International Journal of Foundations of Computer Science Vol. 9 No. 4 (1998) 399–429
© World Scientific Publishing Company

SORT ORDER PROBLEMS IN RELATIONAL DATABASES

KIM S. LARSEN
Department of Mathematics and Computer Science, Odense University
Campusvej 55, DK-5230 Odense M, Denmark

Received 16 June 1997
Revised 31 July 1998
Communicated by D. F. Hsu

ABSTRACT

A relation of degree k can be sorted lexicographically in $k!$ different ways, i.e., according to any one of the possible permutations of the schema of the relation. Such permutations are referred to as sort orders. When evaluating unary and binary relational algebra operators using sort-merge algorithms, sort orders fulfilling the constraints enforced by the operators are chosen for the operand relations. The relations are then sorted according to their assigned sort orders, and the result is obtained by merging. Should the operands already be sorted according to one of the permissible sort orders, then only a merging is required. The sort order of the result will depend on the sort orders of the operands.

When evaluating whole relational algebra expressions, the result of one operation will be used as an operand to the next. It is desirable to choose sort orders in such a way that the result of one operation will automatically fulfill the requirements of the next. In general, one would like to find a minimal number of operators in the expression for which this cannot be obtained, bearing in mind the overall goal of minimizing the total work.

We show that this problem is NP-hard, and that the corresponding decision problem is NP-complete. However, most simplifications of the original problem give rise to efficient algorithms. In fact, most frequently occurring queries can be analyzed in linear time in the size of the query. This is due to the fact that only a very limited number of subsets of all permutations of schemas can be encountered in the algorithms, which means that compact representations for these subsets can be found.

Keywords: Databases, relational algebra, query optimization, sort orders.

1. Introduction

We consider the problem of determining an optimal sort order[a] assignment for a relational algebra expression such that it can be evaluated fast using sort-merge techniques. If the operands of an expression have been assigned incomparable sort orders, or if the sort order produced by the operator given the sort orders of the

[a]The name "sort order" is somewhat unsatisfactory, since it is merely a sequence of attribute names, and, thus, not an order (such as a partial order). However, this is the term used in the literature.

argument(s) cannot be used by the next operator, we consider this to be a *violation*, since it will be necessary to resort or make a new index. We define an *optimal* sort order assignment to be one with a minimal number of violations.

This problem was first considered by Smith and Chang[10]. They present a simple linear-time algorithm with the purpose of approximating the exact solution. Viewing the relational algebra expression as a parse tree, their algorithm makes one pass up the tree and one down. The possibilities in their algorithm are limited since they only pass on one candidate sort order to the surrounding expression. Similar heuristics have been used in commercial query optimizers, though this has not been documented very well in the form of scientific papers (see Selinger et. al.[7], though).

In contrast to the heuristics-based techniques used up until now, we present algorithms for finding exact solutions. In greater detail, our results are the following: First of all, we characterize the problem, i.e., we prove exactly when the problem becomes NP-hard.[b] Next, we have designed algorithms for different variations of the original problem. It turns out that we can analyze queries where no relation name appears more than once in linear time. To be precise, the analysis can be performed in time $O(k \log(k) n)$, where n is the size of the query and k is the maximal degree of any relation in the query (the number of attributes in the largest schema). Clearly, this is a class of very frequently occurring queries.

For the class of queries where a relation name can appear more than once the problem is NP-hard. However, if there exists a sort order with no violations, it is possible to find such a sort order in time $O(k^2 \log^2(k) n^2)$. When the minimal number of violations increases, the problem becomes computationally harder. Using reduction techniques, it is possible to solve the problem in time $O(\binom{n}{p} k^2 \log^2(k) n^2)$, if there is a solution with at most p violations. This becomes quite unacceptable as p grows (up to $\frac{n}{2}$). However, if there are many violations, sort-merge techniques will not be efficient anyway, so alternative methods should be sought.

The problem we solve here is a clean version of a problem which will vary with the query language under consideration. For example, the sort order assignments we have chosen to call "optimal" may not under all circumstances in a specific query language be the best ones to use. This work should be seen as a theoretical basis for concrete implementations. In a later section, we discuss possible variations of the problem and modifications of the solutions, and we discuss the technique's place in the large selection of other optimization techniques. The techniques presented here are not intended to necessarily replace existing techniques, but rather supplement them. This is possible, since our proposal does not conflict with standard optimization techniques, though there may be some specialized techniques which cannot be used simultaneously.

In the next section, we discuss sort-merge algorithms for evaluating relational algebra expressions, and we define the sort order problem formally. In section 3, we prove that the problem is NP-hard. In section 4, we develop a novel compact

[b]The corresponding decision problem is NP-complete. In the rest of the paper, whenever we say that a problem is NP-hard, it is also the case that the corresponding decision problem is NP-complete.

representation for permutations of attribute names. Simple relational algebra expressions are treated in section 5. In section 6, we discuss techniques for solving systems of inequalities. This will be used in the algorithms for the computationally harder cases presented in section 7. Finally, in section 8, we discuss the relationship between our techniques and general query optimizers, and discuss possibilities for future work.

2. Sort-Merge Algorithms and Sort Order Problems

We briefly describe one of several standard definitions of relational algebra[3]. More details can be found in the database textbooks[12].

Let $\textsc{Att}$ be a set of elements, here called *attribute names*, and let $\textsc{Dom}$ be a set of *values*. A *relation* r is a pair consisting of a *schema* $\textsc{Sch}(r)$, which is a finite subset of $\textsc{Att}$, and a *finite set of tuples*, which are total functions from $\textsc{Sch}(r)$ to $\textsc{Dom}$. We will often use R to denote the schema of r. A tuple with domain R will often be called a tuple over R.

The set of all *relational algebra expressions* is defined by the following grammar:

$$e ::= r \mid e \cup e \mid e - e \mid e \bowtie e \mid \sigma_{A\, op\, B}(e) \mid \pi_X(e) \mid \delta_{A \leftarrow B}(e)$$

where r can be any relation name, A and B are attribute names, X is a sequence of attribute names, and *op* is one of $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$. In the order listed, the constructors in the grammar above are referred to as relation name, union, difference, natural join, selection, projection, and renaming.

It is standard practice to abuse notation by letting a sequence of attribute names X denote the corresponding set of attribute names $\{X\}$ whenever convenient. We also use r to refer to the component of the pair which is the set of tuples, so we can write $t \in r$, for example, where t is a tuple of r.

A tuple t over R restricted to some set $X \subseteq R$ is denoted $t[X]$, i.e., its domain of definition is limited to $\{X\}$. Renaming an attribute A to B in a tuple t is denoted $t_{A\leftarrow B}$. Formally, $t_{A\leftarrow B}[B] = t[A]$ and $t_{A\leftarrow B}[C] = t[C]$, for all $C \neq B$. The semantics of the relational algebra operators is given in Table 1. Often different attributes are allowed to have different domains (types). However, the results in this paper are independent of whether or not a typed algebra is used, so for presentation purposes, we have chosen the simpler option of only one domain.

Schemas of expressions can be determined statically, so we can extend $\textsc{Sch}$ to expressions. We let E denote the schema of a relation expression e.

The sort-merge technique for evaluating relational algebra expressions is a part of many database implementations. In general, it is one of the best ways to evaluate expressions, when relations are to be kept as sets[c] and one of the most efficient ways of evaluating a join of large relations. Some evaluation methods based on hashing can be more efficient in some cases, but the sort-merge technique also has the advantage of being quite general and simple to implement. Special cases

[c]By definition, a relation in relational algebra *is* a set of tuples, but sometimes duplicate tuples are allowed in query language implementations.

expression	requirement	schema of result	tuples in result
$r_1 \cup r_2$	$R_1 = R_2$	R_1	$\{t \mid t \in r_1 \vee t \in r_2\}$
$r_1 - r_2$	$R_1 = R_2$	R_1	$\{t \mid t \in r_1 \wedge t \notin r_2\}$
$r_1 \bowtie r_2$		$R_1 \cup R_2$	$\{t \mid t[R_1] \in r_1 \wedge t[R_2] \in r_2\}$
$\sigma_{A\, op\, B}(r_1)$	$A, B \in R_1$	R_1	$\{t \mid t \in r_1 \wedge t(A)\, op\, t(B)\}$
$\pi_X(r_1)$	$\{X\} \subseteq R_1$	$\{X\}$	$\{t[X] \mid t \in r_1\}$
$\delta_{A \leftarrow B}(r_1)$	$A \in R_1, B \notin R_1$	$R_1 \backslash \{A\} \cup \{B\}$	$\{t_{A \leftarrow B} \mid t \in r_1\}$

Table 1: Semantics of the relational algebra operators.

of evaluating joins when one relation is small or has an index,[d] are often treated differently (see Merrett[6] and Desai[4] for more details).

The sort-merge idea is quite simple. Let r_1 and r_2 be two relations with $R_1 = R_2 = \{A, B\}$. To compute $r_1 - r_2$, for example, first choose a sort order; either AB or BA, sort[e] both r_1 and r_2 lexicographically with respect to the same sort order, and merge to obtain the result. Notice that the result will automatically be sorted according to this sort order as well.

The merge is a little more complicated when a natural join is to be computed. For example, assume that we wish to compute $r_1 \bowtie r_2$, where $R_1 = \{A, B\}$ and $R_2 = \{B, C\}$. In this case, we sort r_1 on BA and r_2 on BC (common attributes first). Again, the result is obtained by merging. If the B column of the relations has only one occurrence of each integer, then the merge is as simple as before. However, if for example $[A : 0, B : 0]$ and $[A : 1, B : 0]$ are in r_1 and $[B : 0, C : 2]$ and $[B : 0, C : 3]$ in r_2, then the Cartesian product of these tuples has to be output, so a small loop might be required in each merging step. However, the important conclusion is that merging is still an efficient evaluation method for large relations[6], and certain sort orders can be used, whereas others cannot. Note also that due to new developments in external sorting[1], sort-merge techniques have become even more attractive when the domains of attributes are strings.

When considering whole expressions, instead of simply an operator applied to two relations, new possibilities arise. Consider the expression $(r_1 - r_2) \bowtie r_3$, where $R_1 = R_2 = \{A, B\}$ and $R_3 = \{B, C\}$. If $r_1 - r_2$ is evaluated first without any information about the context in which it appears, then we might choose to sort both relations according to the sort order AB and then merge to obtain the partial result. Proceeding to the natural join, we observe that the schema of the difference intersected with R_3 equals $\{B\}$, so for the purpose of merging, r_3 should be sorted according to BC, and the result of the difference according to BA. This means that the result of the difference has to be resorted. Of course, this could have been avoided by initially choosing the sort order BA for the relations r_1 and r_2.

[d] In the context of databases, an index is some efficient way of obtaining random access to tuples in a relation or just some way of efficiently accessing all of the tuples in a fixed order. Often some variant of a B-tree[2] is used with the values from one of the attributes as keys.

[e] More likely, we would create an index (or use an existing one). In the rest of the paper, we will interchangeably say that a relation is sorted according to some sequence of attributes or that it has an index for that sequence.

expression	requirement	output
r		indexes
$r_1 \cup r_2$	$s_1 = s_2$	s_1
$r_1 - r_2$	$s_1 = s_2$	s_1
$r_1 \bowtie r_2$	$s_1 = t \cdot s_1',\ s_2 = t \cdot s_2',$ where $\{t\} = \{s_1\} \cap \{s_2\}$	$s_1 \cdot s_2'$ or $s_2 \cdot s_1'$
$\sigma_{A=B}(r_1)$		s_1
$\pi_X(r_1)$	$s_1 = t \cdot s_1',$ where $\{t\} = \{X\}$	t
$\delta_{A\leftarrow B}(r_1)$		$s_{1A\leftarrow B}$

Table 2: Sort-merge operator requirements and possible outputs.

