Starburst
ANALYTICS ANYWHERE

Starburst

Performance Whitepaper

July 2021

In this Document

SECTION ONE

Introduction

Trino was built from the ground up to run fast queries at scale. It was built to solve the problem associated with using Hive to run interactive queries on large data warehouses. The creators of the Trino (formerly known as Presto SQL) project, needed to handle hundreds of users issuing tens of thousands of queries each day against petabytes of data. Rather than moving your data to yet another system, Trino was made to read the data from where it was stored.

The Primary Goal of Trino

One of the first concepts that is foreign to folks who are new to Trino is that it is a distributed query engine that has no storage. Trino's primary goal is to sit atop of a data lake which directly accesses a filesystem or object storage and utilizes metadata to discover and retrieve relevant data to a query. Being able to run federated queries was a pleasant side effect that developed from addressing some requests at Facebook to query internal services and of Trino being open source needing to provide federated queries over common databases in the community.

To someone who knows a little about query engines but hasn't developed one themselves, they may think that creating a cost-based optimizer, pushdown predicates, and other improvements like dynamic filtering are features that would be included in the design from day one. You wouldn't be wrong if we weren't talking about a distributed system that services multiple storage layers (i.e. databases). Writing a cost-based optimizer for a single system relies on the information available within a single system, many times on the same node. The fact is that Trino not only has to obtain various statistics and other metadata from multiple systems, it needs to create a plan that will take advantage of its parallel MPP architecture.

This is where Starburst offers a great deal of value. Starburst connectors expose a lot of statistics which in turn allows the query engine to generate faster and more optimal query plans. Since Trino is multi-tenant, it has to apply all of these optimizations while not overburdening any one node in the system. This is a very critical aspect as taking down a node can cause multiple queries to fail depending on the system boundaries that are configured.

Needless to say, this is something that grew and developed over the years rather than something that was started on day one. This same principle applies to other performance improvements such as applying pushdown predicates. As you can imagine, it will always depend more on what pushdown predicates have been made available in the connector that will dictate the ability to push down a given predicate. This is yet another way in which the Starburst connectors optimize the total sum performance of your queries. Pushing down a filter on a query will avoid the underlying database sending back unnecessary data for the given query. This saves time, network, and CPU resources and cost. Optimizing joins in a distributed manner is no exception to the rule either. The idea initially is to make the join work as quickly and correctly using the basic MPP principles. Dynamic filtering takes these optimizations one step further by distributing a list of join keys from the inner table to the outer table nodes to filter out rows as they stream from the outer table. This adds a lot of complexity to the system and if not done correctly could result in incorrect results.

Knowing this provides the proper context to appreciate the complexity of the following, and informs everyone why these features that have existed in databases for ages are not as straightforward to implement in distributed systems. This whitepaper explores the performance improvements existing in the open source Trino project as well as the improvements made by the Starburst team.

SECTION TWO

Cost-based Optimizer

Depending on the complexity of your SQL query there are many, often exponential, query plans that return the same result. However, the performance of each plan can vary drastically; taking only seconds to finish or days given the chosen plan.

That places a significant burden on analysts who will then have to know how to write performant SQL. This problem gets worse as the complexity of questions and SQL queries increases. In the absence of an optimizer, queries will be executed with syntactic ordering in the way the query was written. That's a burden that isn't reasonable to place on a group of users who know what questions need answering but who aren't, and shouldn't have to be, wizards with SQL. An analyst should be able to submit a query and get a fast result without having to worry about how to optimally structure the question in SQL.

With this problem in mind, developers at Starburst developed Trino's first Cost-Based Optimizer (CBO). Packaged alongside the 195e release (and higher), the CBO's primary job is to explore the space of possible query plans and to find the most optimal. The CBO removes the burden from the end user of knowing how to write performant SQL.

The Cost-Based Optimizer (CBO) has shown impressive results in industry standard benchmarks since its release in early April 2018, claiming speeds up to 13 times faster than the competition. Thanks to its extensive decision-making process, based on the shape of the query, filters, and table statistics, the CBO can evaluate and execute the most optimal query plan among the countless alternatives.

