A Combined Framework for Grouping and Order Optimization

Thomas Neumann, Guido Moerkotte
tneumann|moerkotte@informatik.uni-mannheim.de

Fakultät für Mathematik und Informatik,
University of Mannheim, Germany

Abstract

Since the introduction of cost-based query optimization by Selinger et al. in their seminal paper, the performance-critical role of interesting orders has been recognized. Some algebraic operators change interesting orders (e.g. sort and select), while others exploit them (e.g. merge join). Likewise, Wang and Cherniack (VLDB 2003) showed that existing groupings should be exploited to avoid redundant grouping operations. Ideally, the reasoning about interesting orderings and groupings should be integrated into one framework.

So far, no complete, correct, and efficient algorithm for ordering and grouping inference has been proposed. We fill this gap by proposing a general two-phase approach that efficiently integrates the reasoning about orderings and groupings. Our experimental results show that with a modest increase of the time and space requirements of the preprocessing phase both orderings and groupings can be handled at the same time. More importantly, there is no additional cost for the second phase during which the plan generator changes and exploits orderings and groupings by adding operators to subplans.

1 Introduction

The most expensive operations (e.g. join, grouping, duplicate elimination) during query evaluation can be performed more efficiently if the input is ordered or grouped in a certain way. Therefore, it is crucial for query optimization to recognize cases where the input of an operator satisfies the ordering or grouping requirements needed for a more efficient evaluation. Since a plan generator typically considers millions of different plans – and, hence, operators –, this recognition easily becomes performance critical for query optimization, often leading to heuristic solutions.

The importance of exploiting available orderings has been recognized in the seminal work of Selinger et al [4]. They presented the concept of interesting orderings and showed how redundant sort operations could be avoided by reusing available orderings, rendering sort-based operators like sort-merge join much more interesting.

Along these lines it is beneficial to reuse available grouping properties, usually for hash-based operators. While heuristic techniques to avoid redundant group-by operators have been given [1], groupings have not been treated as thoroughly as orderings. One reason might be that while orderings and groupings are related (every ordering is also a grouping), groupings behave somewhat differently. For example, a tuple stream grouped on the attributes $\{a, b\}$ need not be grouped on the attribute $\{a\}$. This is different from orderings, where a tuple stream ordered on the attributes (a, b) is also ordered on the attribute (a). Since no simple prefix (or subset) test exists for groupings, optimizing groupings even in a heuristic way is much more difficult than optimizing orderings. Still it is desirable to combine order optimization and the optimization of groupings, as the problems are related and treated similarly during plan generation. Recently, some work in this direction has been published [7]. However, this only covers a special case of grouping, as we will discuss in some detail in Section 3.

Experimental results have shown that the costs for order optimization can have a large impact on the total costs of query optimization [3]. Therefore, some care is needed when adding groupings to order optimization, as a slowdown of plan generation would be unacceptable. In this paper, we integrate groupings by con-

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

structing a state machine for groupings and combining it with a state machine for orderings. Experimental results show that this efficiently handles orderings and groupings at the same time, with no additional costs during plan generation and only modest one time costs. Actually the operation needed for grouping optimization during plan generation can be performed in $O(1)$, basically allowing to exploit groupings for free.

The rest of the paper is organized as follows. In Section 2, we introduce some notations and formalize the problems of order optimization and grouping optimization. Section 3 describes related work. This is followed by a rough sketch of our approach in Section 4. The detailed algorithm is described in Section 5. Finally, an experimental evaluation of our algorithm is presented in Section 6. Conclusions are drawn in Section 7.

2 Problem Definition

The described framework combines order optimization and the handling of grouping in one consistent set of algorithms and data structures. In this section, we give a more formal definition of the problem and the scope of the framework. First, we define the operations of ordering and grouping (Sections 2.1 and 2.2). Then we briefly discuss functional dependencies (Section 2.3) and how they interact with algebraic operators (Section 2.4) and finally we explain how the framework can be used for plan generation (Section 2.5).

2.1 Ordering

During plan generation, many operators require or produce certain orderings. To avoid redundant sorting it is required to keep track of the orderings a certain plan satisfies. The orderings that are relevant for query optimization are called *interesting orders* [4]. 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, and
2. all orderings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

This includes the final ordering requested by the given query, if this is specified.

The interesting orders are *logical orderings*. This means that they specify a condition a tuple stream must meet to satisfy the given ordering. In contrast, the *physical ordering* of a tuple stream is the actual succession of tuples in the stream. Note that while a tuple stream has only one physical ordering, it can satisfy multiple logical orderings. For example, the stream of tuples $((1,1),(2,2))$ with schema (a,b) has one physical ordering (the actual stream), but satisfies the logical orderings a, b, ab and ba.

Some operators, like `sort`, actually influence the physical ordering of a tuple stream. Others, like `select`, only influence the logical ordering. For example, a `sort[a]` produces a tuple stream satisfying the ordering (a) by actually changing the physical order of tuples. After applying `select[a=b]` to this tuple stream, the result satisfies the logical orderings $(a),(b),(a,b),(b,a)$, although the physical ordering did not change. Deduction of logical orderings can be described by using the well-known notion of *functional dependencies* (FD) [5]. In general, the influence of a given algebraic operator on a set of logical orderings can be described by a set of functional dependencies.

