Reihe Informatik
11 / 2003

An Efficient Framework for Order Optimization

Thomas Neumann   Guido Moerkotte

An Efficient Framework for Order Optimization

Thomas Neumann, Guido Moerkotte

July 3, 2003

Abstract

Since the introduction of cost-based query optimization, the performance-critical role of interesting orders has been recognized. Some algebraic operators change interesting orders (e.g. sort and select), while others exploit interesting orders (e.g. merge join). The two operations performed by any query optimizer during plan generation are 1) computing the resulting order given an input order and an algebraic operator and 2) determining the compatibility between a given input order and the required order a given algebraic operator can beneficially exploit. Since these two operations are called millions of times during plan generation, they are highly performance-critical. The third crucial parameter is the space requirement for annotating every plan node with its output order.

Lately, a powerful framework for reasoning about orders has been developed, which is based on functional dependencies. Within this framework, the current state-of-the-art algorithms for implementing the above operations both have a lower bound time requirement of $\Omega(n)$, where n is the number of functional dependencies involved. Further, the lower bound for the space requirement for every plan node is $\Omega(n)$.

We improve these bounds by new algorithms with upper time bounds $O(1)$. That is, our algorithms for both operations work in constant time during plan generation, after a one-time preparation step. Further, the upper bound for the space requirement for plan nodes is $O(1)$ for our approach. Besides, our algorithm reduces the search space by detecting and ignoring irrelevant orderings. Experimental results with a full fledged query optimizer show that our approach significantly reduces the total time needed for plan generation. As a corollary of our experiments, it follows that the time spent for order processing is a non-neglectable part of plan generation.

1 Introduction

In their seminal paper Selinger et al. made several important contributions to the area of query optimization—or, more specifically, its subarea of plan generation [5]. They introduced cost-based query optimization. Based on some cost model, every plan is annotated with its expected execution costs. Among all plans generated, the cheapest is then selected for execution. Another important contribution was the introduction of dynamic programming for systematic plan generation. Today, most serious plan generators use dynamic programming (e.g. [3]) or its top-down counterpart memoization (e.g. [1]). The third important contribution—which will be the focus of this paper—was the introduction

of *interesting orders*. An ordering is interesting for the query optimizer if it is generated by some algebraic operator (e.g. a (clustered) index scan or a sort operator) or if it can be beneficially exploited by some algebraic operator (like a group or merge-join operator). Last, an ordering is interesting if it is the result ordering demanded by the query. Keeping track of interesting orders can greatly reduce query execution costs since the plan generator can economize on sort operators. Let us call the subarea of plan generation that is concerned with handling interesting orders *order optimization*.

To see that there are more intricacies in order optimization than plain bookkeeping of produced and consumed orderings, consider a tuple stream that is sorted on some attribute a. After a selection with predicate $a = b$ has been applied, we can infer that the tuple stream is also ordered on the attribute sequence (a, b) and (b, a) or even on the single attribute b. We call the first ordering on (a) the *physical ordering* of the tuple stream. The inferred orderings are called *logical orderings*. As we have seen, algebraic operators (e.g. select operators) are able to change the set of logical orderings implied by a given physical ordering. Order optimization now involves not only bookkeeping of orderings, but also order inference. Simmen et al. introduced a powerful and flexible framework for order inference [6]. It is based on functional dependencies and proved to be quite powerful. Hence, we will adopt it here.

Let us take a closer look at the requirements a plan generator demands from the order optimization component. The plan generator is mainly interested in efficiently answering the following two questions:

1. Does the output of a subplan (operator) satisfy a certain ordering (demanded by some other operator the plan generator would like to add to the plan)?

2. How does the set of logical orderings change when an operator is applied?

The last question needs some explanation. In principle, it would be possible to keep track of just the physical ordering of a tuple stream and to do the inference on demand. However, this approach is quite inefficient. It implies a lot of recomputations of inferred logical orderings since subplans are highly shared. Hence, typical plan nodes (i.e. operators in a plan) materialize the set of logical orderings that can be inferred from a given physical input order (see [6]). Then, whenever an operator is applied that changes this set of logical orderings, the following happens. With every operator a set of functional dependencies is associated. With these the order optimization component can infer additional logical orderings. The inferred (new) logical orderings are added to the existing ones to give the result logical orderings for that operator. Obviously, this requires a concise representation of the set of logical orderings: since even for a medium sized query there are millions of plans and several millions of operators in the plans, this representation must be small to save storage costs. Further, the above questions must be answered very efficiently, since the according functions are called millions of times during plan generation. (For search space size estimates see the paper by Ono and Lohman [4].) We call the function that computes the answer to the first question *contains* and the one that computes the answer to the second question *inferNewLogicalOrderings*.

Simmen et al. provide algorithms for *contains* and *inferNewLogicalOrderings* but leave out some essential details that affect runtime. However, we can still calculate the lower time bound $\Omega(n)$ for both algorithms. Here, n is the total number of functional dependencies relevant for the whole query. Further, their ordering representation consumes at

least $\Omega(n)$ storage per plan node. Our contributions are implementations for both algorithms that after a perparationn step have time bounds $O(1)$ and a representation for the set of logical orderings for a plan node that consumes only $O(1)$ (4 bytes) storage. At the core of our approach is an algorithm that transforms the order inference problem into state transitions of a deterministic finite state machine (DFSM) where every state represents a set of logical orderings. Hence, plans only need to be annotated by a state. Another advantage of our transformation approach is that it prunes 'ordering' which are not relevant for the query. This reduces the search space of the plan generator.

The rest of the paper is organized as follows. In Section 2, we introduce some notations and formalize the problem of order optimization. Section 3 describes related work. This is followed by a rough sketch of our approach in Section 4. The detailed transformation algorithm is described in Section 5. Finally, examples illustrate the transformation algorithm in Section 6. Here, we also experimentally evaluate our algorithm. Section 7 then contains more experimental results comparing our algorithm to Simmen's algorithm. All experiments are performed with a query optimizer. The results show that the influence of order optimization on total plan generation time is very high. By our improved algorithm for order optimization, total plan generation time can be cut by a factor of 2 for simple chain queries with five relations and 60 larger queries containing 10 relations. Conclusions are drawn in Section 8. A proof of correctness of our transformation algorithm is given in Appendix A.

2 Problem Definition

Since the term *order optimization* is not used unambiguously, this section states what is meant by it here and what the expectations of a plan generator for this component are. For illustration purposes, we assume a bottom-up query optimizer as described e.g. by Lohman [3]. But the techniques developed here are not limited to a certain plan generator type.

