Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering

Giedrius Slivinskas Christian S. Jensen
Department of Computer Science
Aalborg University
{*giedrius, csj*}*@cs.auc.dk*

Richard T. Snodgrass
Department of Computer Science
University of Arizona
rts@cs.arizona.edu

Abstract

Most real-world database applications contain a substantial portion of time-referenced, or temporal, data. Recent advances in temporal query languages show that such database applications could benefit substantially from built-in temporal support in the DBMS. To achieve this, temporal query representation, optimization, and processing mechanisms must be provided. This paper presents a general, algebraic foundation for query optimization that integrates conventional and temporal query optimization and is suitable for providing temporal support both via a stand-alone temporal DBMS and via a layer on top of a conventional DBMS. By capturing duplicate removal and retention and order preservation for all queries, as well as coalescing for temporal queries, this foundation formalizes and generalizes existing approaches.

1. Introduction

Most real-world database applications manage time-referenced data. For example, this aspect applies to financial, medical, and travel applications; and being time-variant is one of Inmon's defining properties of a data warehouse [11]. Recent advances in temporal query languages [8, 13] show that such applications may benefit substantially from a DBMS with built-in temporal support. The potential benefits are several: application code is simplified and more easily maintainable, thereby increasing programmer productivity [21], and more data processing can be left to the DBMS, potentially leading to better performance.

In contrast, the built-in temporal support offered by current database products is limited to predefined, time-related data types, e.g., the Informix TimeSeries DataBlade and the Oracle8 TimeSeries cartridge, and extensibility facilities that enable the user to define new, e.g., temporal, data types. However, temporal support is needed that goes beyond data types and extends the query language itself.

Developing a DBMS with built-in temporal support from scratch is a daunting task that may only be accomplished by major DBMS vendors that already have a DBMS to modify and have large resources available. This has led to the consideration of a layered, or *stratum*, approach where a layer, implementing temporal support, is interposed between the applications and a conventional DBMS [3, 22]. The layer maps temporal SQL statements to regular SQL statements and passes them to the DBMS, which is not altered. With this approach, it is feasible to support a temporal SQL that strictly extends SQL, thus not affecting legacy applications.

This paper offers a foundation for conventional and temporal query optimization that is applicable to both the integrated and the layered architecture, thus making it relevant for a DBMS vendor that plans to incorporate temporal features into their product, as well as to third-party developers that want to implement a temporal layer on top of a conventional DBMS. The foundation offers comprehensive, precise, and integrated coverage of order preservation and duplicate removal and retention for all queries, as well as of coalescing for temporal queries. (In coalescing, tuples with adjacent time periods and otherwise identical attribute values are consolidated.)

The foundation is enabled by a temporally extended algebra, which enhances existing relational algebras based on multisets by integrating the handling of order and adding temporal support. In addition to conventional relations, the algebra employs temporal relations timestamped with time periods, which are useful for implementation because of their granularity independence and fixed-width format. Previously proposed user-level temporal relations may be mapped to this format [14], and the user-level data model and query language may be point-based or interval-based [4]. More generally, the algebra is independent of the specific user-level temporal relational query language and data model employed, and it provides support for the two main classes of temporal statements found in the literature: (1) statements that use built-in temporal semantics and are evaluated conceptually at each point of time and (2) state-

ments that explicitly manipulate values of (new) temporal abstract data types with convenient operations and predicates defined on them. The temporal aspect considered here is valid time [12], which captures when data was, is, or will be true in the modeled reality; the approach can be extended to also handle transaction time, either alone or in combination with valid time.

In the algebra, relations are defined as lists, and six kinds of relation equivalences are defined. Specifically, two relations can be equivalent as lists, multisets, and sets, and they can be snapshot-equivalent as lists, multisets, and sets. For example, the last type of equivalence occurs when all corresponding pairs of snapshot relations that may be derived from a pair of temporal relations are the same when considered as sets.

These types of equivalences come into play because queries specify different types of results, depending on whether ordering, duplicate removal, or coalescing are specified in the query statement. For example, an SQL query not including `ORDER BY` and `DISTINCT` at the outermost level specifies a result of type multiset, thus opening the possibility of applying transformations that do not preserve list equivalence. The different types of equivalences make it possible to systematically exploit transformation rules and to optimize a query according to the type of the expected result. The paper provides transformation rules that preserve these types of equivalences and describes when a rule of some type is applicable to a query. Finally, an algorithm is provided that generates equivalent query evaluation plans.

Some work has been reported on non-temporal relational algebras for multisets [1, 7, 9], with the most recent of these works [9], 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. But although SQL only allows sorting at the outermost level, pushing down sorting in a query plan can improve performance. By formalizing relations as lists and offering integrated support for query transformations that preserve list equivalences, we allow sorting to be performed early during query evaluation. In addition, we state precisely when list, multiset, and set based equivalences, including their temporal counterparts, are applicable. Recent work on query optimization by Leung et al. [16] 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.

More than a dozen temporal relational algebras have been proposed over the last two decades [18, 19], but all the algebras known to the authors are set-based and hence do not adequately address issues related to duplicates, order, and coalescing. Existing work on temporal query optimization [10, 17] primarily considers the processing of joins and semijoins. It does not delve into general query optimization and does not address duplicates, order, and coalescing.

The paper is structured as follows. Section 2 motivates the need for the proposed foundation for query optimization, defines the underlying database structures, and presents the extended relational algebra operations. The different types of algebraic equivalences are described in Section 3, and the concrete transformation rules that preserve the different equivalence types are provided in Section 4. Sections 5 and 6 describe how to determine when different transformation rules are applicable and provide a query plan enumeration algorithm. Section 7 concludes and offers research directions.

2. An extended algebra

We initially motivate for the proposed query optimization framework. The remainder of this section first describes requirements to the extended algebra, then defines its database structures and the operations on them.

2.1. Example

The example assumes a layered architecture, where the stratum performs some of the query optimization and processing, in addition to translating temporal query language statements to SQL. Specifically, complex temporal operations such as temporal aggregation, temporal duplicate elimination, and coalescing are often not processed efficiently in conventional DBMSs and might advantageously be supported by the stratum.

In the temporal relations `EMPLOYEE` and `PROJECT` in Figure 1, we assume a closed-open representation for time periods and assume the time values to denote months during some year. For example, John is in Sales from January to August (not including the latter), and he is in Advertising from June to November. Consider the query "Which employees worked in a department, but not on any project, and when?" In particular, the user requires the result relation to be sorted, coalesced, and without duplicates in its snapshots. The snapshot of a temporal relation at time t is the conventional relation containing those tuples (without the time periods) from the temporal relation that have time periods containing t.

To exemplify the concepts of coalescing and temporal duplicates (duplicates in snapshots), let us examine the `EMPLOYEE` relation after being projected on `EmpName`, `T1`, and `T2` (see the top-left relation in Figure 3 in Section 2.5). This projected relation is not coalesced; the first and third tuples (and the second and third tuples) for Anna have adjacent time periods and can be merged. Also, it contains temporal duplicates; its snapshot at, e.g., time 6 contains duplicate tuples for John.

