployment to fit your demand. Each data source is configured as a catalog, and you can query as many catalogs as you want in each query.

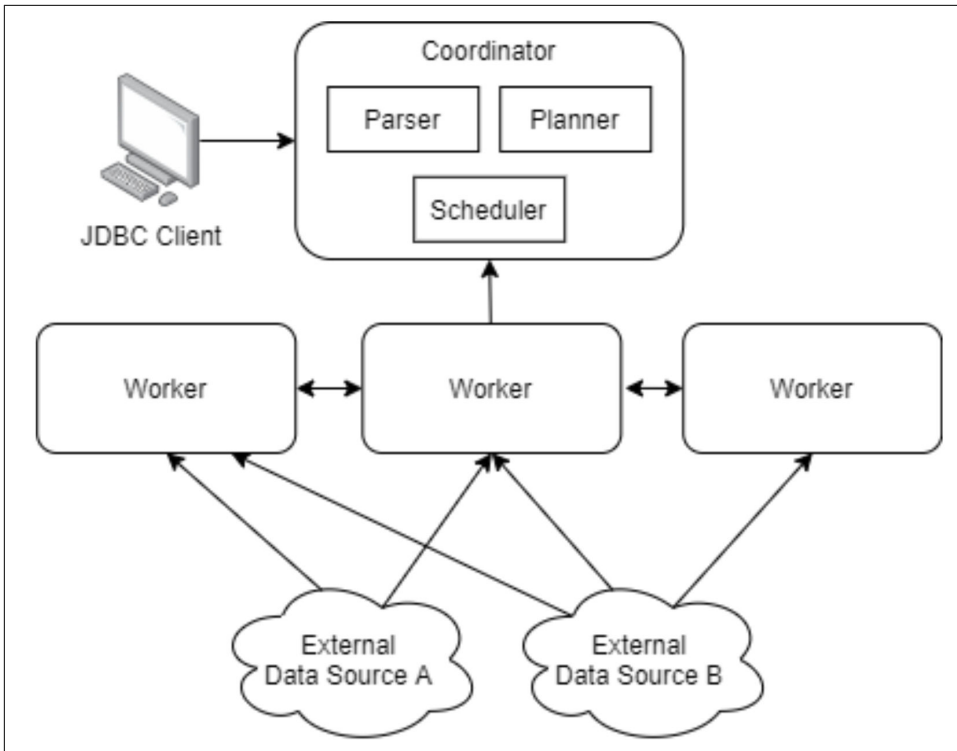


Figure 1-6. How Presto executes a SQL query

As shown in [Figure 1-6](#), Presto is accessed through a JDBC driver and integrates with practically any tool that can connect to databases using JDBC. The Presto CLI is often the starting point when beginning to explore Presto. Either way, the client connects to the coordinator to issue a SQL query. That query is parsed and validated by the coordinator, and transformed into a query execution plan. This plan details how a query is going to be executed by the Presto workers. The query plan typically begins with one or more table scans in order to pull data out of your external data sources. This will be explained more in the next section.

Users can query a single data source with Presto. This approach can be particularly useful when querying data lakes like HDFS, Amazon S3, Google Cloud Store and others via the connector that Presto calls the Hive connector. Presto becomes the separated query engine that uses the metadata from an external catalog (configured via the Hive connector) and processes data stored in the data lake.

This approach can also be useful to run analytics on a non-relational data source like mongoDB or Elasticsearch using Presto SQL via BI and reporting tools. This single stack approach means Presto can be a fast query engine for any database and data lake.

Of course, as a federated engine, we see many Presto deployments that are connected to multiple data sources. This allows end users to query a data source at a time, using the same interface without having to switch between systems or think of them as different data systems. For example, you could have a single BI dashboard with 5 graphs with data pulled from 5 different data backends. Federating data sources makes easy work of that task.

One of the most popular connectors, is the Hive connector, that allows users to query data using the same metadata you would use to interact with HDFS or Amazon S3. Using this connector, Presto is able to read data from the same schemas and tables using the same data formats -- ORC, Avro, Parquet, JSON, and more. Because of this connectivity, Presto is a drop-in replacement for organizations using Hive or Spark SQL.



Presto's Hive connector would have better been named a 'data lake connector'. Facebook created Presto as a replacement for Hive, so why would there be a connector to the replacement? It's not, the Hive connector is referring to the Hive metastore. The Hive connector enables Presto to query any data lake via a metadata catalog whether that is a Hive metastore or another catalog like Amazon Glue. The metadata catalog integration is a very important aspect of a disaggregated computational engine because this is what maps files stored in data lakes into databases, tables and columns and allows for SQL to be applied to query files.

In addition to the Hive connector, you'll find connectors for MySQL, PostgreSQL, Kafka, Elasticsearch, Cassandra, MongoDB, and many others.

Taking federation a step further, a query can combine data from two or more different data sources. For example, you can join data sets and select data between Kafka and MySQL, or between MongoDB and PostgreSQL, or between Amazon S3 and Pinot (Presto with Pinot is discussed in detail in [Chapter 3](#)), or between Elasticsearch, S3 and MySQL -- you get the idea. The benefits of doing so allows end users to analyze more data without the need to move or copy data into a single data source. Even though a NoSQL database like MongoDB doesn't support SQL, Presto nonetheless has a way to query it via the Presto connector for MongoDB. That connector allows the use of MongoDB collections as tables in Presto. Instead of moving the data in MongoDB over to another system and maintaining a data pipeline when there are changes, or you could simply query both systems in place with Presto. So Presto has the advantage of querying across disparate systems which may have different data models.

Connectors are being contributed to the Presto project all the time. If you need one and see that it's not currently available, you can write your own connector in Java. If

you don't want to write in Java, you can use the Apache Thrift connector and write a Thrift service in Go, Python, JavaScript, or another preferred language. Thrift is a software library and tools to enable efficient and reliable communication between distributed services. Thus, you can integrate almost anything with Presto. As a result, Presto has the potential to become the primary way to perform analytics on all your data

To summarize this section, you can work with data in three different ways, the difference between 2 and 3 is primarily whether or not JOINS are used across systems:

1. Presto configured with only one data source - gives you fast analytics.
2. Presto configured with multiple data sources, each queried independently - gives you fast, federated analytics (Figure 1-7, left diagram)
3. Presto configured with multiple data sources, correlated and queried together - gives you fast, federated, unified analytics (right diagram)

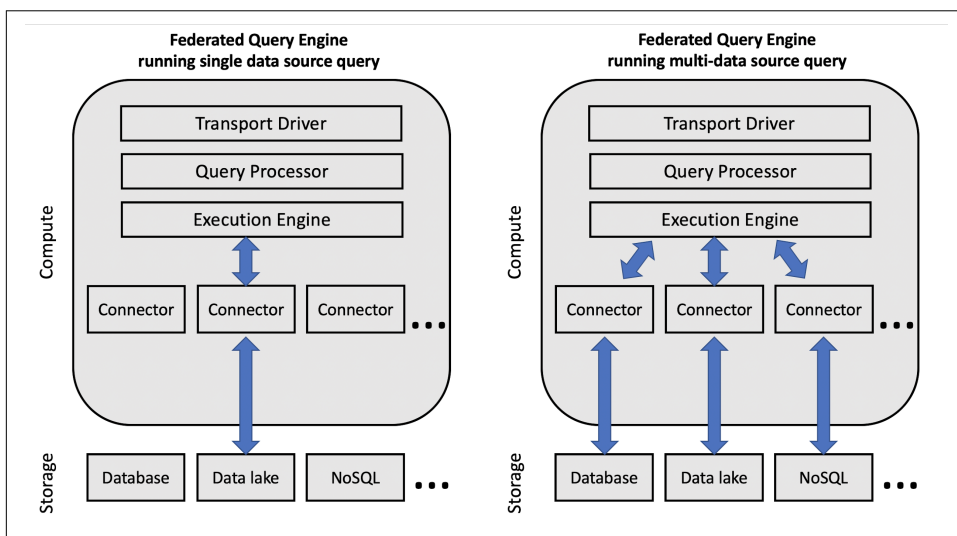


Figure 1-7. How federated query engines like Presto allow you to run single data source or multi-data source queries (federated queries).

Presto query processing explained

Once a SQL query is received by the coordinator via one of its interfaces (CLI, JDBC etc), the coordinator parses the query into an internal representation called a syntax tree. The analyzer uses this to break the query down into consumable parts in a logical plan.

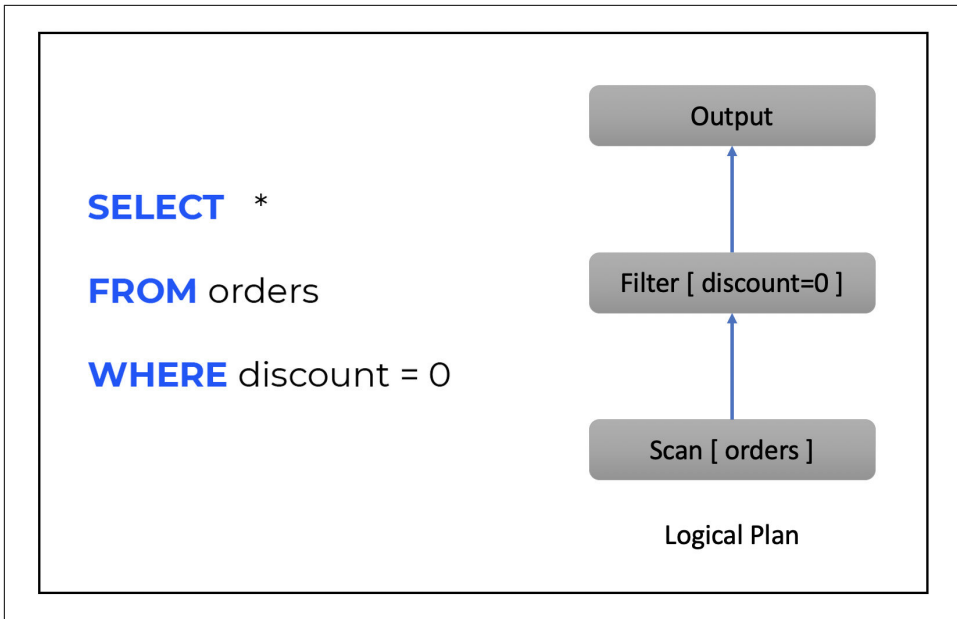


Figure 1-8. Syntax tree

The query optimizer takes the logical plan and converts it into a physical plan that includes an efficient execution strategy for the query. Presto has a rules-based and a cost-based optimizer.

The data sources connected to Presto may be of varied types and complexity, i.e. they could be simple filesystems, object stores, or highly optimized databases. Presto contains several rules, including well-known optimizations such as predicate and limit pushdown, column pruning, and decorrelation. It can make intelligent choices on how much of the query processing can be pushed down into the data sources, depending on the source's abilities. For example, some data sources may be able to evaluate predicates, aggregations, function evaluation, etc., and by pushing these operations closer to the data Presto achieves significantly improved performance by minimizing disk I/O and network transfer of data. The remainder of the query, e.g. joining data across different data sources, will be processed by Presto on its workers, and the final result set is transferred to the client via the coordinator.

