Optimizing Join Enumeration in Transformation-based Query Optimizers

Anil Shanbhag
IIT Bombay
anil@cse.iitb.ac.in

S. Sudarshan
IIT Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

Query optimizers built on the Volcano/Cascades framework, which is based on transformation rules, are used in many commercial databases. Transformation rulesets proposed earlier for join order enumeration in such a framework either allow enumeration of joins with cross-products (which can significantly increase the cost of optimization), or generate a large number of duplicate derivations. In this paper we propose two new rulesets for generating cross-product free trees. One of the rulesets is a minor extension of a simple but inefficient ruleset, which we prove is complete (we also show that a naive extension of an efficient ruleset leads to incompleteness). We then propose an efficient new ruleset, which is based on techniques proposed recently for top-down join order enumeration, but unlike earlier work it is cleanly integrated into the Volcano/Cascades framework, and can be used in conjunction with other transformation rules. We show that our ruleset is complete (i.e., it generates the entire search space without cross products) while avoiding inefficiency due to duplicate derivations. We have implemented this ruleset in the PyroJ Optimizer (an implementation of the Volcano optimizer framework) and show that it significantly outperforms the alternatives, in some cases by up to two orders of magnitude, in terms of time taken.

1. INTRODUCTION

Query optimization has been studied for many years, and in particular the subproblem of finding an optimal join order has a very long history. Bottom-up enumeration of join orders using dynamic programming is a well known technique for join order optimization. However, top-down join enumeration has the benefit of allowing cost based pruning during join order enumeration, and has been proposed as an alternative in, e.g., [2]. In addition, transformation-based optimizers such as those based on the Volcano/Cascades framework [11, 10] are also inherently top-down.

It is well known that plans with cross-products are likely to be inefficient, and for many join-graph topologies, the number of plans with cross-products is vastly more than the number of cross-product free plans. A well-accepted heuristic used in optimizers is to consider all bushy join trees, but exclude cross products from the search, presuming that all considered queries span a connected query graph [14].

Two approaches have done well at exploring this restricted search space and finding the optimal plan: dynamic programming based bottom-up join enumeration, and memoization based top-down join enumeration. Recent work in the area of efficiently exploring the space of cross-product free join orders includes an efficient dynamic programming based bottom-up algorithm DPccp by Moerkotte and Neumann [13] and top-down enumeration strategy TDMinCutConservative by Fender et al. [6].

All the above work focusses only on join orders, whereas real queries involve many other operators such as aggregations, outer-joins and so on. While join-order enumeration techniques have been extended to support aggregation [1], or outer-joins (e.g. [8, 9, 5, 4, 12]), these extensions require significant algorithmic changes, making extensibility difficult.

The Volcano/Cascades framework for optimization using transformation rules, in contrast, is inherently extensible, and has been implemented in several widely-used commercial database systems such as Microsoft SQL Server. Not only can this framework easily handle aggregations and outerjoins using transformation rules, but it can also handle optimization of nested subqueries in a clean fashion [7].

Flexibility often comes at the cost of efficiency if the underlying implementation is not done in the best possible manner. As shown by Pellenkoft et al. [15], using associativity/commutativity rules to enumerate the space of join orders results in an excessive number of duplicates, increasing the worst case time complexity from $O(3^n)$ to $O(4^n)$ for bushy join orders with n relations. An alternative ruleset for efficiently enumerating bushy join orders was proposed in [15]; however, that approach enumerates all join trees, including those with cross products.

Thus, an open question is, how can we efficiently explore the space of cross-product free bushy join trees in the context of a transformation-based optimizer.

We address the above problem in this paper. The contributions of this paper are as follows.

- We analyze existing rulesets used in transformation-based optimizers [15] for their ability to explore the cross-product free search space efficiently.
  - We first consider a simple ruleset consisting of

Proceedings of the VLDB Endowment, Vol. 7, No. 12

commutativity and left-associativity with cross-product suppression, i.e., a modification to not apply a transformation if it introduces a cross-product. We show that this ruleset is *complete*, i.e., any cross-product free tree can be transformed to any other cross-product free tree by a sequence of transformations from the ruleset. On the other hand, this ruleset generates duplicate derivations, which results in inferior performance, as our experimental results show.

– We show that the RS-B2 ruleset proposed by [15], modified to include cross-product suppression, fails to explore the entire search space.

- We present an extension to Volcano framework which, instead of considering one or two adjacent join operators, performs transformations on maximal subtrees consisting of inner join operators. This extension allows us to use the partitioning strategies proposed, for example, in [6], in the context of a transformation-based optimizer, and thereby generate only cross-product free trees.

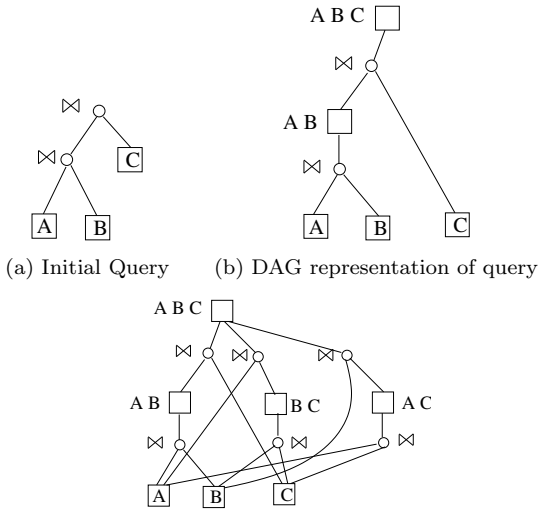
  In particular, we show how to get the benefit of efficient exploration of the space of cross-product free join trees, while retaining the extensibility of the Volcano framework; our approach allows other transformation rules to be easily introduced without any changes in the optimization algorithm.

  A key issue that needs to be addressed is the presence of multiple subtrees of joins under a single logical equivalence node. For example, a subexpression $({}_A\gamma_{sum(B)}(r \bowtie s)) \bowtie t$ has a join tree consisting of a single join, with inputs t and $({}_A\gamma_{sum(B)}(r \bowtie s))$, where the aggregation operator ${}_A\gamma_{sum(B)}$ computes the sum of B, grouped by A. However, the application of a transformation rule which pushes the aggregation operator γ below the join could result in the expression $({}_A\gamma_{sum(B)}(r) \bowtie s) \bowtie t$. This expression has a join subtree with s, t, and ${}_A\gamma_{sum(B)}(r)$ as the leaves. Note that both these subtrees are under the same equivalence node in Volcano, since they are equivalent to each other.

  The join-enumeration ruleset we propose, which we call RS-Graph, can handle such cases without repeated effort in partitioning. We show that the RS-Graph ruleset is complete, and in practice it does not generate duplicate derivations.

- We have implemented our ruleset in the PyroJ optimizer which is an implementation of the Volcano framework. We present results of a performance study showing that the RS-Graph ruleset is not only practical, but significantly outperforms alternative rulesets, in some cases by up to two orders of magnitude in terms of execution time.

The rest of the paper is organized as follows. Section 2 gives an overview of a transformation-based query optimizers, existing rulesets used in transformation-based query optimizers and top-down join enumeration without cross-products. In Section 3 we prove the completeness / incompleteness of existing rulesets. Section 4 introduces the RS-Graph rule for join enumeration. We summarize our experimental results in Section 5, and conclude in Section 6.



(a) Initial Query (b) DAG representation of query

(c) Expanded DAG after transformation (Commutativity not shown explicitly)

Figure 1: Initial Query and Logical AND-OR DAG (LQDAG) Representation

2. BACKGROUND AND RELATED WORK

We briefly review background material on which the rest of the paper is based.

2.1 Query Optimization in Volcano

The Volcano/Cascades optimization framework [11, 10] is based on a system of transformation rules (also known as equivalence rules), which specify that the result of a particular transformation of a query tree is the same as the result of the original query tree. The key contribution of this framework is the efficient implementation of the transformation based approach.

