On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products

Sophie Cluet[1] and Guido Moerkotte[2]

[1] INRIA, Domaine de Voluceau, 78153 Le Chesnay Cedex, France
[2] Lehrstuhl für Informatik III, RWTH-Aachen, 52056 Aachen, Germany

Abstract. Producing optimal left-deep trees is known to be NP-complete for general join graphs and a quite complex cost function counting disk accesses for a special block-wise nested-loop join [2]. Independent of any cost function is the dynamic programming approach to join ordering. The number of alternatives this approach generates is known as well [5]. Further, it is known that for some cost functions — those fulfilling the ASI property [4] — the problem can be solved in polynomial time for acyclic query graph, i.e., tree queries [2, 3].
Unfortunately, some cost functions like sort merge could not be treated so far. We do so by a slight detour showing that this cost function (and others too) are optimized if and only if the sum of the intermediate result sizes is minimized. This validates the database folklore that minimizing intermediate result sizes is a good heuristic. Then we show that summarizing the intermediate result sizes has the ASI property. It further motivates us to restrict the subsequent investigations to this cost function called C_{out} for which we show that the problem remains NP-complete in the general case.
Then, we concentrate on the main topic of the paper: the complexity of producing left-deep processing trees possibly containing cross products. Considering cross products is known to possibly result in cheaper plans [5]. More specifically, we show that the problem (LD-X-Star) of generating optimal left-deep processing trees possibly containing cross products is NP-complete for star queries. Further, we give an algorithm for treating star queries which is more efficient than dynamic programming. The NP-completeness of LD-X-Star immediately implies the NP-completeness for the more general tree queries.

1 Introduction

Not only in deductive databases but also in object bases, where each single dot in a path expression corresponds to a join, the optimizer is faced with the problem of ordering large numbers of joins. The standard and, maybe even today, prevailing method to determine an optimal join order is dynamic programming [6]. In 1984, the proof for the NP-completeness of join ordering for cyclic queries was presented together with an algorithm ordering joins for tree queries optimally in $O(n^2 \log n)$ time [2].[3] This algorithm was subsequently improved to $O(n^2)$

[3] n denotes the number of relations.

time complexity [3]. A heuristic for join ordering applying this algorithm to the minimal spanning tree of the join graph started the investigation of non-trivial heuristics for join ordering [3]. However, these algorithms rejected cross-products. Lately, Ono and Lohman gave real world examples that abandoning cross products can lead to more expensive plans than those which incorporate a cross product [5]. Furthermore, they gave $n2^{n-1} - n(n+1/2)$ as the number of processing trees generated by dynamic programming in order to derive the cheapest left-deep processing tree possibly containing a cross product.

The question arises whether there exists a polynomial algorithm for treating the problem of generating optimal left-deep trees considering cross products. For general query graphs, this is unlikely, since already the generation of ordinary left-deep trees without cross products is NP-complete. For tree queries, the complexity of generating optimal left-deep trees possibly containing cross products is — so far — an open question. In this paper, we show that even for star shaped query graphs, which are a special case of a general tree query, the optimization problem is NP-complete.

Of course, every complexity result for an optimization problem highly depends on the chosen cost function. For example, in [2], a complex cost function counting disk accesses for a special block-wise nested-loop algorithm was used. Furthermore, the proof exploits some special features of this cost function, not present in other cost functions. Within this paper, we concentrate on a very easy cost function: the sum of the sizes of the intermediate results. Let us call this cost function C_{out}. This choice is motivated by several facts. First, it is a very simple cost function. Second, as will be shown, NP-completeness for C_{out} implies NP-completeness for other cost functions, too. More specifically, we will show that optimizing other cost functions is equivalent to optimizing C_{out} which formally justifies the database folklore that optimizing intermediate result sizes is a good thing to do.

The paper is organized as follows. In the next section, we first give a short introduction to the problem and present our notation. We then motivate our decision for considering C_{out} as a basic cost function for our complexity investigation on generating optimal left-deep processing trees possibly containing cross products. Section 3 then presents the proof of the NP-completeness of the LD-X-Star problem and ends with the sketch of an algorithm more efficient than dynamic programming. Section 4 presents open problems for future research.

