From Batch Processing to Real Time Analytics: Running Presto® at Scale

Zhenxiao Luo*, Lu Niu‡, Venki Korukanti§§, Yutian Sun†, Masha Basmanova†, Yi He‡, Beinan Wang*, Devesh Agrawal§, Hao Luo*, Chunxu Tang*, Ashish Singh‡, Yao Li*, Peng Du§, Girish Baliga§, Maosong Fu*
*Twitter, †Facebook, ‡Pinterest, §Uber
Email: rooservelt.luo@gmail.com

***Abstract*—Presto is an open source distributed query engine used widely at Facebook, Uber, Twitter, Pinterest, and many other internet companies. Since open sourced in 2013, the Presto community has made several rounds of design and implementations, to support a variety of use cases, including interactive analytics, real time reporting and dashboard, ETL workloads, A/B testing, monitoring and alerts, etc. In this paper, we'd like to introduce some of the most important features and performance improvements the open source Presto community made in recent years, which enables companies running Presto at scale, supporting millions of queries per day, with hundreds of thousands of machines. Specifically, how Presto provides unified SQL on heterogeneous storage systems without data copy; how Presto deals with complex data, including nested columnar data and schema evolution; How Presto supports geospatial queries efficiently, and how file list cache works in Presto. We also talk about cluster federation, and Presto on cloud. Experimental results and our production experience could help others running interactive SQL systems at scale.**

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

I. Introduction

Presto [1] is an open source distributed query engine used widely at Facebook, Uber, Twitter, Pinterest, and many other internet companies. Presto's performance, scalability, and reliability features make it a good fit for a variety of use cases, including real time dashboards, A/B testing, ad-hoc analytics, and warehouse ETL jobs. During recent years, the industry is leveraging Presto for more challenging use cases, e.g. geospatial analytics, processing complex data efficiently, unified SQL on heterogeneous storage systems, running Presto on cloud. To address the challenges, the Presto community contributed the following design and implementations:

- It is not uncommon for big companies to have multiple storage systems. Ideally, engineers, data analysts, and product managers would love to run SQL analytics on a unified view of all data, no matter where the data resides. While, data copy between different storage systems is computationally expensive. Presto meets the needs in an efficient way. Presto connectors enable unified SQL on heterogeneous storage systems without data copy. With predicate pushdown, limit pushdown, projection pushdown, and aggregation pushdown, Presto connectors leverage the power of underlying systems, and provide unified SQL with on demand data streaming.
- Nested data is used widely in big companies. While, most systems at scale could not handle nested data processing efficiently. We will share our journey of improving Presto performance for highly nested data. With nested column pruning, columnar reads, predicate pushdown, dictionary pushdown, lazy reads, and vectorized reads, we improved Presto performance by more than 10X. The new version of Presto could process highly nested data at the speed of flattened data. Furthermore, we also talk about how Presto supports schema evolution for nested data, which provides flexibility to data consumers.
- There are increasing needs to use Presto for geospatial analytics. To support geospatial queries efficiently in Presto, we build QuadTree [2] on the fly for geospatial queries, which achieves more than 50X speedup.
- It is challenging for one single Presto cluster to scale larger than 1000 machines. We design and implement Presto gateway, which provides cluster federation, supporting multiple clusters with a unified view.
- Cache is a common technique to improve query performance. We will discuss how Presto leverages file list cache, and file footer cache to improve performance.
- With graceful expansion and shrink, Presto could leverage the elasticity when running on clouds. We implement *PrestoS3FileSystem*, which provides File System interface on top of AWS S3 [3]. A number of performance improvements are implemented in *PrestoS3FileSystem*, e.g. Lazy seek, exponential backoff, etc.

Section II talks about Presto use cases at Uber, Twitter, Facebook, Pinterest, and other companies. Section III is a general introduction to Presto. In section IV, we talk about Presto connectors, which provides unified SQL on heterogeneous storage systems. Section V is about how Presto deals with complex data, including nested columnar data and schema evolution. In section VI, we discuss how to support geospatial queries in Presto efficiently. Section VII is about caching in Presto. Section VIII covers how to run cluster federation, to achieve better scalability for Presto. In section IX, we talk about running Presto on cloud, including graceful expansion and shrink, and AWS S3 optimizations. Section X shows our experimental results. We outline key related work in section XI. Finally, we will share our experience.

§Author was affiliated with Uber during the contribution period

II. Presto use cases

A. Uber use case

Uber is running more than 10 Presto clusters [14], with more than 3 thousand machines, supporting more than 2 million queries per day, with more than 10 thousand weekly active users. Presto is used widely at Uber, from marketplace pricing, growth marketing, city operations, geospatial analytics, to ad-hoc querying, reporting, dashboards, etc. Uber data is highly nested, with Apache Parquet™ as its default file format. More than 50 Petabyte of Apache Hadoop®[1] Distributed File System data are processed by Presto daily.

B. Twitter use case

Twitter is running Presto both on premise and on Google Cloud [25]. 5 Presto clusters are running on around 4 thousand machines. Twitter has a lot of nested data, with Apache Parquet and Lzo-thrift format. Presto is used mostly for interactive analysis. There are around 40 thousand Presto queries per day. Around 50 Petabyte of data are processed by Presto daily.

C. Facebook use case

Facebook is running thousands of Presto nodes across several data centers, processing hundreds of petabytes of data and quadrillions of rows per day [1]. Presto is used widely at Facebook, including interactive and BI queries, long-running batch extract-transform-load(ETL) jobs, end-user facing high performance dashboards, SQL analytics for multiple internal NoSQL systems, and Facebook's A/B testing infrastructure.

D. Pinterest use case

Pinterest has a cloud native infrastructure, where data are stored in Amazon S3, and Presto clusters are built on top of Amazon Web Services (AWS) EC2 [17]. Presto clusters are comprised of a fleet of 1000+ r5.12xl EC2 instances, which has hundreds of TBs memory and tens of thousands vcpu cores. Within Pinterest, there are close to thousands of monthly active users (out of total 2000+ Pinterest employees) using Presto, who run multi-million queries on these clusters per month.

Presto is also widely used at Linkedin [26], Netflix [19], Airbnb [27], and many other companies. The Presto open source community [28] has 19,433 commits, with 4.4 thousand forks, 12.9 thousand stars, and more than 266 releases.

III. How Presto works

Each Presto cluster has one *coordinator* and multiple *workers*. As shown in figure 1, Presto coordinator parses incoming SQL, and tokenizes it into *Abstract Syntax Tree*(AST). Analyzer generates logical plan from *Abstract Syntax Tree*(AST). Then optimizers run several rounds of optimizations, and finally generate a physical plan. The fragmenter divides the plan into fragments. Each running plan fragment is called a stage, which could be executed in parallel. Stage consists of tasks, which are processing one or many splits of input data. Scheduler assigns tasks on worker execution slots. Some workers are scanning files, some workers are streaming data from underlying connectors, and some workers are running SQL aggregations, joins, etc.

