Optimizing Join Enumeration in Transformation-based Query Optimizers

B.Tech. Project 1st Stage Report

Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology (Honors)

Student:
Anil Shanbhag
Roll No: 100050082

Guide:
Dr. S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

Transformation-based query optimizers explore the search space exhaustively using transformation rules. This report looks at a specific problem of enumeration of all valid bushy join trees without cross products. We show that existing rule sets either fail to explore the complete space or explore the entire space but generate exponential number of duplicates. A literature survey of techniques used for join enumeration in different settings is presented including DP-based and memoization-based algorithms for plan generation. We present two new methods for enumerating the search space which perform superior to existing methods.

Acknowledgement

I wish to express my sincere gratitude and indebtedness to my guide, Dr. S. Sudarshan for his constant support and guidance throughout the project.

Anil Shanbhag
B.Tech. IV
CSE, IIT Bombay

Contents

Chapter 1

Introduction

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. The process of choosing a suitable plan from all alternatives is known as query optimization. The basis of this choice are a cost model and statistics over the data. Essential for the costs of a plan is the execution order of join operations in its operator tree, since the runtime of plans with different join orders can vary by several orders of magnitude.

1.1 Transformation-based Query Optimizers

Long ago, Ono and Lohman[1] gave a lower bound of $O(3^n)$ with n the number of relations, on the complexity join enumeration, by counting how many join operators are considered in the bottom-up dynamic programming enumeration algorithm of System R and Starburst. Bottom-up dynamic programming and top-down memoization based join enumeration algorithms have been found which achieve this lower bound and hence are very efficient for join enumeration. However there is advantage to reordering other operators and choosing among the new alternatives based on cost estimation. DP based techniques have been extended to deal with other operators (eg: aggregations, outer joins), however there is no general technique for extension. For example, to deal with with sub-queries Seshadri et al suggested a clever iterative algorithm in which an extended bottom-up enumeration module is called many times - but we are forced beyond bottom-up framework.

Rule-based query optimizers are extensible as they are described using a set of transformation rules. All possible rules are applied on each alternative until no new information is produced. However the trade-off of rule-based optimizers is efficiency.

1.2 The Search Space

An exhaustive search for an optimal solution over all possible operator trees is computationally expensive. To decrease complexity, the search space must be restricted. For the optimization problem discussed in this report, a well-accepted heuristic is applied: We consider all possible bushy join trees, but exclude cross products from the search, presuming that all considered queries span a connected query graph [1]. Our goals are:

- Complete space enumeration
- Remain in the space of cross product free join trees
- Avoid generation of duplicates

The goal is to enable cost-based selection of alternatives that are generated via transformation rules, while being as efficient as top-down join enumerator.

1.3 Organization of the Report

The rest of the report is organized as follows: In Chapter 2, we introduce general concepts in traditional query optimization. In Chapter 3, a literature survey of different join enumeration algorithms is presented. In Chapter 4, we prove the in-effectiveness of existing rule sets. In Chapter 5, we present a new rule set which improves on the drawbacks of existing rule sets. In Chapter 6, we present a new method for join enumeration in a rule-based setting. In Chapter 7, we look into possible future steps in this direction. Finally we conclude the report in Chapter 8.

Chapter 2

Query Optimization Overview

In this chapter, we describe the design of cost-based transformational query optimizer, based on Volcano Framework.

Query optimization is probably the most critical component of modern database systems, since the ubiquity of the database system depends on the efficiency with which a user's queries are executed by the system. As queries submitted to the database system by the user using a high-level declarative language do not specify an execution plan, it is the job of the optimizer to generate a good plan, and query optimization can achieve substantial improvements in runtime and resource usage during execution of the query.

Traditionally, query optimizers follow a three step approach [2]:

- Generate all execution plans that are logically equivalent to the user-submitted query.
- Estimate the cost of executing each of the alternate plans.
- Search through the space of all generated plans to find the plan with least cost.

2.1 Space of Logically Equivalent Plans

All plans that compute the same result, having the same properties, as that computed by the original query plan, are called to be logically equivalent to the given query.

The parser parses the query to generate a query tree. From the initial query tree, the space of all logically equivalent plans can be generated using transformation rules.

2.1.1 Logical Plan Space

The initial query tree consists of logical operators which are like the relational operators select, project, join, etc, and which do not specify the physical algorithm to be used during actual computation. All logical plans that are equivalent to the initial query plan constitute the logical plan space of the query.

The logical plan space can be generated by applying logical transformation rules to operators in the logical plan of the query. The new plans generated are further transformed whenever possible, until no new plans can be generated using the given set of rules.

2.1.2 Physical Plan Space

The logical plan must be converted to a physical plan than can be executed on the database system. The implementation rules define which logical operator may be implemented using which physical algorithm.