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* $o = (A_{o_1}, \ldots, A_{o_m})$ $(1 \le o_i \le n)$ if and only if for all $1 \le i < j \le r$ the following condition holds:

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

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

Let R be a tuple stream satisfying both the logical ordering $o = (A_1, \ldots, A_n)$ and the functional dependency $f = B_1, \ldots, B_k \rightarrow B_{k+1}$[1] with $B_i \in \{A_1 \ldots A_n\}$. 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 before this position in o. For example consider a tuple stream satisfying the ordering (a,b); after inducing the functional dependency $a, b \rightarrow c$ the tuple stream also satisfies the ordering (a,b,c), but not the ordering (a,c,b). 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 $o \vdash_{A_i \rightarrow A_j} O_1$, the second is O_2 defined as $o \vdash_{A_j \rightarrow A_i} O_2$, 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. For example, consider a tuple stream satisfying the ordering (a); after inducing the equation $a = b$ the tuple stream also satisfies the orderings $(a,b),(b)$ and (b,a). 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 $o \vdash_{\emptyset \rightarrow A} O'$. For example, consider a tuple stream satisfying the ordering (a,b); after inducing the equation $c = const$ the tuple

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

stream also satisfies the orderings $(c,a,b),(a,c,b)$ and (a,b,c).

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

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

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 $o' \in \Omega(O,F)$.

2.2 Grouping

It was shown in [7] that similar to order optimization, it is beneficial to keep track of the groupings satisfied by a certain plan. Traditionally, group-by operators are either applied after the rest of the query has been processed or are scheduled using some heuristics [1]. However, the plan generator could take advantage of grouping properties produced e.g. by avoiding re-hashing if such information was easily available.

Analogous to order optimization we call this *grouping optimization* and define that the set of *interesting groupings* for a given query consists of

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

This includes the grouping specified by the group by clause of the query, if any exists.

These groupings are similar to logical orderings, as they specify a condition a tuple stream must meet to satisfy a given grouping. Likewise functional dependencies can be used to infer new groupings.

More formally, a tuple stream $R = (t_1,\ldots,t_r)$ in attributes $A_1,\ldots,A_n$ *satisfied the grouping* $g = \{A_{g_1}\ldots,A_{g_m}\}$ $(1 \le g_i \le n)$ if and only if for all $1 \le i < j < k \le r$ the following condition holds:

$$\begin{aligned}&\forall 1 \le l \le m \quad t_i.A_{g_l} = t_k.A_{g_l}\\ \Rightarrow\ &\forall 1 \le l \le m \quad t_i.A_{g_l} = t_j.A_{g_l}\end{aligned}$$

Two remarks are in order here. First, note that a grouping is a set of attributes and not – as orderings – a sequence of attributes. Second, note that given two groupings g and $g' \subset g$ and a tuple stream R satisfying the grouping g, R need not satisfy the grouping g'. For example the tuple stream $((1,2),(2,3),(1,4))$ with the schema (a,b) is grouped by $\{a,b\}$, but not by $\{a\}$. This is different from orderings, where a tuple stream satisfying a ordering o also satisfies all orderings that are a prefix of o.

New groupings can be inferred by functional dependencies as follows: Let R be a tuple stream satisfying both the grouping $g = \{A_1,\ldots,A_n\}$ and the functional dependency $f = B_1,\ldots,B_k \to B_{k+1}$ with $\{B_1,\ldots,B_k\} \subseteq \{A_1,\ldots,A_n\}$. Then R also satisfies the grouping $g' = \{A_1,\ldots,A_n\} \cup \{B_{k+1}\}$. Let G' be the set of all groupings that can be constructed this way from g and f. Then we use the following notation: $g \vdash_f G'$. For example $\{a,b\} \vdash_{a,b\to c} \{a,b,c\}$. Let e be the equation $A_i = A_j$. Then $g \vdash_e G'$ where G' is the union of the following three sets. The first set is G_1 defined as $g \vdash_{A_i\to A_j} G_1$, the second is G_2 defined as $g \vdash_{A_j\to A_i} G_2$, and the third is the set of groupings derived from g where a possible occurrence of A_i is replaced by A_j or vice versa. For example $\{a,b\} \vdash_{b=c} \{a,c\}$. Let e be an equation of the form $A = const$. Then $g \vdash_e G'$ is defined as $g \vdash_{\emptyset\to A} G'$. For example $\{a,b\} \vdash_{c=const} \{a,b,c\}$.

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

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

Let $\Omega(G,F)$ be $\bigcup_{i=0}^{\infty}\Omega_i(G,F)$. We write $g \vdash_F g'$ if and only if $g' \in \Omega(G,F)$.

2.3 Functional Dependencies

The reasoning about orderings and groupings assumes that the set of functional dependencies is known. The process of gathering the relevant functional dependencies is described in detail in [5], predominantly there are three sources of functional dependencies:

1. key constraints
2. join predicates
3. filter predicates

However the algorithm makes no assumption about the functional dependencies, if for some reason an operator induces another kind of functional dependency this can be handled the same way.

