Presto's History-based Query Optimizer

Pranjal Shankhdhar
Meta Platforms
Menlo Park, CA, USA
pranjalssh@meta.com

Feilong Liu
Meta Platforms
Menlo Park, CA, USA
feilongliu@meta.com

Jay Narale
Uber Technologies
San Francisco, CA, USA
jay.narale@uber.com

James Sun
Meta Platforms
Menlo Park, CA, USA
jamessun@meta.com

Rebecca Schlussel
Meta Platforms
Menlo Park, CA, USA
rschlussel@meta.com

Lyublena Antova
Meta Platforms
Menlo Park, CA, USA
lyublena@meta.com

ABSTRACT

An important feature of modern query optimizers is the ability to produce a query plan that is optimal for the underlying data set. This requires the ability to estimate cardinalities and computational costs of intermediate query plan nodes, which is highly dependent on both the query shape and the underlying data distribution. Traditional methods include collecting statistics on base tables and implementing cardinality and computational cost derivation inside the optimizer, which is error-prone for complex query shapes. This paper presents Presto's novel history-based optimization framework (HBO), which collects execution histories and uses them to optimize similar queries in the future. The framework produces accurate estimates for complex query shapes in a lightweight, automated manner, and adapts automatically to changes in underlying data distributions. We present the design and implementation of the HBO framework and provide details on its use in various optimization rules, as well as details on implementing the statistics store on top of a Redis key-value store. We also present the results of running HBO in production in two large data infrastructure organizations (Meta and Uber).

PVLDB Reference Format:
Pranjal Shankhdhar, Feilong Liu, Jay Narale, James Sun, Rebecca Schlussel, and Lyublena Antova. Presto's History-based Query Optimizer. PVLDB, 17(12): 4077 - 4089, 2024.
doi:10.14778/3685800.3685828

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

1 INTRODUCTION

An important component of every query engine is its query optimizer. This is the part of the system responsible for taking the input query tree (typically an abstract query tree produced by the parser/analyzer) and converting it into an efficient execution plan. As the complexity of queries grows, so does the search space of possible plans, and having a good query optimizer becomes critical for navigating that search space and producing an efficient execution plan. Today, most enterprise-grade query optimizers are cost-based [7, 11, 15, 26, 28, 30, 31, 37], meaning they use a costing function to predict how computationally expensive a query plan is and select the one with the lowest cost estimate for execution. The costing module typically uses knowledge of data statistics and computation cost to compare different query plans and guide the optimizer into selecting the best query plan. This module often relies heavily on estimated data distribution and cardinalities.

```
select * from
    (select * from R
    where type='X' and date='...') R
join
    (select * from S
    where status='Y' and date='...') S
on (R.id=S.id)
```

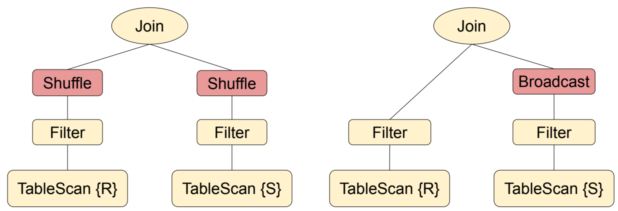
Figure 1: Example SQL join query

To demonstrate the above, consider the query snippet in Figure 1, which joins two tables after applying some filters.

In a distributed query engine there may be multiple alternative query plans that evaluate the above query, where some are more computationally expensive than others. Depending on the cardinalities of the two sides, a different join order may work better. Moreover, due to data being distributed, the optimizer is also responsible for selecting a distribution strategy that collocates data to be joined on the same nodes. Figure 2 shows two alternative plans for the query of Figure 1. If one of the sides is small, the query optimizer may choose to broadcast that side to all worker nodes and leave the large join side as is (Figure 2b), whereas in other cases, a better query plan will repartition (shuffle) the two sides on the corresponding join attribute (see Figure 2a).

The most important component when costing query plans is cardinality estimation: research has found a strong correlation between the quality of cardinality estimators and query performance [22]. Traditional cost-based optimizer typically rely on an offline process to collects statistics about input data such as number of rows, number of distinct values and histograms summarizing data distribution. This is then paired with compile-time estimation of cardinalities of intermediate nodes in the query plan, which takes into account things like selectivity of filter and join predicates, and number of

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685828



(a) Partitioned join **(b) Broadcast join**

Figure 2: Alternative plans for the SQL query in Example 1

distinct keys for predicting cardinalities of various operator nodes. However, this approach has several limitations and disadvantages. First, it requires data to be analyzed before it can be queried. In addition, cardinality estimators makes a number of simplifying assumptions such as data uniformity, independence of filters and columns, etc. They are often incapable of estimating selectivity of complex expressions, such as conditional expressions, function calls, and multi-key aggregations. There have been attempts to store more complex statistics such as multi-column and join histograms, but those require additional time and space to compute, and are often non-trivial to work with. As a result, it is not surprising that even industry-strength cardinality estimators routinely produce large errors in estimation [22]. Beyond the traditional cost estimation approaches, the past several years have seen an explosion in learning-based approaches to cardinality and cost estimation (see Section 7 for a review). While these are promising and overcome many of the simplifying assumptions in traditional methods, they require an even bigger upfront effort in training and refining the models. Learning-based approaches also present a challenge to operations, as they may provide less robustness in situations that call for predictable performance. They are also hard to debug when something goes wrong: explicability and provenance in learned models is still an active area of research.

To overcome the challenges presented above, in this paper we present Presto's history-based query optimizer (HBO) that has been used in production for several years at several large data infrastructure groups including those of Meta and Uber. In a nutshell, HBO tracks query execution statistics at the operator node, and uses those to predict future performance for similar queries. This is possible due to the observation that while complex, queries are repetitive in nature: they are often generated by a template and follow the same structure. This is true for many large scale offline pipelines (such as daily or hourly ETL and data analytics jobs), as well as in more interactive use cases such as dashboards, where again queries are often generated by a template and follow the same structure.

HBO solves many of the problems that previous approaches failed at, namely:

- **Accuracy**: Statistics are recorded during actual execution runs and tracked at the operator level, thus eliminating large estimation errors introduced from deriving cardinalities of complex expressions using only base table statistics.
- **Automation**: Histories are tracked by a light-weight process with every query run, thus avoiding the need for sampling overhead or model training for priming the statistics estimators.
- **Adaptiveness**: Changes to the underlying data distribution are automatically reflected in tracked histories and used in future optimizations.
- **Explicability**: users and DBAs can view where the estimated data came from and how far it is from the actual value.

The rest of the paper is organized as follows. Section 2 gives background on Presto - the underlying open-source database and its existing query optimizer. Section 3 presents the architecture of HBO, including how and when statistics are tracked, and their usage in various optimizations. In Section 4 we describe the APIs for communicating with the stats store, and give details on implementing the stats store in the Redis [4] open-source key-value store. Section 5 describes the tools we built to make HBO easy to operate. Section 6 presents extensive experimental evaluation of using HBO in production at two large data infrastructure organizations, namely those of Meta and Uber. Finally, we review related work in Section 7 and conclude in Section 8.

2 BACKGROUND

Below we give background on Presto and its existing query optimization framework.

2.1 Presto query engine

Presto [29, 33] is a distributed query engine used for low-latency interactive use cases as well as long-running ETL jobs at Meta. It was originally launched at Meta (called Facebook at that time) in 2013, and donated to the Linux Foundation in 2019. Presto supports ANSI-SQL and has an extensible framework of plugins that allows reading data from various sources such as Hive, Iceberg, etc. Presto was designed as a shared-everything system: all queries share resources in a cluster without hard isolation. It was designed for latency over scalability, so the query optimizer helps support this goal.