EMPLOYEE

EmpName	Dept	T1	T2
John	Sales	1	8
John	Advertising	6	11
Anna	Sales	2	6
Anna	Advertising	2	6
Anna	Sales	6	12

PROJECT

EmpName	Prj	T1	T2
John	P1	2	3
John	P2	5	6
John	P1	7	8
John	P3	9	10
Anna	P2	3	4
Anna	P2	5	6
Anna	P3	7	8
Anna	P3	9	10

Result

EmpName	T1	T2
Anna	2	3
Anna	4	5
Anna	6	7
Anna	8	9
Anna	10	12
John	1	2
John	3	5
John	6	7
John	8	9
John	10	11

Figure 1. Example relations

The desired result of the previous query is given at the bottom-right in Figure 1. We proceed to use this query to illustrate the importance of properly considering duplicates, order, and coalescing during query optimization.

To compute the result, the stratum initially uses a straightforward mapping of the user-level query to an initial algebra expression, shown in Figure 2(a). The query is entirely computed in the DBMS; the last operation applied is a transfer operation T^S that transfers its argument from the DBMS to the stratum. Allowing also a reverse transfer operation, T^D, permits query plans to flexibly partition computation between the stratum and the DBMS.

The next operations, sorting ($sort$), coalescing ($coal^T$), and temporal duplicate elimination ($rdup^T$), are performed to obtain the user-required format. The $rdup^T$ operation ensures that no snapshots have duplicates, and $coal^T$ ensures that value-equivalent tuples (tuples with the same non-temporal attribute values) with adjacent time periods are merged.

The temporal difference ($\backslash^T$) is the central operation in this query. It returns the employees that are present in EMPLOYEE, but not in PROJECT, along with the time periods when this occurred. It turns out that to obtain the correct result, the left argument is not allowed to contain duplicates in snapshots; this is ensured by the $rdup^T$ operation prior to the difference.

Transformation rules that preserve different types of equivalences are applicable to different parts of the query. This is illustrated by the shaded regions in Figure 2(a). First, transformations below the $sort$ need not preserve order. The operations below $sort$ are not sensitive to order, and the $sort$ ensures that whatever result is produced by the operations below, it is correctly ordered at the end. Second, temporal difference is sensitive to duplicates in its left argument, so the lower-left $rdup^T$ may affect the result of the difference. However, the presence or absence of duplicates is not relevant for the operations below this $rdup^T$, as well as for the operations that are on the right branch of the temporal difference. Also, it does not matter if the relation produced by the temporal difference contains duplicates or not, due to the subsequent $rdup^T$ operation. As a result, transformation rules applied to the darkly shaded region need not preserve duplicates. Third, transformations applied below the coalescing operation need not preserve the time periods; coalescing returns a unique relation for all snapshot-equivalent argument relations whose snapshots do not contain duplicates. The top $rdup^T$ ensures that the argument to the coalescing operation does not contain duplicates in snapshots. Sections 5 and 6 elaborate on these concepts and describe when different types of transformation rules are applicable.

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 2(b). In this tree, the transfer operation has been pushed down, indicating that the stratum performs temporal duplicate elimination, coalescing, and difference. The $sort$ operation was pushed down because the DBMS sorts faster than the stratum. The parts of a query relegated to the DBMS (here, those below T^S operations) are not optimized by the stratum; instead these are expressed in the language supported by the DBMS, e.g., SQL, and are then passed to the DBMS, which will perform its own optimization. In the stratum, coalescing is performed before difference because the left argument to the temporal difference is expected to be smaller than the result of the temporal difference.

We use this example throughout the paper and explain in more detail the concepts represented by the shaded regions and the generation of equivalent query trees.

2.2. Requirements

Several requirements should be kept in mind when designing the algebra. It is a fundamental requirement that the algebra be formally defined. Equally fundamental, the algebra must be suitable for implementation, which has several implications. The algebra must incorporate ordering, duplicate removal and retention, and coalescing. This implies that the relations, over which the operations will be defined, should be lists, thereby incorporating both duplicates and order. In addition, it is attractive to use conventional, fixed-size tuples, which implies the use of time periods (as opposed to temporal elements, which are finite unions of time periods). To be independent of the granularity of time, definitions of operations should be expressed in terms of the start and end times of periods only.

The algebra must extend the conventional relational algebra and must accommodate both classes of temporal state-

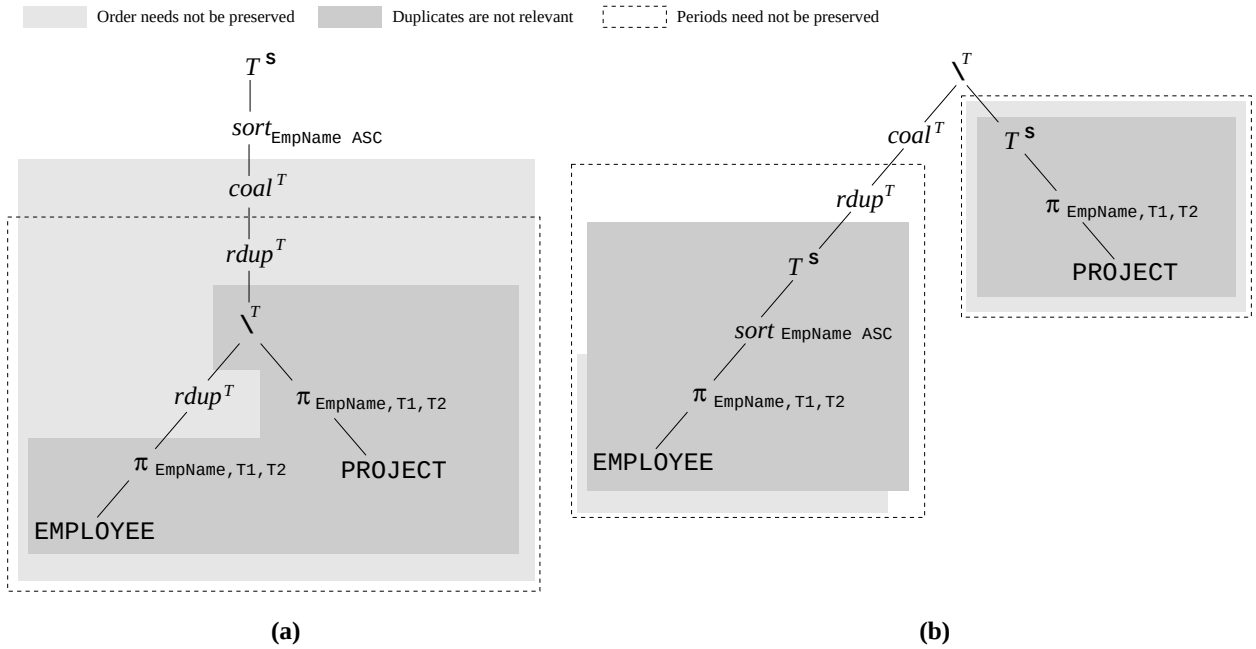

Figure 2. Algebraic expressions for the query

ments mentioned in the introduction, namely statements with built-in temporal semantics and statements that explicitly manipulate values of time data types. To conveniently accommodate the first class of statements, we introduce temporal operations that are counterparts of existing relational algebra operations, in the sense that they are snapshot-reducible to these. A temporal operation op_1 is snapshot-reducible to operation op_2 if for any point in time t and for any temporal relation r, the snapshot at t of the result of applying op_1 to r is equal to the result of applying op_2 to the snapshot of r at time t. For example, temporal duplicate elimination is snapshot reducible to duplicate elimination.