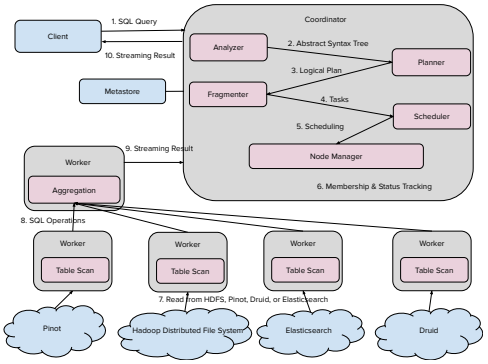

Fig. 1. How presto works internally

Presto supports columnar file formats, including Parquet [4] and ORC [5]. Internally, Presto is a vectorized engine, which processes a bunch of in memory encoded column values vectorized, instead of row by row. Presto leverages ASM [6] to do dynamic code generation for expression evaluation. Although Presto could spill intermediate results to disk, most Presto deployments are doing in memory only processing.

IV. Presto connectors

Presto has a connector interface and implementations to run SQL queries on heterogeneous storage systems. In big companies like Uber, Twitter, and Pinterest, it is not uncommon to have multiple storage systems, each serving its own specific needs. For example, Uber is leveraging Apache Pinot™ [7] for real time streaming processing, Elasticsearch [8]for real time monitoring, and Hadoop Distributed File System(HDFS) [9] for batch analysis. Twitter is running Apache Druid™ [10] for real time analytics, Hadoop Distributed File System(HDFS) and Google Cloud Storage [11] for batch analysis. Pinterest is running Druid for real time monitoring, and AWS S3 for batch analytics. There are many other storage systems serving online traffic and transactions, e.g. MySQL [12] is used widely in all companies with transaction support.

Users have the need to run SQL analytics on a unified view of all data. For example, it is desirable to join Hadoop batch data with Pinot real time data to get fresh Uber Eats reports. Traditionally, pipelines are created to copy data from one system to another. Data engineers are maintaining pipelines to copy data from MySQL to Hadoop, from Elasticsearch to Hadoop, and from Pinot or Druid to Hadoop. While, this data copy solution is not desirable:

- Data copy is computationally expensive. It is not uncommon to get hours of data copy latency before data becomes queryable. Data copy also consumes huge network bandwidth.

[1]Apache®, Apache Hadoop®, Apache Parquet™, Apache Pinot™, Apache Druid™ are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

- maintaining data copy pipelines is resource intensive. Engineers have to spend precious time carefully implementing the copy and transformation logic. Whenever system upgrades or API changes, the pipelines have to be re-implemented. Engineers also have to deal with fault-tolerance, backfill, etc.
- maintaining multiple copies of the same data is resource intensive. A majority of users only need interactively exploratory of data.

Presto connector solves the problem gracefully. Presto connector provides:

- *ConnectorMetadata*, which defines schemas, tables, columns etc.
- *ConnectorSplitManager*, which defines how Presto divide the underlying data into splits, and process them in parallel.
- *ConnectorSplit*, which defines one processing unit, or one shard of underlying data.
- *ConnectorRecordSetProvider*, which defines upon getting data streams from underlying systems, how Presto parse and transform them into Presto engine.

To get a unified view of all data, Presto connector introduces *catalog.schema.table* for each table. *catalog* marks connector name. For example, in *Presto-MySQL-connector*, a table's qualified name is: *mysql.schemaName.tableName*. Similarly, in *Presto-Druid-connector*, a table's qualified name is: *druid.shcemaName.tableName*. In *Presto-Elasticsearch-connector*, we map each Elasticsearch index into a table. Each Elasticsearch field is mapped into a column. *Presto-Pinot-connector* and *Presto-Druid-connector* are more straightforward, where Pinot table and Druid table are mapped into Presto table.

A. Pushdown

An implemented Presto connector enables users to run SQL analytics on a unified view of all data, without data copy. Users could join Hadoop data with MySQL data using *Presto-Hive-connector* and *Presto-MySQL-connector*, no need to copy any data between Hadoop and MySQL. Hadoop data and MySQL data are streamed in Presto pages into the Presto engine. Presto did the join and stream results to end users.

When Presto is streaming data from connectors, it is desirable to pushdown projections, predicates, and limits to underlying storage systems. For example, it is desirable to let MySQL only stream filtered, projected, and limited rows into Presto, instead of streaming the whole table into Presto. Presto connector has mechanisms to support projection pushdown, predicate pushdown, and limit pushdown. All presto connectors have implemented these pushdown techniques.

B. AggregationPushdown

Druid and Pinot are real time systems, which have in memory bitmap indices, inverted indices, pre-aggregations or dictionaries, enabling sub-second query latency. It is desirable to pushdown aggregations to *Presto-Druid-connector* and *Presto-Pinot-connector*, so that only aggregated results are streamed into the Presto engine. This not only greatly improves query latency, as Druid and Pinot could leverage their in memory data structure to speedup aggregation queries, but also saves network bandwidth. Instead of streaming filtered data into Presto engine, and letting Presto do the aggregation, with aggregation pushdown, Presto connector only streams aggregated data into Presto engine, and lets connectors do the aggregations.

A flexible approach would be to push down the entire expression which is currently represented as an *Abstract Syntax Tree*(AST) to Presto connectors. One problem with this approach is that the AST evolves over time such as when new language features are added. Additionally the AST does not contain type information as well as enough information to perform function resolution. We resolve this by storing function resolution information in the expression representation itself as a serializable *functionHandle*. This makes it possible to consistently reference a function when we reuse the expressions containing the function.

We replaced Presto's old *Abstract Syntax Tree*(AST) based expression representation with a new representation called *RowExpression*. [13] *RowExpression* is completely self-contained and can be shared across multiple systems. The new representation has several subtypes shown in table 1.

With *RowExpression*, we could pushdown arbitrary sub-expressions to Presto connectors. Pushdown optimizations could be implemented for each connector as a connector specific optimizer rule, based on the connector's storage characteristics, e.g. indices, sort order, in memory data cache. In *Presto-Druid-connector* and *Presto-Pinot-connector*, we implemented aggregation pushdown, which will execute aggregations in Druid and Pinot, and only stream aggregated results to Presto, as shown in figure 2.

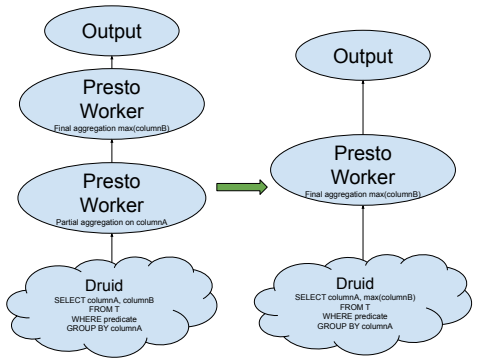

Fig. 2. Aggregation pushdown in presto connector

Generally, Druid and Pinot could support sub-second query latency, while they only have limited support for joins and subquery. Presto connectors bridge the gap between sub-second query latency, and full SQL functionalities. With Presto connectors and aggregation pushdown, users could get sub-second query latency via *Presto-Druid-connector* and *Presto-Pinot-connector*, and get full SQL support, with joins, subquery functionalities in Presto.