The physical plan space is generated by applying the implementation rules to all plans in the logical plan space.

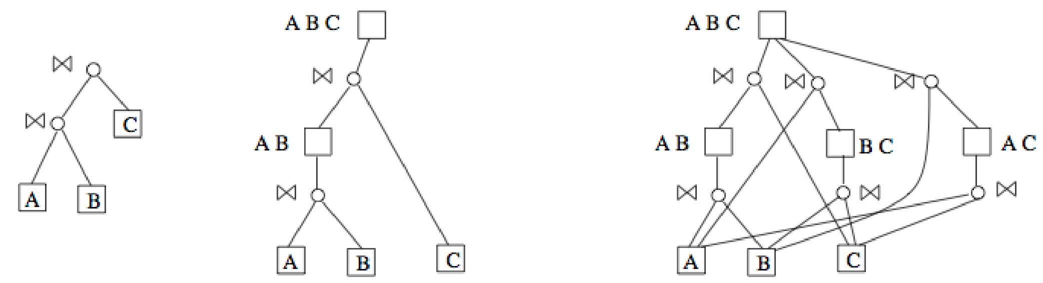
2.2 Logical Space Generation

We describe the generation of logical plan space in greater detail. The logical plan space is the set of all semantically equivalent logical plans of the input query.

Definition 2.1. A Logical Query DAG is a directed acyclic graph whose nodes can be divided into equivalence nodes and operation nodes; the equivalence nodes have only operation nodes as children and operation nodes have only equivalence nodes as children.

Given a set of rules, we construct a Logical Query DAG from the given input query and expand the DAG to encompass the space of all equivalent logical plans. For example, the query tree of Figure 2.1(a) for the query ($A \bowtie B \bowtie C$) is initially represented in the LQDAG formulation, as shown in Figure 2.1(b). The equivalence nodes are shown as boxes, while the operation nodes are shown as circles.

The initial LQDAG is then expanded by applying all possible transformations on every node of the initial LQDAG representing the given query. In the example, suppose the only transformations possible are join associativity and commutativity. 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 LQDAG of Figure 2.1(b). These are represented in the LQDAG shown in Figure 2.1(c). For exact algorithm refer to pseudocode of procedure ExpandDAG presented in Figure 2.2.



(a) Initial Query (b) DAG representation of query (c) Expanded DAG after transformations

Figure 2.1: Logical Plan Space Generation for $A \bowtie B \bowtie C$

2.3 Cost Model

The cost of executing a particular physical plan must be estimated for the purpose of finding the optimal plan. As the actual cost is not available during optimization, it is imperative for the estimates to be close to the actual costs in order to find the best plan.

The cost model defines how a cost is to be estimated for a particular operator when executed on the database. The optimizer makes use of database statistics, like histograms, and various other rules, for estimating the cost.

```
Procedure EXPANDDAG
Input:    $eq$, the root equivalence node for the initial $LQDAG$
Output:   The expanded $LQDAG$
Begin
    for each unexpanded logical operation node $op \in child(eq)$
        for each $inpEq \in input(op)$
            EXPANDDAG($inpEq$)
        apply all possible logical transformations to $op$
            /* may create new equivalence nodes */
        for each resulting logical expression $E$
            if $E \notin LQDAG$
                add $E$'s root operation node to $child(eq)$
            else if the previous instance $E' \in eq'$ where $eq' \not\equiv eq$
                unify $eq'$ with $eq$ /* may trigger further unifications */
        mark $op$ as expanded
End
```

Figure 2.2: Algorithm for LQDAG Generation

2.4 Search for Optimal Plan

The search strategy forms an important part of the optimization scheme. As there are exponentially many alternative plans to be considered, it is infeasible to enumerate all the plans and pick the one with the lowest cost. The optimizer must incorporate pruning techniques to reduce the size of the search space that will be explored during optimization.

Chapter 3

Related Work

In this chapter we describe prior work in the area of finding optimal join order.

3.1 TreeOpt Algorithm

Some of the earliest work on finding optimal join order was done by Ibraraki and Kameda[3]. They describes an algorithm for finding the optimal nesting order for tree queries ie: queries whose join graphs form a tree for nested loop join. Finding a solution for general graphs is NP-Hard, however for this special case it is possible to find the optimal solution in polynomial time.

A cost function f is said to satisfy the *adjacent sequence interchange* property (ASI property in short) if there exists a rank function r_f such that $f(AS_1S_2B) <= f(AS_1S_2B)$ if and only if $r_f(S_1) <= r_f(S_2)$. It is known that if a cost function f satisfies the ASI property, an optimal sequence can be found in $O(nlog(n))$ where the order constraints are series parallel. In our case, the order constraints are directed tree which is a special case of series parallel constraints.