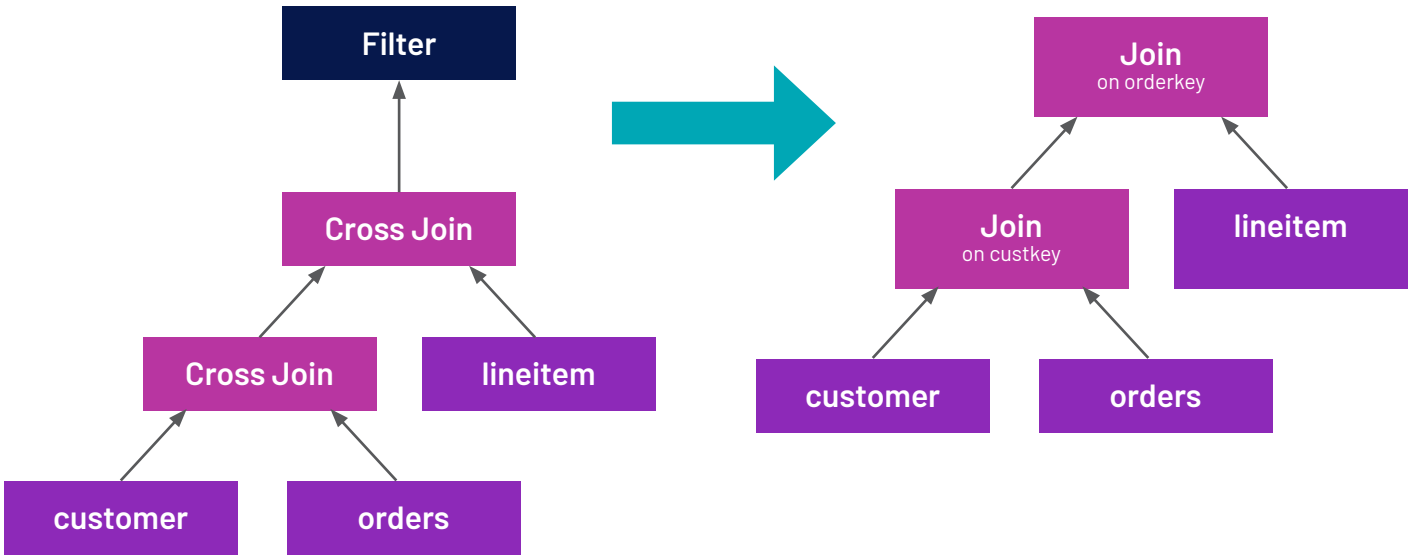
Background

An analysis of the CBO and its methods of statistical analyses first require a contextual framework. Let's consider a data scientist who wants to understand which of the company's customers spend the most money. They submit a query similar to that below.

```
SELECT c.custkey, sum(l.price)
FROM customer c, orders o, lineitem l
WHERE c.custkey = o.custkey AND l.orderkey = o.orderkey
GROUP BY c.custkey
ORDER BY sum(l.price) DESC;
```

Once the above query is put into action, Trino must create a plan for its execution. It does so by first transforming the query into its simplest possible plan. It will create `CROSS JOINS` for the "`FROM customer c, orders o, lineitem l`" part of the query and `FILTER` for "`WHERE c.custkey = o.custkey AND l.orderkey = o.orderkey`". This initial plan is quite naïve, as `CROSS JOINS` will produce humongous amounts of intermediate data. In fact, there is no point in even trying to execute such a plan, and Trino never does. Instead, it transforms the plan to create one more in line with the user's expectations, as shown below.

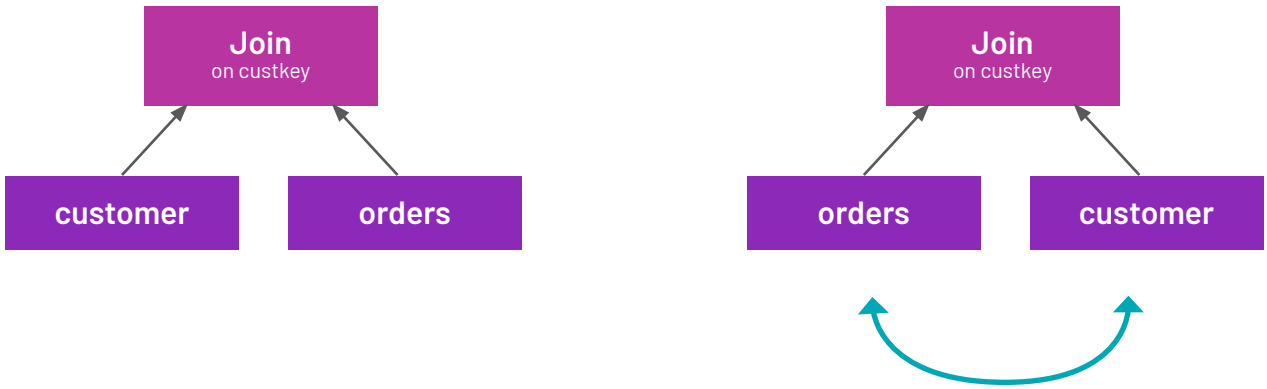
Note: for succinctness, only part of the query plan is drawn, without aggregation ("`GROUP BY`") and sorting ("`ORDER BY`").



Indeed, this is much better than the original `CROSS JOINS` plan. Yet still, a more optimal execution plan can be reached if costs are considered.

Cost-Based Optimizer

Without going into database internals on how `JOIN` is implemented, it is important to understand that it makes a big difference which table is right and which is left in the `JOIN`; the simple explanation being that the table on the right needs to be kept in the memory while the `JOIN` result is calculated. Consequently, the following plans produce the same result but have potentially different execution time or memory requirements.



CPU time, memory requirements, and network bandwidth usage are the three dimensions that contribute to query execution time, both in a single query and concurrent workloads. These dimensions capture the "cost" or overall efficiency of each query plan. Thus, it is important to consider the position and size of the tables within your `JOIN` in order to limit the cost of your query.