The orderings that are relevant for query optimization are called *interesting orders*[1] [5]. The set of *interesting orders* for a given query consists of

1. all orderings required by an operator of the physical algebra that may be used in a query execution plan for the given query,

2. all orderings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

Note that the latter also includes the orderings demanded e.g. by the **order by** clause of a SQL query, since this ordering can potentially be produced by a `sort` operator. We treat interesting orders as *logical orderings*. This means that they specify a condition a tuple stream must meet to satisfy the given ordering. This condition is formally stated below. In contrast, the *physical ordering* of a tuple stream is the actual succession of tuples in the stream. We already saw in the introduction that the logical ordering a tuple stream satisfies can change although the physical ordering does not.

[1]The term *interesting ordering* is preferable, but since *interesting order* is a well-known term, we stick to it. However, otherwise we speak of orderings.

Some operators, like `sort`, actually influence the physical ordering of a tuple stream. Others, like `select`, only influence the logical ordering. Deduction of logical orderings can be described by using the well-known notion of *functional dependencies* (FD) [6]. In the example presented in the introduction, we used the dependency $a \rightarrow b$ which can easily be derived from $a = b$. Providing a framework for ordering inference is the great contribution of Simmen et al. [6]. In general, the influence of a given algebraic operator on a set of logical orderings can be described by a set of functional dependencies.

To exploit available logical orderings, the plan generator needs access to the order optimization component, which we describe as an *abstract data type* (ADT). An instance of this abstract data type `LogicalOrderings` represents a set of logical orderings, and wherever necessary, an instance is embedded into a plan node. Note that it must represent a set of logical orderings and not just a single one, since a tuple stream typically satisfies several logical orderings. The two main operations the abstract data type `LogicalOrderings` must provide are

1. a membership test (called `contains(LogicalOrdering)`) which tests whether the set contains the logical ordering given as a parameter and
2. an inference operation (called `inferNewLogicalOrderings(set<FD>)`). Given a set of functional dependencies, it computes a new set of logical orderings a tuple stream satisfies.

The intuitive approach is to explicitly maintain the set of all logical orderings compatible with the given physical ordering, and to update this set according to new algebraic operators applied to a tuple stream. For example, if a sort operator sorts a tuple stream by (a, b), the result is compatible with logical orderings $\{(a, b), (a)\}$. After a selection operator with selection predicate $x = const$ is applied, the set of logical orderings changes to $\{(x, a, b), (a, x, b), (a, b, x), (x, a), (a, x), (x)\}$. Since the size of the set increases quadratically with every additional selection predicate of the form $v = const$, a naive representation as a set of logical orderings is not useful in practice. This led Simmen et al. to introduce a more concise representation, which is discussed in the next section.

We now formalize the problem. Let $R = (t_1, \ldots, t_r)$ be a stream (ordered sequence) of tuples in attributes $A_1, \ldots, A_n$. Then R *satisfies the logical ordering* $o = (A_{o_1}, \ldots, A_{o_m})$ $(1 \leq o_i \leq n)$ if and only if for all $1 \leq i < j \leq r$ the following condition holds:

$$\begin{array}{lll} & (t_i.A_{o_1} \leq t_j.A_{o_1}) & \\ \wedge & \forall 1 < k \leq m & (\exists 1 \leq l < k(t_i.A_{o_l} < t_j.A_{o_l})) \vee \\ & & ((t_i.A_{o_{k-1}} = t_j.A_{o_{k-1}}) \wedge \\ & & (t_i.A_{o_k} \leq t_j.A_{o_k})) \end{array}$$

Next, we need to define the inference mechanism. Given a physical ordering $p = (A_{o_1}, \ldots, A_{o_m})$ of a tuple stream R, then R obviously satisfies any logical ordering that is a prefix of p including p itself.

Let R be a tuple stream satisfying both the logical ordering $o = (A_1, \ldots, A_n)$ and the functional dependency $f = B_1, \ldots, B_k \rightarrow B_{k+1}$[2]. Then R also satisfies any logical ordering derived from o as follows: add B_{k+1} to o at any position such that all of $B_1, \ldots, B_k$ occurred

[2]Any functional dependency which is not in this form can be normalized into a set of FDs of this form.

before this position in o. Let O' be the set of all logical orderings that can be constructed this way from o and f after prefix closure. Then, we use the following notation: $o \vdash_f O'$. Let e be the equation $A_i = A_j$. Then $o \vdash_e O'$ where O' is the prefix closure of the union of the following three sets. The first set is O_1 defined as $o \vdash_{A_i \to A_j} O_1$, the second is O_2 defined as $o \vdash_{A_j \to A_i} O_2$, and the third is the set of logical orderings derived from o where a possible occurrence of A_i is replaced by A_j or vice versa. Let e be an equation of the form $A = const$. Then O' ($o \vdash_e O'$) is derived from o by inserting A at any position in o. This is equivalent to $o \vdash_{\emptyset \to A} O'$.

Let O be a set of logical orderings and F be a set of functional dependencies (and possibly equations). We define the sets of inferred logical orderings $\Omega_i(O, F)$ as follows:

$$\begin{aligned} \Omega_0(O,F) &:= O \\ \Omega_i(O,F) &:= \Omega_{i-1}(O,F) \cup \\ &\qquad \bigcup_{f \in F, o \in \Omega_{i-1}(O,F)} o' \text{ with } o \vdash_f o' \end{aligned}$$

Let $\Omega(O, F)$ be the prefix closure of $\bigcup_{i=0}^{\infty} \Omega_i(O, F)$. We write $o \vdash_F o'$ if and only if $o' \in \Omega(O, F)$.

Assuming that the ADT represents a set of logical orderings O, `contains(`o`)` for a single logical ordering must check whether $o \in O$. And `inferNewLogicalOrderings(`F`)` must compute a representation for $O' = \Omega(O, F)$.

Last, let us see why we do not need an extra operation in case the physical ordering changes. Assume an operator changes the physical ordering of a tuple stream to p. Then the ADT must compute the set of logical orderings derivable from p, that is $\Omega(\{p\}, F)$ where F is the set of functional dependencies the tuple stream satisfies. That is, the ADT also needs an efficient constructor. However, this constructor is far less often called than the other two operations. Nonetheless, our constructor works in $O(1)$.

3 Related work

There exist only very few papers on order optimization. As already mentioned, the seminal paper by Selinger et al. [5] is the first one, introducing the problem area. They also coined the term *interesting orders*. Later papers usually concentrate on techniques to exploit, push down or combine orders, not on the abstract handling of orders during query optimization.

A more recent paper by Simmen et al. [6] introduces the already presented framework based on functional dependencies for reasoning about orderings. While we reuse their framework, our essential contribution is that we introduce more efficient algorithms.

