A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering

Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass

TR-49

A TimeCenter Technical Report

Title A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering

Copyright © 2000 Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. All rights reserved.

Author(s) Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass

Publication History February 2000. A TIMECENTER Technical Report.

Abstract

Most real-world databases contain substantial amounts of time-referenced, or temporal, data. Recent advances in temporal query languages show that such database applications may 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 foundation for query optimization that integrates conventional and temporal query optimization and is suitable for both conventional DBMS architectures and ones where the temporal support is obtained via a layer on top of a conventional DBMS. This foundation captures duplicates and ordering for all queries, as well as coalescing for temporal queries, thus generalizing all existing approaches known to the authors. It includes a temporally extended relational algebra to which SQL and temporal SQL queries may be mapped, six types of algebraic equivalences, concrete query transformation rules that obey different equivalences, a procedure for determining which types of transformation rules are applicable for optimizing a query, and a query plan enumeration algorithm.

The presented approach partitions the work required by the database implementor to develop a provably correct query optimizer into four stages: the database implementor has to (1) specify operations formally; (2) design and prove correct appropriate transformation rules that satisfy any of the six equivalence types; (3) augment the mechanism that determines when the different types of rules are applicable to ensure that the enumeration algorithm applies the rules correctly; and (4) ensure that the mapping generates a correct initial query plan.

1 Introduction

Most real-world database applications rely on time-referenced data. For example, this applies to financial, medical, and travel applications; and being time-variant is one of Inmon's defining properties of a data warehouse [Inm96]. Recent advances in temporal query languages [EJS98, JS99] show that such applications may benefit substantially from running on a DBMS with built-in temporal support. The potential benefits are several: application code is simplified and more easily maintainable, thereby increasing programmer productivity [Sno99], 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 [YYW00]. 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 feasible by DBMS vendors that already have a code base to modify and have large resources available. This has led to the consideration of a layered, or stratum, approach where a layer that implements temporal support is interposed between the user applications and a conventional DBMS [Böh95, TJS98]. The layer maps temporal SQL statements to regular SQL statements and passes them to the DBMS, which remains unaltered.

This paper offers a foundation for conventional and temporal query optimization that is applicable to both the integrated and the layered architecture, making it relevant for DBMS vendors planning to incorporate temporal features into their products, as well as to third-party developers that want to implement temporal support. The foundation offers comprehensive, precise, and integrated coverage of duplicates and ordering 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 that enhances existing relational algebras based on sets or multisets by integrating the handling of order; the algebra also adds temporal support. In addition to conventional relations, the algebra employs temporal relations timestamped with time periods, which are the most useful for implementation because of their granularity independence and fixed-size format. Previously proposed user-level temporal relations may be mapped to this format [JSS94]. 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) statements that explicitly manipulate values of (new) temporal abstract data types with convenient operations and predicates defined on them. The temporal aspect considered is valid time [JD98], which captures when data was, is, or will be true in the modeled reality; the approach can be extended to also handle transaction time, alone as well as in combination with valid time.

In the algebra, relations are defined as lists, and six kinds of equivalences are defined on them. 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. (The snapshot of a temporal relation at time t contains those tuples (without the time periods) from the temporal relation that have time periods containing t.)

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 opening the possibility of applying transformations that do not preserve list equivalence. The paper provides a set of transformation rules that satisfy different equivalences. This set goes beyond all 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. Finally, an algorithm is provided that generates equivalent query evaluation plans.

Some work has been reported on non-temporal relational algebras for multisets [Alb91, DGK82, GM00], 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 Access allow the `ORDER BY` clause in combination with the `TOP` predicate in subqueries, thus requiring intermediate results to be sorted.

Because relations are formalized as lists, comprehensive support for sorting is achieved. In addition, a mechanism is offered that determines when list, multiset, and set based equivalences, including their temporal counterparts, are applicable during query optimization. Recent work by Leung et al. [Leu98] 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 [MS91, OS95], 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 [GS90, LM93] primarily considers the processing of joins and semijoins in isolation, does not delve into general query optimization, and does not address duplicates, order, and coalescing.

The paper is structured as follows. Section 2 describes the layered architecture. Section 3 defines the underlying database structures and presents the extended relational algebra operations. The different types of algebraic equivalences are described in Section 4, and the concrete transformation rules that preserve the different equivalence types are provided in Section 5. Sections 6 and 7 give a procedure for determining when transformation rules preserving the different types of equivalence are applicable and provide a query plan enumeration algorithm. The extensibility of the framework is briefly discussed in Section 8. Section 9 surveys related work, and Section 10 concludes and offers research directions.

2 Architecture

In this section, we discuss the layered, or stratum, architecture. We describe the functionality of the query optimizer and give the overall structure of the stratum.

Several papers discussing stratum architectures for a temporal DBMS have been published, e.g., [TJS98], and several prototype temporal DBMSs have been implemented, e.g., [Böh95, Böh98]. Most of the proposed temporal strata translate temporal query language statements to SQL and perform no systematic optimization or processing. However, dividing processing between the stratum and the underlying DBMS may improve query performance since complex temporal operations such as temporal aggregation, temporal duplicate elimination, and coalescing are often not processed efficiently in conventional DBMSs, but might be supported by the stratum. We will use the term "stratum" to mean an *augmented* stratum that, in addition to the mapping, performs some of the query optimization and processing.

Figure 1 shows the processes involved in optimizing and evaluating a query. The stratum receives a temporal query language statement as input. First, the query statement is mapped to an initial plan, which is expressed in a temporal algebra suited for the internal representation. The stratum's query optimizer, e.g., using transformation rules, generates a number of possible query evaluation plans. The plans are costed, and one is selected for processing. The fourth step is the only step that is specific to the stratum architecture; here the fragments of the plan to be performed by the DBMS are translated into SQL. Finally, the resulting SQL and stratum expressions are evaluated to obtain the result. In our paper, we focus on the second step: we develop a temporally extended algebra, transformation rules, and a query plan enumeration algorithm.

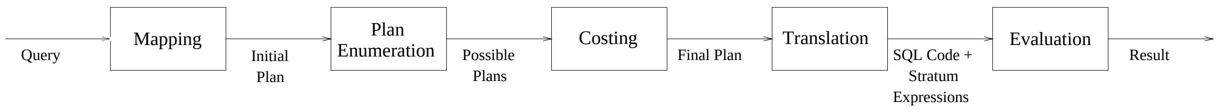


Figure 1: Query Optimization and Evaluation in a Temporal Stratum

Figure 2 depicts an example stratum architecture. The components here perform the tasks described in Figure 1. The query processing controller passes relevant query fragments to the underlying DBMS and to the internal query evaluator, collects the results, and outputs the result relation. Statistics such as the running time and characteristics of result relations may be used by the cost estimator to update its cost models for the DBMS.

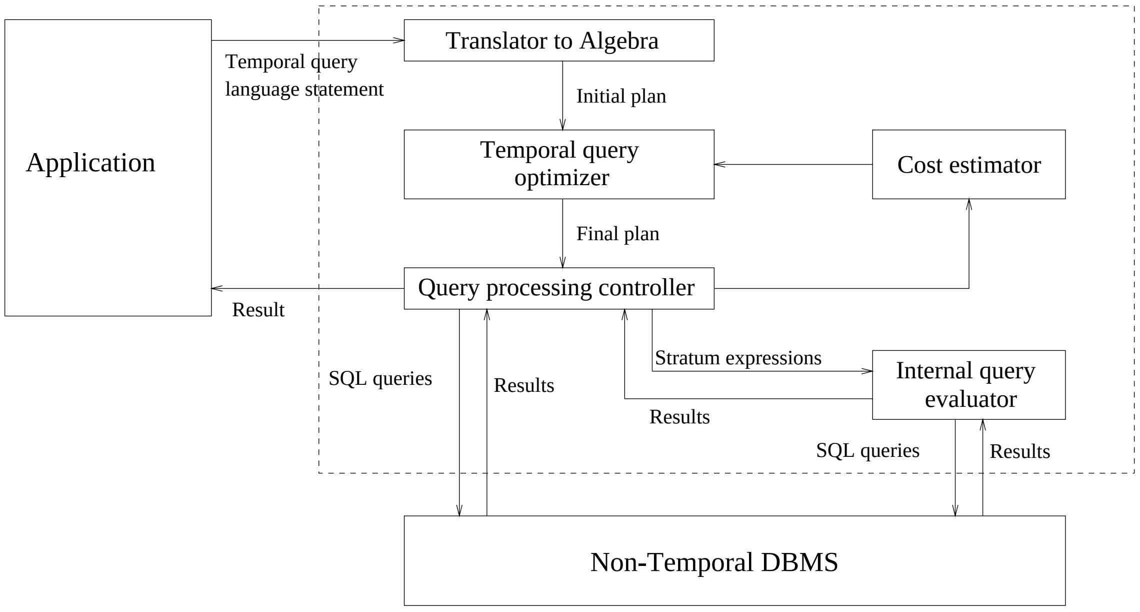


Figure 2: Architecture of a Temporal Stratum

3 An Extended Algebra

In this section, we present our extended algebra. First, we discuss requirements for the algebra and define its relation structures. Then, we describe and define fundamental algebra operations. Finally, we briefly consider the mapping from queries to the algebra and give an example query.

3.1 Requirements

It is a fundamental requirement that the algebra be formally defined. Equally fundamental, the algebra must be suitable for implementation, which has several implications that will be clear as we get into the details. Next, the algebra must incorporate duplicates, ordering, and coalescing. This implies that the relations must be lists. 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, the operations should be defined using the start and end times of the argument tuples' time periods only.

The algebra must extend the conventional relational algebra and must accommodate both classes of temporal statements 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 former class, we introduce temporal operations that are counterparts of existing relational algebra operations, in the sense that they are snapshot-reducible

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

Figure 3: Relation EMPLOYEE

to these. A temporal operation op_1 is *snapshot-reducible* to operation op_2 if for any point in time and for any temporal relation r, the snapshot at time t of the result of applying op_1 to r is equal to the result of op_2 applied to the snapshot of r at time t [Sno87]. For example, temporal duplicate elimination is snapshot reducible to duplicate elimination.

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. For example, coalescing should not affect duplicates; a separate duplicate elimination operation should be available. 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.

3.2 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 3.1 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. $\square$

Consider temporal relation EMPLOYEE in Figure 3. We assume a closed-open representation for time periods and let the time values denote months during some year. For example, John is in Sales from January through July and in Advertising from June through October. Relation schema EMPLOYEE consists of the attributes EmpName, Dept, T1, and T2 and 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 Bettini et al. [Bet98].

Definition 3.2 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. $\square$

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 EMPLOYEE relation from Figure 3 contains tuples $t_1 = \{(\texttt{EmpName}, \texttt{John}), (\texttt{Dept}, \texttt{Sales}), (\texttt{T1}, 1), (\texttt{T2}, 8)\}$, $t_2 = \{(\texttt{EmpName}, \texttt{John}), (\texttt{Dept}, \texttt{Advertising}), (\texttt{T1}, 6), (\texttt{T2}, 11)\}$, $t_3 = \{(\texttt{EmpName}, \texttt{Anna}), (\texttt{Dept}, \texttt{Sales}), (\texttt{T1}, 2), (\texttt{T2}, 6)\}$, $t_4 = \{(\texttt{EmpName}, \texttt{Anna}), (\texttt{Dept}, \texttt{Advertising}), (\texttt{T1}, 2), (\texttt{T2}, 6)\}$, and $t_5 = \{(\texttt{EmpName}, \texttt{Anna}), (\texttt{Dept}, \texttt{Sales}), (\texttt{T1}, 6), (\texttt{T2}, 12)\}$. The list $\langle t_1, t_2, t_3, t_4, t_5\rangle$ then is the EMPLOYEE relation in our example.

We distinguish between snapshot (also termed *conventional*) and temporal relations. We reserve two specific attribute names, T1 and T2, for denoting the time period start and end, respectively, of the period of validity for each tuple in a temporal relation. The schema of a snapshot relation does not contain these two attributes; a schema of a temporal relation does contain them. 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.

3.3 Algebra Operations

We first describe briefly all the fundamental algebra operations, discussing how they preserve order, duplicates, and coalescing. We define all operations in Sections 3.3.2–3.3.16.

3.3.1 Overview of Operations

Table 1 lists all operations. Selection (σ), projection (π), union ALL ($\sqcup$), Cartesian product ($\times$), difference ($\setminus$), duplicate elimination ($rdup$), and aggregation (ξ) 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 (exemplified by the difference between regular duplicate elimination and temporal duplicate elimination, to be discussed in Sections 3.3.9 and 3.3.10, respectively). We also add sorting and coalescing; the latter merges value-equivalent tuples with adjacent time periods. Our definition of coalescing is different from that given by Böhlen et al. [BSS96], due to the requirement of minimality (see Section 3.1) and our relations being defined as lists. The coalescing of Böhlen et al. merges value-equivalent tuples with adjacent or *overlapping* time periods; in our algebra, this result is achieved by combining temporal duplicate elimination and coalescing. Union ($\cup$) originates from the union operation for multisets given in [Alb91]. 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. Its temporal counterpart is denoted by $\cup^T$.

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	n.a.
$r_1 \setminus r_2$	$Order(r_1)$	$\geq (n(r_1) - n(r_2))$ and $\leq n(r_1)$	Retains	n.a.
$rdup(r)$	$Order(r)$	$\leq n(r)$	Eliminates	n.a.
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$	$Prefix(Order(r), GroupPairs)$	$\leq n(r)$	Eliminates	n.a.
$r_1 \times^T r_2$	$Prefix(Order(r_1), Order(r_1) \setminus TimePairs)$	$\leq n(r_1) \cdot n(r_2)$	Retains	Destroys
$r_1 \setminus^T r_2$	$Prefix(Order(r_1), Order(r_1) \setminus TimePairs)$	$\leq 2 \cdot n(r_1)$	Retains	Destroys
$rdup^T(r)$	$Prefix(Order(r), Order(r) \setminus TimePairs)$	$\leq 2 \cdot n(r) - 1$	Eliminates	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
$sort_A(r)$	A	$= n(r)$	Retains	Retains
$coal^T(r)$	$Prefix(Order(r), Order(r) \setminus TimePairs)$	$\leq n(r)$	Retains	Enforces
$r_1 \cup r_2$	unordered	$\geq n(r_1)$ and $\leq (n(r_1) + n(r_2))$	Retains	n.a.
$r_1 \cup^T r_2$	unordered	$\geq n(r_1)$ and $\leq (n(r_1) + 2 \cdot n(r_2))$	Retains	Destroys

Table 1: Overview of Operations

Table 1 includes fundamental operations, as well as the temporal operations needed to conveniently accommodate query statements with built-in temporal semantics [BJ97, EJS98]. We omit derived operations (idioms), except regular and temporal unions, which can be expressed via union ALL and regular (temporal) difference. We include the latter two idioms to illustrate how we can deal with the union operation provided in [Alb91]. The addition of other 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 [GM00], in that this latter algebra works on multisets, not lists. However, some of our operations, specifically selection, projection, Cartesian product, difference, union ALL, duplicate elimination, and aggregation 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 [GM00] because each operation of the latter may be expressed by one of the seven operations just listed.

Table 1 also shows, for each operation, the order and cardinality of the result relation, and how the operation handles regular duplicates and coalescing. This table makes use of several auxiliary functions. Function $Order(r)$ returns a list of attributes paired with a sorting type (ascending or descending) for a relation r, for example, $Order(r) = \langle(\mathtt{A}, \mathtt{ASC}), (\mathtt{B}, \mathtt{DESC})\rangle$. For an unordered relation, the function returns an empty list. Note that in the special case when the sorting list A is a prefix of $Order(r)$, the order of $sort_A(r)$ is $Order(r)$. The lists $ProjPairs$, $TimePairs$, and $GroupPairs$ include, respectively, the projection attributes, the temporal attributes, and the grouping attributes paired with ASC or DESC. The $TimePairs$ list is equal to $\langle(\mathtt{T1}, \mathtt{ASC}), (\mathtt{T1}, \mathtt{DESC}), (\mathtt{T2}, \mathtt{ASC}), (\mathtt{T2}, \mathtt{DESC})\rangle$.

Function *Prefix* returns the largest prefix of its first argument such that the prefix would contain only elements included in the second argument. For example, if relation r is sorted on $Order(r) = \langle(\mathtt{A}, \mathtt{ASC}), (\mathtt{B}, \mathtt{ASC}), (\mathtt{C}, \mathtt{DESC})\rangle$, and we project it on A and C, the $ProjPairs$ list would be $\langle(\mathtt{A}, \mathtt{ASC}), (\mathtt{A}, \mathtt{DESC}), (\mathtt{C}, \mathtt{ASC}), (\mathtt{C}, \mathtt{DESC})\rangle$. The *Prefix* function on the two lists would return $\langle(\mathtt{A}, \mathtt{ASC})\rangle$, i.e., the result of the projection would be sorted on A.

We denote the cardinality of relation r by $n(r)$. The lower bound is 0 in all cases not specified in the table.

