Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

Wolfgang Scheufele Guido Moerkotte

RWTH Aachen
Lehrstuhl für Informatik III
52056 Aachen, Germany
{*ws*|*moer*}*@gom.informatik.rwth-aachen.de*

Abstract

The generally accepted optimization heuristics of pushing selections down does not yield optimal plans in the presence of expensive predicates. Therefore, several researchers have proposed algorithms for the optimal ordering of expensive joins and selections in a query evaluation plan. All of these algorithms have an exponential run time. For a special case, we propose a polynomial algorithm which – in one integrated step – computes the optimal join order and places expensive predicates optimally within the join tree. The special case is characterized by the following statements:

1. only left-deep trees are considered,
2. no cross-products are considered,
3. the cost function has to exhibit the ASI property, and
4. cheap selections are pushed before-hand.

1 Introduction

Traditional work on algebraic query optimization has mainly focused on the problem of ordering joins in a query. Restrictions like selections and projections are generally treated by "push-down rules". According to these, selections and projections should be pushed down the query plan as far as possible. These heuristic rules worked quite well for traditional relational database systems where the evaluation of selection predicates is of neglectable cost and every selection reduces the cost of subsequent joins. As pointed out by Kemper, Moerkotte and Steinbrunn [8] and Hellerstein and Stonebraker [4], this is no longer true for modern database systems like extensible DBMSs or object-oriented DBMSs where users are allowed to implement arbitrary complex functions in a general-purpose programming language. Even in traditional SQL database systems expensive predicates occur due to subqueries which can not be unnested.

Several researchers have proposed algorithms for the optimal ordering of expensive joins and selections. All these algorithms bear exponential worst-case complexity.

In this paper we generalize the approaches of Ibaraki, Kameda [6] and Krishnamurthy, Boral, Zaniolo [10] for ordering joins to capture joins and selections – both with expensive predicates. We use a refined version of the standard cost function given in [10] to account for expensive join and selection predicates. Concerning the search space and the class of admissible queries, we adopt the assumptions of [10]. That is, we consider *acyclic join graphs* only and produce *left-deep processing trees without cross products.*

The rest of the paper is organized as follows. After summarizing related work in the next section, we present the background for the rest of the paper in section 3. In section 4 we present an algorithm for ordering expensive selections and joins. Section 5 concludes the paper.

2 Related Work

Several researchers addressed the problem of ordering binary joins in an n-way join. The standard and — even today — most prevailing method to solve this optimization problem is *dynamic programming* [17]. A reason for its popularity might be the simplicity and universality of the method. It can be applied to all join graphs, all query processing trees – even with cross products, and basically all cost functions. A fast implementation of a dynamic programming algorithm for bushy trees and cross products is described in [21]. In [15], Ono and Lohman discussed the complexity of dynamic programming algorithms for the join ordering problem. They also gave the first real world examples to show that abandoning cross products can lead to more expensive plans.

An NP-hardness result for the join ordering problem for general query graphs was established in 1984 [6]. Later on, a further results showed that even the problem of determining optimal left-deep trees with cross products for star queries is NP-complete [1]. The first polynomial time optimization algorithm was devised by Ibaraki and Kameda [6] in 1984. Their IK-algorithm solved the join ordering problem for the case of left-deep processing trees without cross products, acyclic join graphs and a nontrivial cost function counting disk accesses for a special block-wise nested-loop algorithm. The IK-algorithm was subsequently improved by Krishnamurthy, Boral and Zaniolo [10] to work in time $O(n^2)$. They assumed that the database is memory resident and used a simpler and more common cost function. The IK- and KBZ-algorithms both built on the notion of *rank*. The rank is a numerical value that can be associated with every left-deep tree. The algorithm is based on the fact that the relative order of two adjacent connected subparts of a left-deep tree is basically determined by the size of the rank of each subpart supposed that no cross products are present in the tree. In the next section, we review the IK-KBZ-algorithm in some detail.

In [1], the authors present an algorithm to find optimal left-deep processing trees with cross products for star queries in time $O(2^c + n \log n)$ where c is the number of cross products in a solution. Concerning the area of non-exact (heuristic) algorithms, there are several approaches, too. The first published nontrivial heuristic for treating cyclic queries appeared in [10]. They applied the KBZ-algorithm to a minimal spanning tree of the join graph. Several other approaches use general heuristic search methods like *branch-and-bound, A^*, tabu search*, etc. to find orderings with small costs. The class of randomized algorithms for the join ordering problem includes all stochastic optimization methods like *random*

sampling, simulated annealing, genetic algorithms and several *hybrid approaches*. A comparative overview over the best-known exact, heuristic and randomized approaches can be can be found in [9, 18].

The above mentioned approaches order joins only. Whereas this optimization problem attracted much attention in the database research community, much less investigations took place for optimizing boolean expressions not containing any join predicate. Within the database community, there are only two examples the authors are aware of [7, 8].