Refer to Algorithm 1. Set one of the nodes in tree as root R, call TreeOpt (T, R). Every node in the tree has a rank given by rank function r_f. Until we reduce the tree to a chain, find a wedge in the tree and reduce it a chain where the nodes are listed in the order of their rank. However there might be ordering constraints within in a chain. The subroutine Normalize in called on each chain prior to reduction which combines nodes S_i and Sj in the chain if $r_f(S_i) > r_f(S_j)$ and S_i occurs before S_j, a new label S_k is assigned for combined entity containing $S_i \cup S_j$ and its rank is recalculated.

TreeOpt is called n times, each time with different node as root. In the end we have an optimal tree for each node as root. The tree with the lowest cost among these is indeed the optimal tree. Complexity : $O(n^2 log(n))$.

Algorithm 1 Finding Optimal Nesting Order for Tree Queries

```
1: function TreeOpt($T, R$)
2:     $rt \leftarrow$ directed tree rooted at R
3:     while $rt$ is not a chain do
4:         Find a wedge in the tree $rt$
5:         Apply Normalize to each chain of the wedge.
6:         Merge the two chains into one by ordering the nodes by their
   ranks.
7:     end while
8: end function
```

For optimizing cyclic join graphs the strategy is to first transform the graph into an acyclic join graph before running TreeOpt. This acyclic join graph needs to chosen with care. Note that when ordering the nodes, the goal was to reduce the number of page fetches. This in nested loop join means keeping size of intermediary relation low. Hence one good candidate for the tree is Minimum Spanning Tree with edge weights as $|R_i||R_j|\sigma_{R_iR_j}$. However it is shown experimentally that this method is not reliable.

3.2 Heuristic for Cyclic Queries

TreeOpt generates optimal plan for any acyclic join graph in polynomial time. However for cyclic queries, it does a bad job at even producing a near optimal plan. Raghavan et al. [4] propose a new algorithm called FAB (Forward and Backward) algorithm for producing better plans for cyclic join graph in polynomial time.

FAB approach uses a global-impact based ordering, in which the global impact of each relation explored and used to decide the ordering. The global impact of a relation R_i is the product of the size of all other relations and join selectivity not related to R_i. Intuitively if node R_i has least impact, it means that it itself has a combination of large size and low join selectivity. The FAB algorithm picks the node with least impact and puts it as the last node to be joined. It then recalculates the impact of each node after removing the element from the join graph.

This approach works quite well for cyclic join graphs but note it not guaranteed to produce optimal plan always. The paper proposes a query aware approach where given a query, we try to infer if the join graph is linear, acyclic or cyclic. For the first two, use the TreeOpt algorithm and for third use FAB.

Note that FAB and TreeOpt are both used to generate left deep join trees. They do not help in any way to find the optimal solution in the space of all bushy join trees.

3.3 DP and Memoization based join enumeration

Two approaches have done well at exploring and finding the optimal plan in the space of all bushy join orderings : bottom-up join enumeration via dynamic programming and top-down join enumeration via memoization. Moerkotte and Neumann [5] presented an efficient dynamic programming based algorithm (DPccp). DeHaan and Tompa [6] proposed an efficient top-down algorithm (TDMinCutLazy). Fender and Moerkotte [7] proposed an alternative top-down join enumeration strategy (TDMinCutBranch), which is almost as efficient as DPccp. In the following year, Fender et al.[8] proposed another top-down enumeration strategy TDMinCutConservative which is currently the best.

