RS-Constrained Graph rule for Join Order Enumeration in Query Optimizers

B.Tech. Project Report
by
Karan Vaidya (130050019)

under the guidance of
Prof. S. Sudarshan

Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai 400 076
Abstract

Transformation-based query optimizers designed for the Volcano/Cascades framework are used heavily in commercial databases. Join-order enumeration techniques used in these optimizers may lead to enumeration of joins with cross-products or duplicate derivations, both of which are undesirable. The RS-graph rule is a top-down join-order enumeration technique for the Volcano/Cascades framework which generates cross-product free join-orders. Even the RS-graph rule might partition the join graph more than required. We suggest a modification to it RS-ConstrainedGraph rule that handles these cases more efficiently where duplicate operations might occur in RS-Graph rule.
I wish to sincerely thank my guide Dr. S. Sudarshan, for his guidance and unwa- 
vering support, without whom this endeavour would have remained fruitless. 
I also wish to express my gratitude to my friend and teammate Sivaprasad S, for 
his constant support throughout the project.

Karan Vaidya (130050019)  
IIT Bombay
Contents

Abstract i
Acknowledgments ii
1 Introduction 1
2 Background 3
  2.1 RS-graph rule .............................................. 4
  2.1.1 Definitions ............................................. 5
  2.1.2 Algorithm ............................................. 6
3 Motivation for RS-Constrained Graph 7
4 Algorithm 9
  4.1 RS-Constrained Graph ..................................... 9
5 Proof of Correctness 13
  5.1 RS Constrained Graph .................................... 13
  5.2 Constrained Partition .................................... 14
6 Time Complexity Comparison 15
  6.0.1 RS-graph .............................................. 15
  6.0.2 RS-B1 .................................................. 16
  6.0.3 RS-Constrained graph .................................. 16
  6.1 Constraint-free Operator push-down ..................... 17
  6.2 Query ..................................................... 17
    6.2.1 RS-Constrained graph ................................ 17
    6.2.2 RS-graph ............................................ 18
    6.2.3 RS-B1 .............................................. 19
    6.2.4 Analysis ............................................. 20
6.3 Query ................................................................. 20
  6.3.1 RS-Constrain graph ............................................. 20
  6.3.2 RS-graph ........................................................ 21
  6.3.3 RS-B1 ........................................................... 22
  6.3.4 Analysis ........................................................ 22

7 Conclusion and Future Work ........................................ 23
Chapter 1

Introduction

Query optimization in relational database systems has been studied for a long time and still remains a field of interest. Particularly, the problem of finding an optimal join order has seen the use of many techniques. Join order plans with cross products may lead to inefficiency, and hence are to be avoided. This adds a further constraint to join order enumerations.

Techniques for enumeration can be classified broadly into two categories. Bottom-up enumeration using dynamic programming and Top-down enumeration using memoization. Top-down enumeration allow cost-based pruning at the time of generation. For example, the TDMinCutConservative strategy by [Fen+12] uses top-down enumeration.

An exhaustive search for an optimal solution over all possible operator trees is computationally expensive. Our goals of join enumeration are:

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

If we are to generate cross-product free join orders in an extensible framework, most transformation rules face the problem of duplicate derivations. Generation of these duplicates leads to increase in the time complexity. For example, associativity/commutativity rules such as the RS-B1 ruleset have a time complexity of $O(4^n)$ for generating bushy join orders with $n$ relations as shown by [PGK97].

The RS-graph rule described in [SS14] for join order enumeration is a also a top-down enumeration rule for the Volcano/Cascades framework. It performs transformations on maximal subtrees. It then uses the partitioning strategies for top-down
enumeration such as [Fen+12] to generate only cross-product free join orders. It is capable of generating join orders without any duplicate derivations for many instances of queries and transformation rules. However, they may lead to excess computation if there are multiple maximal subtrees for a query. These multiple maximal join trees are a result of the interaction of transformation rules involving aggregation, semi-join etc. with the query.

This report makes the following contributions -

- We try to address the cases in which the RS-graph rule does not perform optimally by suggesting changes to the technique

- We propose an improved algorithm called RS-ConstrainedGraph rule that is more efficient than the traditional RS-Graph rule. The new algorithm avoids repetitive generation of same partitions by remembering extra information in the form of constraints.

- We then establish the correctness and completeness of the new algorithm for a class of rulesets.