2 Preliminaries and First Results

2.1 The Join-Ordering Problem

Let us first introduce the join-ordering problem. An instance of a join-ordering problem is fully described by the following parameters. First, n relations $R_1, \ldots, R_n$ are given. Associated with each relation is its *size* $|R_i|$, also denoted by n_i. Second, a *query graph* whose nodes are the relations and whose edges connect two relations by an undirected graph constitutes the second parameter.

The edges of the query graph are labeled by their according *selectivity*. Let (R_i, R_j) be an edge in the query graph. Then, the associated selectivity is $f_{i,j}$. We assume that $0 < f_{i,j} < 1$. If there is no edge between R_i and R_j, i.e., we have a cross product, we assume $f_{i,j} = 1$.

Since there exist several implementations for a join, there exist several cost functions. The most common implementations of a join operator are

1. hash loop join
2. sort merge join
3. nested loop join

The according cost functions are usually given as (see, e.g., [3]):

$$\begin{aligned} C_{hl}(R_i \bowtie R_j) &:= |R_i|1.2 \\ C_{sm}(R_i \bowtie R_j) &:= |R_i|(1 + \log(|R_i|)) + |R_j|(1 + \log(|R_j|)) \\ C_{nl}(R_i \bowtie R_j) &:= |R_i||R_j| \end{aligned}$$

These cost functions are mostly applied for main memory databases.

Sometimes, only the costs of producing the intermediate results is counted for. This makes sense if, e.g., the intermediate results must be written to disk, since then the costs for accessing the disk clearly outweigh the CPU costs for checking the join predicate. This cost function is called C_{out}:

$$C_{out}(R_i \bowtie R_j) := |R_i||R_j|f_{i,j}$$

Compared to the cost function used in [2] to proove the NP-completeness of join ordering, this cost function is very simple. Hence, the question arises if join ordering is still NP-complete for this simple cost function. As we will see, the answer is yes. Before we proove this, we need some more definitions.

For all cost functions, we will assume a binary equivalent whose input are just the sizes n_i and n_j of the according relations R_i and R_j. For example, for C_{out}, we have

$$C_{out}(n_i, n_j) := n_i n_j f_{i,j}$$

The problem considered in this paper is the complexity of computing an optimal *join-order*, i.e., a left-deep join-processing tree. More formally, given an instance of the join-ordering problem, we ask for a sequence s of the n relations such that for some cost function C_x the total cost defined as

$$C(s) := \sum_{i=2}^{n} C_x(|s_1 \ldots s_{i-1}|, s_i)$$

is minimized. Some definitions are needed in order to understand the definition of the cost function. Producing left-deep trees is equivalent to fixing a permutation π of the relations, or fixing a sequence $s_1, \ldots, s_n$ of all relations. The latter is what we do. By $|s_1, \ldots, s_i|$, we denote the intermediate result size of joining the relations $s_1, \ldots, s_i$. For a single relation s_i, we also write within cost functions s_i instead of $|s_i|$ or n_{s_i} in order to denote its size.

2.2 The ΣIR Property

The goal of this section is to cut down the number of cost functions which have to be considered for optimization. More specifically, we will argue that it is already quite interesting to just consider C_{out}. For this, we define an equivalence relation on cost functions.

Definition 1. Let C and C' be two cost functions. Then
$C \equiv C' :\Leftrightarrow (\forall s\ C(s) \text{ minimal} \Leftrightarrow C'(s) \text{ minimal})$
Obviously, $\equiv$ is an equivalence relation.

Next, we overload the binary C_x cost functions for a single join with those resulting from applying it to each join necessary to join a sequence s of relations. For example, we define

$$C_{out}(s) := \sum_{i=2}^{n} |s_1, \ldots, s_i|$$

Now we can define the ΣIR (Sum of Intermediate Results) property.

Definition 2. A cost function C is $\Sigma IR :\Leftrightarrow C \equiv C_{out}$.
Let us consider a very simple example. The last element of the sum in C_{out} is the size of the final join (all relations are joined). This is not the case for the following cost function:

$$C'_{out}(s) := \sum_{i=2}^{n-1} |s_1, \ldots, s_i|$$

Obviously, we have C'_{out} is ΣIR. The next observation shows that we can construct quite complex ΣIR cost functions:

Observation 3. *Let C_1 and C_2 be two ΣIR cost functions. For non-decreasing functions $f_1 : R \to R$ and $f_2 : R \times R \to R$ and a constant c we have that*
$C_1 + c$
$C_1 * c$
$f_1 \circ C_1$
$f_2 \circ (C_1, C_2)$
are ΣIR. Here, $\circ$ denotes function composition and $(\cdot, \cdot)$ function pairing.
There are of course many more possibilities of constructing ΣIR functions.

