Bottom-up Join Enumeration in a Top-down Optimizer

Bhuvnesh Chaudhary, Hans Zeller, Sambitesh Dash, Venkatesh Raghavan

MAPBU, VMWare
Palo Alto CA, USA
{bchaudhary, hzeller, sambiteshd, vraghavan}@vmware.com

ABSTRACT
Greenplum Database is a massively parallel processing (MPP) analytics database that adopts a shared-nothing architecture with multiple cooperating processors. A query submitted to the Greenplum master is optimized by the Orca query optimizer [7] which is based on state-of-the-art top-down query optimization techniques. Joins are commonly used SQL operations. When querying large amounts of data, it is crucial to generate a near-optimal join to ensure that the user query does indeed complete.

In this paper, we present the join enumeration algorithm employed in Orca. Our proposed approach seamlessly handles workloads consisting of a mixture of inner, left and right joins. Optimization effort is directly proportional to the complexity of the join query. Our technique employs an exhaustive approach to generate an optimal join order when the number of join participants is less than 10. For larger join queries, we gracefully transition into a greedy based approach to reduce optimization time. In this work, we have developed a cost model that incorporates dimensions specific to a parallel database setup such as data distribution of the join participants, and dynamic partition elimination opportunities. Lastly, to ensure that we do not re-trace the same search space over and over again we have built data structures that capture paths traversed and work accomplished by the algorithm. We demonstrate the benefits of our proposed technique by running workloads on established database benchmarks as well as customer datasets.

Categories and Subject Descriptors
• Information systems → Query optimization; Relational parallel and distributed DBMSs;

General Terms
Algorithms, Performance.

Keywords
Join Enumeration, Left Outer Joins.

1. INTRODUCTION
Orca is based on the design of the Cascades optimizer [4], using a top-down approach where the optimizer search space is enumerated by applying a set of transformation and implementation rules.

The great advantage of this rule-based approach is that Orca can handle transformations involving many types of operators in addition to joins. Examples are union, group by, common table expressions, sequence functions, and user-defined functions. A purely rule-based approach to join enumeration could be implemented with two transformation rules, a commutativity rule and a left-shift rule (a variant of the associativity rule):

\[ A \Join B \Rightarrow B \Join A \] \quad \text{commutativity rule}

\[ (A \Join B) \Join C \Rightarrow (A \Join C) \Join B \] \quad \text{left-shift rule}

This approach would be extremely slow, however, mainly because of the overhead involved with invoking each rule. To speed up join enumeration, Orca employs a single transformation rule that takes an n-ary join (a single operator joining n tables) as an input and generates multiple trees of binary joins, using a bottom-up join enumeration algorithm. These alternatives are then transformed further by implementation rules and eventually optimized, using a cost model. Our bottom-up enumeration algorithm follows a dynamic programming approach, using a simplified cost model that we will call proxy cost model in this paper.

In the following we will discuss four improvements of the Orca model. In a first step, we increased join enumeration performance by avoiding multiple visits of the same point in the solution space. Second, we altered the enumeration process to support left outer joins (LOJs), to avoid costly transformations and duplicate work that were previously required for LOJs. Third, we created a gradual transition between exhaustive join enumeration to a greedy approach, starting with joins above a certain size (10 tables by default). This replaces the current abrupt change from an exhaustive to a greedy algorithm. Fourth, we put an infrastructure in place where we can enhance the proxy cost model used in join enumeration. Since join enumeration happens in the transformation phase, Orca does not have access to the full physical properties of the operators and therefore uses a proxy cost model, which is currently based on the total data flow in the join tree. We are planning to add information on distribution keys and partition selectors where possible. Information about potential index access paths is a further optimization that could be applied.

Our expectation is that these four improvements will allow us to enumerate more potential join orders in shorter time, with better differentiation and quality of the chosen results. This should lead to reduced total query time (planning time plus execution time), especially for larger queries including outer joins. Total time should ramp up smoothly with increasing complexity of join queries.