In Table 2, we list the merge requirements for each of the operators. Given that the arguments of some operator fulfill those criteria, we also list the possible output sortings. Assume that r_1 and r_2 are sorted according to the sort orders s_1 and s_2, respectively. We use a dot to denote the concatenation of two sequences, and for renaming of A to B in a sequence s, we use $s_{A\leftarrow B}$.

The requirement for projection, for example, simply says that the argument must have a sort order which starts with a sequence consisting of the attribute names in X in any order (otherwise sorting is required in order to remove duplicates). Similarly, the sort orders of the arguments of a join must start with the attribute names they have in common.

Before we state the sort order problem formally, we reduce the problem as much as possible:

When considering the requirements from Table 2, we notice that there are no requirements for the selection operator. Since it also preserves the sort order of its operand, it can simply be deleted from the expression before the analysis. This will, of course, result in a quite different expression, but the constraints on sort orders are exactly the same.

From the definition of requirements and output, it is obvious that all occurrences of difference can be replaced by union.

Also, when the two arguments of a join have the same schema, then the natural join requirements become identical to the union requirements. In other words, union is a special case of join with respect to the sort order problem.[f]

In summary, we are down to only having to consider the operators natural join, projection, and renaming. Actually, there exists a transformation of queries such that projections can also be removed, though this is slightly more complicated. However, the transformation introduces a new relation name, which will then appear several places in the query. As it will be demonstrated later, this makes the problem computationally harder, so we avoid this transformation.

The sort order problem can now be stated:

Definition 1 *Let q be a relational algebra expression, and let f be a function from*

[f]This is not very surprising, since intersection is a special case of join, when the schemas of the operands are identical, and surely, intersection would have sort order requirements identical to union and difference.

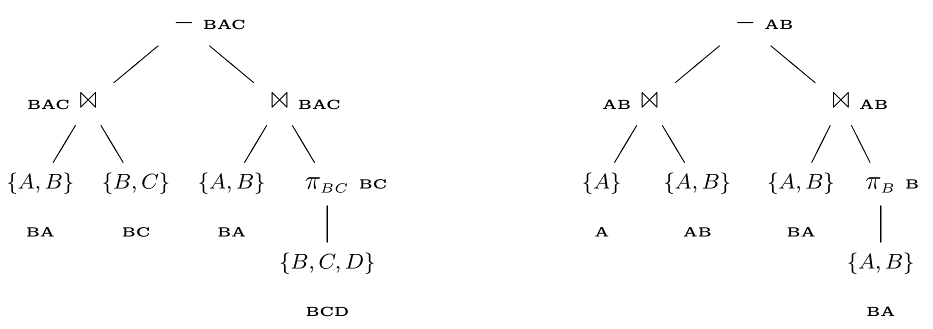

Figure 1: Expressions with annotated sort order assignments.

the set of all subexpressions of q to sequences of attribute names. For each subexpression e, $f(e)$ must be a permutation of the schema of e. Such a function, f, is called a sort order assignment *for e.*

A violation *with respect to f is a subexpression e of q with $e = e' \bowtie e''$, $e = \pi_X(e')$, or $e = \delta_{A \leftarrow B}(e')$, such that either $f(e')$ (or $f(e'')$ in the case of join) do not fulfill the requirements from Table 2, or $f(e)$ is not a possible output according to Table 2. If $e = r$ (a relation name), then it is also a violation if r is not sorted according to $f(e)$ or if r does not have an index for $f(e)$. If there is more than one occurrence of the relation name r, and several of these are assigned the same sort order for which r does not have an index, this is only one violation.*[g]

The sort order problem *is the problem of finding a sort order assignment with a minimal number of violations.*

A sort order assignment with no violations is called perfect.

In Figure 1, we have listed two trees illustrating relational algebra expressions (sometimes called parse trees, query trees, or syntax trees) and sort order assignments. The relations involved are assumed to occur only once, so we have omitted the names of these and just listed their schemas (as sets of attribute names). A sort order assignment has been illustrated by assigning a sequence of attribute names to each node in the parse trees.

Both expressions may have violations at the leaves depending on whether or not the relations are already sorted according to the sort order assigned to that node. Other than that, the first expression does not have any additional violations, whereas the second has one, since the only allowed output from the right-most join, in this case, is BA.

3. NP-Hardness and -Completeness

In this section, we show that the problem of finding a sort order assignment such that the number of violations is minimal is NP-hard. We do this by reducing from vertex cover. The vertex cover problem is described as follows[5]:

[g]Since the sorting of the relation or the creation of one index solves the problem for all occurrences assigned the same sort order.

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Is there a vertex cover of size K or less for G, i.e., a subset $V' \subseteq V$ with $|V'| \leq K$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V'?

Theorem 1 *The sort order problem is NP-hard.*

Proof. Assume that we have a relation r_b with schema $\{B\}$ and a relation r_{ab} with schema $\{A, B\}$.

Furthermore, let the elements of V be numbered from one through p, i.e., V is the set $\{v_1, \ldots, v_p\}$. We take p relations $r_1, \ldots, r_p$, all with schema $\{A, B\}$ and with an index for BA. Assume that for each $i \in \{1, \ldots, p\}$, we have unique attribute names A_i and B_i. We now encode each vertex v_i as follows[h]:

$$c(v_i) = \delta_{AB \leftarrow A_i B_i}((r_{ab} \bowtie r_b) \bowtie r_i)$$

Let the elements of E be numbered from one through m, i.e., $E = \{e_1, \ldots, e_m\}$. Assume that for each $i \in \{1, \ldots, m\}$, we have unique attribute names C_i and D_i.

Each edge $e_i = \{v_j, v_k\}$ is encoded as follows[i]:

$$c(e_i) = \delta_{AB \leftarrow C_i D_i}(r_j \bowtie \delta_{A \leftrightarrow B}(r_k))$$

The whole vertex cover problem is now encoded by the following expression:

$$L = (c(v_1) \bowtie (c(v_2) \bowtie \cdots \bowtie c(v_p) \cdots)) \bowtie (c(e_1) \bowtie (c(e_2) \bowtie \cdots \bowtie c(e_m) \cdots))$$

We claim that a sort order assignment with at most K violations can be found for L if and only if there is a vertex cover of size at most K for G. The proof follows here:

Assume that V' with $|V'| \leq K$ is a vertex cover for G. Consider the expressions $c(v_i)$. Since r_i has a BA index, sort orders can be assigned to each node without creating any violations. The problem is the expressions $r_j \bowtie \delta_{A \leftrightarrow B}(r_k)$. However, either v_j or v_k must be in V'. If v_j is in V', we assign r_j the sort order AB; otherwise, r_k is assigned AB. Since V' is a vertex cover, this creates at most K violations. Finally, L is defined by joining all these $p + m$ expressions. However, there is no overlap of attribute names, so the requirements of these joins are vacuously fulfilled.

For the other implication, assume that there exists a sort order assignment for L with at most K violations. Clearly, an expression $r_j \bowtie \delta_{A \leftrightarrow B}(r_k)$ will contain a violation. In case it is the join requirements which are violated, we can change this to a violation at a relation name by switching the sort orders of r_k and $\delta_{A \leftrightarrow B}(r_k)$. This can only decrease the total number of violations (since assignment of the same sort order to several occurrences of the same relation name counts as at most one violation—depending on whether or not the relation has an index for that sort order). We now define a vertex cover V' of size at most K by: $v_i \in V'$ if and only if somewhere r_i is assigned AB. $\square$

[h]The notation $\delta_{AB \leftarrow CD}(r)$ is the straight-forward generalization of renaming involving several attributes, denoting the simultaneous renaming of A to C and B to D.

[i]The notation $\delta_{A \leftrightarrow B}(r)$ means switching the attribute names A and B. This could equivalently be written $\delta_{AB \leftarrow BA}(r)$.

Notice that no projections were used in the proof. This means that the problem is just as hard if queries contain no projections, or if some other method of evaluating projections is chosen.

The class NP is a class of decision problems (yes/no answers). The sort order decision problem is the following: given a relational algebra expression and a positive integer K, does there exist a sort order assignment f with at most K violations?

Corollary 1 *The sort order decision problem is NP-complete.*

Proof. The proof of Theorem 1 is also a proof of the fact that the sort order decision problem is NP-hard. An NP-hard problem is NP-complete if and only if it is in the class NP, and this is easy to establish for this particular problem: define a sort order assignment by nondeterministically guessing a sequence for each node in the parse tree, and check to see if the number of violations is at most K. $\Box$

4. Representing Sets of Permutations

When considering how to sort a relation over a schema with k attributes, there are $k!$ possibilities. This means that if efficient algorithms are to be found, the representation of sets of permutations is a matter that has to be handled with some care. It turns out that the possible subsets of the set of all permutations of attributes from a given schema, which can be encountered in the algorithms, can be represented by the following grammar:

$$p ::= \textsc{Nil} \mid \langle A_1, \dots, A_k \rangle \mid \mathcal{C}(p, \dots, p) \mid \mathcal{R}(p, \dots, p)$$

where $A_1, \dots, A_k$ is a sequence of attribute names. An expression produced by this grammar, fulfilling that no attribute name appears more than once, is called a *permutation expression.*

The expression $\textsc{Nil}$ represents the empty set of permutations, the expression $\langle A_1, \dots, A_k \rangle$ represents all permutations of the set $\{A_1, \dots, A_k\}$, the expression $\mathcal{C}(p_1, \dots, p_k)$ represents all concatenations of one permutation from each p_i, and the expression $\mathcal{R}(p_1, \dots, p_k)$ represents the union of what $\mathcal{C}(p_1, \dots, p_k)$ and $\mathcal{C}(p_k, \dots, p_1)$ represent ($\mathcal{R}$ stands for *reversed*). For convenience, an expression $\langle A \rangle$ with only one attribute name is simply written A. As an example, $q = \mathcal{R}(\langle A, B \rangle, \mathcal{C}(C, D))$ represents $\{ABCD, BACD, CDAB, CDBA\}$.

More formally, the set of permutations represented by a permutation expression is defined as follows:

Definition 2 *If p is a permutation expression, then its denotation $[\![p]\!]$ is defined recursively by Table 3.*

Some notation: In the following, we use P as short for a comma separated list of permutation expressions, $p_1, \dots, p_{k_p}$. When no confusion can arise, we shall often simply use k instead of k_p. We let P^R stand for the reversed list $p_{k_p}, \dots, p_1$. We use Q, U, V, and T similarly. Finally, we let $\mathcal{A}(P)$ denote the set of attribute names in P, where P is a comma separated list of permutation expressions.

Using compact representations of sets of permutations is crucial for the efficiency of the algorithms to be presented later. To avoid unnecessary redundancy in the

p	$[\![p]\!]$
Nil	$\emptyset$
$\langle A_1,\ldots,A_k\rangle$	*all permutations of* $\{A_1,\ldots,A_k\}$
$\mathcal{C}(p_1,\ldots,p_k)$	$\{t_1\cdots t_k \mid t_1\in[\![p_1]\!],\ldots,t_k\in[\![p_k]\!]\}$
$\mathcal{R}(p_1,\ldots,p_k)$	$[\![\mathcal{C}(p_1,\ldots,p_k)]\!]\cup[\![\mathcal{C}(p_k,\ldots,p_1)]\!]$

Table 3: Semantics of permutation expressions.

expressions, such as in $\mathcal{R}(\mathcal{C}(\langle A,B\rangle))$, for instance, which obviously represents the same set as $\langle A,B\rangle$, we define a *normal form.*

Definition 3 *A permutation expression p is in normal form if and only if:*

- *each $\mathcal{C}$- and $\mathcal{R}$-construct has at least two arguments.*
- *no immediate argument of a $\mathcal{C}$-construct is again a $\mathcal{C}$-construct.*
- *if* Nil *is contained in p, then p =* Nil.

