Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types

Giedrius Slivinskas, Christian S. Jensen

August 11, 2002

TR-70

A TimeCenter Technical Report

Title	Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types
	Copyright © 2002 Giedrius Slivinskas, Christian S. Jensen. All rights reserved.
Author(s)	Giedrius Slivinskas, Christian S. Jensen
Publication History	Proceedings of the Fifth East-European Conference on Advances in Databases and Information Systems, Vilnius, Lithuania, September 25–28, 2001, pp. 55–69. August 2002. A TIMECENTER Technical Report

Abstract

Database management systems are continuously being extended with support for new types of data and more advanced querying capabilities. In large part because of this, query optimization has remained a very active area of research throughout the past two decades. At the same time, current commercial optimizers are hard to modify, to incorporate desired changes in, e.g., query algebras, transformation rules, search strategies. This has led to a number of research contributions that aim at creating extensible query optimizers. Examples include Starburst, Volcano, and OPT++.

This paper reports on a study that has enhanced Volcano to support a relational algebra with added temporal operators, such as temporal join and aggregation. This includes the handling of algorithms and cost formulas for these new operators, six types of query equivalences, and accompanying query transformation rules. The paper shows how the Volcano search-space generation and plan-search algorithms were extended to support the six equivalence types, describes other key implementation tasks, and evaluates the extensibility of Volcano.

Keywords—Query optimization; extensible query optimizers; extended relational algebra; implementation experiences; sets, multisets, and lists

1 Introduction

Query optimization has remained subject to active research for more than twenty years. Much research has aimed at enhancing existing optimization technology to enable it to support the requirements, such as for new types of data and queries, of the many and new types of application areas, to which database technology has been introduced over the years. However, current commercial optimizers also remain hard to extend and modify when new operators, algorithms, or transformations have to be added, or when cost estimation techniques or search strategies have to be changed [Cha98]. As a result, the last decade has witnessed substantial efforts aiming to develop extensible query optimizers that would make such changes easier. Representative examples of extensible query optimizers include Starburst [Haa90], Volcano [GM93], and OPT++ [KW99].

This paper reports on a specific study that has enhances the Volcano extensible query optimizer to support a relational algebra with temporal operators such as temporal join and aggregation. In addition to new operators, cost formulas, selectivity-estimation formulas, and transformation rules, the algebra offers systematic support for order preservation and duplicate removal and retention for all queries, as well as for coalescing for temporal queries (in coalescing, several tuples with adjacent time periods and otherwise identical attribute values are merged into one). To support order, relations are defined as lists, and six kinds of relation equivalences are defined—two relations can be equivalent as lists, multisets, and sets, and two temporal relations can be snapshot-equivalent as lists, multisets, and sets. We report on the design decisions and implementation experiences, and we evaluate Volcano's extensibility.

The temporally extended algebra, presented in [SJS01a], enhances existing relational algebras based on multisets by integrating the handling of order, and by adding temporal support. To support multisets and order, relations are defined as lists, and six kinds of relation equivalences are defined. Specifically, relations can be equivalent as lists, multisets, and sets, and temporal relations can also be snapshot-equivalent as lists, multisets, and sets. As there are six types of equivalences, there are also six types of transformation rules. Their applicability during query rewriting depends on the nature of the query (e.g., whether duplicates have to be removed or whether the result has to be sorted) and on the semantics of the operations in the query.

An important goal of the algebra is to offer a foundation for a layered temporal DBMS that may evaluate temporal queries faster than do current DBMSs. The latter do not have efficient algorithms for expensive temporal operations such as temporal aggregation, while such operations can be evaluated efficiently at the user-application level by algorithms that use cursors to access the underlying data.

New algorithms can be added to a DBMS via, e.g., user-defined routines in Informix [Bli99, Inf] or PL/SQL procedures in Oracle [Ora], but these methods currently do not allow to define functions that take tables as arguments and return tables [JM99]; nor do they allow to specify transformation rules, cost formulas, and selectivity-estimation formulas for the new functions. Because of these limitations, a middleware component with query processing capabilities was introduced, which divides the query processing between itself and the underlying DBMS [SJS01b]. Intermediate relations can be moved between the middleware and the DBMS by the help of transfer operators.

To adequately divide the processing, the middleware has to take optimization decisions—for this purpose, we employ the Volcano extensible optimizer. Use of a separate middleware optimizer allows us to take advantage of transformation rules and cost and selectivity-estimation formulas specific to the temporal operators. We have to also take into account that the DBMS has its own optimizer; therefore, the middleware optimizer does not have to focus on optimizing query parts to be passed to the DBMS for evaluation.

This paper summarizes design issues and experiences from the implementation. While the addition of new temporal operators, their cost and selectivity-estimation formulas, and transformation rules could be done using the extensibility framework provided by Volcano, adding support for set-, multiset-, and list-equivalences—as well as the three equivalences between temporal relations—required changes in Volcano structures, and in its search-space generation and plan-search algorithms.

To our knowledge, no existing extensible query optimizers systematically support sets, multisets, and lists. Sorting is treated differently than the common operators, such as selection or join, and it usually is considered in the query optimization only after the search space of possible query plans has been generated. However, particularly due to recent introduction and increasing use of `TOP N` and `BOTTOM N` predicates in queries [CK97], sorting could be exploited better in query optimization if considered during the search-space generation.

The paper is structured as follows. In Section 2, we describe Volcano's architecture, including its search-space generation and plan-search algorithms. Section 3 describes the enhancements to Volcano that were necessary to support the algebra introduced above. The algebraic framework is described first, with a focus on the parts that posed challenges to Volcano. Then the modifications to the search-space generation and plan-search algorithms of Volcano necessary to support the algebra are described. Section 4 summarizes the implementation experiences and evaluates the extensibility of Volcano. Section 5 covers related work, and Section 6 summarizes the paper.

2 Description of Volcano

The Volcano Optimizer Generator is a software program for generating extensible query optimizers. The input to the program is a `model.input` file, which specifies a query algebra: operators, their implementation algorithms (physical operators), transformation rules, and implementation rules. Transformation rules specify equivalent logical expressions, and implementation rules specify which algorithms implement which operators. The output from the program is an optimizer for queries in the given algebra. The generated Volcano optimizer takes a query as input and returns a physical expression (an expression of algorithms) representing the chosen query evaluation plan.