We also require 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. For example, coalescing should not have any effect on duplicates; a separate duplicate elimination operation should be available for this purpose. As another implication, the operations should retain as much as possible the time periods and the order of the tuples in the argument relation(s). For example, coalescing should retain the ordering of its argument. Combinations of operations, termed idioms, may be included for efficiency, but should be identified as idioms.

2.3. 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 three-tuple $S = (\Omega, \Delta, dom)$, where Ω is a finite set of attributes, Δ is a finite set of domains, and $dom : \Omega \rightarrow \Delta$ is a function that associates a domain with each attribute. □

For example, relation schema EMPLOYEE from Figure 1 is formally a three-tuple (Ω, Δ, dom), where $\Omega = \{\texttt{EmpName}, \texttt{Dept}, \texttt{T1}, \texttt{T2}\}$, $\Delta = \{\text{string}, \mathbb{T}\}$, and $dom = \{(\texttt{EmpName}, \text{string}), (\texttt{Dept}, \text{string}), (\texttt{T1}, \mathbb{T}), (\texttt{T2}, \mathbb{T})\}$. We denote the time domain by $\mathbb{T}$ and use the definition of this domain proposed by, e.g., Bettini et al. [2].

Definition 2.2 A *tuple over schema* $S = (\Omega, \Delta, dom)$ 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. □

Note that the definition of a relation schema instance (relation, for short) corresponds to the definition of a list. A relation can contain duplicate tuples, and the ordering of the tuples is significant. Relation EMPLOYEE from Figure 1 is a list of tuples $\langle t_1, t_2, t_3, t_4, t_5 \rangle$. Tuple t_1 can be expressed as $\{(\texttt{EmpName}, \texttt{John}), (\texttt{Dept}, \texttt{Sales}), (\texttt{T1}, 1), (\texttt{T2}, 8)\}$.

We distinguish between snapshot, or conventional, and temporal relations. We reserve two specific attribute names, T1 and T2, for denoting the time period start and end, respectively, of a temporal relation. The schema of a snapshot relation does not contain these two attributes. Alternatively, we could have chosen to have a single type of relation, but then each temporal operation would have to take the names of the temporal attributes as extra arguments. Using our approach, the operations implicitly know the time attributes.

Operation	Sorting $Order(result)$	Cardinality $n(result)$	Duplicates	Coalescing
$\sigma_P(r)$	$= Order(r)$	$\leq n(r)$	Retains	Retains
$\pi_{f_1,\ldots,f_n}(r)$	$= Prefix(Order(r), ProjPairs)$	$= n(r)$	Generates	Destroys
$r_1 \sqcup r_2$	unordered	$= n(r_1) + n(r_2)$	Generates	Destroys
$r_1 \times r_2$	$= Order(r_1)$	$= n(r_1) \cdot n(r_2)$	Retains	—
$r_1 \setminus r_2$	$= Order(r_1)$	$\geq (n(r_1) - n(r_2))$ and $\leq n(r_1)$	Retains	—
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$	$= Prefix(Order(r), GroupPairs)$	$\leq n(r)$	Eliminates	—
$rdup(r)$	$= Order(r)$	$\leq n(r)$	Eliminates	—
$r_1 \times^T r_2$	$= Order(r_1) \setminus TimePairs$	$\leq n(r_1) \cdot n(r_2)$	Retains	Destroys
$r_1 \setminus^T r_2$	$= Order(r_1) \setminus TimePairs$	$\leq 2 \cdot n(r_1)$	Retains	Destroys
$\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$	$= Prefix(Order(r), GroupPairs)$	$\leq 2 \cdot n(r) - 1$	Eliminates	Destroys
$rdup^T(r)$	$= Order(r) \setminus TimePairs$	$\leq 2 \cdot n(r) - 1$	Eliminates	Destroys
$r_1 \cup r_2$	unordered	$\geq n(r_1)$ and $\leq (n(r_1) + n(r_2))$	Retains	—
$r_1 \cup^T r_2$	unordered	$\geq n(r_1)$ and $\leq (n(r_1) + 2 \cdot n(r_2))$	Retains	Destroys
$sort_A(r)$	$= A$	$= n(r)$	Retains	Retains
$coal^T(r)$	$= Order(r) \setminus TimePairs$	$\leq n(r)$	Retains	Enforces

Table 1. Overview of operations

2.4. Fundamental algebra operations

We describe briefly all the fundamental algebra operations. We then consider temporal duplicate elimination in detail. Other operations are defined elsewhere [20].

Table 1 lists all operations. Selection (σ), projection (π), union ALL ($\sqcup$), Cartesian product ($\times$), difference ($\setminus$), aggregation (ξ), and duplicate elimination ($rdup$) derive from the conventional relational algebra. For the latter four operations, we add temporal counterparts, denoted by superscript T. The temporal operations conceptually evaluate the result at each point of time. This is exemplified by the difference between regular and temporal duplicate elimination, to be discussed in Section 2.5.

Next, union ($\cup$) originates from the union operation for multisets given in [1]. This operation includes a tuple in the result as many times as the tuple occurs in the argument relation that has the most occurrences of that tuple. The temporal counterpart of union is denoted by $\cup^T$. We also add coalescing, which merges value-equivalent tuples with adjacent time periods, and sorting. Our definition of coalescing is different from that given by Böhlen et al. [5], due to the requirement of minimality (see Section 2.2) and our relations being list based. The coalescing of Böhlen et al. merges value-equivalent tuples with adjacent or *overlapping* time periods; in our algebra, this effect can be achieved by performing temporal duplicate elimination and coalescing.

The algebra includes fundamental operations as well as the temporal operations needed to accommodate query statements with built-in temporal semantics (see Section 2.2). We omit derived operations (idioms), except regular and temporal union, which can be expressed via union ALL and regular (temporal) difference. The addition of idioms, e.g., join (Cartesian product followed by selection and projection), would not introduce any new issues in the framework. However, idioms should be included in an implementation of the algebra.

Our algebra and the algebra presented in [9] are fundamentally different in that the latter works on multisets, while ours works on lists. However, our selection, projection, Cartesian product, difference, union ALL, aggregation, and duplicate elimination operations are not list-sensitive, 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 algebra presented in [9] because each operation of the latter may be expressed by one of the seven operations mentioned above.

