Strategies for Expression Evaluation Using Sort-Merge Algorithms

Kim S. Larsen

Aarhus University

September 1992

Abstract

The sort-merge technique for evaluating relational algebra and calculus expressions was advocated very early and is a very widely used implementation technique. We present an algorithm for query analysis prior to execution with the aim of determining sort orders for every subexpression in such a way that resorting can be avoided during the actual evaluation. We prove that our algorithm will find such a solution, if one exists. In that case, we get the additional benefit of perfect pipelining, which implies that we do not have to save temporary results of evaluating subexpressions. The algorithm's running time is quadratic in the size of the expression.

In case no assignment of sort orders to subexpressions exists such that resorting can be avoided entirely, the aim is to find a minimum number of places to resort. We also consider this problem.

1 Introduction

Sort-merge algorithms have been used for a long time for evaluating relational algebra and calculus expressions. The technique was advocated very early [Mer83] and is quite dominant as an evaluation technique; partly due to its simplicity. Many textbooks contain details and motivation; see [Des90], for

example. Before we give an account of the contents of this paper, we briefly describe the sort-merge algorithms. More detailed descriptions can be found in the textbooks, but it seems appropriate to include a short summary here.

So, how do we evaluate $r_1 \cup r_2$, say, using a sort-merge algorithm? The reason for using an algorithm at all instead of simply concatenating the two sets of tuples is that we want to avoid duplicates in the result. Much can be said both for and against forbidding duplicates in relations, but when the decision is made to avoid them, a sort-merge algorithm would often be used.

Assume that the schemas of r_1 and r_2 are $R_1 = R_2 = \{A, B, C\}$. We could choose the *sort order* BCA, meaning that we intend to sort the relations lexicographically with B being most significant, C second most significant, and A lest significant. If we *sort* r_1 and r_2 separately according to this ordering, we can afterwards *merge* them into the result. The point being, of course, that if r_1 and r_2 contain the same tuple, then we will see these two tuples at the same time during the merge, and we can eliminate one of them. We also note that the result is automatically BCA-sorted.

What about the natural join of two relations, $r_1 \bowtie r_2$? Here, the motivation for using a sort-merge algorithm is not removal of duplicates, but efficiency [Mer83]. If $|r_1| = |r_2| = n$, then $|r_1 \bowtie r_2|$ may be as much as n^2, so this is the best complexity we can hope for in general. However, if the size of the result is an order of magnitude smaller than n^2, then we can do better. If we first sort r_1 and r_2 and then merge, we obtain a complexity of $O(n \cdot \log(n))$, if the result is at most of size $O(n \cdot \log(n))$. The arguments have to be sorted such that the attribute names in $R_1 \cap R_2$ are the most significant in the ordering. Here is an example. If $R_1 = \{A, B, C, D\}$ and $R_2 = \{A, B, E, F\}$, then one possible choice is to sort r_1 according to $BACD$ and r_2 according to $BAFE$, i.e., both sequences start with the intersection of the schemas, $\{A, B\}$, and in the same order. After sorting, we merge the two relations, exploiting the fact that all the tuples from r_1 and r_2 which agree on A and B will come in sequence, and we output the Cartesian product of the CD part of the tuples from r_1 and the FE part of the tuples from r_2. We observe that without any extra sorting, we can output this result in two different ways; either according to $BACDFE$ or according to $BAFECD$.

The remaining relational operators are either very similar to union or join, or they are uninteresting because there are no requirements on the sort orders.

At this point, we want to emphasize that the collection of sort-merge algorithms described above is just one choice among many. In a real implementations one would probably not sort a relation entirely. One would accept that a relation is sorted on the first attribute name, or that there is an index for the first attribute name, or something similar. One would then take care of the sorting on the remaining attributes during the merge. Also, one might detect identical binary operators appearing next to each other and use a k-way merge [Knu73]. There are a great variety of possible implementation decisions and it is impossible to cover them all. In this paper, we choose one of them in order to demonstrate the technique. The proofs and algorithms presented can easily be adapted to other models.

We will now describe the *sort-merge problem*. Our objective is to avoid resorting. This goal can be achieved if the sort-merge problem can be solved. We can phrase the sort-merge problem as follows. Given à query, does there exist an assignment of sequences to all subexpressions such that the sort-merge requirements are fulfilled for each operator and such that the sequence assigned to a subexpression can be produced by a sort-merge algorithm *without* extra sorting; given that the arguments are already sorted according to their assigned sequences. Furthermore, if the same relation name appears more than once, then all occurrences have to be assigned the same sequence.

Consider the query $q = (r_1 \times r_2) - (\delta_{C \leftarrow D, D \leftarrow C}(r_3 \bowtie r_4))$, where $R_1 = R_4 = \{A, B, D\}, R_2 = \{C\}$, and $R_3 = \{A, B, C\}$. One solution to the sort-merge problem for this query is listed below.

Subexpression	Assigned sequence
r_1	BAD
r_2	C
$r_1 \times r_2$	$BADC$
r_3	BAC
r_4	BAD
$r_3 \bowtie r_4$	$BACD$
$\delta_{C \leftarrow D, D \leftarrow C}(r_3 \bowtie r_4)$	$BADC$
q	$BADC$

Notice that all the requirements are fulfilled. For instance, the sequences assigned to r_3 and r_4 agree on $R_3 \cap R_4$ and the two expressions in the difference are assigned identical sequences ($BADC$). Also, we notice that sequences

assigned to subexpressions represent sort orders which can be obtained without extra sorting, if the arguments are sorted according to their assignments. For instance, if r_1 is sorted according to BAD and r_2 is sorted on C, then we can output one of the orderings $BADC$ or $CBAD$ from $r_1 \times r_2$ without additional sorting (and $BADC$ has been chosen here).

The consequences of the above query having a solution are that if we sort the argument relations r_1, r_2, r_3, and r_4 according to their assigned sequences, then no further sorting is necessary. ln addition, we can pipeline the tuples from these argument relations through the whole expression a few at a time such that no temporary relations are needed; we can simply write out the result directly. If one or more of the argument relations are already sorted, then it might be possible to take advantage of that depending on whether these concrete orderings appear in any of the solutions. We can easily adjust our results to take this into account.

The sort-merge problem has been considered before in [SC75]. They have designed an algorithm which simply makes a pass up the syntax tree of a query and a pass down the syntax tree, and sometimes it finds a solution. In contrast, we always find a solution if one exists. In [SC75], Smith and Chang concentrate on describing how to sort or create dictionaries, whereas we work on a higher level of abstraction and instead put emphasis on finding an exact solution.

The problem of finding a sort order assignment becomes more and more difficult as the size of queries grow. Several occurrences of the same relation name make queries particularly difficult to handle. Consider a query like $(\pi_A(r_1) \times r_2) - exp$, for example, where $R_1 = \{A, B\}, R_2 = \{B\}$, and r_1 appears in exp. In this query, the complication of two occurrences of the same relation name implies that the top-most operator, difference, cannot be dealt with by simply considering its immediate arguments, First, information from r_1 in the projection has to be propagated to the other occurrences of r_1 in exp. Using the algorithm of [SC75], it is just as likely that the occurrences of r_1 in exp are sorted on B first as on A first.

In this paper, we present an algorithm which finds a solution if one exists (in fact, we find all solutions and then select one). We do this by first generating a system of inequalities, which constitutes the constraints that the operators in the query impose on their arguments and limit the possible orderings they

can output.

Our algorithm operates on sets of permutations. If a subexpression has schema $\{A, B, C\}$, then we begin by considering all the permutations of this set $(ABC, ACB, BAC, BCA, CAB, CBA)$ as candidates for solutions. During the process, we gradually limit this set. Representing these sets of permutations directly leads to an exponential-time algorithm, and though queries and schemas are usually of a reasonable size, we would like to obtain a polynomial time algorithm. If all subsets of permutations could appear as values in our algorithm, then we would not have any hope of improvement over the naive approach. Fortunately, this is not the case, and we develop a considerable more compact notation for the possible subsets.

The outline of the paper is as follows. In section 2, we state the sort-merge problem formally. In section 3, we answer the question of how to solve inequality systems. In section 4, we develop a novel concept of *permutation expressions*. In section 5, the solution to the sort-merge problem is presented in the form of an algorithm, the complexity of which we analyze. In section 6, we present an algorithm which finds a minimum number of places to resort, when no solution exists which avoid resorting entirely. In section 7, we conclude.

2 The Sort-Merge Problem

In this section, we define the sort-merge problem formally. First, we need some basic definitions on relational algebra and sequences of attribute names.

Definition 2.1 Let ATT be a set of attribute names and DOM a set of values. A relation r is a pair consisting of a schema $\text{SCH}(r)$, which is a finite subset of ATT, and a finite set of tuples, which are total functions from $\text{SCH}(r)$ to DOM.