It is easy to put a permutation expression in normal form. Empty constructs, $\langle\rangle$, $\mathcal{C}()$, and $\mathcal{R}()$ can be removed. For $\mathcal{C}$- and $\mathcal{R}$-constructs with only one argument, the $\mathcal{C}$ or $\mathcal{R}$ can simply be removed, i.e., $\mathcal{C}(p)$ and $\mathcal{R}(p)$ can both be changed to p. Furthermore, an expression $\mathcal{C}(p_1,\ldots,\mathcal{C}(q_1,\ldots,q_{k_q}),\ldots,p_{k_p})$ can be replaced by $\mathcal{C}(p_1,\ldots,q_1,\ldots,q_{k_q},\ldots,p_{k_p})$. Clearly, permutation expressions can be put in normal form using a bottom-up strategy in linear time in the size of the expression. From now on, we assume that expressions are always in normal form.

We need three operations on permutation expressions corresponding to the operators we need in the reduced relational algebra expressions: projection, join, and renaming. Renaming is denoted $p\lfloor A\leftarrow B\rfloor$ and simply changes all occurrences of As in p to Bs. Projection is denoted $p|_X$, where X is a set of attribute names. The set of sequences is projected down onto the set X. For example, $\mathcal{R}(\langle A,B\rangle,\mathcal{C}(C,D))|_{CD}=\mathcal{C}(C,D)$ and $\mathcal{R}(\langle A,B\rangle,\mathcal{C}(C,D))|_{AC}=\mathcal{R}(A,C)$.

The join operation on permutation expressions is the most complicated. Its purpose is to compute the possible output permutations from a relational algebra join operation (see section 2) given that the possible inputs are represented by two permutation expressions. The result should be the sequences agreeing on common attributes followed by the rest of each of the expressions in any order. We continue the example above. First, notice that $\langle C,E\rangle$ represents the sequences CE and EC. Joining with $\mathcal{R}(\langle A,B\rangle,\mathcal{C}(C,D))$ should then give us the sequences $CDABE$, $CDBAE$, $CEDAB$, and $CEDBA$. Phrased using permutation expressions:

$$\mathcal{R}(\langle A,B\rangle,\mathcal{C}(C,D))\otimes\langle C,E\rangle=\mathcal{C}(C,\mathcal{R}(E,\mathcal{C}(D,\langle A,B\rangle)))$$

4.1. Implementing operations on permutation expressions

As described above, three operations on permutation expressions will be needed in the algorithms, and it is important that these operations can be implemented efficiently.

It is easy to implement renaming and projection of permutation expressions efficiently, so we will only give an informal description of this.

Renaming a permutation expression by $p\lfloor A \leftarrow B \rfloor$ is simply a matter of going through and copying the expression p, changing the occurrence of A to a B, i.e., the operation is linear in the size of the expression.

A projection $p|_X$ can be carried out by first going through and copying the expression p, except that attribute names not in X are not copied. After this operation, the expression is put in normal form. Since X is a part of p, $|X|$ is smaller than n, where n is the size of p, so this can be done in time $O(n \log n)$.

Continuing the example, in carrying out the operation $\mathcal{R}(\langle A, B\rangle, \mathcal{C}(C, D))|_{CD}$, the first pass results in $\mathcal{R}(\langle , \rangle, \mathcal{C}(C, D))$. Bottom-up, first $\langle , \rangle$ is removed because it is empty, and then the $\mathcal{R}$-construct is removed, since it is down to one argument and therefore redundant. This leaves $\mathcal{C}(C, D)$, which is now in normal form. As another example, the first pass in carrying out the operation $\mathcal{R}(\langle A, B\rangle, \mathcal{C}(C, D))|_{AC}$ results in $\mathcal{R}(\langle A, \rangle, \mathcal{C}(C,))$. Again, the expression is brought in normal form bottom-up resulting in $\mathcal{R}(A, C)$.

The join of two permutation expressions is somewhat harder to implement. However, as we demonstrate below, it can be implemented such that it runs in time $O(n \log n)$, where n is the size of the permutation expression. The remaining part of this section is devoted to establishing this.

We proceed as follows: First, a prefix operations is defined, which from a set of sequences selects the ones that start with a given set of attribute names. Next, we want to define an intersection operator. As a necessary step in obtaining this, we define an algorithm which changes two expressions $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ to two expressions $\mathcal{C}(U)$ and $\mathcal{C}(V)$ with the same intersection, i.e., $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = [\![\mathcal{C}(U)]\!] \cap [\![\mathcal{C}(V)]\!]$, but with the additional property that they have the same number of immediate arguments. Finally, using all these definitions, the join can be defined.

We will take some care in defining all of these operations to make it clear that the definitions can be turned into efficient algorithms.

Definition 4 *A list of permutation expressions $p_1, \ldots, p_k$ is X-*initial *for some nonempty set of attribute names X if*

$$\exists i\colon\ 1 \leq i \leq k,\ \mathcal{A}(p_1, \ldots, p_{i-1}) \subseteq X,\ \mathcal{A}(p_i) \cap X \neq \emptyset,\ \mathcal{A}(p_{i+1} \cdots p_k) \cap X = \emptyset$$

*If a list is X-*initial*, then this unique i is called the* extent *of X.*

Now we can define the prefix operator PF recursively in the structure of permutation expressions.

Definition 5 PF *is defined recursively in Table 4. The first case that applies is chosen. In the table, $X' = X \setminus \mathcal{A}(p_1, \ldots, p_{i-1})$.*

Proposition 1 PF *is well-defined and $[\![\mathrm{PF}_X(p)]\!]$ is exactly the set of sequences from $[\![p]\!]$ which start with the attribute names in X.*

Proof. As there is always the "otherwise" option, PF defines some action for all p. Furthermore, recursive use of PF is always carried out on strictly smaller arguments, in the sense that the *semantic* set that an expression denotes (the function $[\![\cdot]\!]$)

p	Condition	$\mathrm{PF}_X(p)$
	$X = \emptyset$	p
	$\mathcal{A}(p) = X$	p
$\langle Y \rangle$	$X \subset Y$	$\mathcal{C}(\langle X \rangle, \langle Y \backslash X \rangle)$
$\mathcal{C}(P)$	P is X-initial with extent i	$\mathcal{C}(p_1, \ldots, p_{i-1}, \mathrm{PF}_{X'}(p_i), p_{i+1}, \ldots, p_k)$
$\mathcal{R}(P)$	P is X-initial	$\mathrm{PF}_X(\mathcal{C}(P))$
$\mathcal{R}(P)$	P^R is X-initial	$\mathrm{PF}_X(\mathcal{C}(P^R))$
	otherwise	Nil

Table 4: Prefix of permutation expressions.

becomes smaller. Finally, we observe from Definition 4 that if $p_1, \ldots, p_k$ is X-initial, then $p_k, \ldots, p_1$ is not, unless $\mathcal{A}(p_1, \ldots, p_k) = \mathcal{A}(p_k, \ldots, p_1) = X$. We have argued that PF is well-defined.

It is fairly obvious that PF works correctly. The crucial observation is that if P is not X-initial, then $[\![\mathcal{C}(P)]\!]$ does not contain any sequences starting with all the attributes from X. $\square$

Now we turn our attention to the intersection which will be defined using PF and the following property of PF:

Proposition 2 *If $\emptyset \neq X \subset \mathcal{A}(p)$ and $\mathrm{PF}_X(p) \neq$ Nil, then $\mathrm{PF}_X(p)$ is of the form $\mathcal{C}(p_1, \ldots, p_k)$ and $\exists i\colon\ 1 \leq i < k,\ X = \mathcal{A}(p_1, \ldots, p_i)$.*

Proof. Easy proof by induction in the structure of p following the definition of PF. $\square$

Now we define the concept of *comma equalization*:

Definition 6 *Two permutation expressions $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are* comma equalized *if they have the same number of arguments k and $\forall i \in \{1, \ldots, k\}\colon\ \mathcal{A}(p_i) = \mathcal{A}(q_i)$.*

Proposition 3 *Let $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ be permutation expressions and assume that they contain the same attribute names, i.e., $\mathcal{A}(P) = \mathcal{A}(Q)$. Assume that there exists a k such that $k < k_p$, $k < k_q$, and $\forall i \in \{1, \ldots, k\}:\ \mathcal{A}(p_i) = \mathcal{A}(q_i)$. The following holds:*

1) *if $\mathcal{A}(p_{k+1}) \backslash \mathcal{A}(q_{k+1}) \neq \emptyset$ and $\mathcal{A}(q_{k+1}) \backslash \mathcal{A}(p_{k+1}) \neq \emptyset$, then $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = \emptyset$.*

2) *if $\mathcal{A}(p_{k+1}) \subset \mathcal{A}(q_{k+1})$, then either $\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1})$ equals Nil, in which case $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = \emptyset$, or $\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1})$ is of the form $\mathcal{C}(U)$, in which case*

$$[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = [\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(q_1, \ldots, q_k, u_1, \ldots, u_{k_u}, q_{k+2}, \ldots, q_{k_q})]\!]$$

3) *if $\mathcal{A}(q_{k+1}) \subset \mathcal{A}(p_{k+1})$, then the symmetric to 2) holds.*

Proof. Let $s_1 \cdots s_{k_p}$ and $t_1 \cdots t_{k_q}$ be two sequences such that $\forall i\colon\ s_i \in [\![p_i]\!]$ and $\forall i\colon\ t_i \in [\![q_i]\!]$. We prove the three statements separately.

1) For $s_1 \cdots s_{k_p}$ and $t_1 \cdots t_{k_q}$ to be identical, the sequences $s_1 \cdots s_k$ and $t_1 \cdots t_k$ would have to be identical and one of the sequences s_{k+1} and t_{k+1} would have to be a prefix of the other. This is not possible because then one of the sets $\mathcal{A}(p_{k+1})$ and $\mathcal{A}(q_{k+1})$ would be contained in the other.

```
Algorithm: Comma equalize
Input: $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ in normal form with $\mathcal{A}(P) = \mathcal{A}(Q)$
Output: comma equalized $\mathcal{C}(U)$ and $\mathcal{C}(V)$, if they exist, such that
        $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = [\![\mathcal{C}(U)]\!] \cap [\![\mathcal{C}(V)]\!]$.
Method:
  let $exp_p$, $exp_q$, $k$ be $\mathcal{C}(P)$, $\mathcal{C}(Q)$, 0
  while $k < k_q$ do
    if $\mathcal{A}(p_{k+1}) = \mathcal{A}(q_{k+1})$ then
      $k := k + 1$
    else
      if $(\mathcal{A}(p_{k+1}) \backslash \mathcal{A}(q_{k+1}) \neq \emptyset) \wedge (\mathcal{A}(q_{k+1}) \backslash \mathcal{A}(p_{k+1}) \neq \emptyset)$ then
        abort "Empty intersection"
      else
        rename if necessary such that $\mathcal{A}(p_{k+1}) \subset \mathcal{A}(q_{k+1})$
        let $exp$ be $\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1})$
        if $exp$ is NIL then
          abort "Empty intersection"
        else
          comment $exp$ is of the form $\mathcal{C}(T)$, and $exp_q$ is $\mathcal{C}(Q)$
          let $exp_q$ be $\mathcal{C}(q_1, \ldots, q_k, T, q_{k+2}, \ldots, q_{k_q})$
        endif
      endif
    endif
  endwhile
  output $exp_p$, $exp_q$
```

Figure 2: Algorithm "Comma equalize".

2) The form of $\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1})$ was stated in Proposition 2. Here, s_{k+1} has to be a prefix of t_{k+1} in order for $s_1 \cdots s_{k_p}$ and $t_1 \cdots t_{k_q}$ to be identical, so if $\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1}) = \textsc{Nil}$, no sequences from $[\![\mathcal{C}(P)]\!]$ and $[\![\mathcal{C}(Q)]\!]$ can be identical.

   From the above it follows that from the sequences in $[\![q_{k+1}]\!]$, only the sequences in the set $[\![\mathrm{PF}_{\mathcal{A}(p_{k+1})}(q_{k+1})]\!]$ can match sequences in $[\![\mathcal{C}(P)]\!]$. The result follows from Proposition 2.