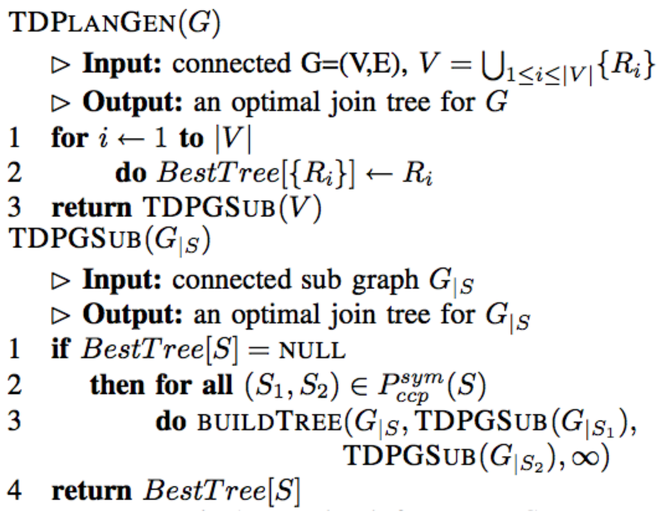
3.3.1 Generic Top-Down Enumeration

Top-Down join enumeration has a clear advantage over Bottom-Up style as it enables pruning. We describe a generic top-down enumeration via TDPlanGen (see pseudocode in Figure 3.1). Like in dynamic programming, TDPlanGen initializes the building blocks for atomic relations first (line 2). In line 3, the subroutine TDPGSub is called, which traverses recursively through the search space. At each invocation of TDPGSub, a connected subgraph $G_{|S}$ is given. 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 condition 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 $P_{ccp}^{sym}(S)$. A subroutine BuildTree is called which recursively calls TDPGSub on subgraph induced by S_1 and subgraph induced by S_2. The recursive decent stops when $|S| = 1$ or the best tree is already known. Once all the possibilities have been tried. BuildTree updates memotable *BestTree* if a plan better than existing plan is found. After all $(S_1, S_2) \in P_{ccp}^{sym}(S)$ have been tried, the entry in $BestTree[S]$ corresponds the optimal join tree for set of relations in S.



```
TDPlanGen($G$)
    ▷ Input: connected G=(V,E), $V = \bigcup_{1 \le i \le |V|} \{R_i\}$
    ▷ Output: an optimal join tree for $G$
1   for $i \leftarrow 1$ to $|V|$
2         do $BestTree[\{R_i\}] \leftarrow R_i$
3   return TDPGSub($V$)
TDPGSub($G_{|S}$)
    ▷ Input: connected sub graph $G_{|S}$
    ▷ Output: an optimal join tree for $G_{|S}$
1   if $BestTree[S] =$ NULL
2      then for all $(S_1, S_2) \in P_{ccp}^{sym}(S)$
3             do buildTree($G_{|S}$, TDPGSub($G_{|S_1}$),
                                TDPGSub($G_{|S_2}$), $\infty$)
4   return $BestTree[S]$
```

Figure 3.1: Pseudocode for TDPLANGEN

Depending on the choice of partitioning algorithm, the overall performance of TDPlanGen can vary by orders of magnitude. Since TDPlanGen is top-down we can easily apply existing pruning techniques to prevent exploration of entire sub-trees. Later in section 6.1 we describe the MinCutConservative partitioning algorithm and how we can use it in transformational setting.

Chapter 4

Effectiveness of Existing Rule Sets

This chapter considers different rule sets used for join enumeration in transformation based query optimization using Volcano/Cascades framework and analyse them for:

- Completeness : Ability to generate the space of all cross-product free join orders starting from an initial plan without cross products.
- Efficiency : Number of duplicates generated in the process of exploring the space. The lower the better.

We start with a basic rule set consisting of commutativity and associativity called rule set RS-B0. RS-B0 consists of redundant rules and this redundancy is addressed in rule set RS-B1. Pellenkoft et al.[9] showed that using RS-B1 leads to the generation of an exponential number of duplicates. They presented a new rule set RS-B2 which uses a new rule 'Exchange' in conjunction with RS-B0 and some added constraints to generate all alternate bushy join trees without generating duplicates. However we show that RS-B2 achieves this by generating trees outside of search space and a restriction to remain in search space leads to incompleteness.

Definition 4.1. 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 presence of join predicate between them.

Definition 4.2. A Query Tree Q is a bushy plan showing the way the relations will be joined together.

4.1 Rule set RS-B0

The simplest set of rules used (to generate the bushy space) are :

- 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
  $A \bowtie B \rightarrow B \bowtie A$

4.1.1 Efficiency

The set is redundant because we can drop Left (or Right) Associativity and still generate the same space. This redundancy is addressed in rule set RS-B1. The number of duplicates is atleast as many as RS-B1.

4.1.2 Completeness

We show later in section 4.2.2 that a subset of these rules is sufficient to enumerate the entire space.

4.2 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
  $A \bowtie B \rightarrow B \bowtie A$

4.2.1 Efficiency

Pellenkoft et al. [9] have shown that RS-B1 leads to exponential number of duplicates. The number of duplicates generated during the construction of MEMO-structure encoding all bushy trees for a clique join graph on n relations is $4^n - 3^{n+1} + 2^{n+2} - n - 2$.

4.2.2 Completeness

We show that this minimal set is sufficient to enumerate the cross product free space assuming the query is fully connected.

Problem Description : Given a connected join graph J and query tree Q, show that using RS-B1 we can reach any other valid query tree Q' from Q while remaining in cross-product free space.