We now describe the AND-OR DAG data-structure used in Volcano, which is the key technique for efficiently representing the given query and all its equivalent plans. Our description is based on [17]. An AND-OR DAG is a directed acyclic graph whose nodes can be divided into AND-nodes and OR-nodes; the AND-nodes have only OR-nodes as children and OR-nodes have only AND-nodes as children. An AND-node in the AND-OR DAG corresponds to an algebraic operator, such as the join operator ($\bowtie$) or a select operator (σ). It represents the expression defined by the operator and its inputs. Hereafter, we refer to the AND-nodes as operator nodes. An OR-node in the AND-OR DAG represents a set of logical expressions that generate the same result set; the set of such expressions is defined by the children AND nodes of the OR node, and their inputs. We shall refer to the OR-nodes as equivalence nodes henceforth.

The given query tree is initially represented in the AND-OR DAG formulation. For example, the query tree of Figure 1a is initially represented in the AND-OR DAG formulation, as shown in Figure 1b. Equivalence nodes (OR-nodes) are shown as boxes, while operator nodes (AND-nodes) are shown as circles.

The initial AND-OR DAG is then expanded by applying all possible logical transformations on every node of the initial DAG created from the given query. Suppose the only transformations possible are join associativity and commu-

tativity. Then the plans $A \bowtie (B \bowtie C)$ and $(A \bowtie C) \bowtie B$, as well as several plans equivalent to these modulo commutativity can be obtained by transformations on the initial AND-OR DAG of Figure 1b. These are represented in the DAG shown in Figure 1c. We call the AND-OR DAG representation after all logical transformations have been applied as the (expanded) Logical Query DAG (or LQDAG).

Note that the LQDAG has exactly one equivalence node for every subset of A, B, C; the node represents all ways of computing the joins of the relations in that subset.

In this paper we assume that before a transformation rule is applied at an operator node, the child equivalence nodes have been fully explored. Thus the children will not change after a rule has been applied. This property holds for Volcano as well as Cascades/Columbia optimizers.

Each equivalence node is assigned a unique ID when it is created. Given an operator node and its child equivalence nodes, we often need to check if that node is already present in the LQDAG. To perform this check, Volcano maintains a hash index on entries of the form "op(ID_1, ID_2, ..)" where ID_i is the ID of the ith child of the operator. The LQDAG along with associated indices is also called the "memo" structure in the Volcano/Cascades terminology.

Transformation rules generate trees where the internal nodes are operator nodes and leaf nodes are equivalence nodes. Such trees are inserted into the LQDAG taking care to ensure that if any subtree is already present in the DAG, we reuse the equivalence node under which it is present, instead of creating a new equivalence node. The above mentioned hash index is used for this task; for details see [11]. We note that the insertion process is very efficient, and its cost is effectively linear in the size of the tree being inserted.

Each operator node can have different physical implementations; for example, a join operator can be implemented as hash join, nested loop join or as merge join. Once the LQDAG has been generated, physical implementation rules are applied on the logical operators to generate the physical AND-OR DAG, which is called the Physical Query DAG or PQDAG for short. Cost-based pruning can be used in the process of expansion, to reduce the size of search space. We search through the space of all generated plans to find the plan with least estimated cost. Details of cost estimation and selection of the best plan are not relevant to our goal, and hence not discussed.

2.2 Rulesets used for join reordering

As explained by Pellenkoft et al. [15], a number of different rulesets have been used to explore the space of all join trees in a transformation-based optimizer. Note that none of the rulesets described in [15] suppresses cross products.

Rule set RS-B0: The simplest set of rules used (to generate the bushy space) is RS-B0.

- Left Associativity: $A \bowtie (B \bowtie C) \rightarrow (A \bowtie B) \bowtie C$
- Right Associativity: $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
- Commutativity: $A \bowtie B \rightarrow B \bowtie A$

RS-B0 is redundant since Right Associativity can be derived from Left Associativity and Commutativity. The following ruleset avoid this redundancy.

Rule set RS-B1: Rule set RS-B1 consists of subset of rules of RS-B0 namely :

- Left Associativity: $A \bowtie (B \bowtie C) \rightarrow (A \bowtie B) \bowtie C$
- Commutativity: $A \bowtie B \rightarrow B \bowtie A$

Even this ruleset can lead to duplicate derivations. As an example, consider a sequence of application of left associativity(LA) and commutativity(C) rules. In the following $[BC]$ denotes an equivalence node, under which we would have $B \bowtie C$ as well as $C \bowtie B$. The sequence $A \bowtie_1 [BC] \rightarrow_{LA} [AC] \bowtie_2 B \rightarrow_C B \bowtie_3 [AC] \rightarrow_{LA} [BC] \bowtie_4 A \rightarrow_C A \bowtie_5 [BC]$. $A \bowtie_5 [BC]$ is a duplicate of original expression $A \bowtie_1 [BC]$.

Pellenkoft et al. [15] showed that using RS-B1 for join enumeration generates an exponential number of duplicates. The complexity of join enumeration increases from $O(3^n)$ to $O(4^n)$ where n is number of relations. To avoid duplicate derivations, [15] presented a new rule set which generates the space of all bushy join trees without generating duplicates. The ruleset is as follows:

Rule set RS-B2:

- R_1 (Commutativity): $A \bowtie_0 B \rightarrow B \bowtie_1 A$
  Disable application of R_1, R_2, R_3, R_4 on new operator $\bowtie_1$
- R_2 (Left Associativity):
  $A \bowtie_0 (B \bowtie_1 C) \rightarrow (A \bowtie_2 B) \bowtie_3 C$
  Disable application of R_2, R_3, R_4 on new operator $\bowtie_3$
- R_3 (Right Associativity):
  $(A \bowtie_0 B) \bowtie_1 C \rightarrow A \bowtie_2 (B \bowtie_3 C)$
  Disable application of R_2, R_3, R_4 on new operator $\bowtie_2$
- R_4 (Exchange):
  $(A \bowtie_0 B) \bowtie_1 (C \bowtie_2 D) \rightarrow (A \bowtie_3 C) \bowtie_4 (B \bowtie_5 D)$
  Disable application of R_1, R_2, R_3, R_4 on new operator $\bowtie_4$

Disabling rules prevents duplicate derivation of the same join tree. Using RS-B2 in the example quoted previously, application of commutativity on $A \bowtie_1 [BC]$ generates $[BC] \bowtie_4 A$, left associativity generates $[AC] \bowtie_2 B$. Using commutativity on previous generates $B \bowtie_2 [AC]$. All the rules are disabled on the 3 newly generated alternatives, hence the duplicate $A \bowtie_5 [BC]$ is not generated.

2.3 Top-Down Join Enumeration Without Cross-Products

Definition 1. A *join graph* is a pair H = (V,E) where V is the set of base relations $R_1, ..., R_n$ and E is a set of edges. An edge between R_i and R_j indicates the presence of a join predicate between the relations.

When the join graph is sparse in comparison to a clique, the number of join trees without cross products is much smaller than the total number of join trees [14, 13]. Plans containing a cross product are rarely optimal, and hence it is desirable to exclude plans with cross-products from the search space generated.

Two approaches have done well at exploring and finding the optimal plan in this reduced space of all bushy join orderings without cross products: bottom-up join enumeration via dynamic programming and top-down join enumeration via memoization. Moerkotte and Neumann [13] presented an efficient dynamic programming based algorithm

(DPccp). The problem with bottom-up join enumeration algorithms is that they don't allow for pruning which is possible in top-down join enumeration algorithms.

TDMinCutLazy was the first efficient top-down join enumeration algorithm proposed by DeHaan and Tompa [2]. Fender and Moerkotte [3] proposed an alternative top-down join enumeration strategy (TDMinCutBranch), which is almost as efficient as DPccp. TDMinCutBranch used a graph-based enumeration strategy which generates only valid, i.e. cross-product free partitions, unlike TDMinCutLazy which used a generate and test approach, i.e. it first generates a partition and then checks whether it is valid. In the following year, Fender et al. [6] proposed another top-down enumeration strategy TDMinCutConservative which is currently the best. It is easier to implement and gives better runtime performance in comparison to TDMinCutBranch.