Concerning the problem of ordering both joins and selections only few approaches are known. By modeling selections as joins with artificial relations, the authors of [2, 11] propose a traditional optimization algorithm to order joins and selections.
In [4, 5] Hellerstein and Stonebraker present an algorithm for the optimal placement of expensive selections within a join graph. By applying their "predicate migration algorithm" to every possible join ordering, they gave an $O(m!\,(m+s)^4 \log(m+s))$ algorithm for ordering m joins and s selections.
The authors of [19] introduced a new type of evaluation plans – called *bypass evaluation plans* – which are superior to traditional plans in the case of *disjunctive queries*. They gave an $O(2^{n-1}n!)$ algorithm for determining near-optimal bypass plans for disjunctive queries with expensive predicates. By reducing the search space there is also a faster algorithm which runs in time $O((n+1)!)$ and yields optimal bypass plans for randomly generated queries with a high probability.
The table below compares the different approaches of ordering joins and selections. The parameter m denotes the number of relations and n denotes the total number of operators in the query.

	dynamic progr. [17]	IK-KBZ [6, 10]	dynamic progr. [16]	predicate migration [4, 5]	bypassing joins [19]	this paper
query graph	arbitrary	acyclic	arbitrary	arbitrary	arbitrary	acyclic
query type	conjunctive	conjunctive	arbitrary	conj. selection predicates	arbitrary	conjunctive
processing trees	bushy trees	left-deep trees	bushy trees	bushy trees	bypass plans	left-deep trees
cross prod.	yes	no	yes	yes	no	no
ordering of	joins	joins	selections and joins	selections and joins	selections and joins	selections and joins
cost function	arbitrary	ASI	arbitrary	(ASI)	arbitrary	ASI
join costs	arbitrary	linear	arbitrary	linear	arbitrary	linear
caching	no	no	yes	yes	no	no
expensive predicates	no	no	yes	yes	yes	yes
semijoins	no	no	no	no	yes	no
worst-case complexity	$O(3^m)$	$O(m^2)$	$O(5^m + m3^m)$	$O(m!\,n^4 \log n)$	$O(2^{n-1}n!)$	$O(n^2)$

3 Preliminaries

We consider simple *conjunctive queries* [20] involving only single table selections and binary joins (*selection-join-queries*). A query is represented by a set of query predicates $p^{[1]}, \ldots, p^{[n]}$, where $p^{[i]}$ is either a selection predicate p_i which refers to a single relation R_i or a join predicate $p_{i,j}$ connecting relations R_i and R_j.

Let $R_1, \ldots, R_n$ be the relations involved in the query. Associated with each relation is its *size* $n_i = |R_i|$. The predicates in the query induce a *join graph* $G = (\{R_1, \ldots, R_n\}, E)$, where E contains all pairs $\{R_i, R_j\}$ for which there exists a predicate $p_{i,j}$ relating R_i and R_j. We assume that the join graph is *acyclic*. For every join predicate $p_{i,j} \in P$ relating relations R_i, R_j, there is an associated *selectivity*

$$f_{i,j} := \frac{|R_i \bowtie R_j|}{|R_i \times R_j|} = \frac{|R_i \bowtie R_j|}{|R_i| \times |R_j|}$$

and for every selection predicate $p_i \in P$, we assume the existence of a *selectivity*

$$f_i := \frac{|\sigma_{p_i}(R)|}{|R|}$$

The selectivity is the expected fraction of tuples that qualifies in a join or a selection. The evaluation of a join or selection predicate can be of different costs. We denote by $c_{i,j}$ the costs of evaluating predicate $p_{i,j}$ for one tuple of $R_i \times R_j$. Similarly, c_i denotes the per-tuple-costs associated with the selection predicate p_i.

A *processing tree* for a select-join-query is a rooted binary tree with its internal nodes having either one or two sons. In the first case the node represents a selection operation and in the latter case it represents a binary join operation. The tree has exactly n leaves, which are the relations $R_1, \ldots, R_n$. A processing tree defines a partial order on the nodes of the tree, corresponding to possible query execution strategies. Internal nodes can be viewed as intermediate result relations and leaf nodes represent base relations. Processing trees are classified according to their shape. The main distinction is between *left-deep trees* and *bushy trees*. In a left-deep tree the right son of an internal node is always a leaf. Otherwise it is called a *bushy-tree*. Obviously, every processing tree corresponds to an expression in the relational algebra. In case of left-deep trees, every processing tree can be represented by an algebraic expression of the form

$$(\ldots((R_1\, \psi_1)\, \psi_2)\, \ldots\, \psi_m),$$

where the *unary* operators ψ_i $(i = 1 \ldots n)$ are either selections σ_{p_i} or joins $\bowtie_{p_{i,j}} R_j$. R_1 is called the *starting relation*. The rest of this paper deals only with left-deep processing trees.