The last two columns reflect the behavior of the operation with respect to duplicates and coalescing. An operation 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 even if duplicates do not exist 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 nontemporal operations that have temporal counterparts).

The next sections define the algebra operations listed in Table 1. Overall, an attempt has been made to define operations conducive to efficient implementation. For example, union ALL simply concatenates its arguments. In these 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 $r \in \mathcal{R}, r = \langle t_1, t_2, \ldots, t_n\rangle$. Similarly, we 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. Also, we let $\mathcal{R}^{sn}$ be a set of all relations with tuples not having any temporal support.

We use λ-calculus for the definitions [Gor87]. The definitions do not imply actual implementation algorithms, but *do* constrain the implementation algorithms to produce the same results, taking order and duplicates into account.

3.3.2 Selection

Selection operation $\sigma : [\mathcal{R} \times \mathcal{P}] \to \mathcal{R}$ corresponds to the well-known selection operation in the relational algebra [GM00]. The set of all possible selection predicates is denoted by $\mathcal{P}$. The argument predicate is expressed as a subscript, e.g., $\sigma_P(r)$. The schema of the result relation is the same as schema of the argument relation.

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

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 the argument relation r is empty (we denote an empty relation by $\perp$), the operation returns it. Otherwise, the second line is processed, which says that if the relation 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) appended (@) to the result of the operation applied to the remaining part of the relation.

The auxiliary functions $head$, $tail$, and @ are defined in Appendix A.

3.3.3 Projection

Projection operation $\pi : [\mathcal{R} \times \mathcal{F} \times \ldots \times \mathcal{F}] \rightarrow \mathcal{R}$ corresponds to its relational counterpart. $\mathcal{F}$ is a set of arithmetic expressions $f_i : \mathcal{T} \rightarrow \mathcal{T}$, which can include any possible attribute names and which return single-attribute tuples. After f_i is applied, the resulting schema contains one attribute name, one type, and one mapping from the attribute name to the type. Functions $f_1, \ldots, f_n$ are expressed as a subscript, e.g., $\pi_{f_1,\ldots,f_n}(r)$.

For example, with the schema $S = (\Omega, \Delta, dom)$, $\mathtt{A}, \mathtt{B} \in \Omega$, $(\mathtt{A}, \mathrm{int}), (\mathtt{B}, \mathrm{int}) \in dom$, one possible function f_i is $\mathtt{A} + 2 \cdot \mathtt{B}\ \mathtt{AS}\ \mathtt{C}$.

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

The schema of the result relation follows from the definition of tuple concatenation ($\circ$); see Appendix A.

The projection operation can be used to add new attributes to the schema. If a new non-temporal attribute is added, its value is set to `NULL` for each tuple of the argument relation. If a new temporal attribute is added, its value for each tuple of the argument relation is set to the current time (if the attribute is `T1`) or the maximum timestamp value (if the attribute is `T2`).

3.3.4 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 relation. The schemas of both argument relations and the result relation are the same.

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

3.3.5 Cartesian Product

Operation $\times : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}^{sn}$ computes the Cartesian product of two argument relations. The definition uses the auxiliary function $OneLoop : [\mathcal{T} \times \mathcal{R}] \rightarrow \mathcal{R}^{sn}$. The resulting schemas of $\times$ and $OneLoop$ follow from the definition of tuple concatenation. The only exception is that if the attribute domain of the resulting schema contains any of the two special temporal attributes, those attributes are prefixed by "1", because the result of this operation is to be a snapshot relation, which cannot include attributes named `T1` or `T2`.

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

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

The definition essentially performs a nested-loop Cartesian product.

3.3.6 Temporal Cartesian Product

Operation $\times^T : [\mathcal{R}^T \times \mathcal{R}^T] \rightarrow \mathcal{R}^T$ returns a temporal Cartesian product of two argument temporal relations. The definition uses auxiliary the function $OneLoop^T : [\mathcal{T}^T \times \mathcal{R}^T] \rightarrow \mathcal{R}^T$. The resulting schemas of $\times^T$ and $OneLoop^T$ follow from the definition of tuple concatenation. The attribute domain of the resulting schema retains the original timestamps of both argument relations and, in addition, has two new timestamps.

$$\begin{aligned} \times^T \triangleq\ & \lambda r_1, r_2.((r_1 = \perp) \vee (r_2 = \perp)) \rightarrow \perp, \\ & \qquad OneLoop^T(head(r_1), r_2)\ \sqcup\ (tail(r_1) \times r_2) \end{aligned}$$

$$\begin{aligned} OneLoop^T \triangleq\ & \lambda t, r.(r = \perp) \rightarrow \perp, \\ & \qquad DoesOverlap^T(t, head(r)) \rightarrow \\ & \qquad\qquad (t \circ head(r) \circ GetIntersectingTuple^T(t, head(r)))\ @\ OneLoop^T(t, tail(r)), \\ & \qquad OneLoop^T(t, tail(r)) \end{aligned}$$

Function $\mathit{DoesOverlap}^T$ checks if the time periods of two argument tuples overlap. Function $\mathit{GetIntersectingTuple}^T$ intersects the time periods of two argument tuples and, if they overlap, forms a new tuple containing the intersection time period; otherwise, it returns `NULL`. Both functions are defined in Appendix A.

The temporal Cartesian product retains the original timestamps of both its arguments, prefixed by "1" and "2". This make it possible to accommodate selection predicates involving time attributes from more than two relations [BJ97]. The prefixed timestamps can be removed by a subsequent projection if they are not needed.

With the chosen definition, the temporal Cartesian product is not snapshot reducible to the regular Cartesian product. However, temporal Cartesian product followed by projection that removes the prefixed timestamps is snapshot reducible to the regular Cartesian product.

3.3.7 Difference

Operation $\setminus : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}^{sn}$ returns all tuples of the first argument relation that are not in the second argument relation. The schemas of both argument relations and the result relation are the same, with the exception that we prefix all temporal attributes, if any, by "1" in the result schema.

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

Function *isIn* returns True if the argument tuple exists in the argument relation. Function *remove* removes the first occurrence of the argument tuple from the argument relation. Both functions are defined in Appendix A.

3.3.8 Temporal Difference

Operation $\setminus^T : [\mathcal{R}^T \times \mathcal{R}^T] \rightarrow \mathcal{R}^T$ performs temporal difference. Both argument relations and the output relation have the same schema, where non-temporal attribute values are denoted as $a_1, \ldots, a_n$.

$$\begin{aligned}\setminus^T \triangleq \lambda r_1, r_2.((r_1 = \perp) \vee (r_2 = \perp)) \rightarrow r_1, \\ (\mathit{OverTpl}^T(\mathit{head}(r_1), r_2) = \mathit{undef}) \rightarrow \mathit{head}(r_1) @ (\mathit{tail}(r_1) \setminus^T r_2), \\ (s_1 \sqcup \mathit{tail}(r_1)) \setminus^T (s_2 \sqcup \mathit{remove}(\mathit{OverTpl}^T(\mathit{head}(r_1), r_2), r_2))\end{aligned}$$

where s_1 and s_2 are defined below.

For each tuple from the first argument relation, we look for tuples in the second argument relation that overlap with it. If we find an overlapping tuple, we remove the overlapping temporal part from both tuples and perform the difference again on the remaining parts of the relations, the contents of which depend on the type of the overlap. Allen [All83] identified thirteen relationships between intervals, and Figure 4 shows the nine different cases of overlapping (the other four, nonoverlapping predicates are before, before^{-1}, meets, and meets^{-1}). We use the additional relations s_1 and s_2—which contain from zero to two tuples—for adjusting the relations; s_1 provides the remainder of A, and s_2 provides the remainder of B.

$$s_1 = \begin{cases} \langle \mathit{nontemporal} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T2} \circ \mathit{head}(r_1).\mathtt{T2} \rangle & \textbf{if } \text{Case } 1,\ 7 \\ \perp & \textbf{if } \text{Case } 2,\ 3,\ 8,\ 9 \\ \langle \mathit{nontemporal} \circ \mathit{head}(r_1).\mathtt{T1} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T1} \rangle \sqcup & \\ \quad \langle \mathit{nontemporal} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T2} \circ \mathit{head}(r_1).\mathtt{T2} \rangle & \textbf{if } \text{Case } 4 \\ \langle \mathit{nontemporal} \circ \mathit{head}(r_1).\mathtt{T1} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T1} \rangle & \textbf{if } \text{Case } 5,\ 6 \end{cases}$$

$$s_2 = \begin{cases} \perp & \textbf{if } \text{Case } 1,\ 2,\ 4,\ 5 \\ \langle \mathit{nontemporal} \circ \mathit{head}(r_1).\mathtt{T2} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T2} \rangle & \textbf{if } \text{Case } 3,\ 6 \\ \langle \mathit{nontemporal} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T1} \circ \mathit{head}(r_1).\mathtt{T1} \rangle & \textbf{if } \text{Case } 7,\ 8 \\ \langle \mathit{nontemporal} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T1} \circ \mathit{head}(r_1).\mathtt{T1} \rangle \sqcup & \\ \quad \langle \mathit{nontemporal} \circ \mathit{head}(r_1).\mathtt{T2} \circ \mathit{OverTpl}^T(\mathit{head}(r_1), r_2).\mathtt{T2} \rangle & \textbf{if } \text{Case } 9 \end{cases}$$

$$\mathit{nontemporal} = \mathit{head}(r_1).a_1 \circ \ldots \circ \mathit{head}(r_1).a_n$$

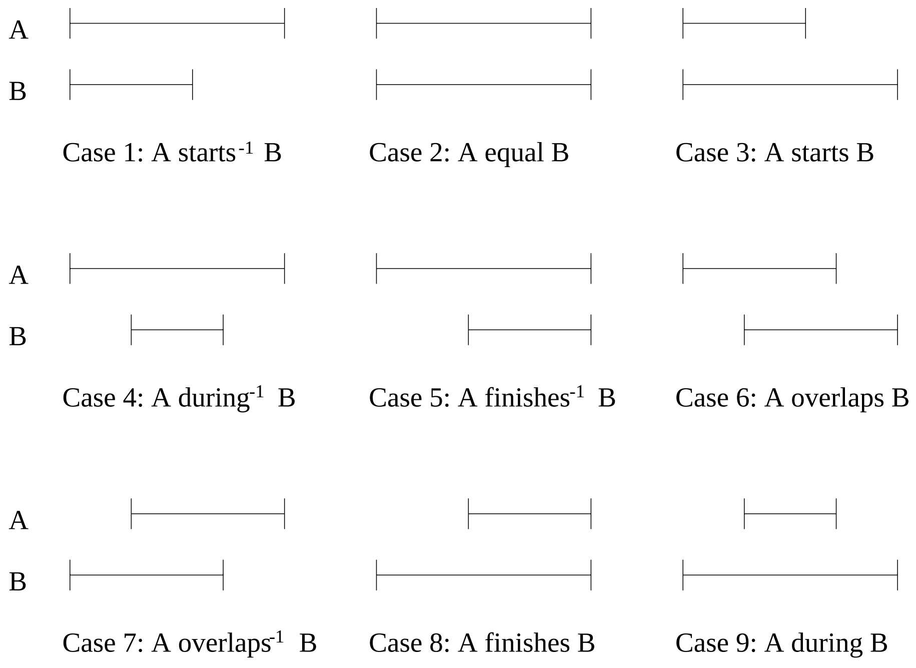


Figure 4: Nine Cases of Overlapping

The lower bound for the cardinality of the result relation is 0 because tuples with huge time periods in the second argument relation may eliminate all tuples with short time periods from the first argument relation. The upper bound is twice as big as the number of tuples in the first argument relation, because for each tuple of the first argument relation, we may get two new tuples in the result (cf. Case 4 in Figure 4).

Consider the temporal difference among relations `EMPLOYEE` (see Figure 3) and `PROJECT` (see Figure 5) projected on `EmpName`, `T1`, and `T2`, i.e., $\pi_{\texttt{EmpName,T1,T2}}(\texttt{EMPLOYEE}) \setminus^T \pi_{\texttt{EmpName,T1,T2}}(\texttt{PROJECT})$. The result is given to the right in Figure 5. Note that temporal difference is sensitive to duplicates. For example, the second tuple for Anna with times 2 and 6 from the `EMPLOYEE` relation is directly transfered to the result because all value-equivalent (tuples with the same non-temporal attribute values) overlapping tuples from the `PROJECT` relation were eliminated by the first such tuple for Anna.

3.3.9 Duplicate Elimination

Operation $rdup : \mathcal{R} \to \mathcal{R}^{sn}$ removes regular duplicates from the argument relation. This operation retains the first occurrence of each tuple and removes all subsequent occurrences, if any. The schemas of the argument and result relations are the same, with the exception that the temporal attributes in the resulting schema, if any, are prefixed by “1”.

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

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.

3.3.10 Temporal Duplicate Elimination

Operation $rdup^T : \mathcal{R}^T \to \mathcal{R}^T$ removes duplicates from all snapshots of the argument temporal relation. The argument and result relations have the same schema. Note that this operation also removes regular duplicates.

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

Figure 5: Relation PROJECT and the Result Relation

Figure 6 shows the EMPLOYEE relation projected on $L = \{\texttt{EmpName}, \texttt{T1}, \texttt{T2}\}$ and the results of regular and temporal duplicate elimination applied to this relation. Relation R2 does not contain regular duplicates (there is only

$\texttt{R1} = \pi_L(\texttt{EMPLOYEE})$

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

$\texttt{R2} = rdup(\texttt{R1})$

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

$\texttt{R3} = rdup^T(\texttt{R1})$

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

Figure 6: Results of Regular and Temporal Duplicate Elimination

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). Temporal duplicate elimination preserves the order of the argument relation and is defined below.

$$
\begin{aligned}
rdup^T \triangleq \lambda r.(r = \perp \vee\ tail(r) = \perp) &\rightarrow r, \\
(OverTpl^T(head(r), tail(r)) = undef) &\rightarrow head(r) \ @\ rdup^T(tail(r)), \\
&\ rdup^T(head(r) \ @\ ChangeTuple(OverTpl^T(head(r), tail(r)), tail(r), r_{new})) \\
\text{where } r_{new} = \langle OverTpl^T(head(r), tail(r))\rangle &\setminus^T \langle head(r)\rangle.
\end{aligned}
$$

Function $OverTpl^T$, defined in Appendix A, scans the argument relation and finds the first tuple whose time period overlaps with the argument tuple and is value-equivalent with it (e.g., the first two tuples of R1 overlap and are value-equivalent). If there is no such tuple, we retain the first tuple. Otherwise, the period of validity of 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 $ChangeTuple^T : [\mathcal{T}^T \times \mathcal{R}^T \times \mathcal{R}^T] \rightarrow \mathcal{R}^T$ 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 the 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.

$$
\begin{aligned}
ChangeTuple \triangleq \lambda t, r, r_{new}.(r = \perp) &\rightarrow r, \\
(t = head(r)) &\rightarrow r_{new} \sqcup tail(r), \\
&\ head(r) @ ChangeTuple(t, tail(r), r_{new})
\end{aligned}
$$

The operation may return at most $2 \cdot n(r) - 1$ tuples. If we have x value-equivalent tuples in the argument relation, we cannot have more than $2 \cdot x$ different time values in those tuples, which means that the maximum number of valid time periods involving those time values is $2 \cdot x - 1$. In addition, x can at most be $n(r)$, and if $x < n(r)$ then the maximum cardinality is smaller.

3.3.11 Aggregation

Operation $\xi : [\mathcal{R} \times \Omega \times \ldots \Omega \times \mathbb{F} \times \ldots \times \mathbb{F}] \rightarrow \mathcal{R}^{sn}$ 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 $\mathbb{F}$; aggregate function $F_i : \mathcal{R} \rightarrow \mathcal{T}$ takes a relation as argument and returns a single-attribute tuple containing the aggregate value. After F_i is applied, the schema of the result tuple contains one attribute, one type, and one mapping from the attribute name to the type. An example of aggregate function is `AVG(C) 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. The only exception is that, in the resulting schema, temporal attributes, if any, are prefixed by "1." Our definition corresponds to that provided by Klug [Klu82] and Garcia-Molina et al. [GM00].

$$\begin{aligned}
\xi \triangleq \lambda r, G_1, \ldots, G_n, F_1, \ldots, F_m.(r = \perp) \rightarrow r,\\
&(\mathit{head}(r).G_1 \circ \ldots \circ \mathit{head}(r).G_n\\
&\quad \circ F_1(\mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r))) \circ \ldots\\
&\quad \circ F_m(\mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r))))\\
&\quad @\, \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r \setminus \mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r)))
\end{aligned}$$

The definition uses auxiliary function $\mathit{GetGroup} : [\mathcal{R} \times \mathcal{T} \times \Omega \times \ldots \times \Omega] \rightarrow \mathcal{R}$, which returns all tuples from the argument relation that have grouping-attribute values equal to those of the argument tuple.

$$\begin{aligned}
\mathit{GetGroup} \triangleq \lambda r, t, G_1, \ldots, G_n.(r = \perp) \rightarrow \mathit{undef},\\
&(t.G_1 = \mathit{head}(r).G_1 \wedge \ldots \wedge t.G_n = \mathit{head}(r).G_n) \rightarrow\\
&\qquad (\mathit{head}(r) @\, \mathit{GetGroup}_{G_1,\ldots,G_n}(\mathit{tail}(r), t)),\\
&\mathit{GetGroup}_{G_1,\ldots,G_n}(\mathit{tail}(r), t)
\end{aligned}$$

If there are no grouping attributes, the function returns a list with all tuples of the relation.