The tasks of an optimizer implementor include the specification of the `model.input` file and the coding of the support functions for operators and rules. For example, for each operator, the optimizer implementor should specify how to derive properties (for example, the cardinality) of the result relation.

In the next sections, we describe the functioning of the Volcano optimizer.

2.1 Two Stages of Query Optimization

The Volcano optimizer optimizes queries in two stages. First, the optimizer generates the entire search space consisting of logical expressions generated using the initial query plan (to which the query is mapped to) and the set of transformation rules. The search space is represented by a number of equivalence classes. An equivalence class may contain one or more logically equivalent expressions, also called elements; each of these includes an operator, its parameter (for example, predicates for the selection), and pointers to its inputs (which are also equivalence classes).

Consider a simple example query, which performs a join on the `EmpID` attribute of `POSITION` and `SALARY` relations. Its one possible initial plan is shown in Figure 1(a) and its search space is shown in Figure 1(b).

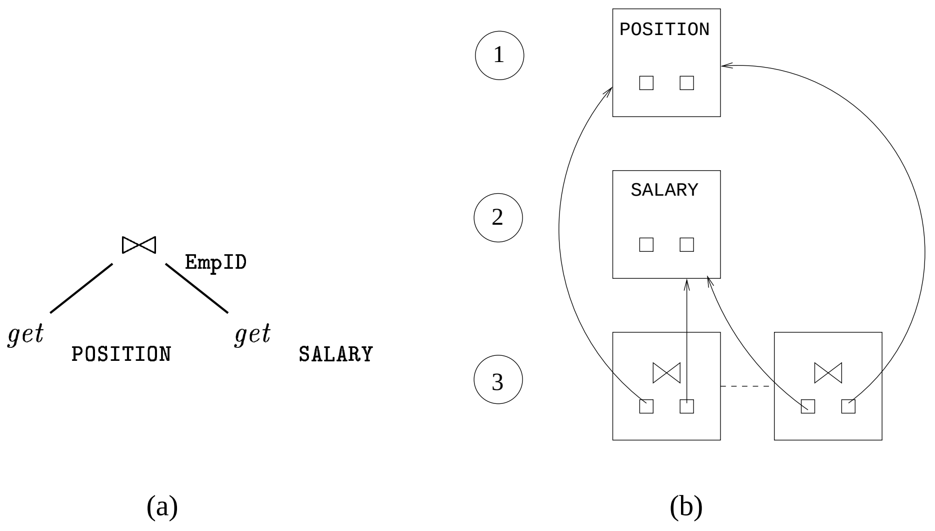

Figure 1: Initial Query Plan

The elements of classes 1 and 2 represent logical expressions returning partial results of the query, i.e., the operators retrieving, respectively, the `POSITION` and `SALARY` relations. The elements of class 3 represent logical expressions returning the result of the complete query; either the first or the second element may be used. Essentially, the given search space represents only two plans which differ in the order of the join arguments.

During the second stage of Volcano's optimization process, the actual search for the best plan is performed. Here, the implementation rules are used to replace operators by algorithms, and the costs of diverse subplans are estimated. For the given query, the number of plans to be considered is greater than two, because the relations may be retrieved by using either full scan or index scan, and the join has several implementations, such as nested-loop, sort-merge, or index join. One possible evaluation plan is to scan both relations and perform a nested-loop join.

The following two sections describe the search-space generation and the plan-search algorithms in more detail.

2.2 Stage One: Search-Space Generation

Figure 2 outlines simplified pseudocode for $Generate$ and $MatchRule$, the two main functions for the search-space generation. Initially, one element is created for each operator in the original query expression, and then $Generate$ is invoked on the top element.

```
$Generate$(element $e$):
    **for each** child $i$ of $e$
        $Generate(i)$
    **for each** element $ee \in class(e)$
        **do**
            applied := $MatchRule$(e)
        **while** (applied = True)
    **return**
```

```
$MatchRule$(element $e$):
    $r$ := $findMatchingRule(e)$
    **if** $r \neq NULL$
        create elements and classes based on $r$'s right-hand side
        **for each** new element $ee$
            **if** class($ee$) $\neq$ class($e$)
                $Generate(ee)$
        **return** True
    **else**
        **return** False
```

Figure 2: Pseudocode for the Search Space Generation

The $Generate$ function repeatedly invokes the $MatchRule$ function, which applies a transformation rule to the given element, choosing from the list of applicable rules that have not so far been applied to the element (as found by the $FindMatchingRule$ function in $MatchRule$). The application of a transformation rule in $MatchRule$ may trigger the creation of new elements and classes; for each newly generated element, the $Generate$ function is invoked.

For the query in Figure 1(a), the search space is generated as follows. Initially, three elements representing the three query-tree operators are created (first elements of equivalence classes 1–3 in Figure 1(b)). Then, the $Generate$ function is invoked for the first element of class 3, which, in turn, invokes $Generate$ for the first elements of classes 1 and 2. The latter two $Generate$ calls do not do anything because no rules apply to the elements of class 1 and 2. For the first element of class 3, however, the join commutativity rule is applied, and a second element pointing to switched join arguments is added to class 3. Then, the $MatchRule$ function is invoked on the new element of class 3, but no new elements are generated: the join commutativity rule is applied again, but its resulting right-hand element already exists in the search space.

2.3 Stage Two: Plan Search

When searching for a plan, the Volcano optimizer employs dynamic programming in a top-down manner, and it uses two mutually recursive functions, $FindBestPlan$ and $Optimize$. Figure 3 shows the simplified pseudocode for the functions.

First, the optimizer invokes the $FindBestPlan$ function for the first element of the top equivalence class—e.g., class 3 in Figure 1(b)—and the cost limit infinity (the cost limit can be lower in subsequent calls to the function). If all elements of the class containing the argument element have already been optimized (line 1), no further optimization for the element is necessary: if the plan has been found and its cost is lower than the cost limit, it is returned, otherwise NULL is returned (lines 2–5). However, if the optimization for the class has not been completed, the $Optimize$ function is invoked (line 7).

The $Optimize$ function for each algorithm implementing the top operator (in our case, join) recursively invokes the $FindBestPlan$ function for the inputs of the algorithm (lines 5–6). The $remaining\_cost$ argument is used to prune the search when it is clear that the search would not come up with a more efficient plan than the current one. If optimization of the inputs is successful (line 8), and if the found plan is the first for the equivalence class containing the argument element or it beats the cost of the existing best plan (line 9), it is saved along with its cost (lines 10–11).