2.4 Algebraic Operators

To illustrate the propagation of orderings and groupings during query optimization, we give some rules for concrete (physical) operators in Figure 1. Note that these rules somewhat depend on the actual implementation of the operators, e.g. a blockwise nested loop join might actually destroy the ordering if

operator	requires	produces
scan(R)	-	$O(R)$
select(S,$a = b$)	-	$\Omega(O(S), a = b)$
bnl-join(S_1,S_2)	-	$O(S_1)$
sort(S,$a_1, \ldots, a_n$)	-	$(a_1, \ldots, a_n)$
hash(S,$a_1, \ldots, a_n$)	-	$\{a_1, \ldots, a_n\}$
sort-merge(S_1,S_2,$a = b$)	$(a) \in O(S_1) \wedge (b) \in O(S_2)$	$\Omega(O(S_1), a = b)$
hash-join(S_1,S_2,$a = b$)	$\{a\} \in O(S_1) \wedge \{b\} \in O(S_2)$	$\Omega(O(S_1), a = b)$

Figure 1: Propagation of orderings and groupings

the blocks are stored in hash tables. As a shorthand, we use the following notation:

- $O(R)$ set of logical orderings and groupings satisfied by the physical ordering of the relation R
- $O(S)$ inferred set of logical orderings and groupings satisfied by the tuple stream S

2.5 Plan Generation

To exploit available logical orderings and groupings, the plan generator needs access to the combined order optimization and grouping component, which we describe as an *abstract data type* (ADT). An instance of this abstract data type `OrderingGrouping` represents a set of logical orderings and groupings, and wherever necessary, an instance is embedded into a plan note. The main operations the abstract data type `OrderingGrouping` must provide are

1. a constructor for a given logical ordering or grouping,
2. a membership test (called `contains(LogicalOrdering)`) which tests whether the set contains the logical ordering given as parameter,
3. a membership test (called `contains(Grouping)`) which tests whether the set contains the grouping given as parameter, and
4. an inference operation (called `infer(set<FD>)`). Given a set of functional dependencies and equations, it computes a new set of logical orderings and groupings a tuple stream satisfies.

These operations can be implemented by using the formalism described before: `contains(LogicalOrdering)` tests for $o \in O$, `contains(Grouping)` tests for $o \in G$ and `infer(F)` calculates $\Omega(O, F)$ respectively $\Omega(G, F)$. Note that the intuitive approach to explicitly maintain the set of all logical orderings and groupings is not useful in practice. For example, if a sort operator sorts a tuple stream on (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 problematic. This led Simmen et al. to introduce a more concise representation, which is discussed in the next section. As Simmen's technique is not easily applicable to groupings, currently no algorithm exists to efficiently maintain the set of available groupings. We close this gap. Further, our approach avoids these problems by only implicitly representing the set. Before presenting our approach, let us discuss the existing literature in detail.

3 Related Work

Very few papers exist on order optimization. While the problem of optimizing interesting orders was already introduced by Selinger et al.[4], later papers usually concentrated on exploiting, pushing down or combining orders, not on the abstract handling of orders during query optimization.

A more recent paper by Simmen et al.[5] introduced a framework based on functional dependencies for reasoning about orderings. The main idea was that instead of storing the potentially large set of logical orderings for each plan, only the initial ordering and the (usually much smaller) set of all induced functional dependencies is stored. When testing if a plan satisfies a given logical ordering, both the initial and the requested ordering are *reduced* using the available functional dependencies: An attribute is removed from an ordering if it is determined by an earlier attribute. E.g. given the ordering (a, b) and the functional dependency $a \rightarrow b$, the ordering can be reduced to (a), as the attribute b is redundant. After the reduction, two orderings can be compared using a simple prefix test. The main problem with this approach is that it requires a reduction step for each comparison. Although the reduced version of the initial ordering can be cached, the required ordering has to be reduced for every comparison. Since such comparisons are performed millions of times during plan generation, the performance impact is quite severe [3]. Also note that the reduction algorithm is not applicable for groupings (which of course was never intended by Simmen): Given the grouping $\{a, b, c\}$ and the functional depen-

dencies $a \rightarrow b$ and $b \rightarrow c$, the grouping would be reduced to $\{a, c\}$ or to $\{a\}$, depending on the order in which the reductions are performed. This problem does not occur with orderings, as the attributes are sorted and can be reduced back to front.

In previous work [3] we presented a framework that also used functional dependencies to reason about orderings, but described these orderings as finite state machines. The main idea was that since the interesting orderings and the functional dependencies were already known before starting the plan generation, the possible transitions between orderings could be precomputed and stored as a state machine. Then, during plan generation the orderings could be treated as states in the state machine, allowing very efficient comparisons and inference ($O(1)$ after the preparation step). More details about this approach will be given in Section 4. Experimental results have shown that modeling order optimization as state machines is very efficient and has a very positive influence on the runtime of plan generation.