For the above cost functions C_{hl}, C_{sm}, and C_{nl}, we derive for a sequence s of relations

$$C_{hl}(s) = \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}|$$

$$C_{sm}(s) = \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}|(1 + \log(|s_1, \ldots, s_{i-1}|)) + \sum_{i=1}^{n} |s_i|(1 + \log(|s_i|))$$

$$C_{nl}(s) = \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| * s_i$$

We investigate which of these cost functions are ΣIR.

Let us consider C_{hl} first. From

$$\begin{aligned} C_{hl}(s) &= \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}| \\ &= 1.2|s_1| + 1.2\sum_{i=2}^{n-1} |s_1, \ldots, s_i| \\ &= 1.2|s_1| + 1.2C'_{out}(s) \end{aligned}$$

and Observation 3, we conclude that C_{hl} is ΣIR for a fixed relation to be joined first. If we can optimize C_{out} in polynomial time, then we can optimize C_{out} for a fixed starting relation. Indeed, by trying each relation as a starting relation, we can find the optimal. Thus, we stay within PTIME.

Now, consider C_{sm}. Since

$$\sum_{i=2}^{n} |s_1, \ldots, s_{i-1}|(1 + log(|s_1, \ldots, s_{i-1}|))$$

is minimal if and only if

$$\sum_{i=2}^{n} |s_1, \ldots, s_{i-1}|$$

is minimal and $\sum_{i=1}^{n} |s_i|(1+\log(|s_i|))$ is independent of the order of the relations within s — that is constant — we conclude that C_{sm} is ΣIR.

Last, we have that C_{nl} is not ΣIR. To see this, consider the following counter example with three relations R_1, R_2, and R_3 of sizes 10, 10, and 100, resp. The selectivities are $f_{1,2} = \frac{9}{10}$, $f_{2,3} = \frac{1}{10}$, and $f_{1,3} = \frac{1}{10}$. Now,

$$\begin{aligned} |R_1R_2| &= 90 \\ |R_1R_3| &= 100 \\ |R_2R_3| &= 100 \end{aligned}$$

and

$$\begin{aligned} C_{nl}(R_1R_2R_3) &= 10 * 10 + 90 * 100 = 9100 \\ C_{nl}(R_1R_3R_2) &= 10 * 100 + 100 * 10 = 2000 \\ C_{nl}(R_2R_3R_1) &= 10 * 100 + 100 * 10 = 2000 \end{aligned}$$

We see that $R_1R_2R_3$ has the smallest intermediate result size but produces the highest cost. Hence, C_{nl} is not ΣIR.

The rest of the section deals with the complexity of producing optimal left-deep join trees, that is, we do not consider cross products yet. The next subsection deals with the general problem, Sect. 2.4 treats tree queries.

2.3 On the Complexity of Optimizing C_{out}

Since the cost function C_{out} is much simpler than the cost function used to proove the NP-completeness of the general join ordering problem [2], we give a simple sketch of a proof that the join-ordering problem remains NP-complete even if the simple cost function C_{out} is considered. This seems necessary, since the proof in [2] makes use of some special features of the cost functions which are absent in C_{out}.

Theorem 4. *The join-ordering problem with the cost model C_{out} is NP-complete.*

Proof (sketch of). Obviously the join-ordering problem $\in$ NP. We will restrict the join-ordering problem to the Clique problem which is known to be NP-complete [1]. (The question asked in the Clique problem is, whether a graph G contains a clique of at least size K or not.) We will represent all n nodes in a graph G by relations of cardinality 1. If there is an edge between two nodes in G, then the selectivity between the corresponding relations is set to $\frac{1}{2}$. Let G be a graph where each relation has at least one connection to another relation. Now, it is obvious that if there is a clique of size K, then the optimal sequence must start by the K relations involved in this clique.

From this it also follows that the more general problem, where cross products are considered, is NP-complete too.