- We also analyze the performance of RS-ConstrainedGraph rule against RS-Graph and RS-B1 rule.
Chapter 2

Background

Given an input query, the query optimiser can work in three different steps. Firstly generate the semantically equivalent rewritings of the query. One can apply various transformation rules to a given query to get other logically equivalent forms in the logical plan space. This needs to be efficiently done and compactly stored. A logical query can be expressed as a directed acyclic graph whose nodes can be equivalence nodes or operator nodes. The logical space can be generated by expanding on this DAG.

Second, generate the physical plans corresponding to each query generated in first step. Given a logical query, there could be multiple execution plans. There could be different algorithms to execute the plan. It could also be needed that various properties be enforced at different levels in the plan. The physical operation nodes can be algorithms for computing the logical operations or enforcers that enforce the required physical property.

Finally, we have to search the plan space generated in the second step for the best plan. Given the cost of the operations, we need to find the plan with the minimum cost. This can be done efficiently by top-down memoization based approach or bottom-up dynamic programming. As the search space can be huge, this can be coupled with bounding and pruning.

Top-down process of exploring an equivalence node applies all possible rules to the children of the node before considering the node itself. That is, the children of a node are explored fully before it.

A query can be expressed as an AND-OR DAG which is a directed acyclic graph with two types of nodes -
• AND-nodes/Operator nodes: Represent relational algebra operators such as join operation ($\bowtie$) and projection operation ($\Pi$)

• OR-nodes/Equivalence nodes: Represent equivalence nodes which are sets of logical expressions that generate the same result set. The set of such expressions can be described by their child nodes in the AND-OR DAG.

Clearly, the AND-nodes can have only OR-nodes as children and OR-nodes have only AND-nodes as children.

We can then apply logical transformations on the AND-OR DAG to generate all logically equivalent forms of the query. This expanded AND-OR DAG is called as the Logical Query DAG (LQDAG). An LQDAG has the entire logically equivalent space of the query. Physical space estimation and pruning is then done on this DAG.

### 2.1 RS-graph rule

The RS-graph rule by [SS14] is an extension to the Volcano LQDAG algorithm which generates and maintains the sets of equivalence nodes at the leaves of maximal join trees (which we call join-sets) under each equivalence node.
2.1.1 Definitions

- A base equivalence node in an expanded LQ-DAG is an equivalence node that has no join operator as a child. It can be either a base relation, or it must have only non-join operators as children.

- A join equivalence node in an expanded LQ-DAG is an equivalence node such that at least one of its child operator nodes is a join operator.

- 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.

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

- 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.
• 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.

2.1.2 Algorithm

We initialize the join-set of a base equivalence B as \((B, \text{null})\). We then execute the following algorithm -

**Algorithm 1** RS-graph rule \((P \leftarrow A \bowtie B)\)

1: \(\text{JoinTrees} \leftarrow []\)
2: \(\text{for } j_A \in A.\text{JoinSets} \text{ do}\)
3:  \(\text{for } j_B \in B.\text{JoinSets} \text{ do}\)
4:  \(j_P \leftarrow \text{Merge}(j_A, j_B)\)
5:  \(\text{if } j_P \notin P.\text{JoinSets} \text{ then}\)
6:  \(P.\text{JoinSets} \leftarrow P.\text{JoinSets} \cup j_P\)
7:  \(\text{Graph} \leftarrow \text{CreateGraph}(j_P)\)
8:  \(\text{Partitions} \leftarrow \text{Partition}(\text{Graph})\)
9:  \(\text{Trees} \leftarrow \text{CreateTrees}(\text{Partitions})\)
10: \(\text{JoinTrees} \leftarrow \text{JoinTrees} \cup \text{Trees}\)

The algorithm utilizes the following functions -

• \(\text{Merge}(j_A, j_B)\) : Given \(j_A = (V_A, \text{Pred}_A)\) and \(j_B = (V_B, \text{Pred}_B)\), \(\text{Merge}(j_A, j_B) = (j_A \cup j_B, \text{Pred}_A \text{ and } \text{Pred}_B)\)

• \(\text{CreateGraph}(j_P)\) : Generates a join graph for the given join-set whose nodes are the base equivalence nodes. An edge between nodes indicates a join predicate between them. Is used to generate cross-product free join orders

• \(\text{Partition}(\text{Graph})\) : Generates disjoint subsets of the join graph such that their union is the entire graph, and they represent connected subgraphs by themselves. The MinCutConservative algorithm in [Fen+12] can generate these subsets efficiently. MinCutConservative generates all the possible partitions uniquely without any repetition and without symmetric counterparts.

