Search Space Improvements: Plan Constraints

This document discusses state-of-the-art with respect to plan constraints for use in plan lockdown and hint scenarios and Presto's current limitations in this area. The discussion also presents detailed design considerations for removing these limitations. This document is one in a series of search space-related discussion documents that focus on expanding and better controlling the space of plan alternatives that the Presto optimizer considers.

Background

Relational query languages provide a high-level declarative interface for accessing relational data. The job of the query optimizer is to translate a declarative query into an execution plan that correctly and efficiently returns the final query result [1]. The optimization process involves enumerating alternative execution plans from a search space of feasible alternatives and choosing one that minimizes a cost estimate of the efficiency of the execution plan. The optimizer generates its search space by applying transformation rules that logically map a query to equivalent algebraic representations and implementation rules that map those algebraic representations to execution plans consisting of physical operators, executable by the query evaluation engine.

The evolution of any query optimizer typically involves expanding the plan search space by considering new execution plans that were previously unknown (and costing them appropriately). In this expanded search space, it is possible that the optimizer chooses for execution a query plan that is more computationally expensive than was previously chosen. This could be due to incorrect statistics, plan cost calculations, heuristics, etc. One common stumbling block for users is when query performance degrades due to different query plans after a database version upgrade. A common approach to avoid this problem is some mechanism that allows the user to "lockdown" a known, well-performing query plan for an individual query. This allows the user to instruct the optimizer to pick a query plan with a set of characteristics that enable efficient execution. Some of these characteristics could be - the order in which some tables are joined, the access methods for some tables, the order in which some predicates are evaluated, etc. In Presto, these transformations are performed through

well-defined optimizer rules that operate independent of each other, and can therefore be easily reversed. The lockdown of cost-based decisions, i.e. choosing one execution strategy from a pool of logically equivalent ones based on cost, are the ones that benefit most. Consequently, in this document we shall focus on describing a framework that will allow a user to lockdown the cost-based decisions of the Presto optimizer.

An extension to the lockdown use-case is when the user can come up with a query plan that performs better than the plan that the optimizer produces. In this case, the better plan may run faster, use less resources, or optimize for a different set of criteria than the query optimizer in question. Now it becomes important for the database system to keep the user in the loop during query planning and allow her to express certain guidelines for a query that would allow/nudge the optimizer to produce the desired plan. Also, such a feature would enable a user to model what-if scenarios in order to manually generate a query plan with some desired characteristics and allow for explanatory analysis of the plan search space.

Existing Approaches

Most commercially available cost-based optimizers support some form by which a user could express plan constraints that enable these two use-cases. Common terms used by DB vendors to describe this set of features include “plan constraints”, “optimization guidelines” or “plan hints”.

There are two broad approaches as to how these plan constraints are stored and represented. Vendors such as Oracle & Vertica provide a grammar/syntax where the plan constraint may be expressed inline with the sql query. The parser then picks up the constraint, validates it, and then passes it on to the query optimizer.

Other systems such as DB2 and SqlServer, provide mechanisms to specify and store these optimization guidelines independent of a sql query, allow the database to map these stored plan constraints to an incoming query and then pass these constraints on to the optimizer as applicable. Let us call these two approaches – inline plan constraints vs independent plan constraints.

Presto Design Considerations

The primary characteristics of a plan constraint specification for Presto would include

- Specification of logical and physical join orders
- Optional specification of distribution methods for joins
- What-if planning based on cardinality, e.g. plan a query as if table t1 has 1000 rows instead of the value 100 stored in the meta data

Additional important characteristics that we should be open to (but may not address in the first pass) include

- Join algorithms (Java Presto today only supports hash joins, but Prestissimo will include merge joins as well)
- Access methods for relations (e.g. indexes)

In the rest of this section, we will describe through some examples the desired behavior in Presto. The syntax used here is merely illustrative and may not be the final version

Example 1:

Consider this query:

```
SELECT *
FROM t1, t2, t3, t4
WHERE a1=a2 AND a1=a3 AND a1=a4;
```

Suppose the optimizer returns the plan

```
(((t3 hash join t2) [P] hash join (t4 hash join t1) [P]) [P]
```

This is a bushy join where all joins are partitioned (indicated by “[P]” after each join - “[R] is used for replicated/broadcast.) Suppose instead the user wants a left-deep join plan in the order t4-t3-t2-t1 with t1 broadcast. The query can be submitted with an inline join constraint string to get the desired join plan:

```
SELECT /*! JOIN(((t4 t3) [P]) t2) [P] t1) [R] */ *
```

FROM t1, t2, t3, t4

WHERE a1=a2 AND a1=a3 AND a1=a4;

This would force the desired plan

(((t4 hash join t3) [P] hash join t2) [P] hash join t1) [R]

The inline constraint string here is a standard Presto comment with an additional ‘!’ added to the leading ‘/*’)

Since hash join is currently the only join method used for equijoins in Presto, the JOIN plan constraint is treated as both a logical and a physical join constraint. For the actual implementation to handle other join methods, JOIN could be recognized to specify the logical join order, while physical join constraints could be handled separately, e.g. MERGE_JOIN(t2 t1), if merge join is an option.