3) Symmetric to 2).

□

An algorithm based on the cases listed in Proposition 3 is now given in Figure 2.

Proposition 4 *Algorithm* Comma equalize *solves the problem stated in its specification.*

Proof. Since the expressions are in normal form, each q_i must contain at least one attribute name, and for $i \neq j$, we have $\mathcal{A}(q_i) \cap \mathcal{A}(q_j) = \emptyset$. So, even though k_q can increase every time exp_q is changed, $|\mathcal{A}(Q)|$ is an upper bound.

p	q	*Condition*	$p\triangle q$
p	$\langle X\rangle$		p
$\mathcal{C}(P)$	$\mathcal{R}(Q)$		$\mathcal{C}(P)\triangle\mathrm{PF}_{\mathcal{A}(p_1)}(\mathcal{R}(Q))$
$\mathcal{R}(P)$	$\mathcal{R}(Q)$	$\exists T\colon\ \mathcal{C}(P)\triangle\mathcal{R}(Q)=\mathcal{C}(T)$	$\mathcal{R}(T)$
$\mathcal{C}(P)$	$\mathcal{C}(Q)$	$\exists\,\mathcal{C}(U),\mathcal{C}(V)\colon$ *comma equalized*	$\mathcal{C}(u_1\triangle v_1,\ldots,u_{k_u}\triangle v_{k_v})$
		otherwise	NIL

Table 5: Intersection of permutation expressions, leaving out symmetric cases.

The last "else" case cannot be chosen more than $|\mathcal{A}(Q)|$ times, since the number of attribute names available are divided up into more q_i's each time this case is chosen. So, after a bounded number of visits to this "else" case, we must choose another case and either abort or increase k. We have proven that the algorithm terminates.

With respect to correctness, an invariant for the loop is that for all $1\leq i\leq k\colon\ \mathcal{A}(p_i)=\mathcal{A}(q_i)$, where $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are the current values of exp_p and exp_q, respectively. Clearly, when $k=k_q$ the algorithm terminates and we have obtained the desired result. Of course, the algorithm might terminate earlier with an "Empty intersection", if justified according to Proposition 3.

The invariant holds since k is increased only when $\mathcal{A}(p_{k+1})=\mathcal{A}(q_{k+1})$, and in the last "else" case, the k first arguments of $\mathcal{C}(Q)$ are left unchanged. □

Because of the reverse operator, $\mathcal{R}$, on permutation sequences, it will later be necessary to know that comma equalization is symmetric, i.e., we could start at the other end of the two expressions $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ and obtain the same result.

Proposition 5 *If algorithm* Comma equalize *applied to $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ gives $\mathcal{C}(U_1)$ and $\mathcal{C}(U_2)$, and applied to $\mathcal{C}(P^R)$ and $\mathcal{C}(Q^R)$ gives $\mathcal{C}(V_1)$ and $\mathcal{C}(V_2)$, then $U_1={V_1}^R$ and $U_2={V_2}^R$.*

Proof. By structural induction in P, based on the observation that if for some i and j we have that $\mathcal{A}(p_1,\ldots,p_i)=\mathcal{A}(q_1,\ldots,q_j)$, then the comma equalization is found for the expressions $\mathcal{C}(p_1,\ldots,p_i)$ and $\mathcal{C}(q_1,\ldots,q_j)$ and for the expressions $\mathcal{C}(p_{i+1},\ldots,p_{k_p})$ and $\mathcal{C}(q_{j+1},\ldots,q_{k_q})$, independently. □

With the help of comma equalization, we can now define intersection:

Definition 7 *Let p and q be permutation expressions such that $\mathcal{A}(p)=\mathcal{A}(q)$. The intersection of p and q, denoted $p\triangle q$, is defined recursively by Table 5. The operation is symmetric in its two arguments, so we only list one of each of these symmetric cases. In the table, $\mathcal{C}(U)$ and $\mathcal{C}(V)$ are the outputs from algorithm* Comma equalize*, i.e., $[\![\mathcal{C}(P)]\!]\cap[\![\mathcal{C}(Q)]\!]=[\![\mathcal{C}(U)]\!]\cap[\![\mathcal{C}(V)]\!]$.*

Proposition 6 *The intersection of permutation expressions is well-defined and implements $\cap$ correctly, i.e., $\forall p,q\colon\ [\![p\triangle q]\!]=[\![p]\!]\cap[\![q]\!]$.*

Proof. We argue that the process eventually terminates. In the first and last case, we terminate immediately. Let us consider an expression $p\triangle q$ and the value $|[\![p]\!]|+|[\![q]\!]|$. Clearly, $|[\![\mathrm{PF}_{\mathcal{A}(p_1)}(\mathcal{R}(Q))]\!]|<|[\![\mathcal{R}(Q)]\!]|$, since $\emptyset\subset\mathcal{A}(p_1)\subset\mathcal{A}(Q)$, so in the second and third case, this *semantic size* of the two arguments decrease. For the

Case	*Result*
$X = \mathcal{A}(p), X = \mathcal{A}(q)$	$p \triangle q$
$X = \mathcal{A}(p), X \subset \mathcal{A}(q)$	$\mathcal{C}(p \triangle \mathcal{C}(q_1, .., q_j), \mathcal{C}(q_{j+1}, .., q_{k_q}))$
$X \subset \mathcal{A}(p), X = \mathcal{A}(q)$	$\mathcal{C}(\mathcal{C}(p_1, .., p_i) \triangle q, \mathcal{C}(p_{i+1}, .., p_{k_p}))$
$X \subset \mathcal{A}(p), X \subset \mathcal{A}(q)$	$\mathcal{C}(\mathcal{C}(p_1, .., p_i) \triangle \mathcal{C}(q_1, .., q_j), \mathcal{R}(\mathcal{C}(p_{i+1}, .., p_{k_p}), \mathcal{C}(q_{j+1}, .., q_{k_q})))$

Table 6: Join of permutation expressions.

fourth case, $\mathcal{C}(P) \triangle \mathcal{C}(Q)$, this value may remain constant, but then the number of attribute names in the expressions decrease. We have argued that $\triangle$ is well-defined.

Now, we prove that $\triangle$ implements $\cap$ correctly. It is obvious that $p \triangle \langle X \rangle$ should equal p, since $\langle X \rangle$ represents all possible permutations of $\mathcal{A}(p)$ (recall that $\mathcal{A}(p) = \{X\}$).

The proof of correctness is by induction in the number of iterations. As already discussed, we can use induction in the lexicographical order of first the semantic size of the arguments, and second, the size of the set of attribute names in the expressions. It is obvious in the light of Proposition 4 that the fourth case gives rise to correct transformations.

The correctness of $\mathcal{C}(P) \triangle \mathcal{C}(Q) = \mathcal{C}(P) \triangle \mathrm{PF}_{\mathcal{A}(p_1)}(\mathcal{R}(Q))$ is obvious since all sequences from $[\![\mathcal{C}(P)]\!]$ must start with a sequence from $[\![p_1]\!]$.

For $\mathcal{R}(P) \triangle \mathcal{R}(Q)$, observe that semantically this is

$$([\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!]) \cup ([\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q^R)]\!]) \cup ([\![\mathcal{C}(P^R)]\!] \cap [\![\mathcal{C}(Q)]\!]) \cup ([\![\mathcal{C}(P^R)]\!] \cap [\![\mathcal{C}(Q^R)]\!])$$

To justify the transformation, we need to observe that

- if $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] \neq \emptyset$, then $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q^R)]\!] = \emptyset$, and vice versa.
- if $\mathcal{C}(P) \triangle \mathcal{C}(Q) = \mathcal{C}(T)$, then $\mathcal{C}(P^R) \triangle \mathcal{C}(Q^R) = \mathcal{C}(T^R)$.

The first observation follows easily from the definitions, and the second from Proposition 5 and the definition of $\triangle$ on arguments of the form $\mathcal{C}(P)$ and $\mathcal{C}(Q)$. $\square$

Finally, the most complicated operation on permutation expressions can be defined from prefix and intersection.

Definition 8 *Let p and q be permutation expressions and let $X = \mathcal{A}(p) \cap \mathcal{A}(q)$. If $X \subset \mathcal{A}(p)$, then $\mathrm{PF}_X(p) = \mathcal{C}(P)$, for some P, and there exists an i such that $X = \mathcal{A}(p_1, \ldots, p_i)$. If $X \subset \mathcal{A}(q)$, then $\mathrm{PF}_X(q) = \mathcal{C}(Q)$, for some Q, and there exists a j such that $X = \mathcal{A}(q_1, \ldots, q_j)$. The join of the permutation expressions p and q, denoted $p \otimes q$, is defined recursively in Table 6.*

Proposition 7 *The operation $\otimes$ correctly computes the join of permutation expressions.*

Proof. We only treat the fourth and most difficult case. The remaining proofs are special cases of the proof to follow.

If there exists a $t \in [\![\langle X \rangle]\!]$ such that we have $s_p \in [\![p]\!]$ and $s_q \in [\![q]\!]$ with $s_p = t \cdot s'_p$ and $s_q = t \cdot s'_q$ for some s'_p and s'_q, then $t \cdot s'_p \cdot s'_q$ and $t \cdot s'_q \cdot s'_p$ should belong to

$[\![p \otimes q]\!]$. By Proposition 1, $[\![\mathrm{PF}_X(p)]\!]$ and $[\![\mathrm{PF}_X(q)]\!]$ are exactly the subsets of $[\![p]\!]$ and $[\![q]\!]$, respectively, for which such t's exist. Finally, from Proposition 6, it follows that $[\![\mathcal{C}(p_1, \ldots, p_i) \triangle \mathcal{C}(q_1, \ldots, q_j)]\!]$ is exactly the set of sequences which are prefixes of sequences in both sets. □

The join of two permutation expressions p and q can be performed in time $O(n \log n)$, where n is the size of the input, i.e., the number of symbols in the expressions p and q together. It is clear that all the involved operations (prefix, comma equalization, and intersection) on permutation expressions are basically linear in the structure of the arguments, except for some tests of set inclusion. However, initially sorting all the attributes in subexpressions of the form $\langle X \rangle$ will make these tests linear as well. The time it takes to perform this initial sorting is bounded by $O(n \log n)$.

5. Solving the Problem for Simple Queries

In section 2, we have showed that it is sufficient to solve the problem for expressions built from relation names combined with joins, projections, and renamings. However, in section 3, we proved that even this was NP-hard. That result was established even without considering schema sizes; only the size of the relational algebra expression was used as the measure of the problem size.

The important issue, to be considered next, is to identify the subproblems, which can be solved efficiently, and for this more practical problem, schema sizes become relevant. Though, it is relatively unusual that a relation has a schema of size more than 10, say, if it has even that modest a degree, there are more than 10! permutations to consider. So, in practice, it is crucial that the representations of sets of permutations can be reasonably compact. If any subset of the set of all permutations of a schema could be relevant in the algorithms, then we would not be able to find a compact representation. Fortunately, permutation expressions from section 4 can be used. This means that any subset of the set of permutations of a set of size k that we consider in the algorithms can be represented in space $O(k)$, and operations on it can be carried out in time $O(k \log k)$, as demonstrated in the previous section.

The large class of expressions without multiple occurrences of the same relation name has an efficient algorithm to determine optimal sort order assignments. In the following we refer to such expressions or queries as *simple*. This algorithm has the same flavor as the one by Smith and Chang[10], the significant difference being that while they consider only a few possible sequences, we consider all $k!$, where k is the schema size. Also with multiple occurrences of the same relation names, it is sometimes possible to find sort order assignments efficiently, but the algorithms become significantly more complicated. We return to those issues in section 7.

For the class of simple expressions, i.e., expressions without multiple occurrences of the same relation name, a greedy algorithm is optimal. There are two stages in the algorithm. In the first, we find possible solutions; in the second, we select a sort order assignment from these. It is easiest to express this as one pass up and one pass down the parse tree. The algorithm is given in Figure 3.