2.4 Tree Queries, C_{out}, and the ASI Property

In this subsection, we assume that the query graph is acyclic, i.e., a tree. Still, we do not consider cross products. For two special cases of a tree, we know the number of alternatives generated by the dynamic programming approach to join-ordering [5]. For chain queries, i.e., where the query graph is a chain, dynamic programming generates $(n-1)^2$ alternatives. For star queries, i.e., where there exists one relation to which all other relations are connected and there exists no other connection, dynamic programming generates $(n-1)2^{(n-2)}$ nodes. Note that the dynamic programming approach is independent of the chosen cost function. Nevertheless, the question arises, whether one can do better with specialized algorithms, if something is known about the cost functions.

The answer is yes, if the cost function has the ASI (Adjacent Sequence Interchange) property [4]. For these cost functions, there exist polynomial time algorithms (the fastest is $O(n^2)$) producing optimal left-deep trees for tree queries [2, 3]. Let us shortly review this approach.

From a query graph, a *precedence tree* is constructed by arbitrarily choosing one relation as a root and directing the edges away from it. The main idea then is, to produce for every possible precedence graph the optimal solution and take the cheapest of these. The optimal solution for a precedence graph is obtained by an algorithm that is an adaptation of the Monma/Sydney procedure for *job sequencing* with precedence constraints[4]. The ASI property allows to assign a rank to each relation such that if a sequence of relations is ordered by rank, it is optimal. Furthermore, if two relations linked by a precedence edge have unordered ranks, it guaranties that the two relations have to stick together in the optimal sequence. Thus, the idea is to (i) stick together relations that cannot be parted in the optimal sequence and (ii) merge the different chains using rank ordering.

Let us now come back to this ASI property. The selectivity $f_{i,j}$ of an edge within the original query graph corresponds to a selectivity of an edge within the precedence graph. For notational convenience, this selectivity is renamed to f_j, if the relation R_i is the (immediate) predecessor of the relation R_j within the precedence graph. We will rename the root node to R_1 and define f_1 to be 1.

Assume that we can write a cost function in the following form where f_i is used for denoting the selectivity attached to the relation s_i, given s_{i-1}.

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

for some arbitrary functions g_j. Then, for sequences S_1 and S_2 of relations, we can define this cost function recursively by

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

with

$$\begin{aligned} T(\epsilon) &= 1 \\ T(s) &= \prod_{i \in [1,n]} (f_i * s_i) \end{aligned}$$

We have that C is well-defined and, for all sequences s, $C(s) = Cost(s)$.

Definition 5. A cost function C has the ASI property, if and only if there exists a rank function $rank(s)$ for sequences s, such that for all sequences a and b and all non-empty sequences v and u the following holds:

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

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

Lemma 6. *Let C be a cost function which can be written in the above form. Then C has the ASI property for the rank function*

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

for nonempty sequences s.

Since C_{hl} and C_{nl} can be written in the above mentioned special form for cost functions, they have the ASI property and tree queries involving these cost functions can be solved in polynomial time [3]. Further, C_{sm} cannot be written in the above form. This is the reason why it was so far abandoned from being treated by the Monma/Sidney-procedure [3]. Nevertheless, since C_{sm} is ΣIR, it suffices to show that C_{out} has the ASI property. If so, we can also treat tree queries involving C_{sm} in polynomial time. But obviously, C_{out} can be written in the above form with $g_j(s_j) = f_j s_j$. Hence,

Observation 7. *C_{out} has the ASI property.*

Summarizing, for tree queries, especially for chain and star queries, optimal left-deep trees can be constructed in polynomial time for all cost functions mentioned in this section; moreover, for all cost functions being ΣIR or having the ASI property.

Let us take a look at the *rank* function for C_{out}, in case of one relation only:

$$rank(s_i) = \frac{T(s_i) - 1}{C(s_i)} = \frac{f_i s_i - 1}{f_i s_i}$$

Since $\frac{x-1}{x}$ is strictly increasing, ordering by *rank* is the same as ordering by $f_i s_i$, for sequences consisting of a single relation.

All this only holds if we do not consider cross products. But as pointed out by Ono and Lohman, introducing cross products can lead to considerably cheaper plans [5]. Consequently, the next section deals with the complexity of constructing optimal left-deep join trees where some joins may in fact be cross products. Further, the dynamic programming approach considers $n2^{n-1} - \frac{n(n+1)}{2}$ — independently of the join graph [5] and the cost function. The question arises, whether we can do better for tree queries and the C_{out} cost function.