TABLE I
SELF CONTAINED ROWEXPRESSIONS

ExpressionType	*Represents*
ConstantExpression	Literal values such as (1L, BIGINT), (”string”, VARCHAR)
VariableReferenceExpression	Reference to an input column and a field of the output from previous relation expression.
CallExpression	Function calls, which includes all arithmetic operations, casts, UDFs.
SpecialFormExpression	Special built-in function calls. E.g. IN IF, IS_NULL, AND, DEREFERENCE
LambdaDefinitionExpression	Definition of anonymous (lambda) functions. E.g. (x:BIGINT,y:BIGINT):BIGINT → x+y

We also implemented *Presto-Iceberg-connector* [21] and *Presto-Hoodie-connector* [20], which enables Presto querying update-able data lakes.

V. DEALING WITH COMPLEX DATA TYPES

Production data is always nested. To leverage columnar storage and deal with nested data, Parquet is popular as the default columnar file format.

A. Schema Evolution

Nested structs are used widely at Uber [14], Twitter [25], and Pinterest [17] , where users define one high level column with struct type. The struct consists of 20 or sometimes up to 50 fields. Each field could be another struct, which has subfields inside. It is not uncommon to see more than 5 levels of nesting. In addition, users are making schema changes to structs. Some users are adding fields to structs, some users are deleting fields from structs. We have to make company wide schema evolution rules, so that the query engine could return meaningful results with frequent schema updates.

We allow adding new fields to existing struct. When querying newly added fields in old data(before the fields are added in the schema), Presto will return null. Removing existing fields from struct is also allowed. When data is continuously ingested into the already removed field, Presto just ignores them.

Field rename and type change are not allowed. In Parquet, field name is used to identify metastore schema and Parquet file schema. Field name change will trigger schema mismatch between metastore schema and Parquet file schema. In addition, Presto is type strict, we do not allow automatic type coercion when querying Parquet via Presto.

Schemas are managed as a service outside of Presto, which tracks different versions of schemas, enforces schema evolution rules, and guarantees schema matching between Parquet file schema and metastore schema.

B. Parquet

Columnar file formats, e.g. Parquet and ORC, enables Presto to answer queries more efficiently. By not having to scan and discard unwanted data in rows, columnar storage saves disk space and improves query performance for larger data sets. Another benefit of Parquet is, Parquet is storing nested fields as separate columns on disk. This gives us the opportunity not to scan unwanted fields even within the same struct.

In Parquet, data is first horizontally partitioned into groups of rows, then within each group, data is vertically partitioned into columns. Data for a particular column is stored together via compression and encoding to save space and improve performance. Each Parquet file has a footer that stores codecs, encoding information, as well as column-level statistics, e.g., the minimum and maximum number of column values.

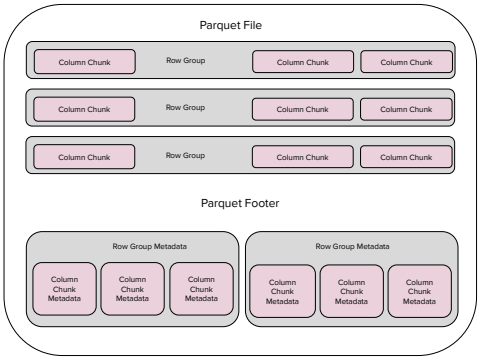

Fig. 3. Parquet is an open source columnar storage format, used widely at Uber, Twitter, and Pinterest, partitioning data horizontally into rows and then vertically into columns for easy compression

C. Old Parquet Reader

On a theoretical level, Parquet was the perfect match for our Presto architecture, but would this magic transfer to our system's columnal needs? Actually, Parquet is supported in Presto using the original open source Parquet reader. While working well with open source Presto, this reader neither fully incorporates columnar storage nor employs performance optimizations with Parquet file statistics, making it ineffective for our use case.

To tackle this performance issue, we developed a new Parquet reader for Presto to leverage the potential of Parquet in our data analytics system. Below is an example query to determine which drivers to target in a specific city on a given date based on expected rider demand:

```
SELECT base.driver_uuid FROM
rawdata.schemaless_mezzanine_trips_rows
WHERE datestr = '2017-03-02'
AND base.city_id in (12)
```

In this scenario, the nested table *rawdata.schemaless_mezzanine_trips_rows* stores more than 100 terabytes of raw trip data in Parquet. Using the above example, we will demonstrate how queries are processed using both the original open source reader and an open source reader of our own invention.

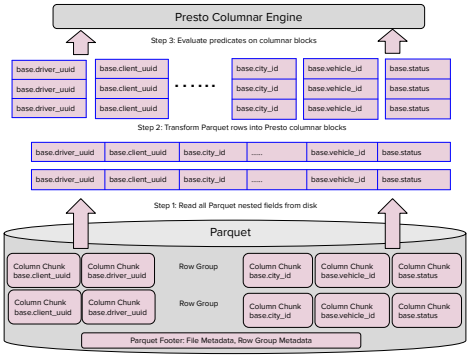

Fig. 4. The legacy open source Parquet reader does not fully incorporate columnar storage, making it inefficient to analyze data at scale

The original reader conducts analysis in three steps: (1) reads all Parquet data row by row using the open source Parquet library; (2) transforms row-based records into columnar Presto blocks in-memory for all nested columns; and (3) evaluates the predicate (base.city_id=12) on these blocks, executing the queries in our Presto engine.

To accommodate data's size and scale at Uber, Twitter, and Pinterest, we created a new open source Parquet reader that uses memory and CPU more efficiently. This new reader implements the following optimizations geared towards enhancing performance and speeding up querying.

D. Nested Column Pruning

One way the new reader optimizes querying is by skipping over unnecessary data, referred to as nested column pruning. As the name suggests, this optimization is most effective when used with nested data. The new reader executes nested column pruning in three steps: (1) read only required columns in Parquet; (2) transform row-based records into columnar blocks; and (3) evaluate the predicate on columnar blocks in the Presto engine.

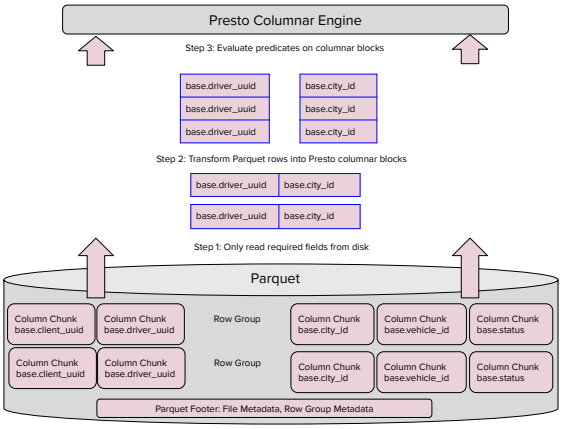

Fig. 5. Brand new Parquet reader can skip over unnecessary data via nested column pruning

E. Columnar Reads

The new reader can also read columns in Parquet directly instead of row-by-row and then execute a row-to-column transformation, which speeds up querying. It executes columnar reads in two steps: (1) read only required columns in Parquet and build columnar blocks on the fly, saving CPU and memory to transform row-based records into columnar blocks, and (2) evaluate the predicate using columnar blocks in the Presto engine.