3.3.12 Temporal Aggregation

Operation $\xi^T : [\mathcal{R}^T \times \Omega^{nt} \times \ldots \Omega^{nt} \times \mathbb{F} \times \ldots \times \mathbb{F}] \rightarrow \mathcal{R}^T$ performs temporal aggregation according to given grouping attributes and pairs of aggregation functions with aggregation attributes. Set Ω^{nt} includes all non-temporal attributes of the schema of the argument relation—temporal attributes cannot be grouping or aggregation attributes.

The operation returns one tuple for each unique sequence of grouping attributes and for each "minimal" common time period of tuples that have equal values for the grouping attributes. The tuples of each group are sorted on the time attributes in ascending order. The schema of the result relation follows from the definition of concatenation. Our definition corresponds to the definition given in [KS95].

Let us consider an example query that counts the number of employees working on each project (see relation `PROJECT` in Figure 3). The query is expressed as $\xi^T_{\texttt{Prj},\texttt{COUNT(EmpName)}}(\texttt{PROJECT})$, and the result is shown in Figure 7. Temporal aggregation is defined next.

Prj	COUNT(EmpName)	T1	T2
P1	1	2	3
P1	1	7	8
P2	1	3	4
P2	2	5	6
P3	1	7	8
P3	2	9	10

Figure 7: Result of Temporal Aggregation

$$\begin{aligned}
\xi^T \triangleq \lambda r, G_1, \ldots, G_n, F_1, \ldots, F_m.(r = \perp) \rightarrow r,\\
&\mathit{OneGroupLoop}^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(\mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r)),\\
&\quad \mathit{minVal}\,(\mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r))), \mathit{maxVal}\,(\mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r))))\\
&\qquad \sqcup \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r \setminus^T \mathit{GetGroup}_{G_1,\ldots,G_n}(r, \mathit{head}(r)))
\end{aligned}$$

Function $\mathit{OneGroupLoop}^T : [\mathcal{R}^T \times \mathbb{T} \times \mathbb{T} \times \Omega^{nt} \times \ldots \Omega^{nt} \times \mathbb{F} \times \ldots \times \mathbb{F}] \to \mathcal{R}^T$ returns aggregate tuples for the argument relation and the argument time period, which is composed by the minimum and maximum time values found among the tuples in the group (as found by functions minVal and maxVal, defined in Appendix A). All tuples of the argument relation have the same grouping-attribute values. The function finds all "minimal" common time periods and outputs one tuple with aggregate values for each period.

$$\begin{aligned}\mathit{OneGroupLoop}^T \triangleq \lambda r, c_1, c_2, G_1, \ldots, G_n, F_1, \ldots, F_m.&(\mathit{MinTime}^T(r, c_1, c_2) = \mathit{undef}) \to \perp,\\ &(\mathit{head}(r).G_1 \circ \ldots \circ \mathit{head}(r).G_n\\ &\quad \circ F_1(\mathit{GetOverlapping}^T(r, c_1, \mathit{MinTime}^T(r, c_1, c_2)))\\ &\quad \circ \ldots \circ F_m(\mathit{GetOverlapping}^T(r, c_1, \mathit{MinTime}^T(r, c_1, c_2))))\\ &\quad @\ \mathit{OneGroupLoop}^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r, \mathit{MinTime}^T(r, c_1, c_2), c_2)\end{aligned}$$

Auxiliary function $\mathit{MinTime}^T : [\mathcal{R}^T \times \mathbb{T} \times \mathbb{T}] \to \mathbb{T}$, defined in Appendix A, scans the argument relation and returns the minimum timestamp value, which is bigger than the first argument timestamp value, but smaller than or equal to the second argument timestamp value.

Auxiliary function $\mathit{GetOverlapping}^T : [\mathcal{R}^T \times \mathbb{T} \times \mathbb{T}] \to \mathcal{R}^T$ returns all tuples from the argument relation that overlap with the period defined by the two argument timestamp values.

$$\begin{aligned}\mathit{GetOverlapping}^T \triangleq \lambda r, c_1, c_2.&(r = \perp) \to \mathit{perp},\\ &(\mathit{head}(r).\mathtt{T1} < c_2 \wedge \mathit{head}(r).\mathtt{T2} > c_1) \to\\ &\qquad \mathit{head}(r) @ \mathit{GetOverlapping}^T(\mathit{tail}(r), c_1, c_2),\\ &\mathit{GetOverlapping}^T(\mathit{tail}(r), c_1, c_2)\end{aligned}$$

Temporal aggregation may return at most $2 \cdot n(r) - 1$ tuples, where $n(r)$ is the cardinality of the argument relation. The reasoning is similar to the one given for temporal duplicate elimination.

3.3.13 Sorting

Function $\mathit{sort} : [\mathcal{R} \times \mathcal{O}_\Omega] \to \mathcal{R}$ 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 $\mathit{InsertTuple} : [\mathcal{T} \times \mathcal{R} \times \mathcal{O}_\Omega] \to \mathcal{R}$, which inserts a tuple into a sorted argument relation, maintaining its order. We denote the argument order by a.

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

Function $\mathit{MustPrecede} : [\mathcal{T} \times \mathcal{T} \times \mathcal{O}_\Omega] \to$ Boolean returns True if the first argument tuple must precede the second argument tuple according to the argument order.

Function sort invokes $\mathit{InsertTuple}$ for each of its tuples.

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

3.3.14 Coalescing

Operation $\mathit{coal}^T : \mathcal{R}^T \to \mathcal{R}^T$ coalesces value-equivalent tuples of the argument relation, but retains duplicates in snapshots. To effect this, all that is necessary is to coalesce those value-equivalent tuples that meet, i.e., if the time-period end of one tuple is equal to the time-period start of the other tuple. The argument and result relations have the same schema, where the non-temporal attribute values are denoted as $a_1, \ldots, a_n$.

$$\begin{aligned}\mathit{coal}^T \triangleq \lambda r.&(r = \perp) \to r,\\ &(\mathit{MeetTpl}^T(\mathit{head}(r), \mathit{tail}(r)) = \mathit{undef}) \to \mathit{head}(r) @ \mathit{coal}^T(\mathit{tail}(r)),\\ &\ \mathit{coal}^T((\mathit{head}(r).a_1 \circ \ldots \circ \mathit{head}(r).a_n\\ &\quad \circ \mathit{min}\ (\mathit{head}(r).\mathtt{T1}, \mathit{MeetTpl}^T(\mathit{head}(r), \mathit{tail}(r)).\mathtt{T1})\\ &\quad \circ \mathit{max}\ (\mathit{head}(r).\mathtt{T2}, \mathit{MeetTpl}^T(\mathit{head}(r), \mathit{tail}(r)).\mathtt{T2}))\\ &\quad @\ \mathit{remove}(\mathit{MeetTpl}^T(\mathit{head}(r), \mathit{tail}(r)), \mathit{tail}(r)))\end{aligned}$$

If a value-equivalent tuple that meets the first tuple exists, the operation combines into one the first tuple and the tuple that meets with it. Function $MeetTpl^T$, defined in Appendix A, finds the first tuple in the argument relation that meets and is value-equivalent with the argument tuple. Auxiliary functions max and min take single-attribute tuples as arguments, compare the values of those tuples, and return a new single-attribute tuple.

To perform coalescing with duplicate elimination, one has to perform temporal duplication elimination first and then coalesce as defined above. Alternatively, a combined operation may be defined.

Figure 8 shows relation R1 (from Figure 6) coalesced. The third and fifth tuples of R1 were merged into the third tuple of the result relation. The fourth tuple remains the same in the argument and result relations.

$\texttt{R4} = coal^T(\texttt{R1})$

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

Figure 8: Coalescing of Relation R1

3.3.15 Union

Operation $\cup : [\mathcal{R} \times \mathcal{R}] \to \mathcal{R}^{sn}$ returns the union of two argument relations while restricting the number of duplicates for each tuple to the maximum number of duplicates of that tuple in an argument relation. This operation is an extension of the union operation for multisets described in [Alb91]. The schemas of both argument relations and the result relation are the same, but, as in the Cartesian product all temporal attributes, if any, in the result are prefixed by "1." Union is defined via union ALL and difference.

$$\cup \triangleq \lambda r_1, r_2 . r_1 \sqcup (r_2 \setminus r_1)$$

3.3.16 Temporal Union

Operation $\cup^T : [\mathcal{R}^T \times \mathcal{R}^T] \to \mathcal{R}^T$ returns the temporal counterpart of the above-described operation, $\cup$. The upper bound for the cardinality of the result derives from the cardinalities of union ALL and temporal difference.

$$\cup^T \triangleq \lambda r_1, r_2 . r_1 \sqcup (r_2 \setminus^T r_1)$$

3.4 Mapping to the Algebra

The mapping of a user-level query to an algebra expression depends on the specific user-level language adopted, but our operations are sufficient for SQL and a wide range of temporal query languages [BJ97, Sno95, EJS98]. Temporal duplicate elimination, temporal difference, temporal aggregation, and temporal Cartesian product (followed by an appropriate projection; recall Section 3.3.6) are snapshot-reducible to their regular counterparts, simplying the mapping from languages that have built-in temporal semantics. Selection, projection, union ALL, and sorting do not have temporal counterparts, as they are snapshot-reducible to themselves when their parameters do not involve the time attributes.

3.5 Example Query

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

Consider temporal relations EMPLOYEE and PROJECT from Figures 3 and 5 and the query "What 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 desired result of the query is given in Figure 9. Anna worked in Sales from February through May, but she was on project P2 during March and May, and so the result includes just two months during this time, February and

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 9: The Result Relation

April. Anna also worked in Advertising during this period, but the user requested that duplicates in the snapshots be eliminated. Finally, notice that no value-equivalent tuples have adjacent time periods, and that the result is sorted on `EmpName`.

To compute this result, the stratum initially uses a straightforward mapping of the user-level query to an initial algebra expression, shown in Figure 10(a). The last operation applied, T^S, transfers its argument from the DBMS to the stratum; it is initially assumed that the query is entirely computed in the DBMS. 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 last operation ensures that no snapshots have duplicates, and the first operation ensures that value-equivalent tuples with adjacent time periods are merged.

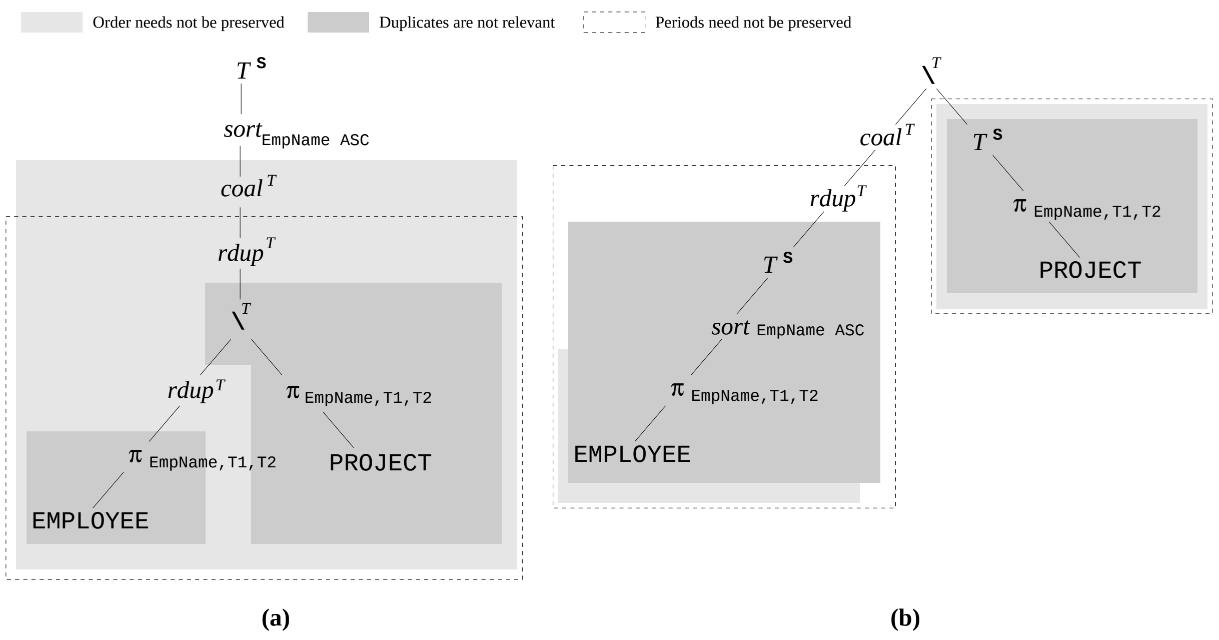
The temporal difference ($\backslash^T$) returns the employees in `EMPLOYEE`, but not in `PROJECT`, along with the time periods when this occurred. To obtain the correct result, the left argument is not allowed to contain duplicates in its snapshots; this is ensured by the $rdup^T$ operation. Duplicate elimination is necessary because temporal difference is sensitive to duplicates. For example, Anna worked in two departments, but on only one project in March; thus, temporal difference would include one tuple for Anna for March in the result. However, this would be wrong because the query requires only those times when employees worked in some department, but did not work on *any* project. (Difference and temporal difference are analogous to SQL's `EXCEPT ALL` in their handling of duplicates; the stated query is more similar to SQL's `EXCEPT`, which requires the left-hand $rdup^T$ to yield the correct result.)

Transformation rules that preserve different types of equivalences are applicable to different parts of a query. This is illustrated by the regions in Figure 10(a). First, transformations below the $sort$ need not preserve order; this is indicated by the lighter shading. The operations below $sort$ are not sensitive to order, and the $sort$ ensures that whatever result is produced by the operations below, this 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; this is indicated by the darker shading. 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 periods (indicated by the dashed line); 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 6 and 7 formalize these concepts and give a procedure for determining these regions in a query.

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 10(b). In this tree, the transfer operation has been moved below the temporal difference operation, indicating that the stratum performs temporal duplicate elimination, coalescing, and temporal 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



Order needs not be preserved
Duplicates are not relevant
Periods need not be preserved
T^S
$sort_{\text{EmpName ASC}}$
$coal^T$
$rdup^T$
$\backslash^T$
$rdup^T$
$\pi_{\text{EmpName,T1,T2}}$
$\pi_{\text{EmpName,T1,T2}}$
EMPLOYEE
PROJECT
(a)
$\backslash^T$
$coal^T$
$rdup^T$
T^S
$sort_{\text{EmpName ASC}}$
$\pi_{\text{EmpName,T1,T2}}$
EMPLOYEE
T^S
$\pi_{\text{EmpName,T1,T2}}$
PROJECT
(b)

Figure 10: Algebraic Expressions for the Query “Which employees worked in a department, but not on any project, and when?”

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, due to the splitting of timestamp periods, as observed in Figure 9.

We use this example throughout the paper.

4 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, six types of equivalences between relations are introduced: list equivalence ($\equiv_L$), multiset equivalence ($\equiv_M$), set equivalence ($\equiv_S$), snapshot list equivalence ($\equiv^S_L$), snapshot multiset equivalence ($\equiv^S_M$), and snapshot set equivalence ($\equiv^S_S$).

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. 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).

Definition 4.1 Relations r_1 and r_2 are *list equivalent*, $r_1 \equiv_L r_2$, if and only if function $\equiv_L \ : [\mathcal{R} \times \mathcal{R}] \rightarrow$ Boolean returns True.

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

□

Definition 4.2 Relations r_1 and r_2 are *multiset equivalent*, $r_1 \equiv_M r_2$, if and only if function $\equiv_M \ : [\mathcal{R} \times \mathcal{R}] \rightarrow$ Boolean returns True.

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

Definition 4.3 Relations r_1 and r_2 are *set equivalent*, $r_1 \equiv_S r_2$, if and only if function $\equiv_S \ : [\mathcal{R} \times \mathcal{R}] \rightarrow$ Boolean returns True.