There are different implementations of the join operator, each leading to different cost functions for the join and hence to different cost functions for the whole processing tree. Common implementations of a binary join operator are (see [3, 13, 20])

- nested loop join
- hash loop join
- sort merge join

The corresponding cost functions [10] are

$$\begin{aligned} C_{nl}(R \bowtie S) &= |R| \cdot |S| + |R| \cdot |S| \cdot f_{RS} \\ C_{hl}(R \bowtie S) &= 1.2 \cdot |R| + |R| \cdot |S| \cdot f_{RS} \\ C_{sm}(R \bowtie S) &= (|R| \cdot \log |R| + |S| \cdot \log |S|) + |R| \cdot |S| \cdot f_{RS} \end{aligned}$$

Here we made the important assumption that our database is *memory resident*, i.e. there is no paging to disk during execution of a query. The first summand in the cost functions accounts for the costs of iterating over the relations and for checking the join predicate. The second sum which is identical for all cost functions, accounts for the costs to construct the intermediate results. The factor 1.2 stands for the average length of the collision list of the hash table.
In order to approximate the costs of n-way joins, we associate with each of the cost functions C_{nl}, C_{hl}, C_{sm} operating on join-expressions a corresponding binary cost function g working on input sizes:

$$\begin{aligned} g_{nl}(r,s) &= r \cdot s + r \cdot s \cdot f_{RS} \\ g_{hl}(r,s) &= 1.2 \cdot r + r \cdot s \cdot f_{RS} \\ g_{sm}(r,s) &= (r \cdot \log r + s \cdot \log s) + r \cdot s \cdot f_{RS} \end{aligned}$$

Since the size of the intermediate results plays an important role in all cost functions, it is a central problem to determine this sizes. Unfortunately, the problem is quite difficult. In fact, if one wants to determine the size exactly, the only way would be to perform the joins explicitly and count the tuples in the result! This is much to expensive in practice. As a matter of fact, the complexity can be exponential, since the size of intermediate results can grow exponentially if the selectivities are sufficiently close to one. Under the usual assumptions of independent and uniform distribution of attribute values, the following standard approximation holds [20]:

$$|R_1 \bowtie \ldots \bowtie R_k| \approx \prod_{i=1}^{k} |R_i| \prod_{j<i} f_{ij}$$

Hence we can write

$$\begin{aligned} C(R_{\pi(1)} \bowtie \ldots \bowtie R_{\pi(n)}) &= \sum_{k=2}^{n} g_k(|R_{\pi(2)} \bowtie \ldots \bowtie R_{\pi(k)}|, |R_{\pi(k)}|) \\ &= \sum_{k=2}^{n} g_k(\prod_{i=1}^{k} |R_{\pi(i)}| \prod_{j<i} f_{\pi(i)\pi(j)}, |R_{\pi(k)}|) \end{aligned}$$

where g_k is one of the functions C_{nl}, C_{hl} depending on the join algorithm used. Since C_{nl} and C_{hl} are both linear in the first argument, we can "extract" the linear factor and define the unary cost function g as

$$\begin{aligned} g_{nl}(s) &= s + s \cdot f_{RS} \\ g_{hl}(s) &= 1.2 + s \cdot f_{RS} \end{aligned}$$

Henceforth we will use the unary function g. Now, we have

$$C(R_{\pi(1)} \bowtie \ldots \bowtie R_{\pi(n)}) = \sum_{k=2}^{n} |R_{\pi(2)} \bowtie \ldots \bowtie R_{\pi(k)}| \, g_k(|R_{\pi(k)}|)$$

Please note that C covers almost all cost functions for joins as pointed out in [10] and it even covers the nontrivial cost function given in [6]. However, it does not account for expensive join predicates. These will be taken care of in the next section.

Next, we repeat some fundamental results concerning the optimization of *cost functions with ASI-property* and the *IK-KBZ-algorithm* of Krishnamurthy, Boral and Zaniolo [10] for the join ordering problem. The KBZ-algorithm is an improved version of the IK-algorithm of Ibaraki and Kameda [6], who were the first to recognize the applicability of results for sequencing problems with ASI cost functions [14] to the area of optimizing join orders.

As mentioned earlier, every left-deep tree corresponds to a permutation indicating the order in which the base relations are joined with the intermediate result relation. We will henceforth speak of permutations or sequences instead of left-deep processing trees.

We have just seen that all cost functions in the standard cost model for left-deep trees have the form

$$\begin{aligned} Cost(s) &= \sum_{i=2}^{n} |s_1 \dots s_{i-1}| * g_i(|s_i|) \\ &= \sum_{i=2}^{n} (\prod_{j=1}^{i-1} f_j * |s_j|) * g_i(|s_i|) \end{aligned} \tag{1}$$

where the function g_i accounts for the join algorithm used in the respective step. As can easily be verified, there is a recursive definition of these cost functions:

$$\begin{aligned} C(\epsilon) &= 0 \\ C(R_j) &= 0 \qquad \text{if } R_j \text{ is the starting relation} \\ C(R_j) &= g_j(|R_j|) \qquad \text{else} \\ C(s_1 s_2) &= C(s_1) + T(s_1) * C(s_2) \end{aligned}$$

with

$$\begin{aligned} T(\epsilon) &= 1 \\ T(s) &= \prod_{i=1}^{n} f_i s_i \end{aligned}$$

Here, s_1, s_2 and s denote sequences of relations.

We now define the *ASI property*[1] [14] of a cost function.