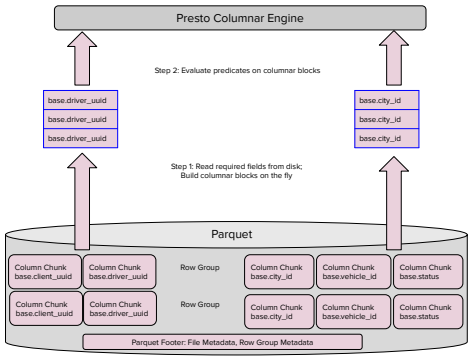

Fig. 6. Brand new Parquet reader enhances querying by reading columns directly as opposed to row-by-row

F. Predicate Pushdown

With our new reader, we can evaluate SQL predicates while scanning Parquet files. By using Parquet statistics, we can also skip reading parts of the file, thereby saving memory and streamlining processing. Predicate pushdowns are primarily used for "needle in a haystack" queries. The new reader executes predicate pushdowns by merging three actions into one step: simultaneously read the required columns in Parquet, evaluate columnar predicates on the fly, and build columnar blocks. In this scenario, the reader skips reading a group of rows if the predicates do not match to the one being queried.

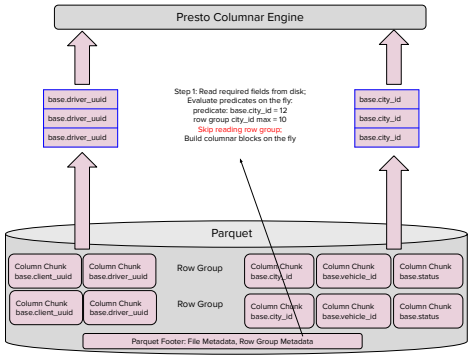

Fig. 7. Predicate pushdowns allow us to skip reading Parquet row groups to save disk IOs. In this example, the query is looking for city_id = 12, one row group city_id max is 10, new Parquet reader will skip this row group

G. Dictionary Pushdown

Even if Parquet statistics match the predicate, we can read the dictionary page for each column to determine whether the dictionary can potentially match the predicate. If not, we can skip reading that row group. Like predicate pushdowns, dictionary pushdowns make querying faster and are most effective for needle in a haystack queries. Dictionary pushdowns are also executed by our new reader in only one step: read required columns in Parquet, evaluate columnar predicates on the fly and build columnar blocks. Similar to predicate pushdowns, dictionary pushdowns enable the reader to skip reading groups of rows if dictionary values do not match the predicate.

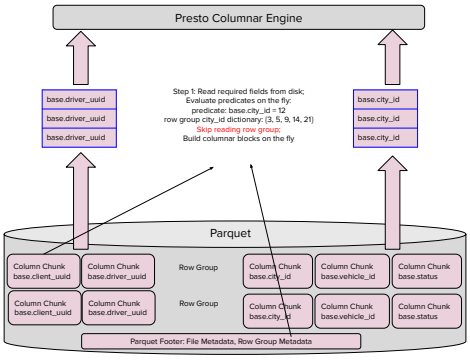

Fig. 8. Like predicate pushdowns, dictionary pushdowns are executed in one step that simultaneously reads and evaluates data columns while building columnar blocks for our Presto engine. In this example, the query is looking for city_id = 12; since one row group's city_id dictionary includes the IDs 3, 5, 9, 14, 21, the new reader will skip this group

H. Lazy Reads

Our reader can also be programmed to read projected columns as lazily as possible. This means that we read projected columns only when they match the predicate, thereby speeding up our querying. Lazy reads are executed in a single step: read the required columns in Parquet, evaluate columnar predicates on the fly, and build columnar blocks only if the predicate matches.

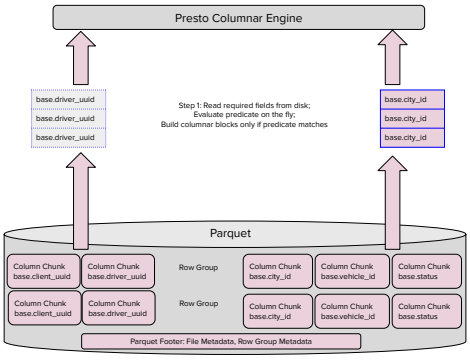

Fig. 9. Lazy reads are performed only when they match the predicate, saving CPU and memory

I. Vectorized Reader

Furthermore, we implement a vectorized reader for parquet. Instead of reading *repetition level*, *definition level*, and *value* one by one, a vectorized parquet reader batch reads 1000 triplets of *repetition level*, *definition level*, and *value*. The decoder state is kept in registers, so that only one end of stream check is needed, with only one status update. For dictionary encoding, the dictionary is cached, so that dictionary lookups are saved. Vectorized parquet reader could seek to non-nullable and non-nested value directly, saving unnecessary reading of *repetition level* and *definition level*.

Since putting our new Parquet reader into production, data is processed anywhere from 2-10x faster compared to when we used the original open source reader.

J. Native Parquet Writer

Presto is also running ETL jobs at Facebook, Uber, Twitter, and Pinterest. It is important to optimize write performance for Presto. The legacy Presto Parquet writer iterates each columnar block in a page and reconstructs every single record, then it consumes each individual record and writes value bytes to Parquet pages. The old Parquet writer was adding unnecessary overhead to convert Presto's columnar in-memory data into row based records, and then doing one more conversion to write row based records to Parquet's columnar on disk file format. The unnecessary data transformation adds significant performance overhead, especially when complex data types are involved such as nested structs.

We design and implement a brand new native Parquet writer for Presto [15], which writes directly from Presto's in-memory data structure to Parquet's columnar file format, including data values, repetition values, and definition values. The native Parquet writer significantly reduces CPU and memory overhead for Presto.

VI. Geospatial Queries

One of the distinct challenges for Uber is analyzing geospatial data at scale. City locations, trips, and event information, for instance, provide insights that can improve business decisions and better serve users. Geospatial data analysis is particularly challenging, especially in a big data scenario, such as computing how many rides start at a transit location, how many drivers are crossing state lines, and so on. For these analytical requests, we must achieve efficiency, usability, and scalability in order to meet user needs and business requirements

A. Geospatial Data Model

Modeling geospatial data has distinct complexities. To model real-world cities and trips into simplified shapes and points represented in big data tables, we use the Well-Known Text (WKT) [29] used in ESRI Geometry API [30] to represent geometries.

A point represents a single location in a two-dimensional space. Internally, we store each point as a pair of (longitude, latitude). e.g.:

POINT (77.3548351 28.6973627)

We simply store a polygon as a collection of points, such that the start point and the end point match. e.g.:

POLYGON ((36.814155579 −1.3174386070000002,
36.814863682 −1.317545867,
36.814863682 −1.318221605,
36.813973188 −1.317910551,
36.814155579 −1.3174386070000002))

B. Geospatial Use Cases

At Uber, geospatial data is organized as geofences, where a geofence is either a polygon or a multi-polygon. We provide internal tools to create, edit, and delete geofences. Every few minutes, any geofence changes will be dumped into a Hadoop table, which is queryable by Presto. In this system, we have a trips table, which records trip start points and endpoints, as well as the cities table, which contains the city identification column city_id and its corresponding geofence column geo_shape.