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

Auxiliary function $\mathit{RemoveAll} : [\mathcal{T} \times \mathcal{R}] \rightarrow \mathcal{R}$ removes all occurences of the argument tuple from the argument relation. The schema of the argument relation is retained for the result relation.

$$\begin{aligned}\mathit{RemoveAll} \triangleq \lambda t, r.&(r = \perp) \rightarrow \perp,\\ &(t = \mathit{head}(r)) \rightarrow \mathit{RemoveAll}(t, \mathit{tail}(r)),\\ &\quad \mathit{head}(r) @ \mathit{RemoveAll}(t, \mathit{tail}(r))\end{aligned}$$

Definition 4.4 Temporal relations r_1 and r_2 are *snapshot list equivalent*, $r_1 \equiv^S_L r_2$, if and only if function $\equiv^S_L$: $[\mathcal{R}^T \times \mathcal{R}^T] \rightarrow$ Boolean returns True.

$$\equiv^S_L \triangleq \lambda r_1, r_2.\mathit{SnapshotListEquiv}(r_1, r_2, 0, \texttt{MAX\_TIMESTAMP})$$ □

Auxiliary function $\mathit{SnapshotListEquiv} : [\mathcal{R}^T \times \mathcal{R}^T \times \mathbb{T} \times \mathbb{T}] \rightarrow$ Boolean computes if two argument relations are snapshot equivalent during the argument time interval.

$$\begin{aligned}\mathit{SnapshotListEquiv} \triangleq \lambda r_1, r_2, c_1, c_2.&(c_1 = c_2) \rightarrow \text{True},\\ &(\tau^T_{c_1}(r_1) \equiv_L \tau^T_{c_1}(r_2)) \rightarrow \mathit{SnapshotListEquiv}(r_1, r_2, c_1 + 1, c_2),\\ &\quad \text{False}\end{aligned}$$

The timeslice operation $\tau^T : [\mathbb{T} \times \mathcal{R}^T] \rightarrow \mathcal{R}^{sn}$ returns all tuples of the argument relation that overlap with the given time chronon, and removes the temporal attributes. Let the schema of the argument relation be $S = (\Omega, \Delta, \mathit{dom})$, where the non-temporal attributes from Ω are $a_1, \ldots, a_n$. The schema of the result relation is $S^{sn} = (\Omega^{sn}, \Delta^{sn}, \mathit{dom}^{sn})$, where $\Omega^{sn} = \{a_1, \ldots, a_n\}$, $\Delta^{sn} = \Delta \setminus \mathbb{T}$, and dom^{sn} is as dom, but without the mappings to type $\mathbb{T}$.

$$\begin{aligned}\tau^T \triangleq \lambda c, r.&(\mathit{head}(r).\texttt{T1} \leq c < \mathit{head}(r).\texttt{T2}) \rightarrow \pi_{a_1,\ldots,a_n}(\langle \mathit{head}(r) \rangle) \ @ \ \tau^T_c(\mathit{tail}(r)),\\ &\tau^T_c(\mathit{tail}(r))\end{aligned}$$

The timeslice operation may introduce duplicates because, like the projection operation, it removes attributes.

We omit the definitions of snapshot multiset and snapshot set equivalence because they are similar to the definition of snapshot list equivalence.

We can exemplify different types of equivalences using different variations of the EMPLOYEE relation (Figure 3) projected on EmpName and the temporal attributes. Figure 6 gives three different instances of this schema (relation R1: without duplicate elimination, relation R2: with duplicate elimination, and relation R3: with temporal duplicate elimination, respectively). Figure 8 gives the coalesced version (relation R4) of the projected relation. Figure 11 gives the result of the projection, followed by sorting (relation S1; $A = \langle(\texttt{EmpName}, \texttt{ASC}), (\texttt{T1}, \texttt{ASC}), (\texttt{T2}, \texttt{ASC})\rangle$) and sorting and coalescing (relation S2).

Relation S1 is multiset and set equivalent to relation R1 because both contain the same tuples, which occur the same number of times. Their snapshots at any point in time are also equivalent as multisets and sets. Neither the relations nor their snapshots are equivalent as lists because the orderings of the tuples are different.

Relations S1 and R2 are not equivalent as lists or as multisets: the orderings of the tuples are different, and the tuple for Anna with times 2 and 6 occurs twice in S1, but once in R2. However, the $\equiv_S$ equivalence holds because the two relations contain the same tuples. Snapshot equivalences between S1 and R2 are undefined because relation R2 is non-temporal.

$\texttt{S1} = sort_A(\texttt{R1})$

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

$\texttt{S2} = coal^T(\texttt{S1})$

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

Figure 11: Variations of Relation EMPLOYEE Projected on EmpName and the Temporal Attributes

Relations S1 and R3 have different tuples, e.g., the tuple for John with times 6 and 11 is present in S1, but not in R3; thus, they are not equivalent as lists, multisets, or sets. Their snapshots are also not equivalent as lists because of different orderings, and they are not equivalent as multisets because the snapshot of S1 at times between 2 and 6 contains two tuples for Anna, while snapshots of relation R3 never contain more than one tuple for Anna. Only equivalence $\equiv_S^S$ holds between relations S1 and R3, meaning that their snapshots are equivalent as sets. For example, S1 and R3 both have the snapshot (as a set) $\{(\texttt{Anna}), (\texttt{John})\}$ at time 3.

Relations S1 and S2 also contain different tuples and are not equivalent as lists, multisets, or sets. However, at each point in time, their snapshots are equivalent as lists, multisets, and sets. Since relation R4, which contains the same tuples as S2, is not sorted the same way as S1 and S2, only equivalences $\equiv_M^S$ and $\equiv_S^S$ hold between S1 and R4

The examples illustrate that we have an ordering between the types of equivalences. For example, two temporal relations being equivalent as multisets implies that they are also equivalent as sets and that their snapshots are equivalent as multisets and sets. We list all implications in the following theorem.

Theorem 4.1 Let r_1 and r_2 be relations. Then the following implications hold. (Implications pointing downward apply only to temporal relations.)

$$\begin{tikzcd}
r_1 \equiv_L r_2 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & r_1 \equiv_M r_2 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & r_1 \equiv_S r_2 \arrow[d, Rightarrow] \\
r_1 \equiv_L^S r_2 \arrow[r, Rightarrow] & r_1 \equiv_M^S r_2 \arrow[r, Rightarrow] & r_1 \equiv_S^S r_2
\end{tikzcd}$$

Proof: Appendix B. □

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 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, when evaluated on the same argument relation(s). That said, the result relations will also be multiset and set equivalent, as well as equivalent according to all three types of snapshot equivalences.

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. (It is also acceptable if the result needs to be snapshot multiset equivalent to the result relation produced by the original plan.) For example, query $\pi_L(\texttt{EMPLOYEE})$ (resulting in relation R1) can return tuples in any order. In general, the type of the result specified by a query determines which transformation rules can be exploited. Section 5 lists all transformation rules, and Sections 6 and 7 describe a mechanism for determining when a transformation rule is applicable.

5 Transformation Rules

In this section, we provide a set of transformation rules for the algebra, which goes beyond all existing rule sets known to the authors. First, we describe transformation rules that derive from the conventional relational algebra. We consider when the existing rules for sets and multisets apply for lists, and we add rules for temporal operations. Then, we discuss duplicate elimination, coalescing, sorting, and transfer rules. (The latter type is specific to the stratum architecture.)

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 4; 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 transformation rules can be verified formally, as the operations and equivalence types have formal definitions. Unlike rules expressed informally, which sometimes later have been found to be in error [Kie85], our rules are theorems amenable to formal proof. Appendix B provides an example proof of one transformation rule. While we believe the other transformations are correct, we have not written out all 90-odd proofs. An automatic theorem prover would be useful in constructing these proofs, which can be quite repetitive.

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.

5.1 Conventional Transformation Rules

The conventional transformation rules derive from the rules for multisets given by [GM00]. Figure 12 shows the conventional transformation rules that do not involve temporal operations. The rules are ordered based on the operation they concern, e.g., rules G1–G5 concern selection. We can distinguish between rules depending on what kind of equivalence they support. First, most rules are valid for lists, e.g., pushing selection down before a Cartesian product or a difference (rules G10, G15) guarantees the list equivalence between the result relations.

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 G9 and G17). Note that unlike in the set- or multiset-based algebras, the order of the arguments to these operations cannot be changed freely because this affects the ordering of the result.

A few rules involving union ALL and regular and temporal union (e.g., rule G2), have equivalence types weaker than $\equiv_M$. Rule G2 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. If we use the union operation, the $\equiv_M$ equivalence type can be achieved (rule G3).

All of these transformations apply equally to nontemporal and temporal relations; and those transformations defined over more than one argument relation also apply to combinations of nontemporal and temporal relations. Rule G5 is an exception and holds only for nontemporal relations. The reason is that regular difference prefixes temporal attributes, and so we need a slightly modified rule for such relations.

Figure 13 shows conventional transformation rules that involve temporal operations. Most rules are counterparts of the rules given in Figure 12; in some cases, pre-conditions involving the temporal attributes apply (e.g., in rule G27). Rule G25 corresponds to rule G5, but has the condition that r should be temporal and involves a projection introducing temporal attributes in the result of regular difference. Since the temporal Cartesian product retains the original temporal attributes, they have to be removed from the result of the two subsequent products (rule G30).

5.2 Duplicate Elimination Transformation Rules

Duplicate elimination rules are given in Figure 14. Rules D1–D4 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 (recall relations R1 and R3 from Figure 6).

Contrary to the commonly considered union ALL and the regular SQL union (which removes duplicates from the result relation of union ALL), 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 (rules D12 and D13).

Rules D14 and D15 follow because aggregations involving only functions MIN and MAX are insensitive to duplicates.

Duplicate elimination cannot be pushed before union ALL because this operation may generate duplicates even if its argument relations do not contain any. Also, duplicate elimination cannot be pushed down before regular (temporal)

(G1)	$\sigma_{P_1 \wedge P_2}(r) \equiv_L \sigma_{P_1}(\sigma_{P_2}(r))$	
(G2)	$\sigma_{P_1 \vee P_2}(r) \equiv_S \sigma_{P_1}(r) \sqcup \sigma_{P_2}(r)$	
(G3)	$\sigma_{P_1 \vee P_2}(r) \equiv_M \sigma_{P_1}(r) \cup \sigma_{P_2}(r)$	
(G4)	$\sigma_{P_1}(\sigma_{P_2}(r)) \equiv_L \sigma_{P_2}(\sigma_{P_1}(r))$	
(G5)	$\sigma_{\neg P}(r) \equiv_L r \setminus \sigma_P(r)$	`[lr]` $\texttt{T1} \notin \Omega_r \wedge \texttt{T2} \notin \Omega_r$
(G6)	$\pi_{f_1,\ldots,f_n}(\pi_{h_1,\ldots,h_m}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(r)$	`[lr]` $attr(f_1, \ldots, f_n) \subseteq \Omega_r$
		`[rl]` $attr(h_1, \ldots, h_m) \subseteq \Omega_r$
(G7)	$\pi_{f_1,\ldots,f_n}(\sigma_P(r)) \equiv_L \sigma_P(\pi_{f_1,\ldots,f_n}(r))$	`[lr]` $attr(P) \subseteq attr(f_1, \ldots, f_n)$
(G8)	$\pi_{f_1,\ldots,f_n}(\sigma_P(r)) \equiv_L \pi_{f_1,\ldots,f_n}(\sigma_P(\pi_{h_1,\ldots,h_m}(r)))$,	`[rl]` $attr(P) \subseteq \Omega_r$
	where $h_i = \{a \mid i \in \{1, \ldots, m\} \wedge (h_i \in \{f_1, \ldots, f_n\} \vee h_i \in attr(P))\}$	
(G9)	$r_1 \times r_2 \equiv_M r_2 \times r_1$	
(G10)	$\sigma_P(r_1 \times r_2) \equiv_L \sigma_P(r_1) \times r_2$	`[lr]` $attr(P) \subseteq \Omega_{r_1}$
(G11)	$\sigma_P(r_1 \times r_2) \equiv_L r_1 \times \sigma_P(r_2)$	`[lr]` $attr(P) \subseteq \Omega_{r_2}$
(G12)	$\pi_{f_1,\ldots,f_n}(r_1 \times r_2) \equiv_L \pi_{A_1}(r_1) \times \pi_{A_2}(r_2)$, where	`[lr]` $\forall i \in \{1, \ldots, n\}\ attr(f_i) \subseteq \Omega_{r_1} \vee attr(f_i) \subseteq \Omega_{r_2}$
	$A_1 = \{f_i \mid i \in \{1, \ldots, n\} \wedge attr(f_i) \subseteq \Omega_{r_1}\}$,	`[rl]` $attr(A_1) \cap attr(A_2) = \emptyset$
	$A_2 = \{f_i \mid i \in \{1, \ldots, n\} \wedge attr(f_i) \subseteq \Omega_{r_2}\}$	
(G13)	$\pi_{f_1,\ldots,f_n}(r_1 \times r_2) \equiv_L \pi_{f_1,\ldots,f_n}(\pi_{A_1}(r_1) \times \pi_{A_2}(r_2))$,	`[rl]` $attr(f_1, \ldots, f_n) \subseteq \Omega_{r_1 \times r_2}$
	where $A_1 = \{a \mid a \in \Omega_{r_1} \wedge a \in attr(f_1, \ldots, f_n)\}$,	
	$A_2 = \{a \mid a \in \Omega_{r_2} \wedge a \in attr(f_1, \ldots, f_n)\}$	
(G14)	$(r_1 \times r_2) \times r_3 \equiv_L r_1 \times (r_2 \times r_3)$	
(G15)	$\sigma_P(r_1 \setminus r_2) \equiv_L \sigma_P(r_1) \setminus r_2$	
(G16)	$\sigma_P(r_1 \setminus r_2) \equiv_L \sigma_P(r_1) \setminus \sigma_P(r_2)$	
(G17)	$r_1 \sqcup r_2 \equiv_M r_2 \sqcup r_1$	
(G18)	$\sigma_P(r_1 \sqcup r_2) \equiv_L \sigma_P(r_1) \sqcup \sigma_P(r_2)$	
(G19)	$\pi_{f_1,\ldots,f_n}(r_1 \sqcup r_2) \equiv_L \pi_{f_1,\ldots,f_n}(r_1) \sqcup \pi_{f_1,\ldots,f_n}(r_2)$	
(G20)	$r_1 \cup r_2 \equiv_M r_2 \cup r_1$	
(G21)	$\sigma_P(r_1 \cup r_2) \equiv_L \sigma_P(r_1) \cup \sigma_P(r_2)$	
(G22)	$\pi_{f_1,\ldots,f_n}(r_1 \cup r_2) \equiv_S \pi_{f_1,\ldots,f_n}(r_1) \cup \pi_{f_1,\ldots,f_n}(r_2)$	
(G23)	$\sigma_P(\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)) \equiv_L \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(\sigma_P(r))$	$attr(P) \subseteq \{G_1, \ldots, G_n\}$
(G24)	$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r) \equiv_L \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(\pi_L(r))$	$attr(G_1, \ldots, G_n, F_1, \ldots, F_m) \subseteq L$

Figure 12: Conventional Transformation Rules

(G25) $\sigma_{\neg P}(r) \equiv_L \pi_{\Omega_r \setminus \{\mathtt{T1},\mathtt{T2}\},1.\mathtt{T1}\ \mathtt{AS}\ \mathtt{T1},2.\mathtt{T2}\ \mathtt{AS}\ \mathtt{T2}}(r \setminus \sigma_P(r))$ $[\mathtt{lr}]\ \mathtt{T1} \in \Omega_r \wedge \mathtt{T2} \in \Omega_r$

(G26) $r_1 \times^T r_2 \equiv_M r_2 \times^T r_1$
(G27) $\sigma_P(r_1 \times^T r_2) \equiv_L \sigma_P(r_1) \times^T r_2$ $[\mathtt{lr}]\ attr(P) \subseteq \Omega_{r_1} \wedge \mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
$[\mathtt{rl}]\ \mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
(G28) $\sigma_P(r_1 \times^T r_2) \equiv_L r_1 \times^T \sigma_P(r_2)$ $[\mathtt{lr}]\ attr(P) \subseteq \Omega_{r_2} \wedge \mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
$[\mathtt{rl}]\ \mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
(G29) $\pi_{f_1,\dots,f_n}(r_1 \times^T r_2) \equiv_L \pi_{f_1,\dots,f_n}(\pi_{A_i}(r_1) \times^T \pi_{A_2}(r_2))$, $[\mathtt{rl}]\ attr(f_1,\dots,f_n) \subseteq \Omega_{r_1 \times^T r_2}$
where $A_1 = \{f_i \mid i \in \{1,\dots,n\} \wedge attr(f_i) \subseteq \Omega_{r_1}\} \cup \{\mathtt{T1},\mathtt{T2}\}$,
$A_2 = \{f_i \mid i \in \{1,\dots,n\} \wedge attr(f_i) \subseteq \Omega_{r_2}\} \cup \{\mathtt{T1},\mathtt{T2}\}$

(G30) $\pi_{A_1}((r_1 \times^T r_2) \times^T r_3) \equiv_L \pi_{A_2}(r_1 \times^T (r_2 \times^T r_3))$, where
$A_1 = \Omega_{(r_1 \times^T r_2) \times^T r_3} \setminus OrigTimestamps$,
$A_2 = \Omega_{r_1 \times^T (r_2 \times^T r_3)} \setminus OrigTimestamps$,
$OrigTimestamps \triangleq \{x_1. \dots x_n.T \mid x_1 \in \{1,2\} \wedge \dots \wedge x_n \in \{1,2\} \wedge T \in \{\mathtt{T1},\mathtt{T2}\} \wedge n \in \mathbb{N}\}$

(G31) $\sigma_P(r_1 \setminus^T r_2) \equiv_L \sigma_P(r_1) \setminus^T r_2$ $\mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
(G32) $\sigma_P(r_1 \setminus^T r_2) \equiv_L \sigma_P(r_1) \setminus^T \sigma_P(r_2)$ $\mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$

(G33) $r_1 \cup^T r_2 \equiv_M^S r_2 \cup^T r_1$
(G34) $\sigma_P(r_1 \cup^T r_2) \equiv_L \sigma_P(r_1) \cup^T \sigma_P(r_2)$ $\mathtt{T1} \notin attr(P) \wedge \mathtt{T2} \notin attr(P)$
(G35) $\pi_{f_1,\dots,f_n,\mathtt{T1},\mathtt{T2}}(r_1 \cup^T r_2) \equiv_S^S \pi_{f_1,\dots,f_n,\mathtt{T1},\mathtt{T2}}(r_1) \cup^T \pi_{f_1,\dots,f_n,\mathtt{T1},\mathtt{T2}}(r_2)$

(G36) $\sigma_P(\xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r)) \equiv_L \xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(\sigma_P(r))$ $attr(P) \subseteq \{G_1,\dots,G_n\}$
(G37) $\xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r) \equiv_L \xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(\pi_{L,\mathtt{T1},\mathtt{T2}}(r))$ $attr(G_1,\dots,G_n,F_1,\dots,F_m) \subseteq L$

Figure 13: Conventional Transformation Rules Involving Temporal Operations

difference, because both difference operations are sensitive to the number of duplicates in both arguments. If tuple t occurs x times in the first argument relation and y times in the second argument relation ($y < x$), it occurs $x - y$ times in the result of regular difference. However, if we were to remove duplicates first, tuple t would occur only once in each argument to the regular difference, and it would be absent from the result.