Since the focus of this paper is the Presto query optimizer, in the following we will focus on this component and will refer the reader to [29, 33] for more details on the overall architecture of Presto and components beyond the query optimizer such as scheduler and coordinator.

2.2 Presto's Query Optimizer

Historically, Presto's query optimizer was designed as a rule-based engine, which implemented multiple standard optimization techniques such as join reordering, projection and filter-pushdowns, and more recently ML-workload specific optimizations that target operations on semi-structured data in the form of maps and arrays. Presto's optimization engine executes rules in a sequential way, as depicted in Figure 3. It starts with the input AST produced by the parser/analyzer component and as output produces the final execution plan passed to the scheduler and execution component. As depicted in this Figure, each optimization rule takes the input plan and transforms it into a new plan. Some optimization rules are invoked for specific patterns in the query tree (and apply more

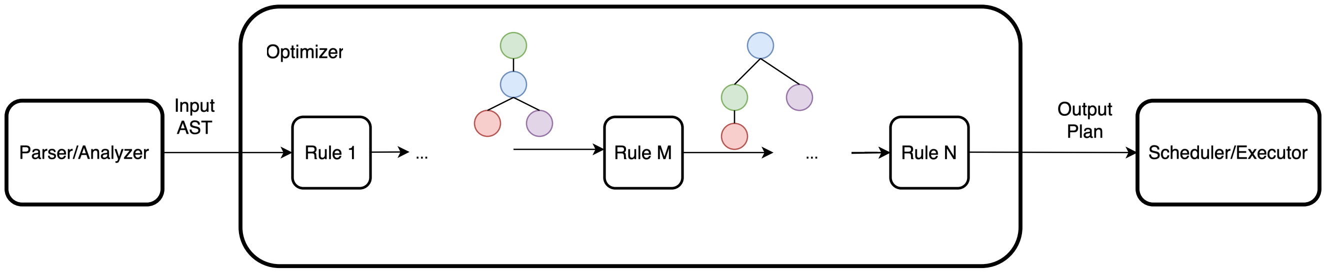


Figure 3: Presto's rule-based query optimizer

localized transformations), whereas others implement a visitor API to walk down and transform the plan tree. Presto also supports a connector optimization framework [1], that allows different connectors to implement optimizations such as selection pushdown in the storage layer. Initially, all optimization rules were heuristic, meaning they performed a transformation without knowledge of the underlying data distribution or compute cost. Thus, many of the query optimization rules came with a configuration parameter, which allowed users to enable or disable optimizations as needed. At the time of writing, Presto implements nearly 150 optimization rules, and about a third of them can be enabled or disabled with a configuration parameter [3].

2.3 Stats and cost estimation in Presto

Around 2016 Presto's optimizer was augmented with basic costing capabilities to aid some of the major optimizations such as join reordering and picking join distribution. The main component of costing is the cardinality estimator, which in turn relies on base table statistics to derive cardinality estimates for intermediate plan nodes. Presto stores statistics at the partition level[1] and include the following:

- overall cardinality of the partition
- column statitstics including
  - Average size
  - Number of distinct values
  - Number of null values
  - Range (min/max) for the values

At the moment, Presto does not support additional well-established data distribution sketches such as histograms [20]. There are ongoing efforts to add support for these from our open-source partners, but at the time of writing these were not part of the main Presto distribution.

Using the base table statistics above, Presto implements a stats calculator that produces cardinality estimates for intermediate nodes in the query. The stats calculator is implemented as a visitor on top of the plan tree, and propagates the base table stats throughout the tree by making a few simplifying assumptions such as uniform distribution of column values, and independence of columns. As an example, to estimate the cardinality of range predicate `x BETWEEN (m, n)`, the stats intersects the range defined by the predicate with the range $[M, N]$ defined in the base column stats to produce $n - m/(N - M)$ as the estimate, and the cardinality of a multi-key group by is estimated as the product of the distinct values for each of the columns.

3 ARCHITECTURE

Presto's cost based optimizer produces useful estimations for simple queries, but the error margin increases exponentially as queries and the underlying expressions get more complex. For example, the cost-based framework is able to decently estimate statistics for expressions with single column constraints - but gets perplexed even if we add a conjunction to it. In this case, it is impossible to get accurate statistics without finding correlation between different columns used in the expression. It is very common in our data warehouse for queries to have complex expressions and user defined functions for business logic. In these cases, traditional cost-based estimation techniques prove ineffective.

Presto's History Based Optimizer (HBO) leverages the fact that most of the data warehouse queries are programmatically generated - usually arising from data pipelines and user dashboards. These queries often share a common structure, differing only by symbols and constants use. These symbols usually represent a different date range predicate to select newly arrived data, or selecting different parameter values in a dashboard. HBO leverages the precise runtime statistics of similar queries that have executed in the past to predict statistics of future execution. The resulting statistics are more accurate and also more verbose than CBO[2]. In this section we dive into the architecture of HBO and discuss how HBO tracks similar queries and predicts statistics. Then, in Section 3.7 we give details on how statistics computed by the HBO framework are used during query optimization to estimate the compute cost of query plans and make other performance related decisions.

3.1 Desiderata

When designing the history-based optimization framework we took several requirements into account:

- **Estimates need to be accurate**: As discussed above, traditional cardinality estimation frameworks fail to produce accurate estimates for complex query shapes.
- **Accommodate changes to both data and queries**: In our use case neither data nor queries are static. However, the changes are often not dramatic: data distribution rarely changes significantly in a short period of time (but may over a longer period). Similarly, queries in a given pipeline or

[1]In our experience most users of Presto choose to partition their data on a temporal column to allow of fast retrieval of data from a specific date/time interval.

[2]Throughout the paper we will refer to the legacy approach to cost estimation as *CBO* to distinguish it from *HBO*, which will denote the new methodology using histories

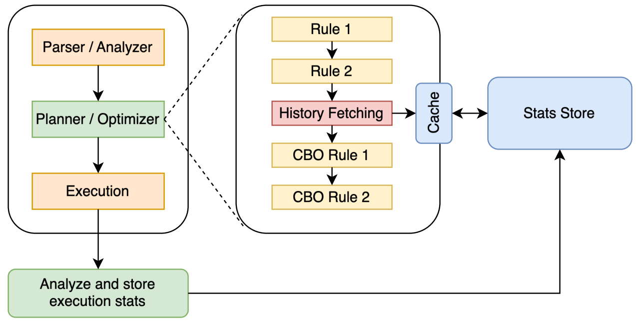


Figure 4: Architecture of HBO

```
def fetchStats(PlanNode) -> Optional[Stats]

# Stats seen by optimizer rules
class Stats:
    source: Enum["HBO", "CBO"]
    cardinality: int
    size : int
    ...
```

Figure 5: API of HBO Framework

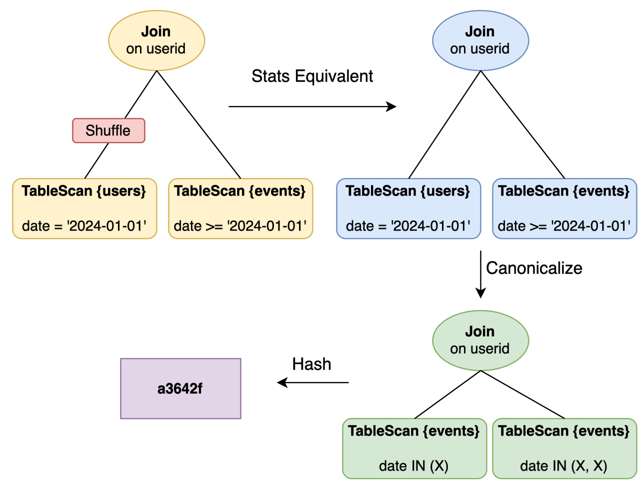


Figure 6: Plan hashing in HBO

dashboard are often generated by code and follow a similar template.

