Bringing Order to Query Optimization

Giedrius Slivinskas† Christian S. Jensen† Richard T. Snodgrass‡

† Department of Computer Science, Aalborg University, Denmark

‡ Department of Computer Science, University of Arizona, USA

Abstract

A variety of developments combine to highlight the need for respecting order when manipulating relations. For example, new functionality is being added to SQL to support OLAP-style querying in which order is frequently an important aspect. The set- or multiset-based frameworks for query optimization that are currently being taught to database students are increasingly inadequate.

This paper presents a foundation for query optimization that extends existing frameworks to also capture ordering. A list-based relational algebra is provided along with three progressively stronger types of algebraic equivalences, concrete query transformation rules that obey the different equivalences, and a procedure for determining which types of transformation rules are applicable for optimizing a query. The exposition follows the style chosen by many textbooks, making it relatively easy to teach this material in continuation of the material covered in the textbooks, and to integrate this material into the textbooks.

1 Introduction

The relational model was originally conceived as a set-based model—relations were sets of tuples. Over the past three decades, this property has been proclaimed a strength as well as a shortcoming of the relational model.

As a reflection of this controversy, the user-level relational query language of choice, SQL, has long offered a peculiar mix of orderedness and unorderedness. To illustrate, an ORDER BY clause permits to sorting of the tuples resulting from query based on any combination of their attributes, using ascending and descending orderings. However, this clause is far from a first class citizen in SQL. Rather, this clause is an add-on that may be used only at the outermost level of a query, for ordering the result. For example, it is impossible to create a view that includes the ORDER BY clause.

With the more recent advent of on-line analytical processing, the ordering of query results had gained new interest and prominence. For example, ordered top-n lists are often of interest in OLAP-style querying. Web search has also led to proposals for exploiting order. Specifically, it is desirable to compute the results of a search in an ordered, page-by-page fashion. This often affords fast computation of the first results and often avoids computation of the entire results.

Developments such as these have led to additions to SQL. For example, Microsoft's SQL Server [Mic] offers a TOP N clause that, when specified with an integer argument N in the SELECT clause of a query, limits the number of tuples returned by the query to at most N. When used in conjunction with the ORDER BY clause, the first N tuples of the result according to the specified order are returned. The Oracle DBMS enables TOP N queries by providing a pseudo-column ROWNUM that assigns rank values to rows according to a given ORDER BY clause [OraDev]. IBM's DB2 supports the clauses "FETCH FIRST N ROWS ONLY" and "OPTIMIZE FOR N ROWS"; the first returns the first N rows, while the second asks the optimizer to deliver the first N rows faster than the rest [IBM].

Whether to allow duplicates in query results, or to insist on relations indeed being sets, has generated much discussion. Informed scientists and practitioners have conducted heated debates on this topic in trade magazines (not unlike in nature to the debates on null values!). This debate has been resolved in the sense that SQL *does* allow duplicates, and query optimization frameworks *have* emerged that consider relations as multisets and thus afford a systematic treatment of duplicates.

However, SQL remains mainly a set-oriented (or multiset-oriented!) language, with order being an add-on. This is perhaps the reason why the handling of order in query optimization is also in some sense an add-on. Query optimization frameworks formalize relations as either sets or multisets, making it difficult to capture, and formally reason about, order.

We believe that, like duplicates, order should be afforded fully integrated treatment in query optimization. The reasons are several. First, order is inherent to the physical representation of data—order thus occurs at the bottom of query plans, which may be exploited to produce better query plans. Second, systematic, unconstrained reasoning about order throughout query plans, e.g., when the queries involve TOP-N like clauses, may lead to better plans.

This paper offers a foundation for relational query optimization that offers comprehensive, sound, and integrated coverage of duplicates and ordering. The foundation is enabled by a relational algebra on relations that are defined as lists and thus can be equivalent as sets, multisets, or lists. These types of equivalences come into play because queries specify different types of results. For example, an SQL query not including ORDER BY and DISTINCT at the outermost level specifies a result of type multiset, thus rendering the application of transformations that need not preserve list equivalence.

The paper provides transformation rules that satisfy the different equivalences and go beyond the existing sets of rules known to the authors. In addition, a practical procedure is offered for determining when a type of transformation rule is applicable to a query.

Some work has been reported on relational algebras for multisets [Alb91, DGK82, GUW00], with the most recent of these, by Garcia-Molina et al., being also the most extensive. This book offers comprehensive coverage of query transformations that preserve set as well as multiset equivalences. Formalizing relations as multisets, sorting is permitted only at the outermost level. However, pushing down sorting in a query plan can improve performance. Moreover, in some cases, the sorting *must* be performed early in the query evaluation. For example, DBMSs such as Microsoft SQL Server allow the ORDER BY clause in combination with the TOP predicate in subqueries, thus requiring intermediate results to be sorted.

Recent work by Pirahesh et al. [PLH97] emphasizes the importance of considering duplicates in DB2's query rewrite rules. However, duplicates are addressed as special cases when defining rewrite rules, and no formal foundation for reasoning about these is offered. Query optimizers such as Volcano [GMc93] initially generate search spaces of query plans without considering ordering, then take order into account when considering the specific operator algorithms to use when transforming a (logical) query plan into a concrete plan that may be executed by the query processor.

Some research has been conducted on algebraic frameworks for queries on lists. Richardson [Ric92] uses an approach based on temporal logic to incorporate lists into an object-oriented data model. Seshadri et al. [SLR94, SLR95] propose a sequence data model and optimization techniques for sequence queries; while the model is relationally complete, the focus is on the processing of operators specific to sequence data such as time series. Our work aims to simplify and minimize the extensions to the conventional relational algebra, as well as permit the treatment of relations as multisets or sets, when order is not important.

Carey and Kossmann [CK97] discuss how to efficiently process TOP N and BOTTOM N queries by extending existing relational query processing architectures, and they propose a number of possible optimizations for such queries. These optimizations fit into this paper's foundation as specific transformation rules.

Our earlier work [SJS01] presented a foundation for temporal query optimization including conventional query optimization that covered duplicates and order, as well as different types of transformation rules. All definitions omitted from this paper are included in that paper, which also covers some additional related work. The present paper considers only conventional query optimization, adds the TOP N operation and consequent transformation rules, and makes the argument that ordered relations should be treated systematically in query optimizers and textbooks.