In addition, the internal engine is columnar, which coincides with the popularity of columnar storage formats like ORC and Parquet, both of which Presto supports. That means that Presto builds columnar blocks and the engine processes those columnar blocks. It's columnar execution on top of columnar storage. Columnar storage formats store data of a single column together (contiguously). — and so it doesn't get much better than that. With both the storage format and the execution mechanism oriented towards a columnar approach, the analytic workloads run faster on Presto.

Different parts of query processing like scans, aggregation, etc. can be executed faster when columns are grouped together. Thus query operators can quickly loop over all the column values potentially loading some or all of them into the CPU in a single memory reference.

Presto Operations

Presto is a distributed system which forms the compute tier and interfaces with both the data layer and the front end tooling. The operations of such a varied data system can be challenging. Operations of Presto include the configuration, administration, security, tuning and troubleshooting of clusters. More specific to the cluster itself, we'll discuss high availability options for the single coordinator node, handling of sizing, scaling concerns, connecting new data sources, and scaling concurrency. Then there is the manageability side with maintenance, updates, and fault tolerance. These are all topics which are important to understand in order to successfully deploy Presto which we will cover in future chapters.

Presto at Scale

The Presto workers we mentioned previously allow Presto to run at any scale. A large infrastructure isn't a requirement though. Presto can be used in small settings or for prototyping before tackling a larger data set. Because of its very low latency, there isn't major overhead for running small queries as you'd see with a batch-orientated system like Hive. You can start by querying megabytes at first, but the same queries can be run on gigabytes, terabytes or even petabytes as your data size increases over time.

It's easy to run a demo on a laptop using a [container tutorial using a Presto sandbox on DockerHub](#). Once you start, your Presto deployments can scale out horizontally, even to Facebook levels².

As a distributed worker system, Presto is horizontally scalable, running on commodity servers. If you have a 50 node cluster, you can add another 50 machines and it just becomes a 100 node cluster with all the benefits therein (i.e. higher performance, concurrency, etc.)

Presto in the Cloud

There are also a number of cloud offerings for running Presto. You can run Presto via Amazon Elastic MapReduce (EMR) and Google Dataproc. Amazon Athena, a server-

2 When Facebook donated PrestoDB to the Linux Foundation and established the Presto Foundation, Facebook announced the scale Presto was being used: "At Facebook alone, over a thousand employees use Presto, running several million queries and processing petabytes of data per day."

less, interactive query service to query data and analyze big data in Amazon S3 using standard SQL, is built on Presto. Other vendors offer Presto as a managed service, such as **Ahana**, that make it easier to set up and operate multiple Presto clusters for different use cases.

Presto as an Analytics Platform

Presto is the engine for many different types of analytical workloads to address a variety of problems. As we discussed, the disaggregated stack leverages Presto along with one or more data stores, metastores, and SQL tools. Next, we describe some of the popular uses for Presto as an analytics platform.

Ad hoc querying

Presto was originally designed for interactive analytics, or ad hoc querying. In today's "Internet era" competitive world and with data systems able to collect granular amounts of data in near-real-time, engineers, analysts, data scientists, and product managers all want the ability to quickly analyze their data and become data-driven organizations making superior decisions, innovating quickly and transforming their businesses for the better.

They either simply type in simple queries by hand or use a range of visualization, dashboarding, and BI tools. Depending on the tools chosen, they can run 10s of complex concurrent queries against a Presto cluster. With Presto connectors and their in-place execution, platform teams can quickly provide access to the data sets that users want. Not only do analysts get access, but also they can run queries in seconds and minutes—instead of hours—with the power of Presto, and they can iterate quickly on innovative hypotheses with the interactive exploration of any data set, residing anywhere.

Reporting and dashboarding

Because of the design and architecture of Presto and its ability to query across multiple sources, Presto is a great backend for reporting and dashboarding. Unlike the first generation static reporting and dashboarding, today's interactive reporting and dashboards are very different. Analysts, data scientists, product managers, marketers and other users not only want to look at KPI's, product statistics, telemetry data and other data, but they also want to drill down into specific areas of interest or areas where opportunity may lie. This requires the backend - the underlying system - to be able to process data fast wherever it may sit. To support this type of self-service analytics, platform teams are required to either consolidate data into one system via expensive pipelining approaches or test and support every reporting tool with every database, data lake and data system their end users want to access. Presto gives data scientists, analysts and other users the ability to query data across sources on their own so

they're not dependent on data platform engineers. It also greatly simplifies the task of the data platform engineers by absorbing the integration testing and allowing them to have a single abstraction and end point for a range of reporting and dashboarding tools.

ETL using SQL

Analysts can aggregate terabytes of data across multiple data sources and run efficient ETL queries against that data with Presto. Instead of legacy batch processing systems, Presto can be used to run resource-efficient and high throughput queries. ETL can process all the data in the warehouse; it generates tables that are used for interactive analysis or feeding various downstream products and systems.

Presto as an engine is not an end-to-end ETL system, nor is Hive or Spark. Some additional tools can be easily added to coordinate and manage numerous on-going time-based jobs, a.k.a. cron jobs, which take data from one system and move it into another store, usually with a columnar format. Users can use a workflow management system like open source Apache Airflow or Azkaban. These automate tasks that would normally have to be run manually by a data engineer. Airflow is an open source project that programmatically authors, schedules and monitors ETL workflows, and was built by Airbnb employees who were former Facebook employees. Azkaban, another open source project, is a batch workflow job scheduler created at LinkedIn to run Hadoop jobs.

The queries in batch ETL jobs are much more expensive in terms of data volume and CPU than interactive jobs. As such the clusters tend to be much bigger. So some companies will separate Presto clusters: one for ETL and another one for ad hoc queries. This is operationally advantageous since it is the same Presto technology and requires the same skills. For the former, it's much more important that the throughput of the entire system is good versus latency for an individual query.

Data lake analytics

Data lakes have grown in popularity along with the rise of Amazon S3-compatible object storage which Amazon AWS has made popular. A data lake enables you to store all your structured and unstructured data as-is and run different types of analytics on it.

A data warehouse is optimized to analyze relational data, typically from transactional systems and business applications, where the data is cleaned, enriched, and transformed. A data lake is different from a data warehouse in that it can store all your data—the relational data as well as non-relational data from a variety of sources, such as from mobile apps, social media, time-series data—and you can derive more new insights from the analysis of that broader data set. Again you can do so without necessarily needing to process that data beforehand.

Presto is used to query data directly on a data lake without the need for transformation. You can query any type of data in your data lake, including both structured and unstructured data. As companies become more data-driven and need to make faster, more informed decisions, the need for analytics on an increasingly larger amount of data has become a higher priority in order to do business.

Real-time analytics with real-time databases

Real-time analytics is becoming increasingly used in conjunction with consumer-facing websites and services. This usually involves combining data that is being captured in real time with historical or archived data. Imagine if an e-commerce site had a history of your activity archived in an object store like S3, but your current session activity is getting written to a real-time database like Apache Pinot. Your current session activity may not make it into S3 for hours until the next snapshot. By using Presto to unite data across both systems, that website could provide you with real-time incentives so you don't abandon your cart, or it could determine if there's possible fraud happening earlier and with greater accuracy. In [Chapter 3](#), we'll show you how to do just that with Presto connected to Pinot, a real-time database.

Open Source Community

Presto was released under the permissive open-source Apache 2.0 license in 2013; however, it is not an Apache Software Foundation (ASF) project. Presto is instead governed by the Presto Foundation under the auspices of The Linux Foundation, who happen to know a lot about open source projects! It was and continues to be developed in the open on their public Github account, and even Facebook continues to put their new mainstream features upstream. This means Facebook is running the same version of code that you can download and use yourself for free. You can depend on the fact that Facebook and many other companies are constantly testing and improving Presto, using the collective power of a community of talented individuals working together to deliver quick development and troubleshooting. As an open source project, there are many offshoots of Presto, also known as forks of the codebase. PrestoDB is the main project and that is the main project of the future. As such, we encourage all new users to use the main upstream project of PrestoDB.

As someone learning more about Presto, helping others learn about this open source project can be very satisfying. As you become more familiar with the project, becoming a contributor can be a lot of fun, even if it is quite challenging at the beginning. If you have your own software project, you can make all the decisions and act very quickly, whereas open source tends to have a lot more discussion. Keeping in mind that the best projects attract the best people, when you contribute, you are getting in touch with some of the most talented engineers and architects out there. While it may be difficult to convince them, when that happens and your code is merged, it may

likely give you a pretty solid sense of satisfaction. Then of course there is the reputation that comes along with a strong github profile. Your contributions highlight your expertise and can be very beneficial for advancing your career.

It is easy to get involved with the Presto Community and become part of the global network of passionate people improving Presto. New learners can start out with fixing documentation or beginner issues. You can join the Presto Foundation [Slack channel](#), engage via [Github Discussions](#), attend the [virtual meetups](#), follow them on Twitter [@prestodb](#), or request a new feature or file a bug: github.com/prestodb/presto.

Presto Foundation

The [Presto Foundation](#) is a non-profit organization that [was formed with The Linux Foundation](#) in September 2019 to support the development and community processes of the [Presto open source project](#). As a part of The Linux Foundation, Presto Foundation ensures the open governance, transparency, and continued success of the project. Presto has experienced a steady growth in popularity over the years, with several companies emerging to support that growth. Members of the Presto Foundation provide essential financial support for the collaborative development process, including tooling, infrastructure, and community conferences. Presto Foundation membership also requires membership to the Linux Foundation. The current premier members are: Facebook, Uber, Twitter, Alibaba, Alluxio, and [Ahana](#). All are welcome, including non-profits and academic institutions at a special membership rate.

Conclusion

In this chapter we tried to provide an overview of the Presto open source project. Where did it come from? How is it different from other data systems? How does it work and for what common use cases?

On a high level, we saw how Presto was initially developed for interactive, ad hoc analytics which needed low-latency queries, enabling end users with exploratory analysis on data sets of any size. We looked at key design considerations which make Presto flexible and suitable for a wide spectrum of use cases: the in-memory architecture, the compliance with the ANSI SQL standard make it immediately accessible without additional integration, the federation of data sources, and the coordinator-worker scalability.

We did brief comparisons of Presto with Hive and Presto with a traditional data warehouse. We saw how Presto is part of the disaggregated compute and storage stack—the latter having multiple back ends via the Presto connector architecture.

We then briefly looked at how Presto handles numerous SQL-based use cases:

- Interactive, ad hoc querying
- Reporting and dashboarding
- ETL
- Data lake analytics
- Real-time analytics with real-time databases

As a fast growing open source project, you heard about ways to join the community and even gained some perspective on the open governance provided by the non-profit Presto Foundation.

Although this first chapter can't make you an instant Presto expert, it hopefully armed you with enough context and high-level understanding of Presto.

We've learned from experience, and that is why this book is focused on both "Learning and Operating Presto." We hope this book will help you become adept at operating Presto for your organization. So let's conclude this chapter and start jumping into getting some hands-on experience with Presto!

Operating Presto at Scale

*Puneet Jaiswal, Principal MTS, DataFlow-Interactive, Oracle Cloud
(Former Software Engineer, Data Infrastructure - Interactive @ Lyft)*

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

In this chapter, we will be looking into the common issues encountered when operating a Presto Cluster, then running multiple Presto clusters behind a presto-gateway to achieve better quality of service, and the tools and observability required to provide high availability, fault tolerance, projecting and planning usage growth and briefly touching upon the process of performance testing and metrics analysis.