- **Minimal overhead to query processing**: Both traditional and novel learned-based approaches rely on precomputing base table or intermediate statistics using sampling and learning methods, and continuing to do so as data and queries change. We wanted a lightweight mechanism without huge upfront computation.
- **Ease of use and operation**: we want to design a "hands-free" solution that requires no user input to work. At the same time, the solution needs to provide a high-degree of clarity and debugability to be able to answer questions like why a certain query plan was selected.
- **Seamless integration with classic methods for deriving cost**: We want to support a mechanism to fallback to classic cost estimation in case histories are not yet available. Moreover, it should be easy to mix and match the two in a single query to allow for cases where histories are available for a block of the query but not the full query.

In the following sections, we describe our approach, which achieves all of the above requirements.

3.2 Overview

As shown in Figure 3, the query optimizer of Presto applies a sequence of rules to compute the optimal execution plan. In this process, the input plan tree undergoes a series of equivalence-preserving transformations. Some of the transformations (also called cost-based/CBO rules) take cost estimates into account when producing the resulting plan. The main idea behind HBO is to collect run-time statistics for each node in the query plan during query execution, and later during query optimization to search the stats store for previously executed similar query plans.

Figure 5 shows the API available for optimizer rules to get statistics. The API takes a plan node and returns the corresponding statistics if available. Statistics can be derived from either HBO or the traditional CBO framework. Optimizer rules are agnostic to the source of statistics. The Presto optimizer chooses the statistics which are more likely to be accurate - typically preferring HBO statistics. It is not uncommon for CBO to build upon HBO stats for downstream plan nodes in case histories are only present for part of a query. A mix of HBO and CBO statistics boosts coverage and accuracy of stats to a large extent. HBO statistics are more verbose than CBO statistics as well. HBO can store runtime information like shuffle fanout sizes and task parallelism in addition to the classic cardinality and data size statistics. Next, we'll look into the secret sauce for finding similar plan nodes from history.

Finding similar plan nodes is the crux of HBO, and there are several main challenges to doing that:

- Queries change: while queries often follow the same template, parameter values may change across runs
- Query plans change: the same query template may have run with a different (but equivalent) query plan depending on which optimization rules triggered
- Searching for similar query plans has a high overhead: there may be millions of queries executed daily and exhaustive search for previous runs is prohibitively resource intensive.

The solution to the above challenges can be summarized with the following: computing canonical plans and hashing plan nodes. Below, we give details on each of these aspects.

3.3 Computing canonical plans

In a large warehouse, searching through all query plans that were executed previously has an unacceptable overhead. HBO uses a different strategy to transform plan nodes to a canonical form before writing them to the stats store. All similar plan nodes will share the same canonical form, which can be used as a key to read and write stats. We use different strategies to create several canonical forms, and use them in order of their confidence.

There are two main parts to computing the canonical representation of a query: (1) approximate the template that generated the query by ignoring constants used as parameter values, and (2) map equivalent plans into the same canonical query plan.

Canonicalizing Table scans: In our data warehouse tables are usually partitioned on a temporal column, and new data lands into new partitions on a regular basis (for example daily or hourly). As many of our pipelines operate on most recently landed data, the table scan nodes in our query trees typically have a partition filter attached. For example, the table scan plan node in a query like `SELECT * from customers WHERE date = '2024-01-01'` will select data for 'January 1st 2024'. Likely, this query came from a template predicate `WHERE date = $date$`, and subsequent runs of the query will use a different parameter value for the date predicate. To compute the canonical representation of this node, we replace the original predicate with `date IN ('X', 'X')`, mapping the literal value to a symbol X. In doing so we make the simplifying assumption that partitions have similar size, thus treating all partitions as equal. We describe later how to relax this assumption.

Pruning constants: Nodes in a plan tree can contain different constants, leading to different queries from the same template. Ideally, we'd like to scrub most of them and replace them with *'X'* as we do with table scan partitions. We use several strategies based on how conservative we want to be in removing these constants. We store canonical forms arising from all such strategies, and read from the safest canonical form available during stats fetching. Let's take a look at different strategies:

- **Prune constants from equality predicates on partition key:** Prunes constants only from table scan partitions, as seen above.
- **Prune constants from projections:** Prunes all constants from table scan partitions and projection expressions. For example `SELECT *, '2024-01-01'AS date` is canonicalized to `SELECT *, 'X'AS date`. Removing constants like these don't usually lead to varying statistics.
- **Prune constants from equality predicates:** Prunes constants from table scan partitions, select expressions and specific constants from filters. Here we prune constants in equality predicates, for example `WHERE id = 128` is canonicalized to `WHERE id = 'X'`. This strategy is less conservative than the above ones as it assumes a uniform distribution for all *id* values, and may produce incorrect stats if that is not the case and historical stats got recorded for values with different distribution from the current ones.
- **Prune constants from range predicates:** In this strategy we also prune constants used in range predicate filters. Note that this may lead to significant mis-estimations, as for example the predicates `WHERE count >= 1` and `WHERE count >= 1000` can lead to vastly different statistics. Similarly it is dangerous to prune constants in parameters of user defined functions as well. We do not employ this canonicalization strategy as of now but believe it can be used in combination with classic cardinality estimation, where the use of histograms can inform us if the relative distributions of the two predicates are the same and guard the use of historical stats only in that case. This is subject of future work.

Canonicalizing plan nodes: Here we attempt to compute canonical representation of otherwise equivalent query plans. We then proceed to serialize the canonical plan node into a string using the pseudo code below. In essence, the representation of a plan node includes all its descendants along with the node itself:

```
def str(planNode):
    return str(planNode.information) + [str(c) for c in
        planNode.children]
```

The individual canonicalization steps are as follows:

- Coalesce all inner join nodes into one. It is defined as:

```
str(A join B join C) = str(join, [str(A),
    str(B), str(C)])
```

- Sort serialized child plan nodes when their order doesn't matter - in case of unions and inner joins. Combined with above coalescing, this decouples the plan node representation from join order given by user and join decision taken by optimizer. More formally:

```
str(A join B join C) = str(join,
    sorted([str(A), str(B), str(C)]))
```

- All right joins are rewritten as left joins when canonicalizing.
- Canonicalize expressions in projections and filters so expressions like `a > b` and `b < a` end up the same. This can be done recursively as expressions have a tree like structure as well.
- Presto optimizer may introduce temporary variables as the query becomes complex. We inline their values, as they are populated from columns, constants or other temporary variables. This removes variable names from the representation.
- Limit expressions may lead to eager termination of the query, causing incomplete stats in upstream plan nodes. For cases like these, a downstream limit node representation is attached with such plan node. This implies that stats of such plan nodes will only be used when a similar downstream Limit node is present as well. Presto has several operators which do eager termination, and this is enabled for all such cases.

Hashing: A plan node may have hundreds of descendants - and the string representation can be quite big. However, we are not concerned with the contents of this representation. We can simply use a hashing algorithm like *SHA256* to hash the serialized plan node. Stats store will be a key value store with key being the plan node hash, and value being the stats. This can save a lot of storage and network overhead. We will look into details of stats store in Section 4.