Section 2 proceeds to define the extended relational algebra. The different types of algebraic equivalences are described in Section 3, and the concrete transformation rules that obey these are provided in Section 4. Section 5 gives a procedure for determining when a transformation rule is applicable, and Section 6 concludes the paper.

2 An Extended Algebra

To formally capture duplicates and ordering, the algebra to be defined must be based on relations that are lists. Because it is also necessary to treat relations as sets or multisets, the semantics of the algebra operations must follow the conventional relational algebra.

It is also desirable that the operations be minimal and orthogonal—each operation should perform one single function and should minimally affect its argument(s) in doing so. This way, replication of functionality is avoided, and it is easier to combine operations in queries. Combinations of operations, termed *idioms*, may be included for efficiency, but should be identified as idioms.

We proceed to define the algebra, then exemplify the algebra and discuss pertinent properties.

2.1 Database Structures

We define relation schemas, tuples, and relation schema instances in turn. The definitions are the standard ones, but adapted to address duplicates and order.

Definition 2.1 A *relation schema* is a four-tuple $S = (\Omega, \Delta, dom, K)$, where Ω is a finite set of attributes, Δ is a finite set of domains, $dom : \Omega \rightarrow \Delta$ is a function that associates a domain with each attribute, and K is a set of sets of attributes from Ω. □

Consider relation PAYMENT in Figure 1. Relation schema PAYMENT consists of the attributes EmpID and Salary and is formally a four-tuple (Ω, Δ, dom, K), where $\Omega = \{\texttt{EmpID}, \texttt{Salary}\}$, $\Delta = \{\text{number}\}$, $dom = \{(\texttt{EmpID}, \text{number}), (\texttt{Salary}, \text{number})\}$, and $K = \{\{\texttt{EmpID}\}\}$; K is essentially a set of keys for the schema.

PAYMENT

EmpID	Salary
1	100K
2	80K
3	130K
4	110K
5	110K

Figure 1: Relation PAYMENT

Definition 2.2 A *tuple over schema* $S = (\Omega, \Delta, dom, K)$ is a function $t : \Omega \rightarrow \cup_{\delta \in \Delta} \delta$, such that for every attribute A of Ω, $t(A) \in dom(A)$. A *relation schema instance over* S is a finite sequence of tuples over S such that for any tuples t_1, t_2 and for any set of attributes $\{A_1, \ldots, A_n\}$ in K, $t_1(A_1) \neq t_2(A_1) \vee \ldots \vee t_1(A_n) \neq t_2(A_n)$. □

Note that the definition of a relation schema instance (relation, for short) corresponds to the definition of a list. A relation can thus contain duplicate tuples, and the ordering of the tuples is significant. The PAYMENT relation from Figure 1 is then the list $\langle t_1, t_2, t_3, t_4, t_5 \rangle$, where $t_1 = \{(\texttt{EmpID}, \texttt{1}), (\texttt{Salary}, \texttt{100K})\}$ and tuples t_2–t_5 correspond to the other tuples of the figure.

2.2 Algebra Operations

We proceed to define the algebra operations. In the definitions, we use $\mathcal{T}$ to be the set of all tuples of any schema and $\mathcal{R}$ to be the set of all relations, and let $r \in \mathcal{R}, r = \langle t_1, t_2, \ldots, t_n \rangle$. We use λ-calculus for the definitions. The definitions do not imply actual implementation algorithms. The schema of the result relation is the same as the schema of the argument relation unless noted otherwise.

Selection The selection operation $\sigma : [\mathcal{R} \times \mathcal{P}] \rightarrow \mathcal{R}$ corresponds to the well-known selection operation in the relational algebra [GUW00]. The argument predicate P from the set of all possible selection predicates $\mathcal{P}$ is expressed as a subscript, i.e., $\sigma_P(r)$.

$$\begin{aligned} \sigma \triangleq \lambda r, P.(r = \perp) \rightarrow r, \\ (tail(r) = \perp) \rightarrow (P(head(r)) \rightarrow head(r), \perp), \\ (P(head(r)) \rightarrow head(r), \perp) \mathbin{@} \sigma_P(tail(r)) \end{aligned}$$

The arguments of an operation are given before the dot, and the definition is given after the dot. In this definition, the first line says that if r is empty (we denote an empty relation by $\perp$), the operation returns it. Otherwise, the second line is processed, which says that if r contains only one tuple (the remaining part of the relation, $tail(r)$, is empty), we test the predicate P on the first tuple ($head(r)$). If the predicate holds, the operation returns the tuple; otherwise, it returns an empty relation. If the second-line condition does not hold, the operation returns the first tuple or an empty relation (depending on the predicate), with the result of the operation applied to the remaining part of r appended (@).

The standard auxiliary functions *head*, *tail*, @, and tuple concatenation (∘)—as well as the other auxiliary functions used below—are defined elsewhere [SJS99].

Projection In the projection operation $\pi : [\mathcal{R} \times \mathcal{F} \times \ldots \times \mathcal{F}] \rightarrow \mathcal{R}$, $\mathcal{F}$ is a set of arithmetic expressions $f_i : \mathcal{T} \rightarrow \mathcal{T}$, which includes any possible attribute names and which return single-attribute tuples. For the PAYMENT relation, one possible expression f_i is $2 \cdot \texttt{Salary AS X}$. Functions $f_1, \ldots, f_n$ are expressed as a subscript, i.e., $\pi_{f_1,\ldots,f_n}(r)$.

$$\begin{aligned} \pi \triangleq \lambda r, f_1, \ldots, f_n.(r = \perp) \rightarrow r, \\ f_1(head(L_1)) \circ \ldots \circ f_n(head(L_1)) \\ \mathbin{@} \pi_{f_1,\ldots,f_n}(tail(r)) \end{aligned}$$

The schema of the result relation follows from the definition of tuple concatenation.

We also define a foreign key below (for simplicity, foreign keys are defined at the instance level).