The rest of the paper is organized as follows: Section 2 gives an overview of the current join optimization algorithm in ORCA and shows the size of the optimization space. Section 3 talks about the improvements we have recently added and those we are planning...
to add shortly. Section 4 has experimental results, and section 5 presents a summary.

2. JOIN OPTIMIZATION IN ORCA

2.1 Search Space

One complex problem in query optimization is to control the search space when optimizing joins. For a query that joins \( n \) tables, \( T_1, \ldots, T_n \), there are \( n! \) possible join orders, assuming we use a linear tree of binary join operators. When we allow bushy trees [3] (trees where a join operator can have two children that are other joins), this number increases to \((2n-2)! \) / \((n-1)!\) [6]. Enumerating such a space is clearly infeasible, even for medium-sized queries. For example, a 10-way join has roughly 17.6 billion possible bushy join trees. For a 20-way join, there are \( 4.3 \times 10^{27} \) bushy trees.

Virtually all query optimizers employ the technique of dynamic programming to reduce the search space dramatically. Dynamic programming relies on the principle of optimality, which says that the solution to a problem consists of the optimal combination of optimal solutions to smaller subproblems. In other words, it applies the strategy of “divide and conquer” to the join enumeration problem. For join enumeration, the subproblems are the possible joins of a subset of the tables involved. Given a join of \( n \) tables, this results in \( 2^n - 1 \) such subproblems – the number of non-empty subsets of a set of \( n \) tables. We will call these subproblems groups, which is the term used in the Cascades optimization model for groups of semantically equivalent queries.

To make dynamic programming work, however, we have to consider multiple ways to form those subsets from optimal solutions of smaller subsets. This leads to multiple expressions within groups. For example, the set of equivalent joins of tables \( T_1, T_2 \), and \( T_3 \) forms a group, \( \{T_1, T_2, T_3\} \). Within that group, there can be multiple expressions, representing different ways to perform this join, such as \( \{T_1, T_2\} \bowtie \{T_3\} \) and \( \{T_1\} \bowtie \{T_2, T_3\} \). Note that the join expressions in groups refer to other groups, to reflect the principle of optimality. In this example, the second expression represents a join of the optimal way to read table \( T_1 \) with the optimal join of tables \( T_2 \) and \( T_3 \).

Groups and expressions used in dynamic programming reduce the search space dramatically, as do several possible heuristics, like excluding bushy tree and unnecessary cross products. Table 1 shows the number of expressions for dynamic programming, with and without using heuristics [5][6].

<table>
<thead>
<tr>
<th>Type of query and heuristic applied</th>
<th>Space complexity (number of expressions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy algorithm (Orca uses this strategy for large queries)</td>
<td>( o(n) )</td>
</tr>
<tr>
<td>Linear query graph [5], no bushy trees, no unnecessary cross products</td>
<td>( o(n^2) )</td>
</tr>
<tr>
<td>Consider cross products, no bushy trees</td>
<td>( o(2^n) )</td>
</tr>
<tr>
<td>Bushy trees, consider cross products (Orca uses this strategy for small queries)</td>
<td>( o(3^n) )</td>
</tr>
</tbody>
</table>

Table 1: Space complexity of some join enumeration strategies

Since Orca does an exhaustive enumeration, our goal is to find the optimal solution (according to our proxy cost model) with \( o(3^n) \) expressions. Since even that is infeasible for large joins, we employ a hybrid approach that transitions to a greedy search algorithm once the query exceeds a certain size. Therefore, large queries use only \( o(n) \) expressions but will usually require \( o(n^2) \) time, since at every of the \( n \) levels we have to evaluate on average \( n/2 \) choices to find the best table to join next.

2.2 N-ary join expansion

Orca goes through multiple phases when generating a plan for a query:

2.2.1 Preprocessing

The query tree as generated by the parser contains two-way joins that are simplified, if possible, then collapsed into n-ary joins. We changed this phase to collapse inner as well as outer joins (excluding full joins).