3.4 Ever-changing query plans

Presto optimizer continuously transforms plans by applying a sequence of rules. Ever-changing plan nodes makes it harder to search for similar plan nodes from the past - even the canonicalized hash may keep changing as rules run. Moreover we can only know stats for the final plan nodes which are executed - intermediate plan

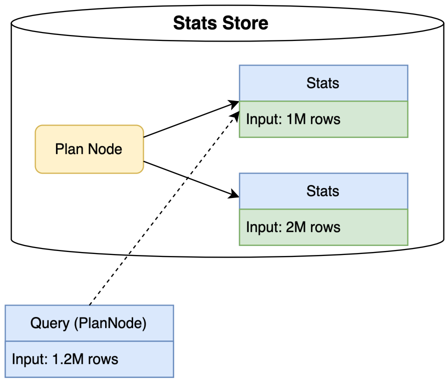


Figure 7: Storing stats for different data

nodes that exist in the optimizer are lost during planning. This makes searching for plan nodes in the stats store very tricky.

HBO overcomes this by introducing an intermediate rule in the optimizer. This rule ("History Fetching" in Figure 4) runs at a later stage in the optimizer just before cost based estimation is required. It goes over the current plan tree and finds relevant plan node histories for all the nodes from the stats store. We keep a copy of the current version of plan nodes and store them alongside. We call these nodes as "Stats-Equivalent" plan nodes. When the query is finished, runtime stats are mapped with the Stats-Equivalent version of plan nodes and written in the stats store.

Plan transformations by optimizer rules can change plan nodes, but they keep the same Stats-Equivalent Plan Node. This mapping is internally maintained by the HBO framework, and rules don't have to explicitly handle it. Whenever a part of plan is modified, its topmost modified node and all unmodified nodes will preserve the Stats-Equivalent mapping. Naturally, rules are correctness preserving - so they will preserve output for the topmost node they modify. This makes the plan hashing robust to small plan modifications made by the optimizer. For example, physical shuffle node is absent in the Stats Equivalent plan in Figure 6.

If needed, we can apply "History Fetching" rule several times in the optimizer. This will fetch stats for newly created plan nodes as well. In Presto, we only use 2 instances of this rule amongst our collection of 150+ optimizer rules. This rule also serves as a checkpoint to batch all calls to the stats store, thus saving overhead from any serial network calls.

3.5 Storing and fetching stats

We have defined procedures to create canonical representations for all plan nodes in the plan. During history fetching phase, we fetch the mapped stats from the stats store for these nodes. As the query runs, Presto workers periodically send aggregated operator (runtime version of plan node) statistics to Presto coordinator. Presto coordinator keeps aggregating stats until the query finishes, and is able to write them into the stats store at the end. Storing stats at granularity of plan nodes allows us to share them across different queries as well. Queries with common subqueries can benefit from this and share some common stats.

3.6 Adapting to change in data

In the above section, we canonicalized table scan plan nodes by treating all partitions as the same. For example, selecting any 2 partitions from a table will lead to the same canonical form. However, it is not uncommon to have partitions with varying sizes. Say, a user chooses to partition their table by column *country*. Due to different demography around the globe, partition sizes will vary significantly.

To account for this, we store several stats mapped to a plan node. With each version of statistics, we associate the total size of the partitions that it read from. If the plan node (including its children) ended up reading from several tables, we store this information for all such tables. Figure 7 shows a simplified case where we store 2 version of stats for the same plan node. These correspond to two previous executions, where the input table contained 1M rows in first case, and 2M in another. When looking up stats for a plan node, we find an entry closest to the current partition sizes that the query is trying to read. In the example, stats for 1M input rows are closest to the query which has 1.2M rows. It is also possible to use statistical models like linear regression to predict statistics for new partition sizes - however we have not found a need for it so far.

In practice, we store both row counts and byte sizes of partitions. A full representation of what we store is described in Section 4.1. If the available statistics have partition sizes that are too far from the partition sizes in our query, we don't use HBO stats. For every plan node, we store stats for several runs with different input statistics. We invalidate older runs when new ones come in, and try to only store runs with varying input statistics. We put a limit on number of different statistics that can be stored. This limit needs to be big enough to capture varying input patterns. We use a gracious limit of 50 in our deployment, while still maintaining a reasonable bound on storage overhead. Most of the plan nodes end up storing many fewer stats.

3.7 Using HBO in query optimization rules

Several presto optimizations use HBO as the cost estimation framework. HBO is more powerful than CBO as it can store various runtime statistics related to scheduling as well. Let's take a look at few optimizations which leverage HBO to come up with better query plans.

- **Join reordering:** Presto implements a dynamic programming algorithm to explore the space of possible join trees and uses the cost estimation framework to select the most optimal join tree.
  HBO can accurately predict stats for join orders that have run in the past. For other join orders, we use CBO to predict their stats. This leads to more accurate cost estimations. However, mixing HBO and CBO stats does not always give the best results due to high variance of CBO stats. For example, HBO may give the right stats for the ideal join order, but CBO may inaccurately favor other join orders. HBO will not be able to try all the join orders to learn from them, but can learn from join orders run in other queries. Note that since we store plan stats per node (as opposed to the full query only), a query containing commonly joined tables will be able to benefit from accurate statistics for

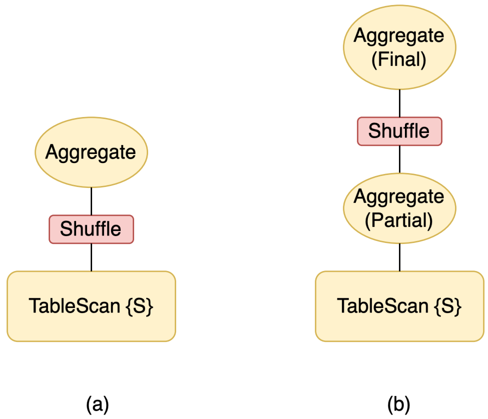


Figure 8: Partial aggregation optimization

the subplan containing those tables computed by other queries previously. HBO can thus predict better join orders, compared to only using CBO.

- **Join distribution type:** HBO is also used in deciding how to perform the distributed join. The Presto optimizer gets the size of join inputs and uses a cost model to decide whether to do broadcast join or repartition join, like the ones in Figure 2.
- **Partial aggregations** Presto can split an aggregation into partial and final stages when the aggregation is composable, such that data is pre-aggregated locally on the worker nodes before it gets shuffled across the network, see Figure 8. Partial aggregation is helpful when it can reduce the size of output. However, if partial aggregation does not decrease the size of intermediate results it will only add overhead in processing. It can also increase the size of intermediate data that gets shuffled due to the additional pre-aggregated values that get stored for every group. Thus, this optimization rule is cost-based and only triggers when the partial aggregation reduces the cardinality by more than a pre-defined ratio (typically 0.5). Note also that while the final size of the aggregated result may decrease significantly, this may not be the case for the partial aggregation stage. Using HBO we can track the output and input size of the partial aggregation and store more accurate data to guide the optimizer into choosing aggregation strategy in the future.
- **Skew mitigation** Outer joins can result in many NULL values for the outer side columns, and when one of these columns is used as a join key, this can cause skew in processing. Presto has an optimization rule which mitigates such skew by coalescing the NULL values to non-NULL values that get more evenly shuffled across workers. It works by rewriting join keys as shown in Figure 9 so that NULL values are cast to non null values that will never match. Since this is a cost-based decision dependent on the ratio of NULL values in the output, we employ the HBO framework to track the number of NULL join keys, and turn on this optimization when the ratio of NULLs exceeds a threshold. This optimization can be generalized to handle other skew scenarios beyond null skew.

```sql
-- Original query:
SELECT * FROM t1 LEFT JOIN t2 ON t1.key = t2.key

-- Rewritten query:
SELECT * FROM t1 LEFT JOIN t2
    ON COALESCE(
        CAST(t1.key AS VARCHAR),
        'l' || CAST(random(hash_partition_count) AS
            VARCHAR)
    ) = COALESCE(
        CAST(t2.key AS VARCHAR),
        'r' || CAST(random(hash_partition_count) AS
            VARCHAR)
    )
```

Figure 9: Query rewrite in NULL skew optimizer to mitigate skew in outer joins

- **Scaled writers** An important problem for INSERT queries is to correctly estimate the number of writer tasks: too many tasks may result in a lot of small files, but too few may create a bottleneck if the data to be written is large. Writer scaling is used in Presto to reduce the number of small files during INSERT operations. With the scaled writer optimization, Presto first starts with one single writer task, and as the size of output data increases, the number of writer tasks also increase. However, one disadvantage of this approach to writer scaling is that it can be a bottleneck when the number of writer tasks is low. We use the HBO framework to record the number of tasks written in historical queries, and start with half of the original number of tasks from history instead of one.