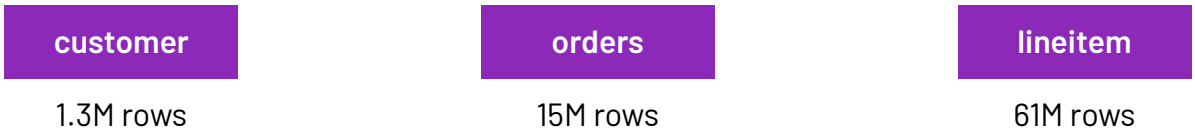
Say our data scientist knows that most of the customers made at least one order, that each of those orders had at least one item, and that many orders had many items. They would then know that "`lineitem`" is the biggest table, "`orders`" is the medium table and "`customer`" is the smallest table. When joining "`customer`" and "`orders`" in this case, they would put "`orders`" on the left side of the `JOIN` and "`customer`" on the right, as `customer` is the smaller table. But the query planner cannot be cognizant of such information for every query they submit. In reality, the planner cannot reliably deduce information from table names alone; for this, statistics are needed.

Table Statistics

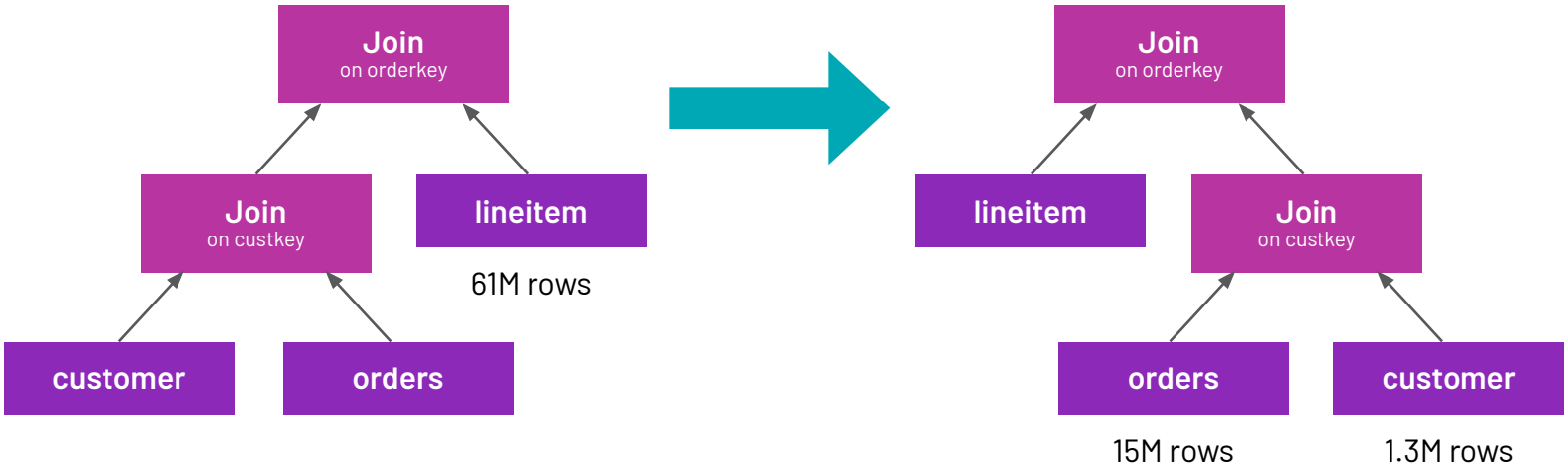
It is important to understand that Trino has a connector-based architecture, and such connectors can provide table and column statistics:

- Number of rows in a table
- Number of distinct values in a column
- Fraction of `NULL` values in a column
- Minimum/maximum value in a column
- Average data size for a column

And so, if some information is missing — the average text length in a varchar column for example — a connector can still provide information in which the Cost-Based Optimizer can take advantage. In our data scientist's example, data sizes can look something like the following:



Having this knowledge, Trino's Cost-Based Optimizer will think up a completely different `JOIN` ordering in the query plan.