Algorithm: Find sort order assignment for simple expressions.
Input: a simple relational algebra expression q.
Output: a sort order assignment for q with a minimal number of violations.
Method:

```
  **forall** relation names $r$ **let** $S_r$ **be** the permutation expression for $r$'s indexes
  **repeat**
    assume that $e_1$ and $e_2$ have been assigned $S_{e_1}$ and $S_{e_2}$, respectively
    **case** $e$ is of the form
      $e_1 \bowtie e_2$: $S_e = S_{e_1} \otimes S_{e_2}$
      $\pi_X(e_1)$: $S_e = S_{e_1}|_X$
      $\delta_{A \leftarrow B}(e_1)$: $S_e = S_{e_1} \lfloor A \leftarrow B \rfloor$
    **endcase**
    **if** $S_e = \text{NIL}$ **then** $S_e = \langle E \rangle$; mark $e$ **endif** (* $E$ is the schema of $e$ *)
  **until** all nodes have an assigned permutation expression

  choose a sort order from $S_q$ and assign it to $q$
  **repeat**
    assume that $e$ has been assigned the sort order $s_e$
    **case** $e$ is of the form
      $e_1 \bowtie e_2$:
        **if** $e$ is not marked **then**
          $s_{e_1}$ ($s_{e_2}$) is $s_e$ with every attribute name not in $E_1$ ($E_2$) deleted
        **else**
          choose $s_{e_1}$ ($s_{e_2}$) from $S_{e_1}$ ($S_{e_2}$)
        **endif**
      $\pi_X(e_1)$:
        **if** $e$ is not marked **then**
          choose $s_{e_1}$ from $S_{e_1}$ such that $s_{e_1}$ has prefix $s_e$
        **else**
          choose $s_{e_1}$ from $S_{e_1}$
        **endif**
      $\delta_{A \leftarrow B}(e_1)$:
        $s_{e_1}$ is $s_e$ with $B$ changed to $A$
    **endcase**
  **until** all node have an assigned sort order
```

Figure 3: Algorithm "Find sort order assignment for simple expressions".

A few notes on the algorithm: Choosing a sort order from a permutation expression can clearly be done in linear time. In section 8, we discuss the possibilities when choosing between several sort orders is an option. A full discussion of how to create a permutation expression for the indexes of a base relation is delayed until section 8. For the simple cases, if a base relation is unsorted and does not have any indexes, we can use $\langle E \rangle$, so that any permutation can be chosen. For consistency, we would also mark the node, since a mark at a node represents the fact that a sorting or the creation of an index is required. If the base relation is sorted or has one index, the permutation expression is of the form $\mathcal{C}(A_1, \ldots, A_k)$.

Theorem 2 *Algorithm "Find sort order assignment for simple expressions" works correctly according to its specification, and its time complexity is $O(k \log(k) n)$, where n is the size of the query and k is the maximum degree of any relation in the query.*

Proof. As long as non-Nil permutation expressions can be found, this is done. Not until a permutation expression becomes Nil (representing the empty set), do we take other actions. When this happens, the node in question is marked (meaning that an index or a sorting will be necessary here), and we consider all possible permutations for this. The only other option, we would have, would be to make the decision of marking a node earlier. However, this can only result in fewer choices at the node actually marked in the algorithm.

For the time complexity, if the expression has size $O(n)$, then $O(n)$ operations are carried out. Since any operation on permutation expressions is at most $O(k \log k)$, the total complexity is then $O(k \log(k) n)$, which for all practical purposes is $O(n)$. $\square$

To show how the algorithm works, we continue the second example from Figure 1. A word of caution: do not fall for the temptation to think that since join is associative and commutative, the expression can be rearranged more efficiently. Recall that a number of reductions have taken place as outlined in section 2. So, some of the joins are really unions or differences, say. In the analysis, we can treat them as joins, but the structure of the query must be preserved. Also, selections appearing in between have been deleted.

In Figure 4, the parse tree to the left has been annotated with permutation expressions, and the parse tree to the right has been annotated with sort orders chosen from the permutation expressions.

The root was first assigned Nil and then the set of all permutations. This was remembered for the second pass by marking it. If the expression was a part of a larger expression, this marked node would be treated as a leaf in the larger expression.

6. Solving Systems of Inequalities

In this section, we briefly discuss how to find maximal solutions to systems of inequalities. This is necessary in order to solve the constraints that multiple occurrences of relation names can impose on our problem. Basically, we adapt Tarski's work on fixed points for functions defined on complete lattices to our concrete problem. Parts of this exposition is from Schwartzbach[8,9].

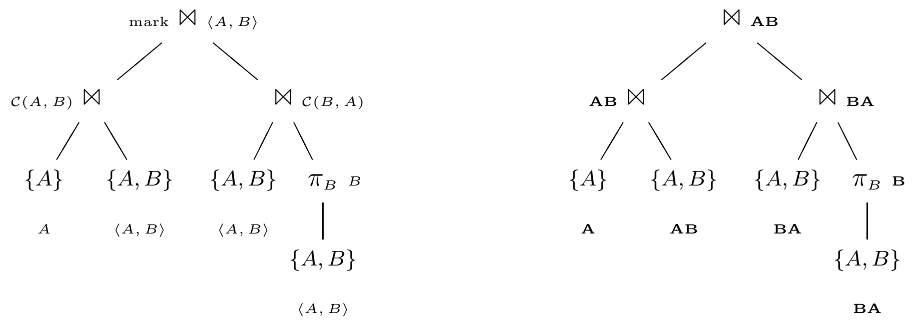

Figure 4: An expression with annotated permutation expressions and sort orders.

The reason for including this material here is two-fold. First, it makes the paper self-contained and it introduces the notation. Second, stronger versions of some of the standard theorems are necessary in order to obtain efficient algorithms. We need to be able to change the system of inequalities gradually without having to compute fixed points from scratch every time; instead, we need to be able to continue the fixed point derivation from the old fixed point. The ability to do this is absolutely crucial for the complexity of the algorithm.

The connection between the results in this section can be described as follows. Our goal is to find the largest solution to an inequational system. This is done by finding the fixed point for some particular function defined from an equational system similar to the inequational system. We then show that the largest solutions to the equational and inequational systems correspond. Only the corollary of Theorem 3 is used in this section, but Theorem 3 will be necessary later.

This first definition is adapted from Tarski[11]:

Definition 9 *A system $(U, \sqsubseteq)$ is a complete lattice if U is a nonempty finite set, $\sqsubseteq$ is a partial order on U, and any subset $A \subseteq U$ has a least upper bound $\sqcup A$ and a greatest lower bound $\sqcap A$.*

A complete lattice has, in particular, two elements, top and bottom, defined by $\top = \sqcup U$ and $\bot = \sqcap U$, respectively.

Theorem 3 *Let $(U, \sqsubseteq)$ be a complete lattice and $f\colon U \to U$ a monotone function. Let $v \in U$ and assume that $f(v) \sqsubseteq v$. Then the largest fixed point for f less than or equal to v can be found as $f^k(v)$, for some $k \in \mathbb{N}$.*

Proof. Let $u = \sqcap\{f^i(v) \mid i \in \mathbb{N}\}$. From $f(v) \sqsubseteq v$ we can prove by simple induction that for all i, $f^{i+1}(v) \sqsubseteq f^i(v)$, using the monotonicity of f. So, we have for all i that $\sqcap\{v, f(v), \ldots, f^i(v)\} = f^i(v)$. Since U is finite, there exists a k such that $u = f^k(v)$.

First, we prove that u is indeed a fixed point. By monotonicity of f, we obtain that

$$f(u) = f(f^k(v)) \sqsubseteq f^k(v) = u$$

and since u is a lower bound that

$$u \sqsubseteq f^{k+1}(v) = f(f^k(v)) = f(u)$$

from which it follows that $f(u) = u$.

Now assume that u' is a fixed point less than or equal to v. We obtain that $u' = f(u') \sqsubseteq f(v)$, since f is monotonic. By induction, we see that $u' \sqsubseteq f^k(v)$. This means that $u' \sqsubseteq u$, so u is the largest fixed point less than or equal to v. □

Corollary 2 *If $(U, \sqsubseteq)$ is a complete lattice with top element $\top = \sqcup U$ and $f\colon U \to U$ is a monotone function, then the largest fixed point can be found as $f^k(\top)$, for some $k \in I\!N$.*

Proof. $f(\top) \sqsubseteq \top$. Of course, the largest fixed point is less than or equal to $\top$, so the result follows. □

Definition 10 *Let $(U, \sqsubseteq)$ be a complete lattice. We define an* inequality system *on $(U, \sqsubseteq)$ to be a finite set of inequalities of the form*

$$M_0 \sqsubseteq f(M_1, \ldots, M_k)$$

where the M_i's are variables and $f\colon U^k \to U$ is monotonic. A solution *$\mathcal{L}$ assigns to each variable M some value $\mathcal{L}(M) \in U$ such that all the inequalities hold. The system is* satisfiable *if a solution exists.*

Definition 11 *Let $\mathcal{S}$ be an inequality system. We define* $\mathrm{EQ}(\mathcal{S})$ *to be the set of equalities, where, for each variable M in the system $\mathcal{S}$, we include*

$$M = H_1 \sqcap H_2 \sqcap \ldots \sqcap H_k$$

The H_i's are all the right-hand sides of inequalities in $\mathcal{S}$ with M on the left-hand side. A solution *$\mathcal{L}$ assigns to each variable M some value $\mathcal{L}(M) \in U$ such that all the equalities hold. The system is* satisfiable *if a solution exists.*

Definition 12 *Assume that F is a set of monotone functions, each of which are from $U^h \to U$, for some h (monotone in each argument). If for $i = 1, \ldots, n$ we have equations*

$$M_i = f_{j_i}(M_{i_1}, \ldots, M_{i_k})$$

where $f_{j_i} \in F$, we can choose to consider the f_j's as functions of all the variables, i.e.,

$$M_i = f_{j_i}(M_1, \ldots, M_n)$$

and then define an iteration *function by*

$$(x_1, \ldots, x_n) \mapsto (f_{j_1}(x_1, \ldots, x_n), \ldots, f_{j_n}(x_1, \ldots, x_n))$$

Given an equality system, we call this the corresponding iteration function.

Proposition 8 *The function $\mathcal{L}$ is a solution to an equality system if and only if it is a fixed point for the corresponding iteration function.*

Proof. Easy observation. □

Proposition 9 *The iteration function defined above is monotone on U^n.*

Proof. Trivial. □

The following observation is from Tarski[11].

Proposition 10 *If $n \geq 2$ is an integer and $(U, \sqsubseteq)$ is a complete lattice, then $(U^n, \sqsubseteq^n)$ is also a complete lattice.*

Proof. Easy. □

Lemma 1 *If $\mathcal{S}$ is an inequality system, then* $\text{EQ}(\mathcal{S})$ *is satisfiable and has a unique largest solution.*

Proof. By Proposition 9, the corresponding iteration function from Definition 12 is monotonic. Since, by Proposition 10, $(U^n, \sqsubseteq^n)$ is a complete lattice, Corollary 2 applies, and the iteration function has a fixed point, which by Proposition 8 gives us the result. □

By Lemma 1, it is now well-defined to talk about the *largest solution* to $\text{EQ}(\mathcal{S})$.