2.2.2 Exploration

During this phase, transformation rules are transforming logical operators into semantically equivalent, different operators. For n-ary joins, we use the join enumeration algorithm described in this paper to generate multiple alternative trees that again consist of binary join operators. We select the top \( k \) such trees generated by our enumeration, based on the proxy cost model.

2.2.3 Implementation

In this next phase, logical operators such as joins are transformed into physical operators like hash joins or nested loop joins.

2.2.4 Optimization

Finally, this phase determines physical properties that are required to produce a correct and optimal execution plan. Example properties are sort order and distribution key in Greenplum’s multi-node environment.

This paper focuses primarily on the exploration phase, and, specifically, on join order enumeration in the exploration phase.

3. IMPROVEMENTS TO ORCA JOIN ENUMERATION

3.1 Truly bottom-up enumeration

Our performance measurements have shown that the existing, simple and elegant recursive method to enumerate all groups and expressions doesn’t scale well. The reason is that it requires too many hash table lookups to avoid enumerating duplicate solutions. Our first optimization was therefore to convert the recursion into a series of nested loops that avoid duplicate checking:

For each level \( i, i = 2, 3, \ldots, n \):

- Build all linear trees by joining all possible \( i-1 \)-way joins on the left with a single table on the right

  For each \( j, 2 \leq j \leq i/2 \):
  - Build all bushy joins with a \( j \)-way join on the right and an \( i-j \)-way join on the left

The most expensive part of this algorithm is to calculate the estimated number of rows for each join, needed to compute its cost. We retain use of a hash table to ensure we make this calculation only once for each of the \( 2^{i-1} \) groups.

Note that we rely on a separate join commutativity rule to avoid the need to enumerate both forms \( A \bowtie B \) and \( B \bowtie A \) of a join (some of the details are omitted in the algorithm above).
3.2 Handling non-inner joins
To handle the non-inner joins, it was necessary to be able to track the non-inner join ON predicates in our n-ary join operator. We introduced a special join predicate operator, attached to the end of our n-ary join operator. The join predicate operator is a list of ON conditions. The 0th child of the list points to all the inner-join ON predicates. Subsequent children of the list point to ON predicates for each of the non-inner joins. We introduced a concept of valid join order to join enumeration by implementing a subset of the rules in [2] for left outer joins. In a nutshell we have three new restrictions – 1) If the original tree had n non-inner joins, then the result will have n non-inner joins too, with the same right children. 2) The ON predicates of the original non-inner joins will be preserved in the result. 3) Any column references (excluding outer references) used in the ON predicate of a non-inner join will be produced by its left and right children. The rest of the join enumeration algorithm remains the same. Currently we handle only Left Outer Joins. But in future we plan to support semi-joins and anti-semi joins too in our join enumeration algorithm.

3.3 Controlling the search space
In section 2.1 we discussed the size of the search space. In summary, we are dealing with 2^n-1 groups and o(3^n) expressions for exhaustive enumeration. Our task is to define an algorithm that starts out with this behavior for smaller queries (e.g. up to 10-way joins) and then transitions to linear growth for the number of expressions. This could be done by limiting the number of expressions at every level i (groups of i-way joins). Alternatively, we could limit the number of groups at level i, in the assumption that each group contains only a limited number of expressions. This latter approach is the one we took, in part because it is easier to implement and in part because it saves further reduces the amount of space required by minimizing the number of groups.

For an n-way join, we have up to k = \binom{n}{i} groups at level i, the number of subsets of size j for a set of size n. Our approach to limit join enumeration is simple, we limit the number of groups to k = \binom{n}{10} for levels i, 2 <= i <= 10. For levels i > 10, the number of allowed groups is 1, leading to a linear growth in groups as n grows. There is no limit for level 1; we always need n groups at the lowest level 1 to represent the tables participating in the join. Since each group also has a limited number of expressions (see section 3.4), we effectively limit the number of expressions as well.