4 STATS STORE AND CONNECTOR FRAMEWORK

This section provides a detailed overview of the structure and methodology of HBO stats store.

4.1 HBO Stats structure

The stats store is essentially a key-value store with (key, value) pairs as depicted in Figure 10. Every plan node is associated with a unique key and a respective statistical value. Further details on the key and value are provided below.

Key: Hash generated from the plan

In Section 3.3, we discussed the process of plan canonicalization, and how each plan node is first transformed into a canonical form. This canonical plan is then serialized into a string, followed by the application of a SHA256 hash algorithm to create a hashed key.

Value: List of execution stats from previous runs

In Section 3.1, we investigated how execution metrics are connected to plan nodes following the query execution. We structure all statistics into thrift object, which allows future schema changes and saves storage/networks overhead using compaction. This becomes our value in the stats store.

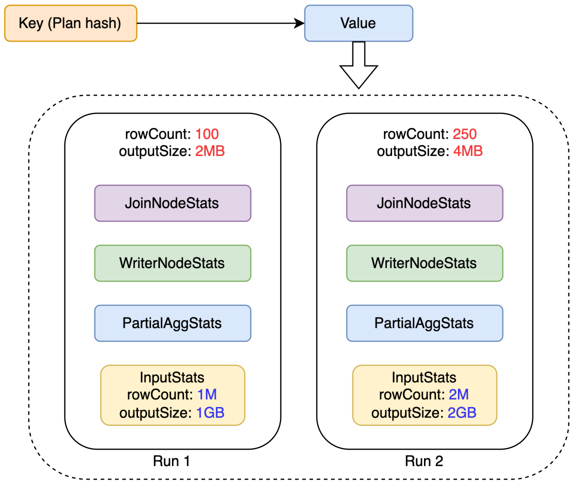


Figure 10: HBO stats structure

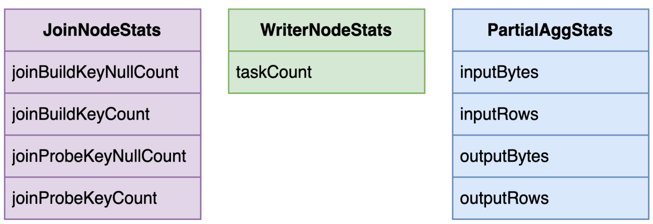


Figure 11: Plan node type-specific statistics

The mapped value contains execution statistics from several runs(Figure 10). Execution statistics include details like row count, output size in bytes, and optional additional information tailored to specific plan nodes and optimizations. Additionally, compared to CBO which needs to compute expensive stats like NDVs, HBO only stores simple statistics that can be easily computed during runtime without any additional cost. These details are shows in Figure 11. For example, we store count of null keys in both sides of join plan nodes to detect null skew. Output and input sizes for partial aggregation are stored as well. Table writer tasks store their parallelism as a useful statistic. Additionally, we also maintain a list of input stats for table scans that are present within the current plan tree for each plan node. The input statistics represent sizes of input partitions scanned. If multiple tables are present, we lay out the stats for all of them in a list. These can be sourced from file system metastore, which in our case is Hive Metastore. [34]

Time-To-Live: We assign a TTL(Time-To-Live) for every entry in the stats store. The TTL is refreshed every time an entry is updated. This way, old entries are garbage collected, reducing our storage overhead and removing entries which will never be used again.

4.2 HBO Stats Connector

The primary function of the statistics connector is to store and retrieve the stored HBO stats from our metastore.

The HBO statistics connector in Presto is implemented as a simple modular interface, leveraging Presto's plugin framework for flexibility. Presto plugin framework enables the loading of user-defined, configurable implementations at runtime through plugins.

We saw in section 3.1 that History Fetching rules are responsible for generating all relevant canonical plan nodes. These rules make internal calls to our connector API to retrieve HBO stats from the stats store. Additionally, as illustrated in Figure 4, the coordinator utilizes an in-memory cache to hold these results on a per-query basis. This approach reduces the number of network calls required and guarantees that the optimizer has access to these historical stats during the subsequent stages of planning.

Connector API: The connector API is shown in the below interface

```
# Fetch stats from stats store
def get_stats(planHashes: List[String]) ->
    Dict[String, Stats]

# Store stats in stats store
def put_stats(statistics: Dict[Hash, Stats])
```

Figure 12: Connector API

Redis Connector Implementation: We have open sourced a statistics connector that utilizes Redis [4] as a backend. As an in-memory database, Redis provides exceptionally quick read/write operations, making it highly effective for caching scenarios. In our production setup, we use Redis in a clustered configuration. The connector is integrated with a stateful Lettuce Redis client, engineered to keep persistent connections with the Redis cluster. Additionally, we also take advantage of the asynchronous API of the Lettuce Redis client, which by default utilizes Redis' pipelining capabilities and enhances performance by enabling batch processing of multiple commands. In our production setup, we have observed latencies within the range of tens of milliseconds.

5 OPERATIONAL EXPERIENCE

One of the goals when starting the HBO project to create a framework that is both powerful but also easy to operate from the standpoint of users and DBAs. In this section we describe the user experience when interacting with HBO and the various tooling we built to enable that.

Enabling HBO. Users can enable HBO by setting the following configuration parameters: `track_history_based_plan_statistics` and `use_history_based_plan_statistics`. Both of these parameters are set to true in our clusters. While we don't expect the end user to need to modify these parameter values, they are a powerful tool for debugging issues with HBO, performance testing, and allow disabling the feature to mitigate a problem should one arise.

Explain plans. Presto, like most other database engines provides a way to inspect the output of the query optimizer through an `EXPLAIN` command. In a cost-based optimizer explain plans often contain the stats estimates each plan node. We extended the output of `EXPLAIN` to also show the source of the stats estimation (cost-based or history-based).

As an example, see the explain plan in Figure 13. It is part of a query plan for UNION ALL query. For the first branch of the `UNION`

```
1                 Query Plan
2 -----------------------------------------------------------------------------------------
3 - Output[PlanNodeId 18][orderkey, cnt] => [expr_39:bigint, expr_40:bigint]
4     Estimates: {source: CostBased, rows: 75,175 (1.29MB}
5         - Aggregate(FINAL)[orderkey][PlanNodeId 3]
6             Estimates: {source: HistoryBased, rows: 15,000 (263.66kB)}
7             ...
8         - ScanProject[PlanNodeId 11,324][schemaName=tpch, tableName=lineitem]
9              Estimates: {source: CostBased, rows: 60,175 (1.03MB)]
```

Figure 13: Explain plan with details on statistics

(rows 5-6 in Figure13) the optimizer used history-based statistics from a prior run, as visible through the `source: HistoryBased` tag on the corresponding plan nodes, whereas for the second branch (lines 8-9) and final output (lines 3-4), the optimizer used the classic cost derivation to estimate the size of the output. Note also that the optimizer is seamlessly able to combine statistics derived in a different way, so the final output estimation (75K rows) is the sum of the two types of statistics estimates (15K produced by CBO and 60K extracted from HBO).