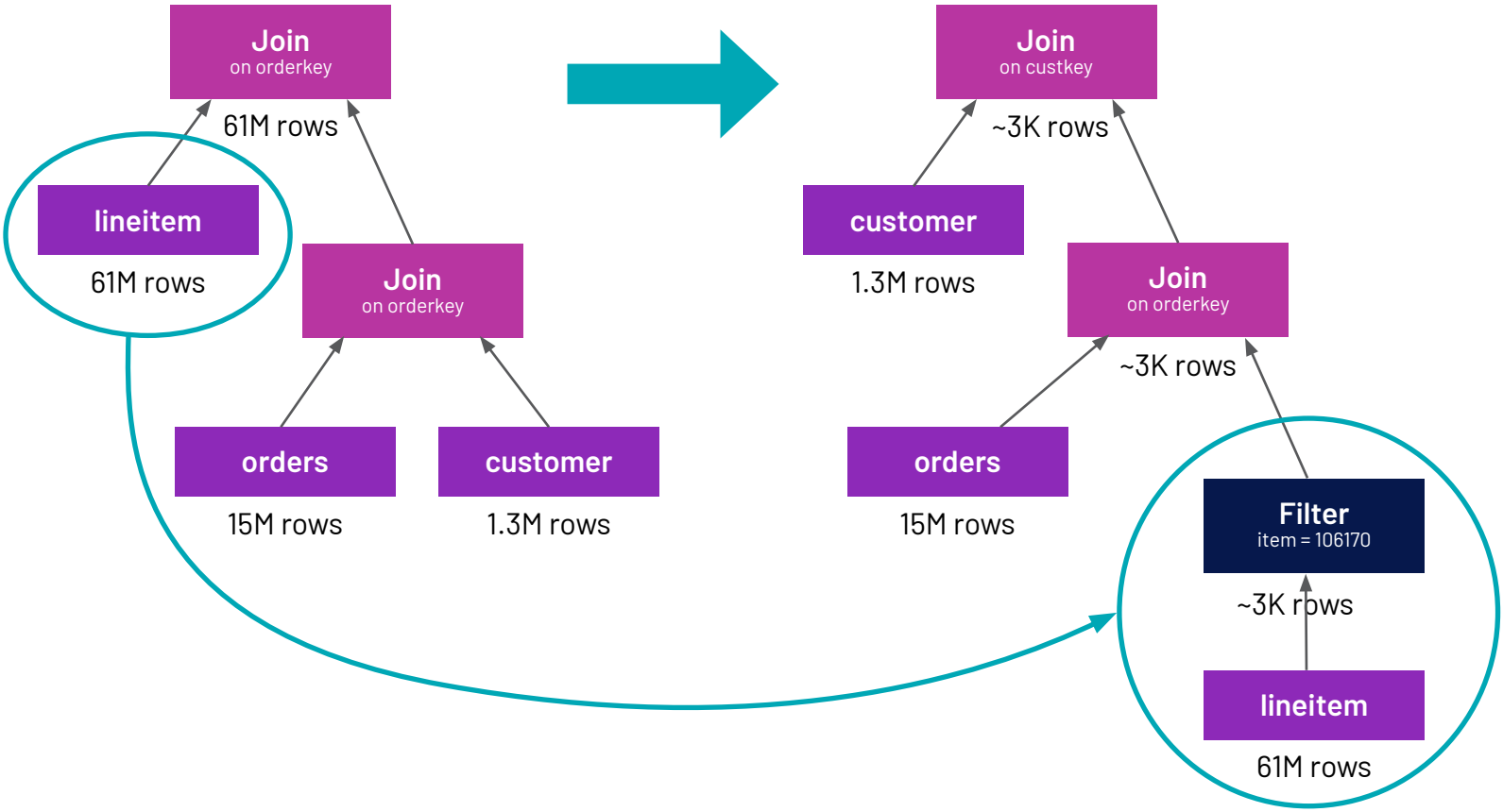


Filter Statistics

As we saw, knowing the sizes of the tables involved in a query is critical to reordering JOINs properly in the query plan. However, knowing just the sizes is not enough. Returning to our example, consider the data scientist wants to understand which customers repeatedly spent the most money on a particular item. For this, they will use a query almost identical to the original, but with the addition of the bolded condition below.

```
SELECT c.custkey, sum(l.price)
FROM customer c, orders o, lineitem l
WHERE c.custkey = o.custkey AND l.orderkey = o.orderkey
      AND l.item = 106170
GROUP BY c.custkey ORDER BY sum(l.price) DESC;
```

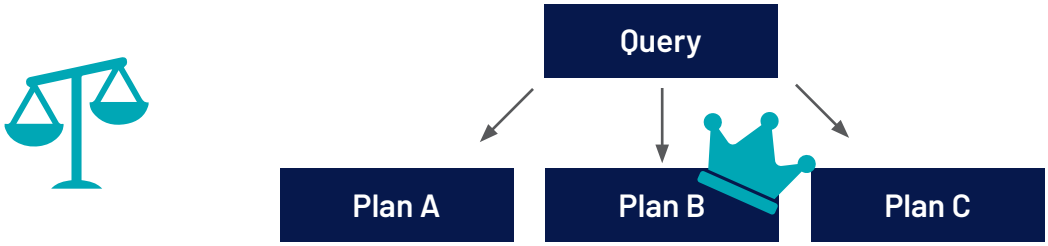
The additional FILTER could be applied after the JOIN or before. However, filtering as early as possible is the best strategy, as it reduces the amount of data you are working with from the beginning. In this case, the actual size of the data involved in the JOIN will be smaller. This can also alter the order in which your JOIN is executed, as shown in our data scientist's example.



Execution Time and Cost

From an external perspective, only three things matter when it comes to query optimization:

- **Execution Time:** The execution time is often called "wall time" to emphasize that we're not really interested in "CPU time" or the number of machines, nodes, or threads involved.
- **Execution Cost:** The execution cost is the actual cost in dollars of running the query. A CFO will be most interested in this metric, as keeping cluster costs as low as possible is a large priority.
- **Concurrent Queries:** A check for concurrent queries ensures that all cluster users can work at the same time. That is, that the cluster can handle many queries at a time, yielding enough throughput that "wall time" observed by each of the users is satisfactory.



It is only possible to optimize for one of the above dimensions. For example, you can have a single node cluster which would mitigate execution costs but drive up execution times and leave workers waiting idly. Contrarily, we may have a thousand-node cluster that cuts down execution times, but greatly increases execution costs for the organization. Ultimately, such tradeoffs must be balanced, such that queries are executed as fast as possible, with as little resources as possible.

In Trino, this is modeled with the concept of cost, which captures properties like CPU cost, memory requirements, and network bandwidth usage. In search of the most optimal solution, different variants of a query execution plan are explored, assigned a cost, and compared. Then, the variant with the least overall cost is selected for execution. This approach neatly balances the needs of each party — cluster users, administrators, and the CFO.