• \(\text{CreateTrees}(\text{Partitions})\) : The disjoint subset generated by partitioning represent equivalence nodes having their own join trees. Join of these equivalence node is a new join order for the equivalence node represented by the Parent
Chapter 3

Motivation for RS-Constrained Graph

The RS-graph rule returns the optimal join-order even in the presence of multiple maximal join-sets. However, efficiency in terms of Time Complexity might suffer. When there are multiple maximal join-sets at an equivalence node, separate join-graphs are created for each join-set and are partitioned separately and join-trees are obtained.

Consider a bushy join tree with $n$ relations and an equivalence node $E$ with two maximal join-sets ($\{A, r_1\}, \{\bowtie_1\}$) and ($\{B, r_1\}, \{\bowtie_2\}$) where only one base equivalence node differs. These multiple maximal join-sets will propagate upwards to the root node. The root node will thus have two maximal join-sets with $n+1$ base equivalence nodes and differing in one node. These will be partitioned separately. Many of the partitions of these join-sets will generate the same equivalence nodes. There is some redundancy in this method.

Such maximal join-sets can only occur at the root node of the input tree to these rules. These multiple maximal join-sets would then be propagated upwards. This propagation of multiple maximal join-sets to parent nodes result in redundancy while partitioning these nodes.

The idea to solve this issue will be to just propagate one representative maximal join-set and some other additional information to generate the required equivalent maximal join-sets in the form of *Constraints*. A constraint class represents a set of maximal join sets that are equivalent to each other. A set of constraints can be represented as

$$C = \{A = \{A_0, A_1\}, B = \{B_0, B_1\}, \ldots\}$$
where each $A$ represents a constraint class with $A_0, A_1...$ being equivalent to each other.

Once at an equivalence node we generate multiple maximal join-sets by rule application, we propagate exactly one representative of these and update the set of constraint classes with a new constraint class. The representative can be any valid one among these, for the purpose we can chose one that is in the query, if valid and cross-product free. We catch hold of multiple maximal join sets at the lower most node, for which the LQDAG generation is done before it’s parents, and propagate them as constraints. In that sense the set of constraints is minimal.

Now we present the algorithm to do a complete and efficient join enumeration.
Chapter 4

Algorithm

4.1 RS-Constrained Graph

At every join node \((P \leftarrow A \bowtie B)\) we pass around one representative maximal join-set. We maintain a set of constraint classes \(C\) that has the set of join-sets that we have seen till now and are equivalent to each other. Constraints from \(C\) needs to be applied to the representative maximal join-set to generate the entire logically equivalent space.

Consider the ruleset

\[
\Pi(A \bowtie B) \equiv \Pi(A) \bowtie B \\
A \bowtie \Pi(B) \\
\Pi(A) \bowtie \Pi(B)
\]

If the following ruleset is applicable at a join node and the query has \(\Pi(A) \bowtie \Pi(B)\) represented in it. Then we pass this onto the representative maximal join-set. We add the set \(\{\Pi(A), \Pi(B), \theta_0\}, \{A, \Pi(B), \theta_0\}, \{\Pi(A), B, \theta_0\}\) to the set of constraint classes \(C\). At every join node \(C\) has to be updated as the union of the constraint classes of the children.

**Algorithm 2** Constrained Graph \((P \leftarrow A \bowtie B)\)

1: **procedure** RS-CONSTRAINEDGRAPH
2: \(P.M = A.M \bowtie B.M\)
3: \(P.Constraints = A.Constraints \cup B.Constraints\)
4: \(FullPartition(P.M)\)
5: \(ConstrainedPartition(P.M, P.C)\)
Once you have a representative maximal join-set then a full partition of that will generate the logical space with just the base relations in the representative maximal join-set. The constraints that we have accumulated till now has to applied to the representative to generate the entire logical space rooted at the current equivalence node. Merely applying the constraints to the representative and calling graph partitioning will cause repeated partitioning of the same sub-graph if the base relations of a constraint are on the same side of the partition.