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

$$e ::= r \mid e \cup e \mid e - e \mid e \bowtie e \mid \sigma_b(e) \mid \pi_X(e) \mid \delta_d(e)$$

where r can be any relation name, b is a boolean expression of the usual restricted form, X is a list of attribute names, and d is a list of pairs of attribute names each of which is of the form $A \leftarrow B$.

Schemas can be determined statically, so we can extend SCH to expressions. We let R denote the schema of a relation r and E the schema of a relation expression e. □

The definition of relational algebra is entirely standard. More details can be found in any introductory textbook on the subject; see [Ull88], for example.

Definition 2.2 If S is a set, then $Permute(S)$ is the set of all *permutations* of S, i.e., all sequences of elements from S where each element in S appears exactly once in each sequence. A dot, "$\cdot$", will denote concatenation of sequences (and sets of sequences).

For example, if $S = \{A, B\}$, then $Permute(S) = \{AB, BA\}$. Also, $\{AB, BA\} \cdot \{CD, DC\} = \{ABCD, ABDC, BACD, BADC\}$.

If s is a sequence $A_1 \cdots A_k$, then we let $\{s\}$ denote the set $\{A_1, \ldots, A_k\}$, A sequence can be renamed in exactly the same way as a relation. We use the notation $s[d]$ for this, where s is a sequence and d is a list of pairs of the form $A \leftarrow B$ as in the renaming of relations. Furthermore, $s|_X$ will denote sequence projection. We also use $s|_{\{X\}}$ with exactly the same meaning as $s|_X$. □

For example, $ABC[B \leftarrow D] = ADC$ and $ABCD|_{AC} = ABCD|_{\{A,C\}} = AC$.

All of these operators are extended to sets of sequences in the obvious way.

A query can contain identical subexpressions, though optimizers will usually remove these. However, a query can certainly contain several identical relation names. Thus, a subexpression does not uniquely identify a position in the query. For that reason, we have to associate a unique identifier with each occurrence of a subexpression; we have chosen to use integers.

Definition 2.3 An *enumeration* of a relational algebra expression e is an assignment of integers to all subexpressions of e, including relation names, such that no two subexpressions are assigned the same number and such that they are numbered from 1 through n for some n. The *size* of e, denoted $|e|$, is n. If the same subexpression appears several times, then each occurrence is assigned its own number.

We let SUBEXP(i) denote the subexpression with number i and let OP(i) denote the outermost operator of SUBEXP(i). Finally, let ARGS(i) denote

the set of numbers assigned to the subexpressions which are arguments to $\textsc{op}(i)$. □

Often, we will not explicitly mention enumerations; instead, we shall assume that we have one given. In the following, we list the requirements for each operator which sort-merge algorithms impose. These are exactly as discussed in the introduction.

Definition 2.4 The *operator requirements* for each operator are listed below. Assume that s_i, $i = 1, 2$, indicates how the output from e_i is sorted.

Expression	Requirement
r	–
$e_1 \cup e_2$	$s_1, s_2 \in \mathit{Permute}(E_1)$ and $s_1 = s_2$.
$e_1 - e_2$	Same as union.
$e_1 \bowtie e_2$	$\exists t, s_1', s_2' : t \in \mathit{Permute}(E_1 \cap E_2), s_i = t \cdot s_i', i = 1, 2.$
$\sigma_b(e_1)$	–
$\pi_X(e_1)$	$\exists t, s_1' : t \in \mathit{Permute}(X)$ and $s_1 = t \cdot s_1'$.
$\delta_d(e_1)$	–

□

As mentioned in the introduction, join can output two different orderings without any extra sorting. This is captured formally in the following definition.

Definition 2.5 Let E_1 and E_2 be sets and let $s_i \in \mathit{Permute}(E_i), i = 1, 2$. Now, $\otimes$ is defined as follows:

$$s_1 \otimes s_2 = \begin{cases} \{s_1 \cdot s_2', s_2 \cdot s_1'\}, & \text{if } \exists t, s_1', s_2' : t \in \mathit{Permute}(E_1 \cap E_2), \\ & s_i = t \cdot s_i',\ i = 1, 2 \\ \emptyset & \text{otherwise} \end{cases}$$

The definition is extended to sets in the natural way. □

Finally, we define the output sortings which can be obtained as output from operations. This concludes the characterization of sort-merge algorithms as we have chosen to present them. Let us emphasize again that this is just one possible choice among many and that our results easily can be adapted to other reasonable assumptions about sort-merge algorithms.

Definition 2.6 We assume that the requirements from definition 2.4 hold and that $s_i \in \mathit{Permute}(E_i),\ i = 1, 2$. We now define the output sortings that can be produced from the given input sortings without additional sorting.

Expression	Ordering
r	$\mathit{Permute}(R)$.
$e_1 \cup e_2$	$\{s_1\}$.
$e_1 - e_2$	Same as union.
$e_1 \bowtie e_2$	$s_1 \otimes s_2$.
$\sigma_b(e_1)$	$\{s_1\}$.
$\pi_X(e_1)$	$\{s_1\|_X\}$.
$\delta_d(e_1)$	$\{s_1[d]\}$.

□

Notice that by definition a relation can produce any sequence over its schema. This is because of our assumption that if a solution exists, then we sort the argument relations to a query according to their assigned sequences, after which no resorting is necessary during the actual evaluation of the query. This is where information about presorted relations or relations with an index could be incorporated. We can also benefit from information about which relations are laid out as search trees. As search tree data will typically cluster, this can sometimes give speed-up comparable to exploiting the fact that a relation is stored in sorted order.

We can now formally list the properties an assignment of sequences to subexpressions in a query should have.

Definition 2.7 If e is a relational algebra expression of size n, then a *sort order assignment* for e is a function

$$f : \{1, \ldots, n\} \to \bigcup_{i \in \{1,\ldots,n\}} \mathit{Permute}(\textsc{Sch}(\textsc{SubExp}(i)))$$

such that for all $i \in \{1, \ldots, n\}$, the following conditions hold, where we assume that $\textsc{Args}(i) = \{i_1, i_2\}, s_1 = f(i_1)$ and $s_2 = f(i_2)$.

- definition 2.4 is fulfilled.
- $f(i)$ can be produced from s_1 and s_2 according to definition 2.6.

- if SubExp(j) and SubExp(j') are identical relation names, then $f(j) = f(j')$.

□

The *sort-merge problem* can now alternatively and more formally be formulated as follows: given a query, does there exist a sort order assignment for it?

Before we conclude this section notice that there is no difference in the treatment of union and difference in the sort-merge problem. In fact, they are both special cases of join. When join is simply an intersection, because the schemas of the arguments are identical, the requirements and properties are like the ones for union and difference. Finally, notice that selection is completely uninteresting in this context, as it has no requirements and it preserves the ordering of the input. Thus, before analyzing a query, we can delete all selections and change all unions and differences to joins. This will cut down on notation and cases in definitions and proofs to come. In summary: we only need to analyze queries containing the operators join, project, and renaming.

3 Solving Systems of Inequalities

Before we move on, we need some theory on how to find maximal solutions to systems of inequalities. Basically, we adapt Tarski's work on fixed points for functions defined on complete lattices to our concrete problem. We do not find any need to comment much on the definitions and proofs in this section.

Definition 3.1 Let $(U, \sqsubseteq)$ be a complete lattice [Tar55] with greatest lower bound $\sqcap$ and least upper bound $\sqcup$. An *inequality system* on $(U, \sqsubseteq)$ consists of 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 $f : U^k \to U$ 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. □

Lemma 3.2 If an inequality system is satisfiable, then it has a unique largest solution.

Proof Let $\{\mathcal{L}_1, \mathcal{L}_2, \ldots\}$ be the set of all solutions. Now, define $\mathcal{L}$ as follows:

$$\forall M : \mathcal{L}(M) = \sqcup_i \mathcal{L}_i(M)$$

Then $\mathcal{L}$ is also a solution since

$$\begin{array}{rcl} \mathcal{L}(M_0) & = & \sqcup_i \mathcal{L}_i(M_0), \text{ by definition} \\ & \sqsubseteq & \sqcup_i f(\mathcal{L}_i(M_1), \ldots, \mathcal{L}_i(M_k)), \text{ since } \mathcal{L}_i \text{ is a solution} \\ & \sqsubseteq & f(\sqcup_i \mathcal{L}_i(M_1), \ldots, \sqcup_i \mathcal{L}_i(M_k)), \text{ since } f \text{ is monotonic} \\ & = & f(\mathcal{L}_i(M_1), \ldots, \mathcal{L}_i(M_k)), \text{ by definition} \end{array}$$

As $\mathcal{L}$ belongs to the set of all solutions and, by definition, is at lest as large as any other solution, it is the unique largest solution. □

Definition 3.3 Let $\mathcal{S}$ be an inequality system. We define $\text{EQ}(\mathcal{S})$ 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 holds. The system is *satisfible* if a solution exists. □

Proposition 3.4 If $\mathcal{S}$ is an inequality system, then $\text{EQ}(\mathcal{S})$ is satisfiable and has a unique largest solution.