The cost of each operation in the query plan is calculated with the type of the operation in mind, taking into account the statistics of the data involved. Now, let's see where the statistics come from.

Statistics

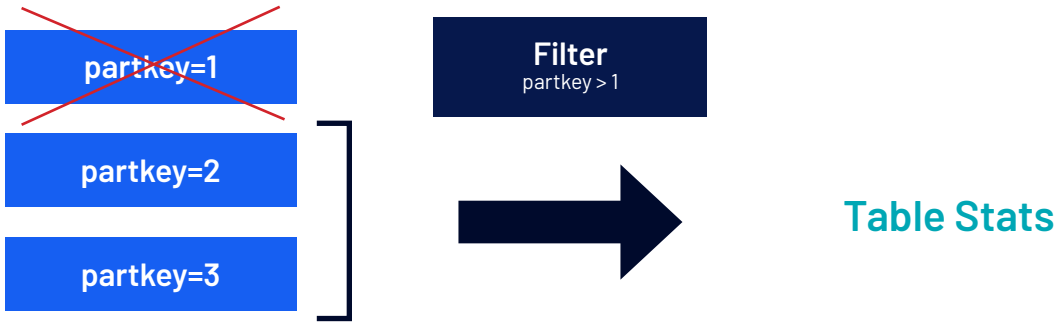
In our data scientist's example, the row counts for tables were taken directly from table statistics, i.e., provided by a connector. But where did "~3K rows" come from? Let's take a deeper dive into some of these details.

A query execution plan is made of "building block" operations, including:

- Table scans (reading the table; at runtime this is actually combined with a `FILTER`)
- `FILTER`s (SQL's `WHERE` clause or any other conditions deduced by the query planner)
- Projections (i.e., computing output expressions)
- `JOIN`s
- Aggregations (in fact there are a few different "building blocks" for aggregations, but that's a story for another time)
- Sorting (SQL's `ORDER BY`)
- Limiting (SQL's `LIMIT`)
- Sorting and limiting combined (SQL's `ORDER BY .. LIMIT ..` deserves specialized support)
- ETC.

The process by which these statistics are computed for such "building blocks" is discussed on the next page.

Table Scan Statistics



As explained earlier, the connector which defines the table is responsible for providing the table statistics. Further, the connector is informed of any filtering conditions that are to be applied to the data read from the table. This may be important in the case of a Hive partitioned table for example, where statistics are stored on a per-partition basis. If the filtering condition excludes some or many partitions, the statistics will consider a smaller data set (remaining partitions) and will be more accurate.

To recall, a connector can provide the following table and column statistics:

- Number of rows in a table,
- Number of distinct values in a column
- Fraction of `NULL` values in a column
- Minimum/maximum value in a column
- Average data size for a column

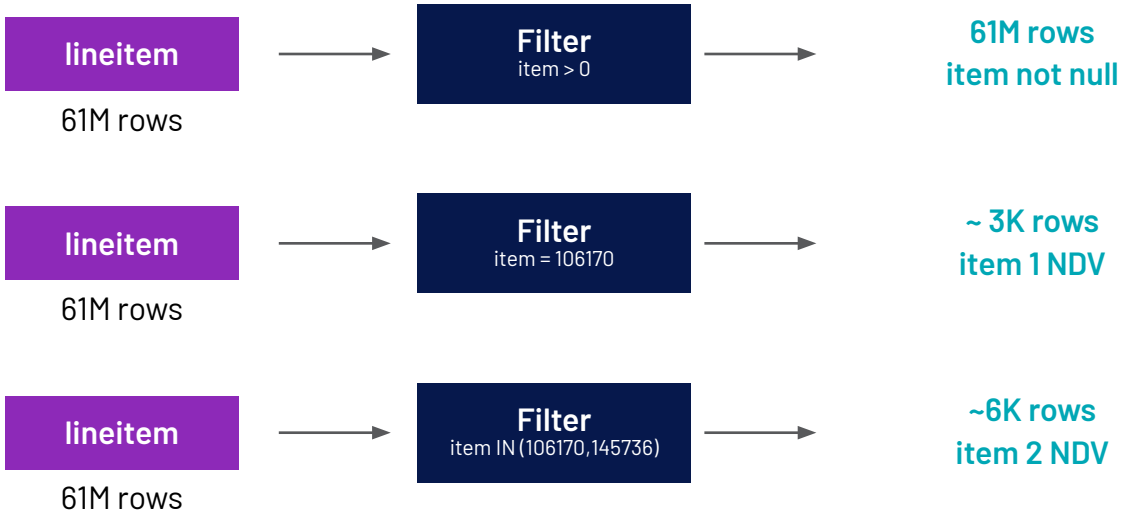
Filter Statistics

When considering a filtering operation, a filter's condition is analyzed, and the following estimations are calculated:

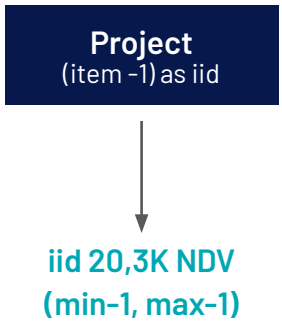
- What is the probability that the data row will pass the filtering condition? From this, the expected number of rows after the filter is derived
- Fraction of `NULL` values for columns involved in the filtering condition (for many conditions, this will simply be 0%),
- Number of distinct values for columns involved in the filtering condition
- Number of distinct values for columns that were not part of the filtering condition, if their original number of distinct values was more than the expected number of data rows that pass the filter

For example, for a condition like "`l.item = 106170`" we can observe that:

- No rows with "`l.item`" being `NULL` will meet the condition
- There will be only one distinct value of "`l.item`" (106170) after the filtering operation
- On average, number of data rows expected to pass the filter will be equal to the `number_of_input_rows * fraction_of_non_nulls / distinct_values`. (This assumes, of course, that users most often drill down in the data they really have, which is quite a reasonable and safe assumption to make.)



