Search Space Improvements: Comprehensive Join Enumeration

This document discusses state-of-the-art concerning the comprehensive enumeration of inner, outer, anti, and other joins types 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.

One of the critical tasks of an optimizer is to decide the order in which to join tables referenced in a query. A sub-optimal join sequence can execute orders of magnitude less efficiently than an optimal one. The optimizer generates the search space of alternatives by applying commutative and associative rules to produce equivalent join sequences. The join type dictates the applicable reordering rules. Inner joins are fully commutative and associative; hence, all join orders are valid. Outer joins and anti-joins are asymmetric operators with limited commutative and associative properties; consequently, not all sequences of joins are valid when one or both outer joins and anti-joins are present.

Example 1 from [3] illustrates a case where the associative properties of inner and outer join operations do not hold. It shows that inner join operations cannot be transformed associatively with outer joins. Failure of an optimizer to understand invalid reordering properties would lead to an execution plan that produces an incorrect result.

R

tid	a
r1	1
r2	3
r3	5

S

tid	a	b
s1	1	1
s2	1	2
s3	3	3
s4	3	4

T

tid	b
t1	1

```
SELECT  R.tid, S,tid. T.tid
FROM R LEFT JOIN  (S INNER JOIN T ON S.b=T.b) ON R.a = S.a
```

tid	tid	tid
r1	s1	s2
r2	-	-
r3	-	-

```
SELECT  R.tid, S,tid. T.tid
FROM (R LEFT JOIN ON R.a = S.a ) INNER JOIN T ON S.b=T.b
```

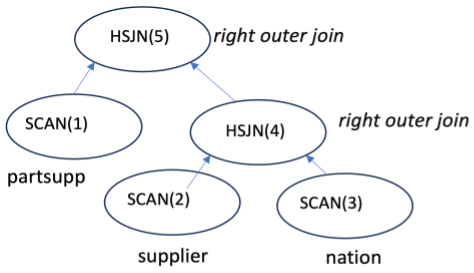
tid	tid	tid
r1	s1	s2

Example 1: invalid inner and outer join reordering

Example 2 illustrates a case where associative and commutative transformations of outer joins are applied to reach an optimal execution plan. Association transformation rules are first applied to form a small composite join where the single row *nation* table expression result is joined with the *supplier* table using a left join operation. After changing the association of the joins, the composite table joins with the larger *partsupp* table via another left join operation. The optimal join order is subsequently reached by applying transformation rules to turn the left joins into right joins with commuted operands. The transformed execution plan now has the optimal-sized build tables for the two hash join operations. Presto executes the sub-optimal sequence of hash joins specified by the original query as it does not attempt to reorder joins when outer joins are involved.

```
-- Original query
SELECT *
FROM (SELECT * FROM nation WHERE name = 'ARGENTINA') a LEFT JOIN
        (supplier s LEFT JOIN partsupp ps ON s.suppkey = ps.suppkey) ON a.nationkey = s.nationkey;

-- Optimal join type, ordering, and execution strategy
SELECT *
FROM partsupp ps RIGHT JOIN
          (supplier s RIGHT JOIN (SELECT * FROM nation WHERE name = 'ARGENTINA') a ON a.nationkey = s.nationkey)
      ON s.suppkey = ps.suppkey ;
```



Example 2: optimal outer join strategy after applying associative and commutative rules

A state-of-the-art optimizer must effectively enumerate valid join sequences involving all types of join operators, as queries with non-inner joins occur in many real-world applications.

Prior Art Overview

The problem of reordering complex queries involving various join types is extensively studied [2][3][4]. The outer join optimization approaches described in the referenced prior art espouse techniques that follow these general steps:

1. A predicate optimization and join simplification step that ensures early evaluation of predicates, and the transformation of join operations preserving non-matches by injecting null rows into simpler joins in cases where null-intolerant predicates[1] are later applied.
2. A conflict detection step that computes for each join operator in a query expression, conflict rules that lead the join enumerator to valid join sequences when unviolated. These conflict rules derive from analysis of the producer-consumer relationships established in the initial query operator tree, in conjunction with known commutative and associative join transformation rules. As will be discussed, the art derives and encodes these rules in various ways.
3. A join enumeration step that extends the conventional join enumeration process with the capability to evaluate the detected conflict rules so that the join enumerator only generates valid join sequences.