Once all algorithms are considered for the operator, the $Optimize$ function invokes the $FindBestPlan$ function for each equivalent logical expression (in our case, for the second element in equivalence class 3) and looks if it can find a better plan (lines 12–14). In case a better plan is found, it is saved in memory as the best one (lines 15–16). Once all elements of the input-element class are considered, the algorithm marks that the optimization of the class is completed.

```
    $FindBestPlan$(element $e$, cost $cost\_limit$):
1       **if** $completed(class(e)) = $ TRUE
2           **if** plan $P$ for $class(e)$ exists in memory **and** $cost(P) < cost\_limit$
3               **return** $P$
4           **else**
5               **return** NULL
6       **else**
7           **return** $Optimize$(e, cost_limit)
```

```
    $Optimize$(element $e$, cost $cost\_limit$):
1       $best\_plan$ := NULL
2       $min\_cost$ := $cost\_limit$
3       **for each** algorithm $a$ implementing $operator(e)$
4           $current\_plan$ := $a$ applied to plans of its inputs
5           **for each** input $i$ of $a$
6               $input\_plan[i]$ := $FindBestPlan(i, remaining\_cost)$
7                   /* $remaining\_cost$ is $min\_cost$ minus the cost of $a$ and the cost of plans of previous inputs */
8           **if for each** input $i$, $input\_plan[i] \neq$ NULL
9               **if** there is no plan $P$ for $class(e)$ in memory **or** $cost(current\_plan) < cost(P)$
10                  $best\_plan$ := $current\_plan$
11                  $min\_cost$ := $cost(current\_plan)$
12      **for each** element $x$, $x \in class(e), x \neq e$
13          $tmp\_plan$ := $FindBestPlan(x, min\_cost)$
14          **if** $tmp\_plan \neq$ NULL **and** $cost(tmp\_plan) < min\_cost$
15              $best\_plan$ := $tmp\_plan$
16              $min\_cost$ := $cost(tmp\_plan)$
17      set $completed(class(e))$ to True
18      **return** $best\_plan$
```

Figure 3: Pseudocode for the $FindBestPlan$ and $Optimize$ Functions

3 Enhancement of Volcano

The implementation of the algebra and its accompanying transformation rules introduces several concepts that did not exist previously in Volcano. We describe these new concepts in Section 3.1. Sections 3.2 and 3.3 concern the actual implementation and describe the modifications of the Volcano search-space generation and plan-search algorithms.

3.1 Algebra and Multi-Equivalence Transformation Rules

First, we overview the architecture for which the algebra has been designed. Next, we describe the actual algebra, the accompanying transformation rules, and their applicability, focusing on the new concepts. Finally, we outline the challenges that these new concepts pose to Volcano.

Architecture The temporally extended relational algebra has been designed for an architecture consisting of a middleware component and an underlying DBMS. Expensive temporal operations such as temporal aggregation do not have efficient algorithms in the DBMS, but can be evaluated efficiently by the middleware, which uses a cursor to access DBMS relations. Consequently, query processing is divided between the middleware and the DBMS; the main processing medium is still the DBMS, but the middleware is used when this can yield better performance.

Algebra The algebra is different from the conventional relational algebra in several aspects. First, it includes temporal operators such as temporal join and temporal aggregation. Next, it contains two transfer operators, T^M and T^D, that allow to partition the query processing between the middleware and the underlying DBMS. The T^M operator transfers a relation from the DBMS to the middleware, and the T^D operator performs the opposite. Finally, the algebra provides a consistent handling of duplicates and order at logical level, by treating duplicate elimination and sorting as other logical operators and by introducing six types of equivalences between relations.

Two relations are equivalent (1) as lists if they are identical lists ($\equiv_L$); (2) as multisets if they are identical multisets taking into account duplicates, but not order ($\equiv_M$); and (3) as sets if they are identical sets, ignoring duplicates and order ($\equiv_S$). Two temporal relations (relations having two attributes indicating a time period) are snapshot-list ($\equiv_L^S$), snapshot-multiset($\equiv_M^S$), or snapshot-set equivalent ($\equiv_S^S$), if their snapshots (projections at a given point in time) are equivalent as lists, multisets, or sets.

Figure 4 shows two temporal relations (relations having two attributes that encode a time period), `POSITION` and `SALARY`. We assume a closed-open representation for time periods and assume the time values for `T1` and `T2` denote months during some year. For example, Tom was occupying position `Pos1` from February to August (not including the latter).

POSITION

PosID	EmpID	EmpName	T1	T2
Pos1	1	Tom	2	8
Pos2	2	Jane	3	8

SALARY

EmpID	Amount	T1	T2
1	100K	2	6
1	120K	6	9
2	110K	3	8

Result

EmpID	EmpName	PosID	Amount	T1	T2
1	Tom	Pos1	100K	2	6
1	Tom	Pos1	120K	6	8
2	Jane	Pos2	110K	3	8

Figure 4: Relations `POSITION` and `SALARY`, and the Result of Temporal Join

A temporal join is a regular join, but with a selection on the time attributes, ensuring that the joined tuples have overlapping time periods; Figure 4 shows the result of temporal join on the `EmpID` attribute of the `POSITION` and `SALARY` relations.

Transformation Rules Six types of equivalences lead to six types of transformation rules, since a transformation rule may satisfy several of the six equivalences. Let us consider two rules for temporal join, $\bowtie^T$ (regular join, but with a selection on the time attributes, ensuring that the joined tuples have overlapping time periods). For a given rule, we always specify the strongest equivalence type that holds; the ordering of equivalence types is given in Figure 5. The join commutativity rule $r_1 \bowtie^T r_2 \rightarrow_M r_2 \bowtie^T r_1$ says that the relations resulting from the left-hand and right-hand sides are equivalent as multisets (and, according to the type ordering, as sets, as well as their snapshots are equivalent as sets and multisets). Meanwhile, the sort push-down rule $sort_A(r_1 \bowtie^T r_2) \rightarrow_L sort_A(r_1) \bowtie^T r_2$, where A belongs to the attribute schema of r_1 and the left-hand side operations are located in the middleware, says that the relations are equivalent as lists and that the other five equivalence types also hold.[1] The latter rule exploits the fact that all temporal join algorithms in the middleware retain the sorting of their left arguments.