Let us sketch their representation and algorithms. For a plan node they keep just a single (physical) ordering. Additionally, they associate all the applicable functional dependencies with a plan node. Hence, the lower bound space requirement for this representation is essentially $\Omega(n)$, where n is the number of functional dependencies derived from the query. Note that the set of functional dependencies is still (typically) much smaller than the set of all logical orderings. In order to compute the function `contains(o)`, Simmen et al. apply a *reduction algorithm* on both the ordering associated with a plan node and the ordering given as an argument to `contains`. Their reduction roughly does

the opposite of deducing more orderings using functional dependencies. Let us briefly illustrate the reduction by means of an example. Assume the physical ordering a tuple stream satisfies is (a) and the required ordering is (a, b, c). Further assume that there are two functional dependencies available: $a \rightarrow b$ and $a, b \rightarrow c$. The reduction algorithm is performed on both orderings. Since (a) is already minimal, nothing changes. Let us now reduce (a, b, c). We apply the second functional dependency first. Using $a, b \rightarrow c$, the reduction algorithm yields (a, b) because c appears in (a, b, c) after a and b. Hence, c is removed. In general, every occurrence of an attribute on the left-hand side of a functional dependency is removed if all attributes of the right-hand side of the functional dependency precede the occurrence. Reduction of (a, b) by $a \rightarrow b$ yields (a). After both orderings are reduced, the algorithm tests whether the reduced required ordering is a prefix of the reduced physical ordering. Note that if we applied $a \rightarrow b$ first, then (a, b, c) would reduce to (a, c) and no further reduction is possible. Hence, the rewrite system induced by their reduction process is not confluent. This problem is not mentioned by Simmen et al., but can have the effect that `contains` returns *false* whereas it should return *true*. The result is that some orderings remain unexploited; this could be avoided by maintaining a minimal set of functional dependencies, but the computation costs would probably be prohibitive. This problems does not occur with out approach. On the complexity side, every functional dependency has to be applied by the reduction algorithm at least once. Hence, the lower time bound is $\Omega(n)$.

In case all functional dependencies are introduced by a single plan node and all of them have to be inserted into the set of functional dependencies associated with that plan node, the lower bound for `inferNewLogicalOrderings` is also $\Omega(n)$.

Overall, Simmen et al. proposed the important framework for order optimization utilizing functional dependencies and nice algorithms to handle orderings during plan generation, but the space and time requirements are unfortunate, since plan generation might generate millions of subplans.

4 Idea

The set of logical orderings compatible with a given physical ordering produced by some subplan can be quite large. However, for plan generation the ADT `LogicalOrderings` does not need to offer access to the set of orderings: it only allows to test if a given interesting order is in the set and changes the set according to new functional dependencies. Hence, it is *not* required to explicitly represent this set; an implicit representation is sufficient as long as the ADT operations can be implemented atop of it. In other words, we need not be able to reconstruct the set of logical orderings from the state of the ADT. This gives us room for optimizations.

The idea of our approach is to represent sets of logical orderings as *states* of a *finite state machine* (FSM). Roughly, a state of the FSM represents a current physical ordering and the set of logical orderings that can be inferred from it given a set of functional dependencies. The edges (transitions) in the FSM are labeled by sets of functional dependencies. They lead from one state to another, if the target state of the edge represents the set of logical orderings that can be derived from the orderings the edge's source node represents by applying the set of functional dependencies the edge is labeled with. We have to use sets of functional dependencies, since a single algebraic operator may introduce

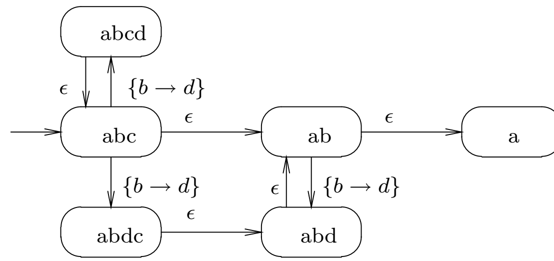

Figure 1: Possible FSM for orderings

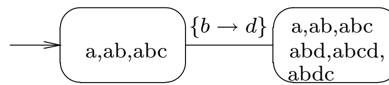

Figure 2: Possible DFSM for the NFSM in Figure 1

more than one functional dependency.

Let us illustrate the idea by a simple example and then discuss some problems. In Figure 1 an FSM for the interesting order (a, b, c) and its prefixes (remember that we need prefix closure) and the set of functional dependencies $\{b \to d\}$ is given. When a physical ordering satisfies (a, b, c), it also satisfies its prefixes (a, b) and (a). This is indicated by the ϵ transitions. The functional dependency $b \to d$ allows to derive the *logical* orderings (a, b, c, d) and (a, b, d, c). This is handled by assuming that the *physical* ordering changes to either (a, b, c, d) or (a, b, d, c). Hence, these states have to be added to the FSM. We further add the transitions induced by $\{b \to d\}$. Note that the resulting FSM is a *non-deterministic finite state machine* (NFSM).

Assume we have an NFSM as above. Then the state of the ADT is a state of the NFSM and the operations of the ADT can easily be mapped to the FSM. Testing for a logical ordering can be performed by checking if the node with the ordering is reachable from the current state by following ϵ edges. If the set must be changed because of a functional dependency the state is changed by following the edge labeled with the functional dependency. Of course, the non-determinism is in our way.

While remembering only the active state of the FSM avoids the problem of maintaining a set of orderings, the FSM is not really useful from a practical point of view, since the transitions are non-deterministic. Nevertheless, the FSM can be considered as a special *non-deterministic finite automaton* (NFA), which consumes the functional dependencies and "recognizes" the possible physical orderings. Further, an NFA can be converted into a *deterministic finite automaton* (DFA), which can be handled efficiently. Remember that the construction is based on the power set of the NFA's states. That is, the states of the DFA are sets of states of the NFA [2]. We do not take the deviation over the finite automaton but instead lift the construction of deterministic finite automatons from nondeterministic ones to finite state machines. Since this is not a traditional conversion, we give a proof of this step in the appendix.

Yet another problem is that the conversion from an NFSM to a *deterministic FSM* (DFSM) can be expensive for large NFSMs. Therefore, reducing the size of the NFSM is another problem we look at. We introduce techniques for reducing the set of functional dependencies that have to be considered and further techniques to prune the NFSM.

Another problem occurs. On the one hand, the conversion from a nondeterministic to

1. Determine the input
   (a)Determine interesting orders
   (b)Determine sets of functional dependencies
2. Construct the NFSM
   (a)Construct the nodes of the NFSM
   (b)Filter functional dependencies
   (c)Add edges to the NFSM
   (d)Prune the NFSM
   (e)Add artificial start node and edges
3. Convert the NFSM into a DFSM
4. Precompute values
   (a)Precompute the compatibility matrix
   (b)Precompute the transition table

Figure 3: Preparation steps of the algorithm

a deterministic FSM aggressively merges nodes, and the deterministic FSM is typically much smaller than the non-deterministic. For example, the DFSM in Figure 2 is the counterpart of the NFSM in Figure 1. On the other hand, we need to keep track of those orderings that can be produced by an algebraic operator like `sort`. This is necessary for an efficient implementation of the ADT constructor. However, this information may get lost during the NFSM to DFSM conversion. We introduce special measures to avoid this problem.