If duplication elimination is applied after an operation that does not manufacture duplicates, we can remove the duplicate elimination using rules D1 and D2. Regular duplicate elimination can be removed if it is performed on top of regular (or temporal) duplicate elimination or regular (or temporal) aggregation. Temporal duplicate elimination can be removed, if it is performed on top of temporal duplicate elimination or temporal aggregation. Hence, rules D1 and D2 imply the following rules.

$$rdup(rdup(r)) \equiv_L rdup(r)$$
$$rdup^T(rdup^T(r)) \equiv_L rdup(r)$$
$$rdup(rdup^T(r)) \equiv_L rdup(r)$$
$$rdup(\xi_{G_1,\dots,G_n,F_1,\dots,F_m}(r)) \equiv_L \xi_{G_1,\dots,G_n,F_1,\dots,F_m}(r)$$
$$rdup(\xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r)) \equiv_L \xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r)$$
$$rdup^T(\xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r)) \equiv_L \xi^T_{G_1,\dots,G_n,F_1,\dots,F_m}(r)$$

5.3 Coalescing Transformation Rules

Rules C1 and C2 in Figure 15 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–C8).

Our list of coalescing transformations extends those provided by Böhlen et al. [BSS96]. Due to the differences in coalescing definitions (see Section 3.3) and because [BSS96] allows duplicates in snapshots of temporal relations, but not regular duplicates, the following three transformation rules (given in [BSS96]) have only type $\equiv_M^S$ and are derivable from rule C2.

(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)	$rdup(\sigma_P(r)) \equiv_L \sigma_P(rdup(r))$	
(D6)	$rdup^T(\sigma_P(r)) \equiv_L \sigma_P(rdup^T(r))$	$\texttt{T1} \notin attr(P) \wedge \texttt{T2} \notin attr(P)$
(D7)	$rdup(\pi_{f_1,\ldots,f_n}(rdup(r))) \equiv_L rdup(\pi_{f_1,\ldots,f_n}(r))$	
(D8)	$rdup^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(rdup^T(r))) \equiv_L rdup^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(r))$	
(D9)	$rdup(r_1 \times r_2) \equiv_L rdup(r_1) \times rdup(r_2)$	
(D10)	$rdup(r_1 \times^T r_2) \equiv_L rdup(r_1) \times^T rdup(r_2)$	
(D11)	$rdup^T(\pi_A(r_1 \times^T r_2)) \equiv_L \pi_A(rdup^T(r_1) \times^T rdup^T(r_2))$, where $A = \Omega_{r_1 \times^T r_2} \setminus \{1.\texttt{T1}, 1.\texttt{T2}, 2.\texttt{T1}, 2.\texttt{T2}\}$	
(D12)	$rdup(r_1 \cup r_2) \equiv_L rdup(r_1) \cup rdup(r_2)$	
(D13)	$rdup^T(r_1 \cup^T r_2) \equiv_L rdup^T(r_1) \cup^T rdup^T(r_2)$	
(D14)	$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(rdup(r)) \equiv_L \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$	$AggrFunctions(F_1,\ldots,F_m) \subset \{\texttt{MIN},\texttt{MAX}\}$
(D15)	$\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(rdup^T(r)) \equiv_L \xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$	$AggrFunctions(F_1,\ldots,F_m) \subset \{\texttt{MIN},\texttt{MAX}\}$

Figure 14: Duplicate Elimination Transformation Rules

(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)	$coal^T(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(coal^T(r))))) \equiv_L coal^T(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(r))))$	
(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
(C11)	$coal^T(rdup^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(coal^T(r)))) \equiv_L coal^T(rdup^T(\pi_{f_1,\ldots,f_n,\texttt{T1},\texttt{T2}}(r)))$	r does not have duplicates in snapshots

Figure 15: Coalescing Transformation Rules

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

The transformation rules have $\equiv_M^S$ type because Cartesian product, temporal difference, and projection destroy coalescing. The projection in the first rule is necessary because the temporal Cartesian product retains the timestamps of its arguments. The first transformation can be modified to have type $\equiv_L$ if we require that the arguments do not have duplicates in snapshots (rule C9). Adding the same requirement, the second 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. For the third rule, we not only have to add the same requirement as for rules C9 and C10 but also to eliminate duplicates before the top coalescing—otherwise projection would have potentially introduced duplicates in snapshots, leading to different tuples in the result.

5.4 Sorting Transformation 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.

(S1)	$sort_A(r) \equiv_L r$	$IsPrefixOf(A, Order(r))$
(S2)	$sort_A(r) \equiv_M r$	
(S3)	$sort_A(sort_B(r)) \equiv_L sort_A(r)$	$IsPrefixOf(B, A)$
(S4)	$sort_A(\sigma_P(r)) \equiv_L \sigma_P(sort_A(r))$	
(S5)	$sort_A(\pi_{f_1,\ldots,f_n}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(sort_A(r))$	[lr] $attr(A) \subseteq \Omega_r$
		[rl] $attr(A) \subseteq attr(f_1, \ldots, f_n)$
(S6)	$sort_A(r_1 \times r_2) \equiv_L sort_A(r_1) \times r_2$	[lr] $attr(A) \subseteq \Omega_{r_1}$
(S7)	$sort_A(r_1 \times^T r_2) \equiv_L sort_A(r_1) \times^T r_2$	[lr] $attr(A) \subseteq \Omega_{r_1} \wedge \texttt{T1} \notin attr(A) \wedge \texttt{T2} \notin attr(A)$
		[rl] $\texttt{T1} \notin attr(A) \wedge \texttt{T2} \notin attr(A)$
(S8)	$sort_A(r_1 \setminus r_2) \equiv_L sort_A(r_1) \setminus r_2$	
(S9)	$sort_A(r_1 \setminus^T r_2) \equiv_L sort_A(r_1) \setminus^T r_2$	$\texttt{T1} \notin attr(A) \wedge \texttt{T2} \notin attr(A)$
(S10)	$sort_A(\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)) \equiv_L \xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}(sort_A(r))$	$attr(A) \subseteq \{G_1, \ldots, G_n\}$
(S11)	$sort_A(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)) \equiv_L \xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(sort_A(r))$	$attr(A) \subseteq \{G_1, \ldots, G_n\}$
(S12)	$sort_A(coal^T(r)) \equiv_L coal^T(sort_A(r))$	$\texttt{T1} \notin attr(A) \wedge \texttt{T2} \notin attr(A)$
(S13)	$sort_A(rdup(r)) \equiv_L rdup(sort_A(r))$	
(S14)	$sort_A(rdup^T(r)) \equiv_L rdup^T(sort_A(r))$	$\texttt{T1} \notin attr(A) \wedge \texttt{T2} \notin attr(A)$

Figure 16: Sorting Transformation Rules

Rule S3 requires B to be a prefix of A. If A is a prefix of B, we can eliminate $sort_A$ from the left-hand side of rule S3 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 preserves the ordering. All operations, except $\sqcup$, $\cup$, and $\cup^T$, fully or partially preserve the ordering of their first argument.

5.5 Transfer Transformation Rules

The transfer transformation rules are essential 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 stratum architecture).

Transfer operations can be applied only to relations that are in the appropriate location, e.g., T^S can only be applied to a relation in the DBMS. This implies that any path from a leaf to the root of a valid expression must encounter a non-empty alternation of T^S and T^D, starting and ending with T^S (since the data starts in the database and end up in the stratum, to be subsequently sent to the application).

Figure 17 gives general transformation rules on generic operations, denoted by *op*. A rule transferring operation *op* to the stratum can be applied only if this operation has an implementation in the stratum, and a rule transferring operation *op* to the DBMS can be applied only if *op* can be translated into SQL. For example, one instance of transformation rule T1 is $\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r) \equiv_M T^D(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(T^S(r)))$.

Several rules, e.g., T5 and T6, are of equivalence type $\equiv_{L,A}$, where A is the order list specified by the $sort$ operation. Two relations are $\equiv_{L,A}$ equivalent if they are $\equiv_M$ equivalent and their projections on A are $\equiv_L$ equivalent. The $\equiv_{L,A}$ equivalence is a slightly less restrictive equivalence than $\equiv_L$; the $\equiv_L$ equivalence implies $\equiv_{L,A}$ equivalence.

If a rule transfers an operation from the stratum to the DBMS, or vice versa, the relations produced by the left-hand and right-hand sides 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 $\equiv_L$ transformation rules given in Sections 5.1–5.4 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 Figures 12–16.

(T1)	$op\,(r) \equiv_M T^D(\,op\,(T^S(r))$
(T2)	$r_1\,op\,r_2 \equiv_M T^D(T^S(r_1)\,op\,T^S(r_2))$
(T3)	$op\,(r) \equiv_M T^S(\,op\,(T^D(r))$
(T4)	$r_1\,op\,r_2 \equiv_M T^S(T^D(r_1)\,op\,T^D(r_2))$
(T5)	$sort_A(r) \equiv_{L,A} T^D(sort_A(T^S(r))$
(T6)	$sort_A(r) \equiv_{L,A} T^S(sort_A(T^D(r))$
(T7)	$T^S(T^D(r)) \equiv_M r$
(T8)	$T^D(T^S(r)) \equiv_M r$
(T9)	$T^S(T^D(sort_A(r))) \equiv_{L,A} sort_A(r)$
(T10)	$T^D(T^S(sort_A(r))) \equiv_{L,A} sort_A(r)$

Figure 17: Transfer Transformation Rules

6 Applicability of Transformation Rules

Queries expressed in some user-level query language are mapped to an initial algebraic expression, to which the optimizer then applies transformation rules according to some given strategy. The resulting, new algebraic expressions must, when evaluated, return a relation that is 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 query language used and on the actual query statement.

Having a query plan, we name the required equivalence between results the *top equivalence* and assign it to the root of the query tree. We then propagate the required equivalences to the operations below 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 top equivalence. For example, if operation $rdup$ is placed at the root and the top equivalence is $\equiv_M$, an operation below $rdup$ requires only $\equiv_S$ equivalence because arbitrarily introducing or removing duplicates does not affect the top equivalence.

The required equivalences constrain the types of transformation rules that can be applied during query plan enumeration. There are no restrictions on when rules with equivalence type $\equiv_L$ can be applied—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. Although this does not hold for any of the other five types of rules, such rules may still be applicable. In the example above, an $\equiv_S$ rule may be applied to the query part below $rdup$.

Using some temporal variants of SQL, e.g., [BJ97], as the user-level language, the top equivalence is $\equiv_M$ or $\equiv_{L,A}$, depending on whether the query given includes `ORDER BY A`. The presence of `ORDER BY` specifies a result relation that is a list, but if `ORDER BY` does not occur, the query specifies a multiset, and the order of the result tuples is 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 in the user-level query. The top equivalence cannot be one of the snapshot-equivalence types, for queries that must faithfully preserve the time periods from the base relations cannot arbitrarily return any of the snapshot-equivalent result relations. However, snapshot-equivalence type rules can be applied when they satisfy the equivalence type between the results of the original plan and the new plan; we describe those cases below. Other temporal variants of SQL [EJS98] may have statements that call for only snapshot-list or snapshot-multiset equivalence. The mapping from the user-level language to the algebra should indicate the top equivalence required, as required by that language's semantics.

First, we consider an operation tree for an example query and describe which types of transformation rules can be applied to which locations. To enable the formal procedure of determining when a transformation rule is applicable to a query plan, we then introduce properties for the operations in an operation tree. Section 6.2 defines the properties, and Section 6.3 describes how to update them during query optimization. Finally, Section 7 describes how to use those properties to determine the applicability of transformation rules.

6.1 Example

Consider again the operation tree given in Figure 10(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 to be preserved (the lighter shaded region), we can apply $\equiv_M$ transformation rules. Specifically, in the subtree below the $sort$ operation, relations may be treated 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 elimination operation, which ensures that the left argument of the temporal difference does not contain duplicates in snapshots (see Section 3.5). (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, indicated by the dashed region. 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 the coalescing operation, $\equiv_M^S$ rules can be applied; $\equiv_S^S$ rules can be applied where duplicates are not relevant.

The next section describes how the shaded regions are determined.

6.2 Definitions of Properties

Table 2 introduces three Boolean properties of operations, which correspond to the shaded regions in Figure 10. Each operation in a tree has values for these properties. For each combination of the property values, Table 3 gives an equivalence type that should hold for results of that operation. Two relations are $\equiv_{L,A}^S$ equivalent if they are $\equiv_M^S$ equivalent and their projections on A and the time attributes are $\equiv_L^S$ equivalent. The time attributes are needed for the latter equivalence to be defined.

Property Name	**Description**
$OrderRequired$	True **if** the result of the operation must preserve some ordering
$DuplicatesRelevant$	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: Properties of an Operation in an Operation Tree

$OrderRequired(op)$	$DuplicatesRelevant(op)$	$PeriodPreserving(op)$	Equivalence type
True	True	True	$\equiv_{L,A}$
True	True	False	$\equiv_{L,A}^S$
True	False	True	$\equiv_{L,A}$
True	False	False	$\equiv_{L,A}^S$
False	True	True	$\equiv_M$
False	True	False	$\equiv_M^S$
False	False	True	$\equiv_S$
False	False	False	$\equiv_S^S$

Table 3: Combinations of Property Values and Corresponding Equivalence Types

The three properties can be used to determine whether a type of transformation rule is applicable. A type of transformation rule can be applied if the result produced by the right-hand side is equivalent to the result produced by the left-hand side according to the required equivalence type, as specified by the properties for the top operation.

The three properties are propagated from the root and down the tree (in the terminology of attributed syntax trees, these are *inherited attributes* [Knu68]). For the root, the properties are set in accordance with the specific user-level query language and query statement. For example, some variant of SQL may require (1) the result to be sorted if the `ORDER BY` clause is specified at the outer-most level, (2) the result always either to contain duplicates (`DISTINCT` is not specified) or not (`DISTINCT` is specified), and (3) the result always to contain the same time periods independently of which query plan is chosen. Consequently, for the root, the $OrderRequired$ property is set to True only if the

`ORDER BY` clause is specified at the outer-most level, and the $DuplicatesRelevant$ and $PeriodPreserving$ properties are always set to True.

The definitions of the three properties use two auxiliary Boolean properties $MayHaveDups$ and $MayHaveDupsInSn$, which are propagated from the leaf operations to the root (and thus are termed *derived* or *synthesized attributes* [Knu68]). These properties indicate whether a relation may contain duplicates and duplicates in snapshots, respectively. Moreover, the $DuplicatesRelevant$ property is used in the definition of the $PeriodPreserving$ property, and the latter property is used in the definition of the $OrderRequired$ property.

During query optimization, the properties are first set for the initial query evaluation plan that is passed to the query optimizer. First, properties $MayHaveDups$ and $MayHaveDupInSn$ are propagated bottom-up. Then, properties $DuplicatesRelevant$, $PeriodPreserving$, and $OrderRequired$ are propagated top-down in the given sequence.