[1]To be precise, the relations are $\equiv_{L,A}$ equivalent, i.e., their projections on A are $\equiv_L$ equivalent. We will use $\equiv_L$ equivalence for simplicity.

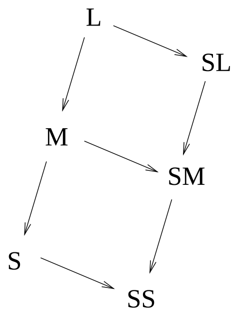

Figure 5: Ordering of Equivalence Types

Applicability of Transformation Rules Transformation rules that do not guarantee $\equiv_L$ equivalence cannot always be applied, as illustrated by the following example. Consider a query that performs a temporal join on the `EmpID` attribute of the `POSITION` and `SALARY` relations and sorts the result by `EmpName`. One possible initial plan for this query is shown in Figure 6(a). The bottom operators represent relations `POSITION`

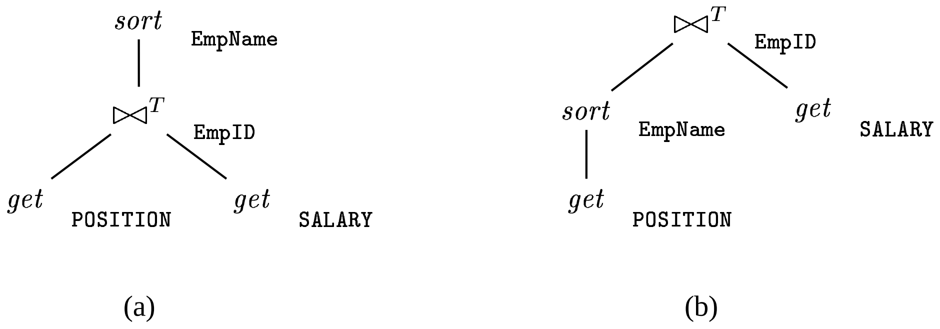

Figure 6: Initial Query Plan

and `SALARY` transferred to the middleware; to achieve this, at least two operations are necessary (a table scan in the DBMS and the actual transfer), but to simplify the example, we view them as one operation and do not consider any transformation rules related to these operations. Temporal join and sorting are performed in the middleware.

Let us consider rule $r_1 \bowtie^T r_2 \rightarrow_M r_2 \bowtie^T r_1$. This rule can be applied to switch the arguments of the join. However, if we apply the sort push-down rule first and move the sorting below the temporal join, before the temporal join's left argument (leading to the plan shown in Figure 6(b)), the application of join commutativity rule would lead to an incorrectly ordered query result. Thus, to be able to tell when an $\rightarrow_M$ rule is applicable, the optimizer needs to know the importance of order at each node in the query tree, i.e., whether the result of the operation at the node has to preserve some order or not. In the algebra, this importance is determined by the *OrderRequired* property. To determine the applicability of rules of other types, two other properties *DuplicatesRelevant* and *PeriodPreserving*, are used additionally. These rules are explained in Table 1.

Next, for each rule of a given type, Table 2 shows the applicability condition for operator nodes on the left-hand side of the rule (we revisit this aspect in Section 3.2).

Having an initial query plan, the properties for operators are set in a top-down manner and then adjusted

Property Name	Description
$OrderRequired$	True **if** the result of the operation must preserve some order
$DuplicatesRelevant$	True **if** the operation cannot arbitrarily add or remove regular duplicates
$PeriodPreserving$	True **if** the operation cannot replace its result with a snapshot-equivalent one

Table 1: Operation Properties

every time a new transformation rule is applied. For the top operator, the properties are set in accordance with the specific user-level query language and query statement. For example, some variant of SQL may require (1) the result to be sorted if the `ORDER BY` clause is specified at the outer-most level, (2) the result always either to contain duplicates (`DISTINCT` is not specified) or not (`DISTINCT` is specified), and (3) the result always to contain the same time periods independently of which query plan is chosen. Consequently, for the top element, the $OrderRequired$ property is set to True only if the `ORDER BY` clause is specified at the outer-most level, and the $DuplicatesRelevant$ and $PeriodPreserving$ properties are always set to True. For the other operators, the properties are set according to the property values of their parents, e.g., if some operator is an input to the $sort$ operator, its $OrderRequired$ property will be set to False, because its resulting relation may be replaced (via some transformation rule) by a multiset-equivalent relation, and the correct order of the result will still be ensured by the following $sort$ operator. For more details about setting the property values, we refer to [SJS01a].

Support in Volcano Volcano provides a framework of adding new operators and transformation rules, which allows a rather straightforward addition of temporal operators and transfer operators, their cost formulas, selectivity-estimation formulas, and schema propagation formulas. The difficult part is to incorporate different types of transformation rules. While different rule types can be added by just introducing an extra *type* attribute to each rule, to control their applicability is more difficult. The property mechanism cannot directly be used because of Volcano's search-space structure. Having a Volcano search space, values of the three properties cannot be determined for an element, because it is impossible to know the property values of the elements above since the same equivalence-class element may be used as input by different elements of different equivalence classes, as shown later in Figure 7 where the first element of equivalence class 2 is used both by two elements of equivalence class 3 and by two elements of class 4. Therefore, the determination of the properties can only occur during the actual search, which is performed top down.

3.2 Adjustment of the Search-Space Generation

Since it is impossible to determine properties during the search-space generation, we generate a complete search space by applying transformation rules of *all* types, and then filter away invalid elements during the actual search. The identification of invalid elements is enabled by recording, for each element, a type that represents the combinations of the three property values for which this element may be used. We use six possible type values—L, M, S, SL, SM, SS—which correspond to the six equivalence types. Consequently, the relationship between each element type and the combination of properties corresponds to the one shown in Table 2. For example, if all properties are True, only L type elements are valid. Intuitively, the element type tells how the relation generated by this element will be equivalent to the first element of the equivalence class.