A recent paper by Wang and Cherniack[7] presented the idea of combining order optimization with the optimization of groupings. Based upon Simmen's framework, they annotated each attribute in an ordering with the information whether it is actually ordered by or grouped by. For a single attribute a they write $O_{a^O}(R)$ to denote that R is ordered by a, $O_{a^G}(R)$ to denote that R is grouped by a and $O_{a^O \rightarrow b^G}$ to denote that R is first ordered by a and then grouped by b (within blocks of the same a value). Before checking if a required ordering or grouping is satisfied by a given plan, they use some inference rules to get all orderings and groupings satisfied by the plan. Basically, this is Simmen's reduction algorithm with two extra transformations for groupings. In their paper the check itself is just written as $\in$, however, at least one reduction on the required ordering would be needed for this to work (and even that would not be trivial, as the stated transformations on groupings are ambiguous). The promised details in the cited technical report are currently not available, as the report has not appeared yet. Also note that as explained above, the reduction approach is fundamentally not suited for groupings. In WangÄs and CherniackÄs paper this problem does not occur, as they only look at a very specialized kind of grouping: As stated in their Axiom 3.6, they assume that a grouping $O_{a^G \rightarrow b^G}$ is first grouped by a and then (within the block of tuples with the same a value) grouped by b. However, this a very strong condition that is usually not satisfied by a hash-based grouping operator. Therefore, their work is not general enough to capture the full functionality offered by a state-of-the-art query execution engine.

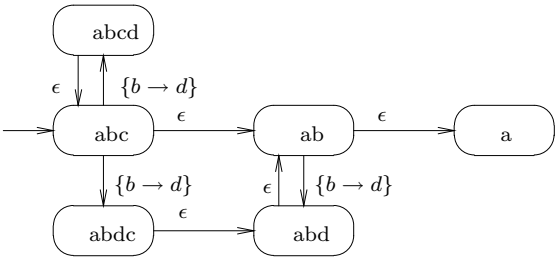

Figure 2: Possible FSM for orderings

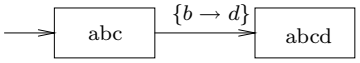

Figure 3: Possible FSM for groupings

4 Idea

As we have seen, explicit maintenance of the set of logical orderings and groupings can be very expensive. However, the ADT `OrderingGrouping` required for plan generation does not need to offer access to this set: It only allows to test if a given interesting order or grouping 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 and groupings from the state of the ADT.

In previous work [3], a framework was presented that provided an implicit representation of the set of logical orderings by using a *finite state machine* (FSM). An example of this is shown in Figure 2. The states are used to represent *physical orderings* and the edges are labeled with functional dependencies. *Logical orderings* are handled by pretending that the physical ordering changes as allowed by the functional dependency. Since one physical ordering can imply multiple logical orderings, ϵ-edges are used. They also provide a mechanism to compute the prefix closure. As a result, the FSM is a *non-deterministic finite state machine* (NFSM). Before the actual plan generation, the NFSM is converted into a *deterministic FSM* (DFSM), [3] describes some techniques to do this efficiently. It proposes techniques to avoid producing a large DFSM. Representing the set of orderings as FSM is very attractive, since during plan generation only the state of the FSM has to be remembered. Aside from the construction of the FSM this allows for order optimization operations in time $O(1)$.

The idea of our combined framework is to construct a similar FSM for groupings and integrate it into the FSM for orderings, thus handling orderings and groupings at the same time. An example of this is shown in Figure 3. Here, the FSM for the grouping $\{a, b, c\}$ and the functional dependency $b \rightarrow c$ is shown. We represent states for orderings as rounded boxes and

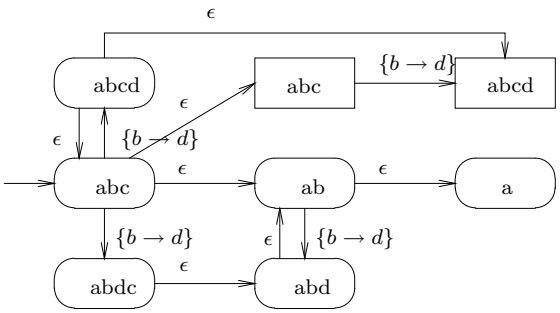

Figure 4: Combined FSM for orderings and groupings

states for groupings as rectangles. Note that although the FSM for groupings has a start node similar to the FSM for orderings, it is much smaller. This is due to the fact that groupings are only compatible with themselves, no nodes for prefixes are required. However, the FSM is still non-deterministic: given the functional dependency $b \rightarrow c$, the grouping $\{a, b, c, d\}$ is compatible with $\{a, b, c, d\}$ itself and with $\{a, b, c\}$; therefore, there exists an (implicit) edge from each grouping to itself.

The FSM for groupings is integrated into the FSM for orderings by adding ϵ edges from each ordering to the grouping with the same attributes; this is due to the fact that every ordering is also a grouping. Note that although the ordering (a, b, c, d) also implies the grouping $\{a, b, c\}$, no edge is required for this, since there exists an ϵ edge to (a, b, c) and from there to $\{a, b, c\}$.

After constructing an FSM as described above, the ADT can easily be mapped to the FSM: The state of the ADT is a state of the FSM and testing for a logical ordering or grouping can be performed by checking if the node with the ordering or grouping is reachable from the current state by following ϵ edges (as we will see, this can be precomputed to yield the O(1) time bound for the ADT operations). If the state of the ADT must be changed because of functional dependencies, the state in the FSM is changed by following the edge labeled with the functional dependency. However, the non-determinism of this transition is a problem. Therefore, for practical purposes the NFSM must be converted into a DFSM.