Geospatial data is used for promotions, driver supply identification, and events regulation, among other use cases. For example, Uber could launch a promotion for free basketball tickets for riders who are taking Uber trips to a Warriors game. In this scenario, our engineer would create a new geofence that contains Oracle Arena Stadium, then write a Presto query to target all users who are taking trips to the stadium before the game for the promotion, and, then, randomly select the winner, as portrayed in figure 10.

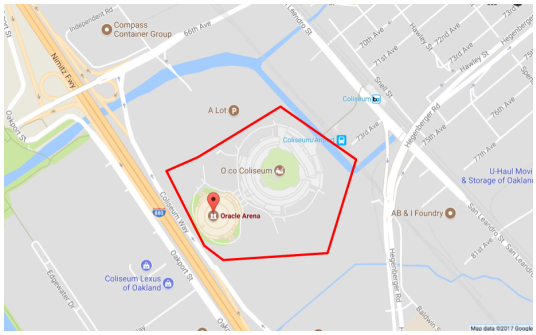

Fig. 10. An example geofence containing Oracle Arena Stadium

C. Challenges

To discern how many trips occur within a given city on a specific date, we run the following SQL query:

```
SELECT c.city_id, count(*)
FROM trips_table as t
JOIN city_table as c
ON st_contains(
    c.geo_shape,
    st_point(t.dest_lng, t.dest_lat)
)
WHERE datestr = '2017-08-01'
GROUP BY 1
```

st_point is the standard geo function to construct a two-dimensional point using longitude and latitude. *st_contains* is the standard geo function to compute whether a point is within a geo shape.

This query needs to compute *st_contains* for each point and geofence pair. For a real city, it is not uncommon to see its geofence composed of hundreds or thousands of points. The time cost of executing *st_contains* for one pair of point and geofence is proportional to the number of points in the geofence. Given that millions of Uber trips are requested each day across hundreds of cities, this simple query could cost hundreds of millions of *st_contains*, which in turn computes hundreds of thousands Point-Point operations. One simple query with a brute force Hive MapReduce execution could take days to complete!

D. QuadTree

This huge time commitment is insufficient for Uber's use case. To ensure we provide optimal trip experiences for our users, we need to optimize how we processed this data.

Quadtrees represent a partition of space in two dimensions by decomposing the region into four quadrants, sub-quadrants, and so on until the contents of the cells meet some criterion of data occupancy. For example, in figure 11, we built QuadTree to index a 4X4 square space.

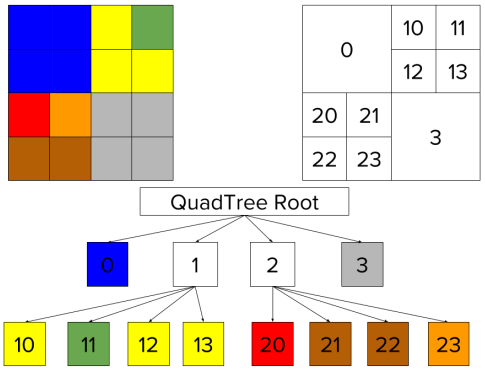

Fig. 11. QuadTree indexes 4X4 squares

For our use case, we can use rectangles and squares to divide and index real city boundaries.

Using QuadTree, the majority of bounded rectangles that do not contain target point could be filtered out. We run geospatial functions (e.g., *st_contains*) only for rectangles that contain target point.

E. Presto Geospatial Plugin

Using the Presto plugin framework, we implemented a Presto Geospatial plugin [18], which possesses a variety of

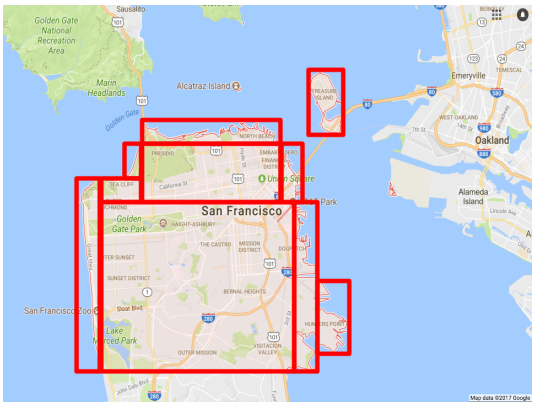

Fig. 12. Using QuadTree to index San Francisco and glean valuable data for our Presto analytics

geo-functions. One of them is a Presto aggregation function, *build_geo_index*, which serializes/deserializes geospatial polygons into a QuadTree. During query execution, we build a QuadTree on the fly. QuadTree is used to filter out geofences that do not contain target point. During this phase, majority of geofences would be filtered out. Finally, we run *st_contains* for remaining geofences. To improve usability, we added Presto query optimizations to automatically rewrite user queries into optimized ones, as portrayed in figure 13.

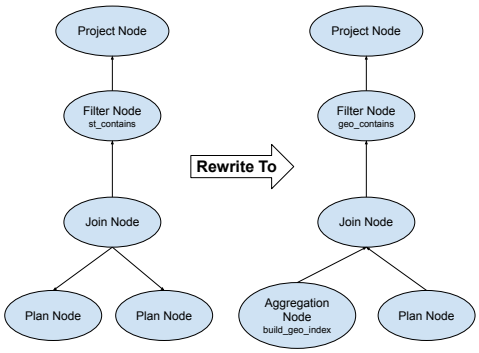

Fig. 13. Presto optimizes a query using QuadTree

The Presto GeoSpatial plugin is running in production at Uber. Among our GeoSpatial traffic, more than 90% is completed within five minutes. Compared with the brute force Hive MapReduce execution, our Presto Geospatial Plugin is more than 50X faster, leading to greater efficiency.

Our users are happy with the five minutes query latency, therefore, we did not optimize geospatial query performance further. Actually, we could build geo-indices offline, and during runtime, let Presto make use of the pre-built indices.

VII. Cache

In production experience, we found the single Hadoop Distributed File System (HDFS) NameNode *listFiles* performance degradation, could hurt Presto performance badly. To address the performance challenge, we initiated two efforts, one is to roll out HDFS Observer NameNode [31] in production. The other is to do caching inside Presto [16]:

A. File list cache

Presto coordinator caches file lists in memory to avoid long *listFile* calls to remote storage, e.g. Hadoop Distributed File System (HDFS). This can only be applied to sealed directories. For open partitions, Presto will skip caching those directories to guarantee data freshness. One major use case for open partitions is to support the need of near-real time ingestion and serving. In such cases, the ingestion engines (e.g., micro batch) will keep writing new files to the open partitions so that Presto can read near-real time data. With file list cache enabled for 5 of our most popular tables, our production traffic shows overall *listFile* calls is reduced to less than 40%.

B. File handle and footer cache

Presto worker caches the file descriptors in memory to avoid long *getFileInfo* calls to remote storage. Also, a worker caches common columnar files and stripe footers in memory. The current supported file formats are ORC and Parquet. The reason to cache such information in memory is due to the high hit rate of footers as they are the indexes to the data itself. With file handle and footer cache, our production traffic shows almost 90% of *getFileInfo* calls could be reduced.