Table 1 also describes, for each operation, the order and cardinality of the result relation and how the operation handles regular duplicates and coalescing. Function $Order(r)$ returns a list of attributes paired with a sorting type (ascending or descending) for relation r (e.g., $\langle$`A ASC, B DESC`$\rangle$). For an unordered relation, the function returns an empty list. Lists $ProjPairs$, $TimePairs$, and $GroupPairs$ include, respectively, the projection attributes, the temporal attributes, and the grouping attributes paired with `ASC` or `DESC`. Function $Prefix$ returns the largest common prefix of its two arguments. For example, if a relation is sorted on `A`, `B`, and `C`, and we project it on `A` and `C`, the result is sorted on `A`. Although omitted from the table, the time attributes may in special cases be present in the order of a relation resulting from coalescing. Also note that in the special case where the sorting list A is a prefix of $Order(r)$, the order of $sort_A(r)$ is $Order(r)$.

We denote the cardinality of relation r by $n(r)$. An oper-

ation may (1) eliminate regular duplicates so that the result relation would only have distinct tuples, (2) retain regular duplicates, i.e., the result relation would have distinct tuples *only* if the argument relation(s) contains only distinct tuples, or (3) generate regular duplicates that do not derive from duplicates existing in the argument relation(s). In a similar manner, an operation may (1) enforce coalescing, so that its result relation is coalesced, (2) retain coalescing, i.e., its result relation is coalesced *only* if its argument relation is coalesced, or (3) destroy coalescing. Note that coalescing is undefined for snapshot relations (which are returned by operations that have temporal counterparts).

The next section defines temporal duplicate elimination. Overall, an attempt has been made to define operations conducive to efficient implementation. For example, union ALL simply concatenates its arguments.

2.5. Temporal duplicate elimination

Let $\mathcal{T}^T$ be the set of all tuples with temporal support, and let $\mathcal{R}^T$ be the set of all relations with such tuples. Operation $rdup^T : \mathcal{R}^T \rightarrow \mathcal{R}^T$ removes duplicates from all snapshots of the argument relation. The argument and result relations have the same schema. Note that this operation also removes regular duplicates because they qualify as duplicates in snapshots.

Figure 3 shows the EMPLOYEE relation projected on $L = \langle \texttt{EmpName}, \texttt{T1}, \texttt{T2} \rangle$ and also the results of regular and temporal duplicate elimination applied to this relation. Relation R2 does not contain regular duplicates (there is only one tuple for Anna with times 2 and 6), and relation R3 does not contain duplicates in snapshots (note the timestamps of the second tuple). Time attributes in R2 are prefixed by "1" because the result of regular duplicate elimination is a snapshot relation and thus cannot include attributes named T1 or T2.

We use λ-calculus for the definitions. The definitions do not imply the actual implementation algorithms, but *do* constrain the implementation algorithms to produce the same results, taking order and duplicates into account. We define temporal duplicate elimination below.

$$
\begin{aligned}
&rdup^T \triangleq \lambda r.(r = \perp \vee tail(r) = \perp) \rightarrow r,\\
&\quad (Over^T(head(r), tail(r)) = undef) \rightarrow\\
&\qquad\qquad head(r) \mathbin{@} rdup^T(tail(r)),\\
&\qquad rdup^T(head(r) \mathbin{@} Change^T(Over^T(head(r), tail(r)),\\
&\qquad\qquad tail(r), r_n))\\
&\text{where } r_n = \langle Over^T(head(r), tail(r)) \rangle \setminus^T \langle head(r) \rangle
\end{aligned}
$$

The arguments to the operation are given before the dot, and the definition is given after the dot. The first line says that if the argument relation r is empty ($\perp$) or its part without the first tuple ($tail(r)$) is empty, the operation returns r. Otherwise, the second line is processed, which says that we apply function $Over^T$ to the first tuple ($head(r)$) and the rest of the relation. Function $Over^T : [\mathcal{T}^T \times \mathcal{R}^T] \rightarrow \mathcal{T}^T$ scans the argument relation and finds the first tuple whose time period overlaps with the argument tuple and which is value-equivalent with it. (For example, the first two tuples of R1 overlap and are value-equivalent.) If there is no such tuple, we return the first tuple concatenated (@) with the result of $rdup^T$ applied to the rest of the relation. Otherwise (the fourth and fifth lines), the operation returns the result of $rdup^T$ applied to the modified argument relation, where the overlapping tuple is changed to the result of subtracting the first tuple of the relation from the overlapping tuple. The result can contain zero, one, or two tuples, depending on how the time periods of the tuples are related. Function $Change^T : [\mathcal{T}^T \times \mathcal{R}^T \times \mathcal{R}^T] \rightarrow \mathcal{R}^T$ finds the argument tuple in the first argument relation, then replaces the tuple with the second argument relation (since the temporal difference may return two tuples, we use "relation" as result type). For example, the time period of the second tuple of R3 is obtained by subtracting the time period of the first tuple of R1 from that of the second tuple of R1.

R1 $= \pi_L$(EMPLOYEE)

EmpName	T1	T2
John	1	8
John	6	11
Anna	2	6
Anna	2	6
Anna	6	12

R2 $= rdup$(R1)

EmpName	1.T1	1.T2
John	1	8
John	6	11
Anna	2	6
Anna	6	12

R3 $= rdup^T$(R1)

EmpName	T1	T2
John	1	8
John	8	11
Anna	2	6
Anna	6	12

Figure 3. Regular and temporal duplicate elimination

3. Relation equivalences

The query optimizer does not always need to operate on relations as lists. For example, if ORDER BY is not specified in a query, it is enough to consider the underlying relations as multisets. To enable such different treatment of relations, we distinguish between six types of equivalences between relations: list equivalence ($\equiv_L$), multiset equivalence ($\equiv_M$), set equivalence ($\equiv_S$), snapshot list equivalence ($\equiv_L^S$), snapshot multiset equivalence ($\equiv_M^S$), and snapshot set equivalence ($\equiv_S^S$). Two relations are equivalent as lists if they are identical lists; as multisets if they are identical multisets taking into account duplicates, but not order; and as sets if they are identical sets, ignoring duplicates and order. Snapshot list equivalence holds between two temporal relations when snapshots of those relations at

each point of time are equivalent as lists. Similar conditions imply snapshot multiset equivalence (at each point in time, the relations should be equivalent as multisets) and snapshot set equivalence (at each point in time, the relations should be equivalent as sets). Formal definitions may be found in the associated technical report [20].

We can exemplify the different types of equivalences using the relations in Figure 3. Relations R1 and R2 are not equivalent as lists or as multisets because the tuple for Anna with times 2 and 6 occurs twice in R1, but once in R2. However, the $\equiv_S$ equivalence holds because the two relations contain the same tuples. Snapshot equivalences between the two relations are undefined because relation R2 is nontemporal.