We need to apply them and do graph partitioning efficiently that repeated partitioning of the same sub-graph is avoided. We apply the constraints to the representative in a dfs fashion to generate all the equivalent maximal join-sets and at each step call modified $MinCutConservative$ to generate only relevant partitions where the constrained base relations are in 2 different partitions. While doing dfs at any node representing a maximal join-set $M'$ if any new constraint is being represented in $M'$, say $D_0$, then it, $D$, can be added to the set of applicable constraints $C_1$ for further partitioning. It might not be possible to generate all the partitions without any repetitions in many cases if we have a large number of constraints each of which overlapping with each other. Let

$$C = \{ A = \{ A_0, A_1\ldots \}, B = \{ B_0, B_1\ldots \}, \ldots \}$$

For further discussion we use the elements of each class indexed with 0 ($A_0, B_0\ldots$) to represent the join-set that can be represented in the representative. Let the maximal join set we get by applying the constraints be $M' = \{ op(R_1), op(R_2), op(R_3) \}$ Let

$$A_0 = \{ op(R_1), op(R_2) \}$$

$$A_1 = \{ op(R_1), R_2 \}$$

$$B_0 = \{ R_2, op(R_3) \}$$

$$B_1 = \{ op(R_2), op(R_3) \}$$

The maximal join-set $M'$ has $A_0$ and $B_1$ being represented in it. While doing dfs if we replace $B_1$ by $B_0$, then we’ll end up having $A_1$ instead $A_0$. While doing constrained partition the idea of having the constrained relations on either sides of the partition is based on the assumption that we will have a constraint free equivalent form of the maximal join-set that would have been covered in the $FullPartition$. This is not the case here when there is an overlapping of the base relations between constraints. This can be solved by choosing only a subset of the constraints carefully to minimize the repetition in partitioning.
Algorithm 3 Constrained Partition

1: procedure ConstrainedPartition($M, C$) 
2: \hspace{1em} $C_1 \leftarrow \{c \in C \mid c \text{ is represented in } M\}$ 
3: \hspace{1em} $G \leftarrow \text{ConstraintGraph}(C_1)$ 
4: \hspace{1em} $C \leftarrow \text{MaximumIndependent}(G)$ 
5: \hspace{1em} $dfsConstraintApply(M, C)$ 

We consider only those constraint classes that are represented in the maximal join-set being considered. Further we select only those constraints that when considered doesn’t affect the correctness of the enumeration as above. Consider two constraint classes $A$ and $B$ that are being represented in the maximal join-set. One easy way to avoid it would be to choose them both only if they don’t overlap with each other i.e, for each $a \in A$ and $b \in B$ $a \cap b = \phi$. But we can do better. We can always choose $A$ and $B$ if $a \cap b = a_0 \cap b_0$ as even in this case we can reach a constraint free equivalent maximal join-set by application of rules. This forms the basis of the proof idea discussed later.

Algorithm 4 Generation of Constraint Graph

1: procedure ConstraintGraph($C = \{A_1, A_2...A_n\}$) 
2: \hspace{1em} $G.Nodes \leftarrow \{A_1, A_2...A_n\}$ 
3: \hspace{1em} for $A = \{a_0, a_1...a_k\}$ in $C$ do 
4: \hspace{2em} for $B = \{b_0, b_1...b_l\}$ in $C$ do 
5: \hspace{3em} for $a$ in $\{a_1...a_k\}$ do 
6: \hspace{4em} for $b$ in $\{b_1...b_l\}$ do 
7: \hspace{5em} if $a \cap b \neq \phi$ and $a \cap b \neq a_0 \cap b_0$ then 
8: \hspace{6em} $G.AddEdge(A, B)$ 
9: \hspace{1em} return $G$

In every step in the dfs, if we have identified the constraint classes that needs to be considered, $C$, we need to partition them so that these constraints are in 2 different partitions. We call the routine MinCutConservative at every node in the dfs. Corresponding to each constraint we add 2 nodes to the join graph, $a$ and $b$, each of which is connected to the node corresponding to every base relation in the the constraint. If there are $k$ constraints that are applicable then we add $k$ $a$ nodes and $k$ $b$ nodes.

Now if we partition the graph, $S$ into $C$ and $S\setminus C$ such that all $a_i$s are in $C$ and all $b_i$s are in $S\setminus C$ or all $a_i$s are in $S\setminus C$ and all $b_i$s are in $C$ such that $C$ and $S\setminus C$
are connected. Also if there is a single constraint the neither of the partitions can be \( \{a_1\} \) or \( \{b_1\} \). We extend MinCutConservative Partitioning from [Fen+12] to do constrained partitioning. We start with all \( a_i \)s in \( C \) and start expanding \( C \) as in MinCutConservative. We emit the pair only if \( C \) and \( S\setminus C \) are both connected. While extending \( C \) and generating \( S\setminus C \) continue ahead only if \( b_i \)s are in \( S\setminus C \). This will ensure that at least 2 relations in a constraint are in different partitions.