Proof See [Tar55]. □

Lemma 3.5 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 Let $\mathcal{L}$ be a solution to $\mathcal{S}$. If $H_i = f(M'_1, \ldots, M'_p)$, then we let $\mathcal{L}(H_i)$ denote the value $f(\mathcal{L}(M'_1), \ldots, \mathcal{L}(M'_p))$. Now, assume that for some $M_q, \mathcal{L}(M_q) \sqsubset \sqcap_i \mathcal{L}(H_i)$. 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 $H_{i'}$,

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

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

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

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 $\mathcal{S}$. □

There is a standard technique for solving an equality system by iterating a certain function until a fixed point is obtained.

Definition 3.6 Assume that F is a set of monotone functions, each of them from $U^h \to U$, for some h (we mean 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 3.7 A 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 3.8 The iteration function defined above is monotone on U^n.

Proof Trivial. □

Lemma 3.9 Let $(U, \sqsubseteq)$ be a complete lattice and $f : U \to U$ 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 $\sqcap\{v, f(v), \ldots, f^i(v)\} = f^i(v)$. We conclude that 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 as 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 $u' = f(u') \sqsubseteq f(v)$ as f is monotonic. By inductions we see that $u' \sqsubseteq f^k(v)$. But then $u' \sqsubseteq u$, so u is the largest fixed point less than or equal to v. □

Corollary 3.10 If $(U, \sqsubseteq)$ is a complete lattice with top element $\top = \sqcup U$ and $f : U \to U$ is a monotone function, then the largest fixed point can be found as $f^k(\top)$, for some $k \in \mathbb{N}$.

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

Proposition 3.11 The largest solution to an inequality system can be found by iteration of a certain function a finite number of times until a fixed point is reached.

Proof If U is a complete lattice, then U^n is as well [Tar55]. Now combine lemma 3.5, proposition 3.7, proposition 3.8, and corollary 3.10. □

Observation 3.12 In cumputing the largest fixed point in a complete lattice by iteration, there are three costs to consider:

1. the number of iterations to find the fixed point
2. the cost of computing each element of the iteration
3. the cost of checking whether a fixed point has been reached or not

□

4 Representing Sets of Permutations

A set of size k gives rise to $k!$ permutations. In this paper, we are dealing with sets of permutations and implementing algorithms which manipulate these. Because of the potential size of a naive implementation, we have to develop more sophisticated techniques to obtain a good runtime performance. Fortunately, it turns out that we only need to be able to represent a limited class of sets of permutations. The grammar below reflects this.

Definition 4.1 The set of all *permutation expressions* is generated by the following grammar.

$$p ::= \textsc{Nil} \mid A \mid \mathcal{P}(\{A_1, \ldots, A_k\}) \mid \mathcal{C}(p, \ldots, p) \mid \mathcal{R}(p, \ldots, p)$$

where $A, A_1, \ldots, A_k \in \textsc{Att}$.

We shall use $[\![p]\!]$ to represent the set of permutations which p denotes.

$[\![\textsc{Nil}]\!]$ is the empty set of permutations, $[\![A]\!]$ is $\{A\}$, $[\![\mathcal{P}(\{A_1, \ldots, A_k\})]\!]$ is the set of all permutations of the set $\{A_1, \ldots, A_k\}$. $[\![\mathcal{C}(p_1, \ldots, p_k)]\!]$ represents the concatenation of all permutations from the k expressions i.e., $[\![\mathcal{C}(p_1, \ldots, p_k)]\!] = [\![p_1]\!] \cdots [\![p_k]\!]$. Finally, $[\![\mathcal{R}(p_1, \ldots, p_k)]\!]$ is all the permutations which $\mathcal{C}(p_1, \ldots, p_k)$ denotes together with the permutations which $\mathcal{C}(p_k, \ldots, p_1)$ denotes, i.e., $[\![\mathcal{R}(p_1, \ldots, p_k)]\!] = [\![\mathcal{C}(p_1, \ldots, p_k)]\!] \cup [\![\mathcal{C}(p_k, \ldots, p_1)]\!]$ ($\mathcal{R}$ stands for *reversed*).

We let $\textsc{Att}(p)$ denote the set of attribute names which are used in the expression p. This is called the *base set* of p. □

Example 4.2 As an example,

$$[\![\mathcal{R}(A, \mathcal{R}(B, C))]\!] = \{ABC, ACB, BCA, CBA\}$$

and

$$\textsc{Att}(\mathcal{R}(A, \mathcal{R}(B, C))) = \{A, B, C\}$$

□

In order to avoid redundancy, we define a *normal form* for permutation expressions. When performing operations on permutation expressions, results should always be brought in normal form.

In the following, we use P as short for a comma separated list of permutation expressions, $p_1, \ldots, p_{k_p}$ and P' as short for $p_2, \ldots, p_{k_p}$. Thus P can also be written p_1, P'. When no confusion can arise, we shall often simply use k instead of k_p. Finally, we let $\tilde{P}$ stand for the reversed list $p_{k_p}, \ldots, p_1$. We shall use Q, U, V, and T similarly.

Definition 4.3 A permutation expression p is in normal form if and only if

- each subexpression $\mathcal{P}(X)$ has $|X| \geq 3$.
- 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 $\textsc{Nil}$ is contained in p, then $p = \textsc{Nil}$.

□

Proposition 4.4 If p an q are permutation expressions in normal form, then

$$p = q \Leftrightarrow [\![p]\!] = [\![q]\!]$$

Proof Easy. □

It is easy to put a permutation expression into normal form. $\mathcal{P}(\emptyset)$ can be replaced by $\textsc{Nil}$, $\mathcal{P}(\{A\})$ by A, and $\mathcal{P}(\{A, B\})$ by $\mathcal{R}(A, B)$. For $\mathcal{C}$- and $\mathcal{R}$-constructs with only one argument, we can simply remove the $\mathcal{C}$ or the $\mathcal{R}$, i.e., $\mathcal{C}(P)$ is changed to P and $\mathcal{R}(P)$ to P. Furthermore, an expression

$\mathcal{C}(p_1, \ldots, \mathcal{C}(Q), \ldots, p_k)$ can be replaced by $\mathcal{C}(p_1, \ldots, Q, \ldots, p_k)$. Finally, if a NIL appears in an expression, then the whole expression is replayed by NIL.

In section 2, we used a number of functions on sets of permutations. These were: $\otimes$, concatenation, projection, and renaming. Later, we will also use $\cap$. As we will use permutation expressions in our algorithms instead of the actual sets of permutations, which they represent, we have to define similar functions on permutation expressions, e.g., we need a function $\triangle$ such that for any permutation expressions p and q, where $\text{ATT}(p) = \text{ATT}(q) : [\![p \triangle q]\!] = [\![p]\!] \cap [\![q]\!]$. It is especially hard to define $\otimes$ for permutation sequences and we shall do this in several steps. First we define

$$\mathit{Prefix}_X(M) = \{s \in M \mid \exists s', s'' : s' \in \mathit{Permute}(X), s = s' \cdot s''\}$$

where M is a set of permutations (not a permutation expression) and X is a set of attribute names.

This is the set of sequences from M which start with a permutation of the elements from X. We want a similar function for permutation sequences. This will be a help when the other operators are to be defined.

Definition 4.5 An ordered list of permutation expressions $p_1, \ldots, p_k$ is X-*initial* for some set of attribute names X if $\exists i : 1 \leq i \leq k$ such that

$$\text{ATT}(p_1, \ldots, p_{i-1}) \subseteq X,\ \text{ATT}(p_i) \cap X \neq \emptyset,\ \text{ATT}(p_{i+1} \cdots p_k) \cap X = \emptyset$$

This unique i is called the *extent* of X. □

Now we can define **PF** recursively in the structure of permutation expressions, where **PF** is the function on permutation expressions which will correspond to *Prefix*.