Algorithm 1 TDPlanGen(G)

Input: connected G=(V,E)
Output: optimal join tree for G

```
1: for i ← 1 to |V| do
2:     BestTree[{R_i}] ← R_i
3: end for
4: return TDPGSub(V)
```

Algorithm 2 TDPGSub($G_{|S}$)

Input: connected sub graph $G_{|S}$
Output: optimal join tree for $G_{|S}$

```
1: if BestTree[S] = null then
2:     for all S_1, S_2 ∈ P^sym_ccp do
3:         BuildTree(G_|S, TDPGSub(G_|S1), TDPGSub(G_|S2))
4:     end for
5: end if
6: return BestTree[S]
```

Algorithm 1, from [6] shows a generic top-down join enumeration algorithm TDPlanGen TDPlanGen initializes the building blocks for atomic relations first (line 2). In line 4, the subroutine TDPGSub (Algorithm 2) is called, which traverses recursively through the search space. TDPGSub takes as argument a connected subgraph $G_{|S}$. If the best tree for this graph is not known, a suitable partitioning strategy is used to partition S into two sets S_1 and S_2 such that the following conditions are satisfied:

- S_1 with $S_1 \subset S$ induces a connected subgraph $G_{|S_1}$
- S_2 with $S_2 \subset S$ induces a connected subgraph $G_{|S_2}$
- $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \phi$
- $\exists (v_1, v_2) \in E$ — $v_1 \in S_1 \wedge v_2 \in S_2$

If (S_1, S_2) is valid, so is (S_2, S_1). The set of all pairs (S_1, S_2) such that symmetric pairs are counted only once is denoted by $P^{sym}_{ccp}(S)$. For each $(S_1, S_2) \in P^{sym}_{ccp}(S)$, TDPGSub is recursively called on the subgraph induced by S_1 and the subgraph induced by S_2. The resulting best trees from both the calls are passed as arguments to subroutine BuildTree. The BuildTree subroutine creates a join tree for set S by combining the best join trees for S_1 and S_2, calculates its cost and updates the memo-table entry $BestTree[S]$ if the cost is lower than cost of current plan for S. The recursive descent stops when $|S| = 1$ or the best tree is already known. After all $(S_1, S_2) \in P^{sym}_{ccp}(S)$ have been tried, the entry in $BestTree[S]$ corresponds the optimal join tree for set of relations in S.

Different top-down join enumeration algorithms use different partitioning strategy to compute $P^{sym}_{ccp}(S)$. In our implementation we use the MinCutConservative partitioning strategy, described in [6].

Extensions of the top-down enumeration algorithms to handle non-equijoin predicates spanning more than 2 relations, such as $R_1.a + R_2.b = R_3.c$, and to handle outer-joins, are presented in [4, 5, 12].

3. COMPLETENESS OF RULESETS

In order to avoid generating trees with cross-products in a transformation-based optimizer, we need to ensure that on rule application, the alternative plans generated are within the space of cross-product free join trees. To achieve this, one approach is as follows. Every time a rule application leads to an alternative with cross-product, the alternative is discarded and rule is marked as applied. We call this *cross-product suppression*. We denote the ruleset RS-B1 with cross-product suppression as RS-B1-CPS. Similarly, ruleset RS-B2 with cross-product suppression as RS-B2-CPS and ruleset RS-B0 with cross-product suppression as RS-B0-CPS.

A ruleset is said to be *complete* if it can be used to generate the space of all cross-product free join orders starting from an initial plan without cross-products.

In Section 3.1 we show that the rulesets RS-B0-CPS and RS-B1-CPS are indeed complete. In Section 3.2 we give an example to show that ruleset RS-B2-CPS is unable to generate all trees in the space of all cross-product free join orders, and hence is incomplete.

3.1 Completeness of Rulesets RS-B0-CPS and RS-B1-CPS

Our goal is to show, given a connected join graph J and cross-product free query tree Q on J, using RS-B1-CPS we can reach any other cross-product free query tree Q' on J, starting with Q and remaining in the cross-product free space (i.e., each intermediate tree is cross-product free).

Note that ruleset RS-B0-CPS is clearly a superset of ruleset RS-B1-CPS. Hence it is sufficient to show ruleset RS-B1-CPS is complete to prove RS-B0-CPS is also complete.

The input join graph J is connected, hence we can number the nodes $1, 2, \ldots, n$ such that $(\ldots((R_1 \bowtie R_2) \bowtie R_3)\ldots) \bowtie R_n$ is a left-deep join tree without cross products. Such a tree can be constructed iteratively by picking any node as the initial one-node tree T_1. At each step, any of the nodes n_k in J that is not in T_i but is connected to at least one of the nodes in the current tree T_i is picked, and T_{i+1} is created by joining T_i with n_k. This process is continued until all nodes are added to the join tree. The resultant join tree is clearly left-deep, and does not contain any cross-product.

Lemma 1. *Given a cross-product free tree with relations $R_1, R_2, ...R_k$, it is possible to transform it into $T \bowtie R_k$ using RS-B1-CPS where T is a join tree computing the join of relations $R_1, R_2..R_{k-1}$.*

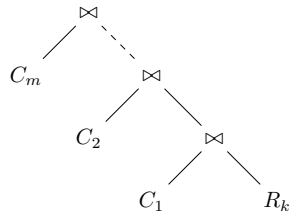


Figure 2: Reduced Query Tree

Proof. The goal of this lemma is to show that given a query tree, we can move the highest numbered relation to the top so that it is the last relation to be joined.

Let LDT be any cross-product free left-deep join tree on the relations in the given tree; as shown earlier, such a tree must exist.

Let R_k be the highest numbered relation and let R_1 .. R_{k-1} be the other relations in the tree. Using the commutativity rule, we can transform the given query tree to a form shown in Figure 2 where $C_1, C_2, ...C_m$ are join trees consisting of some number of relations. Application of commutativity rule does not introduce cross-products, hence all intermediate states are cross-product free. The depth of relation R_k is m.

Base Case: When m = 1 , R_k is already the last relation to be joined. Claim 1 holds trivially.

Induction on depth m: Assume Lemma 1 holds for $m <= j$. Let the depth of R_k be $m = j + 1$. Consider the subtree formed by $R_1, R_2, ..R_k$ in LDT. Since R_k is the last relation to be joined in the subtree, C_1 can be joined with at least one of the other C's say C_i. We pull C_i down to C_1 by repeatedly using a sequence of rules as follows.

We start from query tree as shown in Figure 3a, where X denotes the right subtree that is joined with C_{i-1}. We apply commutativity at $\bowtie_2$ (Figure 3b), apply left associativity at $\bowtie_1$ (Figure 3c) and finally apply commutativity at $\bowtie_3$ to get a join tree as shown in Figure 3d.

Note that each intermediate step ensures cross-product freedom, since C_i has a join predicate with C_1, which is part of the subtree denoted X, and so does C_{i-1}.

At the end of this sequence of transformations C_i has come one step closer to C_1. The same sequence of steps can be applied repeatedly on C_i. Each time the sequence of rules is applied, C_i comes one level closer to C_1.

Finally when C_i is one level above C_1, as shown in Figure 4a, we use left associativity to get the tree in Figure 4b. This tree is also cross-product free since C_i and C_1 are connected by a join predicate, and so are R_k and C_1.

Note that the depth of R_k has now reduced from $j + 1$ to j The induction hypothesis holds for depth $m \leq j$. Hence the lemma is proved.

□

Lemma 2. *Given any join graph J, any cross-product free query tree on J can be transformed into any cross-product free left-deep join tree on J using RS-B1-CPS.*

Proof. Assume the relations in the desired left-deep join tree are numbered R_1 to R_n as mentioned earlier. As shown in Lemma 1, the transformation rules in RS-B1-CPS can move R_n to the top so that it is the last relation to be joined.