Definition 3.1 *(ASI property)*
A cost function C has the ASI property, if there exists a rank function $r(s)$ for sequences s, such that for all sequences a and b and all non-empty sequences u and v the following holds:

$$C(auvb) \leq C(avub) \Leftrightarrow r(u) \leq r(v)$$

For a cost function of the above form, we have the following lemma:

Lemma 3.1 *Let C be a cost function which can be written in the above form. Then C has the ASI property for the rank function*

$$r(s) = \frac{T(s) - 1}{C(s)}$$

for nonempty sequences s.

[1] adjacent sequence interchange property

Let us consider sequences with a fixed starting relation, say R_1. Since we do not allow any cross products in a processing tree, the second relation in a feasible join sequence is restricted to relations which are adjacent to R in the join graph. Similar restrictions hold for all following relations. These restrictions define a *precedence relation* on the set of all base relations. The graph of the precedence relation is a directed version of the join tree with R_1 being the root and all other relations directed away from the root. This shows that we actually have a *sequencing problem with tree-like precedence constraints* where the cost function satisfies the *ASI-property*.

For the unconstrained sequencing problem — that is, if we had no precedence constraints — sorting the relations according to their rank leads to an optimal sequence![2] But if precedence constraints are present, they often make it impossible to sort the relations according to their ranks. The next result provides a means to resolve the conflict of ordering according to the precedence constraints and ordering according to the rank. A *composite relation* is defined as an ordered pair (r, s) where r and s are either single or composite relations and the condition $r(r) > r(s)$ holds. Rank, cost and size of the composite relation are defined to be the respective values of the sequence rs. The precedence relation generalizes to sequences of relations in a straightforward way. In [14] it is shown that if for two composite sequences r and s, where r precedes s in the precedence tree and $r(r) > r(s)$, there is an optimal sequence with r immediately preceding s. By iterating the process of tying pairs of composite relations together whose rank and precedence stay in conflict, we eventually arrive at a sequence of composite relations which is sorted by rank. This process of iterated tying is called *normalization*.

The reader can probably already anticipate the outlines of a recursive algorithm for solving the join ordering problem with a given starting relation: One starts at the bottom of the directed join tree and works upwards. To obtain the optimal sequence for a subtree of relations where all children are chains one simply normalizes each of the chains and merges them according to the ranks. The resulting sequence is again rank ordered and we replace the subtree by the corresponding chain of composite relations. By considering every base relation as starting relation, computing the optimal sequence for this starting relation and then choosing the cheapest of these sequences, we can determine an optimal sequence for the join ordering problem.
This is basically the *IK-algorithm* described in [6]. In [12], Lawler gives an efficient implementation of this algorithm that runs in time $O(n \log n)$, using a set representation instead of the straightforward sequence representation. The following description of the IK-algorithm is taken from [6].

Algorithm NORMALIZE(S):
input: a chain of nodes S
output: chain of nodes
1 **while** there is a pair of adjacent nodes, S_1 followed
by S_2, in S such that $r(S_1) > r(S_2)$ **do**
2 Find the first such pair (starting from the beginning of S) and
replace S_1 and S_2 by a new composite node (S_1, S_2).

[2] This is not true for the join ordering problem, where the analog problem would consider cross products.

Algorithm TREE-OPT(Q):
input: tree query Q with specified root R, the relations referenced in Q,
their sizes, and the predicates of Q together with their selectivity factors
output: optimal join ordering for Q with starting relation R
1 Construct the directed tree T_R with root R.
2 **If** T_R is a single chain, **then** stop. (The desired join ordering is given by the chain.)
3 Find a subtree whose left and right subtrees are both chains.
4 Apply NORMALIZE to each of the two chains.
5 Merge the two chains into one by ordering the nodes by increasing ranks,
and go to step 2.

Algorithm IK(Q):
input: tree query Q with specified root R, the relations referenced in Q,
their sizes, and the predicates of Q together with their selectivity factors
output: optimal join ordering for Q
1 Let S be some fixed initial ordering and C the costs of S
Let $R_1, \ldots, R_n$ be the relations involved in the query.
2 **for** $i \leftarrow 1$ **to** N **do**
3 Apply TREE-OPT to the directed join tree with root R_i
4 **If** the optimal join ordering starting with R_i has better costs than C, **then**
update C and S
5 **return** S

A slightly more efficient version of this algorithm is the *KBZ-algorithm* of Krishnamurthy, Boral and Zaniolo [10]. Their algorithm has a worst case time complexity of $O(n^2)$. The idea is to reuse the computed optimal sequence for a starting relation R to compute an optimal sequence for a starting relation R' being adjacent to R in the join graph. This leads to a considerable reduction of work. For details we refer to [10].

4 Ordering Expensive Selections and Joins

Let us extend the notion of a precedence tree to capture select-join queries. Suppose we are to use a distinguished relation – say R_1 – as the starting relation in the processing tree (the leftmost leaf). Then, since no cross products are allowed, the join tree becomes a rooted tree with R_1 as the root. We can extend the directed tree to incorporate all the selection operators as follows. For every selection operator $\psi_i = \sigma_{p_i}$ relating to a single relation R_i, we add a new successor node to R and label it with ψ_i. The resulting tree defines a precedence relation among the operators ψ_i and we call it the *precedence tree* of the query with respect to the starting relation R_1. For an example see the end of this section.