Figure 7 shows the search space for the query in Figure 6(a), generated using the join commutativity rule and the sort push-down rule (the first one guarantees $\equiv_M$ equivalence, while the second one guarantees $\equiv_L$ equivalence). The search space is generated as follows. Initially, four elements representing the four query-tree operators are created (the first elements of equivalence classes 1–4 in Figure 7). Then, the

Rule type	**Applicability condition,** $\forall op \in lhs$
$\rightarrow_L$	True
$\rightarrow_M$	$\neg OrderRequired(op)$
$\rightarrow_S$	$\neg DuplicatesRelevant(op) \land \neg OrderRequired(op)$
$\rightarrow_L^S$	$\neg PeriodPreserving(op)$
$\rightarrow_M^S$	$\neg OrderRequired(op) \land \neg PeriodPreserving(op)$
$\rightarrow_S^S$	$\neg DuplicatesRelevant(op) \land \neg OrderRequired(op) \land \neg PeriodPreserving(op)$

Table 2: Applicability of a Rule According to its Type

$Generate$ function is invoked for the first element of class 4, which, in turn, invokes $Generate$ for the first elements of classes 3, 1, and 2. The $Generate$ calls for the latter two do not do anything because no rules apply to the elements of class 1 and 2. For the first element of class 3, however, the join commutativity rule is applied, and a second element representing switched join arguments is added to class 3. For the first element of class 4, the sort push-down rule is applied and two new elements are created, one of which is added to class 4 and one of which becomes the only element of class 5. Finally, the join commutativity rule is applied to the second element of class 4, yielding the third element in the class.

Note the types at the bottom right of each element. The first elements of classes 1–3 have equivalence type M only, because the base relations are retrieved from the DBMS, and we do not know in which order the DBMS will deliver them. It may actually happen that a subquery whose top element is the first element of class 3, when run twice, would return relations that are only multiset equivalent.

The third element of class 4 is only $\equiv_M$ equivalent to the other two elements of that class. Since the query requires a sorted result (the *OrderRequired* property value for the top operator is True), only the two first elements of class 4 will be used during plan search. Below, we discuss how the element types are determined.

During the search-space generation, new elements are added after applying transformation rules. For a transformation rule, we give below a procedure for how to set the types of elements resulting from the right-hand side of the rule.

1. The top-element type (the element representing the top operator in the right-hand side of the rule) is set to the type which is the *greatest common descendant* of the transformation-rule type and the types of the elements participating in the left-hand side of the transformation rule.

2. The top-element type is set to a stronger type than specified in 1 only if the right-hand side expression contains an operation—such as sorting or duplicate elimination—that would enforce a "stronger" equivalence between the new top element and the old top element.

3. The types of other new elements resulting from the right-hand side of the rule are set to any value, but they have to be equal to or stronger than the top-element type.

For example, the greatest common descendant of types M and SM is SS. Let us consider the search-space in Figure 7: the join commutativity rule applied to the second element of class 4 results in the third element of class 4, and its type is set to M, which is the greatest common descendant of L (the type of the second element) and M (the type of the rule).

Now let us consider another query, which performs a selection on relation r transferred to the middleware and then sorts it; see its search space in Figure 8(a). After transformation rule $sort_A(\sigma_P(r)) \rightarrow_L \sigma_P(sort_A(r))$ is applied to the first element of class 3, the new top element—which becomes the second

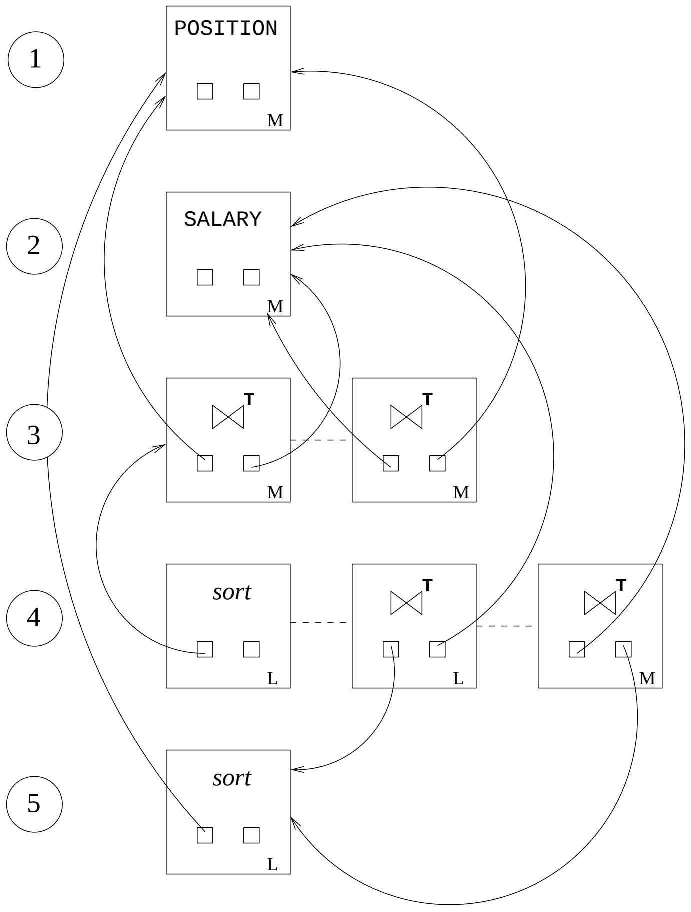

Figure 7: Search Space

element of class 3—is of type L (Figure 8(b)). Even if the sorting is not at the top level, the result is correctly ordered because the selection retains the order of its argument.

In the given examples, the types of the new non-top elements are set to L. Generally, the types of non-top elements are not important for the correctness, as long as they are not descendants of the new top element type (see the equivalence-type ordering in Figure 5). Therefore, they should be set aiming to have as small search space as possible, i.e., if an element has to be inserted, first we can look in the existing search space if the same element (with any type) exists there, and if it does, we do not need to insert it anew. If no elements exist, a new element should have L type, because most rules are of $\rightarrow_L$ type and it is likely that, if this element is to be attempted to be inserted again as a top-level element, its type will be L.

The presented type setting procedure has been included in the $MatchRule$ function, between lines 4 and 5 (Figure 2).

3.3 Modification of the Plan Search

For the actual search, the code controlling the validity of elements depending on their type has to be added to Volcano. Figure 9 outlines the modified pseudocode for the $FindBestPlan$ and $Optimize$ functions; the added or modified lines are denoted by $*$.

