Optimizing Join Enumeration in Transformation-based Query Optimizers

BTP Presentation

Anil Shanbhag
Roll No: 100050082

Guide: Prof. S. Sudarshan

Outline

- Introduction to Query Optimization
- Related Work
- Evaluate existing rule sets
- Rule set RS-B3
- N-ary Join Rule

Introduction to Query Optimization

Queries submitted to database are typically in high-level language. The goal of query optimization it to find the most-efficient way to execute the given query. Query optimizers typically follow a three step approach:

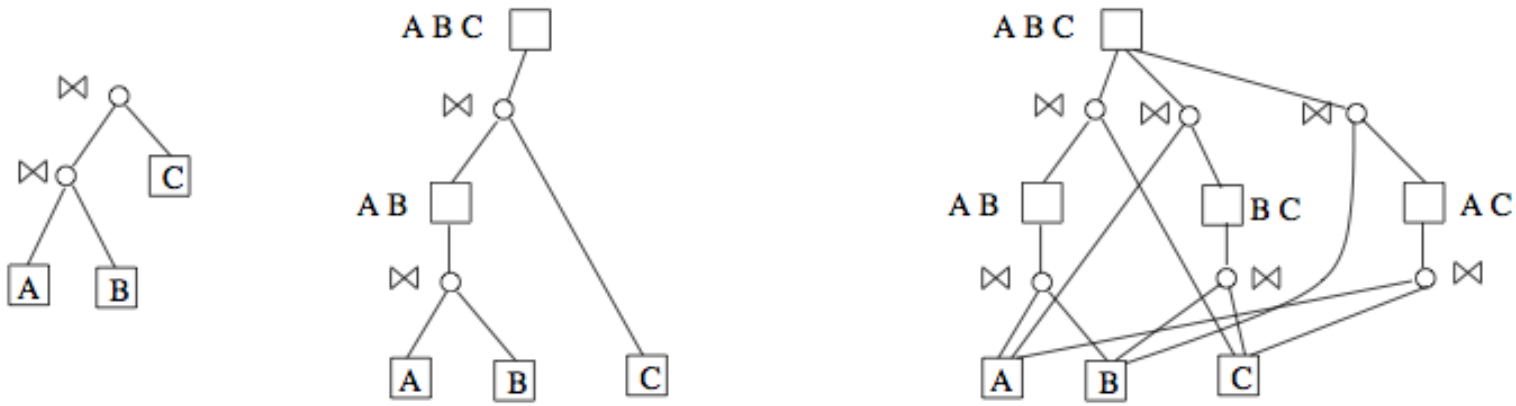
- Generate all execution plans that are logically equivalent to the user-submitted query.
- Estimate the cost of executing each of the alternate plans.
- Search through the space of all generated plans to find the plan with least cost.

Space of Logically Equivalent Plans

The steps involved in generating the space of logically equivalent plans :

- Query is parsed to generate a query tree
- Logical Query DAG is constructed from the query tree.
- Expand the LQDAG using transformation rules to encompass the space of all equivalent logical plans. This is called Logical Plan Space.
- The implementation of logical operators is not defined, adding the different implementations for each operator we get the Physical Plan Space.

Example



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

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

Motivation

- Long ago, Ono and Lohman gave a lower bound of $O(3^n)$ with n the number of relations, on the complexity join enumeration.
- Exhaustive search is computationally expensive.
- *Heuristic*: Consider all possible bushy join trees, but exclude trees with cross products from the search (presuming that all considered queries span a connected query graph)
- Interesting work lately on bottom-up dynamic programming and top-down memoization based join enumeration algorithms that achieve the lower bound and explore the reduced space effectively.
- On the other hand, transformation-based optimizers generate exponential number of duplicates.

Problem Statement

The question is can join enumeration in transformation-based query optimizers be as efficient as in DP/memoization based join optimizers ?
Starting from a connected query tree, our goal is to

- Generate the space of all join trees without cross products
- Always be within the space
- Avoid generation of duplicates

Finding a solution for general graphs is NP-Hard, however for tree queries ie: queries whose join graphs form a tree for nested loop join, it is possible to find the optimal solution in polynomial time.

Cost function f satisfies *adjacent sequence interchange* property if there exists a rank function r_f such that $f(AS_1S_2B) <= f(AS_1S_2B)$ if and only if $r_f(S_1) <= r_f(S_2)$. If yes, optimal sequence can be found in $O(nlog(n))$ where the order constraints are series parallel.