Relations R1 and R3 have different tuples, e.g., the tuple for John with times 6 and 11 is present in R1, but not in R3; thus, they are not equivalent as lists, multisets, or sets. Their snapshots are also not equivalent as lists or as multisets because the snapshot of R1 at times between 2 and 6 contains two tuples for Anna, while snapshots of relation R3 never contain more than one tuple for Anna. The only equivalence that holds between the two relations is $\equiv_S^S$, meaning that their snapshots are equivalent as sets.

We have an ordering between the types of equivalences. For example, the equivalence $\texttt{R1} \equiv_M sort_{\texttt{T1 ASC}}(\texttt{R1})$ implies that both relations are equivalent as multisets and sets, and that their snapshots are equivalent as multisets and sets. We list all implications in the following theorem.

Theorem 3.1 Let r_1 and r_2 be relations. Then the following implications hold. (Implications pointing downward apply only to temporal relations.)

$$\begin{array}{ccccc} r_1 \equiv_L r_2 & \Rightarrow & r_1 \equiv_M r_2 & \Rightarrow & r_1 \equiv_S r_2 \\ \Downarrow & & \Downarrow & & \Downarrow \\ r_1 \equiv_L^S r_2 & \Rightarrow & r_1 \equiv_M^S r_2 & \Rightarrow & r_1 \equiv_S^S r_2 \end{array}$$

Proof: [20] □

The different types of equivalences can be exploited in query optimization. Transformation rules (to be discussed in Section 4) can be divided into six categories, one for each type of equivalence. For example, we may have a rule $expr_1 \rightarrow_L expr_2$, 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. Another rule $expr_1 \rightarrow_S expr_3$ says that if we replace $expr_1$ by $expr_3$, the new plan will yield to a result relation that may only be set equivalent to the result relation produced by the original plan, because the application of this rule does not preserve either duplicates or the order. This may be acceptable though, if the result needs to be a set. For example, query $rdup^T(\pi_L(\texttt{EMPLOYEE}))$ (resulting in relation R3) can return distinct tuples in any order. In general, the type of the result specified by a query affects which transformation rules can be exploited. Section 4 lists transformation rules, and Sections 5 and 6 describe how to determine when a transformation rule of some type is applicable.

4. Transformation rules

In this section, we describe transformation rules involving conventional operations, duplicate elimination, coalescing, sorting, and transfer operations in turn, listing central rules. The full rule set, which extends all existing rule sets known to the authors, can be found in [20].

The transformation rules are given as equivalences that express that two algebraic expressions are equivalent according to one of the six 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, and it may have pre-conditions. All transformation rules can be verified formally, as the operations and equivalence types have formal definitions. Unlike rules expressed informally, which sometime later have been found to be in error, e.g., in [15], the rules here are theorems with formal proofs.

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

4.1. Conventional transformation rules

Conventional relational algebra rules for multisets [9] differ in how they are extended to support lists and temporal operations. Most rules are valid for lists and have counterparts for the corresponding temporal operations; in some cases, pre-conditions involving the temporal attributes apply. Commutativity rules, e.g., for Cartesian product and union, satisfy only the $\equiv_M$ equivalence because the different order of the arguments leads to differently ordered tuples in the results. A few rules, involving regular and temporal union, have equivalence types weaker than $\equiv_M$.

4.2. Duplicate elimination transformation rules

Rules *D1–D4* in Figure 4 indicate when duplicate elimination is not necessary. Note that if we perform a temporal duplicate elimination on a temporal relation, the result relation is only $\equiv_S^S$ equivalent to the argument relation (compare relations R1 and R3 from Figure 3).

Conventional duplicate elimination rules may be found in [9], and they can easily be extended to lists. The only addition is two new rules for regular and temporal union (see

(D1)	$rdup(r) \equiv_L r$	r does not have duplicates
(D2)	$rdup^T(r) \equiv_L r$	r does not have duplicates in snapshots
(D3)	$rdup(r) \equiv_S r$	
(D4)	$rdup^T(r) \equiv_S^S r$	
(D5)	$rdup(r_1 \cup r_2) \equiv_L rdup(r_1) \cup rdup(r_2)$	
(D6)	$rdup^T(r_1 \cup^T r_2) \equiv_L rdup^T(r_1) \cup^T rdup^T(r_2)$	

(S1)	$sort_A(r) \equiv_L r$	$IsPrefixOf(A, Order(r))$
(S2)	$sort_A(r) \equiv_M r$	
(S3)	$sort_A(sort_B(r)) \equiv_L sort_A(r)$	$IsPrefixOf(B, A)$

(C1)	$coal^T(r) \equiv_L r$	r is coalesced
(C2)	$coal^T(r) \equiv_M^S r$	
(C3)	$coal^T(\sigma_P(r)) \equiv_L \sigma_P(coal^T(r))$	$\texttt{T1} \notin attr(P) \wedge \texttt{T2} \notin attr(P)$
(C4)	$\pi_{f_1,\ldots,f_n}(coal^T(r)) \equiv_S \pi_{f_1,\ldots,f_n}(r)$	$\texttt{T1} \notin attr(f_1,\ldots,f_n) \wedge \texttt{T2} \notin attr(f_1,\ldots,f_n)$
(C5)	$coal^T(coal^T(r_1) \sqcup coal^T(r_2)) \equiv_L coal^T(r_1 \sqcup r_2)$	
(C6)	$coal^T(coal^T(r_1) \cup^T coal^T(r_2)) \equiv_L coal^T(r_1 \cup^T r_2)$	
(C7)	$coal^T(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(coal^T(r))) \equiv_L coal^T(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r))$	
(C8)	$coal^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(coal^T(r))) \equiv_L coal^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(r))$	r does not have duplicates in snapshots
(C9)	$coal^T(\pi_A(r_1 \times^T r_2)) \equiv_L \pi_A(coal^T(r_1) \times^T coal^T(r_2))$, where $A = \Omega_{r_1 \times^T r_2} \setminus \{1.\texttt{T1}, 1.\texttt{T2}, 2.\texttt{T1}, 2.\texttt{T2}\}$	r_1 and r_2 do not have duplicates in snapshots
(C10)	$coal^T(r_1 \setminus^T r_2) \equiv_M coal^T(r_1) \setminus^T coal^T(r_2)$	r_1 does not have duplicates in snapshots

Figure 4. Transformation rules

D5–D6). Contrary to the commonly considered union ALL and regular SQL union (which removes duplicates from the result relation of union ALL) operations, our regular and temporal union operations do not generate new duplicates if their argument relations do not contain any duplicates, which means that we can push duplicate elimination below regular or temporal union.

4.3. Coalescing transformation rules

Rules *C1* and *C2* show when we can eliminate coalescing; rule *C1* can be used to derive other transformation rules that eliminate superfluous coalescing. Rule *C3* says that coalescing and selection commute only if the selection predicate does not involve the temporal attributes. If we project a coalesced relation on non-temporal attributes, coalescing is not necessary if we consider the relations as sets (rule *C4*). For a number of operations, coalescing their arguments and results is equivalent to coalescing their results only (rules *C5–C7*).