For each operator ψ_i we define the cost factor d_i as

$$d_i = \begin{cases} c_i & \text{if } \psi_i \equiv \sigma_{p_i} \\ g(|R_i|) * c_{j,i} & \text{if } \psi_i \equiv \ltimes_{p_{j,i}} R_i \end{cases}$$

for $i > 1$ and $d_1 = 0$, where c_i and $c_{j,i}$ denote the cost of evaluating p_i and $p_{j,i}$ for one tuple, respectively. j is the index of the unique predecessor of R_i in the precedence tree. The size factors h_i are defined as

$$h_i = \begin{cases} f_i & \text{if } \psi_i \equiv \sigma_{p_i} \\ |R_i| * f_{j,i} & \text{if } \psi_i \equiv \ltimes_{p_{j,i}} R_i \end{cases}$$

d_i accounts for the costs incurred by applying operator ψ_i to an intermediate result R whose generation was in accordance with the precedence tree. Whenever ψ_i is applied to such an intermediate result R, we expect R to grow by a factor of h_i. We call a sequence *feasible*, if it satisfies all ordering constraints implied by the present attributes in the predicates of the operators, as expressed in the precedence tree. In the following we identify permutations and sequences.

For a sequence s we define the costs[3]

$$Cost(s) = |R_1| \sum_{i=2}^{n} F_{i-1}^{s} d_{s(i)} = |R_1| \sum_{i=2}^{n} \prod_{j=2}^{i-1} h_{s(j)} d_{s(i)},$$

where $s(i)$ is the i-the element of the operator sequence s and the intermediate result size F is given by

$$F_i^s = \prod_{j=2}^{i} h_{s(j)}$$

Then, we have to solve the following optimization problem

$$\text{minimize}_s \; [Cost(s)],$$

where the minimization is taken over all feasible sequences s.
Since R_1 just contributes a constant factor, it can easily be dropped from $Cost(s)$ without changing the optimization problem.

Our goal is to apply the IK and KBZ algorithms. Hence, we have to find a rank function for which the cost function satisfies the ASI property. We do so by first recasting the cost function into a more appropriate form. For

$$F(s) = \prod_{k \in s} h_k$$

we define the binary function C as

$$C(j, \epsilon) = C(\epsilon, j) = c_j \text{ for } j \in \{1, \ldots, n\}$$

and

$$C(s, t) = C(s', s'') + F(s)C(t', t'') \text{ for sequences } s, t$$

where $s = s's''$ and $t = t't''$ with $|s| > 1 \Rightarrow |s'| \geq 1 \wedge |s''| \geq 1$, and $|t| > 1 \Rightarrow |t'| \geq 1 \wedge |t''| \geq 1$.

[3]Empty sums equal 0, empty products 1.

A simple induction shows that C is consistent, that is, $s_1 s_2 = s_1' s_2'$ implies $C(s_1, s_2) = C(s_1', s_2')$. With the binary C being consistent, the unary C defined as

$$\begin{aligned} C(j) &= c_j \text{ for } j \in \{1, \ldots, n\} \\ C(st) &= C(s,t) \text{ for sequences } s,t \text{ with } |s| \geq 1 \ \wedge \ |t| \geq 1 \end{aligned}$$

is well-defined. Another simple proof by induction shows that the functions *Cost* and the unary C are equal for all feasible s. Summarizing, the following three lemmata hold.

Lemma 4.1 *The binary cost function C is consistent.*

Lemma 4.2 *The unary cost function C is well-defined.*

Lemma 4.3 *The unary cost function C and the cost function* Cost *are the same.*

Now, we come to the central Lemma of our paper, which will allow us to apply the IK and the KBZ algorithms to our problem of optimally ordering expensive selections and joins with expensive predicates simultaneously.

Lemma 4.4 *C satisfies the ASI property [14] with*

$$r(s) = \frac{F(s) - 1}{C(s)}$$

being the rank *of a sequence s.*

Proof: We have to proof that

$$C(ustv) \leq C(utsv) \Leftrightarrow r(s) \leq r(t)$$

for all sequences u and v. Since

$$\begin{aligned} C(ustv) &= C(us) + F(us)C(tv) \\ &= C(u) + F(u)C(s) + F(us)[C(t) + F(t)C(v)] \\ &= C(u) + F(u)C(s) + F(us)C(t) + F(us)F(t)C(v) \end{aligned}$$

the following holds

$$\begin{aligned} C(ustv) - C(utsv) &= F(u)[C(t)(F(s) - 1) - C(s)(F(t) - 1)] \\ &= F(u)C(t)C(s)[r(s) - r(t)] \end{aligned}$$

With this equation the ASI property follows for C. □

Using the results summarized in section 3, we can apply the IK- or KBZ-algorithm. Both guarantee to find an optimal solution in time $O(n^2 \log(n))$ and $O(n^2)$, respectively. Since we do only consider strict left-deep trees, *non-expensive* selections will be placed after the corresponding join in any case! To avoid this drawback, we push cheap selections down the query tree prior to the invocation of the algorithm. Note that this preprocessing step changes the sizes of some relations which must be respected.