Projection Statistics



Projections ("`l.item - 1 AS iid`") are similar to filters, except that they do not impact the expected number of rows after the operation.

For a projection, the following types of column statistics are calculated (if possible for the given projection expression):

- Number of distinct values produced by the projection
- Fraction of `NULL` values produced by the projection
- Minimum/maximum value produced by the projection

Naturally, if "`iid`" is only returned to the user, then these statistics are not useful. However, if it's later used in a `FILTER` or `JOIN` operation, these statistics are important to the proper estimation of the number of rows that meet the `FILTER` condition or are returned from the `JOIN`.

SECTION THREE

Connecter-based Performance Improvement

Starburst provides connectors to the most popular data sources included in many of these connectors are a number of exclusive enhancements. Many of Starburst's connectors when compared with open source Trino have enhanced extensions such as parallelism, pushdown and table statistics, that drastically improve the overall performance. Parallelism distributes query processing across workers, and uses many connections to the data source at the same time for increased overall performance. Pushdown and Starburst Cached Views are discussed in the section below.

Pushdown

Introduction

In addition to the performance improvement associated with the Cost-Based Optimizer, Trino can push down the processing of queries, or parts of queries, into the connected data source. This means that a specific predicate, aggregation function, or other operation, is passed through to the underlying database or storage system for processing.

The results of this pushdown can include the following benefits:

- Improved overall query performance
- Reduced network traffic between Trino and the data source
- Reduced load on the remote data source

Support for pushdown is specific to each connector and the relevant underlying database or storage system.

Predicate and Column Projection Pushdown

In addition to the computation aspects discussed in the CBO section above, the amount of data that is brought back to the cluster can also be an important factor affecting performance. For example, I/O costs and network factors can mean that the transfer of data from source systems to the Starburst cluster can either be rate limiting or represent a significant chunk of time. As a consequence, there are several optimizations built in to ensure that only a minimal amount of data is brought back to the cluster for processing. After all, you don't want to transfer unnecessary amounts of data to the cluster just to filter it out after it gets here.

The majority of connectors will support some form of predicate pushdown to reduce the number of rows fetched from the source. Depending on the specific data source, connectors may leverage advanced capabilities, such as partition pruning, index scans, block-level min/max indices, hash-based bucketing, data clustering, to skip reading unnecessary data.

Additionally, only columns necessary for further query stages will be requested. That allows a connector to utilize inherent data source capabilities to fetch needed columns efficiently. For example, data may be organized in a columnar fashion at the source to facilitate fast access.

It is worth noting that the above mentioned pushdown optimizations can be applied very broadly when reading data from RDBMS, HDFS, Object Storage, and NoSQL systems. The specific features and implementation will naturally differ between the connectors and file formats. As an example, unpartitioned Hive tables with text files will have quite limited benefit from pushdown, while the same data organized in partitions and stored in a columnar file format such as ORC or Parquet, will benefit significantly.

As an example, we loaded the TPCH data set into a PostgreSQL database and then queried it using the PostgreSQL connector:

```
SELECT nationkey, lower(name)
FROM nation
WHERE regionkey = 1;
```

In this case, PostgreSQL will run the following query:

```
SELECT nationkey, name
FROM nation
WHERE regionkey = 1;
```

and then Starburst will apply the `lower()` function before returning the result to the client.

Aggregation Pushdown

Aggregation pushdown means that, when possible, aggregation operations are executed in source systems. That has the effect of reducing the amount of data transferred over the network to the cluster. Starburst's optimizer will detect when aggregation pushdown is possible and execute it on the fly for analysts.

Aggregation pushdown can take place provided the following conditions are satisfied:

- If aggregation pushdown is generally supported by the connector.
- If pushdown of the specific function or functions is supported by the connector.
- If the query structure allows pushdown to take place.

You can check if pushdown for a specific query is performed by looking at the EXPLAIN plan of the query. If an aggregate function is successfully pushed down to the connector, the explain plan does not show that Aggregate operator. The explain plan only shows the operations that are performed by Trino.

Let's consider another example in which the following SQL statement is executed in Starburst against data in a PostgreSQL database:

```
SELECT regionkey, count(*)
FROM nation
GROUP BY regionkey;
```

In this case, PostgreSQL will run the following query:

```
SELECT regionkey, count(*)
FROM nation
GROUP BY regionkey;
```

Basically, the entire query was eligible for pushdown which can be confirmed by inspecting the output of `EXPLAIN`.