We assume that Q itself does not have any cross products ie: Q belongs to search space. Given the input join graph J is completely connected we can number the nodes $1, 2...n$ such that $(..(R_1 \bowtie R_2)...) \bowtie Rn$ is a valid left-deep join tree. To prove the problem statement, we prove a stronger claim :

Claim 1 : Any query tree can be transformed into left-deep join tree and left-deep join tree can be transformed into any query tree in the search space. Hence to go from query tree Q_1 to Q_2, there exists atleast one route ie: Q_1 to left-deep tree to Q_2

Let us break the process on conversion from one tree to another into steps. In step 1 we move R_n to the top so that it is last relation to be joined and then work on remaining set of relations numbered $1, 2, .., n-1$. Generalized in step i, we move relation R_{n-i+1} to the top and recurse on remaining relations 1 ... n-i. The decent stops when $i = n-1$. This brings us to Claim 2.

Claim 2: Having a query tree with relations $R_1, R_2, ...R_k$, it is possible to build tree $[R_1R_2..R_{k-1}] \bowtie R_k$ where $[R_1R_2..R_{k-1}]$ represents a smaller join tree with relations $R_1, R_2, ..R_{k-1}$

Now the goal is given an query graph, move the highest numbered relation to the top so that its the last relation to be joined. Let k be the highest numbered relation and 1 .. k-1 are the other relations in the graph. View the tree as in Figure 4.1 where $C_1, C_2, ...C_m$ are equivalence classes consisting of some number of relations. We say the height of the tree is m+1. By induction on tree height, we show that R_k can be brought to top while remaining in cross-product free space.

Base Cases: When m = 0 , there is nothing to do, just repeat the procedure on X. When m = 1, X and C_1 should have a join predicate between them (because R_k is the last relation to be join in left-deep tree). Applying left associativity we get required tree.

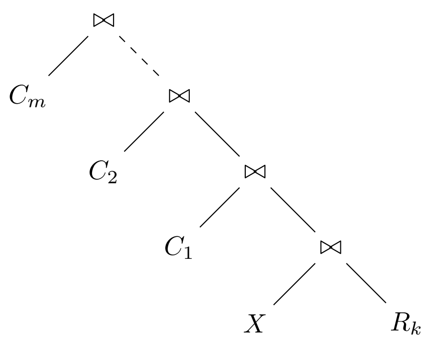


Figure 4.1: Reduced Query Tree

Induction on height m: Assume claim 2 holds for height $m <= j$. For height $m = j + 1$, we know that since R_k was the last relation to be joined in the left-deep tree, X can be joined with atleast one of the other C's say C_i. Pull C_i down to X using a sequence of $C \rightarrow LA \rightarrow C$. Finally use Left Associativity to reduce height to j which is solvable. Hence Claim 2 proved, we can convert any tree into left-deep tree chosen earlier.

The last thing remaining to be shown is how we can go from left-deep tree to any other join tree. It is not difficult to see that this is equivalent to have the rules right associativity and commutativity and going from the join tree to left-deep tree. Right associativity can be done using a sequence of left associativity and commutativity : $RA \Rightarrow C \rightarrow C \rightarrow LA \rightarrow C$. We can use the same arguments as above to complete the proof.

4.3 Rule set RS-B2

Pellenkoft et al. [9] presented a modified rule-set which generates the space of all bushy join trees with no duplicates. The rule set :

- R_1 : Commutativity
  $A \bowtie_0 B \rightarrow B \bowtie_1 A$
  Disable R_1, R_2, R_3, R_4 for application 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 R_2, R_3, R_4 for application 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 R_2, R_3, R_4 for application on new operator $\bowtie_2$

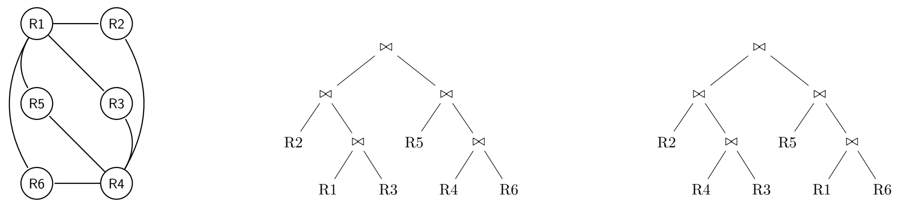
- R_4 : Exchange
  $(A \bowtie_0 B) \bowtie_1 (C \bowtie_2 D) \rightarrow (A \bowtie_3 C) \bowtie_4 (B \bowtie_5 D)$
  Disable R_1, R_2, R_3, R_4 for application on new operator $\bowtie_4$

It is interesting to visualize the way exploration happens using RS-B2. Start at the topmost join operator, its left and right subtrees are completely explored. All the rules are applied at the top join operator and it gets locked. Now we explore all the newly generated join trees. In a way the locking happens one round from bottom to top, exchanges happen at the top(root) join operator and the root operator gets locked. Finally there is another round of locking happens this time from the top downwards.