Some operators, like *sort*, change the physical ordering. In the NFSM, this is handled by changing the state to the node corresponding to the new physical ordering. Implied by its construction, in the DFSM this new physical ordering typically occurs in several nodes. For example, (a, b, c) occurs in both nodes of the DFSM in Figure 2. It is, therefore, not obvious which node to choose. We will take care of this problem during the construction of the NFSM (see Section 5.3).

The DFSM for real queries (e.g. those in TPC-R) is much smaller than the NFSM. The reason is that most nodes in the NFSM are *artificial nodes* that can be merged aggressively (see Section 5.3). However, theoretically a DFSM can be exponential in the size of the NFSM it is constructed from. We briefly come back to this problem in Section 8.

5 Detailed Algorithm

5.1 Overview

Our approach consists of two phases. The first phase is the preparation step taking place before the actual plan generation starts. The output of this phase is then used to implement the ADT. Then the ADT is used during the second phase where the actual plan generation takes place. The first phase is performed exactly once and quite involved. Most of this section covers the first phase. Only Section 5.6 deals with the ADT implementation.

Figure 3 gives an overview of the preparation phase. It is divided into four major steps. The different steps are discussed in the following subsections. Subsection 5.2 briefly

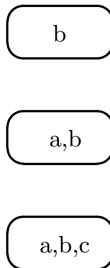

Figure 4: Initial NFSM for sample query

reviews how the input to the first phase is determined and, more importantly, what it looks like. Section 5.3 describes in detail the construction of the NFSM from the input. The conversion from the NFSM to the DFSM is only briefly sketched in Section 5.4, for details see [2]. From the DFSM some values are precomputed which are then used for the efficient implementation of the ADT. The precomputation is described in Section 5.5, while their usage and the ADT implementation are the topic of Section 5.6. Section 5.7 contains some important techniques to reduce the size of the NFSM. They are applied in Steps 2 (b) and 2 (d). During the discussion, we illustrate the different steps by a simple running example. More complex examples can be found in Section 6.

5.2 Determining the Input

Since the preparation step is performed immediately before plan generation, it is assumed that the query optimizer already has determined which indices are applicable and which algebraic operators can possibly be used to construct the query execution plan.

Before constructing the NFSM, the set of interesting orders and the sets of functional dependencies for each algebraic operator are determined. We denote the set of sets of functional dependencies by $\mathcal{F}$. It is important for our algorithms to work that we note which of the interesting orders are (1) produced by some algebraic operator or (2) just tested for. Note that the interesting orders which satisfy (1) may additionally be tested for as well. We denote those orderings under (1) by O_P, those under (2) by O_T. The total set of interesting orders is defined as $O_I = O_P \cup O_T$. The orders produced are treated slightly differently in the following steps. For details on this step we refer to [5, 6].

To illustrate subsequent steps, we assume that the set of sets of functional dependencies

$$\mathcal{F} = \{\{b \rightarrow c\}, \{b \rightarrow d\}\}$$

and the interesting orders

$$O_I = \{(b), (a, b)\} \cup \{(a, b, c)\}$$

have been extracted from the query. We assume that those in $O_T = \{(a, b, c)\}$ are tested for but not produced by any operator, whereas those in $O_P = \{(b), (a, b)\}$ may be produced by some algebraic operators.

5.3 Constructing the NFSM

A regular NFSM consists of a tuple (Σ, Q, D, q_o), where

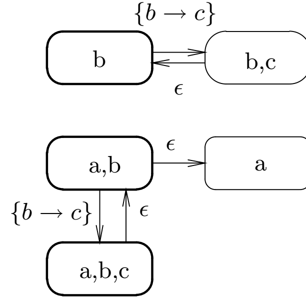

Figure 5: NFSM after adding edges

- Σ is the input alphabet,
- Q the set of possible states,
- $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation and
- q_0 the initial state.

This definition has to be changed slightly for the algorithm. As we will see later, it is important to know which states represent interesting orders (Q_I) and which are artificial states that are only required for the algorithm itself (Q_A). Additionally, we have an artificial start node q_0. Summarizing, we define $Q = Q_I \cup Q_A \cup \{q_0\}$ with $Q_I \cap Q_A = \emptyset$ and $q_0 \notin Q_I \cup Q_A$.

The interesting orders have been partitioned into those orderings which can be directly produced by an operator (O_P) and those which are only tested for (O_T). For each interesting order, we introduce a state representing that order. The resulting state sets are called Q_I^P and Q_I^T, respectively. Then $Q_I = Q_I^P \cup Q_I^T$ where $Q_I^P \cap Q_I^T = \emptyset$. The FSM constructed for the example so far is shown in Figure 4.

The artificial nodes are constructed by considering functional dependencies

$$Q_A = (\Omega(O_I, \mathcal{F}) \setminus O_I).$$

The input alphabet Σ consists of the sets of functional dependencies and the labels of the artificial edges which link to the states Q_I^P that represent the (produced) interesting orders O_P:

$$\Sigma = \mathcal{F} \cup Q_I^P.$$

We will use artificial edges as entry points to allow a fast constructor implementation for the ADT. In order to support artificial edges, we define the domain of the transition relation D as

$$\begin{aligned} D \subseteq \quad & ((Q \setminus \{q_0\}) \times (\mathcal{F} \cup \{\epsilon\}) \times (Q \setminus \{q_0\})) \\ \cup \quad & (\{q_o\} \times Q_I^P \times Q_I^P) \end{aligned}$$

The edges are formed by the functional dependencies and the artificial edges:

$$\begin{aligned} D_{FD} &= \{(o, f, o') \mid o \in Q, f \in \mathcal{F} \cup \{\epsilon\}, o' \in Q, o \vdash_f o'\} \\ D_A &= \{(q_0, o, o) \mid o \in Q_I^P\} \\ D &= D_{FD} \cup D_A \end{aligned}$$

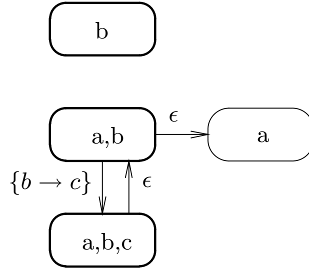

Figure 6: NFSM after pruning artificial nodes

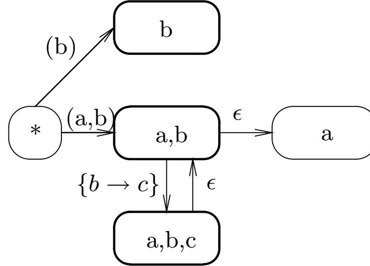

Figure 7: Final NFSM

First, edges corresponding to functional dependencies are added. In our example, this results in the NFSM shown in Figure 5. Note that the functional dependency $b \to d$ has been pruned, since d does not occur in any interesting order. The NFSM can be further simplified by pruning the artificial node (b, c) which cannot lead to a new interesting order. The result is shown in Figure 6. A detailed description of both pruning techniques can be found in Section 5.7.