Observability. To track the overall health of HBO and provide insights into its operations at scale, we instrumented the code in various places to automatically track runtime aspects of HBO, including the plan nodes for which HBO was recorded or used (indexed by query id and plan node id), timers to track HBO latency and others. This allowed us to build dashboards exposing various operational properties of HBO including: coverage, overall accuracy, and latency of HBO. We use some of this tooling in our experimental section next but they are also part of the daily operation of our warehouse.

6 EXPERIMENTS

In this section, we evaluated HBO over two production workloads to answer the following questions

- What's the percentage of queries which can benefit from history statistics? (Section 6.2)
- How accurate the estimation from history is? (Section 6.2)
- What's the overhead of running HBO? (Section 6.4)
- How much improvement we get from HBO? (Section 6.3)

6.1 Setup

To answer the above questions, we performed a shadow experiment in the following way. We derived several workloads using real production queries with minimal changes: queries were rewritten so as not to impact any production tables - and writing data to temporary tables instead. Besides that, queries were identical to the original ones. For running the workload, we used a test cluster with the same configuration as production ones. As we had only one test cluster, compared to several production ones - we shadowed traffic from different clusters a few days at a time to capture almost all of the production workloads during testing.

We enabled HBO in our test cluster and shadowed multiple days of production workload which had HBO disabled. We first gave a couple days for HBO to start tracking the workloads and storing stats. Then we proceeded to measure several metrics and compared them against the metrics from production.

Table 1: Different HBO metrics from Meta and Uber

Metric	Meta	Uber
Accuracy	93%	90%
Query coverage	95%	95%
Plan node coverage	80%	85%
Query plan changes	30%	50%
Bad plans fixed	80%	74%

Query shapes: Our warehouse workloads include varying use-cases across dashboards, A/B testing, ETL, machine learning, graph processing, etc.[33]. We go over some details of query shapes for a better understanding of the workloads.

Relative distribution of these top plan nodes across all query plans is shown in Figure 15. 40% of nodes correspond to common Filter/Project operations, while 20% to table scans. Our optimizations described above target Aggregation, Join and TableWriter plan nodes. Let's go over how many such nodes exist in our workloads.

- Aggregations take up 30% of plan nodes, and are present in 50% of queries.
- Joins take up 6% of plan nodes, but are usually present in 25% of total queries. 90% of these queries use <= 5 joins and 99% use <= 10 joins.
- TableWriter nodes take up 5% of plan nodes, but are present in 40% of queries.

Further details on number of plan nodes in a query is given in Section 6.2.

6.2 HBO coverage and accuracy of statistics

Table 1 shows different metrics regarding observed after HBO deployment in both Meta and Uber workloads.

We found that the HBO statistics achieve P90 accuracy of 92.8%, meaning for 90% of the queries the HBO statistics are no more than 7.2% off from the actual cardinality. When HBO is applied, it yields highly accurate statistics. This means as more queries and their plan nodes get stats from HBO, they will naturally improve. Let's discuss that next.

Our analysis shows query coverage of 95%, meaning 95% of queries get statistics from historical runs. As we store stats at the granularity of plan nodes, stats are a mix of HBO and CBO stats in these queries. We measured distribution of stats coming from HBO

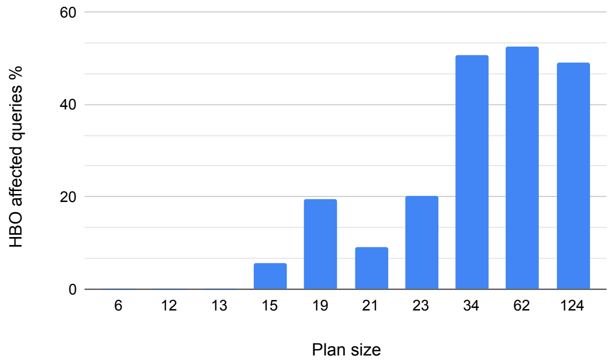


Figure 14: HBO affected queries by plan sizes

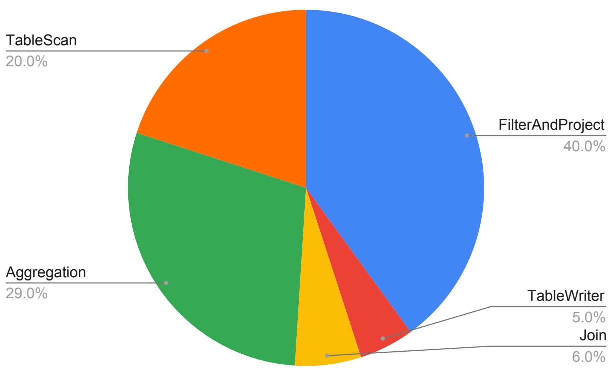


Figure 15: Top plan nodes distribution

vs CBO. 80% of plan node stats estimations come from HBO, and the other 20 % from CBO.

HBO resulted in plan changes for 30-50 % of queries in our warehouses. We ran some backtests to check how many queries previously had sub-optimal plans. Backtests included simple heuristics on runtime statistics to detect bad plans. We found 74-80% of these sub-optimal query plans improved to optimal plans using HBO. In the next section, we will go over these improvements in detail.

Figure 14 looks into how HBO affects queries of different plan sizes. The chart divides queries into 10 uniform buckets sorted by size of query plan(number of nodes in the plan tree) and plots percentage of queries affected by HBO in each bucket. We see that queries with larger query plans need HBO more than smaller ones. Larger queries are more complex, and consume more resources as well. CBO heuristics fall off with plan size, and HBO is able to step in and improve query plans. Almost 50% of queries with > 34 plan nodes are improved by HBO. On the other hand, 0% of queries with < 15 plan nodes need HBO. CBO is able to optimize these small queries by itself.

6.3 Improvement from HBO

This section shows the improvement on performance and resource utilization from HBO optimizations. We also show results for the optimizations described in Section 3.7. Overall P50 CPU and latency

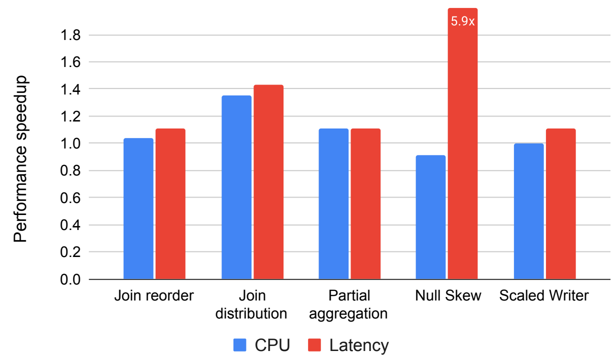


Figure 16: P50 performance improvement from rules using HBO

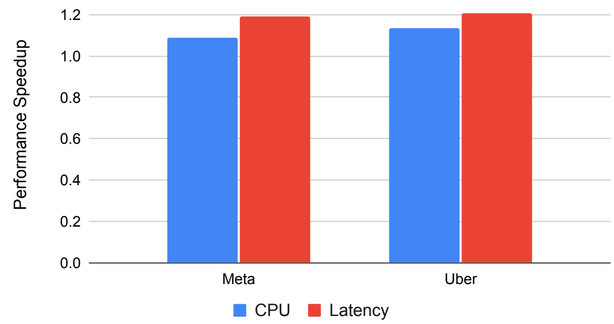


Figure 17: Overall P50 CPU and latency improvement from HBO

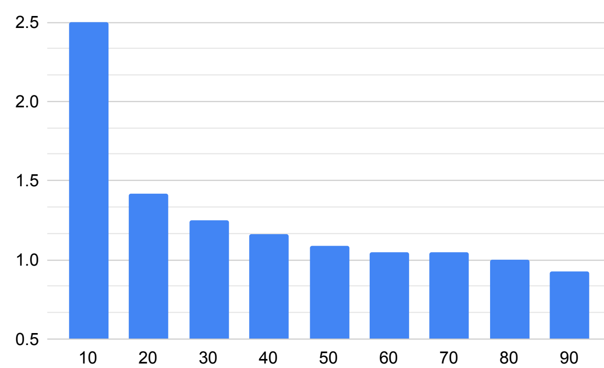


Figure 18: Percentiles of CPU improvement from HBO

improvements are shown in Figure 17. The vertical axis is the ratio of CPU cost before HBO optimization over CPU cost after HBO optimization for our workload, and similar for latency as well. Larger values means more improvement.