A number of factors can prevent a push down:

- Adding a condition to the query
- Using a different aggregate function that cannot be pushed down into the connector
- Using a connector without pushdown support for the specific function

As a result, the explain plan shows the Aggregate operation being performed by Trino. This is a clear sign that now pushdown to the remote data source is not performed, and instead Trino performs the aggregate processing.

Join Pushdown

Join pushdown provides similar performance benefits to the discussion above on aggregation pushdown in that join operations can be performed in source systems prior to transferring the data to the cluster. Join pushdown allows the connector to delegate the table join operation to the underlying data source. For selective join operations that has the impact of reducing the amount of data transferred over the network to the cluster. This can result in performance gains, and allows Trino to perform the remaining query processing on a smaller amount of data.

The specifics for the supported pushdown of table joins varies for each data source, and therefore for each connector.

Limited Pushdown

A `LIMIT` or `FETCH FIRST` clause reduces the number of returned records for a statement. Limit pushdown enables a connector to push processing of such queries of unsorted record to the underlying data source.

A pushdown of this clause can improve the performance of the query and significantly reduce the amount of data transferred from the data source to Trino.

Queries include sections such as `LIMIT N` or `FETCH FIRST N ROWS`.

Implementation and support is connector-specific since different data sources have varying capabilities.

Top-N Pushdown

The combination of a `LIMIT` or `FETCH FIRST` clause with an `ORDER BY` clause creates a small set of records to return out of a large sorted dataset. It relies on the order to determine which records need to be returned, and is therefore quite different to optimize compared to a Limit pushdown.

The pushdown for such a query is called a Top-N pushdown, since the operation is returning the top N rows. It enables a connector to push processing of such queries to the underlying data source, and therefore significantly reduces the amount of data transferred to and processed by Trino.

Queries include sections such as `ORDER BY` ... `LIMIT N` or `ORDER BY` ... `FETCH FIRST N ROWS`. Implementation and support is connector-specific since different data sources support different SQL syntax and processing.

Starburst Cached Views

Starburst Cached Views is a collection of performance features. Starburst Cached Views includes table scan redirection which allows read operations in a source catalog to be transparently replaced by reading the data from another catalog, the target catalog. The cache service which manages the configuration and synchronization of the source and target catalogs. And the cache service command line interface (CLI) which provides a terminal-based interface for listing and configuring redirections managed by the cache service.

Table Scan Redirection

For those use cases where the data needing to be access-optimized is in a relational or NoSQL database, Starburst has a platform feature that allows for some, or all, of the data to be mirrored into a faster, closer or even more cost-effective location. Table scan redirection enables Starburst to offload data from tables accessed in one catalog to equivalent tables accessed in another catalog. This can improve performance by shifting data access to a more performant system. It can also reduce load on a data source.

Once enabled, all queries will automatically be directed to the mirrored data location by the Starburst platform. This allows for a simple and efficient deployment that does not affect users or applications. Schedules can be set up to refresh all or incremental portions of the data as desired.

Redirection is transparent to the user, and therefore provides performance improvements without the need to modify queries.

A typical use case is the redirection from a catalog configuring a relational database to a catalog using the Hive connector to access a data lake. That catalog can also take advantage of Hive connector storage caching.

An additional use case is when queries need to access data spread over complex environments such as multi-cloud or hybrid on-prem/cloud. Under those circumstances performance can take a hit if you need to pull large amounts of data over the network. Table scan redirections allow you to materialize the data from remote locations in a center of data gravity. That access pattern allows us to get good performance when federating between geographically distinct locations.

Redirection of table scans is performed by Starburst after applying authentication and permission checks from the source catalog.

Most Starburst connectors support table scan redirection. A comprehensive list can be found in the documentation.

The target catalog can use an identical connector for maximum compatibility, or any other connector. Data types are translated based on the type mapping of the connector used for the source and target catalog. This type mapping can be customized to work around unsupported types by setting an explicit type mapping for the target catalog.

If table properties like partitioning, bucketing, sorting are used, then the target can only be Hive as other connectors don't support these table properties.

Cache Service

Starburst's cache service provides the ability to configure and automate the management of table scan redirections. The service connects to an existing Starburst installation to run queries for copying data from the source catalog to the target catalog. The target catalog is regularly synchronized with the source and used as a cache.

The cache service can be run as a standalone service or within the coordinator process. You can interact with it using its REST API, or the cache service CLI.

SECTION FOUR

Dynamic Filtering

Dynamic filtering optimizations significantly improve the performance of queries with selective joins by avoiding reading of data that would be filtered by join condition. In this respect, dynamic filtering is similar to join pushdown discussed above, however it is the equivalent of inner join pushdown across data sources. As a consequence we derive the performance benefits associated with selective joins when performing federated queries. That significantly boosts the performance of query federation use cases.

Consider the following query which captures a common pattern of a fact table store_sales joined with a filtered dimension table date_dim:

```
SELECT count(*)
FROM store_sales JOIN date_dim
     ON store_sales.ss_sold_date_sk = date_dim.d_date_sk
WHERE d_following_holiday='Y' AND d_year = 2000;
```

Without dynamic filtering, Trino pushes predicates for the dimension table to the table scan on `date_dim`, and it scans all the data in the fact table since there are no filters on `store_sales` in the query. The join operator ends up throwing away most of the probe-side rows as the `JOIN` criteria is highly selective.