Lemma 2 *Let $\mathcal{S}$ be an inequality system. Then the largest solution to* $\text{EQ}(\mathcal{S})$ *is also a solution to $\mathcal{S}$; and it is the largest solution to $\mathcal{S}$.*

Proof. First note that $\mathcal{S}$ has at least one solution, namely the trivial solution, where all variables are set to the bottom element of the lattice. Let $\mathcal{L}$ be a solution to $\mathcal{S}$. Assume that for some M_q, $\mathcal{L}(M_q) \sqsubset \sqcap_i \mathcal{L}(H_i)$, where $\mathcal{L}(H_i)$ means $f(\mathcal{L}(M'_1), \ldots, \mathcal{L}(M'_p))$, if $H_i = f(M'_1, \ldots, M'_p)$. Define $\mathcal{L}'$ by

$$\mathcal{L}'(M) = \begin{cases} \mathcal{L}(M), & \text{if } M \neq M_q \\ \sqcap_i \mathcal{L}(H_i), & \text{if } M = M_q \end{cases}$$

Clearly $\mathcal{L}'$ is larger than $\mathcal{L}$. It is also a solution (to $\mathcal{S}$) as for all $H_{i'}$,

$$\begin{array}{rcl} \mathcal{L}'(M_q) & = & \sqcap_i \mathcal{L}(H_i), \text{ by assumption} \\ & \sqsubseteq & \sqcap_i \mathcal{L}'(H_i), \text{ as } \mathcal{L}' \text{ is larger than } \mathcal{L} \text{ and } \sqcap \text{ is monotonic} \\ & \sqsubseteq & \mathcal{L}'(H_{i'}), \text{ property of } \sqcap \end{array}$$

and if $M_j \neq M_q$, then for all $H_{i'}$,

$$\begin{array}{rcl} \mathcal{L}'(M_j) & = & \mathcal{L}(M_j), \text{ by definition} \\ & \sqsubseteq & \sqcap_i \mathcal{L}(H_i), \text{ as } \mathcal{L} \text{ is a solution} \\ & \sqsubseteq & \sqcap_i \mathcal{L}'(H_i), \text{ as } \mathcal{L}' \text{ is larger than } \mathcal{L} \text{ and } \sqcap \text{ is monotonic} \\ & \sqsubseteq & \mathcal{L}'(H_{i'}), \text{ property of } \sqcap \end{array}$$

By repetition of the above, the largest solution to $\mathcal{S}$ must have $\mathcal{L}(M_j) = \sqcap_i \mathcal{L}(H_i)$ for all M_j's and corresponding right-hand sides. Thus, the largest solution to $\mathcal{S}$ is to be found among the solutions to $\text{EQ}(\mathcal{S})$. The result now follows from the trivial observation that any solution to $\text{EQ}(\mathcal{S})$ is also a solution to $\mathcal{S}$. □

7. Solving the General Problem

Solving the problem for relational algebra expressions with multiple occurrences of the same relation name is more difficult. However, also some of these problem instances have efficient solutions, though a corner of NP-hard problems remains out of reach.

First, we consider the case where there is a perfect solution, i.e., a solution where no resorting is necessary. It turns out that if such a solution exists, then it can be found in polynomial time. The algorithm for this can also be used as the basis for

$\bowtie$	$\downarrow$	$M_1 \subseteq M\lvert_{E_1}$
		$M_2 \subseteq M\lvert_{E_2}$
	$\uparrow$	$M \subseteq M_1 \otimes M_2$
π_X	$\downarrow$	$M_1 \subseteq \mathcal{C}(M, \langle E_1 \backslash X \rangle)$
	$\uparrow$	$M \subseteq M_1\lvert_X$
$\delta_{A \leftarrow B}$	$\downarrow$	$M_1 \subseteq M\lfloor B \leftarrow A \rfloor$
	$\uparrow$	$M \subseteq M_1\lfloor A \leftarrow B \rfloor$

Table 7: Generating inequalities.

an algorithm which performs exhaustive search for a minimal number of places to resort.

7.1. When a perfect solution exists

The real problem in the complex case of dealing with multiple occurrences is circularity. It is not possible in a simple bottom-up manner to define the possible sequences, since choices made for a subexpression containing one occurrence of a certain relation name r will affect the possible choices near other occurrences.

The first step is to set up a number of constraints ruling out sort order sequences which can definitely not be used:

Definition 13 *Let e be a relational algebra expression with operators join, projection, and renaming. We define an inequational system* $\textsc{InEq}(e)$ *over the variables $M_1, M_2, \ldots, M_n$, where n is the number of subexpressions of e. One variable is associated with each subexpression.*

For a given subexpression with associated variable M, and with variables M_1 (and for $\bowtie$ also M_2) associated with the immediate arguments of the top operator of the subexpression, Table 7 lists the inequalities generated by that subexpression.[j] *Additionally, if two subexpressions e_1 and e_2 are identical relation names, we include $M_1 \subseteq M_2$ and $M_2 \subseteq M_1$.*

The idea is that if, for example, we know that only sequences in M can be used for $e_1 \bowtie e_2$, then no sequences outside $M\lvert_{E_1}$ can be used for e_1. This is expressed by the inclusion $M_1 \subseteq M\lvert_{E_1}$.

Of course, the last two inequalities in the definition simply express that $M_1 = M_2$. This is in order to capture that sorting or creating an index for one occurrence of a relation name also benefits the other occurrences of that same relation name.

We now give the algorithm (Figure 5). The starting point is the set of inequalities that we discussed above, and we find the largest solution to these, i.e., the solution that rules out fewest sort order assignments. Then we gradually make this solution smaller until all variables are assigned singleton sets. If that can be done, then that solution describes a perfect sort order assignment. The difficulty lies in gradually refining the solution. Great care must be taken to assure that not all

[j] The symbols $\downarrow$ and $\uparrow$ are only included to improve readability. They refer to the parse tree, and indicate whether the sequences in M limits the ones that can be in M_1 (and possible M_2), or the other way around.

Algorithm: Find sort order assignment
Input: an expression e
Output: a sort order assignment if one exists
Method:

```
  let $\mathcal{S}$ be the system $\textsc{InEq}(e)$
  let $\mathcal{L}$ be the largest solution to $\textsc{Eq}(\mathcal{S})$
  while $(\forall M_i\colon\ |[\![\mathcal{L}(M_i)]\!]| \geq 1) \wedge (\exists M_i\colon\ |[\![\mathcal{L}(M_i)]\!]| > 1)$ do
    choose $M_i$ such that $|[\![\mathcal{L}(M_i)]\!]| > 1$
    comment $\mathcal{L}(M_i)$ contains an $\langle X \rangle$ or an $\mathcal{R}$-construct
    if $\mathcal{L}(M_i)$ contains an $\langle X \rangle$ then
      let $p$ be $\mathcal{L}(M_i)$ with $\langle X \rangle$ replaced by any other
        permutation expression over the set $X$
    else
      choose an inner-most $\mathcal{R}(P)$ in $\mathcal{L}(M_i)$
      let $p$ be $\mathcal{L}(M_i)$ with $\mathcal{R}(P)$ replaced by $\mathcal{C}(P)$ or $\mathcal{C}(P^R)$
    endif
    let $\mathcal{S}$ be $\mathcal{S}$ with the addition of the inequality $M_i \subseteq p$
    let $\mathcal{L}$ be the largest solution to $\textsc{Eq}(\mathcal{S})$
  endwhile
  if $\exists M_i\colon\ |[\![\mathcal{L}(M_i)]\!]| = 0$ then
    output "No solution exists."
  else
    comment We now have $\forall M_i\colon\ |[\![\mathcal{L}(M_i)]\!]| = 1$
    let $f(i) = s$ if and only if $[\![\mathcal{L}(M_i)]\!] = \{s\}$
    output "$f$"
  endif
```

Figure 5: Algorithm "Find sort order assignment".

perfect sort orders are suddenly removed. Fortunately, there are parts of permutation expressions which can be identified syntactically and which can be modified in such a way that the set of candidates for a perfect sort order assignment becomes smaller, but also such that if a perfect sort order exists, there will, in each step, be at least one perfect sort order which is not ruled out.

We now give a few examples illustrating how the algorithm works. Consider the query $(\pi_A(r_1) \times r_2) - r_1$, where $R_1 = \{A, B\}$ and $R_2 = \{B\}$, and recall that difference and Cartesian product are just treated as joins by the algorithm. First we fix an enumeration of the subexpressions involved:

1.	r_1 (left-most)	2.	$\pi_A(r_1)$	3.	r_2
4.	$\pi_A(r_1) \times r_2$	5.	r_1 (right-most)	6.	$(\pi_A(r_1) \times r_2) - r_1$

Then we generate inequalities based on Table 7. The projection gives rise to the following inequalities:

$$M_1 \subseteq \mathcal{C}(M_2, B) \qquad\qquad M_2 \subseteq M_1|_A$$

From the Cartesian product (the join rules), we get the following:

$$M_2 \subseteq M_4|_A \qquad M_3 \subseteq M_4|_B \qquad M_4 \subseteq M_2 \otimes M_3$$

From the difference (the join rules), we get the following:

$$M_4 \subseteq M_6|_{AB} \qquad M_5 \subseteq M_6|_{AB} \qquad M_6 \subseteq M_4 \otimes M_5$$

These can actually be simplified. Since M_6 is over $\{A, B\}$, the projection down to that domain is unnecessary, and the join of permutation expressions is simply an intersection in this case. Finally, since r_1 appears twice, we get the following:

$$M_1 \subseteq M_5 \qquad\qquad M_5 \subseteq M_1$$

We assume that neither r_1 nor r_2 have indixes, so there are no further inequalities.

Applying the simplifications described above and combining right-hand sides, we get the following system:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
M_1 & \subseteq & \mathcal{C}(M_2, B) \triangle M_5 & M_4 & \subseteq & M_2 \otimes M_3 \triangle M_6 \\
M_2 & \subseteq & M_1|_A \triangle M_4|_A & M_5 & \subseteq & M_1 \triangle M_6 \\
M_3 & \subseteq & M_4|_B & M_6 & \subseteq & M_4 \triangle M_5
\end{array}$$

We solve this system by iterating the corresponding iteration function from the top element of the product lattice until we reach a fixed point:

	M_1	M_2	M_3	M_4	M_5	M_6
top	$\langle A, B\rangle$	A	B	$\langle A, B\rangle$	$\langle A, B\rangle$	$\langle A, B\rangle$
1. iteration	$\mathcal{C}(A, B)$	A	B	$\langle A, B\rangle$	$\langle A, B\rangle$	$\langle A, B\rangle$
2. iteration	$\mathcal{C}(A, B)$	A	B	$\langle A, B\rangle$	$\mathcal{C}(A, B)$	$\langle A, B\rangle$
3. iteration	$\mathcal{C}(A, B)$	A	B	$\langle A, B\rangle$	$\mathcal{C}(A, B)$	$\mathcal{C}(A, B)$
4. iteration	$\mathcal{C}(A, B)$	A	B	$\mathcal{C}(A, B)$	$\mathcal{C}(A, B)$	$\mathcal{C}(A, B)$

This simple example illustrates how information moves around in the system. In this case, we found a solution immediately, but often, the first fixed point will not immediately be a solution. For instance, if we consider the very small expression $r_1 \bowtie r_2$, where $R_1 = \{A, B, C\}$ and $R_2 = \{A, D\}$, the first fixed point will assign $\mathcal{C}(A, \mathcal{R}(\langle B, C\rangle, D))$ to the top node, so as it is done in the algorithm, the system has to be restricted further and another iteration is necessary.

That it is necessary to treat renaming can be seen from examples such as $\pi_C(\delta_{A\leftarrow C}(r_1)) \times r_1$ with $R_1 = \{A, B\}$, where only the sort order AB for r_1 is reasonable.

The rest of the section is devoted to the proof of correctness of the manipulations just described informally, and to the proof of complexity.

We assume that all the nodes in the parse tree of the expression e to be analyzed by the algorithm are numbered from 1 through n. In the inequational system, we use variables M_1 through M_n, and we assume that variable M_i is associated with node i. A sort order assignment is thus a function from $\{1, \ldots, n\}$ to sets of sequences of attribute names.

Definition 14 *Let e be a relational algebra expression. A sort order assignment f is* contained in *a solution $\mathcal{L}$ if and only if $\forall i \in \{1, \ldots, n\}\colon\ f(i) \in [\![\mathcal{L}(M_i)]\!]$.*

In the algorithm, we repeatedly add inequalities to the initial inequality system. In the following, we prove that only perfect sort orders which are explicitly ruled out by these inequalities will actually disappear from the maximal solution.