A number of cache techniques are developed for Presto, including Metastore versioned cache, fragment result cache, Alluxio [24] data cache, and affinity scheduler, details in [16].

VIII. Cluster Federation

Presto coordinator is stateful. It is doing query planning, query optimization, task status tracking, worker status tracking, for all queries and workers. There is one single coordinator in one Presto cluster. Due to the limitation of CPU and memory, Presto coordinator could become the bottleneck in large production deployments. For example, at Uber, Twitter, Facebook, and Pinterest, we see performance degradation with Presto coordinator when a single cluster becomes bigger than 1000 machines, or there are more than 500 complex queries running concurrently.

To resolve the scalability problem, we design and implement a cluster federation solution. Using HTTP Redirect, we developed a presto gateway. The gateway will redirect incoming queries to specific presto clusters, based on user name and group information. The user and group to cluster mapping data is stored in MySQL. Presto administrators could play with MySQL to dynamically redirect any traffic to any cluster.

This is very useful, as big companies always have more than one Presto cluster. There are dedicated Presto clusters for latency sensitive queries. A few big clusters are shared by all teams. At Facebook, Uber and Twitter, we are managing more than 5 production clusters. When we are doing cluster maintenance or software upgrade, we will redirect traffic either to shared cluster, or newly launched new cluster, to guarantee no downtime for end users.

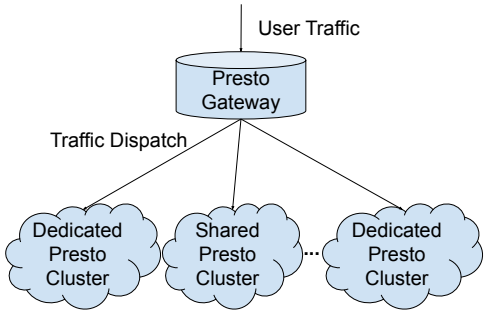

Fig. 14. Presto gateway will dispatch user traffic to appropriate cluster

IX. Presto on Cloud

Another popular use case is running Presto on cloud. We could store data in Amazon S3 or Google GCS, and launch Presto to query it. Presto supports Amazon S3 and Google GCS natively. We developed the *PrestoS3FileSystem* [19], which provides a *FileSystem* api on top of Amazon S3.

Amazon S3 is an object storage system. To support general *FileSystem* api and run it efficiently for Presto, we did a number of optimizations: (1) *Lazy seek*: which saves unnecessary seeks in Amazon S3, and seek to destination until needed for input stream or output stream; (2) *Exponential backoff*: which backoff request exponentially when Amazon S3 is not responding; (3) *Leverage Amazon S3 select*: Amazon S3 select has better performance, we pushdown projections directly to Amazon S3 to get optimal performance. (4) *Multi-part upload*: When loading a big object to Amazon S3, we break it up into multiple parts, and upload to S3 in parallel. This improves uploading throughput and recovery time.

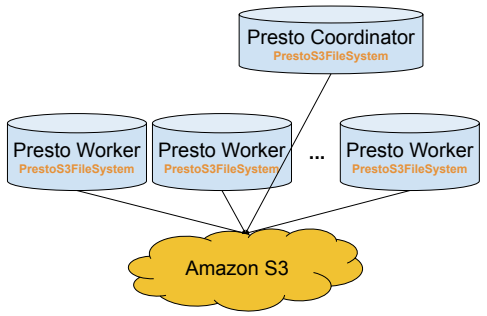

Fig. 15. Presto could expand and shrink gracefully on AWS, querying data from Amazon S3

Presto has graceful expansion and shrink, leveraging cloud elasticity. During busy hours, to expand on Amazon or GCP, we could simply add more workers, configured with the same coordinator. New workers are automatically added to the existing cluster. During non-busy hours, to gracefully shrink workers from existing clusters, administrators could send a command to presto workers. Upon receiving the command, presto worker will enter *SHUTTING_DOWN* state: sleep for *shutdown.grace-period*, which defaults to 2 minutes. After this, the coordinator is aware of the shutdown and stops sending tasks to the worker. The worker will block until all active tasks are complete. The worker will sleep for the grace period again in order to ensure the coordinator sees all tasks are complete. Finally, the presto worker will shut down.

X. Experimental Results

A number of experiments are conducted to show the efficiency and performance of our Presto clusters. Most of the experiments are from our production Presto traffic at Uber, Twitter, and Pinterest.

A. Presto Druid Connector

To demonstrate the efficiency of Presto connectors with predicate pushdown, limit pushdown, and aggregation pushdown, we set up a 100 node druid cluster, and load 100TB production data into it. The Druid cluster is running in Twitter's data center. Each machine has 32 core CPU and 128GB Memory. Hadoop Distributed File System(HDFS) is configured as Druid deep storage. Our Presto cluster has 100 nodes, each has 32 core CPU and 128Gb memory.

20 druid production queries are used in the experiment. 14 of them have predicates, 5 of them have limits, and 12 of them are aggregation queries. We compare the 20 queries performance running them on Druid and on *Presto-Druid connector*.

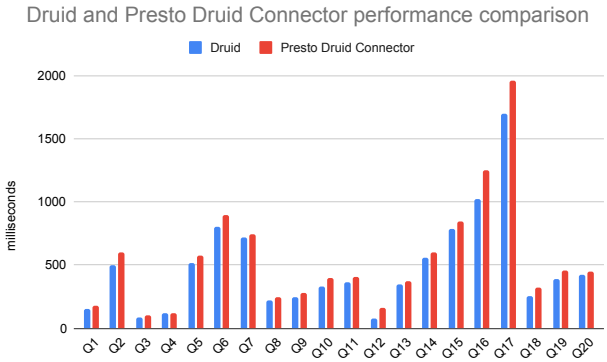

Fig. 16. With predicate pushdown, limit pushdown, and aggregation pushdown, presto druid connector could achieve real time query latency, with less than 15% overhead, compared with druid performance

As shown in figure 16, with predicate pushdown, limit pushdown, and aggregation pushdown, *Presto-Druid connector* adds less than 15% overhead, compared with Druid query latency. Most of the queries complete within 1 second in the *Presto-Druid connector*. This demonstrates that, with pushdown techniques, presto connectors could achieve real time query latency. We could use presto for monitoring, decision making, and real time reports.

B. Parquet Reader Improvements

To demonstrate the efficiency gain of Presto new Parqu reader, we run the second experiment on a 200 node Pres cluster. Each machine has 32 core CPU and 256GB Memo

We take 21 of Uber production Presto queries, 4 of the are table scans, where 2 of them are needle in a haystack ty table scan. 5 of them are group by queries, and another 12 them are joins.

The experiment is running on Uber production data, whi are stored on the Hadoop Distributed File System, in Parqu format.