The framework for logical orderings [3] already described an algorithm for the conversion from NFSM to DFSM that can be reused for the combined framework. Some pruning techniques for groupings are described in Section 5 to minimize the NFSM, but the inclusion of groupings is not critical for the conversion, as the grouping part of the NFSM is nearly independent of the ordering part. In Section 6 we look at the size increase due to groupings. The memory consumption usually increases by a factor of two, which is the minimum expected increase, since every ordering is a grouping.

1. Determine the input
   (a) Determine interesting orders
   (b) Determine interesting groupings
   (c) Determine set of functional dependencies
2. Construct the NFSM
   (a) Construct nodes of the NFSM
   (b) Filter functional dependencies
   (c) Build filters for orderings and groupings
   (d) Add edges to the NFSM
   (e) Prune the NFSM
   (f) 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 5: Preparation steps of the algorithm

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 are the precomputed values 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 is quite involved. Most of this section covers the first phase. Only Section 5.6 deals with the ADT implementation.

Figure 5 gives an overview of the preparation phase. As the pure ordering framework is described in [3], we only briefly describe the general part and concentrate on the changes needed to support groupings. 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, the set of interesting groupings 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 the correctness of our algorithms that we note which of the interest-

ing orders are (1) produced by some algebraic operator or (2) only 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 determining the set of interesting orders we refer to [4, 5]. The groupings are classified similarly to the orderings: We denote the grouping produced by some algebraic operator by G_P, and those just tested for by G_T. The total set of interesting groupings is defined as $G_I = G_P \cup G_T$. More information on how to extract interesting groupings can be found in [7]. Furthermore, for a sample query the extraction of both interesting orders and groupings is illustrated in Section 6.

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

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

the interesting groupings

$$G_I = \{\{b\}\} \cup \{\{b, c\}\}$$

and the interesting orders

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

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

5.3 Constructing the NFSM

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

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

Coarsely Q consists of the relevant orderings and groupings, Σ of the functional dependencies and D describes how the orderings or groupings change under a given functional dependency. Some refinements are needed to provide efficient ADT operations. The details of the construction are described now.

For the order optimization part the states are partitioned in $Q = Q_I \cup Q_A \cup \{q_0\}$, were q_0 is an artificial node to initialize the ADT, Q_I is the set of nodes corresponding to interesting orderings and Q_A is a set of artificial nodes only required for the algorithm itself.

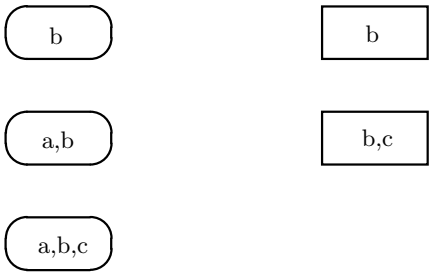

Figure 6: Initial NFSM for sample query

Q_A is described later. Furthermore, the set Q_I is partitioned in Q_I^P and Q_I^T, representing the orderings in O_P and O_T respectively. To support groupings, we add to Q_I^P nodes corresponding to the groupings G_P and to Q_I^T nodes corresponding to the groupings in G_T.

The initial NFSM contains the states Q_I of interesting groupings and orderings. For the example, this initial construction not including the start node q_o is shown in Figure 6. The states representing groupings are drawn as rectangles and the states representing orderings are drawn with rounded corners.

When considering functional dependencies, additional groupings and orderings can occur. These are not directly relevant for the query, but have to be represented by states to handle transitive changes. Since they have no direct connection to the query, these states are called artificial states. Starting with the initial states Q_I, artificial states are constructed by considering functional dependencies

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

. In our example this creates the states (b, c) and (a), as (b, c) can be inferred from (b) when considering $\{b \rightarrow c\}$ and (a) can be inferred from (a, b), since (a) is a prefix of (a, b). The result is show in Figure 7 (ignore the edges).

Sometimes the ADT has to be explicitly initialized with a certain ordering or grouping (e.g. after a `sort`). To support this, artificial edges are added later on. These point to the requested ordering or grouping (states in Q_I^P) and are labeled with the state that they lead to. Therefore, the input alphabet Σ consists of the sets of functional dependencies and produced orderings and groupings:

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

In our example $\Sigma = \{\{b \rightarrow c\}, \{b \rightarrow d\}, (b), (a, b), \{b\}\}$.