The artificial start node q_0 has emanating edges incident to all nodes representing interesting orders in O_I^P. The final NFSM for the example is shown in Figure 7. Note that the node representing (a, b, c) is not linked by an artificial edge since it is only tested for, as it is in Q_I^T.

5.4 Constructing the DFSM

The construction of the DFSM from the NFSM follows the standard power set construction that is used to translate an NFA into a DFA [2]. A formal description and a proof of correctness is given in the appendix. It is important to note that this construction

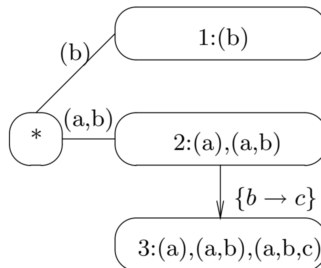

Figure 8: Resulting DFSM

node	(a)	(a,b)	(a,b,c)	(b)
1	0	0	0	1
2	1	1	0	0
3	1	1	1	0

Figure 9: Precomputed values for `contains`

node	$\{b \rightarrow c\}$	(a, b)	(b)
*	*	2	1
1	1	1	1
2	3	2	2
3	3	3	3

Figure 10: Precomputed values for `constructor` and `inferNewLogicalOrderings`

preserves the start node and the artificial edges. The resulting DFSM for the example is shown in Figure 8.

5.5 Precomputing values

To allow for an efficient precomputation of values, every occurrence of an interesting order or functional dependency is replaced by a handle. This allows comparisons in constant time (equivalent entries get the same handle). Further, the DFSM is represented by an adjacency matrix.

The precomputation step itself computes two matrices. The first matrix denotes whether an NFSM node in Q_I, i.e. an interesting order, is contained in a specific DFSM node. This matrix can be represented as a compact bit vector. For our running example, it is given (in a more readable form) in Figure 9. The second matrix contains the transition table for the DFSM relation Δ. Using it, edges in the DFSM can be followed in $O(1)$. For this example, the transition matrix is given in Figure 10.

5.6 During Plan Generation

During plan generation, larger plans are constructed by adding algebraic operators to existing (sub-)plans. Each subplan contains the available orderings in the form of the corresponding state in the DFSM. Hence, the state of the DFSM, a simple integer, is the state of our ADT `LogicalOrderings`.

When applying an operator to subplans the ordering requirements are tested by checking whether the DFSM state of the subplan contains the required ordering of the operator. This is done by a simple lookup in the first precomputed matrix.

If the operator introduces a new set of functional dependencies, the new state of the ADT is computed by following the according edge in the DFSM. This is performed by a quick lookup in the second precomputed matrix.

For "atomic" subplans like table or index scans the ordering is determined explicitly by the operator (either an empty ordering or the ordering resulting from the operator).

The state of the DFSM is determined by a lookup in the transition matrix with start state $*$ and the edge annotated by the produced ordering. For sort operators the state of the DFSM is determined by following the edge as before and then another edge corresponding to the set of functional dependencies that currently hold.

In the example, a sort by (a, b) results in a subplan with ordering 2 (the node 2 is active in the DFSM), which satisfies (a) and (a, b). After applying an operator which induces $b \to c$, the ordering changes to 3, which also satisfies (a, b, c). The transitions and tests can easily be computed using the tables in Figure 9 and Figure 10.

5.7 Reducing the Size of the NFSM

Reducing the size of the NFSM is important for two reasons: First, it reduces the amount of work needed during the preparation step, especially the conversion from NFSM to DFSM. Even more important is that a reduced NFSM results in a smaller DFSM. This is crucial for plan generation, since it reduces the search space: Plans can only be compared and pruned if they have comparable ordering and a comparable set of functional dependencies (see [6] for details). Reducing the size of the DFSM removes information that is not relevant for plan generation and, therefore, allows a more aggressive pruning of plans.

At first, the functional dependencies are pruned. These functional dependencies that can never lead to a new interesting order are removed. For convenience, we extend the definition of $\Omega(O, F)$ and define

$$\Omega(O, \epsilon) \; := \; \Omega(O, \emptyset)$$

Then the set of prunable functional dependencies $\mathcal{F}_P$ can be described by

$$\begin{aligned} \Omega_N(o, f) &:= \Omega(\{o\}, \{f\}) \setminus \Omega(\{o\}, \epsilon) \\ \mathcal{F}_P &:= \{f | f \in \mathcal{F} \wedge \forall o \in O_I : \\ &\quad\; (\Omega(\Omega_N(o, f), \mathcal{F}) \setminus \Omega(\{o\}, \epsilon)) \cap O_I = \emptyset\} \end{aligned}$$

Pruning functional dependencies is especially useful since it also prunes artificial nodes that would be created because of the dependencies. In the example, this removed the functional dependency $b \to d$, since d does not appear in any interesting order. This step also removes the artificial nodes containing d.

The artificial nodes are required to build the NFSM, but they are not visible outside the NFSM, therefore they can be pruned and merged without affecting plan generation. Two heuristics are used to reduce the set of artificial nodes:

1. All artificial nodes that behave exactly the same (that is their edges lead to the same nodes given the same input) are merged and

2. all artificial nodes that can reach important nodes only through ϵ edges are pruned and replaced with corresponding links to the important nodes.

More formally, the following pairs of nodes can be merged:

$$\begin{aligned} \{(o_1, o_2) \;|\;\; & o_1 \in O_A, o_2 \in O_A \wedge \forall f \in \mathcal{F} : \\ & (\Omega(\{o_1\}, \{f\}) \setminus \Omega(\{o_1\}, \epsilon)) = \\ & (\Omega(\{o_2\}, \{f\}) \setminus \Omega(\{o_2\}, \epsilon))\} \end{aligned}$$

The following nodes can be replaced with the next node reachable by an ϵ edge:

$$\{o \mid o \in O_A \land \forall f \in \mathcal{F}: \Omega(\Omega(\{o\},\epsilon),\{f\}) \setminus \{o\} = \Omega(\Omega(\{o\},\epsilon) \setminus \{o\},\{f\})\}$$

In the example, this removed the node (b, c), which was artificial and only led to the node (b).

These techniques reduce the size of the NFSM, but still most nodes are artificial nodes, i.e. they are only created because they can be reached by considering functional dependencies when a certain ordering is available. But many of these nodes are not relevant for the actual query processing. For example, given a set of interesting orders which consists only of a single ordering (a) and a set of functional dependencies which consists only of $a \rightarrow b$, the NFSM will contain (among others) two nodes: (a) and (a, b). The node (a, b) is created since it can be reached from (a) by considering the functional dependency, however, it is irrelevant for the plan generation, since (a, b) is not an interesting order and is never created nor tested for. Actually, in the example above, the whole functional dependency would be pruned (since b never occurs in an interesting order), but the problem remains for combinations of interesting orders: Given the interesting orders (a), (b) and (c) and the functional dependencies $\{a \rightarrow b, b \rightarrow a, b \rightarrow c, c \rightarrow b\}$, the NFSM will contain nodes for all permutations of a, b and c. But these nodes are completely useless, since all interesting orders consist only of a single attribute and, therefore, only the first entry of an ordering is ever tested.