Lemma 3 *Let $\mathcal{S}$ be the system* $\textsc{InEq}(e)$ *with the addition of the following h inequalities:*

$$M_{q_1} \subseteq S_1,\ M_{q_2} \subseteq S_2,\ \ldots,\ M_{q_p} \subseteq S_h$$

where the S_j's are permutation expressions and the M_{q_j}'s are variables (not necessarily distinct) from $\textsc{InEq}(e)$. *Let $\mathcal{L}_\mathcal{S}$ be the largest solution to* $\textsc{Eq}(\mathcal{S})$. *Then for all perfect sort order assignments f, the following holds:*

$$(\forall j \in \{1, \ldots, h\}\colon\ f(q_j) \in [\![S_j]\!]) \ \Rightarrow\ f \textit{ is contained in } \mathcal{L}_\mathcal{S}$$

Proof. Define $\mathcal{L}(M_i) = \{\mathcal{C}(f(i))\}$ for all $i \in \{1, \ldots, n\}$. Then $\mathcal{L}$ is a solution to $\mathcal{S}$. This is an easy observation: compare Definition 13 with the properties of a sort order assignment as defined in Definition 1.

From Lemma 2, we know that $\mathcal{L}(M_i) \subseteq \mathcal{L}_\mathcal{S}(M_i)$. Since $\mathcal{L}(M_i) = \{f(i)\}$, it follows that $f(i) \in \mathcal{L}_\mathcal{S}(M_i)$. □

Corollary 3 *Let f be a perfect sort order assignment for e and let $\mathcal{L}$ be the largest solution to* $\textsc{Eq}(\textsc{InEq}(e))$. *Then $\forall i\colon\ f(i) \in \mathcal{L}(M_i)$.*

Proof. From $h = 0$, i.e., no extra inequalities, so $\mathcal{S}$ is simply $\textsc{InEq}(e)$. □

It becomes important to be able to talk about two attribute names in different subexpressions being semantically identical. However, when there are renamings in an expression or multiple occurrences of the same relation name, it is not clear which attribute names carry the same meaning, so we give a definition:

Definition 15 *Let e be a relational algebra expression. First, we define the relation* immediately visible. *Assume that the nodes i and j, $i \neq j$, correspond to the subexpressions e_i and e_j, respectively.*

If e_i and e_j are two identical relation names and the attribute $A \in E_i$ $(= E_j)$, then

A at i is immediately visible from A at j, and
A at j is immediately visible from A at i.

If $e_i = \pi_X(e_j)$ and $A \in X$, then

A at i is immediately visible from A at j, and
A at j is immediately visible from A at i.
If $e_i = e_j \bowtie e_h$ or $e_i = e_h \bowtie e_j$ for some e_h and $A \in E_j$, then
A at i is immediately visible from A at j, and
A at j is immediately visible from A at i.
If $e_i = \delta_{A \leftarrow B}(e_j)$, then
B at i is immediately visible from A at j, and
A at j is immediately visible from B at i,
and if $C \in E_j \backslash \{A\}$, then
C at i is immediately visible from C at j, and
C at j is immediately visible from C at i.
The relation visible *is now defined as the reflexive and transitive closure of the relation immediately visible, i.e.,*
A at i is visible from A at i, and

if C at k is visible from A at i
and B at j is immediately visible from C at k
then B at j is visible from A at i.
Obviously, attribute names are visible through paths in the syntax tree. We shall refer to these paths as visibility paths.

The definition extends in the natural way to sets and sequences of attribute names.

From knowledge about the structure of a permutation expression $\mathcal{L}(M_i)$, we can infer knowledge about the structure of other permutation expressions $\mathcal{L}(M_j)$, if attribute names from i are visible at j.

Lemma 4 *Let e be a relational algebra expression and let $\mathcal{L}$ be the maximal solution to* $\text{Eq}(\mathcal{S})$*, where $\mathcal{S}$ contains at least all the inequalities in* $\text{InEq}(e)$*.*

1) *Assume that $\mathcal{L}(M_i)$ contains $\langle X \rangle$ as a subexpression. If Y at j is visible from X at i, then $\langle Y \rangle$ is a subexpression of $\mathcal{L}(M_j)$.*

2) *Assume that $\mathcal{L}(M_i)$ contains $\mathcal{R}(U)$ as a subexpression. Let $X = \mathcal{A}(U)$. If Y at j is visible from X at i, then $\mathcal{R}(V)$, for some V with $\mathcal{A}(V) = Y$ and $k_v = k_u$, is a subexpression of $\mathcal{L}(M_j)$.*

Proof. We only give the proof of the first statement; the proof of the second is very similar.

In this proof, if X has cardinality 2, i.e., it is of the form $\{A, B\}$, $\langle X \rangle$ can equivalently be written $\mathcal{R}(A, B)$. In that case, whenever we say $\langle X \rangle$, it could just as well be $\mathcal{R}(A, B)$. Alternatively, the normal form could be changed to disallow the $\langle X \rangle$ construction for X having cardinality 2.

The proof is by induction in the length of the visibility path through which Y is visible from X. For the base case, the length being zero, there is nothing to show as the premise reduces to X at i being visible from X at i. For the induction step, assume that $\langle Y \rangle$ is a subexpression of $\mathcal{L}(M_{j'})$, where $e_{j'} = \pi_Z(e_j)$. We want to argue that $\langle Y \rangle$ is a subexpression of $\mathcal{L}(M_j)$. According to the inequalities from Definition 13, $M_{j'} \subseteq M_j|_Z$ and $M_j \subseteq \mathcal{C}(M_{j'}, \langle E_j \backslash E_{j'} \rangle)$. Since the latter implies that

$M_j|_Z \subseteq M_{j'}$, we have that $M_{j'} = M_j|_Z$. As $Y \subseteq Z$, by definition of permutation projection, $\langle Y \rangle$ is a subexpression of $\mathcal{L}(M_j)$. The above also proves the reverse, namely that if $\langle Y \rangle$ is a subexpression of $\mathcal{L}(M_j)$ and Y is visible at j', then $\langle Y \rangle$ is also a subexpression of $\mathcal{L}(M_{j'})$. The argument for join is very similar, and the fact that the induction step also goes through for renaming is trivial. □

The crucial part of the correctness proof is the following lemma. One could fear that even if the expression e had a perfect sort order assignment and it was contained in $\mathcal{L}$, then maybe adding an extra inequality to the system by replacing a $\langle X \rangle$ construction or a $\mathcal{R}(P)$ construction by another permutation expression would have the effect of ruling out all remaining perfect sort orders assignments, in the sense that no perfect sort order assignment would be contained in the new maximal solution. We prove that this does not happen. We use the notation $x[y/z]$ to denote x with z replaced by y.

Lemma 5 *Let e be a relational algebra expression and let $\mathcal{L}$ be the maximal solution to $\mathcal{S}$, where $\mathcal{S}$ contains at least all the inequalities in* InEq*(e). Assume that there exists a perfect sort order assignment, f, which is contained in $\mathcal{L}$.*

1) *Assume that for some i, $\langle X \rangle$ is a subexpression of $\mathcal{L}(M_i)$. Let $\mathcal{S}'$ be the set of inequalities $\mathcal{S}$ with the addition of $M_i \subseteq \mathcal{L}(M_i)[p/\langle X \rangle]$, where p is any permutation expression over X. If $\mathcal{L}'$ is the maximal solution to $\mathcal{S}'$, then there exists at least one perfect sort order assignment contained in $\mathcal{L}'$.*

2) *Assume that for some i, $\mathcal{R}(Q)$ is a subexpression of $\mathcal{L}(M_i)$ such that Q does not contain any $\mathcal{R}$-constructs. Let $\mathcal{S}'$ be the set of inequalities $\mathcal{S}$ with the addition of $M_i \subseteq \mathcal{L}(M_i)[\mathcal{C}(Q)/\mathcal{R}(Q)]$ and let $\mathcal{S}''$ be $\mathcal{S}$ with the addition of $M_i \subseteq \mathcal{L}(M_i)[\mathcal{C}(Q^R)/\mathcal{R}(Q)]$. If $\mathcal{L}'$ and $\mathcal{L}''$ are the maximal solutions to $\mathcal{S}'$ and $\mathcal{S}''$, respectively, then at least one perfect sort order assignment is contained in each of $\mathcal{L}'$ and $\mathcal{L}''$.*

Proof. We prove each statement separately.

1) Let s be the subsequence of $f(i)$ which belongs to $[\![\langle X \rangle]\!]$, and let t be any sequence from $[\![p]\!]$, i.e., t is over X. We now define a sort order assignment g. Let j be some node, we define the value of g on j. Let $s_1, \ldots, s_k$ be all the strings at j which are visible from s at i. For each s_h, let t_h be the string at j which is visible from t at i through the same visibility path as s_h was visible from s through. The value of g on j should now be $g(j) = f(j)[t_1/s_1, \ldots, t_k/s_k]$. We prove that g is a perfect sort order assignment contained in $\mathcal{L}'$.

   From Lemma 4, it follows that $Y_1, \ldots, Y_k$ exist such that for all h, s_h and t_h are strings over Y_h, and $\langle Y_h \rangle$ is a subexpression of $\mathcal{L}(M_j)$. Therefore, each two sets have to be identical or disjoint. This means that g is well-defined, since $g(j)$ is a permutation of E_j.

   We will argue that g is a perfect sort order assignment. If e_j and $e_{j'}$ are identical relation names, then exactly the same sequences are visible at the two nodes, so $g(j) = g(j')$. Now, assume that $e_j = \pi_Z(e_{j'})$. If some sequences s'_h and t'_h are visible at j', then $\langle Y_h \rangle$ is a subexpression of $\mathcal{L}(M_{j'})$, by Lemma 4. From Definition 1, it is obvious that only sequences which are Z-prefixed can belong to $[\![\mathcal{L}(M_{j'})]\!]$. Thus, either $Y_h \subseteq Z$ or $Y_h \subseteq E_{j'} \setminus Z$. It now follows from the definition of visibility that $g(j)$ and $g(j')$ agree on the Z-part. We have proven that g is perfect. The fact that g is contained in $\mathcal{L}'$ follows from Lemma 3 and from the fact that all substrings in $g(j)$ which are different from the ones in $f(j)$ are over Y_h for some subexpression $\langle Y_h \rangle$ of $\mathcal{L}(M_j)$.

The argument for join is very similar. Renaming is trivial.

2) Since $[\![\mathcal{R}(Q)]\!] = [\![\mathcal{C}(Q)]\!] \cup [\![\mathcal{C}(Q^R)]\!]$ and since f is contained in $\mathcal{L}$, the subsequence of $f(i)$, s, which belongs to $[\![\mathcal{R}(Q)]\!]$, must belong to either $[\![\mathcal{C}(Q)]\!]$ or $[\![\mathcal{C}(Q^R)]\!]$. The sequence s is of the form $s_1 \cdot s_2 \cdots s_{k_q}$, where s_h belongs to $[\![q_h]\!]$ for all h. Let t be the sequence $s_{k_q} \cdots s_2 \cdot s_1$ (which belongs to $[\![\mathcal{C}(Q^R)]\!]$).

We can now proceed using s and t as we did in 1). The sort order assignment g is defined in exactly the same way and the proof of g being well-defined is practically the same. Now we consider the proof of g being perfect. Join is the more difficult case, so assume that $e_j = e_{j_1} \bowtie e_{j_2}$. If $\mathcal{R}(U)$ is a subexpression of $\mathcal{L}(M_j)$, where $\mathcal{A}(U)$ at j is visible from $\mathcal{A}(Q)$ at i, then there are the following cases: $\mathcal{A}(U) \subseteq E_{j_1} \setminus E_{j_2}$, $\mathcal{A}(U) \subseteq E_{j_2} \setminus E_{j_1}$, or $\mathcal{A}(U) \subseteq E_{j_1} \cap E_{j_2}$. These cases are similar to the ones from 1).