Definition 4.6 If for some permutation expression p we have $\text{ATT}(p) = X$, then we define $\mathbf{PF}_X(p) = p$. Also, $\mathbf{PF}_\emptyset(p) = p$. Otherwise, we refer to the table below. If none of those possibilities apply, then we define $\mathbf{PF}_X(p) =$ NIL.

p	Condition	$\mathbf{PF}_X(p)$
$\mathcal{P}(Y)$	$X \subset Y$	$\mathcal{C}(\mathcal{P}(X), \mathcal{P}(Y \setminus X))$
$\mathcal{C}(P)$	P is X-initial (extent i)	$\mathcal{C}(p_1, \ldots, p_{i-1}, \mathbf{PF}_{X'}(p_i, p_{i+1}, \ldots, p_k)$
$\mathcal{R}(P)$	P is X-initial	$\mathbf{PF}_X(\mathcal{C}(P))$
$\mathcal{R}(P)$	$\tilde{P}$ is X-initial	$\mathbf{PF}_X(\mathcal{C}(\tilde{P}))$

where $X' = X \setminus (\bigcup_{j \in \{1,\ldots,i-1\}} \textsc{Att}(p_j))$. □

Proposition 4.7 **PF** is well-defined and implements *Prefix* correctly, i.e.,

$$\forall p : \ \mathit{Prefix}_X([\![p]\!]) = [\![\mathbf{PF}_X(p)]\!]$$

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 an expression denotes (the function $[\![\cdot]\!]$) becones smaller. Finally, we observe from definition 4.5 that if P is X-initial, then $\mathcal{R}(P)$ is not, unless $\textsc{Att}(P) = \textsc{Att}(\mathcal{R}(P)) = X$, in which case we would not use the table. We have argued that **PF** is well-defined.

It is farly easy to check that **PF** implements *Prefix* correctly. The crucial observation is that if P is not X-initial, then $\mathit{Prefix}_X([\![\mathcal{C}(P)]\!]) = \emptyset$. □

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

Proposition 4.8 If $\emptyset \neq X \subset \textsc{Att}(p)$ and $\mathbf{PF}_X(p) \neq \textsc{Nil}$, then $\mathbf{PF}_X(p)$ is of the form $\mathcal{C}(p_1, \ldots, p_k)$ and $\exists i : \ 1 \leq i < k$, $X = \textsc{Att}(p_1, \ldots, p_i)$.

Proof Easy proof by induction in the structure of p following the definition of **PF**. □

We need the following concept.

Definition 4.9 Two permutation wcpressions $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are *comma equalized* if they have the same number of arguments and $\forall i \in \{1, \ldots, k\}$: $\textsc{Att}(p_i) = \textsc{Att}(q_i)$, where k is the number of arguments. □

Given two permutation expressions $\mathcal{C}(P)$ and $\mathcal{C}(Q)$, we need to be able to find a $\mathcal{C}(U)$ and a $\mathcal{C}(V)$, if they exist, such that $\mathcal{C}(U)$ and $\mathcal{C}(V)$ are comma equalized and such that $[\![\mathcal{C}(U) \cap [\![\mathcal{C}(V)]\!]\!] = [\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!]$. This will be a first step towards defining intersection.

The following proposition will be a help in gradually finding two such comma equalized expressions, if they exist.

Proposition 4.10 Let $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ be permutation expressions and assume that they contain the same attribute names, i.e., $\textsc{Att}(\mathcal{C}(P)) = \textsc{Att}(\mathcal{C}(Q))$.

Assume that a $k \in I\!N$ exists such that $k < k_p,\ k < k_q$ and $\forall i \in \{1, \ldots, k\}$: $\textsc{Att}(p_i) = \textsc{Att}(q_i)$. The following holds

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

2. if $\textsc{Att}(p_{k+1}) \subset \textsc{Att}(q_{k+1})$, then either $\mathbf{PF}_{\textsc{Att}(p_{k+1})}(q_{k+1})$ is Nil, in which case $[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = \emptyset$, or $\mathbf{PF}_{\textsc{Att}(p_k+1)}(q_{k+1})$ is of the form $\mathcal{C}(T)$ in which case

$$[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = [\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(q_1, \ldots, q_k, t_1, \ldots, t_{k_t}, q_{k+2}, \ldots, q_{k_q})]\!]$$

3. if $\textsc{Att}(q_{k+1}) \subset \textsc{Att}(p_{k+1})$ then the symmetric to 2) holds.

Proof We prove the three results separately.

1. Let $s_1 \cdots s_{k_p}$ and $t_1 \cdots t_{k_q}$ be two sequences such that $\forall i : s_i \in [\![p_i]\!]$ and $\forall i :\ t_i \in [\![q_i]\!]$. For $s_1 \cdots s_{k_p}$ and $t_1 \cdots t_{k_q}$ to be identical, the $s_1 \cdots s_k$ 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 $\textsc{Att}(p_{k+1})$ and $\textsc{Att}(q_{k+1})$ would be contained in the other.

2. The form of $\mathbf{PF}_{\textsc{Att}(p_{k+1})}(q_{k+1})$ was stated in proposition 4.8. Reusing notation from the proof of 1), 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 $\mathbf{PF}_{\textsc{Att}(p_{k+1})}(q_{k+1}) = \textsc{Nil}$, or equivalently, $\mathit{Prefix}_{\textsc{Att}(p_{k+1})}([\![q_{k+1}]\!]) = \emptyset$, then no sequences from $[\![\mathcal{C}(P)]\!]$ and $[\![\mathcal{C}(Q)]\!]$ can be identical.

   From the above it follows that only sequences in $\mathit{Prefix}_{\textsc{Att}(p_{k+1})}([\![q_{k+1}]\!])$ can match sequences in $[\![\mathcal{C}(P)]\!]$. Again, by using proposition 4.7, we obtain $\mathit{Prefix}_{\textsc{Att}(p_{k+1})}([\![q_{k+1}]\!]) = \mathbf{PF}_{\textsc{Att}(p_{k+1})}(q_{k+1}) = [\![\mathcal{C}(T)]\!]$, from which the result follows by definition of $[\![\cdot]\!]$.

3. Symmetric to 2).

□

Because of the nature of proposition 4.10, it seems most natural to proceed by defining a comma equalization function using an algorithmic approach.

Also, this is the only proposition which does not translate directly into an algorithm, and we feel the need to convince the reader that an algorithm can be defined bred on this. The algorithm is, of course, based on the cases listed in proposition 4.10.

Algorithm: Comma Equalize
Input: $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ such that $\textsc{Att}(P) = \textsc{Att}(Q)$
Output: comma equalized $\mathcal{C}(U)$ and $\mathcal{C}(V)$, if they exists, such that
$[\![\mathcal{C}(P)]\!] \cap [\![\mathcal{C}(Q)]\!] = [\![\mathcal{C}(U)]\!] \cap [\![\mathcal{C}(V)]\!]$
Method:

```
let exp_p, exp_q, k be C(P), C(Q), 0
while k < k_q do
  if Att(p_{k+1}) = Att(q_{k+1}) then
    k := k + 1
  else
    if (Att(p_{k+1}) \ Att(q_{k+1}) ≠ ∅) ∧ (Att(q_{k+1}) \ Att(p_{k+1}) ≠ ∅) then
      abort "Empty intersection"
    else
      rename if necessary such that Att(p_{k+1}) ⊂ Att(q_{k+1})
      let exp be PF_{Att(p_{k+1})}(q_{k+1})
      if exp is Nil then
        abort "Empty intersection"
      else
        comment exp is of the form C(T), and exp_q is C(V)
        let exp_q be C(q_1, ..., q_k, T, q_{k+2}, ..., q_{k_q})
      endif
    endif
  endif
endwhile
output exp_p, exp_q
```

Lemma 4.11 Algorithm *Comma Equalize* solves the problem stated in its specification.

Proof By definition of permutation expressions, each q_i must contain at least one attribute name, and for $i \neq j$, we have $\textsc{Att}(q_i) \cap \textsc{Att}(q_j) = \emptyset$. This limits the length of exp_q, which is k_q. The last "else" case cannot be

chosen more than a constant number of times as the constant number of attribute names available are divided up into more q_i's each time this ease is chosen. So, after a constant 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, we assume the induction hypothesis that for all $1 \leq i \leq k : \textsc{Att}(p_i) = \textsc{Att}(q_i)$, 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 what we want. Of course, the algorithm might terminate earlier with an "Empty intersection", if justified according to proposition 4.10.

Now, if $\textsc{Att}(p_k+1) = \textsc{Att}(q_k+1)$ then we immediately obtain the induction hypothesis for $k+1$. The last "else" can only be chosen a constant number of times as already argued. And as proved in proposition 4.10, this "else" case preserves the intersection property, i.e., $[\![exp_p]\!] \cap [\![exp_p]\!] = [\![exp'_p]\!] \cap [\![exp'_q]\!]$, where exp_p and exp_q are the values before execution of the "else" case and exp'_p, and exp'_q are the values afterwards. So, eventually, we will make progress by increasing k or we will halt as we observe that $[\![exp_p]\!] \cap [\![exp_q]\!] = \emptyset$. □

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

Proposition 4.12 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 $\mathcal{C}(\tilde{P})$ and $\mathcal{C}(\tilde{Q})$ gives $\mathcal{C}(V_1)$ and $\mathcal{C}(V_2)$ then $U_1 = \tilde{V}_1$ and $U_2 = \tilde{V}_2$.