Definition 2.3 A set of attributes $\{A_1, \ldots, A_n\}$ of relation schema instance r_1 constitute a *foreign key of relation schema instance* r_1 *with respect to a key* $\{B_1, \ldots, B_n\}$ *of relation schema instance* r_2 if and only if $\pi_{A_1,\ldots,A_n}(r_1) \subseteq \pi_{B_1,\ldots,B_n}(r_2)$. □

Union-all Operation $\sqcup : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}$ returns the union of two argument relations, *retaining duplicates*. The operation appends the second relation to the first one.

$$\begin{aligned} \sqcup \triangleq \lambda r_1, r_2.(r_1 = \perp) \rightarrow r_2, \\ head(r_1) \mathbin{@} (tail(r_1) \sqcup r_2) \end{aligned}$$

Cartesian Product Operation $\times : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}$ computes the Cartesian product of two argument relations in nested loop fashion. The definition uses the auxiliary function $Loop : [\mathcal{T} \times \mathcal{R}] \rightarrow \mathcal{R}^{sn}$. The schemas resulting from $\times$ and $Loop$ follow from the definition of tuple concatenation.

$$\begin{aligned} \times \triangleq \lambda r_1, r_2.((r_1 = \perp) \vee (r_2 = \perp)) \rightarrow \perp, \\ Loop(head(r_1), r_2) \sqcup (tail(r_1) \times r_2) \end{aligned}$$

$$\begin{aligned} Loop \triangleq \lambda t, r.(r = \perp) \rightarrow \perp, \\ (t \circ head(r)) \mathbin{@} Loop(t, tail(r)) \end{aligned}$$

Informally, nested-loop join is a nested-loop Cartesian product followed by a selection involving attributes from both arguments of the Cartesian product, and, possibly, followed by a projection.

Difference Operation $\setminus : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}$ returns all tuples of the first argument relation that are not in the second argument relation.

$$\begin{aligned} \setminus \triangleq \lambda r_1, r_2.((r_1 = \perp) \vee (r_2 = \perp)) &\rightarrow r_1, \\ \mathit{isIn}(\mathit{head}(r_1), r_2) &\rightarrow \\ (\mathit{tail}(r_1) \setminus \mathit{remove}(\mathit{head}(r_1), r_2)&), \\ \mathit{head}(r_1) \mathbin{@} (\mathit{tail}(r_1) \setminus r_2&) \end{aligned}$$

Function *isIn* returns True if the argument tuple exists in the argument relation, and function *remove* removes the first occurrence of the argument tuple from the argument relation.

Duplicate Elimination Operation $\mathit{rdup} : \mathcal{R} \rightarrow \mathcal{R}$ removes duplicates from the argument relation. This operation retains the first occurrence of each tuple and removes all subsequent occurrences, if any.

$$\begin{aligned} &\mathit{rdup} \triangleq \lambda r.(r = \perp) \rightarrow r, \\ &\quad \mathit{isIn}(\mathit{head}(r), \mathit{tail}(r)) \rightarrow \\ &\qquad \mathit{rdup}(\mathit{head}(r) \mathbin{@} \mathit{remove}(\mathit{head}(r), \mathit{tail}(r))), \\ &\quad \mathit{head}(r) \mathbin{@} \mathit{rdup}(\mathit{tail}(r)) \end{aligned}$$

If the first tuple of the argument relation can be found in the remaining part of the relation, the operation removes that found tuple. Otherwise, the operation returns the first tuple concatenated with the result of the operation applied to the remaining part of the relation.

Aggregation Operation $\xi : [\mathcal{R} \times \Omega \times \ldots \times \Omega \times \mathbb{F} \times \ldots \times \mathbb{F}] \rightarrow \mathcal{R}$ performs aggregation according to given grouping attributes and aggregation functions. The set of attributes in the schema of the argument relations is denoted by Ω, and the set of all aggregation functions is denoted by $\mathbb{F}$; an aggregate function $F_i : \mathcal{R} \rightarrow \mathcal{T}$ takes a relation as argument and returns a single-attribute tuple containing the aggregate value. An example of an aggregate function is `AVG(Salary) AS D`.

The operation returns one tuple for each unique sequence of grouping attributes. The schema of the result relation follows from the definition of concatenation. Our definition corresponds to that provided by Klug [Klu82] and Garcia-Molina et al. [GUW00].

$$\begin{aligned} &\xi \triangleq \lambda r, g_1, \ldots, g_n, F_1, \ldots, F_m.(r = \perp) \rightarrow r, \\ &\quad (\mathit{head}(r).g_1 \circ \ldots \circ \mathit{head}(r).g_n \\ &\qquad \circ F_1(\mathit{GetGroup}_{g_1,\ldots,g_n}(r, \mathit{head}(r))) \circ \ldots \\ &\qquad \circ F_m(\mathit{GetGroup}_{g_1,\ldots,g_n}(r, \mathit{head}(r)))) \\ &\qquad @\ \xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(r \\ &\qquad\qquad \setminus \mathit{GetGroup}_{g_1,\ldots,g_n}(r, \mathit{head}(r))) \end{aligned}$$

The definition uses the auxiliary function *GetGroup*, which returns all tuples from the argument relation that have grouping-attribute values equal to those of the argument tuple. If there are no grouping attributes, the function returns a list with all tuples of the relation.

Sorting Operation $\mathit{sort} : [\mathcal{R} \times \mathcal{O}_\Omega] \rightarrow \mathcal{R}$ sorts the argument relation. We denote the set of all possible orders for attributes from Ω by $\mathcal{O}_\Omega$. The list $\langle(\mathtt{A}, \mathtt{ASC}), (\mathtt{B}, \mathtt{DESC})\rangle$ is an example of an order. First, we define auxiliary function $\mathit{InsertTuple} : [\mathcal{T} \times \mathcal{R} \times \mathcal{O}_\Omega] \rightarrow \mathcal{R}$, which inserts a tuple into a sorted argument relation, maintaining its order. We denote the argument order by a.

$$\begin{aligned} \mathit{InsertTuple} \triangleq \lambda t, r, a.(r = \perp) &\rightarrow \langle t \rangle, \\ \mathit{MustPrecede}(t, \mathit{head}(r), a) &\rightarrow t \mathbin{@} r, \\ \mathit{head}(r) \mathbin{@} \mathit{InsertTuple}(t, &\mathit{tail}(r), a) \end{aligned}$$

Function *MustPrecede* returns True if the first argument tuple precedes the second argument tuple according to the argument order. Function *sort* invokes *InsertTuple* for each of its tuples.

$$\begin{aligned} &\mathit{sort} \triangleq \lambda r, a.(r = \perp) \rightarrow \perp, \\ &\qquad \mathit{InsertTuple}(\mathit{head}(r), \mathit{sort}(\mathit{tail}(r)), a) \end{aligned}$$

Top Operation $\mathit{top} : [\mathcal{R} \times \mathcal{N}] \rightarrow \mathcal{R}$ returns the first n tuples of the argument relation, where n belongs to a set of natural numbers, $\mathcal{N}$.

$$\begin{aligned} &\mathit{top} \triangleq \lambda r, n.(r = \perp \vee n = 0) \rightarrow \perp, \\ &\qquad \mathit{head}(r) \mathbin{@} \mathit{top}_{n-1}(\mathit{tail}(r)) \end{aligned}$$

2.3 Example Query

Having defined these operations, we exemplify their use in query plans, as well as indicate what kinds of transformations may be applied during optimization.