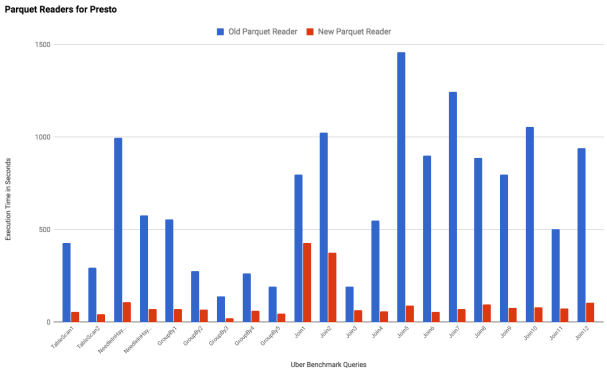

Fig. 17. Brand new Parquet reader demonstrated 2-10X speedup for Uber's benchmark SQL queries

As shown in figure 17, we could see with nested columr pruning, columnar reads, predicate pushdown, dictionary push down, lazy reads and vectorized reader, our new Parquet reade consistently achieves 2X – 10X speedup. For needle in a haystack type table scans, the new Parquet reader is pretty efficient. Actually, when we turned on using the new Parquet reader by default at Uber, Presto traffic P90 query latency reduced from 5 minutes to 40 seconds.

C. Parquet Writer Improvements

To show the efficiency of the Presto native Parquet writer we run the experiment on a 100 node Presto cluster on AWS. The machine is r5.8xlarge, each has 32 CPU and 256GB Memory.

We run the experiments by using Presto writing a list of pages with millions of rows. The following figures show various types of data throughput with Snappy compression, Gzip compression, and no compression.

In figure 18, figure 19, and figure 20, we could see that compared with the old writer, our native Parquet writer could consistently achieve more than 20% throughput. For bigint type with Gzip compression, our native parquet writer performs best, with more than 650% throughput improvements. When writing all columns of TPCH LINEITEM, the throughput gain is around 50%.

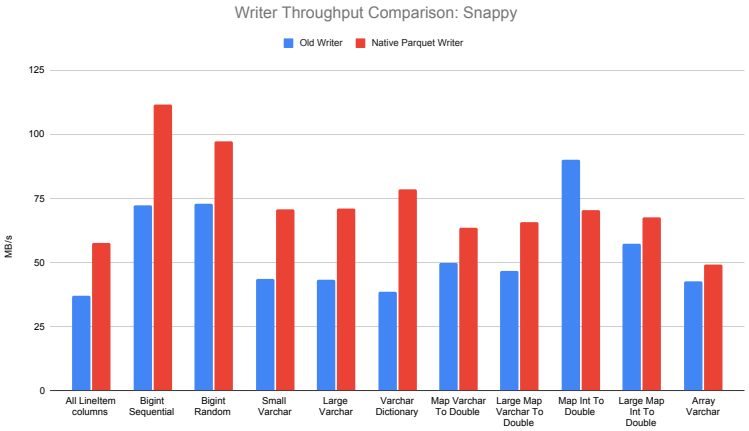

Fig. 18. Native parquet writer consistently improves throughput by 20% for snappy compressed files

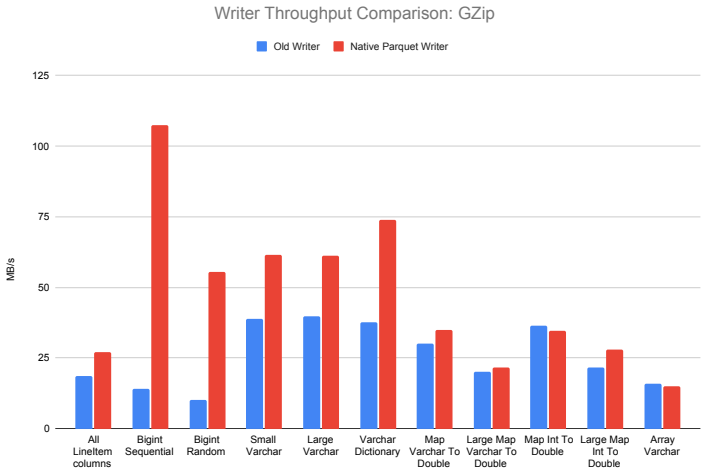

Fig. 19. Native parquet writer consistently improves throughput by 20% for Gzip compressed files

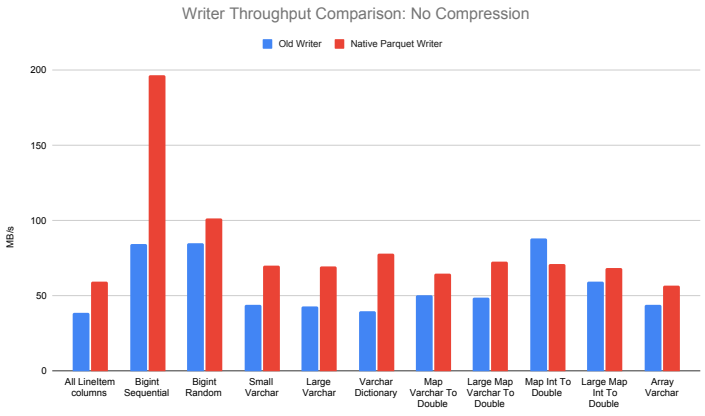

Fig. 20. Native parquet writer consistently improves throughput by 20% for no compressed files

XI. Related Work

Systems that run SQL against large data sets have become popular over the past decade, especially for SQL engines on cloud. A comprehensive examination of the space is outside the scope of this paper. We focus on some of the notable work in the area.

A number of SQL engines are provided on cloud, including Google BigQuery [33], Amazon Redshift [34], Snowflake [35]. Most of them could read external data to varying degrees, however, they are built around an internal data store and achieve peak performance when operating on data loaded into the system. In contrast, Presto is data source agnostic. More than 20 connectors are supported in Presto, e.g. *Presto-Druid connector*, *Presto-Kafka connector*, *Presto-Delta Lake connector*, *Presto-MySQL connector*, *Presto-MongoDB connector*, and *Presto-Cassandra connector*. Users could leverage Presto connectors to run SQL analytics on heterogeneous storage systems without data copy.

Druid [10], Pinot [7], and Clickhouse [32] could achieve sub-second query latency with customized storage techniques, e.g. pre-aggregation, inverted index, columnar storage. They have advantages in query performance, however, sacrifice SQL functionlities, e.g. distributed hash join, sub queries. The Presto community has built *Presto-Druid connector*, *Presto-Pinot connector*, *Presto-Clickhouse connector*, with pushdowns. Users could leverage Presto connectors to query data across Druid, Pinot, and Clickhouse, with full SQL functionalities, and little performance overhead.

Apache Hive [23] provides a SQL-like interface over data stored in Hadoop Distributed FileSystem [9], and executes queries by compiling them into MapReduce [36] jobs. SparkSQL [22] is a modern system built on the popular Spark engine [37], which addresses many of the limitations of MapReduce. SparkSQL can run large queries over multiple distributed data stores, and can operate on intermediate results in memory. However, these systems do not support end-to-end pipelining, and usually persist data to a filesystem during inter-stage shuffles. Although this improves fault tolerance, the additional latency causes such systems to be a poor fit for interactive or low-latency use cases.