Proof First notice that if for some i and j we have that $\textsc{Att}(p_1, \ldots, p_i) = \textsc{Att}(q_1, \ldots, q_j)$, 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 $\mathcal{C}(p_{i+1}, \ldots, p_{k_i})$ and $\mathcal{C}(q_{j+1}, \ldots, q_{k_q})$, independently.

We have two cases to consider, as we are not interested in the output "Empty intersection". We proceed by induction in the number of attribute names in the involved expressions.

The case $\textsc{Att}(p_1) = \textsc{Att}(q_1)$ is easy as we can apply the induction hypothesis directly to $\mathcal{C}(p_2, \ldots, p_{k_p})$ and $\mathcal{C}(q_2, \ldots, q_{k_q})$.

Assume that $\textsc{Att}(p_1) \subset \textsc{Att}(q_1)$. We could then use $\mathbf{PF}_{\textsc{Att}(p_2)}(q_1)$, which we will assume gives $\mathcal{C}(T)$ and proceed with $\mathcal{C}(P)$ and $\mathcal{C}(T, q_2, \ldots, q_{k_q})$. By proposition 4.8, for some i, $\textsc{Att}(t_1, \ldots, t_i) = \textsc{Att}(p_1)$ So, we will get independent results for p_1 and $\mathcal{C}(t_1, \ldots, t_i)$ and for the remaining parts, $\mathcal{C}(P')$ and $\mathcal{C}(t_{i+1}, \ldots, t_{k_t}, q_2, \ldots, q_{k_q})$.

Now, look at $\mathcal{C}(\tilde{P})$ and $\mathcal{C}(\tilde{Q})$ (we use the same indices). As $\textsc{Att}(p_1) \subset \textsc{Att}(q_1)$ we must at some point use $\mathbf{PF}_{\textsc{Att}(p_2)}(q_1)$. This will create a similar situation, isolating p_1 and $t_1, \ldots, t_i$—except that here a major part of the result has already been calculated. The important fact, however, is that we now obtain the same division into independent parts. The result follows by applying the induction hypothesis. □

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

Definition 4.13 We define the intersection $\triangle$ of two permutation expressions as listed below. If none of the cases apply, then we define the result to be Nil. The intersection is only defined when the two arguments are permutation expressions over the same base set. The operation is symmetric in its two arguments, so we will only list one of each of these symmetric cases.

p	q	Condition	$p \triangle q$
A	A		A
p	$\mathcal{P}(X)$		p
$\mathcal{C}(P)$	$\mathcal{R}(Q)$		$\mathcal{C}(P) \triangle \mathbf{PF}_{\textsc{Att}(p_1)}(\mathcal{R}(Q))$
$\mathcal{R}(P)$	$\mathcal{R}(Q)$	$\exists\, T:\ \mathcal{C}(P) \triangle \mathcal{R}(Q) = \mathcal{C}(T)$	$\mathcal{R}(T)$
$\mathcal{C}(P)$	$\mathcal{C}(Q)$	$\exists\, \mathcal{C}(U), \mathcal{C}(V)$: see below	$\mathcal{C}(u_1 \triangle v_1, \ldots, u_{k_u} \triangle v_{k_v})$

In the last condition, $\mathcal{C}(U)$ 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 4.14 The intersection of permutation expressions is welldefined and implements $\cap$ correctly, i.e., $\forall p, q : [\![p \triangle q]\!] = [\![p]\!] \cap [\![q]\!]$.

Proof We argue that the process eventually terminates. It is only when the last three cases of the table is used that it does not terminate immediately. Let us consider an expression $p \triangle q$ and the value $|[\![p]\!]| + |[\![q]\!]|$. Clearly, $|[\![\mathbf{PF}_{\textsc{Att}(p_1)}(\mathcal{R}(Q))]\!]| < |[\![\mathcal{R}(Q)]\!]|$ as $\emptyset \subset \textsc{Att}(p_1) \subset \textsc{Att}(Q)$, so in two of the cases listed, this *semantic size* of the two arguments decrease. For the final

case, $\mathcal{C}(P) \triangle \mathcal{C}(Q)$ this value remains constant, but then the size of the base set decreases. We have argued that $\triangle$ is well-defined.

We shall now prove that $\triangle$ implements $\cap$ correctly. It is obvious that $p \triangle \mathcal{P}(X)$ should equal p, as $\mathcal{P}(X)$ represents all possible sequences. Recall that $\textsc{Att}(\mathcal{P}(X))$ has to equal $\textsc{Att}(p)$, so for the case $\mathcal{P}(X) \triangle \mathcal{P}(Y)$, we must have $X = Y$.

Using an induction argument, it is obvious in the light of lemma 4.11 that the last case gives rise to correct transformations. 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 base set.

The correctness of $\mathcal{C}(P) \triangle \mathcal{C}(Q) = \mathcal{C}(P) \triangle \mathbf{PF}_{\textsc{Att}(p_1)}(\mathcal{R}(Q))$ 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

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

To justify the transformation, we need to observe that

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

The first observation follows easily from the definition of $[\![\cdot]\!]$, and the second from proposition 4.12 and the definition of $\triangle$ on arguments of the form $\mathcal{C}(P)$ and $\mathcal{C}(Q)$. □

Finally, the most complicated operation on permutation sequences can be defined from what we already have: prefix and intersection. We just give the most difficult case here. There are separate cases for $\textsc{Att}(p) \subset \textsc{Att}(q)$, etc.

Proposition 4.15 Let p and q be permutation expressions and let $X = \textsc{Att}(p) \cap \textsc{Att}(q)$. Assume that $X \subset \textsc{Att}(p)$ and $X \subset \textsc{Att}(q)$. If $\mathbf{PF}_X = \mathcal{C}(P)$, $\mathbf{PF}_X(q) = \mathcal{C}(Q)$ and i and j are such that $X = \textsc{Att}(p_1, \ldots, p_i) = \textsc{Att}(q_1, \ldots, q_j)$, then

$$\begin{aligned} & [\![p]\!] \otimes [\![q]\!] \\ = \; & [\![\mathcal{C}(\mathcal{C}(p_1, \ldots, p_i) \triangle \mathcal{C}(p_1, \ldots, p_i), \mathcal{R}(\mathcal{C}(p_{i+1}, \ldots, p_{k_p}), \mathcal{C}(\mathcal{C}(q_{j+1}, \ldots, q_{k_q})))]\!] \end{aligned}$$

Proof If there exists a $t \in \mathit{Permute}(X)$ 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 $t \cdot s'_q \cdot s'_p$ belong $[\![p]\!] \otimes [\![q]\!]$. But by proposition 4.7, $[\![\mathbf{PF}_X(p)]\!]$ and $[\![\mathbf{PF}_X(q)]\!]$ are exactly the subsets of $[\![p]\!]$ and $[\![q]\!]$, respectively, for which such a t will exist. From proposition 4.14, it follows that $[\![\mathcal{C}(p_1, \ldots, p_i) \triangle \mathcal{C}(q_1, \ldots, q_i)]\!]$ is exactly the set of sequences, t, which are prefixes of both sets. The result follows frum the definitiun of $[\![\cdot]\!]$. □

We shall use $\odot$ as the operation on permutation expressions which is equivalent to $\otimes$ as it can be defined from the result above, i.e., $\forall p, q, : [\![p \odot q]\!] = [\![p]\!] \otimes [\![q]\!]$.

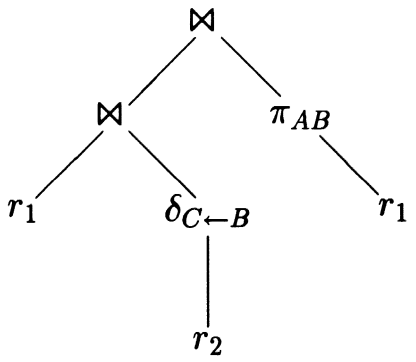
The remaining operations: concatenation, projection, and renaming are easier to define. Concatenation is defined by simply surrounding the arguments with a $\mathcal{C}$-construct, projection is defined by deleting the attribute names not contained in the projection set and then putting the expression in normal form, and finally, renaming is defined by renaming the individual attribute names in the expression. We shall use $\cdot\rfloor_X$ to denote projection of permutation expressions, and $\cdot\lfloor d \rfloor$ to denote renaming.

5 When a Solution Exists

In this section, we formulate the constraints which each relational algebra operator imposes. These will be inequalities (contained in) and we can apply the techniques of section 3 to solve the resulting inequality system. In doing so, we design an algorithm which solves the sort-merge problem and we analyze its complexity.

In this section, we will occasionally talk about the *syntax tree* of an expression. This is a standard definition and we will only illustrate it with an example.

Example 5.1 The expression $(r_1 \bowtie \delta_{C \leftarrow B}(r_2)) \bowtie \pi_{AB}(r_1)$ has the syntax tree



□

When talking about a *node*, this will also refer to the syntax tree.