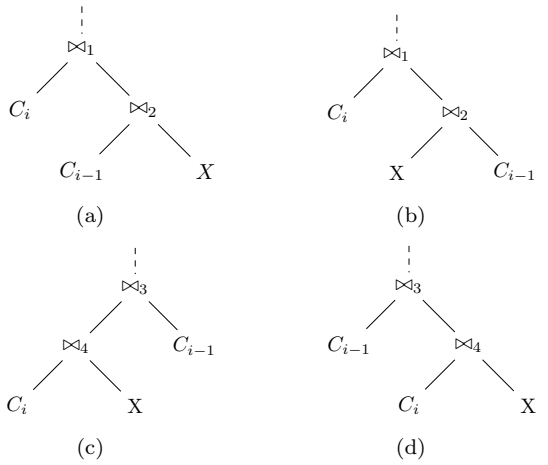
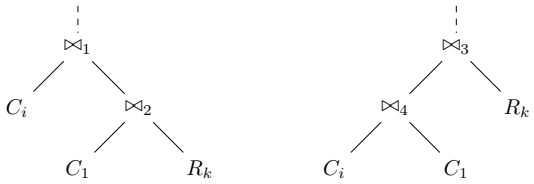


Figure 3: Swapping C_i and C_{i-1}



(a) Apply Left Associativity at $\bowtie_1$ (b) Resulting tree

Figure 4: Reduction in depth of R_k

Rules in RS-B1-CPS can be applied on the left sub-tree of the resultant tree to in turn move R_{n-1} to the top of its subtree. It is easy to show inductively that the desired left-deep join tree can be reached by a series of applications of rules in RS-B1-CPS. □

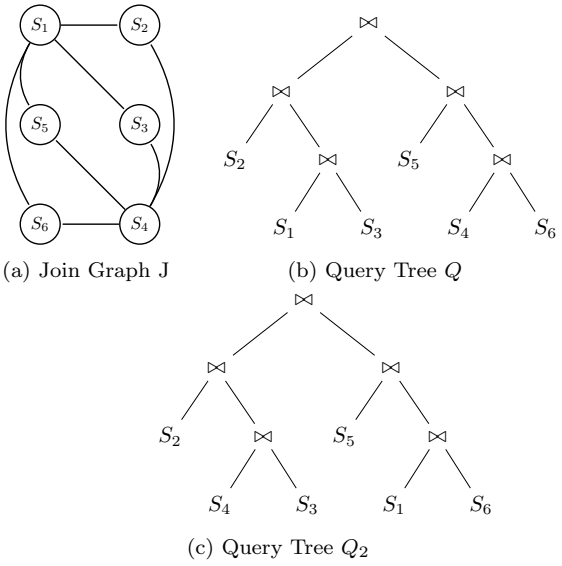
Lemma 3. *Given a join graph J, any cross-product free left-deep query tree T1 on the graph can be transformed into any cross-product free query tree T2 on J, using RS-B1-CPS.*

Proof. As shown in Lemma 2, any cross-product free join tree $T2$ can be transformed to any cross-product free left-deep join tree $T1$. We can reverse the sequence of transformations to transform $T1$ to $T2$, by using right-associativity in place of left-associativity, along with commutativity. Even though RS-B1-CPS does not include right-associativity, the following sequence of transformations using left-associativity and commutativity has the same effect as right-associativity, while avoiding cross-products. (Note that LA denotes left-associativity and C denotes commutativity.)

$(A \bowtie B) \bowtie C \rightarrow_C C \bowtie (A \bowtie B) \rightarrow_C C \bowtie (B \bowtie A) \rightarrow_{LA}$
$(C \bowtie B) \bowtie A \rightarrow_C (B \bowtie C) \bowtie A \rightarrow_C A \bowtie (B \bowtie C)$

It should be clear that if the initial and final states, i.e., $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are cross-product free, so are each of the intermediate steps, so RS-B1-CPS can be used to carry out the above sequence of transformations.

Given that the transformations in going from $T2$ to $T1$ ensured that the intermediate results are all cross-product free, we can carry out the reverse set of transformations



(a) Join Graph J (b) Query Tree Q

(c) Query Tree Q_2

Figure 5: Incompleteness of RS-B2-CPS

as described above using RS-B1-CPS, to transform $T1$ to $T2$. □

Theorem 1. *Ruleset RS-B1-CPS is complete, i.e., given any cross-product-free join tree $T1$, it is possible to generate any other cross-product-free join tree $T2$ by a series of applications of rules in RS-B1-CPS.*

Proof. The proof is straightforward given Lemmas 2 and 3: we choose any cross-product free left deep join tree LDT, and the above lemma show that we can use RS-B1-CPS to transform $T1$ to LDT, and LDT to $T2$. □

Although ruleset RS-B1-CPS is complete, it can be inefficient because it generates duplicates in the same way as RS-B1, and the inefficiency of RS-B1 was shown in [15]. The performance results in Section 5.2 also demonstrate the inefficiency of RS-B1-CPS.

3.2 Incompleteness of Ruleset RS-B2-CPS

We show that RS-B2-CPS cannot generate the space of all join trees without cross-products using an example. Given a set of relations to be joined with join graph J (Figure 5a) and initial query tree Q (Figure 5b), we wish to generate Q_2 (Figure 5c). Query tree Q_2 differs from Q in that the positions of relations S_1 and S_4 have been swapped. Query tree Q_2 contains no cross-products and hence belongs to the space of cross-product free join trees. To show RS-B2-CPS is incomplete, it is sufficient to show that Q_2 cannot be generated using RS-B2-CPS starting from Q.

In RS-B2-CPS, once any of rules R_2, R_3, R_4 have been applied on a join operator, none of them can be applied on the newly generated join operator. Consider the rules that can be applied at the root of Q.

1. Commutativity rule (R_1) generates a mirror image of original tree and doesn't change the tree structurally.

2. Applying Left associativity (R_2) at the root of the tree would create a new join node which is similar to the root of Q except that a subtree from the left child of Q has been moved to the right child. But since R_2, R_3 and R_4 are disabled on the new join node, it is not possible to apply any transformation on the new join node that will bring any part of the right-subtree to the left of the root.

   Applying rules at a node below the root node also cannot move any part of the right-subtree to the left of the root.

   In our example, S_1 has moved to right subtree and S_4 has moved to left subtree; thus applying R_2 at the root of Q cannot possibly lead to the desired tree Q_2.

3. The case of Right associativity (R_3) is symmetric to Left associativity.

4. To apply the exchange rule R_3 at the root of Q to swap S_1 and S_4, we need the left subtree to be of the form $(S_2 \bowtie S_3) \bowtie S_1$ and similarly the right subtree to be $S_4 \bowtie (S_5 \bowtie S_6)$. Since S_2 and S_3 do not have an edge between them in J, $S_2 \bowtie S_3$ is a cross-product. Since the required intermediate state for application of exchange rule does not belong to the space of cross-product free join trees, RS-B2-CPS would not be able to swap S_1 and S_4 at the root of Q.

   Note that it is possible to swap S_2 and S_5, S_2 and S_6, S_3 and S_5, or S_3 and S_6 by applying the exchange operator R_4 after applying some sequence of R_1, R_2, R_3 on the subtrees of Q. However, rules R_2, R_3 and R_4 are disabled on the resultant new join nodes, so it is not possible to perform any further operations on the new join nodes to bring S_1 and S_4 to the desired sides.

Thus, whatever rule is applied at the root of Q, it cannot lead to a sequence of transformations that exchange S_1 and S_4 Note also that application of rules below the root of Q have no impact on moving S_1 and S_4.

As a result, it is not possible to go from Q to Q_2 using any sequence of transformations using RS-B2-CPS. Thus, ruleset RS-B2-CPS is incomplete.

The performance results in Section **??** show that the above incompleteness can lead to the generation of suboptimal plans with a much larger cost than the actual optimal plan.