The most significant change is the addition of properties to the parameter list of each function. The $FindBestPlan$ function uses its input properties to check the validity of its input element, as mentioned in

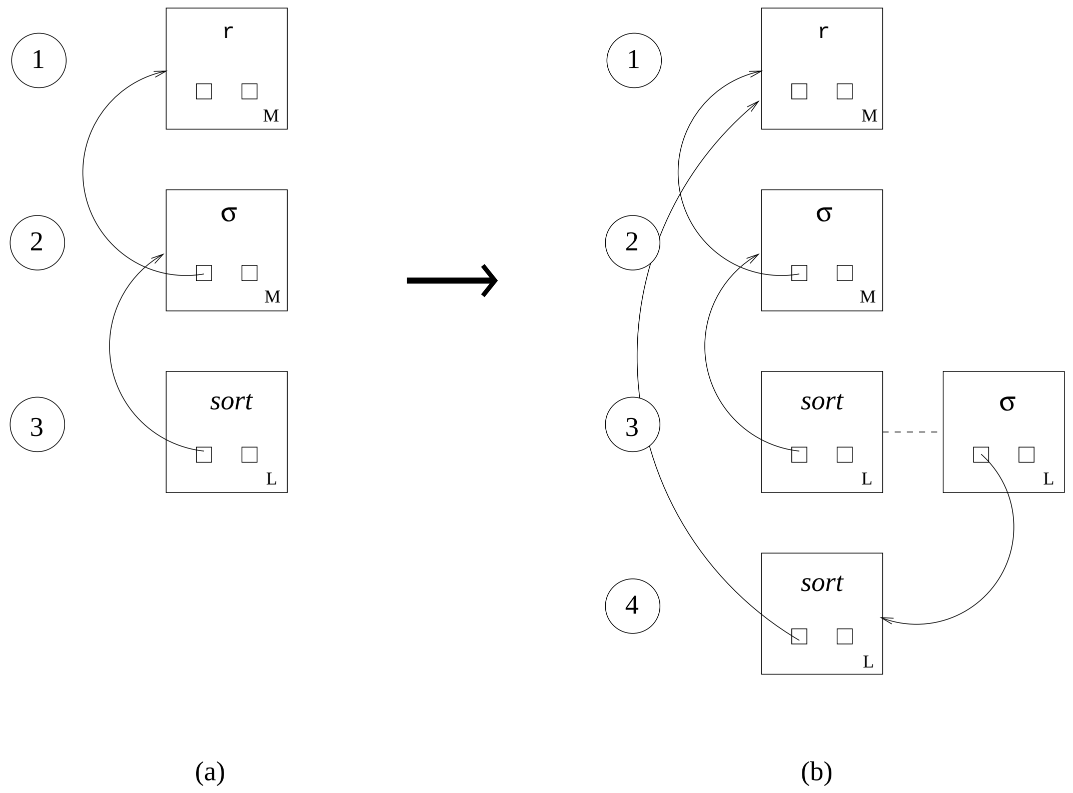

Figure 8: Search Space Before (a) and After (b) Applying $sort_A(\sigma_P(r)) \rightarrow_L \sigma_P(sort_A(r))$

Section 3.2 (the first two lines of $FindBestPlan$). The $Optimize$ function uses its input properties when invoking $FindBestPlan$ on inputs of its input element, to set the property values to be passed as parameters to $FindBestPlan$ (the new line between lines 5 and 6 of $Optimize$).

Since equivalence-class elements might be of any of the six different types, each equivalence class may have up to six physical plans, because plans for different-type elements might differ. For example, it is likely that a type M plan will be simpler and less costly than a type L plan. In the $FindBestPlan$ function, we now have to check for a plan of a type that is stronger than or equal to the input-element type (line 2). In the $Optimize$ function, the best plan is recorded for the type of the input element only (lines 9–11).

4 Experiences

In this section, we consider the extensibility of Volcano in relation to the needs of our framework. First, we evaluate its support for multiple types of equivalence. Then, we discuss other extensions that either required additional implementation effort or could be subject to future improvements of Volcano. Finally, we evaluate the user-friendliness of Volcano, outline the accomplished implementation tasks, and discuss their difficulty.

4.1 Support for Multiple Types of Equivalences

When considering multiple types of equivalences, sorting, duplicate elimination, and coalescing are important operations, because they may change the equivalence type between two relations. For example, if two

```
*    $FindBestPlan$(element $e$, cost $cost\_limit$, boolean $prop$):
*        **if** $prop$ invalidate $e$ due to $type(e)$
*            **return** NULL
1        **if** $completed(class(e)) =$ TRUE
2*           **if** plan $P$ for $class(e)$ with type stronger than or equal to $type(e)$ exists in memory
3                **return** $P$
4            **else**
5                **return** NULL
6        **else**
7            **return** $Optimize$(e, cost_limit)


*    $Optimize$(element $e$, cost $cost\_limit$, boolean $prop$):
1        $best\_plan$ := NULL
2        $min\_cost$ := $cost\_limit$
3        **for each** algorithm $a$ implementing $operator(e)$
4            $current\_plan$ := $a$ applied to plans of its inputs
5            **for each** input $i$ of $a$
*                determine new properties $new\_prop$ according to $prop$ and $operator(e)$
6*               $input\_plan[i]$ := $FindBestPlan(i, remaining\_cost\ new\_prop)$
7                    /* $remaining\_cost$ is $min\_cost$ minus the cost of $a$ and the cost of plans of previous inputs */
8            **if for each** input $i$, $input\_plan[i] \neq$ NULL
9*               **if** there is no plan $P$ for $class(e)$ and $type(e)$ in memory **or** $cost(current\_plan) < cost(P)$
10                   $best\_plan$ := $current\_plan$
11                   $min\_cost$ := $cost(current\_plan)$
12       **for each** element $x$, $x \in class(e), x \neq e$
13           $tmp\_plan$ := $FindBestPlan(x, min\_cost)$
14           **if** $tmp\_plan \neq$ NULL **and** $cost(tmp\_plan) < min\_cost$
15               $best\_plan$ := $tmp\_plan$
16               $min\_cost$ := $cost(tmp\_plan)$
17       set $completed(class(e))$ to True
18       **return** $best\_plan$
```

Figure 9: Modified Pseudocode for the $FindBestPlan$ and $Optimize$ functions

$\equiv_M$ equivalent relations are sorted on A, their sorted versions will be $\equiv_{L,A}$ equivalent. First, we briefly overview Volcano's support for these operations.