We define all properties in turn. Table 4 defines the $MayHaveDups$ property for a non-leaf operation *op* according to the property values of its argument(s). The property holds for *op* if the result relation may contain duplicates. Argument operations are indicated as $child1_{op}$ and, in case *op* is a binary operation, $child2_{op}$. This property can be propagated from the bottom of the tree to the root according to how operations preserve duplicates (recall Table 1); the property is always True for leaf operations which correspond to base relations.

op	$MayHaveDups(op)$	
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}$, $\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}$, $rdup$, $rdup^T$	False	
$\sqcup$	True	
$\pi_{f_1,\ldots,f_n}$	False True	**if** $keyattr(result(child1_{op})) \in attr(f_1,\ldots,f_n)$ **otherwise**
σ_P, $sort_A$, $coal^T$, $\backslash$, $\backslash^T$	$MayHaveDups(child1_{op})$	
$\times$, $\times^T$, $\cup$, $\cup^T$	$MayHaveDups(child1_{op})$ $\vee$ $MayHaveDups(child2_{op})$	

Table 4: The $MayHaveDups$ Property Values of an Operation According to its Child(ren)

Operations ξ, ξ^T, $rdup$, and $rdup^T$ remove duplicates, while operations π and $\sqcup$ may manufacture duplicates. Projection π does not introduce duplicates if the key of its argument (which exists if the argument may not have duplicates) is included in the projection list. For other operations, the property is set according to the property of their arguments.

Table 5 defines the $MayHaveDupsInSn$ property, which holds for a non-leaf operation in a query plan if snapshots of the result relation may contain duplicates. The property is always True for leaf operations if they correspond to temporal relations.

The operations that have temporal counterparts, i.e., $\cup$, $\times$, $\backslash$, ξ, and $rdup$, produce nontemporal relations and cannot have duplicates in snapshots. The same applies to projections that remove temporal attributes. The other cases are similar to those of the $MayHaveDups$ property definition.

Table 6 defines the $DuplicatesRelevant$ property values for a non-root operation *op*. This property depends almost entirely on the parent of the operation, denoted op_p. In particular, the property is independent of the specific op. The parent of the operation is listed in the first column of the table. 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 regular (temporal) duplicate elimination, because then the child operation may deal with duplicates in any way, since they will later be removed; (2) when the parent operation is regular (temporal) difference, the operation in question is located at the right child, and the relation produced by the left child does not contain regular duplicates (duplicates in snapshots); and (3) when the parent operation is regular (temporal) aggregation and the duplicate-sensitive aggregation functions `AVG`, `SUM`, or `COUNT` are not used. For example, function `COUNT` is duplicate-sensitive because the result of an aggregation operation that counts the number of tuples in a relation would be affected by the presence of duplicates.

The next case to consider is when the property does not hold at the parent. Then, the property holds at a child when the parent operation is regular (or temporal) aggregation and the aggregation functions used are `AVG`, `SUM`, or `COUNT`.

op	$MayHaveDupsInSn(op)$	
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}$, $\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}$, $rdup$, $rdup^T$ $\times, \cup, \backslash$	False	
$\sqcup$	True	
$\pi_{f_1,\ldots,f_n}$	False	**if** $\langle \mathtt{T1}, \mathtt{T2} \rangle \notin attr(f_1,\ldots,f_n)$
	$MayHaveDupsInSn(child1_{op})$	**else, if** $keyattr(result(child1_{op})) \in attr(f_1,\ldots,f_n)$
	True	**otherwise**
σ_P, $sort_A$, $coal^T$ $\backslash^T$	$MayHaveDupsInSn(child1_{op})$	
$\times^T$, $\cup^T$	$MayHaveDupsInSn(child1_{op}) \vee MayHaveDupsInSn(child2_{op})$	

Table 5: The $MayHaveDupsInSn$ Property Values of an Operation According to its Child(ren)

op_p	$DuplicatesRelevant(op)$	
σ_P	$DuplicatesRelevant(op_p)$	
$\pi_{f_1,\ldots,f_n}$	$DuplicatesRelevant(op_p)$	
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	False	**if** $AggrFunctions(F_1,\ldots,F_m) \subset \{\mathtt{MIN},\mathtt{MAX}\}$
	True	**otherwise**
$\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	False	**if** $AggrFunctions(F_1,\ldots,F_m) \subset \{\mathtt{MIN},\mathtt{MAX}\}$
	True	**otherwise**
$rdup$	False	
$rdup^T$	False	
$coal^T$	True	
$sort_A$	$DuplicatesRelevant(op)$	
$\times$, *left*	$DuplicatesRelevant(op)$	
$\times$, *right*	$DuplicatesRelevant(op)$	
$\times^T$, *left*	$DuplicatesRelevant(op)$	
$\times^T$, *right*	$DuplicatesRelevant(op)$	
$\backslash$, *left*	True	
$\backslash$, *right*	True	**if** $MayHaveDups(op_{left})$
	False	**otherwise**
$\backslash^T$, *left*	True	
$\backslash^T$, *right*	True	**if** $MayHaveDupsInSn(op_{left})$
	False	**otherwise**
$\sqcup$, *left*	$DuplicatesRelevant(op_p)$	
$\sqcup$, *right*	$DuplicatesRelevant(op_p)$	
$\cup$, *left*	$DuplicatesRelevant(op_p)$	
$\cup$, *right*	$DuplicatesRelevant(op_p)$	
$\cup^T$, *left*	True	**if** $MayHaveDupsInSn(op_{right})$
	False	**otherwise**
$\cup^T$, *right*	True	

Table 6: The $DuplicatesRelevant$ Property Values of an Operation According to its Parent

In addition, the property holds at a child when the parent operation is regular (or temporal) 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 regular duplicates (duplicates in snapshots). Similar conditions apply to regular (temporal) union. The property always holds if the parent operation is coalescing because different numbers of duplicates in the argument might lead to result relations that are not even equivalent as sets.

Table 7 defines the $PeriodPreserving$ property. If this property holds at a parent node, it also holds at a child, except in the following cases: (1) when the parent operation is a projection not involving the time attributes and whose $DuplicatesRelevant$ property does not hold; (2) when the parent operation is regular aggregation, where the time attributes are not among the grouping attributes and the aggregation functions used are not among `AVG`, `SUM`, or `COUNT`; (3) when the parent operation is temporal aggregation; (4) when the parent operation is coalescing and the argument does not have duplicates in snapshots; and (5) when the parent operation is temporal difference and the right argument is the child in question.

op_p	$PeriodPreserving(op)$	
σ_P	$PeriodPreserving(op_p)$	**if** `T1` and `T2` $\notin attr(P)$
	True	**otherwise**
$\pi_{f_1,\ldots,f_n}$	$PeriodPreserving(op_p)$	**if** `T1` and `T2` $\in attr(f_1,\ldots,f_n)$
	True	**else, if** $DuplicatesRelevant(op_p)$ $\vee$ `T1` xor `T2` $\in attr(f_1,\ldots,f_n)$
	False	**otherwise**
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	False	**if** $AggrFunctions(F_1,\ldots,F_m) \subset \{$`MIN`, `MAX`$\}$ $\wedge$ `T1`, `T2` $\notin attr(G_1,\ldots,G_n)$
	True	**otherwise**
$\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	False	
$rdup$	True	
$rdup^T$	$PeriodPreserving(op_p)$	
$coal^T$	False	**if** $\neg MayHaveDupsInSn(op)$
	$PeriodPreserving(op_p)$	**otherwise**
$sort_A$	$PeriodPreserving(op_p)$	
$\times$, *left*	True	
$\times$, *right*	True	
$\times^T$, *left*	$PeriodPreserving(op_p)$	**if** followed by projection removing original timestamps
$\times^T$, *right*	True	**otherwise**
$\backslash$, *left*	True	
$\backslash$, *right*	True	
$\backslash^T$, *left*	$PeriodPreserving(op_p)$	
$\backslash^T$, *right*	False	
$\sqcup$, *left*	$PeriodPreserving(op_p)$	
$\sqcup$, *right*	$PeriodPreserving(op_p)$	
$\cup$, *left*	True	
$\cup$, *right*	True	
$\cup^T$, *left*	$PeriodPreserving(op_p)$	
$\cup^T$, *right*	$PeriodPreserving(op_p)$	

Table 7: The $PeriodPreserving$ Property Values of an Operation According to its Parent

If the property does not hold at the parent operation, the property also does not hold at a child, except in eight cases, namely for the following parent operations: (1) selection with a predicate involving a temporal attribute; (2) projection, if it involves one time attribute or if its $DuplicatesRelevant$ property holds; (3) regular aggregation, where the time attributes are among the grouping attributes or the aggregation functions are among `AVG`, `SUM`, or `COUNT`; (4) regular duplicate elimination; (5) regular Cartesian product; (6) temporal Cartesian product if it is not followed by a projection removing the original time attributes; (7) regular difference; and (8) regular union.

Table 8 defines the *OrderRequired* property. This property also depends almost entirely on the parent of the operation, listed in the first column of the table, and is independent of the specific op. Most often, the $\mathit{OrderRequired}$ 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.

op_p	$\mathit{OrderRequired}(op)$	
σ_P	$\mathit{OrderRequired}(op_p)$	
$\pi_{f_1,\ldots,f_n}$	$\mathit{OrderRequired}(op_p)$	
$\xi_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	$\mathit{OrderRequired}(op_p)$	
$\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}$	$\mathit{OrderRequired}(op_p)$	
rdup	$\mathit{OrderRequired}(op_p)$	
rdup^T	True	**if** $\mathit{MayHaveDupsInSn}(op) \wedge \mathit{PeriodPreserving}(op_p)$
	$\mathit{OrderRequired}(op_p)$	**otherwise**
coal^T	True	**if** $\mathit{MayHaveDupsInSn}(op) \wedge \mathit{PeriodPreserving}(op_p)$
	$\mathit{OrderRequired}(op_p)$	**otherwise**
$sort_A$	False	
$\times$, *left*	$\mathit{OrderRequired}(op_p)$	
$\times$, *right*	True	**if** $\mathit{SequenceRequired}(op)$
	False	**otherwise**
$\times^T$, *left*	$\mathit{OrderRequired}(op_p)$	
$\times^T$, *right*	True	**if** $\mathit{SequenceRequired}(op)$
	False	**otherwise**
$\backslash$, *left*	$\mathit{OrderRequired}(op_p)$	
$\backslash$, *right*	True	**if** $\mathit{SequenceRequired}(op)$
$\backslash^T$, *right*	False	**otherwise**
$\backslash^T$, *left*	True	**if** $\mathit{MayHaveDupsInSn}(op) \wedge \mathit{PeriodPreserving}(op_p)$
	$\mathit{OrderRequired}(op_p)$	**otherwise**
$\sqcup$, *left*	True	**if** $\mathit{SequenceRequired}(op)$
$\sqcup$, *right*	False	**otherwise**
$\cup$, *left*	True	**if** $\mathit{SequenceRequired}(op)$
$\cup$, *right*	False	**otherwise**
$\cup^T$, *left*	True	**if** $\mathit{SequenceRequired}(op)$
	False	**otherwise**
$\cup^T$, *right*	True	**if** $((\mathit{MayHaveDupsInSn}(op) \wedge \mathit{PeriodPreserving}(op_p))$
		$\vee\ \mathit{SequenceRequired}(op))$
	False	**otherwise**

Table 8: The $\mathit{OrderRequired}$ Property Values of an Operation According to its Parent

For the $\mathit{OrderRequired}$ property to hold at an immediate child of rdup^T, either that property must hold for rdup^T, or the child can produce duplicates in its snapshots and the rdup^T is required to preserve the periods of its argument. This entry shows how a requirement being computed top-down relies on properties that are propagated bottom-up.

The operations rdup^T, coal^T, $\backslash^T$, and $\cup^T$ are sequence sensitive when their arguments have duplicates in snapshots (the left argument for $\backslash^T$ and the right argument for $\cup^T$ count), i.e., if they take arguments that are equivalent as multisets, their results may not be equivalent as multisets (however, their snapshots will be equivalent as multisets). Therefore, when one of these four operations occurs in the parent node, it requires that the sequence of tuples in its argument(s) is not changed when periods have to be preserved by the operation and the argument may have duplicates in snapshots.

Note that this is a stronger requirement than that for the $\mathit{OrderRequired}$ property: we cannot change the sequence of tuples even if the change would still preserve some order on the result. This requirement is captured by the auxiliary property $SequenceRequired$, which is True for operation *op* if we cannot change the sequence of tuples in the result of that operation. Table 9 defines the $SequenceRequired$ property (the property is always False for the root).

The $SequenceRequired$ property needs to be checked when setting the $OrderRequired$ property for a number of operations. For example, if the $SequenceRequired$ property is True for a Cartesian product, the orders of *both* arguments of the product matter (the $OrderRequired$ property has to be set to True for both arguments). However, if the $OrderRequired$ holds and the $SequenceRequired$ property does not hold, the $OrderRequired$ property has to be set to True only for the left argument because the right argument cannot contribute to any sensible sorting of the result.

op_p	$SequenceRequired(op)$	
$coal^T$, $rdup^T$ $\setminus^T$, *left* $\cup^T$, *right*	True $SequenceRequired(op_p)$	**if** $MayHaveDupsInSn(op) \wedge PeriodPreserving(op_p)$ **otherwise**
$\setminus^T$, *right*	True $SequenceRequired(op_p)$	**if** $MayHaveDupsInSn(leftsibling)$ $\wedge\ PeriodPreserving(op_p)$ **otherwise**
$\cup^T$, *left*	True $SequenceRequired(op_p)$	**if** $MayHaveDupsInSn(rightsibling)$ $\wedge PeriodPreserving(op_p)$ **otherwise**
$sort_A$	False $SequenceRequired(op_p)$	**if** A includes all attributes of the argument **otherwise**
other operations	$SequenceRequired(op_p)$	

Table 9: The $SequenceRequired$ Property Values of an Operation According to its Child(ren)

With the property propagation as outlined, it might be that the required equivalence type for a leaf-level relation in the query tree is $\equiv_{L,A}$. This may happen if coalescing, temporal duplicate elimination, temporal difference, or temporal union are used and their arguments may have duplicates in their snapshots (as above, the left argument of $\setminus^T$ and the right argument of $\cup^T$ count). In the stratum architecture, this equivalence cannot be satisfied if the underlying relations come from the DBMS in unknown order. Such is the case if the underlying DBMS supports SQL, and the expression below the T^S operation does not include a $sort$ operation, sorting on all attributes. Then the results of the mentioned operations present in the stratum would possibly contain different tuples (even though their snapshots at each point of time would contain the same tuples). For example, the query $coal^T(T^S(r))$, if run several times, may return results that are only snapshot-multiset equivalent because relation r is retrieved from a conventional DBMS. Such queries can be answered only if the top equivalence is $\equiv^S_M$ or $\equiv^S_{L,A}$. The mapping stage should determine if the required top equivalence can be satisfied for the given query, and if not, it should reject the query. An alternative for the stratum implementor would be to modify the mapping stage so that it introduces a $sort$ operation (sorting on all attributes) before the sequence-sensitive operation used in the query, ensuring that the initial query plan satisfies the required equivalence.

Coalescing combined with temporal duplicate elimination, and temporal difference combined with temporal duplicate elimination (if the result is later coalesced) are insensitive to the order of their arguments, and such queries would always return $\equiv_M$ or $\equiv_{L,A}$ equivalent results in the stratum architecture. The query used in Section 3.5 is one such example.

6.3 Adjustment of Properties

When a transformation rule is applied during query optimization, the properties must be adjusted. Since transformation rules may be applied frequently, it is preferable to avoid scanning the whole operation tree both bottom-up and top-down each time a rule is applied, but rather to do incremental, local adjustments. The tables and definitions of the previous section indicate how to accomplish this, by expressing property values in terms of the property values immediately above (or below) them in the operation tree. For example, to adjust the values of the $DuplicatesRelevant$ property for some operation after a transformation, it is enough to know the property value for the operation immediately above the resulting query part.

If a property's value depends on the values in the tree above it (such as *DuplicatesRelevant*), we determine if the application of a transformation rule changes the property values at the bottom node(s). If so, adjustments in the subtree(s) below are necessary.

Similarly, if the value of a property depends on the values below it in the tree, we must determine if the application of a transformation rule changes the property value of the top operation. If it does, we must reconsider the properties of the operations in the part of the tree above the resulting query part.

The adjustment of one property may trigger the adjustment of other properties. For example, the adjustment of the *MayHaveDupInSn* property triggers the adjustment of the *PeriodPreserving* property because the value of the latter for coalescing depends on the *MayHaveDupInSn* property value. Table 10 summarizes the triggered adjustments.

Adjustment	**Triggered top-down adjustment from the top-most adjusted node**
MayHaveDupInSn	*OrderRequired* in all subtrees below the first-met coalescing. *DuplicatesRelevant* in all subtrees below the first-met temporal difference or temporal union. *PeriodPreserving* for the first coalescing and below.
MayHaveDupIn	*DuplicatesRelevant* in all subtrees below the first-met regular difference.
PeriodPreserving	*OrderRequired* in all subtrees below the first-met temporal duplicate elimination, coalescing, temporal difference, or temporal union.
DuplicatesRelevant	*PeriodPreserving* in all subtrees below the first-met projection.
SequenceRequired	*OrderRequired* in all subtrees below.