5.1 Finding Solutions by Fixed Point Iteration

The inequalities presented in the following definition are based on the idea of ruling out sequences which cannot possibly be part of a sort order assignment. Consider (in definition 5.2) the inequality for π_X-up applied to $\pi_X(e)$ (the labels "up" and "down" are only intended to improve readability; they refer to the syntax-tree of the expression). Assume that the possible sequences which could be used for e has been limited to the set M_{i_1}. This can be used to limit the set of sequences, M_i, which can be used for $\pi_X(e)$. First, a sequence assigned to e has to start with the attribute names in X; otherwise we have to resort at this place during the actual evaluation. This is captured by intersecting M_{i_1} with $Permute(X) \cdot Permute(E_{i_1} \backslash X)$. Furthermore, if we consider a specific sequence and no sequence in M_{i_1} starts with this (meaning that it cannot be used in any sort order assignment), then this sequence cannot be used at M_i either.

The construction is based directly on definition 2.4 and 2.6, though this is more apparent in proposition 5.4.

Recall that we only have to consider join, projection, and renaming.

Definition 5.2 Let e be a relational algebra expression and fix an enumeration for e. In the following, we generate a finite set of inequalities (contained in) over the variables $M_1, \ldots M_{|e|}$. We refer to this system as InEq(e). For

each $i \in \{1, \ldots, |e|\}$, the inequalities generated depend on $\textsc{op}(i)$. In the following, $\textsc{Args}(i) = \{i_1, i_2\}$ and E_j is the schema of $\textsc{SubExp}(j)$.

$\bowtie$	down	$M_{i_1} \subseteq (M_i \cap \mathit{Permute}(E_{i_1}) \otimes \mathit{Permute}(E_{i_2})))|_{E_{i_1}}$
		$M_{i_2} \subseteq (M_i \cap \mathit{Permute}(E_{i_1}) \otimes \mathit{Permute}(E_{i_2})))|_{E_{i_2}}$
	up	$M_i \subseteq M_{i_1} \otimes M_{i_2}$
π_X	dowm	$M_{i_1} \subseteq M_i \cdot \mathit{Permute}(E_{i_1} \setminus X)$
	up	$M_i \subseteq (M_{i_1} \cap (\mathit{Permute}(X) \cdot \mathit{Permute}(E_{i_1} \setminus X)))|_X$
δ_d	dowm	$M_{i_1} \subseteq M_i[d^{-1}]$
	up	$M_i \subseteq M_{i_1}[d]$

In addition, for all pairs of identical relation names, $\textsc{SubExp}(i)$ and $\textsc{SubExp}(j)$, include

$$M_i \subseteq M_j \quad \text{and} \quad M_j \subseteq M_i$$

□

Of course, we can now use the technique developed in section 3 to find the largest solution to this system of inequalities.

Proposition 5.3 The system in definition 5.2 is an inequality system in the sense of suction 3.

Proof Let $E_1, \ldots, E_n$ be sets of attribute names. Then the set

$$\{(x_1, \ldots, x_n) \mid \forall i : \ x_i \subseteq \mathit{Permute}(E_i)\}$$

with the ordering pairwise inclusion, i.e.,

$$(x_1, \ldots, x_n) \sqsubseteq (x'_1, \ldots, x'_n) \Longleftrightarrow \forall i : \ x_i \subseteq x'_i$$

and pairwise intersection and union as greatest lower bound and least upper bound, respectively, forms a complete lattice.

It is an easy observation that all the functions used in definition 5.2 are monotonic. □

A simpler system than the one from definition 5.2 can by used after the first iteration (in the process of finding a fixed point). This will give a slight

constant speed-up to use this system in the algorithm we present, but our main reason for including it here is that it is more intuitive. The next result is based on definition 3.3 and definition 3.6.

Proposition 5.4 After one iteration in the equality system defined from the inequality system of definition 5.2, we can continue the iteration using a system build from the simpler inequality system listed below.

$$\begin{array}{lll}
\bowtie & \text{down} & M_{i_1} \subseteq M_i|_{E_{i_1}} \\
 & & M_{i_2} \subseteq M_i|_{E_{i_1}} \\
 & \text{up} & M_i \subseteq M_{i_1} \otimes M_{i_2} \\
\pi_X & \text{dowm} & M_{i_1} \subseteq M_i \cdot \mathit{Permute}(E_{i_1} \setminus X) \\
 & \text{up} & M_i \subseteq M_{i_1}|_X \\
\delta_d & \text{dowm} & M_{i_1} \subseteq M_i[d^{-1}] \\
 & \text{up} & M_i \subseteq M_{i_1}[d]
\end{array}$$

In additions for all pairs of identical relation names, SubExp(i) and SubExp(j), include

$$M_i \subseteq M_j \quad \text{and} \quad M_j \subseteq M_i$$

Proof Let US look at the π_X-inequalities. Because of $M_{i_1} \subseteq M_i \cdot \mathit{Permute}(E_{i_1} \setminus X)$, sequences outside $\mathit{Permute}(X) \cdot \mathit{Permute}(E_{i_1} \setminus X)$ will be removed from M_{i_1} in the first iteration. Because of monotonicity, M_{i_1} can only become smaller, so the "check" in

$$M_i \subseteq (M_{i_1} \cap (\mathit{Permute}(X) \cdot \mathit{Permute}(E_{i_1} \setminus X)))|_X$$

will not have any effect and we can use $M_i \subseteq M_{i_1}|_X$ instead.

The argument is similar for the other inequalities. □

We now present the algorithm. The algorithm is based on the inequality system just presented, except that we work on permutation expressions instead of sets of sequences, i.e., each operator in the inequality system is replaced by its permutation expression counterpart.

A solution is *trivial* in this context if some variable is assigned NIL, i.e., if $\exists M_i : \mathcal{L}(M_i) = \text{NIL}$. The method of finding the largest solution to the inequality system which characterizes a query gives us all possible sort order assignments, so we gradually select one afterwards. Right after the algorithm is presented, we give an example of how it works. Then we prove it correct and analyze its complexity.

Algorithm: Find Sort Order
Input: n expression e
Output: a sort order assignment if one exists
Method:
let $\mathcal{S}$ **be** the system INEQ(e)
let $\mathcal{L}$ **be** the largest solution to INEQ(e)
while $(\forall M_i : |[\![\mathcal{L}(M_i)]\!]| \geq 1) \wedge (\exists M_i : |[\![\mathcal{L}(M_i)]\!]| > 1)$ **do**
choose M_i such that $|[\![\mathcal{L}(M_i)]\!]| > 1$
comment $\mathcal{L}(M_i)$ contains a $\mathcal{P}(X)$ or an $\mathcal{R}$-construct
if $\mathcal{L}(M_i)$ contains $\mathcal{P}(X)$ **then**
let p **be** $\mathcal{L}(M_i)$ with $\mathcal{P}(X)$ replaced by any other expression over base 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 $\mathcal{C}(\tilde{P})$
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 INEQ($\mathcal{S}$)
endwhile
if $\mathcal{L}$ is trivial **then**
output "No solution exists."
else
comment We now have $\forall M_i : |[\![\mathcal{L}(M_i)]\!]| = 1$
let $f(i) = s$ if and only if $[\![\mathcal{L}(M_i)]\!] = \{s\}$
output "f"
endif

To keep things down to a re onable size, we have to give a rather small example.

Example 5.5 Consider the expression $r_1 \bowtie \pi_{AB}(r_2)$ where $R_1 = \{B, D\}$ and $R_2 = \{A, B, C\}$. First we need an enumeration. We choose: r_1 is 1, r_2 is 2, $\pi_{AB}(r_2)$ is 3, and $r_1 \bowtie \pi_{AB}(r_2)$ is 4. Then we list the inequalities from definition 5.2. We abbreviate *Permute* by P.

$$\begin{array}{lcl} M_2 & \subseteq & M_3 \cdot P(\{C\}) \\ M_3 & \subseteq & (M_2 \cap (P(\{A, B\}) \cdot P(\{C\})))|_{AB} \\ M_1 & \subseteq & (M_4 \cap (P(\{B, D\}) \otimes P(\{A, B\})))|_{BD} \\ M_3 & \subseteq & (M_4 \cap (P(\{B, D\}) \otimes P(\{A, B\})))|_{AB} \\ M_4 & \subseteq & M_1 \otimes M_3 \end{array}$$

where the first two inequalities are from projection and the last three from join. From this, we form a system of equations along the lines of definition 3.3.

$$\begin{array}{lcl} M_1 & = & (M_4 \cap (P(\{B, D\}) \otimes P(\{A, B\})))|_{BD} \\ M_2 & = & M_3 \cdot P(\{C\}) \\ M_3 & = & (M_2 \cap (P(\{A, B\}) \cdot P(\{C\})))|_{AB} \cap \\ & & (M_4 \cap (P(\{B, D\}) \otimes P(\{A, B\})))|_{AB} \\ M_4 & = & M_1 \otimes M_3 \end{array}$$