4. GRAPH ENUMERATOR RULE

The Volcano framework is highly extensible, since it is very easy to add new transformation rules. On the other hand, as we saw in Section 3, the transformation rulesets which have been proposed for join enumeration can be quite inefficient. In contrast, state-of-the-art cross-product free join enumeration techniques [3, 6] are very efficient, but do not provide any support for extensibility.

In this section, we propose an approach that combines the benefits of the two approaches. The first step in combining the approaches is to note that we can use join-order enumeration techniques using equivalence nodes in place of relations. As a result, an equivalence node representing the result of an aggregation operation can be treated just like a relation.

Intuitively, our approach is based on finding maximal subtrees containing only join operations in the query DAG,

and then applying cross-product free join-enumeration techniques on the leaves of these maximal join subtrees to generate alternative join orders, instead of using the transformation rules in the rulesets RS-B1-CPS or RS-B2 described earlier. As explained in Section 1, a key complication is the fact that application of rules (such as pushdown/pull-up of aggregation operations) could result in the presence of multiple subtrees of joins under a single logical equivalence node where the subtrees have different sets of leaf nodes. Join enumeration should handle this case without repeated work; in this section, we show how to achieve this goal.

In Section 4.1 we propose an extension to Volcano LQDAG generation algorithm to track the sets of equivalence nodes at the leaves of maximal join trees (which we call join-sets) under each equivalence node. In Section 4.2 we show how the partitioning algorithms used in top-down algorithms for cross-product free join order enumeration can be used on these join-sets. In Section 4.3 we describe how the output of partitioning is converted into trees and added back into the LQDAG. In Section 4.4 we show that our approach is complete, while Section 4.5 addresses issues of efficiency.

4.1 Join-Sets

We first introduce some notation.

Definition 2. A *base equivalence node* in an expanded LQDAG is an equivalence node that has no join operator as a child; such an equivalence node can be either a base relation, or it must have only non-join operators as children.

Definition 3. A *join equivalence node* in an expanded LQDAG is an equivalence node such that at least one of its child operator nodes is a join operator.

Definition 4. A *join tree* in an expanded LQDAG is a tree in the LQDAG where the root is an equivalence node, every internal node is either an equivalence node or a join operator, and every leaf node is an equivalence node,

Definition 5. A *maximal join tree* in an expanded LQDAG is a join tree where every leaf node is a base equivalence node.

Definition 6. A *join-set* for an equivalence node E in an expanded LQDAG is a pair $J = (V, P)$ where V is the set of equivalence nodes at the leaves of some join tree T in the LQDAG, and P is the set of predicates between V. We say that T is a join tree corresponding to the join-set.

Definition 7. A *maximal join-set* for an equivalence node E in an expanded LQDAG is a join-set whose corresponding join tree T is a maximal join tree.

We store with each equivalence node the set of all maximal join-sets for that node. It may appear that having multiple maximal join-sets can increase execution time. However, as we will show later in Section 4.5, with commonly used transformation rules, nodes have only one such maximal join-set.

For example consider the expanded LQDAG as shown in Figure 6. In the DAG, E_i's represent equivalence nodes, R_i's represent equivalence nodes of base relations, γ_i's are aggregation operator nodes and $\bowtie_i$'s are join operator nodes. The DAG was created from an initial tree $(\gamma_1(R_1 \bowtie_1 R_2)) \bowtie_0 R_3$ by application of the aggregation pushdown rule at root operator γ_1. The aggregation operator is pushed down to R_2, creating a new join operator $\bowtie_2$ which is exposed to join

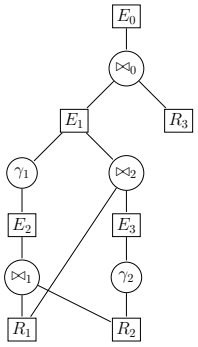


Figure 6: Expanded DAG on applying aggregation pushdown

operators above E_1. Note that R_1, R_2, R_3 and E_3 are base equivalence nodes, E_0, E_1 and E_2 are join equivalence nodes.

Let θ_i denote the join predicate of $\bowtie_i$. Then, the maximal join-set for each node in the LQDAG is as follows: at R_1 : $[(\{R_1\}, null)]$, at R_2 : $[(\{R_2\}, null)]$, at E_3 : $[(\{E_3\}, null)]$, at E_2 : $[(\{R_1, R_2\}, \theta_1)]$, at E_1 : $[(\{R_1, E_3\}, \theta_2)]$, and at E_0 : $[(\{R_1, E_3, R_3\}, \{\theta_2, \theta_3\})]$.

Note that Volcano expands the Logical Query DAG in a top-down fashion. The process of exploring an equivalence node applies rules to children of the equivalence node. Specifically, when an equivalence node E is explored, all rules that match operator children o_i of E are applied. Before applying a rule to operator o_i (and its descendants if any that match the rule pattern) the equivalence nodes that match the leaves of the rule pattern are explored first.

In the process of applying a rule to a child of E, more operator children may get added to equivalence node E. Rules that match these operator nodes are also applied as part of exploring E. Once exploration of E is complete, no more operator children can be added to E. This makes it is easy to compute the set of maximal join-sets at each equivalence node as follows:

- If E corresponds to a base relation, or after exploring E completely, it has no join children, it is a base equivalence node. If E is found to be a base equivalence node add join-set $(\{E\}, null)$ into E's set of join-sets.
- When a join operator node child of equivalence node E is explored, the following steps are taken by the GraphRule algorithm described in Section 4.2: a cross product of join-sets of the left equivalence node S_1 and right equivalence node S_2 s used to create $\{\{s_1, s_2\} | s_1 \in S_1 \wedge s_2 \in S_2\}$. The merged set $s_1 \cup s_2$ along with appropriate join predicates are used to create a join-set s'. If s' is not already present in the set of join-sets of E, it is added to the set of join-sets of node E.

4.2 Transformation Rule

We remove existing join enumeration rules and replace it with our new rule RS-Graph. Rule *RS-Graph* matches pattern $E_1 \bowtie E_2$ and calls GraphRule$(\bowtie, E_1, E_2, parent)$ to get the set of all join operations under equivalence node *parent*.

We now describe the GraphRule subroutine (refer Algorithm 3). Let rule be applied on join operator Op with child

equivalence nodes A,B and parent equivalence node P. For each pair of join-set $(js_A, js_B) \in A.JoinSets * B.JoinSets$, we merge the pair to form a join-set js. We define the merge of the two join-sets $js_A = (V_1, P_1)$ and $js_B = (V_2, P_2)$ as $(V_1 \cup V_2, P_1 \wedge P_2)$.

To avoid recomputing the same join trees, we check if the parent equivalence node contains js. To check if two join-sets at an equivalence node are equal, it is sufficient to check if they have same equivalence nodes. If the join-sets have the same equivalence nodes, then they will also have the same predicates.

Absence of join set js in the parent node indicates a new set of base equivalence nodes being joined. We then generate all join partitions $S_1 \bowtie S_2$ where $S_1 \cup S_2 = V_1 \cup V_2$. We create join graph G from join-set js. We invoke partition on G to generate all partitions. We use the partitions to generate join trees which are then added into the result list.

Algorithm 3 GRAPHRULE(Op, A, B, parent)

Input: Operator op, op's parent equivalence node parent and op's child equivalence nodes A, B
Output: Set of join trees J

```
1: result ← []
2: for all js_A ∈ A.JoinSets do
3:     for all js_B ∈ B.JoinSets do
4:         js ← merge(js_A, js_B)
5:         if not parent.JoinSets.contains(js) then
6:             parent.JoinSets.add(js)
7:             G ← CREATEGRAPH(js)
8:             cuts ← PARTITION(G, φ, φ)
9:             trees ← CREATETREES(cuts)
10:            result.concat(trees)
11:        end if
12:    end for
13: end for
14: return result
```

In the CREATEGRAPH subroutine, given a join-set js, we construct a join graph whose vertices are the base equivalence nodes being joined, and with an edge between two nodes indicating a join predicate between them. Since creation of the join graph is done for each join-set, we use an efficient implementation to create the join graph, described below.