3 Star Queries

Consider a star query where the inner relation — or *center* is called R_0 and the relations $R_1, \ldots, R_n$ are the *satellites*. In any plan, there must exist a k such that some relations $s_1, \ldots, s_k$ are connected by a cross product, then, R_0 is joined and subsequently all the missing relations $s_{k+1}, \ldots, s_n$. The only cost function we will consider for the rest of the paper is C_{out}. Hence, we will write C instead of C_{out}.

The following is helpful in reducing the search space to be considered in order to find an optimal solution.

Lemma 8.
Any optimal sequence must obey

$$\begin{array}{ll} (1) & s_1 \leq s_2 \leq \ldots \leq s_k \\ (2) & f_{s_{k+1},0} s_{k+1} \leq f_{s_{k+2},0} s_{k+2} \leq \ldots \leq f_{s_n,0} s_n \end{array}$$

We call a sequence $s_1, \ldots, s_k$ size-ordered, if and only if $s_1 \leq \ldots \leq s_k$, and we call it ρ-ordered, if and only if $f_{s_1,0} s_1 \leq \ldots \leq f_{s_k,0} s_k$. Instead of $f_{s_i,0}$, we will also write simply $f_{i,0}$ or f_i, if a sequence s is implied by the context.

Something can also be said about the placement of R_0:

Lemma 9.

$$
\begin{aligned}
C(s_1 \cdots s_k R_0 s_{k+1} \cdots s_n) &< C(s_1 \cdots s_{k-1} R_0 s_k s_{k+1} \cdots s_n) \\
&\Leftrightarrow \\
s_k &< \prod_{j<k} f_j n_0
\end{aligned}
$$

Also, on the relations to the left and right or R_0, we have:
Lemma 10.

$$
\begin{aligned}
C(s_1 \cdots s_k R_0 s_{k+1} \cdots s_n) &< C(s_1 \cdots s_{k-1} s_{k+1} R_0 s_k \cdots s_n) \\
&\Leftrightarrow \\
s_k + s_k f_k (\prod_{j<k} f_j) n_0 &< s_{k+1} + s_{k+1} f_{k+1} (\prod_{j<k} f_j) n_0 \\
&\Leftrightarrow \\
s_k - s_{k+1} &< [s_{k+1} f_{k+1} - s_k f_k] n_0 \prod_{j<k} f_j
\end{aligned}
$$

We next consider the cost differences of some more complex swap operations on sequences.
Lemma 11. *Let*

$$
\begin{aligned}
s &= s_1 \cdots s_{k-1} s_k R_0 s_{k+1} \cdots s_{l-1} s_l s_{l+1} \cdots s_n \\
s' &= s_1 \cdots s_{k-1} s_l R_0 s_{k+1} \cdots s_{l-1} s_k s_{l+1} \cdots s_n
\end{aligned}
$$

Then

$$
\begin{aligned}
C(s) = &\sum_{i=2}^{k-1} \prod_{j \le i} s_j \\
&+ (\prod_{j<k} s_j) n_k \\
&+ (\prod_{j<k} s_j) s_k (\prod_{j<k} f_j) f_k n_0 \\
&+ [(\prod_{j<k} s_j) s_k (\prod_{j<k} f_j) f_k n_0] * (\sum_{i=k+1}^{l-1} \prod_{k+1 \le j \le i} f_j s_j) \\
&+ [(\prod_{j<k} s_j) s_k (\prod_{j<k} f_j) f_k n_0] * (\prod_{k+1 \le j < l} f_j s_j) * f_l s_l \\
&+ [[(\prod_{j<k} s_j) s_k (\prod_{j<k} f_j) f_k n_0] (\prod_{k+1 \le j < l} f_j s_j) f_l s_l] * \sum_{i=l+1}^{n} \prod_{l<j \le i} f_j s_j
\end{aligned}
$$