Example 2:

One possible extension is to consider multiple JOIN constraints as choices. I.e. they are logically ORed. This allows the optimizer to limit its search to a set of alternate plans and to choose the “best” among them by costing them relative to each other. In the following example the search space is now limited to two choices

SELECT /*! JOIN(((t4 t3) t2) t1) JOIN((t1 t4) (t3 t2)) */ *

FROM FROM t1, t2, t3, t4

WHERE a1=a2 AND a1=a3 AND a1=a4;

This allows the optimizer to choose between the two join orders. Only joins considered by the optimizer are effective in a constraint.

I.e. the specified constraints act as a filter on the optimizer join choices but cannot force join orders not considered by the optimizer. This is a “generate and filter”

approach, where if the optimizer search space does not include a desired alternative (e.g. a cartesian product) there is no way to force such a plan to be formed.

Example 3:

The CARD (cardinality) constraint allows what-if planning, specifically it tells the optimizer to plan the query as if a table had a given number of rows. This is useful in scenarios where we could model optimizer behavior of based on assumed (or overwritten) statistics:

```
SELECT /*! CARD(t1 1000000) */ *
FROM t1, t2
WHERE a1=a2;
```

The optimizer would then plan the query as if t1 had 1,000,000 rows. The CARD constraint can also be used to inject join result cardinality, e.g. the constraint

```
/*! CARD((t1 t2) 1000000 */
```

would tell the optimizer to plan the query as if the join t1-t2 produces 1,000,000 rows. This applies to any physical join t1-t2 or t2-t1.

Putting aside the actual interface for these plan constraints, one obvious approach is to add the constraints metadata to the Context carried around by iterative optimizer rules. The relevant rules then have access to the context/constraints and can use these to guide the processing. Depending on the use-cases we wind up supporting, this PlanConstraintContext would include a set of APIs that optimizer rules may use. For e.g., the constraints metadata may contain an API that checks whether a PlanNode satisfies any of the given join order constraints, in which case the optimizer rule ReorderJoins can choose this node as the winner in its processing.

Conclusion and Some Open Questions

Plan constraints are widely recognized as a safety net for query optimizers when they make bad plan choices. We believe that such a framework would enable rapid evolution of the Presto optimizer since existing users would be more comfortable with upgrades if there is way to fallback to a previous version of the optimizer/plan.

One open question is regarding the form of these plan constraints. As described previously the two common approaches are inline plan constraints and independent plan constraints. While inline plan constraints are relatively easier to implement in Presto, it is our opinion that the latter mechanism (independent plan constraints) is superior since it allows more latitude in how these constraints get used. There are many traditional use-cases where it is hard/impossible to alter a query string that is submitted to a query engine since these queries are generated by other systems (BI tools, AI bots, etc.). It is obviously useful to be able to map these queries to known or previously seen plan constraints that have benefited the execution of such queries. There is also a potential new use case that is exciting in this new world of generative AI. Imagine a learning optimizer (LO) that works in conjunction with Presto, and that can benefit from historic runs. Such an LO can then influence the existing Presto optimizer to generate better plans by producing a set of optimizer guidelines for different queries (classes of queries) and then match them to incoming queries using the independent plan constraints framework. However, the drawback to independent plan constraints is how and where to store them in the disaggregated world of datalakes since this may introduce an undesirable dependency.

Another open question is regarding identifying relations in the SQL string in order to express them in a constraint. In order for these plan constraints to be concise and easily described, we would like for them to be specified in one place, as opposed to Oracle where every query block can contain a plan constraint. Now consider a query that references the same table multiple times, or a query that includes views. It is a hard problem to uniquely identify these relations or join nodes, especially since some of the containing query blocks may have gone through transformations like decorrelations or query block merges. Ideally each table (or join source) needs a unique id. We could potentially build a mechanism/api that “explains” the query and annotates it with unique ids which are then specified in the constraint. Alternatively, we could refer to the “exposed names”/”correlation names” of the query (as described in SQL standard) in the constraint, and simply error if there are duplicates. We suspect that the latter approach would be good enough to cover most use cases and it would be trivial anyway to rewrite the query string with unique exposed names.