XII. Experiences

A. Collecting statistics is hard

Presto has both rule based optimizer and cost based optimizer. To enable cost based optimizer running effectively, we need to make sure all statistics are up-to-date, including table statistics, partition statistics, and column statistics. We indeed would love to leverage a cost based optimizer for join re-ordering and join algorithm selection. While, in production environments, we found keeping statistics up-to-date is pretty challenging. There are pipelines ingesting data all the time. It is prohibitively computationally expensive to update statistics upon every data ingestion. It is also not affordable to run *analyze table* before every query execution. We tried adding statistics collection only before joins, still, users queries are coming arbitrarily, either we might add unnecessary overhead to small joins, or data are updated so frequently that we could not update statistics in time.

A practical way is using a rule based optimizer, ignoring statistics, and setting Presto session properties to enable specific join algorithms or manually rewrite users queries for join re-ordering. Presto has session properties to turn on broadcast join for all queries in this session. In production cases, we configure distributed hash join as default to support larger joins. For a number of known use cases, where the smaller table size could be broadcasted, we will set Presto session property to turn on broadcast join for these queries, to have better performance. More research is being conducted to see if machine learning models could help to automatically decide join type, with limited and potentially corrupted statistics.

B. A general gateway is hard

We were building a general gateway for all query systems, including Presto, Hive, and Spark. Our original idea was to estimate query cost in the gateway, and do dynamic dispatch based on clusters' status. We also added security checks and admission control in the gateway. Surprisingly, the gateway is a failure. When traffic grew, the gateway could not scale. Sharding could help to some extent, while we still need to deal with load balance and fault tolerance. Many times, our Presto and Spark clusters are in a good state, but the gateway is not reliable.

The problem is resolved by letting users connect directly with each Presto cluster. Actually, building a scalable service supporting millions of queries per day is challenging. Would be nice if we could re-use existing scalable services, instead of building from scratch. Our Presto gateway implementation is based on the Presto coordinator. In addition, a general gateway could not resolve security problems. We need to implement security and admission control inside Presto.

C. Automatic query translation is important

Presto has limitations for big joins. When users are joining two large tables, Presto will return an error, with message *"Insufficient Resource ..."*. Users are not happy with this error. Even after our explanation, users have to spend time translating Presto queries into SparkSQL [22] or Hive [23]. Whenever we do user surveys, the *"Insufficient Resource ..."* error and query translation is always on the top of users' complaints.

We need to resolve the problem either via: adding fault tolerance to Presto, or automatically translate failed Presto queries to other systems. Presto on Spark is a good option, which enables users writing the same Presto SQL, with automatic translation it into SparkSQL.

D. SQL engine is only one part of data systems

In our customer survey, we found users are complaining about data schema not flexible to be updated and performance degradation. After investigation, we found performance degradation is due to the single Hadoop Distributed File System (HDFS) NameNode *listFiles* stuck, which hurts Presto

performance badly. Schema management is another service outside of Presto. Actually, from users' point of view, they would like a smooth SQL experience, including a flexible schema management service, a performant SQL engine, a reliable storage system, an easy to play-with message queue pipeline, etc. Users only care about end to end experience. A performant SQL engine is only one part of a big data system. All parts of big data systems need to collaborate together to provide a nice end to end user experience.

ACKNOWLEDGMENT

We'd like to thank the Presto community for several rounds of design and implementation improvements. They also keep a high quality code review. Thanks goes to: Nezih Yigitbasi, Wenlei Xie, Andrii Rosa, Rebecca Schlussel, Rongrong Zhong, Bin Fan, Shixuan Fan, Jiexi Lin, Leiqing Cai, Tim Meehan, Ying Su, James Petty, and many others, who support Presto at numerous companies in production, answering user questions, and contribute patches and ideas to Presto.

REFERENCES

[1] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL oneverything." in 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, 2019, pp. 1802–1813

[2] Finkel, R.A., Bentley, J.L. "Quad trees a data structure for retrieval on composite keys." Acta Informatica 4, 1–9 (1974). https://doi.org/10.1007/BF00288933

[3] https://aws.amazon.com/s3/

[4] https://blog.twitter.com/engineering/en_us/a/2013/announcing-parquet-10-columnar-storage-for-hadoop

[5] https://orc.apache.org/

[6] Eric Bruneton "ASM 4.0: A Java bytecode engineering library" https://asm.ow2.io/asm4-guide.pdf

[7] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. 2018. "Pinot: Realtime olap for 530 million users." In Proceedings of the 2018 International Conference on Management of Data. 583–594

[8] Clinton Gormley, Zachary Tong "Elasticsearch: The Definitive Guide" O'Reilly Media, Inc. 2015, ISBN:978-1-4493-5854-9

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System" Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium, page 1 -10

[10] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, Deep Ganguli "Druid: a real-time analytical data store" In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. June 2014 Pages 157–168

[11] https://cloud.google.com/storage/

[12] https://www.mysql.com/

[13] https://prestodb.io/blog/2019/12/23/improve-presto-planner

[14] https://eng.uber.com/presto/

[15] https://prestodb.io/blog/2021/06/29/native-parquet-writer-for-presto

[16] https://prestodb.io/blog/2021/02/04/raptorx

[17] https://medium.com/pinterest-engineering/presto-at-pinterest-a8bda7515e52

[18] https://www.oreilly.com/radar/query-the-planet-geospatial-big-data-analytics-at-uber/

[19] https://netflixtechblog.com/using-presto-in-our-big-data-platform-on-aws-938035909fd4?gi=2c425020ac56

[20] https://hudi.apache.org/

[21] https://iceberg.apache.org/

[22] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia "Spark SQL: Relational Data Processing in Spark" In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. May 2015 Pages 1383–1394

[23] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Antony, Hao Liu, Pete Wyckoff "Hive – A Warehousing Solution Over a Map-Reduce Framework" International Conference on Very Large Data Bases (VLDB) VLDB '09, August 24-28, 2009, Lyon, France

[24] Haoyuan Li "Alluxio: A Virtual Distributed File System" https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.pdf

[25] Chunxu Tang, Beinan Wang, Huijun Wu, Zhenzhao Wang, Yao Li, Vrushali Channapattan, Zhenxiao Luo, Ruchin Kabra, Mainak Ghosh, Nikhil Kantibhai Navadiya, Prachi Mishra "Hybrid-Cloud SQL Federation System at Twitter" ECSA (Companion) 2021

[26] Walaa Eldin Moustafa, Wenye Zhang, Sushant Raikar, Raymond Lam, Ron Hu, Shardul Mahadik, Laura Chen, Khai Tran, Chris Chen, and Nagarathnam Muthusamy https://engineering.linkedin.com/blog/2020/coral

[27] https://www.informationweek.com/big-data-analytics/airbnb-boosts-presto-sql-query-engine-for-hadoop

[28] https://github.com/prestodb/presto

[29] https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry

[30] https://github.com/Esri/geometry-api-java

[31] https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ObserverNameNode.html

[32] https://clickhouse.com

[33] https://cloud.google.com/files/BigQueryTechnicalWP.pdf

[34] https://aws.amazon.com/redshift

[35] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner "The Snowflake Elastic Data Warehouse" In Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data. June 2016 Pages 215-226

[36] Jeffrey Dean, Sanjay Ghemawat "MapReduce: simplified data processing on large clusters" In Proceedings of the 2008 Communications of the ACM. January 2008 Pages 107–113

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" In Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation. April 2012 Pages 15-28