Coalescing and duplicate elimination were not implemented in Volcano, and sorting is supported by the so-called *physical properties* of an equivalence class. The possible use of sorting algorithms (termed *enforcers*) is considered during the second phase (plan search) of query optimization. Physical properties are passed as arguments to the $FindBestPlan$ and $Optimize$ functions, and they allow the optimizer to consider different positions of sort enforcers. The use of physical properties increases the code complexity and size—for each algorithm implementing an operator, the optimizer implementor has to write functions deriving physical properties of the algorithm's inputs, checking whether the algorithm satisfies required physical properties, and finding the physical properties that are required from the algorithm's inputs.

In our approach, we treat sorting, duplicate elimination, and coalescing as all the other operators and exploit them in the search-space generation, not using physical properties. While it may be possible to pursue a direction where sorting, duplicate elimination, and coalescing are all treated as enforcers and employ physical properties, we feel that this treatment would add unnecessary complexity to the framework because, fundamentally, sorting, coalescing, and duplicate elimination are just like other operators, having their transformation rules and statistics-derivation formulas. Treating them as algorithms reduces the number of transformation rules, but the complexity in the plan-search algorithm is greatly increased. In addition, it would be problematic to incorporate the statistics-estimation formulas for duplicate elimination and coalescing.

4.2 Other Useful Extensions

Our implementation has indicated the need for new or better support in a number of other areas.

The two-stage query optimization of Volcano forced us to apply all types of transformation rules during the first stage. If one stage with a top-down plan search and generation had been used, it would have been easier to control the applicability of the different types of rules and, possibly, would have improved performance.

The search strategy of Volcano is fixed, and no mechanisms for extending or changing it are provided. Proposed improvements of Volcano that were not part of the available code include a mechanism for heuristic guidance, where rules can be ordered according to their "promise" [GM93]. Such ordering implies that the rules having the best probability to yield better plans would be applied as soon as possible, reducing the overall plan-search time.

We had to add support for equivalence-class elements that point to their own equivalence classes, because this facility was not available in the code supplied. The pointing to the same class often occurs using different equivalence types. For example, sorted relation r is multiset equivalent to r, yielding to a class with two elements (one for r sorted and one for r) where the first (sorting) points to the same class. In addition, we had to implement the linking of equivalence classes; the linking is needed when we apply a rule to an element of a certain class and find that the resulting element already exists in some other class, meaning that both classes represent the same logical expression.

The cardinality of a relation resulting from some equivalence class is estimated when the class is created, according to the selectivity estimation method of the operator represented in the first element. When a new element is added to the class, the cardinality is not reestimated. However, the new element may represent an operator for which we may have a better method for estimating the cardinality. For example, it is easier to estimate the size of a join, than the size of a Cartesian product followed by a selection and a projection. Therefore, we had to ensure that the initial plan would contain operators with good cardinality estimation methods.

4.3 Ease of Extensibility

The main challenge when implementing an extensible query optimizer is to balance the efficiency and extensibility, and our study indicates that Volcano's main emphasis is put on the first aspect.

Volcano is coded in C and does not follow the object-oriented paradigm, which leads to many interconnected structures, which in turn posed difficulties in figuring out where the structures were defined, initialized, and used. The transformation-rule application code is being generated automatically and does not follow any style guidelines, making it difficult to modify (which was needed when incorporating the necessary modifications in the search-space generation). A lot of arrays and structures have predefined sizes and were not being allocated dynamically, occupying more memory than necessary and providing low scalability. On the other hand, the running times of Volcano (for queries not involving many joins) were quite low and did not impose significant overhead, as demonstrated in the performance experiments in [SJS01b].

The actual implementation tasks, their difficulty, and approximate number of lines of resulting code are summarized in Table 3. We divide the entire implementation effort into three subtasks. The first one, adding support for multiple equivalence types, is the most difficult and it has been described in the previous sections. Yet the amount of resulting code was rather small. The other task was to add new operators, and while it resulted in a substantial amount of code, it was not difficult, after learning Volcano's provided framework for adding new operators and transformation rules. The same applies to the last task of adding new algorithms; there, however, the amount of code was smaller, because we did not use physical properties.

Support functions form the biggest part of the code added by the optimizer implementor and their size is proportional to the number of operators and algorithms implemented. In our case, we implemented relation retrieval, selection, projection, join, sort-preserving join, temporal join, Cartesian product, duplicate elimination, aggregation, temporal aggregation, and two transfer operators. Similar behavior of many of these operators (particularly, in the propagation of catalog information) resulted in a lot of code repetition in corresponding support functions.

Task	**Complexity**	**Lines of Code**
Adding equivalence-type support		
Modifying structures	medium	< 200
Modifying search-space generation	high	< 200
Modifying plan search	high	< 200
Adding new operators		
Coding support functions	medium	~ 2500
Coding management of the three properties	medium	~ 400
Coding transformation rules	medium	~ 2300
Adding new algorithms		
Coding support functions	low	~ 1300
Coding implementation rules	medium	< 200

Table 3: Tasks, Their Complexity, and Amount of Code

5 Related Work

This technical report—which offers some extensions to an earlier, published version [SJ01]—takes its outset in the algebraic framework presented in [SJS01a]. The framework has been validated by implementing it using the Volcano extensible query optimizer and the XXL library of query evaluation algorithms; the architecture, cost and selectivity-estimation formulas, and performance studies have been reported in [SJS01b]. The latter paper did not cover the enhancements to Volcano, which are the foci of this paper.

While to our knowledge, nobody has enhanced existing optimizers with support for sets, multisets, and lists, [BKG93] reports on experiences from building the query optimizer for Texas Instruments' Open OODB system using Volcano. That paper concludes that when building an optimizer, it is essential to have a full optimization framework that includes not only an algebra and its transformations, but also the algorithms implementing the operators, formulas for selectivity estimation and costing, and that Volcano provided such a framework. Among the negative sides of Volcano, the paper mentions that much time was spent on writing support functions that derive catalog information and statistics, as well as cost functions. It was also found that the interface for adding new transformation rules and support functions was far from being user-friendly. We agree with these claims, and we draw additional conclusions concerning the support for multiple equivalence types, as well as selectivity estimation and the staged optimization strategy (see Section 4).