**Algorithm 5 Constrained Partition**

1: procedure \( \text{MinCutConservative}(S, \text{Constraints}) \)
2: \( S'=S \)
3: for \( C \in \text{Constraints} \) do
4: \( \text{Check}(S,C,S_2) \)
5: if \( S_2 \neq \emptyset \) then
6: \( S'=S \cup \{a_i,b_i\} \)
7: \( \text{Connect } a_i \text{ to all vertices in } S_2 \)
8: \( \text{Connect } b_i \text{ to all vertices in } S_2 \)
9: \( \text{ConstrainedMinCutConservative}(S',\{a_i\},\emptyset) \)
10: procedure \( \text{ConstrainedMinCutConservative}(S,C,X) \)
11: if \( C=S \) then return
12: if \( C \neq \emptyset \land \text{connected}(C) \) then emit\((S,S\setminus C)\)
13: \( X' \leftarrow X \)
14: for \( v \in (N(C)\setminus X) \) do
15: \( \text{pair } <O, has,b> \) \[ H = \text{GetConnectedParts}(S,C\cup\{v\},\{v\}) \]
16: if \( \text{count}(H,\text{second } = \text{true}) > 1 \) then return
17: \( O' = \text{FindAllb}(H) \)
18: \( \text{ConstrainedMinCutConservative}(S,S\setminus O',X') \)
19: \( X' \leftarrow X' \cup \{v\} \)
Chapter 5

Proof of Correctness

5.1 RS Constrained Graph

Lemma 5.1. Every Maximal Join Set is considered for Partition

Proof. In order to show that all maximal join-sets are being considered it is enough to say that all the constraints are being populated correctly. Let’s assume that corresponding to all the maximal join-sets of size less than or equal to n-1, all the constraints are populated. Now we show that if the above holds true then all the constraints of size less than or equal to n are populated correctly. At every node when we are populating the constraints, the ExpandDAG rule has already been called on the children and all the constraints of size less than or equal to n-1 are available. Now when we apply all the possible the possible rules, we will generate all the maximal join sets of size n correctly and update the constraints accordingly. So the constraints of size less than or equal to n are also generated. Thus by principle of strong induction as we are applying all the constraints in dfs fashion, all the Maximal join sets will be generated.

Lemma 5.2. Every Partition is being generated for every maximal join set considered

Proof. For the maximal join sets, we are doing full partition, every partition is being considered directly. We just need to prove completeness in the case of constrained partition. Let

\[ C = \{ A = \{A_0, A_1\}, B = \{B_0, B_1\}, \ldots \} \]

While doing constrained partition with ConstrainedMinCutConservative partitioning routine we do not consider the cases where \( A_i, \ i \neq 0 \) (the one not repre-
sented in the representative) is on side of the partition. The cases like

\[ S_1, A_1 | S_2 \]

is not considered while partitioning. For this to be complete there should be a constraint free equivalent form of this being considered during FullPartition. If we change to \( A_1 \) to \( A_0 \), the unconstrained version, the partition will be equivalent to

\[ S_1, A_1 | S_2 \]

Similarly we iteratively change all the constrained join-set to unconstrained (0 indexed) version as far as possible. For this to be considered during FullPartition \( S_1, A_1 \) should not have any other constraint, say like \( B_i, \ i \neq 0 \). This could be constrained only if there exist some \( B_i \) such that \( A_1 \cap B_i = A_0 \cap B_0 \). But by our routine ConstraintGraph, we consider only constraints such that \( A_i \cap B_j = \phi \) or \( A_0 \cap B_0 \). So if we iteratively change all the constrained representatives, then we get an unconstrained version whose full partition we have done. Thus every partition will be generated for a given maximal join set.

From above lemmas, we consider all the partitions possible, thus proving the correctness of our proofs.

5.2 Constrained Partition

Lemma 5.3. Every Partition being considered is in accordance with the constraints

Proof. In every partition being considered, all the \( a_i \)s are on one side of the partition and all the \( b_i \)s on the other side. So if any constraint is not being followed, i.e., let’s say all the elements corresponding to \( C_i \) is on one side of the partition. If then are on the side of \( a_i \), then the side of \( b_i \)s will not be connected and if they are on the the side of \( b_i \), then the side of \( a_i \)s will not be connected. Therefore it won’t be returned.