The only additional possible case is $\mathcal{A}(U) = (E_{j_1} \cup E_{j_2}) \setminus (E_{j_1} \cap E_{j_2})$. In any other case, the permutation expression would necessarily contain sequences not in $[\![E_{j_1} \otimes E_{j_2}]\!]$, which is impossible; see Definition 13. So, we can now assume that $\mathcal{R}(U)$ is of the form $\mathcal{R}(u_1, \ldots, u_p, \ldots, u_k)$, where $\mathcal{A}(u_1, \ldots, u_p) = E_{j_1} \setminus E_{j_2}$ and $\mathcal{A}(u_{p+1}, \ldots, u_k) = E_{j_2} \setminus E_{j_1}$. The $u_1, \ldots, u_p$ in $\mathcal{L}(M_j)$ are not immediately surrounded by an $\mathcal{R}$-construct because then $\mathcal{L}(M_j)$ would also contain that $\mathcal{R}$-construct and the one we are currently looking at would not be an inner-most one. This follows from Lemma 4. Furthermore, in this case, we must have $k = 2$. Otherwise at least two of the u_h's would also appear in $\mathcal{L}(M_{j_1})$ or $\mathcal{L}(M_{j_2})$. Since they are not surrounded by an $\mathcal{R}$-construct, the order of the two u_h's would be fixed. However, then the ordering of these two permutation expressions would also be fixed in $\mathcal{L}(M_{j_1}) \otimes \mathcal{L}(M_{j_2})$. Since $\mathcal{L}(M_j) \subseteq \mathcal{L}(M_{j_1}) \otimes \mathcal{L}(M_{j_2})$, we get a contradiction, so $k = 2$. This means that the substring of $f(j)$ that we are changing when defining g from f is of the form $s_1 \cdot s_2$ and it is replaced by $s_2 \cdot s_1$. Thus, there are no violations.

□

Theorem 4 *Algorithm* Find Sort Order Assignment *solves the problem stated in its specification.*

Proof. If we find a solution $\mathcal{L}$ to the system $\mathcal{S}$ in the algorithm such that $\forall M_i : |[\![\mathcal{L}(M_i)]\!]| = 1$, then the sequences in $\mathcal{L}$ clearly form a perfect sort order assignment.

From Corollary 3, it follows that $\mathcal{L}$, the largest solution to $\textsc{Eq}(\textsc{InEq}(e))$, contains every perfect sort order assignment for e. Furthermore, Lemma 5 proves that if the largest solution to $\mathcal{S}$ contains a perfect sort order assignment for e, then so does the largest solution to $\mathcal{S}$ with the inequality $M_i \subseteq p$ added. Thus, either no perfect sort order assignment exists for e or the algorithm will find one. $\square$

We now prove an upper bound on the time complexity. If a permutation expression contains k attribute names, then it has size at most $O(k \cdot \log(k))$. This is an obvious consequence of insisting that permutation expressions be kept in normal form. When we perform operations on permutation expressions, we also operate on the sets, i.e., the X's in $\langle X \rangle$. To do that efficiently, they need to be sorted, but this can be done in $O(k \cdot \log(k))$. Going through the structure of permutation expressions, as we do when we perform operations on them, takes linear time in their sizes, i.e., again $O(k \cdot \log(k))$. The total time to compute the next iteration is then $O(n \cdot k \cdot \log(k))$, where we now let k be the maximum number of attribute names in any permutation expression.

To check for having reached the fixed point, we can simply compare the ith and the $(i-1)$th iteration before continuing. Again, this will take time $O(k \cdot \log(k))$ for each entry. In total, we obtain $O(n \cdot k \cdot \log(k))$ once again.

Finally, it is easy to check that whenever we perform one of our operations on a permutation expression, the semantic size of the result will be at most half the semantic size of the argument. The height of each (component) lattice is bounded by $k!$ (the number of different permutations of a set of size k), so the number of iterations is bounded by $O(n \cdot \log(k!))$. As $k! \in O(k^k)$, we have that $\log(k!) \in O(k \cdot \log(k))$. We get a total of $O(n \cdot k \cdot \log(k))$ iterations.

In summary, we iterate at most $O(n \cdot k \cdot \log(k))$ times performing at most $O(n \cdot k \cdot \log(k))$ work each time around. This gives a total complexity of $O(n^2 \cdot k^2 \cdot \log^2(k))$ to find one fixed point.

In the algorithm, we find a new fixed point for each iteration of the **while**-loop. However, if we calculate the new fixed point from the old one, we can obtain $O(n^2 \cdot k^2 \cdot \log^2(k))$ as the total complexity of our algorithm.

When we find a fixed point $\langle x_1, \ldots, x_n \rangle$, which does not yet consist of singleton sets, we choose a variable M_i and include an extra inequality $M_i \subseteq x$, where $x \subseteq x_i$ (actually $[\![x]\!] \subseteq [\![x_i]\!]$, as we represent our sets using permutation expressions).

We now perform an analysis to find out what happens when we continue iterating from $\chi = \langle x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n \rangle$. Let $\langle y_1, \ldots, y_n \rangle$ be the next value. As $\langle x_1, \ldots, x_n \rangle$ is a fixed point, the y_j's which do not depend on M_i will not have changed. Because of monotonicity, the y_j's which depend on M_i can only become smaller than the corresponding x_j's. Finally, as we have included $M_i \subseteq x$ in our system, $y_i \subseteq x \subseteq x_i$. We have shown that $\langle y_1, \ldots, y_n \rangle \sqsubseteq \chi$. From Theorem 3, it follows that the largest fixed point less than or equal to χ can be found by iterating from χ.

As the semantic size of x is at most half the semantic size of x_i, we can apply exactly the same argument again, and we obtain that the total number of iterations in the **while**-loop as a whole is bounded by $O(n \cdot k \cdot \log(k))$, and we get a total complexity of $O(n^2 \cdot k^2 \cdot \log^2(k))$.

```
Algorithm: Minimal resort
Input: an expression $e$
Output: a minimal number of places to resort
Method:
  for $k := 1$ to $n$ do
    forall $k$-tuples $(i_1, \ldots, i_k)$ do
      let $\mathcal{S}$ be free $\textsc{InEq}(e)$ on $M_{i_1}, \ldots, M_{i_k}$
      let $\mathcal{L}$ be the maximal solution to $\mathcal{S}$
      if $\mathcal{L}$ is nontrivial then
        use the technique from algorithm Find Sort Order to find
        a sort order assignment $f$ contained in $\mathcal{L}$
        output "$f$" and $i_1, \ldots, i_k$
        halt
      endif
    endforall
  endfor
```

Figure 6: Algorithm "Minimal resort".

7.2. Exhaustive search

We will briefly discuss what to do when no solution exists, i.e., when no perfect sort order assignment exists for an expression. In that case, we have to resort somewhere in the expression during the evaluation, and, obviously, we want to find the (a) minimal number of places to do this.

By systematically checking all possibilities, we can use the algorithm Find Sort Order Assignment to find a minimal number of places to resort. Of course, this process will have exponential time complexity. However, for any fixed constant p, if we know that the minimal number of places to resort is less than p, or if we are only interested in solutions, where the number of places to resort is less than p, then the algorithm runs in polynomial time.

First, we need to be able to register in the inequality system that we resort at a certain node.

Definition 16 *Let e be a relational algebra expression. We modify* $\textsc{InEq}(e)$ *to obtain a slightly different inequality system. Define* free $\mathcal{S}$ on M_i *by the following:*

- *substitute M_i with a new variable name in the single up-inequality in which M_i appears.*
- *if e_i is a relation name, then delete all of the inequalities which relates M_i to other expressions, e_j, where e_i and e_j are identical relation names.*

The algorithm can now be defined (Figure 6).

8. Integration into Query Optimizers

Previous (heuristics-based) work on this problem has mixed other optimization issues into the algorithms. This means that applying these kinds of techniques have had implications for a variety of other optimization design decisions. In contrast, we have singled out the problem and given algorithms for exact solutions to queries. This means that our technique can be useful in any query optimizer. Since, if nothing else, it can be used last to obtain more information before choosing the final access plan. In the following, we briefly discuss some of the most common optimization issues in relation to our work.

Algebraic manipulations (like pushing selections etc.) which practically always pay off should be performed before our algorithm is applied.

The next problem is the order in which to evaluate joins, and which algorithms to use. If one relation in a join is small, the best method is to keep the small relation in random access memory and to perform the join by repeatedly joining small chunks of the larger relation with all of the smaller. This means that only the sort order of the larger relation followed by the smaller can be output. In the algorithms, this can be modeled by replacing this join with a unary version and taking the expression giving rise to the smaller relation out for separate analysis.

If more than two relations are to be joined, join orders can be determined first, and the algebraic structure of the query can be changed to reflect this. Some systems use k-way merges, i.e., k (sorted) relations are merged at the same time. This cannot be modeled with the operators we have, but a join of any arity can be defined along the same lines as the binary one, and results carry over. It is also possible to just analyze parts of a query. If hashing is preferred for removing duplicates after a projection, our algorithm could be used only on the remaining parts.

If the query optimizer is capable of finding identical subexpressions, this can also be exploited here. The subexpression can be taken out and analyzed separately. Then the whole query, with a new (non-existing) relation name put in for the identical subexpressions, can be analyzed pretending that the permutation expression associated with the root of the subexpression is the set of indexes for this new relation.

In the algorithms, for generality, we have used the whole schema as a key. In practice, a key is often just one attribute. This will speed-up the analysis, since only the keys are of interest in deciding on sort orders. So, instead of using the schema, it is sufficient to use the part of the schema containing the keys.

Sometimes, our algorithms may find several solutions, i.e., there may be different sets of nodes which form a set of minimal violations to some sort order assignment. In those cases, heuristics on the sizes of temporary relations can be applied after our analysis to choose the best of the ones offered.

The proofs and algorithms presented here can be adapted to many different models as discussed above, but, as usual, it will require some work and experimentation to obtain the best possible solution for a concrete query language.

References

1. L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, "On Sorting Strings in External Memory", *Proc. 29th Ann. ACM Symp. on Theory of Computing* (1997) 540-548.

2. R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indexes”, *Acta Inform.* **1** (1972) 173-189.
3. E. F. Codd, “A Relational Model of Data for Large Shared Data Bases”, *Comm. ACM*, **13 (6)** (1970) 377-387.
4. B. C. Desai, *An Introduction to Database Systems* (West Publishing Company, 1990).
5. M. R. Garey and D. S. Johnson, *Computers and Intractability* (W. H. Freeman, 1979).
6. T. H. Merrett, “Why Sort-Merge Gives the Best Implementation of the Natural Join”, *SIGMOD Record* **13 (2)** (1983) 39-51.
7. P. G. Selinger and M. M. Astrahan and D. D. Chamberlin and R. A. Lorie and T. G. Price, “Access Path Selection in a Relational Database Management System”, *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1979) 23-34.
8. M. I. Schwartzbach, “Type Inference with Inequalities”, Technical Report PB-336, Department of Computer Science, Aarhus University, 1990.
9. M. I. Schwartzbach, “Type Inference with Inequalities”, Proc. Intl. Joint Conf. on Theory and Practice of Software Development, Vol. 1: Colloquium on Trees in Algebra and Programming, S. Abramsky, T. S. E. Maibaum (eds.), *Lecture Notes in Computer Science* **493** (1991) 441-455.
10. J. M. Smith and P. Y.-T. Chang, “Optimizing the Performance of a Relational Algebra Database Interface”, *Comm. ACM* **18 (10)** (1975) 568-579.
11. A. Tarski, “A Lattice-Theoretical Fixpoint Theorem and its Applications”, *Pacific J. Math* **5** (1955) 285-309.
12. J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1 (Computer Science Press, 1988).