Our list of coalescing transformations extends the list provided by Böhlen et al. [5]. Due to the differences in coalescing definitions (see Section 2.4) and because [5] allows duplicates in snapshots of temporal relations, but not regular duplicates, the following three transformation rules (given in [5]) have only type $\equiv_M^S$ and are derivable from rule *C2*.

$coal^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(coal^T(r))) \equiv_M^S coal^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(r))$
$coal^T(\pi_A(r_1 \times^T r_2)) \equiv_M^S \pi_A(coal^T(r_1) \times^T coal^T(r_2))$,
where $A = \Omega_{r_1 \times^T r_2} \setminus \{1.\texttt{T1}, 1.\texttt{T2}, 2.\texttt{T1}, 2.\texttt{T2}\}$
$coal^T(r_1 \setminus^T r_2) \equiv_M^S coal^T(r_1) \setminus^T coal^T(r_2)$

The transformation rules have $\equiv_M^S$ type because projection, Cartesian product, and temporal difference destroy coalescing. The projection in the second rule is necessary because the temporal Cartesian product retains the timestamps of its argument relations [20].

The first two transformations can be modified to have type $\equiv_L$ if we require that the arguments do not have duplicates in snapshots (rules *C8–C9*). Adding the same requirement, the third rule can be modified to have type $\equiv_M$ (rule *C10*). Equivalence type $\equiv_L$ cannot be achieved because temporal difference is sensitive to the distribution of value-equivalent tuples in the left argument; and this distribution may be different for r_1 and $coal(r_1)$. Note that since periods need not be preserved in the right argument to temporal difference, the second coalescing on the right-hand side of the rule is not necessary. However, in cases when coalescing significantly reduces the cardinality of its argument, it might be useful to retain it.

4.4. Sorting transformation rules

Sorting can be eliminated if it is performed on a relation that is already sorted as desired, if we can treat the relation as a multiset, or if there is a subsequent sorting operation (rules *S1–S3*). Predicate $IsPrefixOf$ takes two lists as arguments and returns True if the first list is a prefix of the second one. Transformation rule *S3* requires B to be a prefix of A. If A is a prefix of B, we can eliminate $sort_A$ using rule *S1*.

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 does not destroy the ordering. All operations, except $\sqcup$, $\cup$, and $\cup^T$, fully or partially preserve the ordering of the first argument relation.

4.5. Transfer transformation rules

Transfer transformation rules are used in the stratum architecture. If we have an implementation of the same operation in both the stratum and the DBMS, we have a choice

of where to execute the operation. We can transfer a relation from the DBMS to the stratum using operation T^S, and the other way using operation T^D (these operations were not listed in Table 1 because they are specific to the layered architecture).

If a rule transfers an operation from the stratum to the DBMS or vice versa, the relations produced by the left-hand side and the right-hand side of the rule are only $\equiv_M$ equivalent because we cannot be sure how the DBMS implementation of the operation will sort its result, operation $sort$ being the only exception. For this reason, the previously given $\equiv_L$ transformation rules are only applicable in the stratum, and they have corresponding $\equiv_M$ transformation rules for the DBMS. For brevity, the latter rules are omitted from Figure 4.

5. Applicability of transformation rules

Queries expressed in some user-level query language are mapped to an initial algebraic expression, which is then passed to the optimizer, where transformation rules are applied according to some given strategy. The resulting, new algebraic expressions must, when evaluated, return the same result as the original expression, which we assume correctly computes the user's query. In our case, the optimizer must contend with six different types of transformation rules. For each type of rule, we have to formalize when it can be applied.

5.1. Applicability definition

There are no restrictions on when rules with equivalence type $\equiv_L$ may be applied. Applying such rules has no effect on the result; a transformed expression evaluates to a result identical to that obtained from evaluating the original expression. This does not hold for any of the other types of rules. However, they may still be applicable.

Assuming for specificity that the user-level language is some temporal variant of SQL, a query may, or may not, include `DISTINCT` and `ORDER BY` at the outermost level, which affect the type of the result. The presence of the `ORDER BY` clause in a query specifies a result relation that is a list; if the `ORDER BY` clause is absent from the query, the query specifies a multiset, and the order of the result tuples is immaterial. In this latter case, we can apply transformations that merely preserve multiset equivalence. Further, if `DISTINCT` is included at the outermost level of a query (but `ORDER BY` is not), the query returns a relation that is a set.

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 sets, multisets, or lists, depending on whether or not `DISTINCT` and `ORDER BY` were specified at the outermost level of the user-level query. We formalize the applicability of the transformation rules below, thus linking the user-level language and the algebraic optimization.

Definition 5.1 Assume a query Q, its evaluation plan P, a transformation rule T, a location l in the plan where T will be applied, and the evaluation plan P' obtained by applying rule T to P at l. Then, rule T is *applicable at* location l *in* plan P if and only if $P \equiv_{SQL} P'$, where $\equiv_{SQL}$ is (1) $\equiv_S$ if `DISTINCT` is specified at the outermost level of Q, but `ORDER BY` is not specified at that level, (2) $\equiv_M$ if `DISTINCT` and `ORDER BY` are not specified at the outermost level of Q, or (3) $\equiv_{L,A}$ if `ORDER BY A` is specified at the outermost level of Q. □

The definition uses the equivalence type $\equiv_{L,A}$, where A is the list specified in the `ORDER BY` clause. Two relations are $\equiv_{L,A}$ equivalent if their projections on A are $\equiv_L$ equivalent. Thus, $\equiv_L$ equivalence implies $\equiv_{L,A}$ equivalence.

The $\equiv_{SQL}$ equivalence type cannot be one of the snapshot-equivalence types because a query must faithfully preserve the time periods from base relations and cannot arbitrarily return any of the snapshot-equivalent result relations. However, there are cases where snapshot-equivalence type rules can be applied while complying with Definition 5.1; we describe those cases below. Note also that this definition is a posteriori, in that it compares the resulting query plan with the original one. What is needed is an a priori procedure for determining when a transformation rule is applicable.

First, we use an example operation tree for describing which types of transformation rules can be applied to which query regions. Then, Section 5.3 briefly presents the operation properties used to determine when the different types of transformation rules are applicable. Finally, Section 6 describes how these properties are exploited during query plan enumeration.

5.2. Example

Let us again consider the operation tree in Figure 2(a). The result of evaluating the tree is a list. The shaded regions determine which types of transformation rules are applicable.

In the area where order needs not be preserved (the lightly shaded region), we can apply $\equiv_M$ transformation rules. The subtree below the $sort$ operation can treat relations as multisets because the $sort$ operation ensures that the result is ordered appropriately.

Rules of type $\equiv_S$ can be applied to those query fragments where duplicates are not relevant, which are indicated by the darker shaded region. In this example, these fragments are the subtree below the top temporal duplicate elimination operation, except the bottom temporal duplicate elim-

ination operation, which ensures that the left argument of the temporal difference does not contain duplicates in snapshots (see Section 2). (This illustrates that fragments need not always be whole subtrees; in fact, there exist operation trees for which a particular shading is absent for an entire subtree.)