Common issues when running Presto at scale

Before we look at what it takes to operate Presto at Scale, let’s explore some of the issues and limitations involved with various components of Presto. The most common are Single Point of Failure (SPoF), bad worker state, and issues with the query. By identifying them, we can quickly pinpoint the issue and remediate them as needed.

Then, we will go into detail on operating Presto at scale, including using Presto gateway, using a workload manager for better resource utilization, using the query event framework for better observability, and more best practices to run Presto at scale.



An assumption is made that the Presto Infrastructure would be using the *Hive-Hadoop* connector plugin to access the data.

Presto Coordinator - Single Point of Failure (SPoF)

At present, the Presto coordinator is set up as a single instance and responsible for receiving SQL query, analyzing, planning, execution, and streaming the results back to the client



There is a new feature - disaggregated coordinator - coming soon which will make the coordinator highly available (HA)

For any reason if this node fails, we lose all the queries submitted to the Presto cluster.

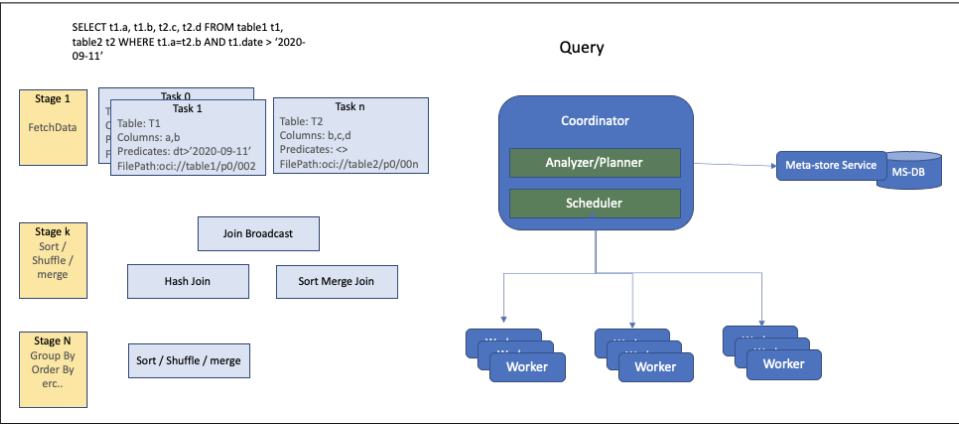


Figure 2-1. Left- Stage/Task representation of a SQL query; Right- Presto components at a high level.

As shown in **Figure 2-1**, a query lands to the coordinator for execution, the coordinator needs to extract information such as what tables, columns, predicates, joins, etc. are involved in the query so it parses the query and analyzes it. Then the coordinator builds an execution plan. To build an efficient execution plan, the coordinator talks to

the metastore service and tries to find the table stats (e.g. how many partitions will be scanned, how many files will be read from the object store and their specific location, etc.). If cost-based optimization is enabled, column stats (for a given set of predicates, how many rows will be fetched) will be used to build an optimal query plan with various stages (each stage with multiple tasks). Now the scheduler looks at the plan and starts scheduling tasks on the available workers.

To avoid the network latency involved in fetching the query metadata from the remote metastore service, you can set the coordinator to cache the metadata in its local memory. However, this will increase the memory footprint of the coordinator.

Bad worker state

As Presto runs on Java Virtual Machine (JVM), and it is an In-Memory compute engine, the memory is managed by the JVM and with the usage, old gen collection time may grow gradually causing a significant amount of GC pause. There could also be out-of-memory errors if a node is trying to load more data in-memory than its capacity. Such events can put the worker in a bad state.

Presto does not retry for failed tasks. Given this, if a worker is in a bad state, any task scheduled on it is deemed to fail. When this happens the query fails and the compute resources spent on other successful tasks/stages for the same query are wasted. We want to minimize such occurrences by removing the bad worker node as soon as possible.

In the following topics, a real-time query logger framework is described, and when a worker goes in a bad state, the queries handled by this worker are deemed to fail and the query failure log is captured into *failureInfo* field. Monitoring this event stream for such errors can help in detecting the bad worker and remediation from such failures can be easily implemented to auto-heal the Presto infrastructure.

Bad queries

A bad SQL query can take over the compute resources and starve the other queries for resources. A SQL query may perform poorly due to the following reasons:

- Badly written inefficient query
 - Bad joins
 - Inefficient comparison operators
 - Inefficient group/order by clauses
 - Missing partition filter
 - Missing predicates
- Queries scanning too much data than the presto cluster could handle

- Large tables with no partition / No partition pruning
- Uneven partitions (skewed data)
- Too much data to load in Presto query engine
- Too many Queries going to `System.information_schema.columns`
 - A high query rate to `System.information_schema` could easily overload the metastore service as these queries are ultimately run against the underlying relational db.

Large scale Presto infrastructure in production

There are multiple aspects that should be analyzed to yield the best performance out of a Presto cluster. Some of these aspects could be derived by understanding the query workload, user behavior, usage/consumption of dependent systems (eg. meta-store / object store, etc), data quality, indexing, partitioning, and distribution on the object store. Understanding these aspects requires collecting rich metrics around them and setting a baseline for various metrics and figuring out the settings which give the sweet spot in the Presto cluster.

Using Presto Gateway to improve infrastructure availability

There could be multiple occasions when a Presto cluster would require a restart. For example, when there is a version upgrade and the Presto Coordinator node is saturated for resources, or if there is a JVM issue such as high GC pause or out of memory. In such cases the node reset or cluster restart is inevitable. Each such incident causes downtime. To avoid it, a Gateway or Query Routing layer can be implemented that can be used to perform query partitioning or routing to multiple Presto clusters. With such a setup, if one Presto cluster requires a restart, it can be disabled at the gateway layer to receive no new queries while serving and finishing the already running queries, and eventually, it gets removed making this process transparent to the end-users. When the same Presto cluster is healthy and ready to serve new queries, the activate API can be called on the Presto-Gateway. Automating this process by adding and removing Presto clusters based on load, would make the infrastructure truly elastic.

With multiple Presto backends behind the gateway, it is possible to implement an A/B testing framework, routing only a subset of production queries to a newer version of the Presto cluster. Once the overall performance looks good on this newer version, you can roll out this version to the rest of the infrastructure.

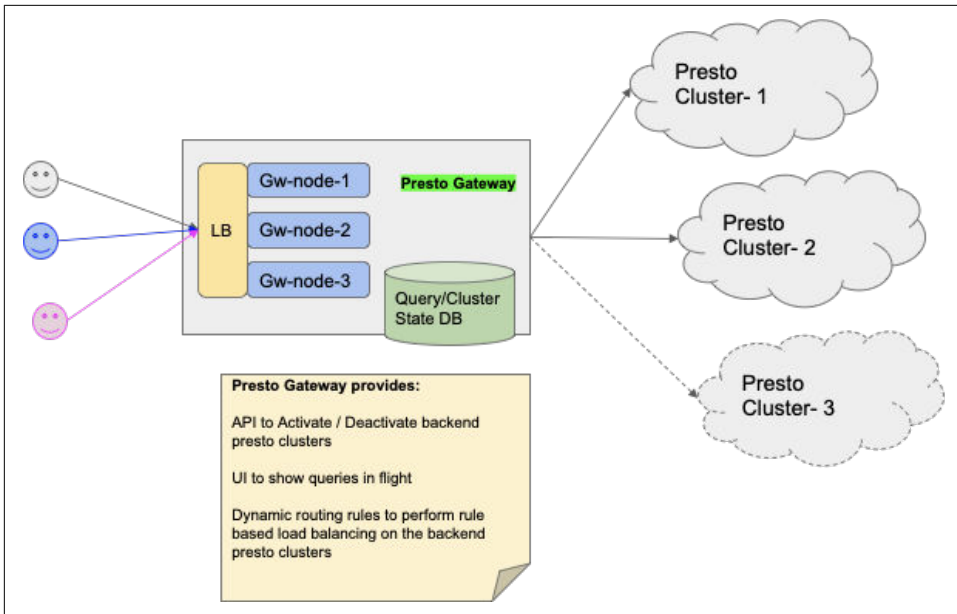


Figure 2-2. Presto Gateway used for rule based query load balancing

Lyft's **Presto Gateway** provides a way to create multiple routing groups, and in each routing group, one or more Presto clusters can be grouped to serve user queries. This gateway provides an API to activate or deactivate a backend Presto cluster. When a backend Presto cluster is deactivated, the gateway stops sending new queries to this cluster while letting the ongoing queries reach this inactive backend cluster until completion. The idea is when an inactive Presto cluster stops serving the ongoing queries, then it is safe to be restarted or shutdown. The gateway provides the flexibility to perform a Presto cluster reset or upgrade without affecting the overall availability.

Presto Gateway provides:

- API to activate/deactivate backend Presto clusters.
- Fast access UI to show queries in flight.
- Dynamic routing rules to perform rule-based load-balancing on the backend presto clusters.
- Configurable settings to run a Presto cluster with an A/B testing methodology

Currently, Lyft's **presto gateway** has been adopted by **Pinterest**, Electronic Arts (EA), and Shopify in production.

Workload Manager - Presto Resource Groups for better resource utilization

A query engine service needs a workload manager to ensure resources are fairly utilized or distributed among the users and/or the queries. It also provides a way to prioritize queries sent by the specific user groups. Presto provides Resource Groups to manage resource utilization on the System.

To enable the Resource Groups-based Workload Manager, add the following to the presto config *etc/resource_groups.properties*.

```
resource-groups.configuration-manager=file
resource-groups.config-file=etc/resource_groups.json
```

The files-based resource groups configuration is structured as below.

```
{
  "rootGroups": [],
  "selectors": [],
  "cpuQuotaPeriod": "1h"
}
```

A resource group represents available Presto resources packaged together having limits related to CPU, memory, concurrency, queueing, priority, etc. The selectors are rules that match an incoming query to one of these resource groups and the *cpuQuotaPeriod* is a global optional field that is used as the max value on which the CPU limits are applied.

With configurable root-groups, sub-groups, and selectors, a very fine grained workload-manager configuration policy can be created to provide access to the Presto cluster. The following advanced resource groups configuration can be added to the file *etc/resource_groups.json*.

```
{
  "rootGroups": [
    {
      "name": "global",
      "hardConcurrencyLimit": 35,
      "maxQueued": 200,
      "jmxExport": true,
      "softMemoryLimit": "80%",
      "subGroups": [
        {
          "name": "user_${USER}",
          "softMemoryLimit": "70%",
          "maxQueued": 20,
          "hardConcurrencyLimit": 5
        }
      ],
    },
    {
      "name": "advance-data-group",
      "softMemoryLimit": "75%",
    }
  ]
}
```

```

        "maxQueued": 40,
        "hardConcurrencyLimit": 10
    }
],
{
    "name": "admin",
    "maxQueued": 100,
    "jmxExport": true,
    "softMemoryLimit": "80%",
    "hardConcurrencyLimit": 20
}
],
"selectors": [
    {
        "user": "data-admin",
        "group": "admin"
    },
    {
        "source": "superset",
        "group": "global.user_${USER}"
    },
    {
        "source": "jupyter",
        "group": "global.advance-data-group"
    },
    {
        "group": "global.user_${USER}"
    }
],
"cpuQuotaPeriod": "1h"
}

```

In this configuration example you can see that globally (non admin, non advance-data-group user), `hardConcurrencyLimit` enforces maximum globally running query concurrency as 35, `maxQueued` sets maximum globally queued queries as 200, `hardConcurrencyLimit` sets maximum query concurrency as 5 per user, and `maxQueued` sets maximum queued queries per user as 20.