Let us consider two relations, `PAYMENT` (recall Figure 1) and `EMPLOYEE` (see Figure 2), and a query which asks to list all employees (their IDs and names) with salaries that are among the top three highest salaries in the company, and requires the result to be sorted on the `Salary` attribute in descending order. Note that the result (given in Figure 2) contains more than three tuples, because several employees get the same salary.

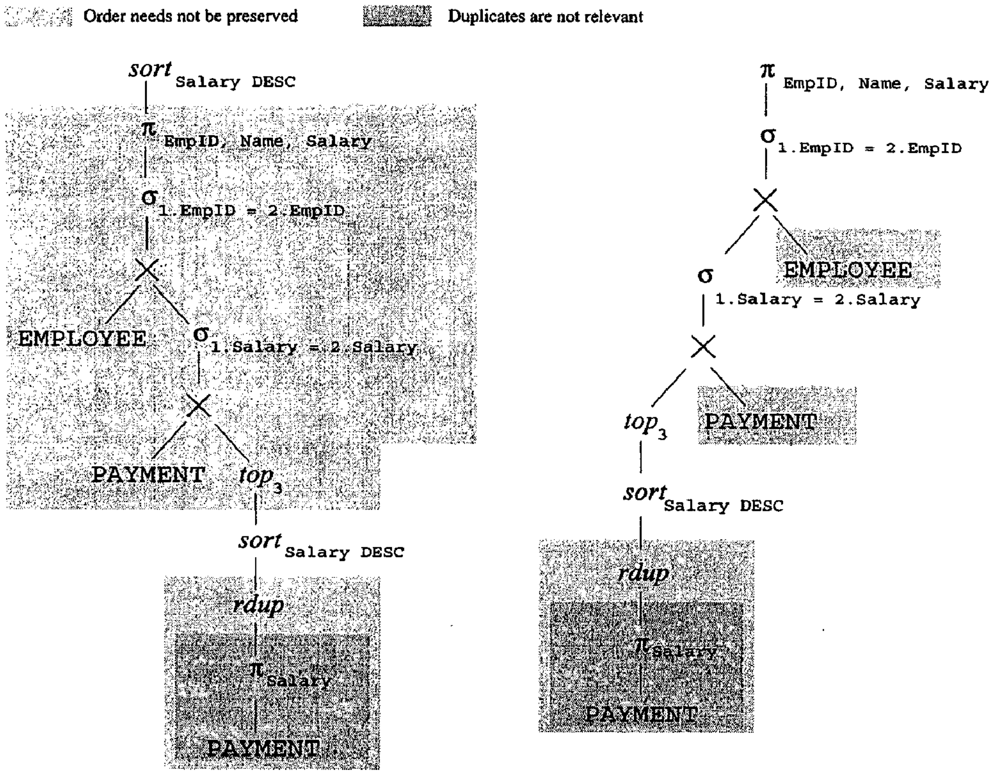
EMPLOYEE

EmpID	Name
1	John
2	Tom
3	Peter
4	Anna
5	Suzanne

Result

EmpID	Name	Salary
3	Peter	130K
4	Anna	110K
5	Suzanne	110K
1	John	100K

Figure 2: Relation `EMPLOYEE` and the Result Relation



(a)

(b)

Figure 3: Initial Query Plan (a) and Resulting Query Plan (b)

Figure 3(a) shows one possible initial query plan. First, the PAYMENT relation is projected on its Salary attribute, then duplicates are removed, and the top three salaries are selected. The Cartesian product and the subsequent selection then find the IDs of all employees that receive a top three salary, and another Cartesian product with a selection is performed on the result and the EMPLOYEE relation in order to obtain the employees' names. Finally, the result is projected on required attributes (for brevity, we do not specify from which relation the common attributes come) and sorted on the Salary attribute.

Transformation rules that preserve different types of equivalences are applicable to different parts of a query. This is illustrated by the regions in Figure 3(a). Transformations below the top *sort* operation and above the *top* operation need not preserve order (indicated by the lighter shading). The top *sort* operation ensures that the result is correctly ordered. Transformations performed below the *rdup* operation need not preserve duplicates, which is indicated by the darker shading.

By systematically exploiting transformation rules preserving different types of equivalences, we are able to achieve an "optimized" query tree such as the one shown in Figure 3(b). In this tree, the orders of the Cartesian products have been switched, so that the left-most relation is the PAYMENT relation projected on the top three salaries. Since the Cartesian product is defined in nested-loop fashion, the order of its left argument is preserved, and, consequently, the top *sort* operation is no longer necessary.

Note that the *rdup*, *sort*, and *top* operations do not have to be separate operations. Since they could be efficiently implemented using a priority heap in main memory, an idiom involving the three operations may be defined and used in query-plan generation.

2.4 Operation Properties

Section 2.2 defined only fundamental operations. The addition of derived operations (idioms), e.g., join (Cartesian product followed by selection and projection) and regular SQL union (union-all followed by duplicate elimination), would not introduce any new issues in the framework. However, idioms should be included in an implementation of the algebra.

The algebra differs fundamentally from the algebra presented in [GUW00], in that this latter algebra works on multisets, not lists. However, all our operations except *top* are list-insensitive, i.e., if their argument relations are identical as multisets (but different as lists), their result relations are also identical as multisets. When we treat relations as multisets, our algebra is at least as expressive as the one presented in [GUW00] because each operation defined there may be expressed by combinations of the first seven operations defined in Section 2.2.

Most operations—such as selection, Cartesian product, difference, duplicate elimination, and *top*—retain the order of their (left) argument. Since the operation definitions constrain the orders of their results, an operation from the conventional relational algebra with several implementation algorithms may result in several operations being added to our algebra. For example, separate definitions are needed for nested-loop join and sort-merge join, since both return differently ordered results.