We call TreeOpt once with each node as root. In this case, the order constraints form directed tree which is a special case of series parallel constraints.

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**

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.

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.

Top-Down Join Enumeration

A generic top-down join enumerator first initializes the costs of the base relations and starting from join graph J, recursively explores the entire space. If the best tree for a 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 $P^{sym}_{ccp}(S)$.

Top-Down Join Enumeration Contd.

The recursive decent stops when $|S| = 1$ or the best tree is already known. Once all the possibilities have been tried, a subroutine 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.

Depending on the choice of partitioning algorithm, the overall performance of TDPlanGen can vary by orders of magnitude.

Top-Down Join Enumeration Contd.

```
TDPlanGen(G)
    ▷ Input: connected G=(V,E), V = ⋃_{1≤i≤|V|} {R_i}
    ▷ Output: an optimal join tree for G
1   for i ← 1 to |V|
2       do BestTree[{R_i}] ← 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) ∈ P^sym_ccp(S)
3           do BuildTree(G_|S, TDPGSub(G_|S_1),
                                TDPGSub(G_|S_2), ∞)
4   return BestTree[S]
```

Figure : Pseudocode for TDPLANGEN

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

Clearly the rules are redundant. We can remove either left-associativity or right-associativity and still explore the same space. We shall show completeness later.

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

Pellenkoft et al. 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 $4^n - 3^{n+1} + 2^{n+2} - n - 2$.
We show later that this rule set is actually complete. Hence it is possible to explore the search space using rules.

- 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 $\bowtie_4$

RS-B2 generates the space of all bushy join trees without duplicates.

Completeness of RS-B1

Problem: 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.

Reduction 1: Sufficient to show one route between Q and Q'. The route we show is Q to valid left deep tree to Q'.

Choice of left deep tree: 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.

Reduction 2: If in Q we can pull up R_n so that its the last relation to be joined, we are done.

Completeness of RS-B1 Contd.

We can reduce the given tree to reduced query tree as shown on right. Height of tree is defined in conventional sense.

Reduction 3: If we can pull R_k one level up ie: reduce the height of reduced query tree by one, we are done.

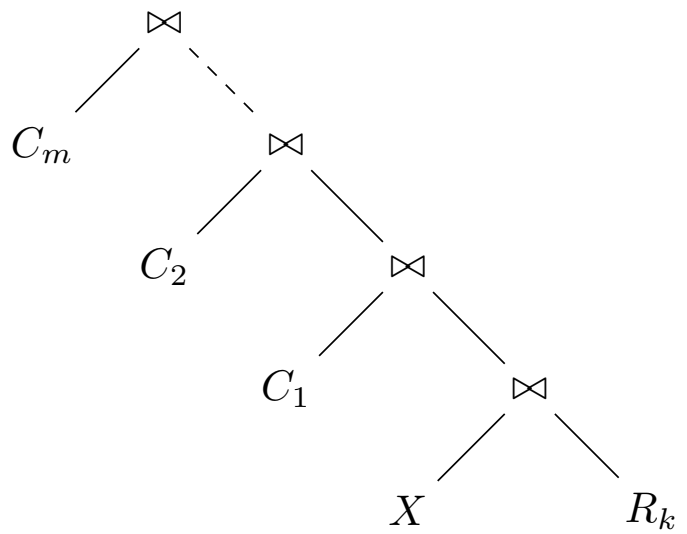


Figure : Reduced Query Tree

Completeness of RS-B1 Contd.

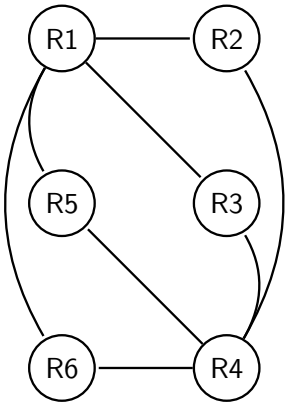
To prove reduction 3: The bases case m $= 0$ holds trivially. For m > 0, since R_k was the last relation to be joined, there exists some C_i which has an edge with X. We pull down C_i using $C \rightarrow LA \rightarrow C$. Finally $X \bowtie C_i$ and renamed as new $C_{i'}$. Height of tree reduced by 1.

Finally for the other direction ie: from left-deep tree to Q', it is equivalent to using 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 : $RA \Rightarrow C \rightarrow C \rightarrow LA \rightarrow C$.

