Cost Model

Given an input of n tuples, a query relation R_i can be joined using *Sort Merge* (S_i) or *Nested Loop* (N_i)

- Sort Merge
  - Cost Model : $A_i + (n * k_s)$
    - A_i is the cost of accessing R_i in the best possible way
    - k_s is a *constant* which approximates the sorting cost to $n * k_s$
- Nested Loop
  - Cost Model : $n * w_i$
    - w_i is the per tuple cost of accessing R_i in Nested Loop method.

Feasible Solutions

Input

- Root : Outermost relation R_0 with n_0 tuples coming from it
- Set of relations $\{R_1, R_2...R_k\}$, each R_i is modelled as (A_i, w_i, a_i) where a_i is the product of number of tuples in R_i and the selectivity of the predicates relevant to R_i

Possible Sequences

- The permutation of relations $R_1, R_2...R_k$ with each relation R_i can be joined using *sort merge* or *nested loop*
- $2^k * k!$ possible feasible sequences

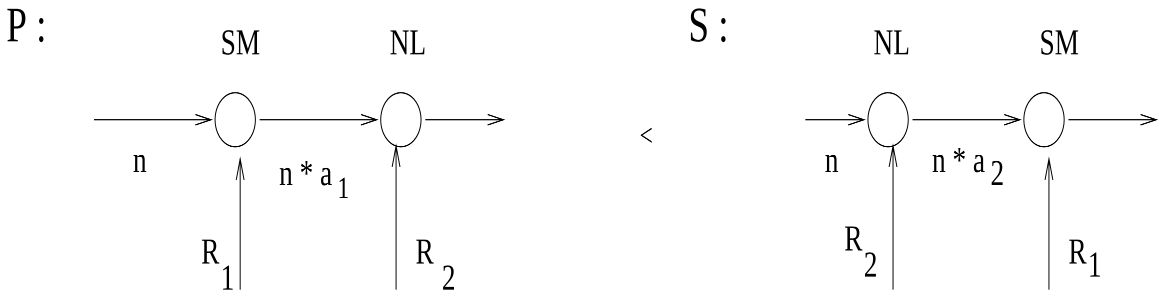
Problem Definition

Example
Let us consider relations R_1, R_2, R_3 with n tuples coming from root R_0
A possible feasible solution is $S_3N_1S_2$ and its cost:

$$C(S_3N_1S_2, n) = A_3 + n * k_s + n * a_3 * w_1 + A_2 + n * a_3 * a_1 * k_s$$

Problem :
Given a **Star Query** of $k+1$ relations we have to find the *minimum* cost feasible solution among the set of all feasible solutions.

Rank Ordering



- $A_1 + n * k_s + n * a_1 * w_2 < n * w_2 + A_1 + n * a_2 * k_s$
  $\Rightarrow (1 - a_1)/k_s > (1 - a_2)/w_2$
- We say that $r(S_1) > r(N_2)$ where r is the **rank** of the relevant join method for a particular relation
- Optimal sequence must contain join methods in decreasing rank order
- This yields an algorithm to find the minimum cost feasible solution in $O(2^k)$ time and **linear** space.

Plain Backtracking Approach

- Given a rank sequence, find all 2^k sequences and calculate the cost for each
- Store the minimum cost feasible sequence

Time complexity is $O(2^k)$ and space complexity is $O(k)$

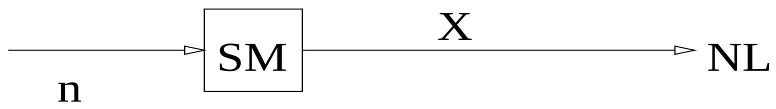
This algorithm can be applied seperately to optimize relations with $a_i < 1$ and $a_i > 1$ which reduces the time complexity.

Best case time complexity is $0(2^{k/2})$

Improvements

Impovements over plain backtracking can be achieved using the property of **local optimality**

Consider a rank sequence for all $a_i < 1$



- S_j is locally optimal over N_j if $C(S_j X, n) < C(X N_j, n)$

This implies, $A_j/n + k_s < f(X)$
where, $f(X) = w_x * (1 - a_j) + w_j * a_x$

$f(X)$ is calculated **incrementally** which may result in generating less number of sequences to get the optimal one.

Example
$S_1 N_1 S_2 N_2 .... S_j N_j$ will give the best time performance of $O(j)$

Some Results

No. of reln.	Plain backtracking	Intelligent backtracking
14	1.28 ms	1.00 ms
15	2.97 ms	2.45 ms
16	5.71 ms	4.54 ms
17	11.07 ms	7.70 ms
18	23.30 ms	14.55 ms
19	40.00 ms	19.87 ms