Ideally, the NFSM should only contain nodes which are relevant for the query; since this is difficult to ensure, a heuristic can be used which greatly reduces the size of the NFSM and still guarantees that all relevant nodes are available: When considering a functional dependency of the form $a \rightarrow b$ and an ordering $o_1, o_2, \ldots, o_n$ with $o_i = a$ for some i $(1 \leq i \leq n)$, the b can be inserted at any position j with $i < j \leq n + 1$ (for the special case of a condition $a = b$ $i = j$ is also possible). So, an entry of an ordering can only affect entries on the right of its own position. This means that it is unnecessary to consider those parts of an ordering which are behind the length of the longest interesting order; since that part cannot influence any entries relevant for plan generation, it can be omitted. Therefore, the orderings created by functional dependencies can be cut off after the maximum length of interesting orders, which results in less possible combinations and a smaller NFSM.

The space of possible orderings can be limited further by taking into account the prefix of the ordering: before inserting an entry b in an ordering $o_1, o_2, \ldots, o_n$ at the position i, check if there is actually an interesting order with the prefix $o_1, o_2, ...o_{i-1}, b$ and stop inserting if not interesting order is found. Also limit the new ordering to the length of the longest matching interesting order; further attributes will never be used. If functional dependencies of the form $a = b$ occur, they might influence the prefix of the ordering and the simple test described above is not enough. Therefore a representative is chosen for each equivalence class created by these dependencies and for the prefix test the attributes are replaced with their representatives. Since the set of interesting orders with a prefix of $o_1, \ldots, o_n$ is a superset of the set for the prefix $o_1, ...o_n, o_{n+1}$ this heuristic can be implemented very efficiently by iterating over i and reducing the set as needed.

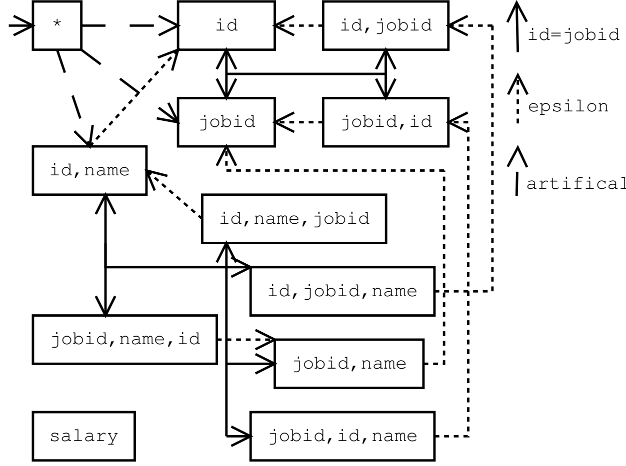

Figure 11: NFSM for a simple query

6 Examples

This section contains two example queries. The first is used to illustrate the transformation algorithm detailed in the previous section. We give the complete NFSM and DFSM. The second query is a more complex query taken from TPC-R. It is used to illustrate the runtime behavior of the transformation algorithm and the gains of the introduced heuristics on a realistic query.

6.1 Simple Query

Consider the following query:

```
select *
from persons,jobs
where persons.jobid=jobs.id and
      jobs.salary>50000
order by jobs.id, persons.name
```

The interesting orders are

$$\begin{aligned} Q_I^P &= \{(id), (jobid), (id, name)\} \\ Q_I^T &= \{(salary)\} \end{aligned}$$

The set of functional dependencies consists of

$$\mathcal{F} = \{\{jobid \rightarrow id, id \rightarrow jobid\}\}.$$

Figure 11 shows the NFSM for the query. Note that the NFSM takes into account that $a = b$ is stronger than $\{a \rightarrow b, b \rightarrow a\}$, e.g. the edge $(id) \stackrel{id=jobid}{\rightarrow} (jobid)$ is added, which is not possible if only the functional dependencies are considered. Also note that the state for salary cannot be reached; since the ordering $(salary)$ is only in Q_I^T (no operator will generate $(salary)$), no edge from the start node is added. In the DFSM, the node does

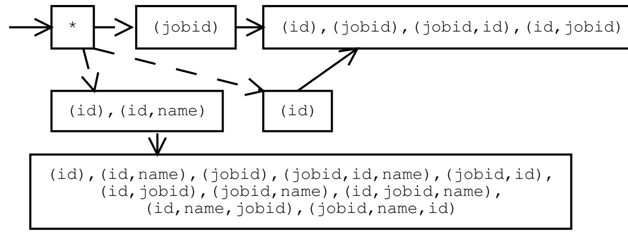

Figure 12: DFSM for NFSM in Figure 11

not appear. Here, we assumed that sorting for a selection does not pay off and that there is no index on salary.

The corresponding DFSM is shown in Figure 12. The DFSM is smaller than the NFSM since the different permutations of id, jobid and name are merged; when the corresponding DFSM node is active, all of them are available.

6.2 TPC-R Query 8

To illustrate the time and space requirements of the algorithm for larger input sizes, we chose a more complex query from the well-known TPC-R benchmark ([7]):

```
select
   o_year,
   sum(case when nation = '[NATION]'
      then volume
      else 0
   end) / sum(volume) as mkt_share
from
   (select
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1-l_discount) as volume,
      n2.n_name as nation
   from part,supplier,lineitem,orders,customer,
         nation n1,nation n2,region
   where
      p_partkey = l_partkey and
      s_suppkey = l_suppkey and
      l_orderkey = o_orderkey and
      o_custkey = c_custkey and
      c_nationkey = n1.n_nationkey and
      n1.n_regionkey = r_regionkey and
      r_name = '[REGION]' and
      s_nationkey = n2.n_nationkey and
      o_orderdate between date '1995-01-01' and
         date '1996-12-31' and
      p_type = '[TYPE]'
   ) as all_nations
   group by o_year
```

order by o_year;

When considering this query, all attributes used in joins and group by clauses are added to the set of interesting orders. This results in the sets

$$
\begin{aligned}
O_I^P \;&=\; \{(o_year), (o_partkey), (p_partkey),\\
&\quad (l_partkey), (l_suppkey), (l_orderkey),\\
&\quad (o_orderkey), (o_custkey), (c_custkey),\\
&\quad (c_nationkey), (n1.n_nationkey),\\
&\quad (n2.n_nationkey), (n_regionkey),\\
&\quad (r_regionkey), (s_suppkey), (s_nationkey)\}\\
Q_I^T \;&=\; \emptyset
\end{aligned}
$$

If appropriate operators of the physical algebra are available, it might be useful to add $O_I^T = \{(r_name), (o_orderdate)\}$; a selection operator never sorts but might exploit ordering.