Next, we illustrate how the IK-algorithm in case of our new rank definition works.

Example: Consider the following select-join-query involving six relations:

$$\sigma_{p_2}(\sigma_{p_3}(\sigma_{p_5}(R_1 \bowtie_{p_{1,2}} R_2 \bowtie_{p_{1,3}} R_3 \bowtie_{p_{2,4}} R_4 \bowtie_{p_{3,5}} R_5 \bowtie_{p_{5,6}} R_6)))$$

The are eight operators. Three of them are (expensive) selections and five are joins. The associated selectivities, relation sizes and cost factors are specified in the three tables below.

n_1	n_2	n_3	n_4	n_5	n_6
50	60	30	10	40	20

$f_{1,2}$	$f_{1,3}$	$f_{2,4}$	$f_{3,5}$	$f_{5,6}$	f_2	f_3	f_6
0.6	0.7	0.05	0.3	0.2	0.5	0.6	0.4

$c_{1,2}$	$c_{1,3}$	$c_{2,4}$	$c_{3,5}$	$c_{5,6}$	c_2	c_3	c_6
6	5	2	7	4	10	4	3

The join graph is shown in Figure 1(a) and Figure 1(b) shows the directed join graph rooted at the starting relation R_1, i.e., the precedence graph.

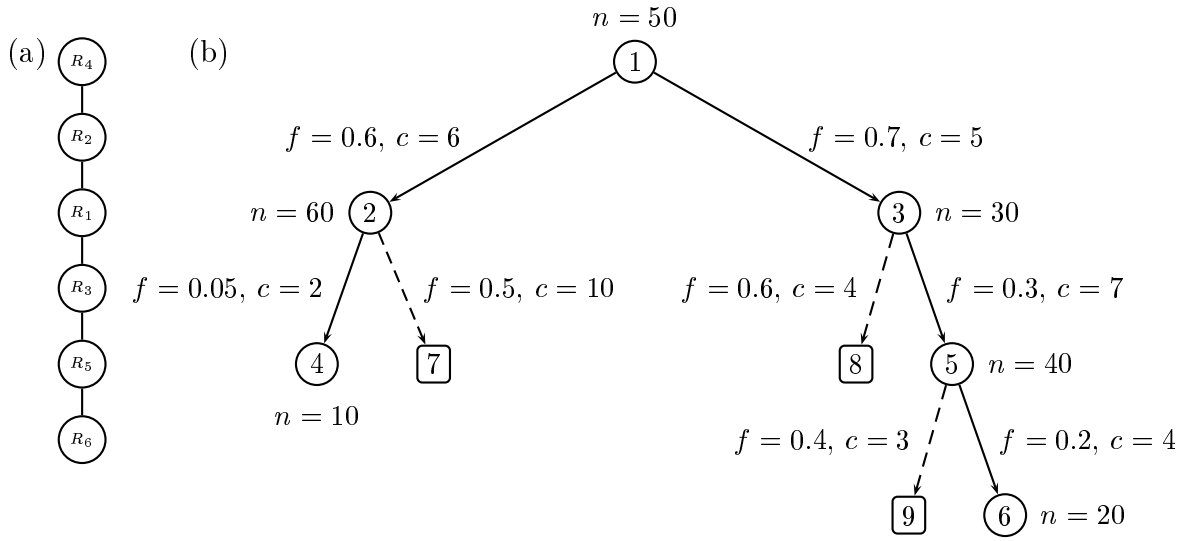


Figure 1: (a) join graph in the example query, (b) the associated precedence tree; selectivities, relation sizes and cost factors are shown. Dashed arrows pointing to square boxes indicate selections and solid arrows pointing to circular boxes correspond to joins.

Instead of considering every relation as a starting relation as the IK-algorithm does, we restrict ourselves to the single starting relation R_1. Since the nodes in the rooted tree uniquely correspond to the operators in the query, we henceforth use them interchangeably. For this we use the following coding scheme. Suppose the query has m selections and involves n base relations. Then, operator ψ_i $(1 < i \leq n)$ corresponds to the join operation $\bowtie_{p_{j,i}} R_i$ where j is the unique predecessor node in the precedence tree. For $n < i \leq n + m$, operator ψ_i corresponds to the selection operator σ_{p_j} where j is the unique predecessor

node of i in the precedence tree. Operator ψ_1 represents an exception, it corresponds to the starting relation R_1. Nodes in the precedence tree are labeled with the number of the corresponding operator, i.e. node i $(1 \leq i \leq n+m)$ corresponds to operator ψ_i. E.g., in our example, node 5 corresponds with operator ψ_5, which is the join $\bowtie_{p_{3,5}} R_5$ whereas node 7 corresponds with operator ψ_7, which is the selection σ_{p_2}.