However, in the algorithm we use permutation expressions. If the operators in the above system are replaced by the permutation expression equivalents and then normalized (definition 4.3), then we obtain

$$\begin{array}{lcl} M_1 & = & (M_4 \triangle (\mathcal{R}(\{B, D\}) \odot \mathcal{R}(\{A, B\})))\rfloor_{BD} \\ M_2 & = & \mathcal{C}(M_3, C) \\ M_3 & = & (M_2 \triangle \mathcal{C}(\mathcal{R}(\{A, B\}), C))\rfloor_{AB} \triangle \\ & & (M_4 \triangle (\mathcal{R}(B, D) \odot \mathcal{R}(A, B)))\rfloor_{AB} \\ M_4 & = & M_1 \odot M_3 \end{array}$$

Now we find the largest fixed point for the function which takes the four variables into their right-hand sides. We iterate from the top element in the lattice which is

$$\langle \mathcal{R}(B, D), \mathcal{P}(\{A, B, C\}), \mathcal{R}(A, B), \mathcal{P}(\{A, B, D\}) \rangle$$

One iteration gives us

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

and after the next iteration we have

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

which is a fixed point. Now this does not directly provide us with a sort order assignment, though we are pretty close in this example. The only variable which is assigned a set that is not a singleton is M_4, i.e., $[\![\mathcal{L}(M_4)]\!] > 1$. As $\mathcal{L}(M_4)$ does not contain any $\mathcal{P}(X)$-constructs, the option we have in the algorithm is to replace $\mathcal{R}(A,D)$ by either $\mathcal{C}(A,D)$ or $\mathcal{C}(D,A)$. We choose to include the inequality $M_4 \subseteq \mathcal{C}(B,\mathcal{C}(D,A))$, which, in normal form, is $M_4 \subseteq \mathcal{C}(B,D,A)$. Now, we obtain a solution where all entries represent singleton sets. The sort order assignment is then $f(1) = BD$, $f(2) = BAC$, $f(3) = BA$, and $f(4) = BDA$. □

We now comment further on the algorithm. We observed in the example above that if $|[\![\mathcal{L}(M_i)]\!]| > 1$ for some M_i and the original solution $\mathcal{L}$, then this could mean that there are several sort order assignments to choose from. So, instead of simply letting the algorithm above choose a p or choose between $\mathcal{C}(P)$ or $\mathcal{C}(\tilde{P})$ in the **while**-loop, we can choose interactively or have another algorithm choose. The database designer might have a preference in the example above, for instance, to have the output stored sorted according to BAD rather than according to BDA. If so, then this information could be stored in a file or programmed into an algorithm and we could use that information when we have the freedom to choose.

We shall now prove this algorithm correct. The correctness is far from trivial and the rest of this section is devoted to the proof. First a simple definition.

Definition 5.6 Let e be a relational algebra expression. A sort order assignment f is *contained in* a solution $\mathcal{L}$ if and only if $\forall i \in \{1,\ldots,|e|\} : f(i) \in \mathcal{L}(M_i)$. □

In the algorithm, we add more and more inequalities to the initial inequality system. In the following, we prove that only sort orders which are explicitly ruled out by these inequalities will actually disappear from the maximal solution. We prove this lemma using sets instead of permutation expressions as has been justified by section 4.

Lemma 5.7 Let $\mathcal{S}$ be the system $\textsc{InEq}(e)$ with the addition of the following p inequalities:

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

where the S_j's are sets 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 sort order assignments f, the following holds:

$$(\forall j \in \{1, \ldots, p\} : f(q_j) \in S_j) \Rightarrow f \text{ is contained in } \mathcal{L}_\mathcal{S}$$

Proof Define $\mathcal{L}(M_i) = \{f(i)\}$ for all $i \in \{1, \ldots, |e|\}$. Then $\mathcal{L}$ is a solution to $\mathcal{S}$. This is an easy observation: compare definition 5.2 with the properties of a sort order assignment as defined in definition 2.7. The only inequalities that do not hold with equality are $\bowtie$-up, π_X-down, and possibly the inequalities from above.

From lemma 3.5, we know that $\mathcal{L}(M_i) \subseteq \mathcal{L}_\mathcal{S}(M_i)$. As $\mathcal{L}(M_i) = \{f(i)\}$, clearly $f(i) \in \mathcal{L}(M_i)$ from which it follows that $f(i) \in \mathcal{L}_\mathcal{S}(M_i)$. □

Corollary 5.8 Let f be a sort order assignment for e and let $\mathcal{L}$ be the largest solution to $\textsc{Eq}(\textsc{InEq}(e))$. Then $\forall i : f(i) \in \mathcal{L}(M_i)$.

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

Not surprisingly, it turns out to be important to be able to talk about two attribute names in different subexpressions being semantically identical. Considering the expression $\pi_X(r)$, it is obvious that an $A \in \textsc{Sch}(\pi_X(r)) = X$ is the same attribute as the A belonging to $\textsc{Sch}(r) = R$. However, when there are renamings in an expression or multiple occurrences of the same relation name, it is less clear which attribute names carry the same meaning. As this is crucial to the correctness proof, we feel that a precise definition is in order.

Definition 5.9 Let e be a relational algebra expression and assume that we have given an enumeration for e. First, we define the relation *immediately visible*.

Assume that $A \in \textsc{Sch}(\textsc{SubExp}(i)) \cap \textsc{Sch}(\textsc{SubExp}(j))$. If $\textsc{SubExp}(i)$ and $\textsc{SubExp}(j)$ are identical relation names, or $j \in \textsc{Args}(i)$ and $\textsc{op}(i)$ is either a join or a project, then

$$A \text{ at } i \text{ is immediately visible from } A \text{ at } j$$
$$A \text{ at } j \text{ is immediately visible from } A \text{ at } i$$

If $\textsc{op}(i)$ is $\delta_d, j \in \textsc{Args}(i)$, and $A \in \textsc{Sch}(\textsc{SubExp}(j))$, then

$$d(A) \text{ at } i \text{ is immediately visible from } A \text{ at } j$$
$$A \text{ at } j \text{ is immediately visible from } d(A) \text{ at } i$$

The relation *visible* is now defined as the reflexive and transitive closure of the relation immediately visible, i.e.,

$$A \text{ at } i \text{ is visible from } A \text{ at } i$$
$$\begin{array}{r}\text{if } C \text{ at } k \text{ is visible from } A \text{ at } i\\ \text{and } B \text{ at } j \text{ is immediately visible from } C \text{ at } k\\ \text{then } B \text{ at } j \text{ is visible from } A \text{ at } i\end{array}$$

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 $\textsc{Sch}(\textsc{SubExp}(i))$ are visible at j.

Lemma 5.10 Let e be a relational algebra expression and let $\mathcal{L}$ be the maximal solution to $\text{EQ}(\mathcal{S})$, where $\mathcal{S}$ contains $\textsc{InEq}(e)$.

1. Assume that $\mathcal{L}(M_i)$ contains $\mathcal{P}(X)$ as a subexpression. If Y at j is visible from X at i, then $\mathcal{P}(Y)$ is a subexpression of $\mathcal{L}(M_j)$.
2. Assume that $\mathcal{L}(M_i)$ contains $\mathcal{R}(U)$ as a subexpression. Let $X = \textsc{Att}(U)$. If Y at j is visible from X at i, then $\mathcal{R}(V)$, $\textsc{Att}(V) = Y$, is a subexpression of $\mathcal{L}(M_i)$. Furthermore, $k_u = k_v$.

Proof We prove each statement separately.

1. 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 $\mathcal{P}(Y)$ is a subexpression of $\mathcal{L}(M_{j'})$, where $\text{OP}(j')$ is π_Z and $\text{ARGS}(j') = \{j\}$. We want to argue that $\mathcal{P}(Y)$ is a subexpression of $\mathcal{L}(M_j)$. But according to the inequalities of proposition 5.4, $M_{j'} \subseteq M_j|_Z$ and $M_j \subseteq M_{j'} \cdot \mathit{Permute}(\text{SCH}(\text{SUBEXP})(j) \backslash Z)$. As, 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 sequence projection, $\mathcal{P}(Y)$ is a subexpression of $\mathcal{L}(M_j)$. The above also proves the reverse that if $\mathcal{P}(Y)$ is a subexpression of $\mathcal{L}(M_{j'})$ and Y is visible at j', then $\mathcal{P}(Y)$ is also a subexpression of $\mathcal{L}(M_{j'})$. The argument for join is very similar. The fact that the induction step also goes through for renaming is trivial.

2. Very similar to 1). □

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

Lemma 5.11 Let e be a relational algebra expression and let $\mathcal{L}$ be the maximal solution to $\mathcal{S}$, where $\mathcal{S}$ contains $\text{INEQ}(e)$. Assume that there exists a sort order assignment f, which is contained in $\mathcal{L}$.