Rules of the snapshot-equivalence types can be applied to those query fragments that need not preserve time periods. This is true for all operations below coalescing because coalescing returns the same result relation for all snapshot equivalent argument relations, if they do not contain duplicates in snapshots (which, in this case, is ensured by temporal duplicate elimination below coalescing). Consequently, below coalescing, $\equiv_M^S$ rules can be applied; $\equiv_S^S$ rules can be applied where duplicates are not relevant.

5.3. Operation properties

The shaded regions in an operation tree are determined using three Boolean properties of operations (see Table 2). Each operation in a tree has a value for each of these properties.

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 2. Operation properties

For example, the *OrderRequired* property does not hold if the $sort$ operation does not exist below in the tree. For all operations in the right branch of a temporal difference, the *DuplicatesRelevant* does not hold if the left argument to the temporal difference does not contain duplicates in its snapshots. Formal definitions of the properties are given elsewhere [20].

During query optimization, the properties are first set for the initial query evaluation plan that is passed to the query optimizer. When a transformation rule is applied, the properties must be adjusted in the transformed area. In most cases, this local adjustment is satisfactory, i.e., properties do not have to be recomputed for all operations in the resulting query tree [20].

The use of the properties in operation trees enables us to formalize when a transformation rule is applicable to a query plan. The next section shows how the properties are used during query plan enumeration.

6. Query plan enumeration

We give a straightforward enumeration algorithm whose purpose is to generate correct query evaluation plans; we consider neither performance nor the subsequent heuristic or cost-based selection of a final query plan.

The inputs to the query plan enumeration algorithm are a set of plans $\mathcal{P}$, containing the initial plan, and a set of transformation rules $\mathcal{TR}$. The output is all query evaluation plans that are possible to obtain using the given set of transformation rules. The algorithm is given in Figure 5. For the algorithm to terminate, the set of transformation rules cannot include all rules given in Section 4. The rules that introduce additional operations, such as $r \rightarrow_S rdup(r)$, could be applied an infinite number of times. Hence, heuristics have to be used to restrict the rule set, as will be described shortly. The algorithm is deterministic, i.e., it generates the same set of query plans independently of the order of transformation rules and locations [20].

Note that operations $rdup^T$, $coal^T$, $\backslash^T$, and $\cup^T$ are order-sensitive, i.e., if they take arguments that are equivalent as multisets, their results may not be equivalent as multisets. We do not cover the resulting complications, but assume that the initial query plan contains those operations only when they preserve multiset equivalence. Such cases include, for example, $coal^T$ combined with $rdup^T$, $coal^T$ when its argument does not have duplicates in snapshots, and $\backslash^T$ when its left argument does not have duplicates in snapshots (for multiset-equivalent right arguments, $\backslash^T$ always returns multiset-equivalent results). The query plan in Figure 2(a) is a suitable input to the algorithm.

for each plan $P \in \mathcal{P}$ **do**
 for each $T \in \mathcal{TR}$ **do**
  for each location l in P matching the left side of T **do**
   if local conditions are satisfied **and**
   $((T$ is a $\equiv_L$ rule$)$
    $\vee$ (T is a $\equiv_M$ rule $\wedge\ \forall op \in l\ (\neg OrderRequired(op))$)
    $\vee$ (T is a $\equiv_S$ rule $\wedge\ \forall op \in l\ (\neg DuplicatesRelevant(op)$
     $\wedge \neg OrderRequired(op))$)
    $\vee$ (T is a $\equiv_L^S$ rule $\wedge\ \forall op \in l\ (\neg PeriodPreserving(op))$)
    $\vee$ (T is a $\equiv_M^S$ rule $\wedge\ \forall op \in l\ (\neg OrderRequired(op)$
     $\wedge \neg PeriodPreserving(op))$)
    $\vee$ (T is a $\equiv_S^S$ rule $\wedge\ \forall op \in l\ (\neg DuplicatesRelevant(op)$
     $\wedge \neg OrderRequired(op) \wedge \neg PeriodPreserving(op))$)
   then apply T to l, yielding P';
    adjust the properties of P';
    add P' to $\mathcal{P}$;
return $\mathcal{P}$

Figure 5. Query plan enumeration algorithm

In the algorithm, when testing the applicability of a transformation rule at some location, the properties of the operations at that location are employed; the operations we consider are those operations in the location that correspond to the operations explicitly mentioned on the left-hand side of the transformation rule and those that correspond to the root nodes of the subtrees mentioned on the left-hand side

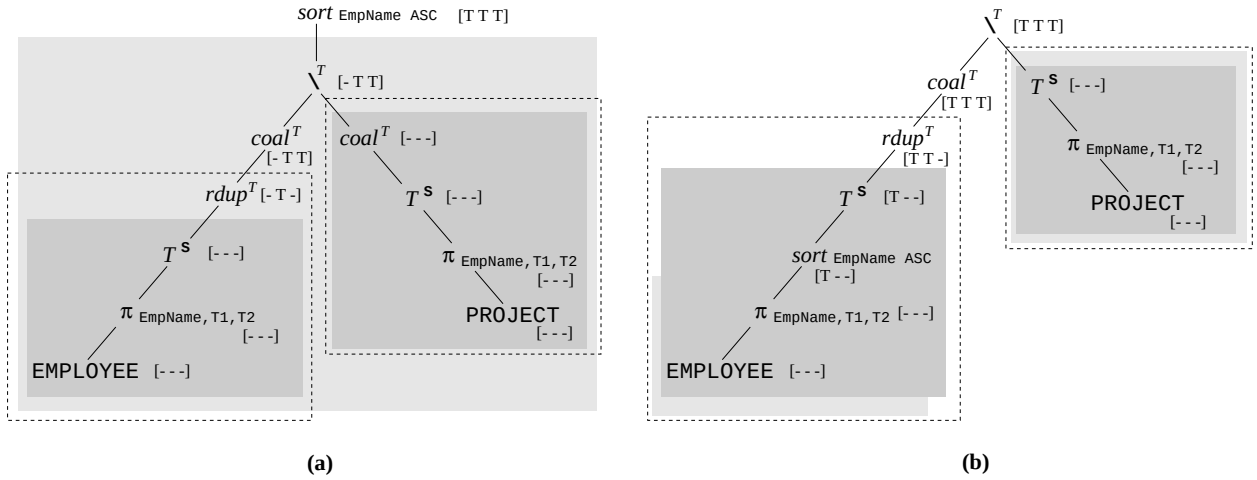

Figure 6. Operation trees with properties and transformation-rule applicability regions

of the transformation rule. For example, when testing the applicability of transformation rule $coal^T(r_1 \setminus^T r_2) \rightarrow_M coal^T(r_1) \setminus^T coal^T(r_2)$, the properties of the operations $coal^T$ and $\setminus^T$ and the operations located at the roots of r_1 and r_2 are used.

The algorithm provides an operational means of determining when a transformation rule is applicable. It has a syntactic component (the left-hand side expression must match in some location) and a semantic component (the preconditions must hold and the properties must be set appropriately). The algorithm generates query plans that are correct.