Table 10: Triggered Property Adjustment

In general, non-local property adjustments will be rare because applications of most of the transformation rules will not lead to the change of the properties of the top (bottom) operation(s). Table 11 describes the adjustments for all transformation rules that require non-local adjustments (excluding triggered adjustments). All these rules either introduce or remove an operation. (Each equivalence in the table represents two transformation rules.)

Equivalence	**Adjustment actions (according to the transformed expression)**
G3, G24, D14	Adjust the *PeriodPreserving* property in all subtrees below.
G5, G25, C1, C4	Adjust the *OrderRequired*, *DuplicatesRelevant*, and *PeriodPreserving* properties in all subtrees below.
D2	Adjust the *DuplicatesRelevant* property in all subtrees below.
D3, D4	Adjust the *MayHaveDups* and *MayHaveDupsInSn* properties above, up to the root. Adjust the *OrderRequired* property in all subtrees below the top-most temporal duplicate elimination, coalescing, temporal difference, or temporal union. Adjust the *DuplicatesRelevant* property in all subtrees below the top-most regular difference, temporal difference, or temporal union. Adjust the *PeriodPreserving* property in all subtrees below the top-most coalescing.
C2, C8, C11, D1	Adjust the *DuplicatesRelevant* and *PeriodPreserving* properties in all subtrees below.
S1	Adjust the *SequenceRequired* and *OrderRequired* properties in all subtrees below.

Table 11: Adjustment of the Properties

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

7 Query Plan Enumeration

We give a straightforward enumeration algorithm whose purpose is to generate correct query evaluation plans; we do not consider the subsequent heuristic or cost-based selection of a final query plan. We also do not consider the performance of the enumeration algorithm, except to note that incremental maintenance of property values improves over full recomputation.

The arguments to the query plan enumeration algorithm are a set of plans $\mathcal{P}$ (initially, $\mathcal{P}$ contains only one 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 18.

For the algorithm to terminate, the set of transformation rules cannot include all rules given in Section 5. The rules that introduce additional operations, such as $r \rightarrow_S rdup(r)$, would be applicable an infinite number of times. Hence, heuristics have to be used to restrict the rule set. For example, one possible heuristic is to not use rules that introduce additional operations, such as $r \rightarrow_S rdup(r)$. Another heuristic performs selections 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 $coal^T(\sigma_P(r)) \rightarrow_L \sigma_P(coal^T(r))$. We assume that a heuristic is in place that ensures termination.

```
**for each** plan $P \in \mathcal{P}$ **do**
  **for each** $T \in \mathcal{TR}$ **do**
    **for each** location $l$ within $P$ that matches 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 $\wedge\ op_{top} \in l\ (\neg OrderRequired(op)))$
              $\vee\ (T$ is a $\equiv_S$ rule $\wedge\ op_{top} \in l\ (\neg DuplicatesRelevant(op) \wedge \neg OrderRequired(op)))$
              $\vee\ (T$ is a $\equiv_L^S$ rule $\wedge\ op_{top} \in l\ (\neg PeriodPreserving(op)))$
              $\vee\ (T$ is a $\equiv_M^S$ rule $\wedge\ op_{top} \in l\ (\neg OrderRequired(op) \wedge \neg PeriodPreserving(op)))$
              $\vee\ (T$ is a $\equiv_S^S$ rule $\wedge\ op_{top} \in l\ (\neg DuplicatesRelevant(op) \wedge \neg OrderRequired(op)$
                                                   $\wedge\ \neg PeriodPreserving(op)))$
      **then** apply $T$ to $l$ yielding $P'$;
            adjust properties of $P'$;
            add $P'$ to $\mathcal{P}$
**return** $\mathcal{P}$
```

Figure 18: Query Plan Enumeration Algorithm

The algorithm provides an operational means of determining when a transformation rule is applicable. It has a syntactic component (the left-side expression must match in some location) and a semantic component (the preconditions must hold and the properties must have appropriate settings). In the algorithm, when testing the applicability of a transformation rule at some location, the properties of the operation at the top of that location is employed. 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 $coal^T$ operation are used.

The algorithm is deterministic, i.e., it generates the same set of query plans independently of the order of transformation rules and locations. This can be seen easily by noting that the algorithm applies all the transformations to each candidate plan at each possible location in all orders. In many cases, the plan P' generated by applying a transformation will already be present in P.

The presence of the stratum imposes additional correctness requirements, specifically that (a) portions evaluated by the underlying DBMS utilize only operations provided by that DBMS, (b) the required equivalence of the T^S operation is satisfied by the DBMS, and (c) portions evaluated by the stratum utilize only operations provided by the stratum. All three requirements must be ensured by the mapping to the initial algebraic expression, which needs to be cognizant of the capabilities of the DBMS and the stratum. Requirements (a) and (c) are ensured in the initial query plan by the presence or absence of transformations that move the transfer operations across operations (see Section 5.5); requirement (b) is satisfied via the appropriate use of properties.

Theorem 7.1 The algorithm given in Figure 18 generates correct query plans.

Proof: To prove the theorem, we need to prove that the algorithm applies a transformation rule of some type only when the result produced by the new query plan is equivalent to the result produced by the original plan according to the top equivalence, which depends on the query language and the actual query statement. The proof is divided

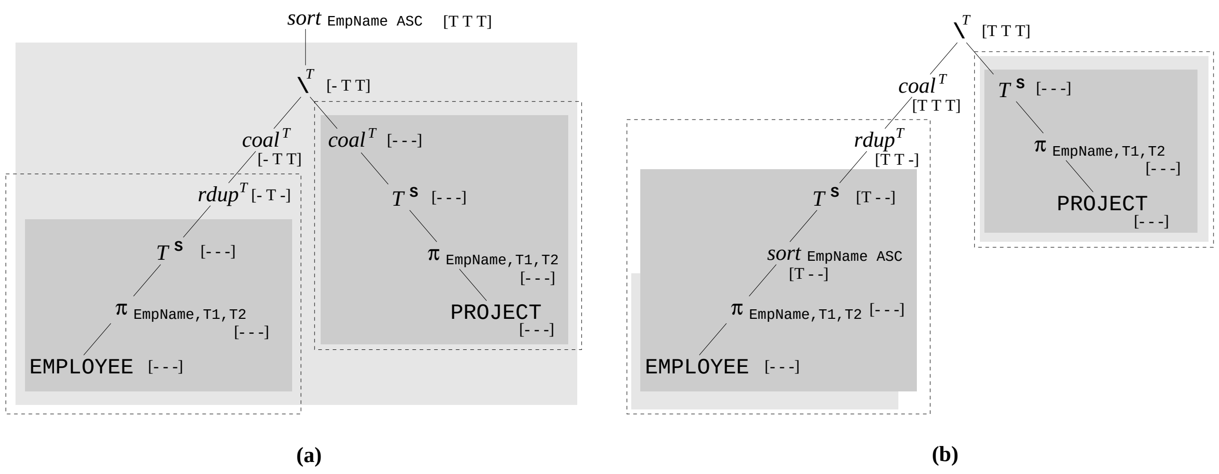


Figure 19: Operation Trees with Properties and Transformation-Rule Applicability Regions

into six parts, one for each type of transformation rule. Appendix B provides a proof that $\equiv_M$ type rules are applied correctly if the top equivalence is $\equiv_{L,A}$ or $\equiv_M$. □

While the algorithm generates correct plans, it does not generate all possible plans—although we exploit transformation rules of "weak" equivalence types, e.g., $\equiv_S$, *all* the possible, correct query plans that may be generated using the different types of transformation rules are not found.

To illustrate how the enumeration algorithm works, we use the example query from Section 3. The initial query plan is given in Figure 10(a). 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. Also, we push the transfer operation down by using transfer rules T1, T2, T5, and T8; the rules of type $\equiv_M$ can be applied below the $sort$ operation.

Then we push the coalescing below the temporal difference by using rule C10 (we can apply this rule because $OrderRequired$ does not hold for the coalescing). The resulting plan is shown in Figure 19(a). For each operation, we list its properties in square brackets in the order $OrderRequired$, $DuplicatesRelevant$, $PeriodPreserving$.

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 by using rules S9, S12, and S14; and we change the location of the $sort$ operation from the stratum to the DBMS by using rules T6 and T8. Figure 19(b) shows the final plan.

8 Extensibility of the Framework

The optimizer implementor can extend the foundation presented here by tailoring it to a specific query language or by adding a new operation.

The former requires the implementor to define the mapping from the query language to the algebra and to determine how the top equivalences should be set for the initial query plans.

When adding a new operation, it must be defined in λ-calculus, related transformation rules must be introduced, and property values for the operation must be determined. In addition, the implementor should consider if the new transformation rules may require non-local property adjustments and should ensure that queries involving the new operation are processed only if they can satisfy the top equivalence when applied repeatedly. The proof of correctness of the enumeration algorithm must be extended to accommodate the new operation. For the stratum architecture, a translation of the new operation to SQL should be developed.

9 Related Research

In this section, we survey how the previous work on relational and temporal algebras addressed duplicates and order. Past work in in conventional and temporal query optimization, as well as in temporal layers, is also covered.

Dayal et al. [DGK82] extend the relational model to include multiset (also called bag) relations. They define selection, join, projection, duplicate elimination, union, intersection, and difference operations for multisets, and provide several algebraic equivalences. In a similar manner, Albert [Alb91] extends union, intersection, difference, and Boolean selection to multisets, giving them semantics that agree with the usual set-theoretic semantics when the arguments are sets. For example, the union defined in [Alb91], unlike concatenation, corresponds to disjunction for Boolean selection. In our algebra, we have both union and concatenation; their difference in relation to disjunction for Boolean selection is exemplified by transformation rules G2 and G3. The recent book by Garcia-Molina et al. [GM00] 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. We define relations as lists, and our set of transformation rules extends their rules to lists, precisely specifying the equivalence type that holds for each rule, and also adds rules for temporal operations.

Leung et al. [Leu98] present query rewrite rules for decorrelating complex queries, as implemented in IBM's DB2. Queries are represented in a query graph model, which is a graph of nodes, each representing a table operation whose inputs and outputs are tables. Duplicates are addressed in a query graph model and in query rewrite rules; in this graph model, each operation can eliminate, preserve, or permit duplicates. Duplicates should be preserved when, for example, the `DISTINCT` clause is not specified, and duplicates are permitted when the operation produces an argument for a universal quantifier, e.g., `ALL`. Consequently, duplicates are addressed as special cases in query rewrite rules. Our algebra and transformation rules incorporate the handling of duplicates and order. We consider operations that eliminate or preserve duplicates. The $\equiv_S$ equivalence type corresponds to "permitting" duplicates, e.g., it allows replacing a query expression with a set-equivalent one.

Mumick et al. [MPR90, Mum90] study the extension of the Magic-Sets technique for programs containing multisets and aggregates. They note that the implementation of multisets is efficient, since duplicate checks are not needed. They provide a formal basis for reasoning about optimization techniques when multisets are generated as intermediate relations, independently of whether the user desires multiset semantics. Our framework integrates the treatment of relations as lists, multisets, and sets.

Grumbach and Milo [GM93] study the expressive power of algebras for manipulating bags. In particular, they study how bag nesting affects expressive power. Libkin and Wong [LW94] provide new techniques for studying the expressive powers of set languages and bag languages that have aggregate functions. We do not focus on studying the expressive power of our proposed algebra other than showing that it extends the conventional relational algebra.

More than a dozen temporal relational algebras have been proposed over the last two decades [MS91, OS95], 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 [GS90, LM93] primarily considers the processing of joins and semijoins. For example, Gunadhi and Segev [GS90] define several temporal joins and discuss their optimization, focusing on temporal selectivity estimation and strategies for optimizing temporal equijoins. That work does not delve into general query optimization and does not address duplicates, order, and coalescing.

Böhlen et al. [BSS96] define coalescing and argue that this operation is not implemented efficiently in conventional DBMSs. The paper uses set-based semantics, and coalescing is defined as merging of value-equivalent tuples.

The recent work of Gadia and Nair [GN98] considers query optimization for a parametric model for temporal databases, presents algebraic identities, and gives a heuristic optimization algorithm. They define a relation as a set of tuples, but they also consider *weakly* equivalent relations, i.e., relations that have the same snapshots. We refine this equivalence into our snapshot-based set equivalences.

Several papers discussing stratum architectures for a temporal DBMS have appeared, e.g., [TJS98], and several prototype temporal DBMSs have been implemented, e.g., [Böh95, Böh98]. Most of the proposed temporal strata translate temporal query language statements to SQL, but do not perform any systematic optimization or processing. Meanwhile, we provide a framework for the division of processing between the stratum and the underlying DBMS.

10 Conclusions and Research Directions

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 on temporal query optimization primarily considered the efficient processing of only some operations, e.g., 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, list-based algebra, which enhances existing relational algebras. The algebra is independent of the specific user-level variant of the relational data model and is also independent of the user-level relational query language.

Six types of equivalences among algebraic query expressions are identified, leading to six types of transformation rules that can be exploited during query optimization. These sets of rules go beyond all such existing sets known to the authors. Depending on whether order, duplicate removal, and 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 the type of a transformation rule 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 four tasks: the database implementor has to (1) specify operations formally in λ-calculus; (2) design appropriate transformation rules, determine for each which of the six equivalences apply, and prove that the transformation rules are correct; (3) augment the setting and adjusting of the properties so that the enumeration algorithm applies the transformation rules correctly; and (4) ensure that the mapping generates a correct initial query plan.

To complete the framework for query optimization and evaluation (recall Figure 1), a number of steps remain. A mapping step, not covered in this paper, converts the query into an initial plan. 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 the language the DBMS supports, and that the operations assigned to the stratum have corresponding implementation algorithms.

The algorithm given in Section 7 generates from this plan a number of query plans according to the heuristics provided. The next step is to select the plan with the expected lowest cost. In the stratum architecture, the challenge is to come up with a unified cost model for stratum and DBMS operations, and with cost functions. Cost functions for operations performed in the DBMS are in general not known, but the statistics are possible to obtain. The issues regarding costing are interesting research challenges. Another challenge is to develop strategies for dividing the processing between the stratum and the DBMS, integrating transformation rules with heuristics and cost estimation techniques. In addition, multiple implementations of operations, e.g., several join implementations that return differently ordered relations, should be considered.

Once a query plan is chosen, the query parts to be performed in the DBMS should be translated into SQL. Results should be returned to the stratum for possible further processing. If the result of the stratum is needed for subsequent operations in the DBMS, a temporary table should be created. The translation from the algebra to SQL is also left for future research. Finally, the operations located in the stratum should be evaluated in an efficient manner. There has been significant work by others on this problem, cf. [ZCF97].

This paper has provided a mechanism for representing queries and for query transformation, which is at the core of query optimization. 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 [Cli97], and transaction time [JD98]. It might be appropriate to use an automatic theorem prover to ensure the correctness of the transformations, the property definitions, and the plan enumeration algorithm for all cases.

References

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

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11): 832–843 (1983).

[Böh98] M. H. Böhlen. The Tiger Temporal Database System. <`www.cs.auc.dk/˜tigeradm/`> current as of 25 Feb 2000.

[BBJ98] M. H. Böhlen, R. Busatto, and C. S. Jensen. Point versus Interval-Based Temporal Data Models. In *Proceedings of the IEEE ICDE,* Orlando, Florida, pp. 192–200 (1998).

[Bet98] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang. A Glossary of Time Granularity Concepts. In [EJS98], pp. 406–413 (1998).

[BJ97] M. H. Böhlen and C. S. Jensen. Temporal Statement Modifiers. Unpublished manuscript, submitted for publication, December 1997.

[Böh95] M. H. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4): 53–60 (1995).

[BSS96] M. H. Böhlen, R T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *Proceedings of VLDB,* Bombay, India, pp. 180–191 (1996).

[Cli97] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of “Now” in Databases. *ACM TODS*, 22(2): 171–214 (1997).

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

[EJS98] O. Etzion, S. Jajodia, and S. Sripada (eds.) *Temporal Databases: Research and Practice*. LNCS 1399, Springer-Verlag (1998).

[GM00] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall (2000).

[GN98] S. K. Gadia and S. S. Nair. Algebraic Identities and Query Optimization in a Parametric Model for Relational Temporal Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(5): 793–807 (1998).

[Gor87] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag (1987).

[GM93] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proceedings of PODS,* Washington, DC, pp. 49–58 (1993).

[GS90] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Proceedings of SSDBM,* Charlotte, NC, pp. 131–147 (1990).

[Inm96] W. H. Inmon. *Building the Data Warehouse*. Second Edition. John Wiley and Sons (1996).

[JD98] C. S. Jensen and C. E. Dyreson, editors. A Consensus Glossary of Temporal Database Concepts. In [EJS98], pp. 367–405 (1998).