4.3.1 Efficiency

Pellenkoft et al. have showed that this scheme does not generate any duplicates and achieves the lower bound given by Ono and Lohman.

4.3.2 Completeness



(a) Join Graph J (b) Query Tree Q (c) Query Tree Q_2

RS-B2 cannot generate the space of all bushy trees without cross products. Given a set of relations to be joined with join graph J (Figure 4.2a) and initial query tree Q (Figure 4.2b), observe that swapping R1 and R4 will lead to a valid query tree Q_2 (Figure 4.2c) which does not have any cross product. Hence Q_2 belongs to search space. However going from Q to Q_2 would require formation of sub-tree R_2,R_3 which will lead to cross product.

Chapter 5

Ruleset RS-B3

In this chapter we address the drawbacks of RS-B1 and ineffectiveness of RS-B2 with our new rule-set RS-B3 which explores the space of cross product free query trees generating order of magnitude lesser duplicates. Section 5.1 describes the rule-set and the locking details. Section 5.2 proves the ability of rule-set to explore the entire space. Section 5.3 shows the effectiveness of the new rule-set in terms of number of duplicates generated.

5.1 Rule-set RS-B3

Given an initial Query Tree Q, for all join nodes enable only R_1 and R_2.

- R_1: Commutativity
  $A \bowtie_0 B \rightarrow B \bowtie_1 A$
  Disable R_1,R_2 for application on new operator $\bowtie_1$

- R_2: Right Associativity
  $(A \bowtie_0 B) \bowtie_1 C \rightarrow A \bowtie_2 (B \bowtie_3 C)$
  Disable R_1,R_2 for application on new operator $\bowtie_2$. Enable R_3 for application on new operator $\bowtie_2$.

- R_3: Left Associativity
  $A \bowtie_0 (B \bowtie_1 C) \rightarrow (A \bowtie_2 B) \bowtie_3 C$
  Enable R_1 for application on new operator $\bowtie_3$.

5.2 Completeness

This method of enumeration is not complete. There exist edge cases. Consider for example join graph as shown in figure 5.1. Let the initial element be $[R_1R_2R_3R_4] \bowtie [R_5R_6R_7R_8]$ and final element $[R_1R_2R_5R_6] \bowtie [R_3R_4R_7R_8]$. In the current scheme, it is not possible to do this. Also note that no locking constrains while restricting the number of times group of relations move around the join to 2 can achieve this. However the ray of hope is that using a small perturbation ie: moving R_4 to right hand side and then starting the enumeration will work.

Note that the case that broke RS-B2 (see Figure **??**) can actually be explored by using RS-B3. The sequence of moves would be $[R_1R_3]$ moves across the top join operator using right associativity, this new equivalence class is explored and $[R_4R_3]$ brought back to left side using left associativity.

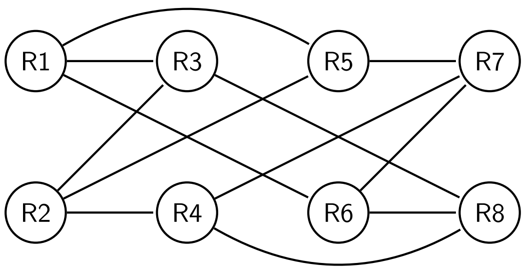


Figure 5.1: Extreme case for RS-B3

5.3 Number of Duplicates Generated

Claim 1: The number of duplicates generated by **RS-B3** during the exploration of a class that combines k relations, on a completely connected graph is: $O(3^l 2^r)$

Proof: In a fully explored class that combines n relations the number of join operators is $A(n) = 2^n - 2$. Consider a class that combines k relations. Using an initial element $[L] \bowtie [R]$ with l the number of relations in $[L]$ and $k - l$ the number of relations in R, the transformation rules generate the following elements. Rule R_1 generates mirror image of original operator. Rule R_2 combines every element of $[L]$ with $[R]$. Note the previous step enables rule R_3. Now rule R_3 combines every element of this newly generated $[R']$ with the leftover of L. Counting the number of alternatives generates :

$$C = \sum_{k=1}^{l-1} \binom{l}{k} A(k+r)$$

$$\begin{aligned} &= \sum_{k=1}^{l-1} \binom{l}{k} (2^{k+r} - 2) \\ &= 2^r(3^l - 2^l - 1) - 2(2^l - 2) \end{aligned}$$

Application of rule R_3 enables commutativity which generates mirror images. To count the number of duplicates generated at this step add all the newly created operator and the initial operators and subtract $A(k)$:

$$\begin{aligned} D &= 2 + 2 * C - A(k) \\ &= 2 + 2^r(3^l - 2^l - 1) - 2(2^l - 2) - (2^k - 2) \\ &= 3^l 2^r - 2^{k+1} - 2^r - 2^{l+1} + 8 \end{aligned}$$