The set of functional dependencies (and equations) contains all join conditions and constant conditions:

$$
\begin{aligned}
\mathcal{F} \;&=\; \{\{p_partkey = l_partkey\}, \{\emptyset \rightarrow p_type\},\\
&\quad \{o_custkey = c_custkey\}, \{\emptyset \rightarrow r_name\},\\
&\quad \{c_nationkey = n_nationkey\},\\
&\quad \{s_nationkey = n2.n_nationkey\},\\
&\quad \{l_orderkey = o_orderkey\},\\
&\quad \{s_suppkey = l_suppkey\},\\
&\quad \{n_regionkey = r_regionkey\}\}
\end{aligned}
$$

On an AMD Athlon XP 1800+ using the gcc 3.2, the complete preparation step has the following space and time requirements, the results with and without pruning techniques (see section 5.7) are shown.

	w/o pruning	with pruning
NFSM size	376 nodes	38 nodes
DFSM size	80 nodes	24 nodes
total time	16ms	0.2ms
precomputed data	3040 bytes	912 bytes

7 Experimental Results

We now consider how order processing influences the time needed for plan generation. Therefore, we implemented both our algorithm and the algorithm proposed by Simmen et al. [6] and integrated them into a bottom-up plan generator based on [3].

To get a fair comparison, we tuned Simmen's algorithm as much as possible. The most important measure was to cache results in order to eliminate repeated calls to the very expensive *reduce* operation. Second, since Simmen's algorithm requires dynamic memory, we implemented a specially tailored memory management. This alone gave us a speed

n	#Edges	t (ms)	#Plans	t/plan	t (ms)	#Plans	t/plan	% t	% #Plans	%. t/plan
5	n-1	2	1541	1.29	1	1274	0.78	2.00	1.21	1.65
6	n-1	9	7692	1.17	2	5994	0.33	4.50	1.28	3.55
7	n-1	45	36195	1.24	12	26980	0.44	3.75	1.34	2.82
8	n-1	289	164192	1.76	74	116562	0.63	3.91	1.41	2.79
9	n-1	1741	734092	2.37	390	493594	0.79	4.46	1.49	3.00
10	n-1	11920	3284381	3.62	1984	2071035	0.95	6.01	1.59	3.81
5	n	4	3060	1.30	1	2051	0.48	4.00	1.49	2.71
6	n	21	14733	1.42	4	9213	0.43	5.25	1.60	3.30
7	n	98	64686	1.51	20	39734	0.50	4.90	1.63	3.02
8	n	583	272101	2.14	95	149451	0.63	6.14	1.82	3.40
9	n	4132	1204958	3.42	504	666087	0.75	8.20	1.81	4.56
10	n	26764	4928984	5.42	2024	2465646	0.82	13.22	2.00	6.61
5	n+1	12	5974	2.00	1	3016	0.33	12.00	1.98	6.06
6	n+1	69	26819	2.57	6	12759	0.47	11.50	2.10	5.47
7	n+1	370	119358	3.09	28	54121	0.51	13.21	2.21	6.06
8	n+1	2613	509895	5.12	145	208351	0.69	18.02	2.45	7.42
9	n+1	27765	2097842	13.23	631	827910	0.76	44.00	2.53	17.41
10	n+1	202832	7779662	26.07	3021	3400945	0.88	67.14	2.29	29.62

Figure 13: Plan generation for different join graphs, Simmen's algorithm (left) vs. our algorithm (middle)

up by a factor of three. We further tuned the algorithm by thoroughly profiling it until no more improvements were possible. For each order optimization framework the plan generator was recompiled to allow for as many compiler optimizations as possible. We also carefully observed that in all cases both order optimization algorithms produced the same optimal plan.

We first measured the plan generation times and memory usage for TPC-R Query 8. The result of this experiment is summarized in the following table. Since order optimization is tightly integrated with plan generation, it is impossible to exactly measure the time spend just for order optimization during plan generation. Hence, we decided to measure the impact of order optimization on the total plan generation time. This has the advantage that we can also (for the first time) measure the impact order optimization has on plan generation time. This is important since one could argue that we are optimizing a problem with no significant impact on plan generation time, hence solving a non-problem. As we will see, this is definitely not the case.

In subsequent tables, we denote by *t(ms)* the total execution time for plan generation measured in milliseconds, by *#Plans* the total number of subplans generated, by *t/plan* the time (in microseconds) needed to introduce one plan operator, i.e. the time to produce a single subplan, and by *Memory* the total memory (in KB) consumed by the order optimization algorithms.

	Simmen	Our algorithm
t (ms)	262	52
#Plans	200536	123954
t/plan (μs)	1.31	0.42
Memory (KB)	329	136

From these numbers, it becomes obvious that order optimization has a significant influence on total plan generation time. It may come as a surprise that fewer plans need to be generated by our approach. This is due to the fact the (reduced) FSM only contains the information relevant to the query, resulting in fewer states. With Simmen's approach, the plan generator can only discard plans if the ordering is the same and the set of functional dependencies is equal (respectively a subset). It does not recognize that the additional information is not relevant for the query.

In order to show the influence of the query on the possible gains of our algorithm, we generated queries with 5-10 relations and a varying number of join predicates —that is, edges in the join graph. We always started from a chain query and then randomly added some edges. For small queries we averaged the results of 100 queries and averaged 10 queries for large queries. The results of the experiment can be found in Fig. 13. In the second column, we denote the number of edges in terms of the number of relations (n) given in the first column. The next six columns contain (1) the total time needed for plan generation (in ms), (2) the number of (sub-) plans generated, and (3) the time needed to generate a subplan (in μs), i.e. to add a single plan operator, for (a) Simmen's algorithm (columns 3-5) and our algorithm (columns 6-8). The total plan generation time includes building the DFSM when our algorithm is used. The last three columns contain the improvement factors for these three measures achieved by our algorithm. More specifically, column *% x* contains the result of dividing the x column of Simmen's algorithm by the corresponding x column entry of our algorithm.

Note that we are able to keep the plan generation time below one second in most cases and three seconds in the worst case whereas when Simmen's algorithm is applied, plan generation time can be as high as 200 seconds. A lot more could be said about these numbers, but for space reasons we restrict ourselves to draw two important conclusions:

1. Order optimization has a significant impact on total plan generation time.

2. By using our algorithm, significant performance gains are possible.

For completeness, we also give the memory consumption during plan generation for the two order optimization algorithms (see Fig. 14). For our approach, we also give the sizes of the DFSM which are included in the total memory consumption. All memory sizes are in KB. As one can see, our approach consumes about half as much memory as Simmen's algorithm.