Accordingly, the domain of the transition relation D is

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

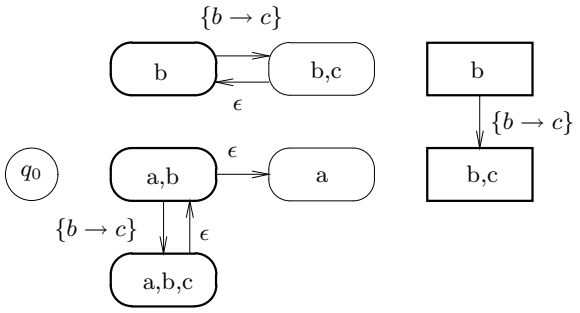

Figure 7: NFSM after adding D_{FD} edges

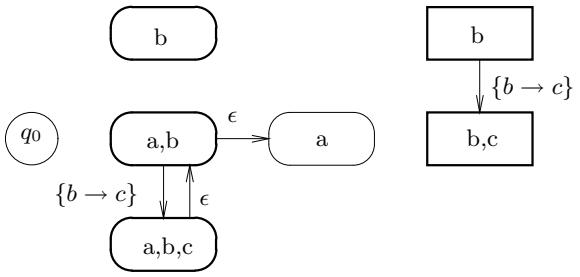

Figure 8: NFSM after pruning artificial nodes

The edges are formed by the functional dependencies and the artificial edges. Furthermore, ϵ edges exist between orderings and the corresponding groupings, as orderings are a special case of grouping:

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

First, the edges corresponding to functional dependencies are added (D_{FD}). In our example, this results in the NFSM shown in Figure 7.

Note that the functional dependency $b \rightarrow d$ has been pruned, since d does not occur in any interesting order or grouping. 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 8. A detailed description of these pruning techniques can be found in [3]. Additional pruning techniques relevant for groupings are described in Section 5.7.

The artificial start node q_0 has emanating edges incident to all nodes representing interesting orders in O_I^P and interesting groupings in G_I^P (D_A). Also, the nodes representing orderings have edges to their corresponding grouping nodes (D_{OG}), as every ordering is also a grouping. The final NFSM for the example is shown in Figure 9. Note that the nodes representing (a, b, c) and $\{b, c\}$ are not linked by an artificial edge since it is only tested for, as they are in Q_I^T.

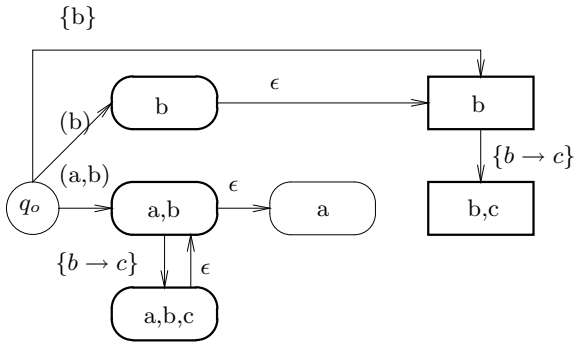

Figure 9: Final NFSM

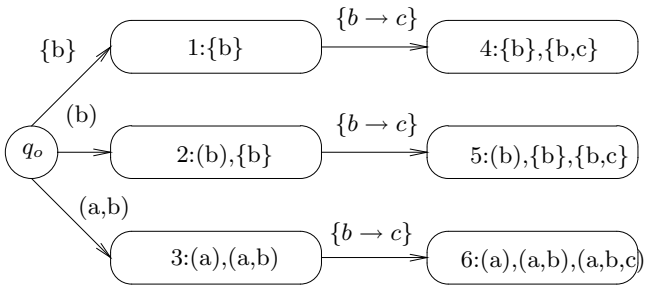

Figure 10: Resulting DFSM

5.4 Constructing the DFSM

The DFSM in constructed as described in [3]. Basically, the the standard power set construction for converting an NFA into a DFA [2]. is used. It is important to note that this construction preserves the start node and the artificial edges, allowing easy initialization of the ADT. The resulting DFSM for the example is shown in Figure 10.

5.5 Precomputing Values

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

The precomputation step itself computes two matrices. The first matrix denotes whether an NFSM

state	1: (a)	2: (a,b)	3: (a,b,c)	4: (b)	5: {b}	6: {b,c}
1	0	0	0	0	1	0
2	0	0	0	1	1	0
3	1	1	0	0	0	0
4	0	0	0	0	1	1
5	0	0	0	1	1	1
6	1	1	1	0	0	0

Figure 11: *contains* Matrix

state	1: $\{b \rightarrow c\}$	2: (a, b)	3: (b)	4: $\{b\}$
q_o	-	3	2	1
1	4	-	-	-
2	5	-	-	-
3	6	-	-	-
4	4	-	-	-
5	5	-	-	-
6	6	-	-	-

Figure 12: *transition* Matrix

node is in Q_I, i.e. an interesting order or an interesting grouping, is contained in a specific DFSM node. This matrix can be represented as a compact bit vector, allowing tests in O(1). For our running example, it is given (in a more readable form) in Figure 11. The second matrix contains the transition table for the DFSM relation D. Using it, edges in the DFSM can be followed in $O(1)$. For the example, the transition matrix is given in Figure 12.

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 and groupings in the form of the corresponding DFSM state. Hence, the state of the DFSM, a simple integer, is the state of our ADT `OrderingGrouping`.

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

For "atomic" subplans like table or index scans the ordering and grouping is determined explicitly by the operator. The state of the DFSM is determined by a lookup in the transition matrix with start state q_o and the edge annotated by the produced ordering or grouping. For sort and group by operators the state of the DFSM is determined as before by following the artificial edge for the produced ordering or grouping and then reapplying the set of functional dependencies that currently hold.

In the example, a sort on (b) results in a subplan with ordering/grouping state 2 (the node 2 is active in the DFSM), which satisfies the ordering (b) and the grouping $\{b\}$. After applying an operator which induces $b \rightarrow c$, the ordering/grouping changes to state 5 which also satisfies $\{b, c\}$.

