Search Space Improvements: Multi-Query Block Merge Optimizations

This document discusses state-of-the-art concerning query block merge transformations and Presto's current limitations in this area. The discussion also presents design considerations for lifting these limitations. This is one in a series of search space-related discussion documents that focus on expanding and better controlling the space of execution 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 resources consumed or time required to execute the 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.

A relational query written in the SQL relational query language might have multiple query blocks due to view references, nested table expressions, and subqueries. A query block is effectively a partition that needs to be optimized independently of the rest of the query. A SQL query may get fragmented into several query blocks depending on how the query is written or these boundaries may be introduced by the query optimizer as a result of some transformations. A query optimizer must be able to effectively merge multiple query blocks into equivalent single-block representations to open up the optimizer search space of join alternatives fully. Failure to do so can result in a highly suboptimal evaluation order. Transformation rules for merging query blocks require careful reasoning about duplicates. The work in [2] defines a set of six collaborating rules whose interplay guarantees the merge of table expressions for a large class of queries.

Example 1

```
CREATE VIEW olderparts AS
```

```
   (SELECT  DISTINCT l.partkey, l.suppkey
    FROM lineitem l, orders o
    WHERE l.orderkey = o.orderkey AND year(o.orderdate) < 1995 );


SELECT p.partkey, p.name, op.suppkey, p.retailprice
FROM part p, olderparts op
WHERE p.partkey = op.partkey AND p.retailprice > 1000;


-- rewritten query


SELECT DISTINCT p.partkey, p.name, l.suppkey, p.retailprice
FROM part p, lineitem l, orders o
WHERE p.partkey = l.partkey AND l.orderkey = o.orderkey AND
        year(o.orderdate) < 1995 AND p.retailprice > 1000;
```

That paper first describes a suite of transformation rules that guarantees the merge of views and nested-table expressions. Example 1 illustrates the considerations using queries referencing the TPCH schema. It first defines a view “olderparts” that returns parts available for order before 1995. The example includes a query that then uses the "olderparts" view to find parts supplied before 1995 and that retail for over 1000. A production-grade optimizer must be able to merge the view query block with the referencing query block in to avoid unnecessarily restricting execution plans to those that first join the “lineitem” and “orders” tables. The example shows an equivalent rewritten query with the merged view and referencing query blocks. The SELMERGE rule of [2] describes the conditions for performing this type of transformation. That rule requires careful consideration of duplicates. The view in the example uses DISTINCT to remove duplicates, so the SELMERGE transformation rule needs to ensure the resulting query does not introduce duplicate parts into the final result. It does this by carefully pulling up the DISTINCT to remove duplicates introduced into the final result. The correctness of the pull-up of the DISTINCT operation requires the subtle determination that the select list of the parent query contained a unique key; otherwise, the removal of too many duplicates would occur. This is true because the parent query selects keys of the “part” table and the “olderparts” view. A separate DISTUP rule [2] makes this determination by analyzing the primary key constraints defined by the TPCH schema.

Example 2

```
WITH  largeorders AS
  (SELECT DISTINCT orderkey
   FROM lineitem
   WHERE quantity > 10 )
SELECT *
FROM customer c
WHERE NOT EXISTS (SELECT 1
                  FROM largeorders lo, orders o
                  WHERE lo.orderkey = o.orderkey AND c.custkey = o.custkey)
— rewritten query

SELECT *
FROM customer c
WHERE NOT EXISTS (SELECT 1
                  FROM lineitem l, orders o
                  WHERE l.orderkey = o.orderkey AND
                        c.custkey = o.custkey AND l.quantity > 10)
```

Example 2 further illustrates the subtleties of duplicate handling. It defines a view “largeorders” that returns orders having an item of quantity exceeding 10. That query uses that view to find customers without large orders. Although that view uses a DISTINCT operation to eliminate duplicate orders, it can be merged directly into the subquery without pulling up the DISTINCT as in the prior example. This is because existential and universally quantified subqueries are insensitive to duplicates. The example shows the rewritten query with the view merged into the subquery.