In the rest of this section, we consider the existing extensible query optimizers that are related to Volcano. Volcano evolved from the Exodus optimizer [GW87], which in [CW96] was found to be very general, but inefficient, hard to use, and requiring a lot of additional implementation code from the optimizer implementor. Volcano contributed an efficient search strategy, as well as better abstractions for costs and properties. The Cascades optimization framework [Gra95] was built on top of Volcano, and it provides a clean interface and implementation that makes full use of C++ classes, as well as more closely integrates transformation rules and implementation rules, which are distinct sets in Volcano. It also allows some "guidance" during plan search, making it easier to heuristically control the application of transformation rules. Since it was intended to be used for Microsoft's SQL Server, its code is not available.

One of the main observations behind the EROC toolkit for building query optimizers [McK96] was that Volcano, while saving the optimizer implementor from the task of writing the search function, still required a lot code for manipulating operator arguments and various support functions such as property derivation. Therefore, EROC aimed to provide a set of carefully defined and implemented abstractions (in C++ classes) that would reduce this task. The toolkit is not publically available, and thus it is difficult to estimate how much effort is required to build an optimizer using its components.

The Starburst optimizer [Haa90] uses query graph models for representing queries, and it employs a two-stage optimization. During the query rewrite phase, rules are used to transform one query graph model to another; and during the plan optimization phase, candidate plans are costed and the cheapest one is selected. Starburst technology is used in IBM's DB2 [PLH97].

The OPT++ [KW99] extensible optimizer also uses an object-oriented design with C++ classes to simplify the extension tasks. OPT++ offers a number of search strategies, including "bottom-up" System R-style [Sel79] and the Volcano search strategy; and it can emulate both Starburst and Volcano.

6 Conclusions

A number of extensible query optimizers are available that aim to facilitate changes in query algebras and additions of new functionality. Our study reports on the enhancement of one prominent such extensible query optimizer, Volcano, to support an extended relational algebra, which—in addition to new temporal operators—contains six types of equivalences between relations that lead to six corresponding types of transformation rules. We describe how Volcano's search-space generation and plan-search algorithms were modified in order to support the algebra, and we evaluate the extensibility of Volcano.

The study indicates that support for sets, multisets, and lists is difficult to add to a pre-existing extensible query optimizer—such support should be considered already during the design of an extensible query optimizer. Volcano's two-staged optimization strategy forces the application of all transformation rules, disregarding their type, during the first stage; if the optimization had occurred in a single stage, we speculate that it would have been easier to control the applicability of the different types of rules and that better performance would have resulted. We also found that, for the modifications we considered, Volcano's interface was not always user-friendly and that the amount of code needed to implement support functions was quite substantial. On the other hand, we found Volcano to be a very useful tool that allowed us to validate our algebra in the middleware architecture more quickly than if we would have had to develop our own optimizer.

This study indicates that extensible query optimizers are useful when testing research ideas and building prototypes. We also believe that extensible optimizers, if developed in industrial strength versions, will prove very useful when building middleware systems that focus on specific functionality suitable for applying conventional relational query optimization techniques. The application of extensible technology to middleware systems is a promising research direction. Due to the increasing use of user-defined routines in conventional DBMSs, optimizer extensibility is also important when creating new DBMSs or modifying existing ones. Finally, the study reported upon here indicates that more research is needed in query optimization and processing that offer integrated support for sets, multisets, and lists.

Acknowledgments

We are grateful to Richard Snodgrass, who took part in the research that lead to the query optimization framework, the Volcano-based implementation of which is reported in this paper.

The research reported here was supported in part by the Wireless Information Management network, funded by the Nordic Academy for Advanced Study through grant 000389, by the Danish Technical Research Council through grant 9700780, and by a grant from the Nykredit Corporation.

References

[BKG93] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proceedings of ACM SIGMOD*, Washington, DC, pp. 287–296 (1993).

[Bli99] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Developing a DataBlade for a New Index In *Proceedings of IEEE ICDE*, Sydney, Australia, pp. 314–323 (1999).

[CW96] M. J. Carey and D. J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *Proceedings of VLDB*, Bombay, India, pp. 3–14 (1996).

[CK97] M. J. Carey and D. Kossmann. Processing Top N and Bottom N Queries. *Data Engineering Bulletin*, 20(3):12–19 (1997).

[Cha98] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of ACM PODS*, Seattle, WA, pp. 34–43 (1998).

[Gra95] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18(3):19–29 (1995).

[GW87] G. Graefe and D. J. DeWitt. The Exodus Optimizer Generator. In *Proceedings of ACM SIGMOD*, San Francisco, CA, pp. 160–172 (1987).

[GM93] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE ICDE*, Vienna, Austria, pp. 209–218 (1993).

[Haa90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE TKDE*, 2(1):143–160 (1990).

[Inf] Informix Software. DataBlade Overview. URL: <`www.informix.com/products/options/udo/datablade/`>, current as of March 9, 2001.

[JM99] M. Jaedicke and B. Mitschang. User-Defined Table Operators: Enhancing Extensibility for OR-DBMS. In *Proceedings of VLDB*, Edinburgh, Scotland, pp. 494-505 (1999).

[KW99] N. Kabra and D. J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, 8(1):55–78 (1999).

[McK96] W. J. McKenna, L. Burger, C. Hoang, and M. Truong. EROC: A Toolkit for Building NEATO Query Optimizers. In *Proceedings of VLDB*, Bombay, India, pp. 111–121 (1996).

[Ora] Oracle Technology Network. Overview of PL/SQL. URL: <`otn.oracle.com/tech/pl_sql/`>, current as of March 9, 2001.

[PLH97] H. Pirahesh, T. Y. C. Leung, and W. Hasan. A Rule Engine for Query Transformation in Starburst and IBM DB2 c/s DBMS. In *Proceedings of IEEE ICDE*, Birmingham, UK, pp. 391-400 (1997).

[Sel79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD*, Boston, MA, pp. 23–34 (1979).

[SJ01] G. Slivinskas and C. S. Jensen. Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types. In *Proceedings of ADBIS*, Vilnius, Lithuania, pp. 55–69 (2001).

[SJS01a] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. *IEEE TKDE*, 13(1):21–49 (2001).

[SJS01b] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable Query Optimization and Evaluation in Temporal Middleware. In *Proceedings of ACM SIGMOD*, pp. 127–138 (2001).