Theorem 6.1 The algorithm given in Figure 5 generates correct query plans.

Proof: To prove the theorem, we need to prove that it applies only those transformation rules that are *applicable* according to Definition 5.1. The proof is divided into six parts, one for each type of transformation rule [20]. □

This theorem achieves *correctness*, but not *completeness*, i.e., correct query plans are generated, and we exploit transformation rules of "weak" equivalence types, e.g., $\equiv_S$, but we do not find *all* possible correct query plans that may be generated using the different types of transformation rules.

To prevent the algorithm from generating an infinite number of plans, heuristics have to be used. For example, one heuristic could be that rules that introduce additional operations, such as $r \rightarrow_S rdup(r)$, should not be used. Another heuristic can be that selections have to be performed as early as possible. Thus, we would allow the transformation rule $\sigma_P(coal^T(r)) \rightarrow_L coal^T(\sigma_P(r))$, but would not use transformation rule $coal^T(\sigma_P(r)) \rightarrow_L \sigma_P(coal^T(r))$.

To illustrate how the algorithm works, we use the example query from Section 2. The initial query plan is given in Figure 2(a). First, we push the transfer operation down. Then, since the result of the temporal difference does not contain duplicates in snapshots (because its left argument does not contain duplicates in snapshots), we apply rule *D2* and remove the top temporal duplicate elimination.

Then we push the coalescing below the temporal difference by using rule *C10* (we can apply this rule because $OrderRequired$ does not hold for each participating operation). The resulting plan is shown in Figure 6(a). For each operation, we list its properties in square brackets in the order $\langle OrderRequired, DuplicatesRelevant, PeriodPreserving \rangle$.

Next, we remove the unnecessary coalescing appearing in the second argument to the temporal difference, using rule *C2*; order and time periods need not be preserved in the right branch of a temporal difference. Finally, we push the $sort$ operation down, and we change the location of the $sort$ operation from the stratum to the DBMS. Figure 6(b) shows the final plan.

7. Conclusions and future work

Temporal query representation, optimization, and processing mechanisms are needed to achieve built-in temporal support in DBMSs. However, previously proposed conventional and temporal algebras have to varying degrees overlooked such aspects as duplicates, ordering, and coalescing. In addition, past work considered the efficient processing of only some operations, e.g., temporal joins, and did not delve into general query optimization.

This paper offers a general foundation for optimizing conventional and temporal queries, which is suitable for providing temporal support via a stand-alone temporal DBMS or via a layer on top of a conventional DBMS. This foundation offers comprehensive and precise handling of duplicates and order for conventional and temporal queries, as well as coalescing for temporal queries. The foundation is enabled

by a temporally extended algebra, which enhances existing relational algebras based on multisets by accommodating order, and also adds temporal support.

Six types of equivalences among algebraic query expressions are distinguished, leading to six types of transformation rules that can be exploited during query optimization. These sets of rules extend all such existing sets known to the authors. Depending on whether order, duplicate removal, or coalescing are required for the result of a query, the query optimizer may apply different types of transformation rules. A practical mechanism is provided for determining when a transformation rule of some type is applicable to a query. Finally, an algorithm that generates equivalent query plans is presented. This approach partitions the work required by the database implementor to develop a provably correct query optimizer into three stages; the database implementor has to (1) specify operations formally in λ-calculus; (2) design appropriate transformation rules, determine which of the six equivalences is appropriate, and prove that the transformation rules are correct; and (3) augment the setting and adjusting of the operation properties so that the enumeration algorithm applies the transformation rules correctly.

Future work includes integrating the provided transformation rules with heuristics and cost estimation techniques, which are necessary to achieve an efficient and effective optimizer. For the layered architecture, strategies for dividing the processing between the layer and the DBMS must be developed. Multiple implementations of operations, e.g., several join implementations that return differently ordered relations, should be considered. In addition, once a specific query language is chosen, checks should be included that, for a query plan, ensure that the tasks assigned to the DBMS are expressible in SQL and that the operations assigned to the layer have corresponding implementation algorithms. Also, the complications arising from order-sensitive operations should be studied further.

Intended as a foundation for the efficient processing of SQL-like queries, the algebra includes the standard operations called for by this type of queries. The operations were specified in recursive-style definitions that used operations such as $head$, $tail$, and concatenation. The inclusion of these and other list operations in the algebra may be explored. In addition, the algebra may be extended to support modifications, NOW-relative values [6], and both valid and transaction time [12].

Acknowledgments

This research was supported in part by the Danish Technical Research Council through grant 9700780, by the U.S. National Science Foundation through grant IIS-9817798, by the Chorochronos project, funded by the European Commission DG XII, contract no. FMRX-CT96-0056, and by a grant from the Nykredit Corporation.

References

[1] J. Albert. Algebraic Properties of Bag Data Types. In *Proceedings of VLDB*, Barcelona, Spain, pp. 211–219 (1991).

[2] C. Bettini et al. A Glossary of Time Granularity Concepts. In [8], pp. 406–413 (1998).

[3] M. H. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4): 53–60 (1995).

[4] M. H. Böhlen, R. Busatto, and C. S. Jensen. Point versus Interval-Based Temporal Data Models. In *Proceedings of IEEE ICDE*, Orlando, Florida, pp. 192–200 (1998).

[5] M. H. Böhlen, R T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *Proceedings of VLDB*, Bombay, India, pp. 180–191 (1996).

[6] J. Clifford et al. On the Semantics of "Now" in Databases. *ACM TODS*, 22(2): 171–214 (1997).

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

[8] O. Etzion, S. Jajodia, and S. Sripada (eds.) *Temporal Databases: Research and Practice*. Springer-Verlag (1998).

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

[10] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Proceedings of SSDBM*, Charlotte, North Carolina, pp. 131–147 (1990).

[11] W. H. Inmon. *Building the Data Warehouse*. Second Edition. John Wiley and Sons (1996).

[12] C. S. Jensen. A Consensus Glossary of Temporal Database Concepts. In [8], pp. 367–405 (1998).

[13] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE TKDE*, 11(1): 36–45 (1999).

[14] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7): 513–547 (1994).

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

[16] T. Y. C. Leung et al. Query Rewrite Optimization Rules in IBM DB/2 Universal Database. In *Readings in Database Systems*, Third Edition, M. Stonebraker and J. Hellerstein (eds.), Morgan Kaufmann, pp. 153-168 (1998).

[17] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases: Theory, Design, and Implementation*, A. U. Tansel et al. (eds.), Benjamin/Cummings, pp. 329–355 (1993).

[18] L. E. McKenzie, Jr. and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4): 501–543 (1991).

[19] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7(4): 513–532 (1995).

[20] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. TIMECENTER Technical Report (1999). <www.cs.auc.dk/TimeCenter>

[21] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann (1999).

[22] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In *Proceedings of IDEAS*, Cardiff, Wales, pp. 4–13 (1998).