5.7 Reducing the Size of the NFSM

Reducing the size of the NFSM is very important because it reduces both preparation time by avoiding large DFSMs and the search space for plan generation, as irrelevant orderings can be ignored. Effective techniques for pruning irrelevant ordering states and merging artificial nodes were already presented [3], we now describe how to avoid irrelevant grouping nodes.

First, in Step 2.3 (see Figure 5) the set of attributes occurring in interesting groupings is determined:

$$A_G = \{a \mid \exists g \in G_I : a \in g\}$$

Now, for every attribute a occurring on the right-hand side of a functional dependency the set of potentially reachable relevant attributes is determined:

$$\begin{aligned} r(a,0) &= \{a\} \\ r(a,n) &= r(a,n-1) \cup \\ &\quad \{a' \mid \exists (a_1 \dots a_m \rightarrow a') \in \mathcal{F} : \\ &\quad\quad \{a_1 \dots a_m\} \cap r(a,n-1) \neq \emptyset\} \\ r(a) &= r(a, |\mathcal{F}|) \cap A_G \end{aligned}$$

This can be used to determine if a functional dependency actually adds useful attributes. Given a functional dependency $a_1 \dots a_n \rightarrow a$ and a grouping g with $\{a_1 \dots a_n\} \subseteq g$, a should only be added to g if $r(a) \not\subseteq g$, i.e. the attribute might actually lead to a new interesting grouping. For example, given the interesting groupings $\{a\}, \{a, b\}$ and the functional dependencies $a \rightarrow c, a \rightarrow d, d = b$. When considering the grouping $\{a\}$, the functional dependency $a \rightarrow c$ can be ignored, as it can only produce the attribute c, which does not occur in an interesting grouping. However the functional dependency $a \rightarrow d$ should be added, since transitively the attribute b can be produced, which does occur in an interesting grouping.

Since there are no ϵ edges between groupings, i.e. groupings are not compatible with each other, a grouping can only be relevant for the query if it is a subset of an interesting ordering (as further attributes could be added by functional dependencies). However a simple subset test is not sufficient, as equations of the form $a = b$ are also supported; these can effectively rename attributes, resulting in a slightly more complicated test:

In Step 2.3 (see Figure 5) the equivalence classes induced by the equations in $\mathcal{F}$ are determined and for each class a representative is chosen:

$$\begin{aligned} E(a,0) &= \{a\} \\ E(a,n) &= E(a,n-1) \cup \\ &\quad \{a' \mid ((a = a') \in \mathcal{F}) \vee ((a' = a) \in \mathcal{F}\} \\ E(A) &= E(A, |\mathcal{F}|) \\ e(a) &= \operatorname{rep} E(A) \quad \text{(arbitrary)} \\ e(\{a_1 \dots a_n\}) &= \{e(a_1) \dots e(a_n)\}. \end{aligned}$$

Using these equivalence classes a mapped set of interesting groupings is produced, that will be used to test if a grouping is relevant:

$$G_I^E = \{e(g) \mid g \in G_I\}$$

Now a grouping g can be pruned if $\nexists g' \in G_I^E : e(g) \subseteq g'$. For example, given the interesting grouping $\{a\}$ and the equations $a = b, b = c$, the grouping $\{d\}$ can be pruned, as it will never lead to an interesting grouping; however, the groupings $\{b\}$ and $\{c\}$ have to be kept, as they could change to an interesting grouping later on.

Note that although they appear to test similar conditions, the first pruning technique (using $r(a)$) is not dominated by the second one (using $e(a)$). Consider e.g. the interesting grouping $\{a\}$, the equation $a = b$ and the functional dependency $a \rightarrow b$. Using only the second technique, the grouping $\{a, b\}$ would be created, although it is not relevant.

6 Experimental Results

Integrating groupings in the order optimization framework allows the plan generator to easily exploit groupings and thus produce better plans. However, order optimization itself might become prohibitively expensive by considering groupings. Therefore, we evaluated the costs of including groupings for different queries.

Since adding support for groupings has no effect on the runtime behavior of the plan generator (all operations are still one table lookup), we measured the runtime and the memory consumption of the preparation step both with and without considering groupings. When considering groupings, we treated each interesting ordering also as an interesting grouping, i.e. we assumed that a grouping-based (e.g. hash-based) operator was always available as an alternative. Since this is the worst-case scenario, it should give an upper bound for the additional costs. All experiments were performed on a 2.4 GHz Pentium IV, using the gcc 3.3.1.