The projection result is ordered on the largest prefix of its argument order that contains the projected attributes. For example, if we project relation r, which is sorted on $\langle$(A, ASC), (B, ASC), (C, DESC)$\rangle$, on A and C, the result would be sorted on A. Similarly, the result of aggregation is ordered by the largest prefix of its argument order that contains the grouping attributes. The result of sorting is the order specified by the sorting parameter if the latter is not a prefix of the argument's order, and the argument's order otherwise. The result of union-all is unordered,

An operation may (1) eliminate duplicates so that the result would only have distinct tuples, (2) retain duplicates, i.e., the result would have distinct tuples *only* if the argument relation(s) contains only distinct tuples, or (3) may generate duplicates in the result even if duplicates do not exist in the argument relation(s). Duplicate elimination and aggregation eliminate duplicates; and selection, Cartesian product, difference, sorting, and *top* retain duplicates. Projection generates duplicates only if the projection attributes do not contain a key of the argument relation, and union-all always generates duplicates.

3 Relation Equivalences

The query optimizer does not always need to consider relations as lists. For example, if ORDER BY is not specified in a query, it is enough to consider relations as multisets. To enable this type of treatment of relations, three types of equivalences between relations are introduced: list equivalence ($\equiv_L$), multiset equivalence ($\equiv_M$), and set equivalence ($\equiv_S$). Two relations are list equivalent if they are identical; multiset equivalent if they are identical as multisets, taking into account duplicates, but not order; and set equivalent if they are identical as sets, ignoring duplicates and order.

Definition 3.1 Let functions $\equiv_L$, $\equiv_M$, and $\equiv_S$ be given, all with signature $[\mathcal{R} \times \mathcal{R}] \rightarrow$ Boolean. Relations r_1 and r_2 are *list equivalent* ($r_1 \equiv_L r_2$), *multiset equivalent* ($r_1 \equiv_M r_2$), and *set equivalent* ($r_1 \equiv_S r_2$) if and only if function $\equiv_L$, $\equiv_M$, and $\equiv_S$ return True, respectively.

$$\begin{aligned}\equiv_L \ \triangleq\ & \lambda r_1, r_2.(r_1 = \perp \wedge r_2 = \perp) \rightarrow \text{True},\\ & (r_1 = \perp \oplus r_2 = \perp) \rightarrow \text{False},\\ & \quad (\mathit{head}(r_1) = \mathit{head}(r_2)) \rightarrow \mathit{tail}(r_1) \equiv_L \mathit{tail}(r_2),\\ & \quad \text{False}\end{aligned}$$

$$\begin{aligned}\equiv_M \ \triangleq\ & \lambda r_1, r_2.(r_1 = \perp \wedge r_2 = \perp) \rightarrow \text{True},\\ & (r_1 = \perp \oplus r_2 = \perp) \rightarrow \text{False},\\ & \quad \mathit{isIn}(\mathit{head}(r_1), r_2) \rightarrow\\ & \qquad\qquad \mathit{tail}(r_1) \equiv_M \mathit{remove}(\mathit{head}(r_1), r_2),\\ & \quad \text{False}\end{aligned}$$

$$\begin{aligned}\equiv_S \ \triangleq\ & \lambda r_1, r_2.(r_1 = \perp \wedge r_2 = \perp) \rightarrow \text{True},\\ & (r_1 = \perp \oplus r_2 = \perp) \rightarrow \text{False},\\ & \quad \mathit{isIn}(\mathit{head}(r_1), r_2) \rightarrow\\ & \qquad \mathit{RmAll}(\mathit{head}(r_1), r_1) \equiv_S\\ & \qquad\qquad \mathit{RmAll}(\mathit{head}(r_1), r_2),\\ & \quad \text{False} \qquad\qquad \square\end{aligned}$$

Auxiliary function *RmAll* removes all occurences of the argument tuple from the argument relation and returns the resulting relation.

We can exemplify different types of equivalences using different variations of the PAYMENT relation (Figures 1 and 4). Relations PAYMENT and PAYMENT$_A$ are not equivalent as lists because the tuple ordering is different, but they are equivalent as multisets and sets. Relations PAYMENT$_A$ and PAYMENT$_B$ are equivalent only as sets, because the tuple for employee ID 3 is repeated twice in PAYMENT$_B$.

PAYMENT$_A$

EmpID	Salary
2	80K
1	100K
3	130K
4	110K
5	110K

PAYMENT$_B$

EmpID	Salary
1	100K
2	80K
3	130K
3	130K
4	110K
5	110K

Figure 4: Variations of the PAYMENT Relation

The examples illustrate that we have an ordering between the types of equivalences. Two relations being equivalent as multisets implies that they are also equivalent as sets, and two relations being equivalent as lists implies that they are equivalent as both multisets and sets.

The different types of equivalences can be exploited in heuristics-based query optimization. Transformation rules (to be discussed in detail shortly) can be divided into three categories, one for each type of equivalence. For example, we may have a rule $\mathit{expr}_1 \rightarrow_L \mathit{expr}_2$, which says that after the replacement of expression expr_1 in the original query plan by expression expr_2, the result relation produced by the new plan will be list equivalent to the result relation produced by the original plan, when evaluated on the same argument relation(s). That said, the result relations will also be multiset and set equivalent.

Another rule $expr_1 \rightarrow_M expr_3$ says that if we replace $expr_1$ by $expr_3$, the new plan will yield a result relation that may only be multiset equivalent to the result relation produced by the original plan, because the application of this rule does not preserve the order. This may be acceptable though, if the result needs to be a multiset. For example, query $\pi_{\mathtt{Salary}}(\mathtt{PAYMENT})$ can return tuples in any order. In general, the type of the result specified by a query determines which transformation rules can be exploited. The next two sections list transformation rules and describe when they are applicable.

4 Transformation Rules

In this section, we provide an extensive set of transformation rules for the algebra. First, we provide rules that derive from the conventional relational algebra. Then we discuss rules involving the duplicate elimination, sorting, and *top* operations.

The rules are given as equivalences that express that two algebraic expressions are equivalent according to one of the three equivalence types from Section 3; we always give the strongest equivalence type that holds. An algebraic equivalence represents both a left-to-right and a right-to-left transformation rule. If necessary, we mark pre-conditions that apply only for the left-to-right transformation by [lr] and pre-conditions that apply only for the right-to-left transformation by [rl]. Pre-conditions with no such marks apply to both directions. All rules can be verified formally, as the operations and equivalence types have formal definitions. We believe the transformations are correct; reference [SJS99] provides an example proof of one transformation rule.

In transformation rules, r can be a base relation or an operation tree. We denote the attribute domain of the schema of relation r by Ω_r. Function *attr* returns the set of attributes present in a selection predicate, projection functions, or a sorting list.