For a non admin (resource group 'global'), advance-data-group is defined using a selector such that any query coming from source jupyter, the max allowed concurrently running query count per user is 10 and max allowed queued query per user is 40.

For an admin user (resource group 'admin'), a selector is defined having user name 'data-admin', enforcing maximum concurrently running queries limit as 20 and max allowed queued query count as 40.

Query Event Listener framework for better observability

Presto provides an event listener plugin interface, and it can be implemented to intercept query events such as *queryCreated*, *queryCompleted*, and *splitCompleted*. Each of these events provides a query event object that contains statistics of the respective event that can be captured as an event log for analyzing the query and run later.

```
package com.facebook.presto.spi.eventlistener;

public interface EventListener {
    default void queryCreated(QueryCreatedEvent queryCreatedEvent) { }
    default void queryCompleted(QueryCompletedEvent queryCompletedEvent) { }
    default void splitCompleted(SplitCompletedEvent splitCompletedEvent) { }
}
```

A query logging pipeline can be easily implemented using this interface.

```
public class QueryLoggerEventListener implements EventListener {
    private static final Logger logger = Logger.get(QueryLoggerEventListener.class);
    // load this from presto - config
    private String clusterName = "xyz";
    private String environment = "Prod-1";
    private String version = "0.239";
    public void queryCreated(QueryCreatedEvent queryCreatedEvent) {
        PrestoQueryEvent.PrestoQueryEventBuilder event = queryCreateEvent(queryCreatedEvent);
        pushEvent(event);
    }
    public void queryCompleted(QueryCompletedEvent queryCompletedEvent) {
        PrestoQueryEvent.PrestoQueryEventBuilder event = queryCompleteEvent(queryCompletedEvent);
        pushEvent(event);
    }
    private void pushEvent(PrestoQueryEvent.PrestoQueryEventBuilder eventBuilder) {
        writeEventMetadataFields(eventBuilder);
        try {
            // Sink this event to a store
            logger.debug("Logging query event: " + eventBuilder.build().toString());
        } catch (Exception e) {
            logger.warn(e, "Error in pushing presto event to the log store!");
        }
    }
}
```

If the ***pushEvent*** method in the above implementation sinks the events to a table - *presto_query_event*, with a schema definition described as below.

```
import lombok.Builder;
import lombok.ToString;
@Builder
@ToString
public class PrestoQueryEvent {
```

```

String id;
long occurredAt;
String eventType;
long createdAt;
long executionStartedAt;
long endedAt;
long wallTimeSecs;
// Failure related
String failureInfo;
int errorCode;
String errorCodeText;
String errorType;
String failureHost;
String failureTask;
String failureType;
// Query stats
long analysisTimeSecs;
int completedSplits;
long cpuTimeSecs;
long queuedTimeSecs;
long totalInputBytes;
long totalInputRows;
long peakUserMemoryBytes;
long peakTaskTotalMemory;
long peakTotalNonRecoverableMemBytes;
// Query context
String clustername;
String environment;
String version;
String user;
String source;
String serverAddress;
// Query metadata
String queryText;
String queryId;
String state;
}

```

Now, querying this table *presto_query_event*, we can easily collect the following metrics:

- Query latency - wait, execution, overall
- Query rate - success, failure, overall
- Query failure rate - by type (user / system), error, overall
- Query count per user / source
- Raw data processing rate - storage throughput
- Avg (p90) raw data processed per query (to understand served query shape)

- DAUs / WAUs - Daily / Weekly active (unique) users (indicates growth, justifies need for scaling)

These are a few critical metrics to look at to understand the overall health and performance of the Presto cluster. These metrics indicate not only the current usage and help in detecting and remediating any ongoing issues, but also help in planning for the organic growth of the platform. For example, query volume per hour and raw data processing rate metrics should be synonymous, as these are tightly coupled and related. Looking at the pattern, it can be figured out when the usage is high and when it is low, and if there is a pattern with time or day of the week, the infrastructure administrator can prepare the Presto infrastructure with more (or fewer) worker nodes so that the query load could be handled gracefully, while at the same time optimizing the infrastructure cost with demand.

Percentile vs Mean average measure

Mean or 50 percentile aggregated metrics are useless as they do not indicate the coverage of the metric over all data points, however choosing P90 or above usually gives a better overview of the system.

Query logs can provide insight into system performance in a number of ways.

Finding the query / user / source outliers. If P95 (95 percentile) query latency is under 1 minute but P99 is at 20 minutes, the infrastructure administrator needs to drill down into various dimensions to understand those 5 percent high latency queries - e.g. the source of such queries (dimensions viz. presto cluster / Schema / Table / Aggregation type / User / Client etc), the amount of data it's scanning, are the queries having the right predicates for pushdown, what time of the day/week these queries run, etc. If the Presto Cluster requires additional worker nodes to support those queries, such queries can take up the resources and cause other queries to starve.

Improving query success rate . Query logs are a good source to review overall failures and understand the root cause. *QueryCompleted* event provides the failure type (user error or server error) along with the stack trace of the exception that was caught when the query had failed.

Query predicate & join analysis. By looking at the query logs, you can find how many queries have scope of optimization, e.g. if queries have table partition filters present or not. If there is a join involved, then are all the tables have partition filters present. How much raw data each query is scanning.

Building a Query Replayer for low-cost performance testing. Running Presto for performance testing does not come cheap as it would require setting up a production-like cluster to run production-like load, also it is hard to simulate production like load in

test environments. In addition to finding whether a new Presto version (or config change) is better than a previous version (or configuration settings), the replayer-based testing identifies an ideal cluster shape for a given or expected amount of load. It means replicating your production environment (may include both compute and storage) with a golden set of test queries and data. Using production performance data in a replayer-based performance testing eliminates the cost involved in preparing a baseline as this data is already logged in a previous run in the *presto_query_event* table as explained above, thus eliminating the cost involved in setting the performance baseline.

Sometimes it is hard to replicate storage for performance testing, in such cases to avoid data corruption, the replayer should strictly run read only queries. That means selecting only those queries from the *presto_query_event* table that do not create or modify existing tables.

Following are the steps performed by the replayer:

- **Step 1.** Prepare a production equivalent test cluster, with the newest Presto version to test.
- **Step 2.** Identify a time range to pick against a production cluster and fetch the queries that do not involve writing the data and the respective performance data from *presto_query_event* table.
- **Step 3.** Pick the queries (for every minute) from the above query log and dispatch the set of queries to the test cluster every minute.
- **Step 4.** As the queries complete, log the performance results for both - the old run and new execution as the query finishes.
- **Step 5.** Turn off the new test cluster as the execution completes, and compare the performance results.

As the overall served query count (or the weekly active user count) are increasing over time, it can be noticed that various latencies would increase (given the fixed infrastructure size), this is an indicator that, to maintain the same quality of service, the infrastructure should be scaled out - this is an example of organic growth and these metrics are a good indicator of when to scale out (or in) the Presto Infrastructure.

Query protection layer to protect the infrastructure from bad actors

Using the presto gateway, we can block or rewrite certain types of SQL queries, the ones that are badly written or deem to fail or harmful for the system (reference above section “Bad Queries”).

Besides the gateway, query blocking functionality can also be built in the event listener plugin. The event listener plugin provides a *queryCreated* method, in that we get access to the query text through the *QueryCreatedEvent* object.

The following example shows how a simple event listener can be added to block queries going to the table *system.information_schema.columns* that do not have any predicates (where clause).

```
// blocking queries not having predicates to the table system.information_schema.columns
public void queryCreated(QueryCreatedEvent queryCreatedEvent) {
    String queryText = queryCreatedEvent.getMetadata().getQuery().toLowerCase();

    if (queryText.contains("system.information_schema.columns") &&
        !queryText.contains("where")) {
        logger.error("Rejected query from event listener: " + queryText);
        throw new PrestoException(StandardErrorCode.COMPILE_ERROR,
            "Querying to system.information_schema.columns without predicates "
            + "is not permitted.");
    }
}
```

A tale of scale: Protecting Metastore from unnecessary calls

Many sql client tools - desktop tools (eg. Data Grip, SQL Developer, DBeaver etc.) or cloud tools (Mode, Looker, Apache SuperSet etc) actively refresh the table/column metadata to provide auto-fill and typeahead features by running a query on table *system.information_schema.columns*, eventually making a call to metastore. If there are lots of tables in Hive and metadata caching is not enabled (enabling caching at metastore service would bring other issues such as a table partition update may not be immediately available), this could cause additional and unnecessary pressure on the metastore and might cause the metastore to function poorly. A metastore call latency metric (check *analysisTimeSecs* field in the *presto_query_event* table) would be helpful to monitor metastore's performance and could alert for any slowness caused to the metastore if it is heavily under pressure.

Choosing node type and JVM settings in production

Selecting the right node type for the coordinator and workers is a bit tricky as it depends a lot on the type of queries run on the system, it should be decided by a long term strategy of choosing the right mix of Memory and CPU for each type. In order to do this, JVM metrics should be collected and monitored. Ideally during peak usage time, the coordinator should have 50 to 70% CPU and memory usage and the workers should have 90% or above CPU usage and 70% or above Memory usage.

If either of the CPU or Memory usage metrics are too high or low as compared to the other one, then it possibly means that the node type is not balanced in terms of resources.

For example, during peak usage time, if the workers are having 99% of CPU usage, but the memory usage is trending only around 45%, then we can deduce that the node type used for the workers is not balanced in terms of Memory and CPU resources. To balance this either (1) for the same amount of memory given, more CPUs should be added in the node or (2) the node has excess memory as compared to the present CPU count.

It is recommended to use Java Runtime Environment (JRE) - version 11 or higher as the old-gen phase garbage collection is not “stop the world”, it significantly reduces the GC pause time, improving the overall performance and availability of the Presto cluster.

A sample JVM configuration (jvm.config) in production would look as follows. In this example, 80% of the available node memory is assigned as max heap memory.

```
-server
-Djdk.attach.allowAttachSelf=true
-Xlog:gc:{{ presto_log_dir }}/gc.log
-XX:+UseG1GC
-XX:+AggressiveOpts
-XX:ReservedCodeCacheSize=150M
-XX:ErrorFile={{ presto_log_dir }}/java_error.log
-Xms{{ 0.80 * available_node_memory_mb }}m
-Xmx{{ 0.80 * available_node_memory_mb }}m
-XX:MaxGCPauseMillis=400
-XX:InitiatingHeapOccupancyPercent=5
-XX:OnOutOfMemoryError="kill -9 %p"
-XX:G1NewSizePercent=1
```

To enable jvm debugging, the following options can be added to allow a remote debugger to connect to the java process.