To examine the impact for real queries, we choose a more complex query from the well-known TPC-R benchmark ([6], Query 8):

```
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, group-by and order-by clauses are added to the set of interesting orders. Since hash-based solutions are possible, they are also added to the set of interesting groupings. This results in the sets

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

Note that here O_I^T and G_I^T are empty, as we assumed that each ordering and grouping would be produced if beneficial. For example, we might assume that it makes no sense to intentionally group by o_year: If a tuple stream is already grouped by o_year it makes sense to exploit this, however instead of just grouping by o_year it could makes sense to sort by o_year, as this is required anyway (although here it only makes sense if the sort operator performs early aggregation). In this case $\{o\_year\}$ would move from G_I^P to G_I^T, as it would be only tested for, but not produced.

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

$$\begin{aligned}
\mathcal{F} &= \{\{p_partkey = l_partkey\}, \{\emptyset \rightarrow p_type\}, \\
&\quad \{o_custkey = c_custkey\}, \{\emptyset \rightarrow r_name\},
\end{aligned}$$

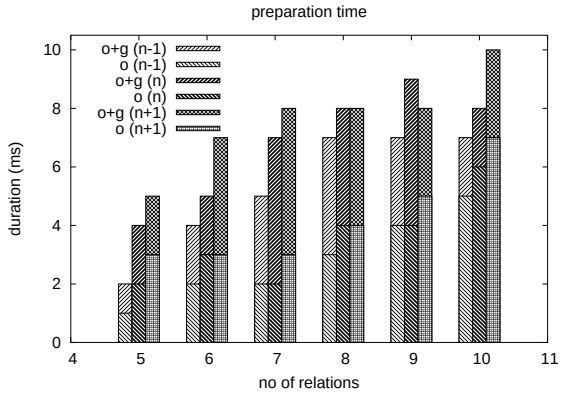

Figure 13: Time requirements for the preparation step

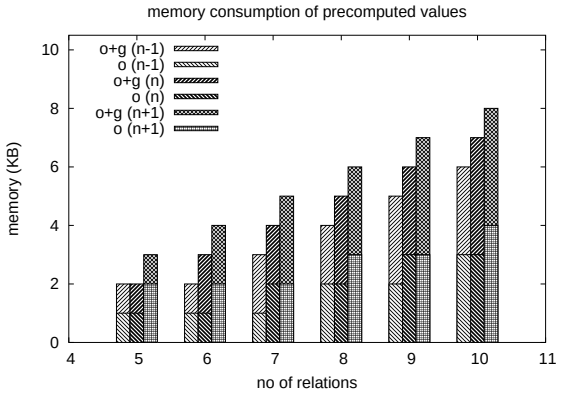

Figure 14: Space requirements for the preparation step

$$\{c\_nationkey = n1.n\_nationkey\},$$
$$\{s\_nationkey = n2.n\_nationkey\},$$
$$\{l\_orderkey = o\_orderkey\},$$
$$\{s\_suppkey = l\_suppkey\},$$
$$\{n1.n\_regionkey = r\_regionkey\}\}$$

To measure the influence of groupings, the preparation step was executed two times: Once with the data as given above and once with $G_I^P = \emptyset$ (i.e. groupings were ignored). The space and time requirements are shown below:

	With Groups	Without Groups
Duration [ms]	0.6ms	0.3ms
DFSM [nodes]	63	32
Memory [KB]	5	2

Here time and space requirements both increase by a factor of two. Since all interesting orderings are also treated as interesting groupings, a factor of about two was expected.

While Query 8 is one of the more complex TPC-R queries, it is not overly complex when looking at order optimization. It contains 16 interesting orderings/groupings and 8 functional dependencies, but they cannot be combined in many reasonable ways, resulting in a comparatively small DFSM. In order to get more difficult examples, we produced randomized queries with $5-10$ relations and a varying number of join predicates. We always started from a chain query and then randomly added additional edges to the join graph. The results are shown for $n-1$, n and $n+1$ additional edges. In the case of 10 relations that means that the join graph consisted of 18, 19 and 20 edges respectively.

The time and space requirements for the preparation step are shown in Figure 13 and Figure 14, respectively. For each number of relations the requirements for the combined framework (`o+g`) and the framework ignoring groupings (`o`) are shown. The numbers in parentheses ($n-1$, n and $n+1$) are the number of additional edges in the join graph.

As with Query 8, the time and space requirements roughly increase by a factor of two when adding groupings. This is a very positive result, given that a factor of two can be estimated as a lower bound (since every interesting ordering is also an interesting grouping here). Furthermore, the absolute time and space requirements are very low (a few ms and a few KB), encouraging the inclusion of groupings in the order optimization framework.

7 Conclusion

The combined framework presented allows a very efficient handling of order optimization and grouping optimization during plan generation. The experimental results showed that with only a modest increase of the one-time costs, groupings can be exploited during plan generation at no additional costs. In summary, using an FSM to keep track of the available orderings and groupings is very efficient and is easily integrated in a plan generator.

One topic for future work is the minimization of the DFSM using the operator structure. Currently, only the NFSM is pruned by detecting irrelevant or redundant nodes. The DFSM could also be pruned by intentionally dropping available logical orderings or groupings when it is clear that the ordering or grouping will never be used (because of operator dependencies). Besides minimizing the DFSM, this technique would also reduce the search space for the plan generator, as more plans could be pruned (since more plans would be dominated by other plans).

References

[1] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366. Morgan Kaufmann, 1994.

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

[3] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. In *Proceedings of the 20th International Conference on Data Engineering, 30 March - 2 April 2004, Boston, MA*. IEEE Computer Society, 2004.

[4] 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.

[5] 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.

[6] 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.

[7] Xiaoyu Wang and Mitch Cherniack. Avoiding sorting and grouping in processing queries. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*. Morgan Kaufmann, 2003.