When dynamic filtering is enabled, Trino collects candidate values for join condition from the processed dimension table on the right side of join. In the case of broadcast joins, the runtime predicates generated from this collection are pushed into the local table scan on the left side of the join running on the same worker.

Dynamic filtering is enabled by default using the `enable-dynamic-filtering` configuration property. To disable dynamic filtering, set the configuration property to false. Alternatively, use the session property `enable_dynamic_filtering`.

Additionally, these runtime predicates are communicated to the coordinator over the network so that dynamic filtering can also be performed on the coordinator during enumeration of table scan splits.

For example, in the case of the Hive connector, dynamic filters are used to skip loading of partitions which don't match the join criteria. This is known as dynamic partition pruning.

The results of dynamic filtering optimization can include the following benefits:

- improved overall query performance
- reduced network traffic between Trino and the data source
- reduced load on the remote data source

Support for push down of dynamic filters is specific to each connector, and the relevant underlying database or storage system.

Analysis and confirmation

Dynamic filtering depends on a number of factors:

- Planner support for dynamic filtering for a given join operation in Trino. Currently inner and right joins with `=`, `<`, `<=`, `>`, `>=` or `IS NOT DISTINCT FROM` join conditions, and semi-joins with `IN` conditions are supported.
- Connector support for utilizing dynamic filters pushed into the table scan at runtime. For example, the Hive connector can push dynamic filters into ORC and Parquet readers to perform stripe or row-group pruning.
- Connector support for utilizing dynamic filters at the splits enumeration stage.
- Size of right (build) side of the join.

You can take a closer look at the `EXPLAIN` plan of the query to analyze if the planner is adding dynamic filters to a specific query's plan. For example, the explain plan for the above query can be obtained by running the following statement:

```
EXPLAIN
SELECT count(*)
FROM store_sales JOIN date_dim
     ON store_sales.ss_sold_date_sk = date_dim.d_date_sk
WHERE d_following_holiday='Y' AND d_year = 2000;
```

The EXPLAIN plan for this query shows dynamicFilterAssignments in the InnerJoin node with dynamic filter df_370 collected from build symbol d_date_sk. You can also see the dynamicFilter predicate as part of the Hive ScanFilterProject operator where df_370 is associated with probe symbol ss_sold_date_sk. This shows you that the planner is successful in pushing dynamic filters down to the connector in the query plan.

Fragment 1 [SOURCE]

```
Output layout: [count_3]
Output partitioning: SINGLE []
Stage Execution Strategy: UNGROUPED_EXECUTION
Aggregate(PARTIAL)
    Layout: [count_3:bigint]
    count_3 := count(*)
    InnerJoin[(“”ss_sold_date_sk”” = “”d_date_sk””)][$hashvalue,
    $hashvalue_4]
        Layout: []
        Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}
        Distribution: REPLICATED
        dynamicFilterAssignments = {d_date_sk -> #df_370}
        ScanFilterProject[table = hive:default:store_sales, grouped =
        false, filterPredicate = true, dynamicFilters = {“”ss_sold_date_
        sk”” = #df_370}]
            Layout: [ss_sold_date_sk:bigint, $hashvalue:bigint]
            Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/
            {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/{rows: 0 (0B),
            cpu: 0, memory: 0B, network: 0B}
            $hashvalue := combine_hash(bigint ‘0’,
            COALESCE(“”$operator$hash_code””(“”ss_sold_date_sk””), 0))
            ss_sold_date_sk := ss_sold_date_sk:bigint:REGULAR
            LocalExchange[HASH][$hashvalue_4] (“”d_date_sk””)
                Layout: [d_date_sk:bigint, $hashvalue_4:bigint]
                Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}
                RemoteSource[2]
                    Layout: [d_date_sk:bigint, $hashvalue_5:bigint]
```

Fragment 2 [SOURCE]

```
Output layout: [d_date_sk, $hashvalue_6]
Output partitioning: BROADCAST []
Stage Execution Strategy: UNGROUPED_EXECUTION
ScanFilterProject[table = hive:default:date_dim, grouped = false,
filterPredicate = ((""d_following_holiday"" = CAST('Y' AS char(1)))
AND (""d_year"" = 2000))]
    Layout: [d_date_sk:bigint, $hashvalue_6:bigint]
    Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/
    {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/{rows: 0 (0B),
    cpu: 0, memory: 0B, network: 0B}
    $hashvalue_6 := combine_hash(bigint '0',
    COALESCE(""$operator$hash_code""(""d_date_sk""), 0))
    d_following_holiday := d_following_holiday:char(1):REGULAR
    d_date_sk := d_date_sk:bigint:REGULAR
    d_year := d_year:int:REGULAR
```

During execution of a query with dynamic filters, Trino populates statistics about dynamic filters in the QueryInfo JSON available through the Web UI. In the queryStats section, statistics about dynamic filters collected by the coordinator can be found in the dynamicFiltersStats structure.

Push down of dynamic filters into a table scan on the worker nodes can be verified by looking at the operator statistics for that table scan. dynamicFilterSplitsProcessed records the number of splits processed after a dynamic filter is pushed down to the table scan.

Dynamic filters are reported as a part of the `EXPLAIN ANALYZE` plan in the statistics for ScanFilterProject nodes.


Starburst
ANALYTICS ANYWHERE