First note that in addition to explicit join predicates, other join predicates might be present implicitly as well; for example if predicates $A.a = B.b$ and $B.b = C.c$ are present, the predicate $A.a = C.c$ is also implicitly present and therefore there should be an edge between A and C in the join graph. We use the standard technique of creating equivalence classes of attributes, such that all attributes in an equivalence class are constrained to have the same value. For each pair of base equivalence classes R_i and R_j in the join-set that have attributes in the same equivalence class, we create an edge between R_i and R_j.

In the PARTITION subroutine, given a connected join graph G with set of equivalence nodes S, we wish to partition it into two disjoint subsets S_1 and S_2 where $S_1 \cup S_2 = S$, and each of S_1 and S_2 induces a connected subgraph in G. If (S_1, S_2) is valid, so is (S_2, S_1). Since the latter gives no additional information, we aim to generate only one of the two.

A number of different graph-based partitioning algorithms have been proposed[2, 3, 6]. In our implementation we use the MINCUTCONSERVATIVE partitioning algorithm proposed by [6]. MINCUTCONSERVATIVE has the best runtime performance among the alternatives, as shown in [6].

4.3 From Partitions to Join Trees

The call to PARTITION returns a set of partitions. Each partition in the set represents a set of base equivalence nodes S_1 such that S_1 and $G \backslash S_1$ induce connected subgraphs in join graph G and they have a join predicate between them. Results of rule application are expression trees which are added back into the LQDAG. Unlike existing rulesets which generate a single output tree on every rule application, RS-Graph generates all possible join operators below the root operator's parent equivalence node for the given set of base equivalence nodes being joined.

Given a particular partition of S into S_1 and $S_2 = S \backslash S_1$, we need to create a join node with equivalence nodes corresponding to S_1 and S_2 as its children. The equivalence node corresponding to S_i represents the join of all the base equivalence nodes in S_i. To avoid creating unnecessary copies due to commutativity, as in [6] we pick S_1 as left child if the ID of equivalence node corresponding to S_1 is less than ID of the equivalence node corresponding to S_2; and otherwise we use S_2 as the left child.

For each S_i, we first need to check if there is already an equivalence node corresponding to S_i; if it does not exist, we need to create a (cross-product free) join tree containing all base equivalence nodes in S_i, and add it to the Volcano memo structure as explained below, which would create the required equivalence node.

To efficiently check if an equivalence node already exists for S_i, whenever we create an equivalence node using RS-Graph, we (conceptually) create an extra n-ary join operator whose children are the base equivalence nodes, sorted in order of ID, and add the n-ary join operator node as a child of the equivalence node. The Volcano memo structure then automatically adds an entry for this n-ary join operator. Thus, given partition S_i, we create an n-ary join operator as above, and look it up in the memo structure; if it is found, we use the existing equivalence node.

To create a cross-product free join tree for S_i, we do a depth first traversal on the subgraph induced by S_i to get a ordered set of base equivalence node $R_1, R_2, ..R_k$ such that $J_1 = ((..(R_1 \bowtie R_2)..) \bowtie R_k)$ forms a cross-product free left-deep join tree J_1. Recall that for all join operators, we wish to ensure that the ID of the left child equivalence node is less than the ID of right child equivalence node. To ensure this, we swap the children of any join node where this property is violated. Let the resultant tree be J_i. We add J_i to the memo structure using the standard Volcano algorithm. This step returns an equivalence node for the root of J_i.[1]

Note also that the above step does not by itself generate all possible subtrees; it only generates all join children j_i of the equivalence node for S. RS-Graph is recursively applied on the child equivalence nodes of the j_i, which eventually generates the entire space of cross-product free join trees.

[1] Note that the Volcano memo structure detects if there is already an equivalence node for J_i, so the test described above for S_i is not essential for correctness, but improves performance by avoiding the creation of J_i and the steps involved in checking if it is already in the memo.

4.4 Completeness

Before we present a proof sketch, we use the example in Figure 6 to explain the intuition. The initial DAG had a single maximal join tree at E_0, which joins E_1 with R_3, and the only child of E_1 was γ_1. The aggregation pushdown rule creates a new maximal join tree $(R_1 \bowtie_2 \gamma_2(R_2)) \bowtie_0 R_3$. The maximal join-set at E_0 is thus R_1, E_3, R_3. If all partitions of this join-set are cross-product free, RS-Graph would generate equivalence nodes corresponding to $R_1 \bowtie R_3$, and $E_3 \bowtie R_3$, in addition to the equivalence nodes shown.

The partitions do not explicitly include $E_1 \bowtie R_3$ which could correspond to the overall plan $(\gamma_1(R_1 \bowtie_1 R_2)) \bowtie_0 R_3$. However, it is worth noting that the partition $\{R_1, E_3\}$, $\{R_3\}$ implicitly represents the join $E_1 \bowtie R_3$, since the equivalence node for $\{R_1, E_3\}$, is E_1 (the memo lookup would in fact return E_1). In other words, any non-base equivalence node E_i is implicitly represented by the set of base equivalence nodes that form a maximal join-set at E_i, and one of the partitions would include all the nodes in the join-set for E_i.

We now sketch a proof of completeness of RS-Graph. Given a join-set $js = (V, E)$, Fender et al. [6] showed that the MinCutConservative partitioning algorithm generates all possible (S_1, S_2) such that S_1 and S_2 are connected and they have a join predicate between them. As long as join-sets are correctly maintained, and after any change in the join sets at a node, the rule is applied at all the ancestors, we will generate all possible cross-product free join trees. We split the proof into two subparts. First, we show that the join-set maintenance correctly stores all maximal join-sets at each equivalence node. Second, rule application using the maximal join-sets generates all the join trees.

If an equivalence node represents a base relation, or after expansion does not have any join operator below it, it is a base equivalence node, and as described earlier, the equivalence node will be added into its own set of join-sets. If the equivalence node is a join equivalence node, the RS-Graph rule will be applied at the join operators below the equivalence node and this will add the appropriate join-sets into the equivalence node. Equivalence nodes below have the correct set of join-sets since in Volcano, the child equivalence nodes are expanded before applying a rule at the operator node, and hence the set of join-sets at the child equivalence nodes will not change subsequently. Thus we can show that the set of join-sets stored at a node contains all maximal join-sets for the node. (This can be proved formally using induction.)

When RS-Graph is applied, for a particular join-set, the partition function generates all possible cross-product free partitions of the join-set, and creates appropriate join nodes (and child equivalence nodes as required).

Now, given any cross-product free join tree T for an equivalence node E we can prove that it is represented in the resultant DAG, as follows. The leaves of T may be any equivalence nodes, not necessarily base equivalence nodes, and such leaves may not be part of the maximal join-set. However, any leaf L_i of T which is not a base equivalence node is represented by the set of base equivalence nodes in one of its maximal join-sets. Given a tree T_i let *base-nodes*(T_i) denote the set containing all base equivalence node leaves of T_i, and all the base equivalence nodes representing the remaining leaves of T_i. Now, base-nodes(T) is a maximal join-set for E, by definition. Let T_l and T_r be the left and right subtrees of T. Thus one of the partitions S_1, S_2 of this join-set would be such that S_1= base-nodes(T_l) and S_2 = base-nodes(T_r). The DAG would then have a join operator, whose children are equivalence nodes representing S_1 and S_2. We can use the above fact to prove that T is represented in the DAG, by an inductive argument, where the base case corresponds to leaves of T.

Thus, the ruleset RS-Graph is complete.

4.5 Efficiency

Since the min-cut partitioning algorithm generates all partitions and the check on line 5 in Algorithm 3 ensures that each join-set is enumerated only once, only one duplicate can be generated, which is the tree from which the join-set was generated. Since we know the pair (js_A, js_B) that were merged to create a merged join-set js, we can remove the initial pair from the set of partitions. Hence the RS-Graph rule does not generate any duplicates within a joinset.