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
```

Conclusion

In this chapter, we looked into general issues faced while running a presto cluster, how various issues (node crash, out-of-memory, gc-pauses, bad queries starving the cluster) could disrupt the productivity. Then we looked at various strategies to scale the presto compute infrastructure, such as, using a presto-gateway brings stability and gives an option to scale the compute infrastructure horizontally with load, providing better infrastructure availability and fault-tolerance; setting resource-groups promotes fair usage and prevents the infrastructure from being abused by the a very small set of actors (queries or users); using query replayer as a low cost capacity plan-

ner or as a performance benchmarking tool and in the end, we looked at the best practices to follow in production environment.

Acknowledgement

A lot of this work was carried out by the Presto Team at Lyft that included Arup Malakar, Aakash Katipally, Alex Berghase, Brennon York, Bhuwan Chopra, Bill Graham, James Taylor, Nishant Rayan, Rong Ling, Ravi Sharma, Sharanya Shanthanam and many others.

Real-time Analytics for Real-time Business Insights: Presto & Apache Pinot

Devesh Agrawal, Former Software Engineer at Uber

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

Like a salad, data is best when you consume it fresh. It gives you the most benefits as soon as it’s made. Fresh data is also known as real-time data, or a series of events being generated from users, machines, or sensors. Fresh can be loosely defined as data that *just* happened a second ago. There are many types of fresh data, depending on the business. For LinkedIn, fresh data is the user clickstream, which is consumed by their flagship analytics product for members: “Who’s Viewed Your Profile.” For Uber, fresh data drives their user experience: getting a ride booked in seconds which shows up in minutes, with pricing based on real-time forecasts of local demand (also known as surge pricing). For other companies, fresh data could uncover fraudulent purchases.

Real-time datastores like Apache Pinot were designed to handle low-latency queries for constantly changing fresh data, in addition to static data. Their common characteristics are near real-time ingestion with Apache Kafka and horizontal scalability to

deliver consistent low-latency OLAP queries on any amount of data. This chapter will introduce Pinot, show how to extend Presto with the Presto-Pinot connector and finally share some hard won production best practices and tuning configurations for getting the best experience out of Pinot.

Introducing Apache Pinot

Apache Pinot is a real-time distributed OLAP datastore, designed to deliver real-time analytics with low latency. It can ingest from real-time stream data sources (such as Apache Kafka) and batch data sources (such as Hadoop HDFS, Amazon S3, Azure ADLS, Google Cloud Storage). It answers queries with single digit to low tens of milliseconds, and its excellent compute efficiency helps it scale to site facing online traffic.

Pinot was first developed at LinkedIn, where it runs at large scale, serving tens of thousands of analytical queries per second while ingesting real-time data from streaming data sources. They needed to be able to analyze the latest data in real time. But, in 2012, the amount of data was growing faster than the LinkedIn team could analyze it. They looked to the current database landscape for a solution. On one hand, operational, transactional databases like RDBMS and NoSQL databases could be scaled to provide fast reads and writes for pinpoint data for a large number of users, but had trouble dealing with the intersection of the analytic nature of the queries and high ingest rates. On the other hand, existing OLAP databases relied upon batch loads and batch queries of data, thus were not designed to handle low-latency queries or to scale for serving thousands of concurrent queries per second. Thus they embarked on developing Pinot.



Apache Pinot was named after Pinot, the notoriously difficult grape varietal for growing and producing wine. Pinot Noir is considered one of the most complex wines. This is an analogy to the real-time data environment: complex and challenging, yet, when done well, powerful in its results.

With Presto connected to Pinot, fresh data can be consumed by applications, data analysts, and scientists using standard ANSI SQL.

A closer look at Pinot

Pinot is a distributed system consisting of many different components as shown in [Figure 3-1](#). In general, each component is independently scaleable and has unique responsibilities. Pinot requires a Zookeeper installation to help it propagate state across components for high availability. We will first cover the Pinot controllers and

servers, and then see how the data is physically distributed across the cluster and finally close with how queries are executed in Pinot.

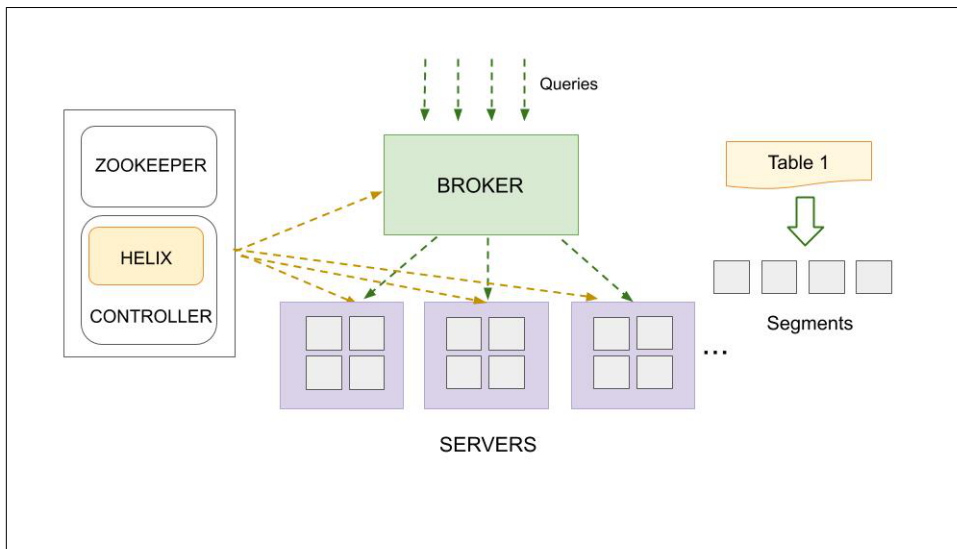


Figure 3-1. The Pinot distributed system

Pinot controller

The Pinot controller manages the state of the pinot cluster: It is the brain that plans the distribution of data across Pinot servers and rebalances the data as servers come and go. It also stores the table metadata like the schema and indexing information. The Pinot controller uses Apache Helix (another open source project out of LinkedIn) to save its state and ensure high availability. Helix is a general cluster management library that uses Apache Zookeeper under the covers.

Pinot servers

Pinot servers are the workhorses of a Pinot cluster and they are responsible for both storing the data and answering queries on it. They are instructed to ingest new data by the controller from either a real-time stream like Kafka or by bulk loading data from a distributed file system like GCS/S3/HDFS. Servers store the data in the form of data segments that will be elaborated next. A note on the naming: Pinot just chooses to call them “Servers”, but they are simply Java processes that can be containerized or virtualized. That is, Pinot supports modern infrastructure like Kubernetes and the cloud equally well, in addition to on premise bare metal.

Pinot data distribution

Logically a Pinot table is split into segments. A segment represents a horizontal slice of a table including all of its indices. Each segment can contain many tens of thousands of rows and is usually sized to be a couple of gigabytes. The segments store the data in a columnar compressed manner. Pinot segments support efficient indices and encoding techniques to make querying super fast. These segments are the atomic unit of data and are not re-packaged once created. Pinot segments are deleted once they are past the configured table retention.

All aspects of the data distribution like the segment size, retention, indices, replication factor etc are fully configurable on a per table basis when creating the Pinot table or afterwards.

The controller is responsible for choreographing the initial assignment, replication, and migration of the segments across the servers. It keeps track of the up-to-date physical location of the segments in the form of what it calls a “Routing Table” (not to be confused with the networking routing table).

Pinot Brokers and Querying

The Pinot Broker is the component in Pinot responsible for serving an HTTP POST endpoint for querying via a SQL like language. It is stateless and indeed the cluster can have multiple brokers, scaled corresponding to the query traffic. The controller keeps track of all the live brokers. The client initiates the query by first asking the controller for the list of live brokers and then makes an HTTP POST call to do the query. An example Pinot SQL query might look like: “SELECT name, COUNT(*) from table_name WHERE country = ‘US’ AND time_column_seconds > 1599516412 GROUP BY name ORDER BY COUNT(*) DESC LIMIT 10,”¹ to return the top 10 American names in a table after a certain time point.

The broker proceeds similarly to Presto coordinator in that it first compiles the query into an internal operator tree. The lower part of the operator tree is converted into an internal query for the Pinot Servers. For the example Pinot query above, the servers would do the filtering and the partial aggregation. The broker then proceeds by asking the controller for the current *Routing Table* of the pinot table being queried, to know which segment is on which server. If there are any partitioning columns in the table, then those are used to prune the list of servers to query. The server specific internal query is scattered to all of the Pinot servers as shown in [Figure 3-2](#). The servers then respond back and return partial results, much the same way as Presto workers. Just like the Presto coordinator, the broker stitches the results together by

¹ This is an example of a Pinot SQL query. Pinot also supports an older Pinot-Query-Language PQL query which has a few differences, for example it would have used the keyword “TOP” instead of the ORDER-BY-DESC-LIMIT.

executing the upper part of the operator tree: For the example above, the partial GROUP BYs returned by the servers would be re-aggregated, sorted on the COUNT and the top 10 results retained.

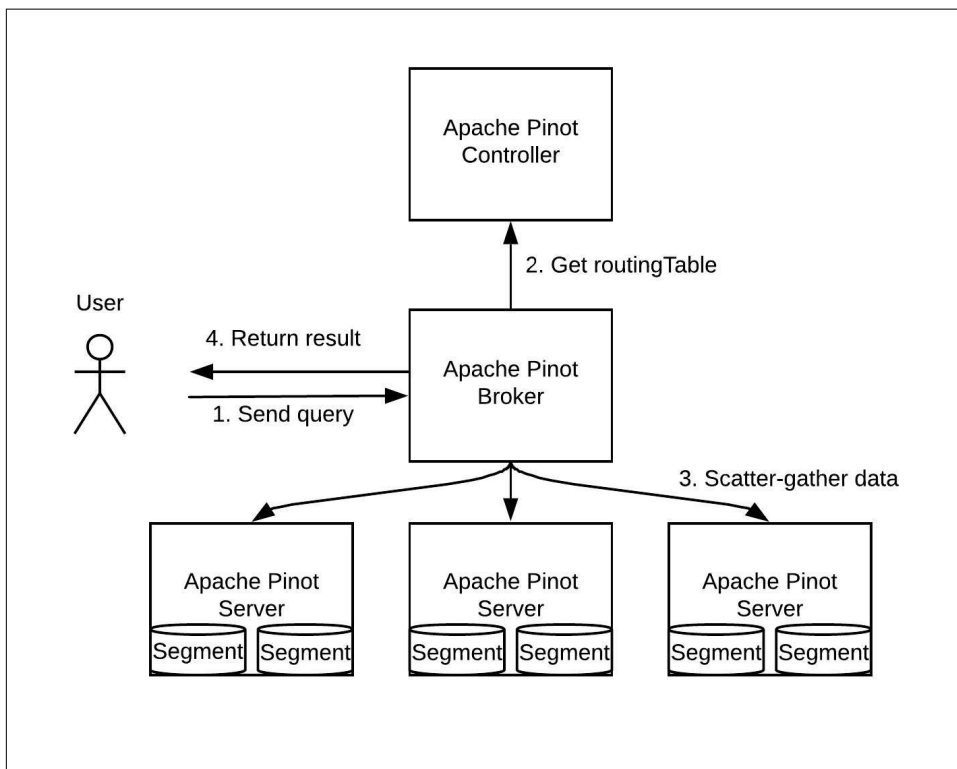


Figure 3-2. Query Processing at the Pinot Broker

Why Use Pinot via Presto?

It's a natural question to ask. Why we should connect Pinot data with Presto querying in the first place? Why would we want to query Pinot via Presto instead of querying it directly via its SQL endpoint?

Consider the common use case of visualization of real-time data contained in your Pinot cluster into a dashboard. Typically you will use a third party tool like Looker, Tableau, Redash or Superset for such visualizations and dashboards. These tools expect full ANSI SQL and ODBC connectivity. They already have very rich and deep support for Presto. Pinot does not yet have full ANSI SQL support, nor does it sup-

port standard database access APIs like ODBC/JDBC. In fact, the only way to integrate these tools natively with Pinot is to write a custom Pinot client for them.²

Another great benefit of accessing Pinot via Presto is that it unlocks advanced SQL capabilities like subqueries, UNIONs and JOINs that are unavailable in Pinot. You are not chained to the single-table query capabilities of Pinot and can instead utilize the full power of Presto SQL.

Accessing Pinot via Presto prevents Data Silos and framework lock in. It allows you to enrich the data stored in Pinot with other data sources. A fairly common use case I encountered at Uber is to store the fact event data in Pinot and use Presto to join it with the dimension tables stored in Hive. As your data usage story evolves, you can easily migrate a table from one framework to another. For example, say you find out that you are doing mostly single key searches in Pinot and that they would be better served by a key-value system like Cassandra. You can load the Pinot table into Cassandra, and just change the catalog name to point to the Cassandra connector. And voila, now the same single key searches are sped up by having them be served by Cassandra.

Lastly, it's important to point out that Pinot isn't the only real-time OLAP system out there. There are other systems like Druid, Elasticsearch etc that can also serve the same use cases as Pinot with a different tradeoff in terms of performance and query capabilities. The great thing is that you can access all of these data sources via Presto: So you can easily play with them at the same time, without having to change your queries each time you want to try out a new system. The users don't see any difference in the queries they write. For all they care, they are simply using Presto via Presto SQL. This flexibility eases the adoption curve and allows you to gradually move your workload from Druid to Pinot, or vice versa !.



Apache Druid is another popular real-time datastore. By using the Presto-Druid connector, you can find similar benefits as those we discuss for Pinot. For Presto-Druid specifics see these [docs](#).

To summarize, accessing Pinot via Presto unlocks the data stored in Pinot and enables it to be used for deep insights and integration with many other third party tools. Having convinced you about how Presto+Pinot is a winning combination, let's dive right into setting up Presto to talk to Pinot.

² The author speaks from first hand experience having tried to integrate Pinot with Superset directly and then realizing that it is better to simply leverage Superset's excellent integration with Presto.

Setting up Presto+Pinot

Having seen how Presto unlocks the true power of Pinot, let's get our hands dirty by wiring them up.

To get started, please choose one of the many options listed in <https://docs.pinot.apache.org/basics/getting-started> for deploying Pinot in your environment. Your choice would depend on how you have deployed Presto. If you are running Presto on bare metal, then the easiest approach would be to run the Pinot Docker on the same machine as the Presto coordinator. Or if you have Presto running in a cloud provider, you can also install Pinot there. The only important consideration is that Presto should have network connectivity to Pinot. Pinot supports installation across all three public clouds, bare metal, Kubernetes and as a stand alone docker container. The Pinot quick start options bundle in Zookeeper and don't require any other dependencies. They come preinstalled with a table `baseballStats` that we will be using to demonstrate Presto-Pinot capabilities.

Connecting a Pinot cluster to Presto

As discussed earlier, the Pinot controller is the brains of the Pinot cluster. So to make Presto access Pinot, all we need to do is to create a Pinot catalog that tells Presto about the controller URIs.

You can connect multiple Pinot clusters to your Presto installation, but each cluster must be added as a separate catalog. For example, say you have a Pinot cluster with controllers running at `controller_host1:9000` and `controller_host2:9000`. You can expose this as a catalog called `mypinotcluster` by creating a file called `etc/catalog/mypinotcluster.properties`, with the contents:

```
# cat etc/catalog/mypinotcluster.properties
connector.name=pinot
pinot.controller-urls=controller_host1:9000,controller_host2:9000
```

Presto will randomly pick one of the above controllers.



The Pinot catalog property file

Like other connectors in the Presto ecosystem, catalogs are configured using a catalog property file. This catalog property file specifies the connector name (“pinot” in this case) and any other connector specific configuration. The Pinot connector has a plethora of configuration options that can be configured by adding them to the above catalog file, or as session properties under the “pinot.” namespace.

Exposing Pinot tables as Presto tables

After adding a Pinot catalog, the tables in the Pinot cluster will automatically show up in Presto. What happens under the hood is the Presto coordinator asks the Pinot controller for all the tables and simply exposes them as Presto tables.

Just to recap, a fully qualified presto table has the form: “Catalog.Schema.TableName”. Pinot does not have a concept of schema, so you can use anything as the schema name. However, for the purposes of output, the Pinot connector states that it has a single schema named ‘default’ containing all of the tables. To refer to a table, you can use any schema name including ‘default’. For example, this is how the preloaded table `baseballStats` bundled with the Pinot distribution shows up. Note that it appears in the schema ‘default’.

```
presto:default> show schemas from pinot;
      Schema
-----
default
information_schema
(2 rows)

Query 20200901_071445_00043_gutrt, FINISHED, 1 node
Splits: 19 total, 19 done (100.00%)
0:00 [2 rows, 35B] [20 rows/s, 351B/s]

presto:default> show tables from pinot.default;
      Table
-----
baseballstats

(1 row)
Query 20200901_071453_00044_gutrt, FINISHED, 1 node
Splits: 19 total, 19 done (100.00%)
0:00 [1 rows, 30B] [12 rows/s, 388B/s]
```



Presto is case insensitive and table names and column names are typically listed in lower case. Pinot on the other hand is case sensitive. Presto will map the case sensitive Pinot table name into an all lowercase version. That is, Presto-Pinot connector will map the Presto table `Baseballstats` (or `baseballstats`) into the case sensitive Pinot version `baseballStats`.

Having seen how Pinot tables appear in Presto, let's now see how the Pinot data types are mapped to Presto types. Most of the Pinot data types have straight forward Presto analogues (for example `STRING` pinot data type corresponds to `VARCHAR` in Presto, `INT` pinot data type corresponds to `INTEGER` in Presto) and so on. Converting Pinot `TIME` and `DATE_TIME` to the Presto `DATE` and `TIMESTAMP` fields

requires a little bit more work, which we will discuss later. Some Pinot types don't have a Presto analogue and are coerced. For example, the Pinot FLOAT type is converted into a Presto DOUBLE and Pinot multi-value types are coerced into a Presto VARCHAR. Pinot columns are typically divided into Metric and Dimension columns. You can GROUP BY on dimension columns, while metric columns are typically aggregated. The listing below shows how to view the Presto schema of a Pinot table.

```
presto:default> describe pinot.any_schema_name_works.baseballstats;
      Column      | Type   | Extra | Comment
-----+-----+-----+-----
homeruns          | integer |       | METRIC
playerstint       | integer |       | METRIC
...
yearid            | integer |       | DIMENSION
hits              | integer |       | METRIC
runsbatteadin    | integer |       | METRIC
...
(25 rows)
```

```
Query 20200901_071514_00046_gutrt, FINISHED, 1 node
Splits: 19 total, 19 done (100.00%)
0:00 [25 rows, 2.37KB] [270 rows/s, 25.7KB/s]
```

Now that we have wired up our Pinot cluster to Presto, we can get to the business of actually querying it !. However, before we do that it's worth talking about how exactly does the Presto-Pinot connector query Pinot.

How Presto queries Pinot

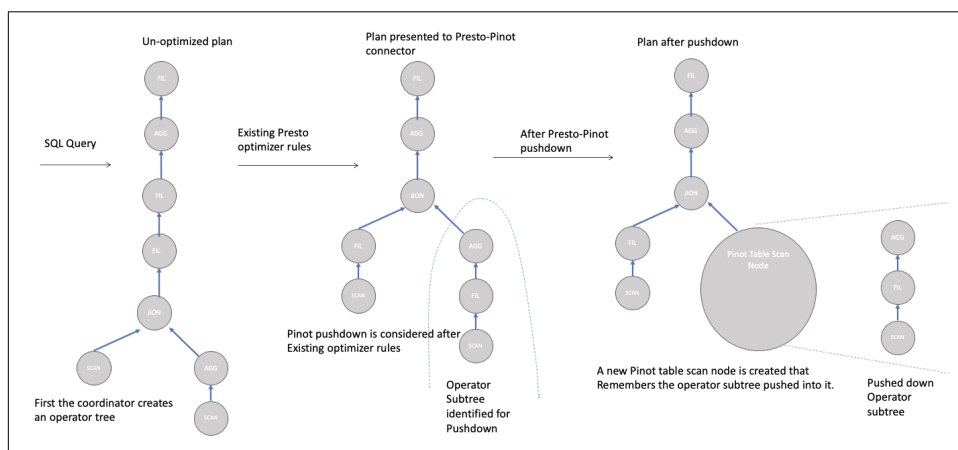


Figure 3-3. Pushdown in Presto-Pinot connector

The Presto-Pinot connector works in consortium with Presto’s query optimizer and attempts to identify the maximal operator subtree that can be pushed down into Pinot. It acts after all of the other Presto optimizer rules have been executed to ensure that it can find good candidates to push down. As shown in [Figure 3-3](#), the pushdown creates a special table scan node in Presto that embeds the operator subtree pushed into Pinot. This special scan node takes care of querying Pinot.

The actual scan can happen in one of two modes depending on the operator tree pushed into the Pinot table scan node as shown in [Figure 3-4](#). Presto prefers querying the Pinot broker if possible, but otherwise Presto can also masquerade as a Pinot broker by directly contacting the Pinot servers and aggregating their results. These two querying modes are referred to as “Direct-to-Broker” and “Direct-to-Server” respectively. The “Direct-to-Broker” mode is preferred because it’s more efficient and allows leveraging some of the optimizations in the Pinot broker, such as partition aware server pruning.

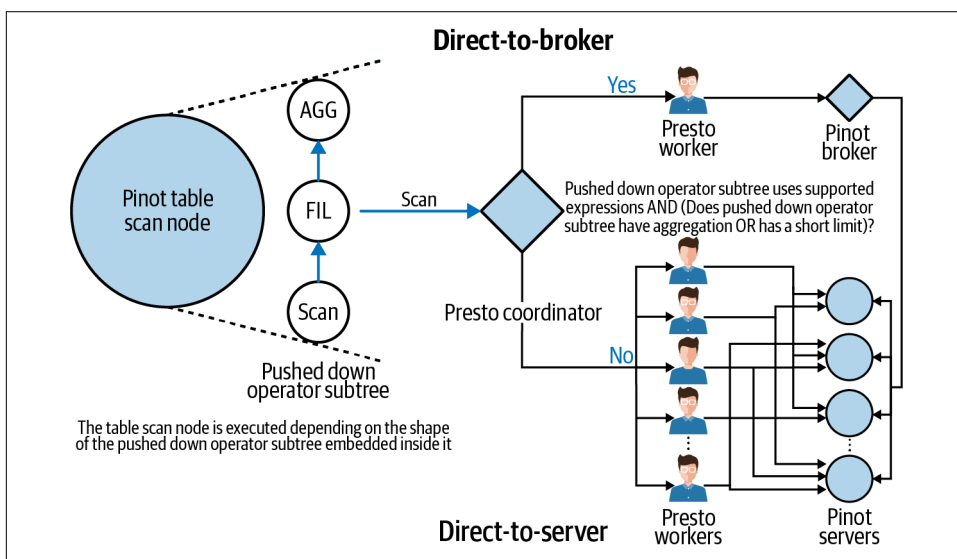


Figure 3-4. Executing the Pinot Table Scan depending on the pushed down operator tree

Presto-Pinot checks for two conditions to decide if it can use the Direct-To-Broker mode. First, it checks if the operator subtree can even be supported by the Pinot broker. The presence of certain unsupported expressions like *substr* can prevent the connector from using Direct-To-Broker. Second, the operator subtree should either include an aggregation or have a “short” limit. A limit is considered “short” if it is below the config value `pinot.non-aggregate-limit-for-broker-queries`, which is set to 25K by default. If these two conditions are satisfied, it proceeds with the Direct-To-Broker mode otherwise falls back to Direct-To-Server.

Direct-to-Broker queries

In this mode, Presto-Pinot constructs a Pinot query for the operator subtree that it wants to push down. A broker is chosen and a single Presto split is created with this constructed query and the chosen broker. The Presto worker executing this singleton split queries the Pinot broker's HTTP endpoint, retrieves the results and converts them into the format expected by Presto.

Direct-to-Server queries

In this mode, Presto-Pinot effectively masquerades as a Pinot broker. But first, it must obtain the *Routing Table* from the real Pinot broker. Armed with the routing table, it knows which segments and servers to query. Presto plans the query as a multiple split query, where each Presto split is responsible for querying no more than a single Pinot server. By default, the connector creates a single split for each segment on the server (configurable via the `pinot.num-segments-per-split` config). All Presto workers execute the splits in parallel. A Presto worker executing the split, will query the specified Pinot server for the specified segments using the Pinot-internal API -- the very same API used between the Pinot broker and the Pinot server. In this mode, the filters are pushed down into the Pinot server-specific query but aggregations are not pushed down.

It's important to note that once the Presto workers have executed the splits, whether in Direct-to-Broker or in Direct-To-Server modes, the rest of the query processing proceeds normally in Presto as dictated by the rest of the operator tree that wasn't able to be pushed into Pinot.

Presto-Pinot querying in action

Having understood the core theory behind how Presto-Pinot queries Pinot, let's get a better feel for it by trying out a few queries. Explaining a Presto query over a Pinot table will print out the query plan being used, including the detail about whether Pinot is being queried in Direct-To-Broker or Direct-To-Server mode.

Let's first consider a simple aggregation query over the previously shown `baseballStats` table. This query simply prints the number of times a player name is repeated in the table.