Chapter 6

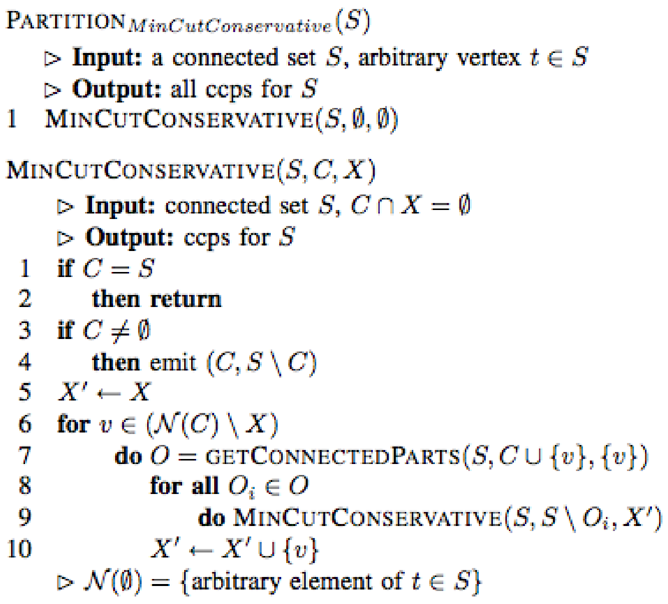
Embedding Top-Down Join Enumerator

In this chapter we propose a new method of doing join enumeration in transformation-based optimizers based on ideas from memoization based join enumerators. DP-based and memoization-based enumeration algorithms[7, 8, 10] for join enumeration are more efficient in comparison to transformation-based ones in that they are able to enumerate the entire space without generating duplicates. In Section 6.1 we describe the algorithm, Section 6.2 describes the process for logical space generation and in Section 6.3 we describe a scheme for combining the two.

6.1 Conservative Partitioning

We describe in greater detail the conservative partitioning algorithm, denoted by MinCutConservative given by Pit Fender et al.[8]. The conservative partitioning algorithm is used in graph based join enumeration algorithm TDMinCutConservative. We try to use it here in a transformation based setting.

The algorithm emits all ccps for a connected vertex set $S \subseteq V$ where V is the vertex set of the query graph G=(V,E). The basic idea of conservative partitioning is to successively enhance a connected set C by members of its neighbourhood $N(C)$ at every recursive iteration. The process starts with a single vertex $t \in S$ through a redefinition of $N(\phi) = t$. This way, we ensure that at every instance of the algorithm's execution C is connected. Since S and $C \subset S$ are connected, for every possible complement $S \setminus C$ there must exist a join edge $(v1, v2)$, where $v1 \in C$ and $v2 \in (S \setminus C)$ holds. If at some point of enlarging C its complement $S \setminus C$ in S is connected as well, the algorithm has found a ccp for S.



```
PARTITION_{MinCutConservative}($S$)
  ▷ Input: a connected set $S$, arbitrary vertex $t \in S$
  ▷ Output: all ccps for $S$
  MINCUTCONSERVATIVE($S, \emptyset, \emptyset$)

MINCUTCONSERVATIVE($S, C, X$)
  ▷ Input: connected set $S$, $C \cap X = \emptyset$
  ▷ Output: ccps for $S$
  if $C = S$
     then return
  if $C \neq \emptyset$
     then emit $(C, S \setminus C)$
  $X' \leftarrow X$
  for $v \in (\mathcal{N}(C) \setminus X)$
       do $O$ = GETCONNECTEDPARTS($S, C \cup \{v\}, \{v\}$)
          for all $O_i \in O$
               do MINCUTCONSERVATIVE($S, S \setminus O_i, X'$)
          $X' \leftarrow X' \cup \{v\}$
  ▷ $\mathcal{N}(\emptyset) = \{$arbitrary element of $t \in S\}$
```

Figure 6.1: Pseudocode for MINCUTCONSERVATIVE

MINCUTCONSERVATIVE performs well even on when star queries are considered, where constructing every possible connected subset C of S produces an exponential overhead because most of the complements $S \setminus C$ are not connected and the partitions $(C, S \setminus C)$ computed this way are not valid ccps. Since the set C is expanded conservatively, this method remains effective. For pseudocode of the algorithm see Figure 6.1.

6.2 Using Graph based enumeration

We replace the existing join enumeration rules for commutativity and left associativity with a new N-ARY JOIN rule. The N-ARY JOIN rule generates all ccps for the given set of n-relations. It is in essence a vanilla coated MINCUTCONSERVATIVE algorithm.

Definition 6.1. A base equivalence class is defined as an equivalence class which is either a base relation or has atleast one child logical operation node which is not a join operator.