Duplicates are possible across maximal join-sets for a given equivalence node: although a partition of one join-set will not be exactly the same as a partition of another join-set, the two may correspond to the same pair of equivalence nodes, leading to duplicate derivations of the same root join operation. The number of such duplicate derivations can be capped by the number of maximal join-sets at an equivalence node. In theory this number could be very large with arbitrary transformation rules. However, as described shortly, with commonly used transformation rules, each equivalence node will have only a single maximal join-set, and hence there is no duplication within an equivalence node with such transformation rules.

Duplication cannot occur across equivalence nodes, since the joins generated by partitions at two different equivalence nodes cannot be equivalent to each other. Note that two equivalence nodes may well have many relations in common, but this does not result in any duplicate work due to partitions.

We now consider the number of maximal join-sets at an equivalence node. With arbitrary transformation rules, there may be more than one maximal join-set for a given equivalence node. However, in practise, for most transformation rules the set of base equivalence nodes under the original and transformed root are identical, and thus applying such a rule cannot result in creation of a new maximal join-set.

In particular RS-Graph itself does not result in creation of new maximal join-sets (nor do any of the rules in RS-B0, RS-B1, and RS-B2). Transformations that push joins below other operators, for example below aggregation (which can be equivalently called aggregation pull-up) do not affect the maximal join-set.

Some rules may add new join children, for example a rule that pulls a join up from below a selection or aggregation on top of equivalence node E_i creates a new join child under the parent operation. If E_i is a join of multiple nodes, there is a potential for multiple join-sets to be created.

Consider the case of join pull-up from below a selection, equivalent to selection pushdown below a join, first. A selection condition can be pushed to intermediate nodes, but eventually it would be pushed to the relation on which it is to be applied. In this case, there would be a unique maximal join-set once the selection has been pushed all the way down; all other join-sets corresponding to intermediate states would be subsets of this maximal join-set.

Consider now the rule to pull a join up from below an aggregation, which transforms $\gamma_{LA}(R_1 \bowtie R_2)$ to $\gamma_{LA}(R_1) \bowtie R_2$ (under some conditions such as R_1 has a foreign key referencing R_2). Such a rule is applicable only when the join is really a semi-join. In general, if there are multiple relations joined, several of the joins may be semi-joins satisfying the join pull-up condition. Again, each of these can be pulled up independent of the others, and there is a unique maximal join-set after all the pull-up transformations have been applied.

Other transformations that can add relations to the join-set include join pull-up from outer-joins and semi-joins, as well as transformation of outer-joins and semi-joins to joins. Although we have not examined all such rules, for rules we have examined follow the same pattern as for selection and aggregation, and result in a unique maximal join-set. A more exhaustive analysis is a topic for future work.

4.6 Discussion

The graph partitioning techniques have been extended by [5, 4] to allow predicates involving multiple relations (which are represented by hyperedges) and to generate only partitions that preserve correctness in the presence of outer-joins and semi/anti-joins. Although we have not currently implemented these extensions, the extensions fit nicely into our framework, and do not require major changes to our algorithms.

5. EVALUATION

We now describe our experimental setup and findings.

5.1 Implementation and Workload

We use the PyroJ optimizer as the base for all our experiments. PyroJ is a Java translation of the Pyro optimizer developed earlier at IIT Bombay [16] which is an implementation of the Volcano optimization framework. Our implementation performs unification of equivalence nodes E_1 and E_2 when a transformation applied to a child expression $e1$ of E_1 results in an expression e_2 that is found to already exist as a child of E_2; see [16] for details.

Note that as in [6], our results are for the case of logical plan generation without pruning, and do not include the overhead of cost estimation or generation of physical plans.

The optimizer ruleset currently includes select push-down, aggregation push-down and pull-up, and join enumeration rules. We note that select push-down is done only as part of a normalization phase (before transformations are run) which creates an initial tree which is cross-product free, with selections pushed down to the base relations.

Our implementation of the RS-Graph rule adds some fields to the equivalence node data structures, but it does not impact the runtime performance of other rulesets in any other way.

We implemented a graph-based SQL generator to generate chain, star, cycle and clique queries of any desired size.

Our experiments were conducted on Intel i5 3.5 GHz with 8 Gbyte of RAM running on Ubuntu 11.10. We found that run times decreased greatly if the same query is optimized multiple times, and ultimately traced the variations to Java's JIT compilation strategy, which performs optimizations dynamically, based on how many times a particular piece of code is executed. Turning off JIT compilation made execution times predictable but larger. We used a Java HotSpot JVM flag -XX:CompileThreshold=1, and ran a large query a few times to ensure that all functions get compiled, before running other tests.

We checked if our implementations using rulesets RS-B1-CPS and RS-Graph generated the same search space, by comparing the number of equivalence nodes and operator nodes in the LQDAG. These matched for all the queries that we tested (RS-Graph generated fewer operator node since it keeps only one of $E_i \bowtie E_j$ and $E_j \bowtie E_i$, an optimization that cannot be exploited by RS-B1-CPS but we checked the counts taking this into account.)

5.2 Efficiency of Rulesets

We now compare RS-B1-CPS, RS-B2 and RS-Graph rule in terms of their efficiency, i.e. the time required to generate the LQDAG using each of the rulesets. We do not add the time required to generating the Physical Query DAG (PQDAG) and the time for finding the optimal plan.

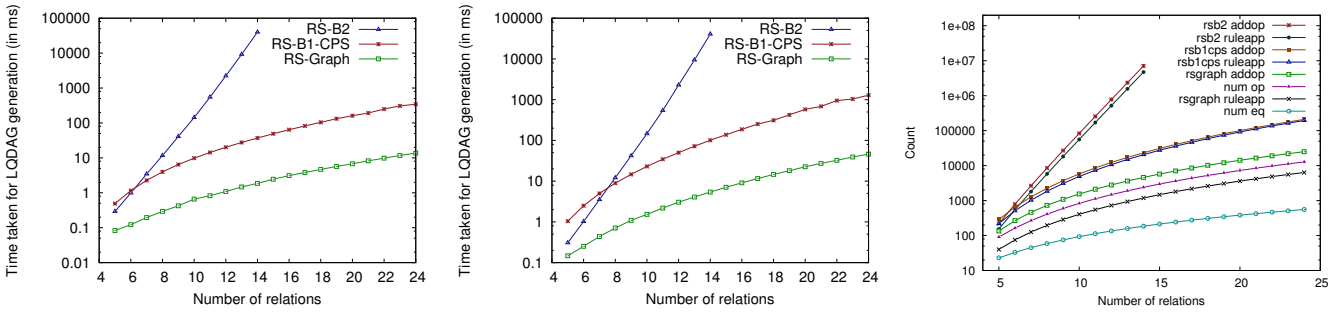
We first evaluate the performance of different rulesets on chain, cycle, star and clique queries of different sizes to verify what benefits RS-Graph provides. These queries do not have any operators other than joins, and hence no other transformations are applicable. Later, in Section 5.3, we present results on the interaction of RS-Graph with aggregation push-down and pull-up rules.

We begin with chain queries. Figure 7a shows how the time taken for LQDAG generation increases with an increase in the number of relations in the chain queries. (Note that all the graphs in this section have a log-scale for the time axis.) A chain query with n relations is the simplest of all queries with a connected join graph. The number of equivalence nodes in the cross-product free space is polynomial (each such node corresponds to an interval from the chain), whereas the number of equivalence nodes with cross products is exponential. RS-B2 performs extremely poorly, because it generates the space of all join trees including those with cross-products. RS-B1-CPS and RS-Graph both take time polynomial in the number of relations, but RS-Graph outperforms RS-B1-CPS by an order of magnitude.

We also measured the breakup of the execution time; we omit details but note that rule application times with RS-Graph ranged from about 1/2 to 1/10th of the overall time, with the rest of the time spent in LQDAG related activities, across the range of queries (chain, cycle, star and clique) that we considered.