Analogously for s'. Thus

$$C(s) < C(s')$$
$$\Leftrightarrow$$
$$s_k + s_k f_k n_0 \prod_{j<k} f_j (1 + \sum_{i=k+1}^{l-1} \prod_{k+1 \leq j \leq i} f_j s_j)$$
$$<$$
$$s_l + s_l f_l n_0 \prod_{j<k} f_j (1 + \sum_{i=k+1}^{l-1} \prod_{k+1 \leq j \leq i} f_j s_j)$$

Hence, $s_k < s_l \wedge s_k f_k < s_l f_l \Rightarrow C(s) < C(s')$.
This is an important observation since we can now derive that if there is no contradiction between the order implied by the size and the one implied by the $s_i f_i$, then ordering the satellites by their size already results in an optimal order, except that the placement of R_0 within the sequence is unknown. But placing R_0 can easily be done using Lemma 9. To summarize:

Theorem 12. *If there is no contradiction between the size-rank and the ρ-rank of the satellites of the star query, optimal left-deep processing trees possibly containing cross products can be generated in polynomial time ($O(n \log(n))$).*

This already looks promising. But the following theorem is slightly discouraging for general star queries. Denote by *LD-X-Star* the problem of generating an optimal join sequence under the consideration of cross products.

Theorem 13. *LD-X-Star is NP-complete.*

Proof. Obviously, LD-Star $\in$ NP. We show that LD-Star is NP-hard by reducing 3DM to LD-Star.

Let

$$\begin{aligned} X &= \{x_1, \ldots, x_q\} \\ Y &= \{y_1, \ldots, y_q\} \\ Z &= \{z_1, \ldots, z_q\} \\ M &= \{m_1, \ldots, m_n\} \subseteq X \times Y \times Z \end{aligned}$$

be an instance of 3DM. W.l.o.g. we assume

1. $n > q$ and
   (Other instances can be checked immediately anyway.)
2. z_q occurs at least in two elements of M.
   (If z_q does not occur at all, we are done. If it occurs only once, we can reduce it to a problem of size $q-1$ and $n-1$.)

We will number the x_i, y_i and z_i subsequently and use the symbols to identify the numbers.

Denote by p_i the i-th prime number greater than or equal to 5. Then we define

$$\begin{array}{lll} x_i & := p_i & 1 \le i \le q \\ y_i & := p_{q+i} & 1 \le i \le q \\ z_i & := p_{2q+i} & 1 \le i < q \\ A & := \prod_{j=1}^{q} x_j \prod_{j=1}^{q} y_j * \prod_{j=1}^{q-1} z_j & \\ z_q & := A^2 & \\ B & := A^3 & \end{array}$$

Note that we can apply a sieve method to get all the needed polynomial number of primes in polynomial time.

Map each $(a_j, b_j, c_j) \in M$, $1 \le j \le n$, to a relation R_j and define

$$n_j := a_j * b_j * c_j$$
$$f_j := \frac{1}{n_j^2}$$

Last, define for relation R_0 its size $n_0 := B^2$. Again, these numbers can be constructed in polynomial time. We will show that:

there exists a solution to 3DM $\Leftrightarrow$ the optimal solution $sR_0\overline{s}$ of the transformed LD-Star problem fulfills $||s|| = q$ and $|s| = B$.

Clearly, if there is no solution to the 3DM problem, no such $sR_0\overline{s}$ exists. Indeed, the conditions we imposed on s ($||s|| = q$ and $|s| = B$) guaranty that it has to contain the 3DM solution. Hence, it remains to proove that if 3DM has a solution, then the optimal join order $sR_0\overline{s}$ fulfills $||s|| = q$ and $|s| = B$ where $s = s_1 \ldots s_l$ is size-ordered and $\overline{s} = s_{l+1} \ldots s_n$ is ρ-ordered.

Let us first compute the cost $C(sR_0\overline{s})$ using the recursive definition given in the previous section. Knowing that $||s|| = q$ and $|s| = B$, we have:

$$\begin{aligned} C(s) &= \sum_{i=2}^{q} \prod_{j=1}^{i} s_j \\ &= s_1 s_2 + s_1 s_2 s_3 + \ldots + s_1 s_2 \ldots s_q \\ &= s_1 s_2 + s_1 s_2 s_3 + \ldots + |s| \\ \frac{C(s)}{|s|} &= \frac{1}{s_3 s_4 \ldots s_q} + \frac{1}{s_4 \ldots s_q} + \ldots + \frac{1}{s_q} + 1 \\ &= 1 + \sum_{i=3}^{q} \prod_{j=1}^{q} \frac{1}{s_j} \\ &=: 1 + C_1(s) \\ C(\overline{s}) &= \sum_{i=q+1}^{n} \prod_{j=q+1}^{i} \frac{1}{s_j} \\ C(sR_0\overline{s}) &= C(s) + T(s)C(R_0) + T(sR_0)C(\overline{s}) \\ &= |s|(1 + C_1(s)) + \frac{|s| n_0}{s_1^2 s_2^2 \ldots s_q^2} + \frac{s_1 s_2 \ldots s_q n_0}{s_1^2 s_2^2 \ldots s_q^2} C(\overline{s}) \end{aligned}$$

$$\begin{aligned} &= |s|(1 + C_1(s)) + \frac{|s|^3}{|s|^2}(1 + C(\overline{s})) \\ &= |s|(2 + C_1(s) + C(\overline{s})) \end{aligned}$$

Now, note that the conditions imposed on s guaranties that z_q appears once and only once in s and, accordingly, at least once in $\overline{s}$ (since z_q occurs at least in two elements of M). Thus, knowing that s is size ordered and $\overline{s}$ is ρ-ordered, we have:

$$\begin{aligned} C_1(s) &= \frac{1}{z_q y_{s_q} z_{s_q}}(1 + \frac{1}{z_{s_{q-1}} y_{s_{q-1}} z_{s_{q-1}}}(1 + \ldots)) \\ C(\overline{s}) &= \frac{1}{z_q y_{\overline{s}_1} z_{\overline{s}_1}}(1 + \frac{1}{z_{\overline{s}_2} y_{\overline{s}_2} z_{\overline{s}_2}}(1 + \ldots)) \end{aligned}$$

Now, knowing that all x_i's y_i's are bigger than 4, we can derive the following upper-bound for $C(sR_0\overline{s})$:

$$C(sR_0\overline{s}) < |s|[2 + \frac{1}{4z_q}] \tag{1}$$

Let us now estimate the cost of any $C(uR_0\overline{u})$ for size-ordered u and ρ-ordered $\overline{u}$ where

$$\begin{aligned} u &= u_1 \ldots u_k \\ \overline{u} &= u_{k+1} \ldots u_n \end{aligned}$$

and $1 < k < n$ (the cases $u = \epsilon$ and $\overline{u} = \epsilon$ trivially result in costs higher than $C(sR_0\overline{s})$).

Analogously to $sR_0\overline{s}$, we have:

$$\begin{aligned} C_1(u) &:= \sum_{i=3}^{k} \prod_{j=i}^{k} \frac{1}{u_j} \\ C(\overline{u}) &:= \sum_{i=k+1}^{m} \prod_{j=k+1}^{i} \frac{1}{u_j} \\ C(uR_0\overline{u}) &= |u|(1 + C_1(u)) + \frac{n_0}{|u|}(1 + C(\overline{u})) \\ &= |u|(1 + C_1(u)) + \frac{|s|^2}{|u|}(1 + C(\overline{u})) \end{aligned}$$

With K defined as

$$K := \frac{|s|}{|u|}$$

we have

$$\begin{aligned} C(uR_0\overline{u}) &= \frac{1}{K}|s|(1 + C_1(u)) + K|s|(1 + C(\overline{u})) \\ &= |s|[K + \frac{1}{K} + \frac{1}{K}C_1(u) + KC(\overline{u})] \end{aligned} \tag{2}$$

Note that $K + \frac{1}{K}$

– ≥ 2

– strictly decreasing for $K < 1$

– strictly increasing for $K > 1$

For $K \geq \frac{3}{2}$ and $K \leq \frac{1}{2}$, it follows from (1) and (2) that $C(uR_0\overline{u}) > C(sR_0\overline{s})$.

Hence, we can assume that $\frac{1}{K} > \frac{1}{2}$. Assume that u_k does not contain z_q. Then $\frac{1}{K}C_1(u) \geq \frac{1}{2}\frac{1}{z_{q-1}x_q y_q} \geq \frac{1}{2z_q}$ and, hence, $C(uR_0\overline{u}) > C(sR_0\overline{s})$.