In this example, we assume that all joins are hash-loop joins. The IK-algorithm works bottom-up. Let us first process the subtree with root 5. The sons of node 5 are the leaves 6 and 9 which are trivially ordered by rank. Node 6 is a join operator and its rank is

$$\begin{aligned} r(\psi_6) &= \frac{F(\psi_6)-1}{C(\psi_6)} \\ &= \frac{f_{5,6}n_6 - 1}{1.2 \cdot c_{5,6}} \\ &= \frac{0.2 * 20 - 1}{1.2 * 4} = 0.625 \end{aligned}$$

Node 9 is a selection operator with rank

$$\begin{aligned} r(\psi_9) &= \frac{F(\psi_9)-1}{C(\psi_9)} \\ &= \frac{f_5 - 1}{c_5} \\ &= \frac{0.4 - 1}{3} = -0.20 \end{aligned}$$

Now we can merge the two nodes. Since $r(\psi_9) < r(\psi_6)$, node 9 has to precede node 6 and we can replace the subtree rooted at 5 with the chain 5-9-6. Next, we examine whether this chain is still sorted by rank. The rank of node 5 is

$$r(\psi_5) = \frac{0.3 * 40 - 1}{1.2 * 7} = 1.31$$

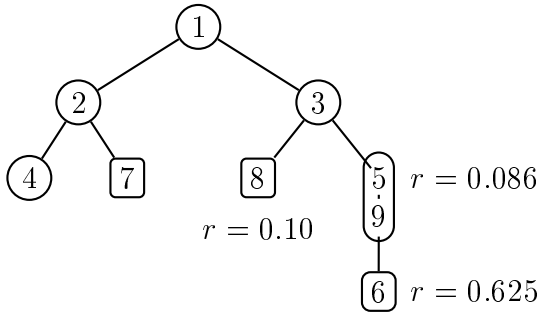
This shows that the ranks of the nodes 5 and 9 contradict their precedence and we have to tie these two nodes together as a composite node (5,9). The rank of the new node $(5,9)$ is

$$\begin{aligned} r(\psi_5\,\psi_9) &= \frac{F(\psi_5\,\psi_9)-1}{C(\psi_5\,\psi_9)} \\ &= \frac{n_5 f_{3,5} f_5 - 1}{1.2c_{3,5} + n_5 f_{3,5} c_5} \\ &= \frac{40 * 0.3 * 0.4 - 1}{1.2 * 7 + 40 * 0.3 * 3} = 0.086 \end{aligned}$$

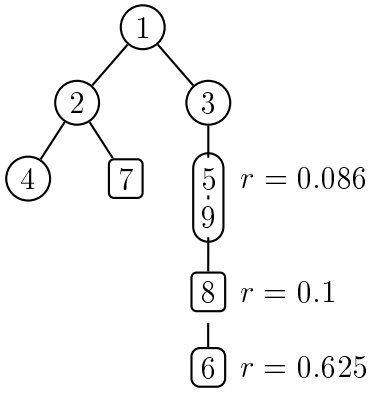
For the rank of the selection node 8 we have

$$r(\psi_8) = \frac{0.6 - 1}{4} = 0.1$$

and the new rooted join tree is



In the next step we merge node 8 and the chain consisting of the composite node (5,9) succeeded by node 6. The corresponding join tree is



The rank of the join node 3 is

$$r(\psi_3) = \frac{30 * 0.7 - 1}{1.2 * 5} = 3.333$$

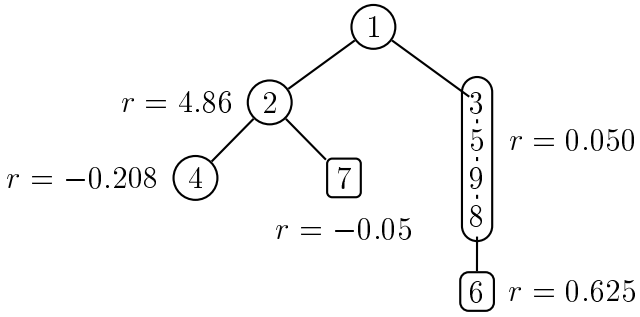
Since the nodes 3 and (5,9) have contradictory ranks, we build the new composite relation (3,5,9) with rank

$$\begin{aligned} r(\psi_3\,\psi_5\,\psi_9) &= \frac{n_3 f_{1,3} n_5 f_{3,5} f_5 - 1}{1.2 \cdot c_{1,3} + n_3 f_{1,3} \cdot 1.2 \cdot c_{3,5} + n_3 f_{1,3} n_5 f_{3,5} c_5} \\ &= \frac{30 * 0.7 * 40 * 0.3 * 0.4 - 1}{1.2 * 5 + 30 * 0.7 * 1.2 * 7 + 30 * 0.7 * 40 * 0.3 * 3} = 0.106 \end{aligned}$$