The time for LQDAG generation for the case of cycle queries, with varying number of relations, is shown in Figure 7b. The graph is very similar to that of the chain queries. As the number of relations increase, the time taken by RS-B1-CPS increases at a much faster rate in comparison to RS-Graph since the number of duplicate derivations due to RS-B1-CPS increases with the number of relations. As before RS-B2 performs very poorly, since it generates an exponential number of join nodes, whereas the number of cross-product free join nodes is polynomial for cycle queries

Figure 7c shows the number of equivalence nodes (`num eq`), and the number of operator nodes (`num op`) (all join nodes in this case) in the LQDAG. It also shows the number of times rules were applied by each technique (the lines with suffix `ruleapp`), and the number of times an operator node is attempted to be added to the LQDAG (suffix `addop`). Note that each partition counts as a rule application for RS-Graph. Comparing the number of rule applications with



(a) Performance results for chain queries (b) Performance results for cycle queries (c) Statistics for cycle queries

Figure 7: Results on chain and cycle queries

the number of operator nodes in the LQDAG indicates how many derivations were duplicates.

It can be seen from the figure that the number of rule applications performed by RS-Graph is about half the number of operator nodes in the DAG, since commutative join cases are suppressed. The number of times operators are added to the DAG by RS-Graph is a little more than the number of operator nodes, even though the top level is duplicate free. There are two reasons: first each rule application results in two addops of the two child n-ary join operators, and second, when an n-ary join operator is new, RS-Graph inserts a whole tree, and lower operator nodes in the tree may be duplicates. RS-B1-CPS has significantly more rule applications than RS-Graph.

We note that statistics for chain queries are similar to those for cycle queries and are omitted for lack of space.

For star queries (Figure 8a), RS-Graph outperforms RS-B1-CPS by a factor of 4 or more in terms of time taken. In turn, RS-B1-CPS outperforms RS-B2, with the margin widening with increasing number of relations. The reason is that for star queries, the only cross-product free trees are those that are left-deep (or can be made left-deep by just commutativity), and hence the number of operator nodes in the LQDAG is $n2^{n-1}$ with n relations, whereas there are 3^n operator nodes with bushy trees when cross-products are allowed, and RS-B2 generates the space of such plans.

These numbers can be cross-checked from Figure 8b; the meaning of the legends in the figure are the same as in Figure 7c. Note that the lines for number of operator-adds and number of rule applications of RS-B1-CPS are identical in this case since associativity generates a non-cross-product result in only half of the times it is applied, but when it succeeds it results in two operator adds.

Lastly we consider clique queries. Clique queries are completely connected, and hence every join tree is cross-product free. As shown in Figure 8c, for clique queries, RS-B2 performs better than RS-B1-CPS because RS-B2 generates all nodes in the DAG without generating any duplicates, whereas RS-B1-CPS generates $4^n/3^n$ duplicates per join node in the DAG, which results in a very significant deterioration of performance. RS-Graph performs about 30 to 50 % better than RS-B2 in this case.

We note that statistics for clique queries follow the same broad trend as those for star queries, although the number of operator nodes are significantly larger since the cross-product free join space now includes bushy trees. We omit the details for lack of space.

In summary, RS-Graph always performs significantly better than both the alternatives; RS-B1-CPS usually outperforms RS-B2, but for clique queries it performs worse by a factor of up to 30.

5.3 Interaction with Other Transformations

We introduced rules for aggregation pull-up and push-down, and studied the performance of RS-Graph and RS-B1-CPS in conjunction with these rules.

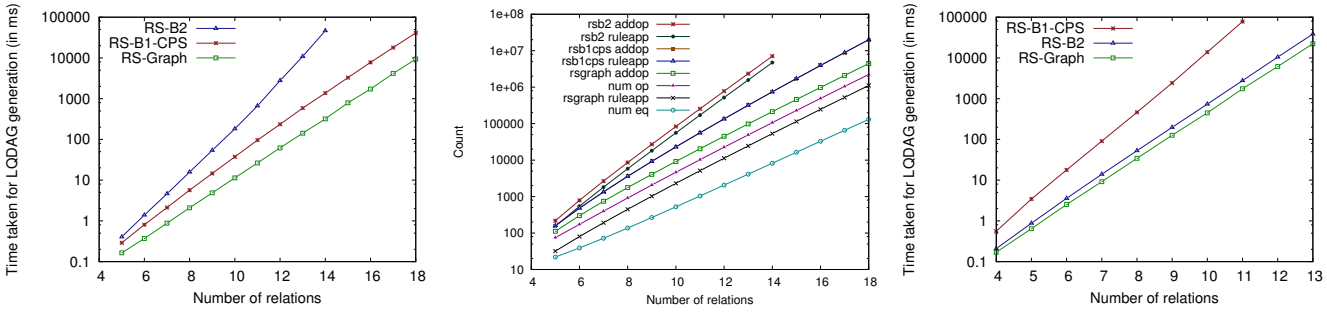
To show the need for using aggregation transformation rules along with join ordering rules, we used a sample query with a fairly selective semi-join (expressed as a join) on top of a query that aggregates a join result. The best plan has the semi-join pushed below the aggregation, and was indeed found by our optimizer using aggregate pull-up in conjunction with RS-Graph. The estimated cost of this plan was less than half of the estimated cost without aggregation pull-up. It is easy to create a set up where the difference can be much more; see for example, [19].

We then checked the efficiency of RS-Graph versus RS-B1-CPS in the presence of aggregation push-down rules. For lack of space, we present results only for one case, a star query with 14 relations with a aggregation on top of the join query, a case which we believe stresses our techniques significantly. The aggregation applies the *min* operation, on an attribute of the central relation, with a group-by on other attributes of the same relation. The joins with the relations model semi-joins, and we assumed that a project is implicitly present with each join operator to allow aggregation push-down without adding extra projection operators. Aggregation push-down succeeds on one of the two inputs of every join on which it is applied.

The results showed that optimization with aggregation push-down combined with RS-B1-CPS took 3.4 seconds, while aggregation push-down combined with RS-Graph took 1.4 seconds. The number of add operator calls was 1.7 million for RS-B1-CPS versus 0.33 million for RS-Graph.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new join transformation rule, RS-Graph, based on top-down join enumeration techniques, which allows transformation based optimizers such as Volcano and Cascades to efficiently generate the space of cross-product free join trees. The transformation rule can co-exist with other transformation rules, such as aggregation transformations and semi/outer-join transformations.



(a) Performance results for star queries (b) Statistics for star queries (c) Performance results for clique queries

Figure 8: Results on star and clique queries

Our experimental results show the significant benefits of our techniques.

One area of future work is in checking the performance after adding more transformations to our optimizer, such as outer-join transformations, and pushing of top-K selections. We would also like to study the efficiency of using outer-join transformation rules, as compared to top-down join enumeration algorithms that have been extended to handle outer-joins, such as [5, 4, 12].

Given incompleteness due to pruning of cross products, completeness of RS-B2 in the presence of cost based pruning (as proposed, for example, in the Columbia optimizer, [18]) also needs to be examined, as part of future work.

7. REFERENCES

[1] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, pages 354–366, 1994.

[2] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD*, pages 785–796. ACM, 2007.

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

[4] P. Fender and G. Moerkotte. Counter strike: generic top-down join enumeration for hypergraphs. *Procs. VLDB Endowment*, 6(14):1822–1833, 2013.

[5] P. Fender and G. Moerkotte. Top down plan generation: From theory to practice. In *ICDE*, pages 1105–1116, 2013.

[6] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, pages 414–425, 2012.

[7] C. A. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD Conference*, pages 571–581, 2001.

[8] C. A. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *ICDE*, pages 402–409, 1992.

[9] C. A. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[10] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[11] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE, 1993.

[12] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *SIGMOD Conference*, pages 493–504, 2013.

[13] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[14] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[15] A. Pellenkoft, C. A. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *VLDB*, pages 306–315, 1997.

[16] P. Roy. *Multi Query Optimization and Applications*. PhD thesis, PhD thesis, Indian Institute of Technology, Bombay, 2001.

[17] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD*, pages 249–260, 2000.

[18] L. D. Shapiro et al. Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*, pages 20–33, 2001.

[19] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.