Lemma 5.4. Every partition in accordance with the constraints is being returned

Proof. As every partition is returned by the MinCutConservative, our algorithm uses similar strategy to get all the partition of the new graph. So it gets the partition of the graph with \( a_i \) and \( b_i \). As all the required partitions of the original graph can be converted to the one with \( a_i \) and \( b_i \). Therefore it’s complete.
Chapter 6

Time Complexity Comparison

[Pra16] compares the performance of RS-Graph with RS-B1. In this section we compute the time complexity for the cases in which RS-Graph was performing worse than RS-B1 for RS-Constrained Graph, RS-Graph and RS-B1. Time complexity in RS-Constrained Graph is calculated by summing up the time complexity in doing full partition and constrained partition. In the case of former the time complexity involved is $O(numberofbaserelations \times numberofpartitions)$. As the number of partitions is exponential in number of base relations, we are dropping the first term. In the case of constrained partition time complexity is not necessarily the number of partitions.

6.0.1 RS-graph

For calculating total number of partitions generated, we try to calculate the following entities -

- Number of Equivalence nodes $N$
- Number of Maximal Join-Sets $M$
- Size of each Maximal Join-Set $S$

Number of partitions generated by a maximal joint set of size $S$ and a bushy join graph $2^{S-1} - 1$.

Number of partitions generated $\sim \sum_N \sum_M 2^S$.

Time taken to pass the maximal join sets at join operator $= \sim \sum_N \text{Number of maximal join sets from left side} \times \text{Number of maximal join sets from left side}$

Total Time taken $\propto \sim \sum_N \sum_M S \times 2^S$
6.0.2 RS-B1

For RS-B1 we calculate the time taken in associativity and commutativity operations. We try to calculate the following entities -

- Number of Equivalence nodes \( N \)
- Number of operators \( O \)
- Number of operators with \( l \) base nodes on left side of operator \( L \)
- Number of operators for Equivalence node with \( L \) base nodes \( N_l \)

Number of commutative operations = \( \sum N \cdot O \)
Number of associative operations = \( \sum_{N} \sum_{L} N_l \)

6.0.3 RS-Constrained graph

For calculating time taken in \( FullPartition \), we try to calculate the following entities-

- Number of Equivalence nodes \( N \)
- Size of each Maximal Join-Set \( S \)

Number of full partitions generated by a maximal joint set of size \( S \) and a bushy join graph \( 2^{S-1} - 1 \).
Number of partitions generated \( \sim \sum N \cdot 2^S \).
Total Time taken \( \propto \sim \sum N \cdot S \cdot 2^S \)

For Calculating Time taken in Constrain Partition, we try to calculate the following entities-

- Number of Maximal Join Set \( M \) with \( k \) constrain
- Number of Constrain Partition of the algorithm \( MS \)
6.1 Constraint-free Operator push-down

\[ \Pi(A \bowtie B) \equiv \Pi(A) \bowtie B \]
\[ \Pi(A) \bowtie \Pi(B) \]

6.2 Query

\[ \Pi(A_1 \bowtie A_2 \bowtie \ldots \bowtie A_n) \]

Any equivalence node in the LQDAG is of the following form

- Join of some \( k \) relations \( A_1 \bowtie A_2 \bowtie \ldots \bowtie A_k \)
- Projection applied on join of some \( k \) relations \( \Pi(A_1 \bowtie A_2 \bowtie \ldots \bowtie A_k) \)

6.2.1 RS-Constrained graph

For easier calculations, we assume that the representative maximal join-set has the base equivalence nodes being \( \Pi(\text{base relation}) \)

**Constrained Free Maximal Join set for FullPartition**

For full partition, we have to consider only the join-sets of the type:

- \{ \Pi(A_1), \Pi(A_2), \ldots, \Pi(A_k) \}, i.e., all the join sets with \( \Pi \)
- \{ \( A_1, A_2, \ldots, A_k \) \}, i.e., all the join sets without any \( \Pi \)

Time complexity in full partition,

- Number of partitions of a join-set of size \( k = 2^{k-1} - 1 \)
- Total time involved will be \( \sum_k 2 \cdot \binom{n}{k} \cdot 2^{k-1} = 3^n \)
Constrained Maximal Join set to consider Constrained Partition

For Constrained Partition, we consider the partition which have at least one relation with \( \Pi \) and one without \( \Pi \) eg. \((\Pi(A_1), A_2, ..., \Pi(A_k))\)

Here if we consider maximal independent set algorithm to get the constraints, then all the pairs of the form \( \Pi(A), B \) will be constrained giving rise to only one partition possible, i.e., all the elements with \( \Pi \) on one side and without \( \Pi \) on other side