We see roughly 1.1x CPU improvement and 1.2x latency improvements on average across all queries affected by HBO in both Meta and Uber. These numbers hold across our diverse workloads

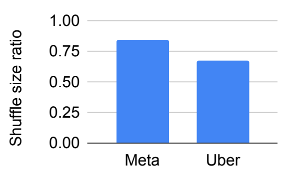


Figure 19: Shuffle size improvement

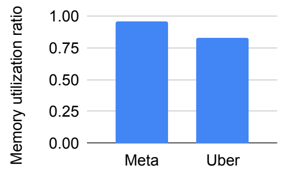


Figure 20: Memory utilization improvement

involving multiple exabyte scale data sources[33]. This is a good efficiency win for our large scale data warehouses.

Performance analysis: Figure 18 shows a histogram of cpu improvements on our workloads. We see that 10% of queries see more than 2.5x speedup, while the central quartile region sees a steady 1-1.2x speedup. For the tail end of 10% queries, we see a performance regression of roughly 8%.

We further analyze the cases with CPU time regression. Some of these queries don't see a regression per se, as other metrics like latency and memory usage improve. However, we do see cases where the query plan became sub-optimal after HBO was applied. In these cases, CBO used to predict incorrect stats, but the query plan luckily turned out fine. When HBO is applied, some of these stats become accurate, but the remaining non-HBO stats in the query lead to sub-optimal query plans. Mixing HBO and CBO stats does not always produce good results due to high variance of CBO stats. However, we have found it to help most of cases(P90), so we keep it enabled for all queries in production. Over time, the sub-optimal cases become less frequent as HBO is able to cover more stats. For example, when running a regressed query again, HBO is able to provide all the stats for the query, and the query plan is good again.

Query Optimizations: In Figure 16, we show CPU and latency improvements for the five optimizations that we discussed in Section 3.7. These specific numbers are from Meta workloads, however Uber workloads experience similar wins as well. All optimizations show improvement in both latency and CPU, except the skew optimization which has a slight regression in CPU. This is expected as we are paying a small computation cost to mitigate skew for latency improvement, which shows 5.9x improvement in latency. Note that we have compressed the vertical axis in this case for visual clarity. All other optimizations show performance improvements of 1.1-1.4x.

Join distribution optimization shows most improvements here (1.4x), since it prevents extra shuffles of large tables and joins are the bottleneck in query execution in many cases. On the other hand, aggregations are usually fast, so improvements from partial aggregation are smaller(1.1x) but these improvements cover a larger set of queries. Intuitively, we may expect improvements from join ordering to be high as well - but we found that many customers find these improvements during implementing the SQL queries. A SQL query with a bad join order can easily fail due to out of memory errors. We observed wins from join ordering to be closer to 1.1x. Scaled writer doesn't yield any CPU savings, as it just changes number of writers, but saves latency by 10%.

Figures 19, 20 show improvement in system related memory metrics in both Meta and Uber workloads after HBO rollout. The total memory usage was reduced by 5% for the Meta workload, and by 17% for the Uber workload. We also see size of shuffles within workers decreased by 17-40% in our workloads, which directly contributes to ingress/egress.

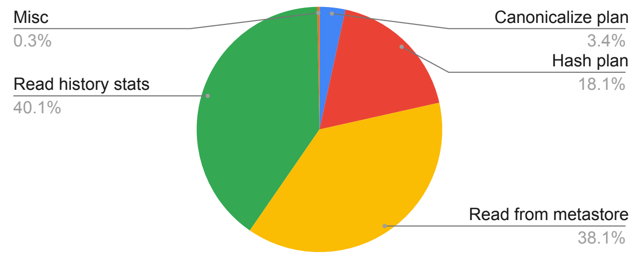


Figure 21: Latency breakdown of HBO optimizer

Table 2: Size of stats for TPCH queries

Query	Plan size	Stats size	Query	Plan size	Stats size
Q1	5	920 B	Q12	7	1.3 kB
Q2	21	3.8 kB	Q13	8	1.4 kB
Q3	10	1.8 kB	Q14	7	1.3 kB
Q4	7	1.3 kB	Q15	14	2.5 kB
Q5	15	2.7 kB	Q16	11	2.0 kB
Q6	4	0.7 kB	Q17	9	1.6 kB
Q7	15	2.7 kB	Q18	16	2.9 kB
Q8	21	3.8 kB	Q19	6	1.1 kB
Q9	15	2.7 kB	Q20	16	2.9 kB
Q10	12	2.2 kB	Q21	26	4.7 kB
Q11	21	3.8 kB	Q22	16	2.9 kB

6.4 Overhead of HBO

The compute overhead of HBO comes in two aspects, the additional latency brought by HBO and the overhead of storing history statistics.

The result shows that the additional latency from HBO can be up to 0.5% of the overall query execution time. This holds for short running queries(seconds). Longer running queries have negligible overhead. We further breakdown the latency from the HBO optimizer, which is shown in Figure 21. The largest overhead comes from reading history data (40.1%) and reading from metastore to get size of input table partitions (38.1%), followed by hashing query plan (18.1%) and canonicalizing query plan (3.4%). Most of the additional latency is dominated by network calls, resulting in overhead on the order of milliseconds.

The storage overhead for HBO is proportional to the number of query plan nodes, with each node consuming 184 bytes to store corresponding history data. Table 2 shows plan sizes and corresponding HBO stats size(for 1 run) for TPCH queries. We see an average of 13 plan nodes per query, averaging 2.3kB per query.

7 RELATED WORK

Traditional cost based optimization. Cost-based optimization has been the bread and butter of both database research and commercial systems [6, 15, 22, 30, 31, 37]. The seminal paper on System

R [10] from the 1970s introduces cost defined in terms of CPU instructions and disk page accesses, and the majority of commercial and open-source database systems have followed suit and implemented a cost-based optimizer, including SQLServer, Teradata, Oracle, PostgreSQL and others. Some of these also offer the ability to specify query hints [2, 5] to guide the optimizer in selecting an optimal plan, thus avoiding mis-calculations in cost estimation. Traditional cost-based optimizers often rely on histograms for approximating the data distribution of numeric columns [20]. A histogram typically divides the range of values into a series of intervals, and stores the count of the number of values that fall into each interval. Most systems support single-column histograms, and some have support for multi-column histograms to better manage correlations across columns [12, 16, 27]. An alternative to histograms is sampling: the query is run on a sample of the data to approximate cardinality of the full result [17, 23]. The framework we propose in this paper - HBO - is an incarnation of the ideas behind cost-based optimization, where we replace the cardinality estimator with a more accurate one based on historical query runs.