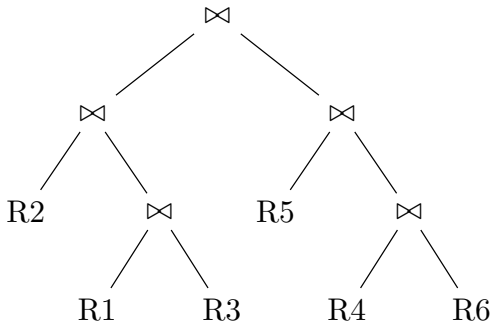
Hence Proved :-)

Incompleteness of RS-B2

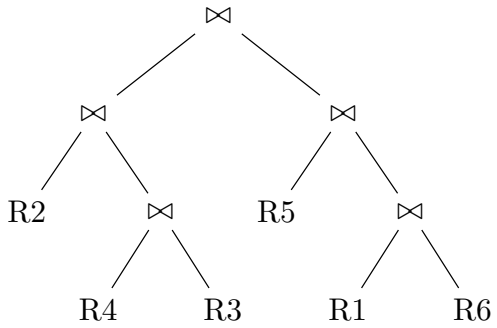
Consider a simple example. Given a set of relations to be joined with join graph J and initial query tree Q, we wish to go to Q'. However using RS-B2 it is not possible as it would lead to formation of subtree $R_2 \bowtie R_3$ which has cross product.



(a) Join Graph J



(b) Query Tree Q



(c) Query Tree Q_2

Rule set RS-B3

- 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 $\bowtie_3$.

Completeness of RS-B3

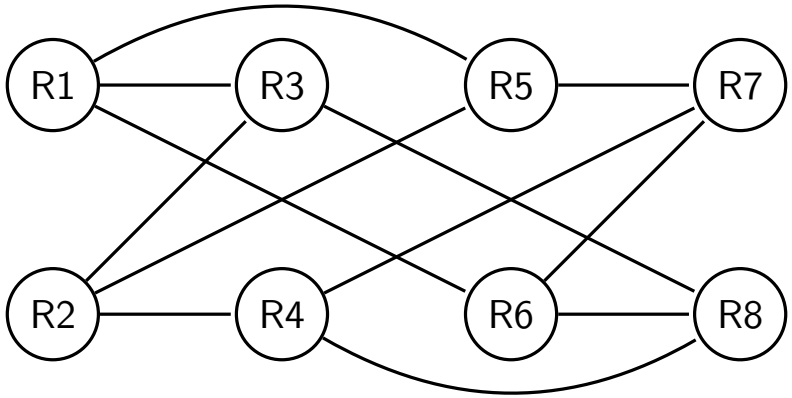


Figure : Extreme case for RS-B3

There exist edge cases. $Q = [R_1 R_2 R_3 R_4] \bowtie [R_5 R_6 R_7 R_8]$ and $Q' = [R_1 R_2 R_5 R_6] \bowtie [R_3 R_4 R_7 R_8]$. 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.

Efficiency of RS-B3

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

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

D is the number of duplicates

Graph-based Join Enumeration

DP-based and memoization-based enumeration algorithms by Pit Fender et al. and De Haan et al. 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.

We propose a new method of doing join enumeration in transformation-based optimizers based on ideas from memoization based join enumerators.

The secret sauce of TDMinCutConservative is MinCutConservative.

```
PARTITION_MinCutConservative(S)
    ▷ Input: a connected set S, arbitrary vertex t ∈ S
    ▷ Output: all ccps for S
1   MinCutConservative(S, ∅, ∅)

MinCutConservative(S, C, X)
    ▷ Input: connected set S, C ∩ X = ∅
    ▷ Output: ccps for S
 1  if C = S
 2     then return
 3  if C ≠ ∅
 4     then emit (C, S \ C)
 5  X' ← X
 6  for v ∈ (N(C) \ X)
 7        do O = GetConnectedParts(S, C ∪ {v}, {v})
 8           for all O_i ∈ O
 9                 do MinCutConservative(S, S \ O_i, X')
10           X' ← X' ∪ {v}
    ▷ N(∅) = {arbitrary element of t ∈ S}
```

Figure : Pseudocode for MinCutConservative

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.

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.

N-ary Join Rule contd.

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 $\text{MINCUTCONSERVATIVE}(G_{|S})$. 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).

Thank You