- Number of ways to choose \( k \) elements is \( \binom{n}{k} \)
- Number of ways to choose \( i \) elements with \( \Pi \) from these \( k \) elements is \( \binom{k}{i} \)
- \( \text{sum} = \sum_k \binom{n}{k} \sum_i \binom{k}{i} \ast 1 = 3^n \)

In this case we consider the Bipartite Graph as there are only constraints of size 2, and employ the graph 2-coloring algorithm to do the partition, then we will get the time complexity to be

\[ \text{Time Complexity} = \sum_k \binom{n}{k} \sum_i \binom{k}{i} \ast k = n \ast 3^n \]

Therefore total time complexity turns out to be \( n \ast 3^n \)

6.2.2 RS-graph

A maximal join-set for the latter kind looks like \((\Pi(A_1), A_2, ..., \Pi(A_k))\). The size of the maximal join-set is \( k \) and the \( i \)th element is either \( \Pi(A_i) \) or simply \( A_i \). This is because in the LQDAG a base node can either be \( \Pi(A_i) \) or \( A_i \).

- Number of Equivalence Nodes with \( k \) relations without \( \Pi \) = \( \binom{n}{k} \)
- Number of maximal join-set of equivalence node without \( \Pi \) = 1
- Time in partitioning equivalence nodes without \( \Pi \) at top = \( \sum \binom{n}{k} (2^{k-1} - 1) \sim 3^n \)
- Number of Equivalence Nodes with \( k \) relations and \( \Pi \) = \( \binom{n}{k} \)
- Number of maximal join-sets of equivalence node \( \Pi(A_1 \bowtie A_2 \bowtie ... \bowtie A_k) \) with \( k \) base relations = \( 2^k - 1 \)
- Time in partitioning equivalence nodes with \( \Pi \) at top = \( \sum \binom{n}{k} (2^k - 1)(2^{k-1} - 1) \sim 5^n \)
• Time spent to pass the maximal joinsets = \( \sum_i \binom{n}{i} \sum_k \binom{k}{i} (2^k - 1)(2^{i-k} - 1) \sim 5^n \)

• Total time spent \( \sim 5^n \)

\[ \text{6.2.3 RS-B1} \]

For an Equivalence Node with \( \Pi \) at the top, the operators are as follows -

• \( \Pi(A) \bowtie B \)
• \( A \bowtie \Pi(B) \)
• \( \Pi(A) \bowtie \Pi(B) \)

**Equivalence nodes without \( \Pi \)**

• Number of Equivalence Nodes with \( k \) relations without \( \Pi = \binom{n}{k} \)
  
  - Number of operators = \( 2^k - 2 \)
  
  - Number of commutative operations = \( 2^k - 2 \)
  
  - Number of associative operations with 1 rel. on left = \( 2^l - 2 \)
  
  - Total no of assoc operations = \( \sum \binom{k}{l} (2^l - 2) \sim 3^k \)

• Total Time for nodes without \( \Pi \) \( \sim \sum \binom{n}{k} (3^k) \sim 4^n \)

**Equivalence nodes with \( \Pi \)**

• Number of equivalence node \( \Pi(A_1 \bowtie A_2 \bowtie \cdots \bowtie A_k) \) with \( k \) base relations = \( \binom{n}{k} \)
  
  - Number of operators = \( (2^k - 2) \times 3 \)
  
  - Number of commutative operations = \( (2^k - 2) \times 3 \)
  
  - Number of associative operations with \( \Pi \) on left with 1 rel. on left = \( 3 \times (2^l - 2) \times 2 \)
  
  - Number of associative operations without \( \Pi \) on left with 1 rel. = \( 2^l - 2 \)
  
  - Total no of assoc operations = \( \sum \binom{k}{l} (3 \times (2^l - 2) \times 3 + 3 \times (2^l - 2)) \sim 7.3^k \)
  
  - Time spent in pushing \( \Pi \) = \( 2 \times (2^k - 2) \)

• Total Time for nodes with \( \Pi \) \( \sim \sum \binom{n}{k} (7.3^k + 3.2^k + 2.2^k) \sim 7.4^n \)

• Total time spent \( \sim 8.4^n \)
6.2.4 Analysis

We find that time taken in RS-Graph is exponentially greater than traditional RS-B1. This additional time taken in Graph-RS can be attributed to same partitions being generated over and over again. For example the equivalence node corresponding to these partitions are the same -

- \((\Pi(C), D)\)
- \((C, \Pi(D))\)
- \((\Pi(C), \Pi(D))\)

RS Constrained Graph fulfills that shortcoming of RS-Graph to make the time-complexity lesser than RS-B1.