1. Assume that for some i, $\mathcal{P}(X)$ is a subexpression of $\mathcal{L}(M_i)$. Let $\mathcal{S}'$ be as $\mathcal{S}$, but with the addition of $M_i \subseteq \mathcal{L}(M_i)[p/\mathcal{P}(X)]$, where p is any permutation expression over X. If $\mathcal{L}'$ is the maximal solution to $\mathcal{S}$, then there exists at least one 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 $M_i \subseteq \mathcal{L}(M_i)[\mathcal{C}(\tilde{Q})/\mathcal{R}(Q)]$. If $\mathcal{L}'$ and $\mathcal{L}''$ are the maximal solutions to $\mathcal{S}'$ and $\mathcal{S}''$, then at lest one 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 $[\![\mathcal{P}(X)]\!]$ and let t be any other sequence over X. Define the sort order assignment g by

$$g(j) = f(j)[t'_1/s'_1, \ldots, t'_k/s'_k]$$

where for all h, s'_h and t'_h at j are visible from s and t at i through the same visibility path.

We prove that g is a sort order assignment contained in $\mathcal{L}'$.

From lemma 5.10, it follows that $Y_1, \ldots, Y_k$ exist such that $\mathcal{P}(Y_h)$ is a subexpression of $\mathcal{L}(M_j)$ for all h. Therefore, each two sets have to be identical or disjoint. It also follows that $s_h, t_h \in \mathit{Permute}(Y_h)$ for all h. This means that g is well-defined in the sense that $g(j) \in \mathit{Permute}(\textsc{Sch}(\textsc{SubExp}(j)))$.

We now argue that g is a sort order assignment. If $\textsc{SubExp}(j)$ and $\textsc{SubExp}(j')$ are identical relation names, then, by definition, exactly the same sequences are visible at the two nodes, so $g(j) = g(j')$. Assume that $\textsc{op}(j)$ is π_Z and that $\textsc{Args}(j) = \{j'\}$. If some sequences s'_h and t'_h are visible at j', then $\mathcal{P}(Y_h)$ is a subexpression of $\mathcal{L}(M_{j'})$, by lemma 5.10. From definition 5.2, it is obvious that only sequences which are Z-prefixed can belong to $\mathcal{L}(M_{j'})$. Thus, either $Y_h \subseteq Z$ or $Y_h \subseteq \textsc{Sch}(\textsc{SubExp}(j'))\backslash Z$, so no substitution "crosses" this "borderline". The result now follows from the definition of visibility. The argument for join is very similar to the argument just given. Renaming is trivial.

The fact that g is contained in $\mathcal{L}'$ follows from lemma 5.7.

2. As $[\![\mathcal{R}(Q)]\!] = [\![\mathcal{C}(Q)]\!] \cup [\![\mathcal{C}(\tilde{Q})]\!]$ and as 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 $\mathcal{C}(\tilde{Q})$. 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 $\mathcal{C}(\tilde{Q})$).

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 very similar. Now we consider the proof of g being a sort order assignment. Join is the more difficult case, so assume that $\text{OP}(j)$ is a join and let $E_{j_1} = \text{SCH}(\text{SUBEXP}(j_1))$ and $E_{j_2} = \text{SCH}(\text{SUBEXP}(j_2))$, where $\text{ARGS}(j) = \{j_1, j_2\}$. If $\mathcal{R}(U)$ is a subexpression of $\mathcal{L}(M_j)$, where $\text{ATT}(U)$ at j is visible from $\text{ATT}(Q)$ at i, then there are the following cases: $\text{ATT}(U) \subseteq E_{j_1} \backslash E_{j_2}$, $\text{ATT}(U) \subseteq E_{j_2} \backslash E_{j_1}$, or $\text{ATT}(U) \subseteq E_{j_1} \cap E_{j_2}$. These cases are similar to the ones from 1).

The remaining case is $\text{ATT}(U) = (E_{j_1} \cup E_{j_2}) \backslash (E_{j_1} \cap E_{j_2})$. In any other case, the permutation expression would necessarily contain sequences not in $E_{j_1} \otimes E_{j_2}$, which is impossible; see proposition 5.4. So, we can now assume that $\mathcal{R}(U)$ is of the form $\mathcal{R}(u_1, \ldots, u_p, \ldots, u_k)$, where $\text{ATT}(u_1, \ldots, u_p) = E_{j_1} \setminus E_{j_2}$ and $\text{ATT}(u_{p+1}, \ldots, u_k) = E_{j_2} \setminus E_{j_1}$. 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 5.10. Furthermore, in this case, we must have $k = 2$. Otherwise at least two of the u_l's would also appear in $\mathcal{L}(M_{j_1})$ or $\mathcal{L}(M_{j_2})$. As they are not surrounded by an $\mathcal{R}$-construct, the order of the two u_l's would be fixed. But then the ordering of these two permutation expressions would also be fixed in $\mathcal{L}(M_{j_1}) \otimes \mathcal{L}(M_{j_2})$. As $\mathcal{L}(M_j) \subseteq \mathcal{L}(M_{j_1}) \otimes \mathcal{L}(M_{j_2})$, we get a contradiction, so $k = 2$. This means that the s' that we are changing when defining g from f is of the form $s'_1 \cdot s'_2$ and it is replaced by $s'_2 \cdot s'_1$. This means that it is correct not to make any changes in g (compared to f) in $(E_{j_1} \cup E_{j_2}) \backslash (E_{j_1} \cap E_{j_2})$.

□

Theorem 5.12 Algorithm *Find Sort Order* 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 sort order assignment.

From corollary 5.8, it follows that $\mathcal{L}$, the largest solution to $\text{EQ}(\text{INEQ}(e))$, contains every sort order assignment for e. Furthermore, lemma 5.11 proves that if the largest solution to $\mathcal{S}$ contains a 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 sort order assignment exists for e or the algorithm will find one. □

5.2 Complexity of Fixed Point Iteration

The algorithm we have developed runs very fast. We have not yet come across an example of size n which required more than n iterations to find a fixed point. In the following, we give a fairly loose, formal bound on the time-complexity. The costs of finding a fixed point are listed in observation 3.12.

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 $\mathcal{P}(X)$. 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. This k will be at most twice the size of the largest schema size of the involved relations.

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, as we can check identity of permutation expressions in time the size of the expressions. This is because the representation is unique when normal form is used. 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 by 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 iteration function, $y_i \subseteq x \subseteq x_i$. We have shown that $\langle y_1, \ldots, y_n \rangle \sqsubseteq \chi$. From lemma 3.9, 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 $O(n^2 \cdot k^2 \cdot \log^2(k))$.

As the number of attribute names in the schemas are usually fixed (and small), a more reasonable measure of the algorithm's complexity is $O(n^2)$, i.e., quadratic in the size of the query.

6 When No Solution Exists

We will briefly discuss what to do when no solution exists, i.e., when no 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) minimum number of places to do this resorting. We conjecture that this problem is NP-hard [GJ79] and we are working on a proof of this, which we hope to include in a later version of the paper.

By systematically checking all possibilities, we can use the algorithm Find

Sort Order to find a minimum 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 minimum number of places to resort is less than p, or if we are only interested in solutions, where the number of the places to resort is less than p, then the algorithm runs in polynomial time.

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

Definition 6.1 Let e be a relational algebra expression equipped with an enumeration. We modify 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 SUBEXP(i) is a relation name, then delete all of the inequalities which relates M_i to other expressions, SUBEXP(j), where the expressions SUBEXP(i) and SUBEXP(j) are identical relation names.

□

The algorithm can now be defined as follows.

Algorithm: Minimum Resort
Input: an expression e
Output: a minimum 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 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

7 Conclusion

By inventing a compact notation for certain sets of permutations, we have designed a very fast algorithm which analyzes a relational algebra query and finds a sort order assignment for it if one exists. Then we avoid resorting and we can save additional time (and temporary space) by pipelining.

When no sort order assignment exists, we want to find the (a) minimum number of places to resort. We have conjectured that this problem is NP-hard, but no proof of this exists yet. We have presented an exponential time algorithm for this problem. However, if the minimum number of places to resort is bounded by a fixed constant, then this algorithm is of polynomial time complexity.

It would be interesting to find heuristics for choosing k-tuples $(i_1, \ldots, i_k)$ in algorithm *Minimum Resort*. We believe that this could result in fast and good approximation algorithms for this problem.

References

[Des90] Bipin C. Desai. *An Introduction to Database Systems.* West Publishing Company, 1990.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[Mer83] T. H. Merrett. Why Sort-Merge Gives the Best Implementation of the Natural Join. *SIGMOD Record*, 13(2):39–51, 1983.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Comm. ACM*, 18(10):568–579, 1975.

[Tar55] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math*, 5:285–309, 1955.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.