4.1 Conventional Rules

The conventional transformation rules derive from the rules for multisets given by [GUW00]; we list them in Figure 5. The rules are ordered based on the operation they concern, e.g., rules C1–C4 concern selection.

Most rules satisfy the list equivalence, but the commutativity rules, e.g., for Cartesian product and union-all, satisfy only the $\equiv_M$ equivalence because the result relations produced by the left- and right-side expressions have differently ordered tuples (see rules C8 and C16). Finally, rule C2 only satisfies $\equiv_S$ equivalence because if both predicates P_1 and P_2 are satisfied for a tuple of r, the right-hand side of the transformation would return two instances of the same tuple.

(C1) $\sigma_{P_1 \wedge P_2}(r) \equiv_L \sigma_{P_1}(\sigma_{P_2}(r))$
(C2) $\sigma_{P_1 \vee P_2}(r) \equiv_S \sigma_{P_1}(r) \sqcup \sigma_{P_2}(r)$
(C3) $\sigma_{P_1}(\sigma_{P_2}(r)) \equiv_L \sigma_{P_2}(\sigma_{P_1}(r))$
(C4) $\sigma_{\neg P}(r) \equiv_L r \setminus \sigma_P(r)$

(C5) $\pi_{f_1,\ldots,f_n}(\pi_{h_1,\ldots,h_m}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(r)$
[lr] $attr(f_1, \ldots, f_n) \subseteq \Omega_r$
[rl] $attr(h_1, \ldots, h_m) \subseteq \Omega_r$
(C6) $\pi_{f_1,\ldots,f_n}(\sigma_P(r)) \equiv_L \sigma_P(\pi_{f_1,\ldots,f_n}(r))$
[lr] $attr(P) \subseteq attr(f_1, \ldots, f_n)$
(C7) $\pi_{f_1,\ldots,f_n}(\sigma_P(r)) \equiv_L \pi_{f_1,\ldots,f_n}(\sigma_P(\pi_{h_1,\ldots,h_m}(r)))$,
where $h_i = \{a \mid i \in \{1, \ldots, m\}$
$\wedge (h_i \in \{f_1, \ldots, f_n\} \vee h_i \in attr(P))\}$
[rl] $attr(P) \subseteq \Omega_r$

(C8) $r_1 \times r_2 \equiv_M r_2 \times r_1$
(C9) $\sigma_P(r_1 \times r_2) \equiv_L \sigma_P(r_1) \times r_2$
[lr] $attr(P) \subseteq \Omega_{r_1}$
(C10) $\sigma_P(r_1 \times r_2) \equiv_L r_1 \times \sigma_P(r_2)$
[lr] $attr(P) \subseteq \Omega_{r_2}$
(C11) $\pi_{f_1,\ldots,f_n}(r_1 \times r_2) \equiv_L \pi_{A_1}(r_1) \times \pi_{A_2}(r_2)$, where
$A_1 = \{f_i \mid i \in \{1, \ldots, n\} \wedge attr(f_i) \subseteq \Omega_{r_1}\}$,
$A_2 = \{f_i \mid i \in \{1, \ldots, n\} \wedge attr(f_i) \subseteq \Omega_{r_2}\}$
[lr] $\forall i \in \{1, \ldots, n\}\ attr(f_i) \subseteq \Omega_{r_1} \vee attr(f_i) \subseteq \Omega_{r_2}$
[rl] $attr(A_1) \cap attr(A_2) = \emptyset$
(C12) $\pi_{f_1,\ldots,f_n}(r_1 \times r_2) \equiv_L \pi_{f_1,\ldots,f_n}(\pi_{A_1}(r_1) \times \pi_{A_2}(r_2))$,
where $A_1 = \{a \mid a \in \Omega_{r_1} \wedge a \in attr(f_1, \ldots, f_n)\}$,
$A_2 = \{a \mid a \in \Omega_{r_2} \wedge a \in attr(f_1, \ldots, f_n)\}$
[rl] $attr(f_1, \ldots, f_n) \subseteq \Omega_{r_1 \times r_2}$
(C13) $(r_1 \times r_2) \times r_3 \equiv_L r_1 \times (r_2 \times r_3)$

(C14) $\sigma_P(r_1 \setminus r_2) \equiv_L \sigma_P(r_1) \setminus r_2$
(C15) $\sigma_P(r_1 \setminus r_2) \equiv_L \sigma_P(r_1) \setminus \sigma_P(r_2)$
(C16) $r_1 \sqcup r_2 \equiv_M r_2 \sqcup r_1$
(C17) $\sigma_P(r_1 \sqcup r_2) \equiv_L \sigma_P(r_1) \sqcup \sigma_P(r_2)$
(C18) $\pi_{f_1,\ldots,f_n}(r_1 \sqcup r_2) \equiv_L \pi_{f_1,\ldots,f_n}(r_1) \sqcup \pi_{f_1,\ldots,f_n}(r_2)$

(C19) $\sigma_P(\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(r)) \equiv_L \xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(\sigma_P(r))$
$attr(P) \subseteq \{g_1, \ldots, g_n\}$
(C20) $\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(r) \equiv_L \xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(\pi_H(r))$
$attr(g_1, \ldots, g_n, F_1, \ldots, F_m) \subseteq H$

Figure 5: Conventional Rules

4.2 Duplicate Elimination Rules

Figure 6 lists duplicate elimination rules. Rules D1–D2 indicate when duplicate elimination is not necessary. Rule D6 follows because aggregations involving only functions MIN and MAX are insensitive to duplicates.

Duplicate elimination cannot be pushed before union-all because the latter may generate duplicates even if its arguments do not contain any. Also, duplicate elimination cannot be pushed down before difference, because difference is sensitive to the number of duplicates in both arguments. If tuple t occurs x times in the first argument and y times in the second argument ($y < x$), it occurs $x - y$ times in the result. However, if we were to remove duplicates first, tuple t would occur only once in each argument to the difference, and it would be absent from the result.

(D1) $rdup(r) \equiv_L r \qquad r$ does not have duplicates
(D2) $rdup(r) \equiv_S r$
(D3) $rdup(\sigma_P(r)) \equiv_L \sigma_P(rdup(r))$
(D4) $rdup(\pi_{f_1,\ldots,f_n}(rdup(r))) \equiv_L rdup(\pi_{f_1,\ldots,f_n}(r))$
(D5) $rdup(r_1 \times r_2) \equiv_L rdup(r_1) \times rdup(r_2)$
(D6) $\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(rdup(r)) \equiv_L \xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(r)$
$\qquad AggrFs(F_1,\ldots,F_m) \subset \{\texttt{MIN},\texttt{MAX}\}$

Figure 6: Duplicate Elimination Rules

If duplication elimination is applied after an operation that does not manufacture duplicates, we can remove the duplicate elimination using rule D1. Thus, duplicate elimination can be removed if it is performed on top of duplicate elimination or aggregation.