Assume $u_k = z_q x_{u_k} y_{u_k}$. Then we define P such that $K = \frac{|s|}{|u|} = \frac{|s|}{P z_q}$. Then $K = \frac{A}{P}$. Since both A and P are odd, we can conclude for $A \neq P$ that either $A \geq P + 2$ or $A \leq P - 2$. For the first case, we have

$$K = \frac{A}{P} \geq \frac{P+2}{P} \geq 1 + \frac{2}{P} > 1 + \frac{2}{A} = \frac{A+2}{A}$$

We now proove that for $K = \frac{A+2}{A}$:

$$K + \frac{1}{K} > 2 + \frac{1}{4z_q} \tag{3}$$

Since $K > 1$ and, hence, $K + \frac{1}{K}$ is strictly increasing, $C(uR_0\overline{u}) > C(sR_0\overline{s})$.

Equation (3) now follows from (remember that $z_q = A^2$):

$$\begin{aligned}
\frac{A+2}{A} + \frac{A}{A+2} &> 2 + \frac{1}{4A^2} \\
&\Leftrightarrow \\
\frac{2A^2 + 4A + 4}{A^2 + 2A} &> \frac{8A^2 + 1}{4A^2} \\
&\Leftrightarrow \\
8A^4 + 16A^3 + 16A^2 &> 8A^4 + 16A^3 + A^2 + 2A \\
&\Leftrightarrow \\
15A^2 - 2A &> 0 \\
&\Leftrightarrow \\
A(15A - 2) &> 0
\end{aligned}$$

The latter is true since $A > 1$.

Now consider the second case. Here, we had $A \leq P - 2$ and hence $\frac{P}{A} \geq \frac{A+2}{A}$. Thus $K = \frac{A}{P} \leq \frac{A}{A+2}$. From (3) we know that $K + \frac{1}{K} > 2 + \frac{1}{4z_q}$ for $K = \frac{A+2}{A}$, and hence also for $K = \frac{A}{A+2}$. Since $K < 1$ and, hence, $K + \frac{1}{K}$ is strictly decreasing, $C(uR_0\overline{u}) > C(sR_0\overline{s})$.

Summarizing, any optimal sequence $uR_0\overline{u}$ must obey $|u| = A^3$ with one occurrence of z_q in u. Since $|u| = A^3$ and $u_k = z_q x_{u_k} y_{u_k}$ implies that $||u|| = q$, we are done.

From the above lemmata, we can infer an algorithm with complexity in the worst case of $\sum_{i=0}^{c} \overline{ni}$ where c is the number of cross products to consider. All we have to do is to generate systematically subsets of the relations $R_1, \ldots, R_n$, which will precede R_0 within the join graph. Since we can stop as soon as the rank of R_0 becomes smaller than the size of any of the remaining relations, we can be sure that we do not consider sets containing more relations than necessary. Further, if the rank of R_0 is still bigger than the size of a remaining relation, we can expand the set. Hence, we have a procedure that is within the claimed complexity class. Note, that it is faster than dynamic programming, if the number of cross products is less than the total number of relations.

4 Conclusion and Future Work

We have shown that the problem of constructing optimal left-deep processing trees for star queries is NP-complete. Hence, it is NP-complete for tree queries. The first open question to answer is whether there exists a polynomial algorithm for treating chain queries, i.e. those, whose join graph is a chain, or whether this problem is NP-complete, too.

Also open for future research is the complexity of generating optimal bushy trees, where except for number of plans dynamic programming generates [5], nothing is known.

References

1. Garey, J., Johnson, D.: Computers and Intractability: a Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)
2. Ibaraki, T., Kameda, T.: Optimal Nesting for computing n-Relational Joins. ACM. Trans. on Database Systems, **9(3)** (1984) 482—502
3. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of Nonrecursive Queries. Proc. Int. Conf. Very Large Databases (VLDB), (1986), 128—137
4. Monma, C., Sidney, J.: Sequencing with Series-Parallel Precedence Constraints. Math. Oper. Res,. **4** (1979), 215—224
5. Ono, K., Lohman, G.: Measuring the Complexity of Join Enumeration in Query Optimization. Proc. Int. Conf. Very Large Databases (VLDB), (1990), 314—325
6. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access Path Selection in a Relational Database Management System. Proc. Int. Conf. Management of Data (ACM SIGMOD), (1979), 23—34