```
EXPLAIN SELECT playerName, count(1) FROM baseballStats GROUP BY playerName
- Output[playerName, _col1] => [playername:varchar, count:bigint]
  Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
  playerName := playername
  _col1 := count
- RemoteStreamingExchange[GATHER] => [playername:varchar, count:bigint]
  Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
  - TableScan[TableHandle {connectorId='pinot', connectorHandle='PinotTable-
Handle{connectorId=pinot, schemaName=default, tableName=baseballStats, isQuery-
```

```

Short=Optional[true],
expectedColumnHandles=Optional[[PinotColumnHandle{columnName=playerName, data-
Type=varchar, type=REGULAR}, PinotColumnHandle{columnName=count, data-
Type=bigint, type=DERIVED}]],
pinotQuery=Optional[GeneratedPinotQuery{query=SELECT count(*) FROM baseball-
Stats GROUP BY playerName TOP 10000, format=PQL, table=baseballStats, expected-
ColumnIndices=[0, 1], groupByClauses=1, haveFilter=false, isQueryShort=true}]]'
  Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: 0.00}
  count := PinotColumnHandle{columnName=count, dataType=bigint,
type=DERIVED}
  playername := PinotColumnHandle{columnName=playerName, dataType=var-
char, type=REGULAR}

```

The bolded parts of the explain output show that the query has been pushed down entirely into Pinot and effectively translated into a broker PQL query: “SELECT count(*) FROM baseballStats GROUP BY playerName TOP 10000”. You can know that it is sent to the broker by the bit “*isQueryShort=true*.” As you can see, Presto has assumed that the PQL query will return no more than 10000 unique players. If this TOP 10000 is not specified, Pinot will assume we want only the TOP 10 players. This can be changed by changing the connector configuration “pinot.topn-large”, which defaults to 10000.

And now let’s consider something more complex: Let say we want to find the average runs scored by a players team if that player has made any home runs ever. This query is a self join of the baseballStats table with different filters on each side of the join.

```

EXPLAIN SELECT a.playerName, AVG(b.runs) FROM baseballstats a JOIN baseball-
stats b ON a.teamid = b.teamid WHERE a.homeRuns > 0 GROUP BY a.playerName
- Output[playername, _col1] => [playername:varchar, avg:double]
  _col1 := avg
  - RemoteStreamingExchange[GATHER] => [playername:varchar, avg:double]
    - Project[projectLocality = LOCAL] => [playername:varchar, avg:double]
      - Aggregate(FINAL)[playername][$hashvalue] => [playername:varchar,
$hashvalue:bigint, avg:double]
        avg := "presto.default.avg"((avg_36))
        - LocalExchange[HASH][$hashvalue] (playername) => [player-
name:varchar, avg_36:row(field0 double, field1 bigint), $hashvalue:bigint]
          - RemoteStreamingExchange[REPARTITION][$hashvalue_37] =>
[playername:varchar, avg_36:row(field0 double, field1 bigint), $hash-
value_37:bigint]
            - Aggregate(PARTIAL)[playername][$hashvalue_43] =>
[playername:varchar, $hashvalue_43:bigint, avg_36:row(field0 double, field1
bigint)]
              avg_36 := "presto.default.avg"((expr_27))
              - Project[projectLocality = LOCAL] => [player-
name:varchar, expr_27:bigint, $hashvalue_43:bigint]
                expr_27 := CAST(numberofgames_3 AS bigint)
                $hashvalue_43 := combine_hash(BIGINT 0,
COALESCE($operator$hash_code(playername), BIGINT 0))
                - InnerJoin[("teamid" = "teamid_8")]
[$hashvalue_38, $hashvalue_40] => [playername:varchar, numberofgames_3:integer]

```



```

Distribution: PARTITIONED
- RemoteStreamingExchange[REPARTITION]
[$hashvalue_38] => [teamid:varchar, playername:varchar, $hashvalue_38:bigint]
Estimates: {rows: ? (?), cpu: ?,
memory: 0.00, network: ?}

- ScanProject[table = TableHandle
{connectorId='pinot', connectorHandle='PinotTableHandle{connectorId=pinot, sche-
maName=default, tableName=baseballStats, isQueryShort=Optional[false], expected-
ColumnHandles=Optional[[PinotColumnHandle{columnName=teamID, dataType=varchar,
type=REGULAR}, PinotColumnHandle{columnName=playerName, dataType=varchar,
type=REGULAR}]]},
pinotQuery=Optional[GeneratedPinotQuery{query=SELECT teamID, playerName FROM
baseballStats__TABLE_NAME_SUFFIX_TEMPLATE__ WHERE (homeRuns > 0)__TIME_BOUND-
ARY_FILTER_TEMPLATE__ LIMIT 2147483647, format=PQL, table=baseballStats, expect-
edColumnIndices=[0, 1], groupByClauses=0, haveFilter=true,
isQueryShort=false}]]'], projectLocality = LOCAL] => [teamid:varchar, player-
name:varchar, $hashvalue_39:bigint]
Estimates: {rows: ? (?),
cpu: ?, memory: 0.00, network: 0.00}/{rows: ? (?), cpu: ?, memory: 0.00, net-
work: 0.00}
$hashvalue_39 := com-
bine_hash(BIGINT 0, COALESCE($operator$hash_code(teamid), BIGINT 0))
playername := PinotColumnHan-
dle{columnName=playerName, dataType=varchar, type=REGULAR}
teamid := PinotColumnHandle{col-
umnName=teamID, dataType=varchar, type=REGULAR}
- LocalExchange[HASH][$hashvalue_40]
(teamid_8) => [numberOfgames_3:integer, teamid_8:varchar, $hashvalue_40:bigint]
Estimates: {rows: ? (?), cpu: ?,
memory: 0.00, network: ?}
- RemoteStreamingExchange[REPARTITION]
[$hashvalue_41] => [numberOfgames_3:integer, teamid_8:varchar, $hash-
value_41:bigint]
Estimates: {rows: ? (?),
cpu: ?, memory: 0.00, network: ?}
- ScanProject[table = TableHandle
{connectorId='pinot', connectorHandle='PinotTableHandle{connectorId=pinot, sche-
maName=default, tableName=baseballStats, isQueryShort=Optional[false], expected-
ColumnHandles=Optional[[PinotColumnHandle{columnName=numberOfGames,
dataType=integer, type=REGULAR}, PinotColumnHandle{columnName=teamID, data-
Type=varchar, type=REGULAR}]]},
pinotQuery=Optional[GeneratedPinotQuery{query=SELECT numberOfGames, teamID FROM
baseballStats__TABLE_NAME_SUFFIX_TEMPLATE__TIME_BOUNDARY_FILTER_TEMPLATE__
LIMIT 2147483647, format=PQL, table=baseballStats, expectedColumnIndices=[0,
1], groupByClauses=0, haveFilter=false, isQueryShort=false}]]'], projectLocal-
ity = LOCAL] => [numberOfgames_3:integer, teamid_8:varchar, $hash-
value_42:bigint]
Estimates: {rows: ? (?),
cpu: ?, memory: 0.00, network: 0.00}/{rows: ? (?), cpu: ?, memory: 0.00, net-
work: 0.00}
$hashvalue_42 :=
combine_hash(BIGINT 0, COALESCE($operator$hash_code(teamid_8), BIGINT 0))

```

```

teamid_8 := PinotColumnHandle{columnName=teamID, dataType=varchar, type=REGULAR}
numberofgames_3 :=
PinotColumnHandle{columnName=numberOfGames, dataType=integer, type=REGULAR}

```

This query is planned as a join of two Pinot “Direct to Server” queries, as can be seen by the presence of *isQueryShort* = *false*. The left side is a PQL query that has the filter *homeRuns* > 0 pushed down. As dictated by the query, this filter is absent on the right side. Also note how the actual join and aggregation happens in Presto as usual.

Date/Time handling in Pinot vs. Presto

Presto has a **rich set of expressions** to work with timestamps, time zones and time intervals. Date and Time is also of central importance in Pinot. So much so, that each Pinot table is required to have at least one designated timestamp column that needs to be specified when creating the table. Since working with time is so common, we will go through several examples to illustrate some common recipes.

Presto can either work with time stored as a numeric value (like seconds since Unix epoch) or as the newly-introduced Pinot time and date/time types.

Time stored as a numeric type

Below we give examples of each using the Meetup sample data present in Pinot’s distribution, in which *event_time* refers to the event timestamp in milliseconds since epoch

Let’s start with something simple: Just a count of the number of meetup events taking place each day:

```

EXPLAIN SELECT DATE_TRUNC('day', FROM_UNIXTIME(event_time/1000.0)), count(1)
FROM meetupRsvp GROUP BY 1
      Output[_col0, _col1] => [date_trunc:timestamp, count:bigint]
Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
_col0 := date_trunc
_col1 := count
RemoteStreamingExchange[GATHER] => [date_trunc:timestamp, count:bigint]
Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
TableScan[TableHandle {connectorId='pinot', connectorHandle='PinotTableHandle{connectorId=pinot, schemaName=default, tableName=meetupRsvp, isQueryShort=Optional[true],
expectedColumnHandles=Optional[[PinotColumnHandle{columnName=date_trunc, dataType=timestamp, type=DERIVED}, PinotColumnHandle{columnName=count, dataType=bigint, type=DERIVED}]],
pinotQuery=Optional[GeneratedPinotQuery{query=SELECT count(*) FROM meetupRsvp GROUP BY dateTimeConvert(DIV(event_time, 1000.0), '1:SECONDS:EPOCH', '1:MILLI-SECONDS:EPOCH', '1:DAYS') TOP 10000, format=PQL, table=meetupRsvp, expectedColumnIndices=[0, 1], groupByClauses=1, haveFilter=false, isQueryShort=true}]]}]
=> [date_trunc:timestamp, count:bigint]
Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: 0.00}

```

```
count := PinotColumnHandle{columnName=count, dataType=bigint, type=DERIVED}
date_trunc := PinotColumnHandle{columnName=date_trunc, dataType=timestamp,
type=DERIVED}
```

As you can see, this was translated into the Pinot’s *dateTimeConvert* UDF query to the broker. Next let’s count the total number of events in the last 24 hours:

```
EXPLAIN SELECT COUNT(1) FROM meetupRsvp WHERE event_time < CAST(TO_UNIXTIME(current_timestamp - interval '24' hour) * 1000 AS BIGINT)
```

```
- Output[_col0] => [count:bigint]
  Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
  _col0 := count
- RemoteStreamingExchange[GATHER] => [count:bigint]
  Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: ?}
  - TableScan[TableHandle {connectorId='pinot', connectorHandle='PinotTableHandle{connectorId=pinot, schemaName=default, tableName=meetupRsvp, isQueryShort=Optional[true],
expectedColumnHandles=Optional[[PinotColumnHandle{columnName=count, dataType=bigint, type=DERIVED}]]}, pinotQuery=Optional[GeneratedPinotQuery{query=SELECT count(*) FROM meetupRsvp WHERE (event_time < 1598897257351), format=PQL, table=meetupRsvp, expectedColumnIndices=[0], groupByClauses=0, haveFilter=true, isQueryShort=true}]]}] => [count:bigint]
    Estimates: {rows: ? (?), cpu: ?, memory: 0.00, network: 0.00}
    count := PinotColumnHandle{columnName=count, dataType=bigint,
type=DERIVED}
```

In this case, Presto has planned it as a direct-to-broker query, and it has converted the constant expression “cast(to_unixtime(current_timestamp - interval ’24’ hour) * 1000 as BIGINT)” to a number 1598897257351 before sending the query to Pinot.

Time stored as a Pinot TIME or DATE_TIME type

Pinot can also store time in special TIME and DATE_TIME type fields that have a notion of granularities. The Pinot connector, however, only supports the following two combinations:

- A Pinot TIME (or DATE_TIME) field with unit “MILLISECONDS”, format “EPOCH” and granularity “1” will map to a Presto TIMESTAMP type, if the config `pinot.infer-timestamp-type-in-schema = true`. (It’s false by default.)
- A Pinot TIME (or DATE_TIME) field with unit “DAYS”, format “EPOCH” and granularity “1” maps to a Presto DATE type, if the config `pinot.infer-date-type-in-schema = true`. (It’s false by default.)

These restrictions are there because Presto stores time internally as milliseconds since epoch and dates as the full number of days since epoch. Loading a table with any other combination (like, for example, a Pinot TIME field with unit MILLISECONDS, format EPOCH and granularity 1000) would fail and that Pinot table would not be queryable via Presto.

Since the resulting Presto types are already in DATE/TIMESTAMP that Presto natively understands, this eliminates the need of from_unixtime/to_unixtime functions above. The queries above can be written more simply (assuming timestamp_column is the name of the column that maps to a Presto TIMESTAMP column) as follows:

- “Select date_trunc(‘day’, timestamp_column), count(1) from baseballStats group by 1”.
- “select count(1) from baseballStats where timestamp_column < current_timestamp - interval ‘24’ hour”.

The Presto-Pinto connector does not yet fully support timezones, but that feature is in active development.

Troubleshooting Common Issues

These are some of the most common issues I have seen in my experience with the Presto-Pinot connector at Uber, which not only runs some of the largest Presto/Pinot clusters, but also developed the Presto-Pinot connector. I also share solutions to these issues.

A large class of issues stems from Presto-Pinot incorrectly choosing Direct-To-Server or Direct-To-Broker query modes, when it really should be using the other mode. Lets cover the problems with choosing these two modes incorrectly, in turn.

Direct-To-Broker query mode causing issues

The heuristic to choose Direct-To-Broker can sometimes misfire and cause issues. There are two common symptoms associated with this:

- The query OOMs out. This typically happens when the aggregation is not small, like when you mistakenly group by on a column that has a high cardinality. This blows up the result set that Presto fetches from Pinot, and requires more memory than what Presto can afford to the singleton Direct-To-Broker split.
- The query times out after 15 or 30 seconds: The query is too heavy weight for the Pinot broker. It’s either doing some complex aggregation or simply scanning too many segments.

The fix is simple: Force the query to go via the Direct-To-Server mode. The recommended way to fix this is by using the session property `pinot.forbid_broker_queries = true` on a per query basis. A heavier hammer would be to disable Direct-To-Broker globally by using the catalog config property `pinot.forbid-broker-queries = true`.

Direct-To-Server query mode causing issues

Recall that Direct-To-Server is usually a fallback if the Direct-to-Broker cannot be chosen. It typically results in the following symptoms:

- Increased load on Pinot clusters due to Presto: This will manifest as other Pinot queries timing out or getting rejected. This is typically caused by the use of Direct-To-Server querying, which is bypassing the rate limits and query processing smarts (like segment pruning) put in place by the Pinot broker.
- Presto-Pinot queries take a long time to finish: A Presto-Pinot query creates a split per segment per server. This is usually beneficial in that it increases the parallelism and the query finishes faster. But sometimes this can create a lot of splits and those splits are stuck waiting for resources in the Presto's queue.

One approach to fix these issues is to make Direct-To-Server a bit easier on Pinot by increasing the number of segments per split. This can be done by increasing the session property `pinot.num_segments_per_split` on a per query basis. Setting `pinot.num_segments_per_split` to 0 is equivalent to setting it to infinity: It creates just a single split for all the segments on that server.

If the above does not help, then we have to get our hands dirty and try to rework the query such that Direct-To-Broker can be chosen. There are two common remedies for this:

- Consider removing any expressions that are preventing Direct-To-Broker. Perhaps you are using a UDF that is not supported by the Pinot Broker. This may be fixed by a simple local rework of the query: For example, consider modifying something like `"SELECT COUNT(lower(playerName)) FROM baseballStats"` to just `"SELECT COUNT(playerName) FROM baseballStats"`, to allow the query to be planned as a broker query. The two forms would return the same result because Pinot does not have the concept of NULLs, and thus `lower(playerName)` will always be non NULL, and thus can be optimized away from inside of COUNT.
- Introduce an artificial NO-OP LIMIT: Recall that Presto-Pinot defines a short non-aggregate query as a query having a limit of under 25K. This threshold is configurable by the configuration property `pinot.non-aggregate-limit-for-broker-queries` (or alternatively the session property `pinot.non_aggregate_limit_for_broker_queries`). Queries without a limit are treated as "large" and are thus never planned as a direct-to-broker. Introduce a fake limit under this threshold to force direct-to-broker: e.g, change `"SELECT playerName from baseballStats where playerState = 'CA'"` to `"SELECT playerName from baseballStats where playerState = 'CA' LIMIT 24000"` if you know that the number of players actually in California is less than 24K, thus making the limit be a No-Op.

If Direct-To-Server query mode causes many problems for you, can you also consider forbidding it outright with the config `pinot.forbid-segment-queries = true` in the catalog property file. With this configuration, queries will either be planned as Direct-To-Broker or fail right away.



“Open Source Community”

Like Presto, Pinot is an open source project and welcomes your involvement. Both Pinot itself and the Presto-Pinot connector are being actively developed and the community would love your feedback and questions in order to improve them. There is a dedicated troubleshooting channel on the [Pinot slack channel](#) to help you integrate it with Presto. For more in-depth information on Pinot, you can head over to pinot.apache.org or check out their SIGMOD 2018 paper [Pinot: Realtime OLAP for 530 Million Users](#)

Summary

This chapter introduced you to extending Presto for real-time business insights using real-time datastores. We walked through an implementation with Presto and Pinot via the Presto-Pinot connector.

About the Author

[Devesh Agrawal](#) is a former Software Engineer at Uber’s Presto team where he led the development of the [Pinot connector with advanced pushdown support](#) in Presto and other latency optimizations to enable real-time serving use cases via Presto. Before flirting with low latency, he dealt with some of the largest queries as a tech-lead of Facebook’s fork of Hive. He is fortunate to have started his database career at Netezza, one of the grandfathers of all modern day analytical engines.

About the Authors

Vivek Bharathan is the Cofounder and Principal Software Engineer at Ahana. Previously, Vivek was a Software Engineer at Uber where he managed Presto clusters with more than 2,500 nodes, processing 35PB of data per day, and worked on extending Presto to support Uber's interactive analytics needs. Prior to Uber, Vivek was an early member of the query-optimizer team at Vertica Systems and made several contributions to the core database engine and the Vertica ecosystem. Earlier in his career at the Laboratory for Artificial Intelligence Research, he developed emerging technologies in decision-support systems and reasoning systems. His Presto contributions include the pushdown of partial aggregations. Vivek holds a M.S. in Computer Science and Engineering from The Ohio State University.

David E. Simmen is Cofounder and CTO of Ahana, overseeing Ahana's technology strategy and driving product innovation. David joined Ahana most recently from Apple where he engineered iCloud database services. Prior to Apple, he was Chief Architect with Splunk and named the first Fellow in the company's history. Prior to Splunk, David was Engineering Fellow and CTO of Teradata Aster. Earlier in his career, David was a Senior Technical Staff Member (STSM) at IBM Research. David is also a named inventor on 37 U.S. patents and has 15 publications to his name.

George Wang is Cofounder and Principal Engineer of Ahana. Previously, as a Senior Software Engineer, George built numerous key features and innovations to support a high throughput and low-latency compute engine for AnalyticDB, a Presto-based cloud service at Alibaba. He was a member of the Presto Foundation Outreach Committee while at Alibaba. Prior to that, George was a Staff Software Engineer at IBM for over 10 years where he developed multiple features and enhancements on DB2 for z/OS. George holds a M.S. in Software Engineering from San Jose State University and B.S in Computer Science and E.E. from UCLA.