Views on stats. Perhaps the closest idea to HBO has been prior work on creating statistical views [14]. In this work, the DBMS provides a "CREATE STATISTICS ON" command that allows users to pre-compute statistics for a given subquery. During query optimization, the engine uses the same idea behind view matching to find whether there are statistical views for the query, and uses the extracted stats to estimate cost. There are several main differences between our and this work: [14] requires users and DBAs to proactively create statistical views before those can be used, possibly with the help with automatic stats view advisors [13], whereas in our approach statistics are gathered automatically with every query run. Statistical views also do not support random query expressions, as they rely on view matching to determine matches. Last but not least, view matching does not scale as well as our plan node hashing, which means it may not be applicable in situations requiring more than hundreds of unique query templates (which is easily the case in large data infra orgs such as Meta and Uber).

Learned optimizers. Recent years have seen an explosion in work on learned query optimization. One important aspect of these covers learned cardinality estimation[18, 21, 32, 35, 36]. Learned cardinality estimation techniques fall into two main categories: (1) learned data models which treat cardinality estimation as a density estimation problem and learn a joint data distribution of each data point, and (2) learned query models, which learn a mapping function between a SQL query and its cardinality on a database.

A third branch of learned optimization looks at learning the optimal values for external knobs that the engine provides to guide the optimizer [9, 24, 25]. While these have shown early promising results, they also require an initial step to learn parameter values. Moreover, they are limited to what the engine exposes as query hints and the scope at which these can be applied: most engines provide knobs only globally for the full query (as opposed to individual operators). And for the engines where hints can be specified at the level of individual operator, the learning step has a prohibitive overhead due to the exponential number of parameter value combinations for each node in the query tree.

Query template extraction and plan hashing. Finally, the ideas of plan hashing and query template extraction have been used in other areas such as automated workload analysis [8, 19].

8 CONCLUSIONS AND FUTURE WORK

In this paper we described the history-based query optimization framework in Presto and our initial findings from running it in production in two large data infrastructure organizations. Our results show that a significant number of queries benefit from historical stats. The query optimizer can make better optimization decisions due to the higher accuracy of historical stats vs traditional cardinality estimation, and we show that this helps improve various aspects of query performance, including CPU, latency and memory utilization.

In the future we plan to extend HBO in a number of ways, including making more optimizations cost-aware, as well as making the framework more robust. In addition, we are looking to improve the following areas of HBO:

Learning from failures: Currently HBO only tracks statistics after a query finished successfully. Many failed queries are from high memory usage due to bad join decisions/skew. HBO can considerably improve user experience here - make failed queries magically work after a retry. Based on our experience with users, a failing query is often retried before considering manual optimization. Failed queries often have incomplete statistics - making it trickier to learn from them. HBO can learn hints from failed executions - as to which join side is larger, how much the skew is - and fix them in the next run.

Tracking mispredictions: As HBO stores and reads statistics from a stats store, it can also track the accuracy of prediction per plan hash. If our hashing strategy leads to statistics with high variance, we can mark related estimations as "low confidence". This also provides a way to monitor accuracy of the framework over time.

Better predictions on underlying data change: Currently, HBO only returns an estimation if we find a similar run in the past which processed roughly an equal amount of data. While keeping many such runs, makes the coverage high - we can take it further using statistical models like linear regression for cases where input data differs too much. For example, its fair to assume output of Scan+Filter operator will double when underlying data doubles.

Global costing: While today many of the cost-based decisions of Presto are local, we are also investigating generalizing the query optimization framework to keep a larger space of possible query plans along the lines of Cascades [15] and doing global costing of query plans. The HBO ideas presented in this paper are extensible to that as well.

ACKNOWLEDGMENTS

We would like to thank Sreeni Viswanadha, Biswapesh Chattopadhyay, Sara Narayan, Kaushik Ravichandran, Rohit Jain, Rongrong Zhong, Masha Basmanova, Pedro Pedreira, Zhongting Hu, Hitarth Trivedi and many of our team members at Meta and Uber for the numerous discussions, code reviews and feedback on the design and implementation of HBO, as well as early drafts of this publication.

REFERENCES

[1] 2019. *Improving the Presto planner for better push down and data federation.* https://prestodb.io/blog/2019/12/23/improve-presto-planner/

[2] 2024. *Oracle query hints.* https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/Comments.html#GUID-D316D545-89E2-4D54-977F-FC97815CD62E__BABEAFGF

[3] 2024. *Presto: Distributed SQL query engine for big data.* https://github.com/prestodb

[4] 2024. *Redis in-memory key-value store.* httsp://www.redis.io

[5] 2024. *SQL server query hints.* https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16

[6] 2024. *Teradata query optimizers.* https://docs.teradata.com/r/Enterprise_IntelliFlex_VMware/SQL-Request-and-Transaction-Processing/Query-Rewrite-Statistics-and-Optimization/Query-Optimizers

[7] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-based query transformation in Oracle. In *VLDB*, Vol. 6. 1026–1036.

[8] Amirhossein Aleyasen, Mark Morcos, Lyublena Antova, Marc Sugiyama, Dmitri Korablev, Jozsef Patvarczki, Rima Mutreja, Michael Duller, Florian M. Waas, and Marianne Winslett. 2022. Intelligent Automated Workload Analysis for Database Replatforming. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2273–2285. https://doi.org/10.1145/3514221.3526050

[9] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (jun 1976), 97–137. https://doi.org/10.1145/320455.320457

[11] Kassem Awada, Mohamed Y Eltabakh, Conrad Tang, Mohammed Al-Kateb, Sanjay Nair, and Grace Au. 2020. Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere.. In *EDBT*. 534–545.

[12] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: a multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/375663.375686

[13] Amr El-Helw, Ihab F. Ilyas, and Calisto Zuzarte. 2009. StatAdvisor: Recommending Statistical Views. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1306–1317. https://doi.org/10.14778/1687553.1687556

[14] César A Galindo-Legaria, Milind M Joshi, Florian Waas, and Ming-Chuan Wu. 2003. Statistics on views. In *Proceedings 2003 VLDB Conference*. Elsevier, 952–962.

[15] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. https://api.semanticscholar.org/CorpusID:260706023

[16] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[17] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1477–1492. https://doi.org/10.1145/2723372.2749438

[18] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for out-of-the-Box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[19] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesus Camacho-Rodriguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *arXiv preprint arXiv:2401.03723* (2024).

[20] Yannis Ioannidis. 2003. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*. Elsevier, 19–30.

[21] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-Depth Study. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1214–1227. https://doi.org/10.1145/3514221.3526154

[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[23] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:8154743

[24] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (jun 2022), 6–13. https://doi.org/10.1145/3542700.3542703

[25] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[26] V. Markl, G. M. Lohman, and V. Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal* 42, 1 (2003), 98–106. https://doi.org/10.1147/sj.421.0098

[27] M. Muralikrishna and David J. DeWitt. 1988. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 28–36. https://doi.org/10.1145/50202.50205

[28] Mary Tork Roth, Laura M Haas, and Fatma Ozcan. 1999. *Cost models do matter: Providing cost information for diverse data sources in a federated system.* IBM Thomas J. Watson Research Division.

[29] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[30] Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, and César Galindo-Legaria. 2012. Query Optimization in Microsoft SQL Server PDW. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 767–776. https://doi.org/10.1145/2213836.2213953

[31] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 337–348.

[32] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and a Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (sep 2021), 85–97. https://doi.org/10.14778/3485450.3485459

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

[34] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1626–1629. https://doi.org/10.14778/1687553.1687609

[35] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready for Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (may 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[36] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* 1, 1, Article 41 (may 2023), 27 pages. https://doi.org/10.1145/3588721

[37] Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, and Khaled Yagoub. 2008. Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11g. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1346–1355. https://doi.org/10.14778/1454159.1454175