To run the MinCutConservative algorithm, we need the join graph induced by the base equivalence classes. For this we store two additional sets at each equivalence node:

- EqSet : A set of all combinations of base equivalence classes being joined to generate this equivalence class. Along with this we need to store the ids of join operators of a feasible join ordering for these base equivalence class. This is used to infer the join graph.
- HSet : A set of hashes. For every set of base equivalence class that we enumerate, we generate a hash and store it in the parent equivalence class as a proof that this set has been enumerated. This is to prevent repeated enumeration of the same set of base equivalence classes.

Given with the above information, when N-ary Join rule is applied at a logical join operator *op* get the EqSet of the left child and right child. For all combination of $S = S_1 \cup S_2$ where $S_1 \in \text{EqSet}_L$ and $S_2 \in \text{EqSet}_R$, generate $hash(S)$ and check to ensure it has not been enumerated before. It it has not been enumerated, call MinCutConservative($G_{|S}$). For every ccp generated add a new operation node to $child(parent(op))$. For every new equivalence class generated, insert it as a left-deep tree got by doing a depth-first traversal on the join graph (such a tree exists as the graph induced by these base equivalence classes is connected).

Chapter 7

Future Work

Over the course of BTP, we have identified a number of promising areas related to query optimization in transformation-based query optimizers for future work :

- **Memoization based Join Enumeration** : We plan to experiment with memoization-based join enumerator as a replacement for existing join enumeration rules (see Section 6). The benefits and overheads need to be analysed.
- **Throw in more join operators** : We plan to analyse effect of adding in more operators like left outer join, full outer join, anti join, semi join and group joins into the bucket of join operators. Even with this extended family of join operators we aim at exploring the space of all valid plans without introducing cross products.
- **Sampling the Search Space** : The search space grows exponentially with respect to number of relations. The idea is to generate a sample of the search space and from this sample give the optimal plan. Galindo-Legaria et al. [11] presented a method for uniform random sample of all join orders. The problems we plan to tackle :
  - Random sample of space of bushy join trees
  - Extending to generating random sample of search space even in presence of other operators
- **Prioritized Search Space Exploration** : Currently the search space is explored depth first. This approach is simple, however has inherent problem that we don't assign priority for traversal of children. Pruning depends on the order of exploration and hence it might be beneficial

to assign 'priority' to each of the children and choose order based on this. Some ideas in [8].

- **Extending to Cascades and Columbia**: The Cascades optimizer generator overcomes many of the shortcomings of Volcano. We wish to check the applicability of things discussed in this report for use in Cascades style framework.

Chapter 8

Conclusion

In this report we presented :

- An overview of query optimization process
- Literature survey of join enumeration techniques used
- Proofs of effectiveness/ineffectiveness of existing rule sets in exploring the space of all valid join trees without cross products
- New rule set RS-B3 for join enumeration
- New technique for complete enumeration without duplicates

Bibliography

[1] Kiyoshi Ono and Guy M Lohman. “Measuring the Complexity of Join Enumeration in Query Optimization.” In: *VLDB*. 1990, pp. 314–325.

[2] Prasan Roy et al. “Efficient and extensible algorithms for multi query optimization”. In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 249–260.

[3] Toshihide Ibaraki and Tiko Kameda. “On the optimal nesting order for computing N-relational joins”. In: *ACM Transactions on Database Systems (TODS)* 9.3 (1984), pp. 482–502.

[4] Venkatesh Raghavan et al. “Multi-join continuous query optimization: Covering the spectrum of linear, acyclic, and cyclic queries”. In: *Dataspace: The Final Frontier*. Springer, 2009, pp. 91–106.

[5] Guido Moerkotte and Thomas Neumann. “Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products”. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 930–941.

[6] David DeHaan and Frank Wm Tompa. “Optimal top-down join enumeration”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 785–796.

[7] Pit Fender and Guido Moerkotte. “A new, highly efficient, and easy to implement top-down join enumeration algorithm”. In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 864–875.

[8] Pit Fender et al. “Effective and Robust Pruning for Top-Down Join Enumeration Algorithms”. In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE. 2012, pp. 414–425.

[9] Arjan Pellenkoft, César A Galindo-Legaria, and Martin Kersten. “The complexity of transformation-based join enumeration”. In: *VLDB*. 1997, pp. 306–315.

[10] Pit Fender and Guido Moerkotte. “Top down plan generation: From theory to practice”. In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pp. 1105–1116.

[11] César A Galindo-Legaria, Arjan Pellenkoft, and Martin L Kersten. “Uniformly-distributed random generation of join orders”. In: *Database TheoryICDT’95*. Springer, 1995, pp. 280–293.