The nodes (3,5,9) and 8 still have contradictory ranks and must be tied together again. The new rank is

$$r(\psi_3\,\psi_5\,\psi_9\,\psi_8) = 0.050$$

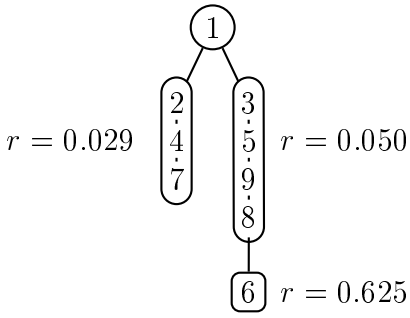
and the new join tree has the form



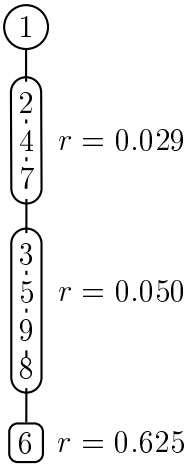
After having linearized the right subtree of R_1, we proceed with the left subtree. The ranks of nodes 2,4 and 7 are

$$r(\psi_2) = 4.86,\ r(\psi_4) = -0.208,\ r(\psi_7) = -0.05$$

We merge the subtree rooted at 2 and then normalize the resulting chain 2-4-7. The pair 2 and 4 has contradictory ranks, hence we build the composit node (2,4). Since the rank of (2,4) is 0.182, which is still greater then the rank of the succeeding node 7, we add 7 to the end of (2,4). The rank of the composite node (2,4,7) evaluates to 0.029 and the new precedence tree is



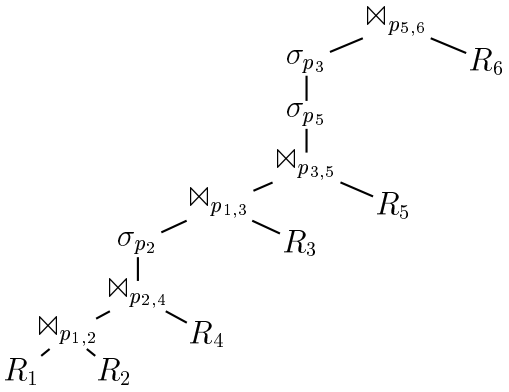
Finally, the left and right chains of R_1 are merged, yielding



As result, we have that the final sequence of operators

$$\psi_1\ \psi_2\ \psi_4\ \psi_7\ \psi_3\ \psi_5\ \psi_9\ \psi_8\ \psi_6$$

which correspond to the following optimal left-deep precessing tree for the starting relation R_1



Analogous computations are made for the precedence trees rooted at the relations R_2, R_3, R_4, R_5 and R_6. The cheapest of all the n operator sequences is the result of the IK-algorithm.

5 Conclusion

We have presented the first polynomial algorithm that optimally orders joins and selections, both with expensive predicates. However, there remain several open questions for further research. First, the question arizes whether — opposed to the two-phase approach proposed in the paper, where cheap selections are pushed before-hand in a separate phase before the algorithm is applied — there exists a polynomial one-phase algorithm. Second, in our result trees, expensive predicates can be evaluated many times, if they occur after several join operations. This can be avoided by caching the results of expensive predicates. The same is true for join predicates, if intermediate results contain duplicates. Clearly, this caching can be applied during the evaluation of the processing tree produced by our algorithm. But then, applying caching obviously changes the costs. Hence, under these circumstances, our algorithm may not produce the optimal result. The question is whether there exists a polynomial time algorithm which produces the optimal processing tree under the assumption that expensive predicate evaluations are cached.

References

[1] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 54–67, 1995.

[2] R. Gamboa D. Chimenti and R. Krishnamurthy. Towards an open architecture for $\mathcal{LDL}$. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 195–203, Amsterdam, Netherlands, August 1989.

[3] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[4] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–277, Washington, DC, 1993.

[5] J. M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, Minneapolis, Minnesota, USA, May 1994.

[6] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[7] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Disjunctive queries in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1994.

[8] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimization of boolean expressions in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 79–90, 1992.

[9] B. König-Ries, S. Helmer, and G. Moerkotte. An experimental study on the complexity of left-deep join ordering problems for cyclic queries. Technical Report 95-4, RWTH-Aachen, 1995.

[10] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 128–137, 1986.

[11] R. Krishnamurthy and C. Zaniolo. Optimization in a logic based language for knowledge and data intensive applications. In *Proc. of the Int. Conf. on Extending Database Technology*, pages 16–33, Venice, Italy, 1988.

[12] E. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.*, 2:75–90, 1978.

[13] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[14] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Math. Oper. Res.*, 4:215–224, 1979.

[15] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 314–325, 1990.

[16] W. Scheufele and G. Moerkotte. A dynamic programming algorithm to order expensive joins and selections. Forthcoming Technical Report, Dep. of Computer Science III, RWTH-Aachen, 52056 Aachen, Germany, 1996.

[17] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, 1979.

[18] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing join orders. Technischer Bericht MIP-9307, Universität Passau, September 1993.

[19] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing joins in disjunctive queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 228–238, Zurich, 1995.

[20] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The New Technologies. Computer Science Press, 1989.

[21] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.