6.3 Query

\[ \Pi(A_1 \bowtie B_1) \bowtie ... \bowtie \Pi(A_n \bowtie B_n) \]

Any equivalence node in the LQDAG is of the following form

- Join of some combination of \(\Pi((A_i \bowtie B_i))\), or \(\Pi(A)\) or \(\Pi(B)\) or A or B
- \(A_i \bowtie B_i\)

There are seven types of representatives for each pair in a given pair in a maximal join set:

- Unconstrained - \(\Pi(A) \bowtie \Pi(B), A, B, \Pi(A), \Pi(B), \phi\)
- Constrained - \(\Pi(A) \bowtie B, \Pi(B) \bowtie A\)
- \(A_1 \bowtie B_1\)

6.3.1 RS-Constrain graph

While getting the constraints using maximal join set, if we make edge if the constrain have some intersection, i.e.. \(a_i \cap b_i \neq \phi\), then we get the below calculation
Constrain Free Maximal Join set to consider Full Partition

Here We can various types of representatives of each pair:

- Single element representative count = i - type = $\binom{n}{i} \cdot 4^i$
- Double element representative count = j - type = $\binom{n-i}{j} \cdot 1$
- Partition count with elements $2^i i + j = 2^{i+2j-1} - 1$

Total time taken = $\sum_i \binom{n}{i} \cdot 4^i \sum_j \binom{n-i}{j} \cdot (2^{i+2j-1} - 1) = 13^n$

Constrained Maximal Join set to consider Constrain Partition

Here We can various types of representatives of each pair:

- k pairs are represented by constrained representative - $\binom{n}{k} \cdot 2^k$
- Single element representative count = i - type = $\binom{n-k}{i} \cdot 4^i$
- Double element unconstrained representative count = j - type = $\binom{n-k-i}{j} \cdot 1$
- Partition count with elements $2k + i + 2j = 2^{k+i+2j-1}$

Total time taken = $\sum_k \binom{n}{k} \cdot 2^k \cdot 2^{k-1} \sum_i \binom{n}{i} \cdot 4^i \sum_j \binom{n-i}{j} \cdot (2^{2i+j-1}) = 17^n$

Therefore total Time is $O(17^n) + O(13^n) = O(17^n)$

If we use the RS Constrain Graph, as this query is a generalization of the query, for which the complexity is calculated above, the time complexity for the same turns out to be $4^{2^n} = 16^n$

6.3.2 RS-graph

Here We can various types of representatives of each pair:

- Single element representative count = i - type = $\binom{n}{i} \cdot 4^i$
- Double element representative count = j - type = $\binom{n-i}{j} \cdot 3^j$
- Partition count with elements $i + 2^j = 2^{2i+j-1} - 1$

Total time taken = $\sum_i \binom{n}{i} \cdot 4^i \sum_j \binom{n-i}{j} \cdot (3^j) \cdot (2^{i+2j-1} - 1) = 21^n$
6.3.3 RS-B1

RS-B1 will give $4^{2n} = 16^n$ on this query, which is why we shifted to new RS-Constraint Graph.

6.3.4 Analysis

In this case as well, we observe that RS-Constraint Graph performs at least as good as RS-B1 rule set. RS-Constraint Graph obviously performs better than RS-Graph. Even if we just consider the non overlapping cases RS-Constraint Graph performs better than RS-Graph, but worse than RS-B1.
Chapter 7

Conclusion and Future Work

The RS-Constrained Graph rule set is an interesting approach for generating logical plan space of a query. We focus on reducing the time complexity of join enumeration by modifying RS-Graph rule. We identify the cases where RS-Graph rule will do repeated partitioning and propose the constrained partitioning strategy to just propagate single maximal join set and additional information in the form of set of constraint classes avoid repeated partitioning. We also propose \textit{ConstrainedMinCutConservative} an efficient way to make use of the additional information in the form of Constraints and generate only the required partitions.

Future Work could involve the following.

- The RS-Constrained Graph algorithm needs to be implemented to empirically observe its benefits
- The RS-Constrained Graph needs to be modified to incorporate all the possible constraints to avoid repeated partitioning completely
- The \textit{ConstrainedMinCutConservative} needs to be modified to compute the number of partitioning in $O(\text{number of partitions})$
Bibliography