For a detailed summary of prior art, please see <link to prior art doc>

On the correct and complete enumeration of the core search space

While the previously described approaches generate valid join sequences concerning known commutative and associative properties of the specific join operation types under their consideration, they do not produce the complete space of such join operation sequences. More recent work in [4] presents techniques for enabling a dynamic programming-based join enumerator to exhaustively generate the core search space of valid join sequences.

They first define the core search space via application of commutative and associative algebraic rules enabling generation of all valid alternatives to a given initial join operator tree. They further discuss extending a conventional dynamic programming-based join enumerator with the capability to generate this core search space. Like the previously described approaches, a conflict detection step first analyzes the query operator tree to derive conflict rules that guide the join enumeration process to valid join sequences.

---

[1] The rules are applied to a canonicalized operator tree where all one-sided join operations are written to match the direction implemented by the rules (e.g. left)

```
DP+
1.  Input: a set of relations R = {R0,...,Rn−1}
2.         a set of operators O with associated conflict rules
    Output: an optimal bushy operator tree
3.  For all Ri ∈R
4.    BestPlan({Ri}) ← Ri
5.    for 1 ≤i< 2n −1 ascending
6.      S ← {Rj ∈ R|(⌊i/2j⌋mod2) = 1}
7.      if |S|=1 continue
8.      for all S1 ⊂ S, S1≠∅
9.        S2 ←S\S1
10.       if (o = Applicable (O , S1, S2)) //if no conflicts based on associative rules Tables 2 and 3
11.           SavePlans( BestPlan(S1) ◦ BestPlan(S2))
12.           if ( Comm(◦) )  //based on Table 1
13.             SavePlans (BestPlan(S2) ◦ BestPlan(S1))
14.           else
15.             SavePlans (Cso (o, BestPlan(S1),  BestPlan(S2) ))//commute and substitute transformation Table 5
16. return BestPlan(R)
```

Figure 1: pseudocode for DP+

The pseudo-code above describes a mechanism for extending a dynamic programming-based join enumerator. The *DP*+ method proceeds as usual by generating join plans of progressively larger subsets S of the input sources. If there is a valid operator *o* to join the pair (S1⊆ S, S2 = S \ S1) it is returned by the *Applicable* method (line 10), and then used to join the best plans for S1 and S2 as recalled from the dynamic-programming table (line 11).

If *o* is also commutative (line 12), DP+ also generates a commutative join plan using *o* (lines 13) If *o* is not commutative, a join plan with commuted operands is formed based on the *cso (commute and substitute)* property (line 15).

The *Applicable* method generates only algebraically valid join sequences in the core search space of a query.

Detailed descriptions and definitions of the methods are described in [4]

Presto Design Considerations

This section discusses the current state of the Presto optimizer concerning join enumeration and how these could be extended to incorporate state-of-the-art join reordering techniques

Presto Join Enumeration Background

Presto supports inner join, cross join, left outer join, right outer join, and full outer join. Direct runtime implementations exist for each of these join operators. In addition, transformation rules can introduce left semi-joins and left anti-joins.

The Presto optimizer capably performs outer join simplification, as well as push down of predicates into outer join operations [2]. The join enumerator, however, only enumerates inner join orderings. It looks for trees of contiguous such join nodes and flattens them into a multi-join node, which is the *enumeration context* input to the join enumerator. If another type of join node is interleaved with these nodes, the enumeration context becomes bifurcated. In this case, each separate partition of the bifurcated join tree represents a different enumeration context. A single enumeration context is currently limited to ten sources.

The join enumerator's search space for a given enumeration context includes bushy trees without composite inner size restriction [6]. There is no explicit notion of join predicate eligibility lists [3]. A predicate is eligible for joining source partition pair ($S1 \subseteq S$, $S2 = S \setminus S1$) if it references output variables of both S1 and S2. The avoidance of cartesian products occurs in a back-handed way. They are enumerated but assigned infinite costs. Presumably, this is to avoid failing on disconnected join contexts.