8 Conclusion

The framework presented in this paper allows a very efficient handling of order optimization during plan generation. After a preparation step with reasonable performance, the plan generation can change and test for orderings in O(1) using only O(1) space per subplan. Experimental results have shown that this can significantly reduce the time needed for plan generation by both reducing the time needed per subplan and reducing the search space, which is essential for handling large queries. Summarizing, using FSMs to keep track of the available orderings allows a very efficient handling of orderings during plan generation.

References

[1] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

n	#Edges	Simmen	Our Algorithm	DFSM
5	n-1	14	10	2
6	n-1	44	28	2
7	n-1	123	77	2
8	n-1	383	241	3
9	n-1	1092	668	3
10	n-1	3307	1972	4
5	n+0	27	12	2
6	n+0	68	36	2
7	n+0	238	98	3
8	n+0	688	317	3
9	n+0	1854	855	4
10	n+0	5294	2266	4
5	n+1	53	15	2
6	n+1	146	49	3
7	n+1	404	118	3
8	n+1	1247	346	4
9	n+1	2641	1051	4
10	n+1	8736	3003	5

Figure 14: Memory consumption for Figure 13, Simmen's algorithm (left) vs. our algorithm (right)

[2] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[3] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 18–27. ACM Press, 1988.

[4] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[5] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[6] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 57–67. ACM Press, 1996.

[7] Transaction Processing Performance Council, 777 N. First Street, Suite 600, San Jose, CA, USA. *TPC Benchmark R*, 1999. Revision 1.2.0. http://www.tpc.org.

A Proof of correctness

The algorithm described in this paper first constructs a non-deterministic FSM and converts it to a deterministic FSM. For this conversion, the NFSM is treated like an NFA which is converted to a DFA. It has to be shown that the DFSM resulting from the conversion is equivalent to the initial NFSM:

A.1 Definitions

An NFA [2] consists of a tuple (Σ, Q, D, q_o, F), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation, q_0 the initial state and F the set of accepting states. All nodes reachable from a given set of nodes Q by following ϵ edges can be described by

$$\begin{aligned} \mathcal{E}_D^0(Q) &= Q \\ \mathcal{E}_D^i(Q) &= \{q' | \exists q \in \mathcal{E}_D^{i-1}(Q), (q, \epsilon, q') \in D\} \\ \mathcal{E}_D(Q) &= \bigcup_{i=0}^{\infty} \mathcal{E}_D^i(Q) \end{aligned}$$

Then the NFA *accepts* an input $w = w_1 w_2 ... w_n \in \Sigma^*$ if $S_n \cap F \neq \emptyset$ where

$$\begin{aligned} S_0 &= \mathcal{E}_D(q_o) \\ S_i &= \mathcal{E}_D(\{q' | \exists q \in S_{i-1} : (q, w_i, q') \in D\}). \end{aligned}$$

Similarly, a DFA [2] consists of a tuple $(\Sigma, Q, \Delta, q_o, F)$ where

$$\begin{aligned} & \Delta \subseteq Q \times \Sigma \times Q \\ \wedge \quad & \forall a, b, c \in Q, d \in \Sigma : \\ & ((a, d, b) \in \Delta \wedge (a, d, c) \in \Delta) \Rightarrow b = c. \end{aligned}$$

So a DFA is an NFA which only allows non-ambiguous non-ϵ transitions. The definition of accepting is analogous to the definition for NFAs.

An NFSM is basically an NFA without accepting states. It consists of a tuple (Σ, Q, D, q_o), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation and q_0 the initial state. While an NFSM does not have aby accepting states it is usually important to know which state is active after a given input, so in a way each state is accepting.

Likewise, a DFSM basically is a DFA without accepting states. It consists of a tuple (Σ, Q, Δ, q_o) where Σ, Q,Δ and q_o are analogous to the DFA. Again, while there is no set of accepting states, it is important to know which one is active after a given input.

A.2 The transformation algorithm

The commonly used algorithm to convert an NFA into a DFA (see [2]) can also be used to convert an NFSM into a DFSM. Since the accepting states are not required for the algorithm, the NFSM can be regarded as an NFA and converted into a "DFA", which is really a DFSM. The correctness of this transformation is shown in the next section.

The algorithm converts an NFSM (Σ, Q, D, q_o) in a DFSM $(\Sigma, Q', \Delta, q'_0)$ with $Q' \subseteq 2^Q$. It first constructs a start node $q'_0 = \mathcal{E}_D(\{q_0\})$ and then determines for all DFSM nodes q' all outgoing edges δ' by expanding all edges in the contained NFSM nodes:

$$\begin{aligned} \delta(q') &= \{(q', \sigma, q'_2 | \sigma \in \Sigma, q'_2 \neq \emptyset, \\ &\quad q'_2 = \{\mathcal{E}_D(q_2) | (q, \sigma, q_2) \in D, q \in q'\}\}. \end{aligned}$$

This results in the DFSM $(\Sigma, Q', \Delta, q'_o)$ with

$$\begin{aligned} Q'_0 &= \{q'_0\} \\ Q'_i &= \textstyle\bigcup_{q' \in Q'_{i-1}} \{q'_2 | \exists \sigma \in \Sigma : (q', \sigma, q'_2) \in \delta(q')\} \\ Q' &= \textstyle\bigcup_{i=0}^{\infty} Q'_i \\ \Delta &= \textstyle\bigcup_{q' \in Q'} \delta(q). \end{aligned}$$

A.3 Correctness of the FSM transformation

Proposition: Given an NFSM (Σ, Q, D, q_o), the DFSM $(\Sigma, Q' \subseteq 2^Q, \Delta, q'_0)$ constructed by using the transformation algorithm for NFA to DFA described in [2] behaves exactly like the NFSM, i.e.

$$\begin{aligned} &1) \quad \forall w \in \Sigma^*, q \in Q, q_0 \xrightarrow{w} q \ \exists q' \in Q' : q'_0 \xrightarrow{w} q' \wedge q \in q' \\ &2) \qquad \forall w \in \Sigma^*, q'_a \in Q', q'_b \in Q', q_a \in q'_a, q_b \in q'_b : \\ &\qquad\qquad (q_a \xrightarrow{w} q_b) \text{ iff } (q'_a \xrightarrow{w} q'_b) \end{aligned}$$

Proof: Proposition 1) trivially follows from the definition of the transformation algorithm, see the definition of δ' and Q' in Section A.2.

The proof for proposition 2) can be derived from the proof in [2], Chapter 2.3: there, it is shown that for all $w \in \Sigma^*$, given a node q in the NFA and a node q' in the transformed DFA with $q \in q'$, a node f' in the DFA contains a node f in the NFA if and only if $q \xrightarrow{w} f$ and $q' \xrightarrow{w} f'$. Since the DFSM is constructed using the same algorithm, this results in proposition 2).

Therefore, the conversion algorithm used to convert an NFA into a DFA can be used to convert the NFSM describing the ordering transitions to a DFSM that behaves the same way as the NFSM.