4.3 Sorting Rules

Sorting can be eliminated if performed on a relation that already satisfies the sorting, if we can treat the relation as multiset, or if there is a subsequent sorting operation. Predicate *IsPrefixOf* takes two lists as argument and returns True is the first is a prefix of the second. The sorting rules are given in Figure 7. Function $Order(r)$ returns a list of attributes paired with a sorting type (ascending or descending) for a relation r, for example, $Order(r) = \langle(\texttt{A},\texttt{ASC}),(\texttt{B},\texttt{DESC})\rangle$. For an unordered relation, the function returns an empty list.

(S1) $sort_A(r) \equiv_L r \qquad IsPrefixOf(A, Order(r))$
(S2) $sort_A(r) \equiv_M r$
(S3) $sort_A(sort_B(r)) \equiv_L sort_A(r)$
$\qquad IsPrefixOf(B, A)$
(S4) $sort_A(\sigma_P(r)) \equiv_L \sigma_P(sort_A(r))$
(S5) $sort_A(\pi_{f_1,\ldots,f_n}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(sort_A(r))$
$\qquad [\texttt{lr}]\ attr(A) \subseteq \Omega_r$
$\qquad [\texttt{rl}]\ attr(A) \subseteq attr(f_1,\ldots,f_n)$
(S6) $sort_A(r_1 \times r_2) \equiv_L sort_A(r_1) \times r_2$
$\qquad [\texttt{lr}]\ attr(A) \subseteq \Omega_{r_1}$
(S7) $sort_A(r_1 \setminus r_2) \equiv_L sort_A(r_1) \setminus r_2$
(S8) $sort_A(\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(r)) \equiv_L$
$\qquad \xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}(sort_A(r))$
$\qquad attr(A) \subseteq \{g_1,\ldots,g_n\}$
(S9) $sort_A(rdup(r)) \equiv_L rdup(sort_A(r))$

Figure 7: Sorting Rules

If we wish to sort the result of some operation, the sorting can be performed on the argument relation(s) for that operation if the operation preserves the ordering. All operations except $\sqcup$ fully or partially preserve the ordering of their first argument.

4.4 TOP N Rules

Rules for the *top* operation are given in Figure 8. Several rules have applicability conditions involving the cardinality of the argument relations. These rules can only be applied if the exact cardinality is known, i.e., if the cardinality is only estimated, these rules are not applicable.

(T1) $top_n(r) \equiv_L r \qquad n(r) \leq n$
(T2) $top_n(\pi_{f_1,\ldots,f_n}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(top_n(r))$
(T3) $top_n(r_1 \times r_2) \equiv_L top_n(top_n(r_1) \times r_2)$
(T4) $top_n(r_1 \times r_2) \equiv_L top_n(r_1 \times top_n(r_2))$
(T5) $top_n(\sigma_P(r_1 \times r_2)) \equiv_L \sigma_P(top_n(r_1) \times r_2)$
$\qquad ((A_1 = B_1 \wedge \ldots \wedge A_n = B_n) \in P)$
$\qquad \wedge \{A_1,\ldots,A_n\}$ is a foreign key of r_1
$\qquad \wedge \{B_1,\ldots,B_n\}$ is a key of r_2
(T6) $top_n(r_1 \sqcup r_2) \equiv_L top_n(r_1) \qquad n(r_1) \geq n$
(T7) $top_n(r_1 \sqcup r_2) \equiv_L r_1 \sqcup top_{n'}(r_2) \qquad n(r_1) + n' = n$

Figure 8: TOP N Rules

5 Applicability of Transformation Rules

Queries expressed in SQL are mapped to an initial algebraic expression, to which the optimizer then applies transformation rules according to some strategy. The resulting, new algebraic expressions must, when evaluated, return relations that are equivalent to the relation returned by the original expression, which we assume correctly computes the user's query. The type of equivalence required between result relations depends on the actual query statement; we name the required equivalence between results the *outer equivalence* and assign it to the root of the query tree.

For SQL queries, the outer equivalence is $\equiv_M$ or $\equiv_{L,A}$ [1], depending on whether the query given includes `ORDER BY A`. The presence of `ORDER BY` specifies a list; otherwise, the query specifies a multiset, rendering order of the result tuples immaterial. Intuitively, we can apply transformation rules to a query evaluation plan if the result relations produced by the new plan and the original plan are equivalent as multisets or lists, depending on whether or not `ORDER BY` was specified.

Having the outer equivalence, we can derive the required equivalence for each operation in the query tree. Due to the different characteristics of operations, an operation somewhere in the query tree may require an equivalence that is not the same as the outer equivalence. For example, in the query tree shown in Figure 3(a), the outer equivalence is $\equiv_L$, but the operations between the top *sort* operation and the *top* operation do not need to preserve order; hence, $\equiv_M$ rules are applicable.

The required equivalences constrain the types of transformation rules that can be applied during query plan enu-

[1] Two relations are $\equiv_{L,A}$ equivalent if they are $\equiv_M$ equivalent and their projections on attribute list A are $\equiv_L$ equivalent. The $\equiv_{L,A}$ equivalence is slightly less restrictive than $\equiv_L$; the $\equiv_L$ equivalence implies the $\equiv_{L,A}$ equivalence.

meration. There are no restrictions on rules of type $\equiv_L$ — these can always be applied safely because a transformed expression evaluates to a result identical as a list to that obtained from evaluating the original expression.

To enable the formal procedure of determining when a transformation rule is applicable to a query plan, we introduce properties for the operations in an operation tree.

5.1 Definitions of Properties

Table 1 introduces two Boolean properties of operations of a query tree. For each combination of the property values, Table 2 gives an equivalence type that should hold for results of that operation. A transformation rule of some type can be applied at some location in a query tree if the result produced by its right-hand side is equivalent to the result produced by its left-hand side according to the required equivalence type, as specified by the properties for the top-most operation at that location. For example, rule G8 guarantees only the $\equiv_M$ equivalence between its right- and left-hand side, but it can be applied to the query plan in Figure 3(a) to both Cartesian products because the required equivalence at each location is $\equiv_M$ (the *OrderReq* property value is False).