The implementation uses a heap data structure to compute the top k groups in linear time.

Figure 1 shows the number of groups for 4, 6, 8 and 10-way joins as solid lines (they are really discrete values, not the smooth lines shown). The dashed lines show how many groups larger joins would have, were they not limited to those of a 10-way join (green curve). The limit of 10 is the default. If so desired, a Greenplum DB user can choose a different value, as we did in section 4.

3.4 Improving the proxy cost model
The principle of optimality used in dynamic programming typically relies on properties to find the optimal solution. For example, when joining two subsets, the order of rows produced by the sub-solutions can be significant. The current dynamic programming algorithm and its proxy cost model do not yet consider all the relevant properties, and we plan to enhance them to consider several new properties.

The first new property to consider is the distribution key. To perform a parallel join, the data must either be distributed on equi-join columns from both sides, or one side of the join must be replicated. If the data is not already distributed or replicated, a motion operator is required to create the correct distribution or replication. Considering the distribution key as a property helps to account for the cost of these motions in the proxy cost model.

Another property is related to partition selection [1]. Orca has the capability to create partition filters from the smaller side of a hash join, to reduce the amount of data read in the larger, partitioned table.

```sql
select *
from fact join dim
on fact.order_date_id = dim.date_id
where dim.year = 2020 and dim.month = 2;
```

If the fact table is partitioned on order_date_id, we can build a list of partition ids relevant for this join from the dim table, after applying the predicate. We might need to read only one partition (for February 2020), instead of the dozens of partitions that might exist in the table.

Adding properties that recognize this situation will help creating plans with good partition selectors.

A third potential property has to do with indexes. This is similar to partition selection, in that an index nested loop join can reduce the amount of data to read from a table.

One issue with these properties is that they are physical characteristics of the tables involved. Join enumeration in Orca happens during the exploration phase (see section 2.2.2), where such physical properties are not usually involved. Therefore, using physical properties during logical exploration will involve a compromise, synthesizing limited information about physical properties where possible, and tolerating somewhat imprecise or missing information.

4. EXPERIMENTAL RESULTS
To test our improvements described in sections 3.1, 3.2 and 3.3, we used our existing test suites, including realistic query workloads from customers. In addition, we designed two tests to perform measurements. The first test simply increases the number of inner joins in a simple query and measures the total time we spend planning the query, including work done before and after join enumeration, but not including the time it takes to execute the query. We don’t expect any difference in execution times for these
simple types of queries, as both the existing and the new join enumeration should generate a near-optimal plan.

Figure 2 shows the result. We can see that the existing algorithm grows very rapidly, before it changes abruptly to a greedy approach, in this case for 13-way joins and above. Note that by default, Orca will make this switch for 11-way joins and above, and our results show that this default is a good choice. We chose the higher limit to illustrate how the two algorithms scale. Our new algorithm shows the gradual transition to a near-linear model (linear space, \(n^2\) time), which should lead to a better trade-off between planning time and plan quality, as the complexity of the query increases.

![Figure 2: Planning times for n-way inner joins](image)

At this point, we see mostly an improvement in planning time. Once we have implemented our last algorithm, described in section 0, we hope to show improved query plan quality.

5. SUMMARY

We presented the join enumeration algorithm of the Orca optimizer, used in Greenplum DB. A brief analysis of the search space shows that efficient algorithms as well as heuristics to limit the search space are required to solve this problem. We then presented four improvements of the current algorithm: More efficient enumeration, handling outer joins more efficiently and with better results, a gradual transition from exhaustive to greedy search, and, finally, a plan to improve plan quality by considering some physical properties during the enumeration phase. We presented experimental results, showing improved performance and planning times that are commensurate with the complexity of the query.

6. ACKNOWLEDGMENTS

We would like to thank Abhijit Subramanya, Ashuka Xue and Chris Hajas for their help reviewing this paper.

7. REFERENCES