Example 3

```
SELECT *
FROM orders
WHERE orderkey IN (SELECT orderkey
                   FROM lineitem
                   WHERE quantity > 10) ;

 — rewritten query after existential to join transformation

WITH tx as (SELECT orderkey
            FROM lineitem
            WHERE quantity > 10)
SELECT DISTINCT o.*
FROM orders o, tx
WHERE o.orderkey = tx.orderkey;

 -- rewritten query after merge of table expression

  SELECT DISTINCT o.*
  FROM orders o, lineitem l
  WHERE o.orderkey = l.orderkey AND l.quantity > 10 ;
```

In addition to merging views and table expressions, an optimizer must also be capable of merging existential subqueries to fully open up its search space of join alternatives. Example 3 illustrates the problem and transformation rule considerations. The query contains a subquery that returns orders having an item with an order quantity of at least 10. The prior art in [2] defines an EtoF rule that transforms conjunctive existential subqueries into non-existential table expressions. The second query in the example illustrates the result of the transformation. Subsequent application of the suite of SELMERG rules would merge the subquery into the parent query block.

The Presto optimizer cannot yet merge the views in example 1 or example 2; consequently, it cannot consider execution plans that join the view tables with tables in the referencing query blocks. Further, Presto defaults to transforming the existential subquery of example 3 directly into a semi-join. This limitation effectively forces join orders with the “lineitem” table as the inner table. The work in [3] is a step in the right direction as it will ultimately merge the subquery as per the final query in Example 2; however, that work cannot handle more complex cases, such as when the parent query block has joins. The following section will discuss various considerations for bringing the state-of-the-art into Presto to improve the optimizer view, table expression, and subquery merge capabilities.

Presto Design Considerations

The examples in the previous section demonstrate opportunities for Presto to open up its join search space by incorporating state-of-the-art multi-block query block transformation techniques. The work in [2] describes a suite of transformation rules guaranteeing the merge of views, table expressions, and subqueries for a large class of queries. Those rules define transformations of a relational calculus-based internal query representation called QGM. An advantage of the QGM model is that it maintains a “select box” construct analogous to an SQL select-project-join query block. The select box body contains a join graph with table references as nodes and predicates as edges. The "select box" represents a single join search space to the join planner. The result of applying the suite of transformation rules is a QGM with fewer select boxes with larger join graphs that provide more degrees of freedom to the join planner.

The Presto optimizer does not have an explicit construct representing a select-project-join query block. The closest analog to that is a tree of contiguous join nodes that collapses into a single multi-join node representing a single search space to the Presto joinplanner. Any non-join nodes interleaved with join nodes effectively create a query block boundary that limits the potential join search space. A key design challenge is to adapt the rules in [2] to a relational algebra-based query representation lacking an explicit query block construct as is used by Presto.

Another option may be to explore implementing these inferred rules as a set of standard iterative optimizers in Presto. This may have some challenges since the Presto optimizer is actually a large set of iterative optimizers that work in isolation. Integrating query block merges into this scheme and ensuring its efficacy will require careful consideration and exhaustive testing.

Summary

A mature query optimizer must include advanced capabilities for merging multiple query blocks into equivalent single-block representations to open up the entire possible join search space. Transformation rules for query block merges require sophisticated reasoning about duplicates. The document discussed the state-of-the-art with for such transformations [2]. It also discussed some of Presto's current limitations in this regard. The discussion also presented design considerations for augmenting Presto to lift these limitations. These included adapting the suite of transformation rules introduced in [2] for a relational algebra-based internal query representation and, in general, making Presto more query block cognizant by pushing up and pushing down plan nodes that unnecessarily introduce query block boundaries that limit the reorderability of joins. Project nodes are notorious for littering Presto plans and restricting the join search space by introducing query block boundaries. The ultimate design should give them careful consideration.

References

[1] Chaudhuri S. An overview of query optimization in relational systems. In: Proceedings of 17th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. 1998. p. 34–43.

[2] Pirahesh H., Hellerstein J.M., Hasan W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In Proc. of ACM SIGMOD 1992.

[3] Transformations of Existential Subqueries using Early-out Joins. Presto Issue #17927

[4] Constraint Support and Optimizations. Presto Issue #16413