Property Name	**Description**
OrderReq	True if the result of the operation must preserve some ordering
DupRelevant	True if the operation cannot arbitrarily add or remove duplicates

Table 1: Properties of an Operation in an Operation Tree

OrderReq(*op*)	*DupRelevant*(*op*)	Type
True	True or False	$\equiv_{L,A}$
False	True	$\equiv_M$
False	False	$\equiv_S$

Table 2: Combinations of Property Values and Corresponding Equivalence Types

During query optimization, the properties are first set for the initial query evaluation plan. For the root, the *OrderReq* property is set to True only if the ORDER BY clause is specified at the outer-most level of the user query, and the *DupRelevant* property is always set to True. Then, the two properties are propagated down the tree from the root.

Table 3 defines the *DupRelevant* property values for a non-root operation *op*. This property depends almost entirely on the parent of the operation, denoted op_p, and it is independent of the specific *op*. For binary operations, keywords *left* and *right* denote the location of *op* relative to its parent. If this property holds at the parent, it also holds at a child, except: (1) when the parent operation is difference, the operation in question is located at the right child, and the relation produced by the left child does not contain duplicates; (2) when the parent operation is duplicate elimination, because then the child operation may deal with duplicates in any way, since they will later be removed; and (3) when the parent operation is aggregation with only the duplicate-insensitive aggregation functions (MIN and MAX).

op_p	$DupRelevant(op)$
$\sigma_P, \pi_{f_1,\ldots,f_n}$, $\sqcup$ (*left* and *right*), $\times$ (*left* and *right*), $sort_A$, top_n	$DupRelevant(op_p)$
$\setminus$ (*left*)	True
$\setminus$ (*right*)	$MayHaveDups(op_{left}(op_p))$
rdup	False
$\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}$	False **if** $AggrFs(F_1,\ldots,F_m) \subset \{\texttt{MIN},\texttt{MAX}\}$ True **otherwise**

Table 3: The *DupRelevant* Property

To set the property for the right child of difference, an auxiliary property *MayHaveDups* is used, which tells if the relation produced by the child operation may contain duplicates. This property is propagated bottom-up from the base relations using the duplicate-preservation properties of operations as described in Section 2.4.

The next case to consider is when the property does not hold at the parent. Then, the property holds at a child in the following situations: (1) when the parent operation is difference, the operation in question is located at the left child, or it is located at the right child, and the relation produced at the left child does not contain duplicates; and (2) when the parent operation is aggregation with at least one duplicate-sensitive function (AVG, SUM, or COUNT).

Table 4 describes the propagation of the *OrderReq* property. This property also depends almost entirely on the parent of the operation. Most often, the *OrderReq* property holds for an operation at a child node when it holds for the operation at the parent node and the parent node operation preserves the order of its argument. For example, if order is required for a select operation (σ), then order will be required of the immediate child of that operation. However, if the parent operation is *sort*, the property does not hold for its immediate child because the order of the argument is immaterial. In contrast, if the parent operation is *top*, the property holds for its immediate child because the order of the argument is important.

When a transformation rule is applied during query optimization, the properties must be adjusted. The top-down

op_p	$OrderReq(op)$
$\sigma_P, \pi_{f_1,\ldots,f_n}$, × (*left*), $\setminus$ (*left*), *rdup*, $\xi_{g_1,\ldots,g_n,F_1,\ldots,F_m}$	$OrderReq(op_p)$
$\sqcup$ (*left* and *right*), × (*right*), $\setminus$ (*right*), $sort_A$	False
top_n	True

Table 4: The *OrderReq* Property

nature of property definitions ensures that adjustments for most of the rules are local, i.e., it is not necessary to scan the whole operation tree [SJS01].

6 Summary

With the advent of on-line analytical processing and the use of database technology in Internet search, the ordering of query results has gained new interest and prominence. Thus, TOP-N like queries have received increased attention in the user community, and major DBMS vendors have included support for such queries into their products over the past few years. However, order is far from a first-class citizen in query optimization, where relations are often viewed as sets or multisets. In contrast, we believe that, like duplicates, order should be afforded fully integrated treatment in query optimization.

This paper presents a foundation for relational query optimization that offers comprehensive and precise handling of duplicates and order. This is enabled by a list-based algebra where relations thus can be equivalent as sets, multisets, or lists. This leads to three types of transformation rules that can be exploited during query optimization, depending on whether the ORDER BY or DISTINCT clauses are specified in an SQL query. In addition, a procedure is offered for determining when a rule of some type is applicable to a query tree. This foundation proposes to handle the sorting and *top* operations as all the other algebra operations during the search-space generation.

While the foundation proposed here may readily be integrated into database textbooks so that students get exposed to the issues related to duplicates and order, much research and engineering remains to be done to reflect the foundation in an efficient query optimizer.

References

[Alb91] J. Albert. Algebraic Properties of Bag Data Types. In *Proc. VLDB*, pp. 211–219 (1991).

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

[DGK82] U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *Proc. PODS*, pp. 117–123 (1982).

[GMc93] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. IEEE ICDE*, pp. 209–218 (1993).

[GUW00] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall (2000).

[IBM] DB2 Universal Database and DB2 Connect for Windows, OS/2 and Unix. Administration Guide. `<www-4.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/document.d2w/report?fn=db2v7d0frm3toc.htm>`, current as of August 2, 2001.

[Kie85] W. Kiessling. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. In *Proc. VLDB*, pp. 241–249 (1985).

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *JACM*, 29(3): 699–717 (1982).

[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 *Proc. IEEE ICDE*, pp. 391–400 (1997).

[Mic] Microsoft SQL Server Product Documentation. `<www.microsoft.com/sql/techinfo/productdoc/2000/>`, current as of July 27, 2001.

[OraDev] Oracle8i Application Developer's Guide - Fundamentals. `<technet.oracle.com/doc/server.815/a68003/toc.htm>`, current as of July 27, 2001.

[Ric92] J. Richardson. Supporting Lists in a Data Model (A Timely Approach). In *Proc. VLDB*, pp. 127–138 (1992).

[SJS99] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. TIMECENTER TR-49 (1999). `<www.cs.auc.dk/TimeCenter>`, current as of July 27, 2001.

[SJS01] 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).

[SLR94] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence Query Processing. In *Proc. ACM SIGMOD*, pp. 430–441 (1994).

[SLR95] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proc. IEEE ICDE*, pp. 232–239 (1994).