The search space for a given enumeration context is enumerated using a naïve memoization algorithm similar to *MemoizationBasic* in [7]. The optimal subplan for each set of sources S is the one that dominates all other alternatives based solely on cost. Generation of commutative join plans for source pairs ($S1 \subseteq S$, $S2 = S \setminus S1$) occurs indirectly via enumeration of the entire power set of S, versus a more explicit approach that enumerates half of the power set and forms joins for both (S1, S2) and (S2, S1) like in DP+ (Figure 1) above

Extending Presto with Comprehensive Join Enumeration Capabilities

The general design considerations are as follows -

- Generation of an enumeration context consisting of plan nodes of all join types, not just inner join nodes as today. During this step, a new join node explicitly represents a left or right anti-join operation. It is transformed back to a one-sided outer join and filter combination after join enumeration[2].
- Canonicalization followed by conflict analysis. Canonicalization of the operator tree transforms right-sided outer joins, semi-joins, and anti-joins into their left-sided counterparts. Conflict analysis determines conflict rules for each join operator by applying the associative patterns described in [4]

2

- Extensions to the memoization-based enumeration process to enable the generation of only valid join sequences. These extensions are correlative to the DP+ of Figure 1.

In addition to the general design considerations just outlined, detailed design considerations are as follows.

- Explicitly handle cartesian products by enumerating only beneficial ones [6]. Specifically, add a test to *Applicable* that returns only cartesian product join operators of benefit. To avoid failure on enumeration contexts with disconnected components, perform a join graph analysis during the canonicalization phase and connect those components explicitly with a tautology join predicate later removed. To enable the generation of beneficial cartesian products, something analogous to companion sets [3] is needed to determine join operator validity.
- Explicitly model and use join predicate extended eligibility lists or EELs [3]. A join predicate is only eligible for combining (S1⊆ S, S2 = S \ S1) if its EEL ⊆ S. All join predicate conjuncts share the same EEL. Explicit use of EELs in conjunction with conflict rules allows conflict rule simplification optimizations [4] while also optimizing the enumeration process via cartesian product deferral.
- The commutative and associative properties of Tables 2, 3, and 4 enable the generation of valid join orders involving semi-joins; however, it is better to eliminate them before the enumeration process by transforming existential subqueries to an inner join and distinct combination and to convert back to semi-joins post-enumeration using early-out join techniques [8] as inner joins are freely reorderable whereas semi-joins are not.

Summary

A critical task in query optimization is to find the optimal evaluation order for the join operations in a query. Inner joins are freely reorderable; hence, all evaluation orders are valid. Other join operations, such as outer joins and anti-joins, have limited commutative and associative properties; consequently, not all join sequences are valid when these operators are present. A state-of-the-art optimizer must effectively enumerate valid join sequences involving all types of join operators, as queries with non-inner joins occur in many real-world applications. Moreover, transformation rules often introduce such join operations. The problem of reordering joins in complex queries involving inner joins, outer joins, anti-joins, and other join types is extensively studied [2][3][4], with [4] being the most comprehensive. This document provides an overview of the state-of-the-art in enumerating complex joins of various types and a blueprint for evolving the Presto optimizer with those capabilities.

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] Cesar A. Galindo-Legaria, et al. Outer join simplification and reordering for query optimization. Transactions on Database Systems, 22(1), 1997

[3] Jun Rao, B. Lindsay, G. Lohman, H. Pirahesh, David E. Simmen Using eels, a practical approach to outer join and antijoin reordering. In Proceedings of the IEEE ICDE Conference, 2001.

[4] Moerkotte, Guido et al. "On the correct and complete enumeration of the core search space." ACM SIGMOD Conference (2013).

[5] Orthogonal optimization of subqueries and aggregation C. Galindo-Legaria, Milind Joshi. Published in ACM SIGMOD Conference 1 May 2001

[6] Measuring the Complexity of Join Enumeration in Query Optimization. K Ono, GM Lohman - VLDB, 1990

[7] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In ICDE, pages 864–875, 2011.

[8] Transformations of Existential Subqueries using Early-out Joins.