[JS99] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1): 36–45 (1999).

[JSS94] 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).

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

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

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. In *Mathematical Systems Theory,* Volume 2, Springer Verlag, pp. 127–145, June, 1968.

[KS95] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of IEEE ICDE,* Taipei, Taiwan, pp. 222–231 (1995).

[LW94] L. Libkin and L. Wong. New Techniques for Studying Set Languages, Bag Languages and Aggregate Functions. In *Proceedings of ACM PODS,* Minneapolis, Minnesota, pp. 155–166 (1994).

[Leu98] T. Y. C. Leung, H. Pirahesh, P. Seshadri, and J. M. Hellerstein. 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).

[LM93] 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).

[MS91] 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).

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of VLDB*, Brisbane, Queensland, Australia, pp. 264-277 (1990).

[Mum90] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is Relevant. In *Proceedings of ACM SIGMOD*, Atlantic City, NJ, pp. 247-258 (1990).

[OS95] G. Özsoyoğlu and R. T. Snodgrass Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4): 513–532 (1995).

[Sno87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2): 247–298 (1987).

[Sno95] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers (1995).

[Sno99] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann (1999).

[TJS98] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In *Proceedings of IDEAS*, Cardiff, Wales, UK, pp. 4–13 (1998).

[YYW00] J. Yang, H. C. Ying, and J. Widom. TIP: A Temporal Extension to Informix. In *Proceedings of EDBT*, Konstanz, Germany (2000) (to appear).

[ZCF97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997

A Auxiliary Operations

First, we define auxiliary operations on tuples. Then, we define auxiliary operations on relations, including fundamental operations such as $head$ and $tail$. For operations $head$, $tail$, and @, the schema of the argument relation is the same as the schema of the result relation.

A.1 Auxiliary Operations on Tuples

Concatenation Function $\circ : [\mathcal{T} \times \mathcal{T}] \rightarrow \mathcal{T}$ concatenates two tuples. Let two tuples be t_1 and t_2 and their corresponding schemas be $S_1 = (\Omega_1, \Delta_1, dom_1)$ and $S_2 = (\Omega_2, \Delta_2, dom_2)$. We define the result tuple t_r and its schema $S_r = (\Omega_r, \Delta_r, dom_r)$ as follows. An attribute name of the schema of the result tuple is prefixed by 1 and 2 only if the attribute appears in the schemas of both argument tuples.

$$
\begin{aligned}
t_r &\triangleq \{(attr, value) \mid ((attr, value) \in t_1 \wedge attr \notin \Omega_2) \vee ((attr, value) \in t_2 \wedge attr \notin \Omega_1)\} \cup \\
&\quad \{(1.attr, value) \mid (attr, value) \in t_1 \wedge attr \in \Omega_2\} \cup \\
&\quad \{(2.attr, value) \mid (attr, value) \in t_2 \wedge attr \in \Omega_1\} \\
\Omega_r &\triangleq \{a \mid (a \in \Omega_1 \wedge a \notin \Omega_2) \vee (a \in \Omega_2 \wedge a \notin \Omega_1)\} \cup \\
&\quad \{1.a \mid a \in \Omega_1 \wedge a \in \Omega_2)\} \cup \{2.a \mid a \in \Omega_2 \wedge a \in \Omega_1)\} \\
\Delta_r &\triangleq \Delta_1 \cup \Delta_2 \\
dom_r &\triangleq \{(attr, type) \mid ((attr, type) \in dom_1 \wedge attr \notin \Omega_2) \vee \\
&\qquad\qquad ((attr, type) \in dom_2 \wedge attr \notin \Omega_1)\} \cup \\
&\quad \{(1.attr, type) \mid (attr, type) \in dom_1 \wedge attr \in \Omega_2\} \cup \\
&\quad \{(2.attr, type) \mid (attr, type) \in dom_2 \wedge attr \in \Omega_1\}
\end{aligned}
$$

For example, the concatenation of tuples $t_1 = \{(\texttt{Name}, \texttt{Bill}), (\texttt{Salary}, 20), (\texttt{T1}, 10), (\texttt{T2}, 20)\}$ and $t_2 = \{(\texttt{Name}, \texttt{Bill}), (\texttt{Department}, \texttt{Sales})\}$ leads to tuple $t_r = \{(1.\texttt{Name}, \texttt{Bill}), (\texttt{Salary}, 20), (\texttt{T1}, 10), (\texttt{T2}, 20), (2.\texttt{Name}, \texttt{Bill}), (\texttt{Department}, \texttt{Sales})\}$.

We chose not to prefix all attributes of the resulting relation, because we want to retain its type and, in our case, a relation type is solely defined by existence of attributes T1 and T2. In addition, if we would prefix all attributes, several concatenations of different tuples would lead to too many prefixes.

DoesOverlap Function $DoesOverlap^T : [\mathcal{T}^T \times \mathcal{T}^T] \rightarrow$ Boolean returns True if the time periods of the argument tuples overlap and False, otherwise.

$$DoesOverlap^T \triangleq \lambda t_1, t_2.(t_1.\texttt{T1} < t_2.\texttt{T2}) \wedge (t_1.\texttt{T2} > t_2.\texttt{T1})$$

DoesMeet Function $DoesMeet^T : [\mathcal{T}^T \times \mathcal{T}^T] \rightarrow$ Boolean returns True if the time periods of the argument tuples meet and False, otherwise.

$$DoesMeet^T \triangleq \lambda t_1, t_2.(t_1.\texttt{T2} = t_2.\texttt{T1}) \vee (t_1.\texttt{T1} = t_2.\texttt{T2})$$

GetIntersectingTuple Function $GetIntersectingTuple^T : [\mathcal{T}^T \times \mathcal{T}^T] \rightarrow \mathcal{T}^T$ intersects time periods of two argument tuples and, if the periods overlap, forms a new tuple containing intersecting time periods, otherwise returns NULL.

$$
\begin{aligned}
GetIntersectingTuple^T \triangleq \lambda t_1, t_2.DoesOverlap^{vt}(t_1, t_2) \rightarrow\ & max\ (t_1.\texttt{T1}, t_2.\texttt{T1}) \circ min\ (t_1.\texttt{T2}, t_2.\texttt{T2}), \\
& undef
\end{aligned}
$$

Auxiliary functions max and min take one-attribute tuples as arguments, compare the values of those tuples, and return a new one-attribute tuple.

The schema of a result tuple of $GetIntersectingTuple^T$ is $S_r = (\Omega_r, \Delta_r, dom_r)$, where Ω_r, Δ_r, and dom_r are defined as follows.

$$
\begin{aligned}
\Omega_r &\triangleq \{\texttt{T1}, \texttt{T2}\} \\
\Delta_r &\triangleq \{\mathbb{T}\} \\
dom_r &\triangleq \{(\texttt{T1}, \mathbb{T}), (\texttt{T2}, \mathbb{T})\}
\end{aligned}
$$

IsValueEquiv Function $IsValueEquiv : \mathcal{T} \times \mathcal{T} \rightarrow$ Boolean returns True if all non-temporal attributes of both argument tuples are equal. The argument tuples have the same schema, where non-temporal attribute-domain values are denoted as $a_1, \ldots, a_n$.

$$IsValueEquiv \triangleq \lambda t_1, t_2.(t_1.a_1 = t_2.a_1 \wedge \ldots \wedge t_1.a_n = t_2.a_n)$$

A.2 Auxiliary Operations on Relations

Head Function $head : \mathcal{R} \rightarrow \mathcal{T}$ returns the first tuple of the argument relation.

$$head \triangleq \lambda r.(r = \perp) \rightarrow undef,\ t_1$$

Tail Function $tail : \mathcal{R} \rightarrow \mathcal{R}$ returns the argument relation without its first tuple.

$$tail \triangleq \lambda r.(r = \perp) \rightarrow undef,\ \langle t_2, \ldots, t_n \rangle$$

According to the definition, $tail$ applied to a relation with one tuple returns an empty relation.

Append Function $@ : [\mathcal{T} \times \mathcal{R}] \rightarrow \mathcal{R}$ prepends the argument tuple to the argument relation.

$$\begin{aligned} @ \triangleq \lambda t, r.(r = \perp) &\rightarrow \langle t \rangle, \\ &\langle t, t_1, \ldots, t_n \rangle \end{aligned}$$

IsIn Function $isIn : [\mathcal{T} \times \mathcal{R}] \rightarrow$ Boolean returns True if the argument tuple exists in the argument relation and False otherwise.

$$\begin{aligned} isIn \triangleq \lambda t, r.(r = \perp) &\rightarrow \text{False}, \\ (t = head(r)) &\rightarrow \text{True}, \\ &isIn(t, tail(r)) \end{aligned}$$

Remove Function $remove : [\mathcal{T} \times \mathcal{R}] \rightarrow \mathcal{R}$ removes the first occurence of the argument tuple from the argument relation. The schema of the argument relation is retained for the result relation.

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

OverlappingTuple Function $OverTpl^T : [\mathcal{T}^T \times \mathcal{R}^T] \rightarrow \mathcal{T}^T$ scans the argument relation and finds the first tuple that overlaps with the argument tuple and is value-equivalent with it.

$$\begin{aligned} OverTpl^T \triangleq \lambda t, r.(r = \perp) &\rightarrow undef, \\ (IsValueEquiv(t, head(r)) \wedge DoesOverlap^T(t, head(r))) &\rightarrow head(r), \\ &OverTpl^T(t, tail(r)) \end{aligned}$$

MeetingTuple Function $MeetTpl^T : [\mathcal{T}^T \times \mathcal{R}^T] \rightarrow \mathcal{T}^T$ scans the argument relation and finds the first tuple that meets with the argument tuple and is value-equivalent with it. The argument tuple, argument relation, and result relation have the same schema.

$$\begin{aligned} MeetTpl^T \triangleq \lambda t, r.(r = \perp) &\rightarrow undef, \\ (IsValueEquiv(t, head(r)) \wedge DoesMeet(t, head(r))) &\rightarrow head(r), \\ &MeetTpl^T(t, tail(r)) \end{aligned}$$

minVal Function $minVal : \mathcal{R}^T \to \mathbb{T}$ scans the argument temporal relation and returns the minimum timestamp value. It uses auxiliary function $findMin : [\mathcal{R}^T \times \mathbb{T}] \to \mathbb{T}$, which scans the argument temporal relation and returns the smallest timestamp value among the argument timestamp value and all argument relation timestamp values. Both functions exploit the fact that, in a temporal relation, `T1` is always smaller than `T2`.

$$\begin{aligned} minVal \triangleq \lambda r.(r = \perp) &\to undef, \\ & findMin(tail(r), head(r).\mathtt{T1}) \end{aligned}$$

$$\begin{aligned} findMin \triangleq \lambda r, c.(r = \perp) &\to c, \\ (head(r).\mathtt{T1} < c) &\to findMin(tail(r), head(r).\mathtt{T1}), \\ & findMin(tail(r), c) \end{aligned}$$

maxVal Function $maxVal : \mathcal{R}^T \to \mathbb{T}$ scans the argument temporal relation and returns the maximum timestamp value. The function is analogous to the $minVal$ function, and it uses auxiliary function $findMax : [\mathcal{R}^T \times \mathbb{T}] \to \mathbb{T}$, which scans the argument temporal relation and returns the biggest timestamp value among the argument timestamp value and all argument relation timestamp values.

$$\begin{aligned} maxVal \triangleq \lambda r.(r = \perp) &\to undef, \\ & findMax(tail(r), head(r).\mathtt{T2}) \end{aligned}$$

$$\begin{aligned} findMax \triangleq \lambda r, c.(r = \perp) &\to c, \\ (head(r).\mathtt{T2} > c) &\to findMax(tail(r), head(r).\mathtt{T2}), \\ & findMax(tail(r), c) \end{aligned}$$

MinTime Function $MinTime : [\mathcal{R}^T \times \mathbb{T} \times \mathbb{T}] \to \mathbb{T}$ scans the argument temporal relation and returns the smallest timestamp value that is bigger than the first argument timestamp value, but smaller than or equal to the second argument timestamp value. It uses auxiliary function $ActualMinTime : [\mathcal{R}^T \times \mathbb{T} \times \mathbb{T}] \to \mathbb{T}$ which does the same, but returns the second argument timestamp value in case a suitable timestamp value is not found in the argument relation (while the $MinTime$ function is undefined in this case).

$$\begin{aligned} MinTime \triangleq \lambda r, c_1, c_2.(r = \perp) &\to undef, \\ (head(r).\mathtt{T1} > c_1 \wedge head(r).\mathtt{T1} \leq c_2) &\to \\ & ActualMinTime(tail(r), c_1, head(r).\mathtt{T1}), \\ (head(r).\mathtt{T2} > c_1 \wedge head(r).\mathtt{T2} \leq c_2) &\to \\ & ActualMinTime(tail(r), c_1, head(r).\mathtt{T2}), \\ & MinTime(tail(r), c_1, c_2) \end{aligned}$$

$$\begin{aligned} ActualMinTime \triangleq \lambda r, c_1, c_2.(r = \perp) &\to c_2, \\ (head(r).\mathtt{T1} > c_1 \wedge head(r).\mathtt{T1} < c_2) &\to \\ & ActualMinTime(tail(r), c_1, head(r).\mathtt{T1}), \\ (head(r).\mathtt{T2} > c_1 \wedge head(r).\mathtt{T2} < c_2) &\to \\ & ActualMinTime(tail(r), c_1, head(r).\mathtt{T2}), \\ & ActualMinTime(tail(r), c_1, c_2) \end{aligned}$$

B Proofs

Theorem B.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: First, we prove that $r_1 \equiv_L r_2 \Rightarrow r_1 \equiv_M r_2$. According to the ($\lambda$-calculus) definition of multiset equivalence given in Section 4, the equivalence $r_1 \equiv_M r_2$ does not hold if, at some step during the iteration, the condition of the second line is True (one of the argument relations is empty while the other one is not) or the condition of the third line is False (the first tuple of the first argument relation is not in the second argument relation). Given that $r_1 \equiv_L r_2$, the condition of the second line must always be False, because both r_1 and r_2 have the same number of tuples (otherwise r_1 and r_2 would have not been list equivalent, because the second-line condition of the list-equivalence definition would evaluate to True) and functions $tail$ and $remove$ used in the recursive call remove exactly one tuple each. Having that $(head(r_1) = head(r_2)) \Rightarrow isIn(head(r_1), r_2)$, the condition of the third line is always True because the third-line condition of the list-equivalence definition must always be True for the relations to be list equivalent, and, during all iteration steps, the list-equivalence and multiset-equivalence definitions operate on the same arguments; the latter is ensured because the call of the $remove$ function in the third line of multiset-equivalence definition is equivalent to the call of the $tail$ function in the third line of list-equivalence definition (since the first tuple of the first argument relation is always equal to the first tuple of the second argument relation).

Similarly, to prove that $r_1 \equiv_M r_2 \Rightarrow r_1 \equiv_S r_2$, we need to show that the condition of the second line of the set equivalence definition is always False and the condition of the third line is always True. The second-line condition is always False because r_1 and r_2 have the same number of tuples and $RemoveAll$ functions remove the same number of tuples from each of the argument relations. Relations r_1 and r_2 must have the same number of tuples, because otherwise they would not be multiset equivalent (the second-line condition of the multiset-equivalence definition would evaluate to True), and $RemoveAll$ functions must remove the same number of tuples from each relation, because otherwise we would be able to find a tuple t in r_1 for which there will be no equivalent tuple in r_2, and the third-line condition of the multiset-equivalence definition would evaluate to False. Note that the $RemoveAll$ functions remove *all* the tuples from both argument relations that are equivalent to $head(r_1)$. Therefore, the third-line condition is always True because if, during some iteration step, the first tuple of the argument relation cannot be found in the second argument relation, it means that its equivalent does not exist in r_2 (it could not have been removed in previous iterations, because it exists in the first argument relation), and relations r_1 and r_2 cannot be equivalent as multisets (again, the third-line condition of the multiset-equivalence definition would evaluate to False).

Finally, we prove the implication $r_1 \equiv_L r_2 \Rightarrow r_1 \equiv_L^S r_2$. According to the definition of snapshot-list equivalence, two relations are snapshot-list equivalent if, at each point of time, their snapshots are equivalent as lists. If relations r_1 and r_2 are equivalent as lists, then, at each point of time c, two equivalent tuples that are in the same positions in those relations either both overlap with time c or neither of them overlap with time c, and, consequently, either both or neither non-temporal counterparts of the tuples are included in $\tau_c^T(r_1)$ and $\tau_c^T(r_2)$. Since the tuples are equal, their non-temporal counterparts are also equal. As the timeslice operation does not change the order of the argument tuples, all equivalent non-temporal counterparts are produced in the same order for both argument relations.

Proofs for the other four implications are similar. □

Theorem B.2 Transformation rule $\sigma_{P_1 \wedge P_2}(r) \equiv_L \sigma_{P_1}(\sigma_{P_2}(r))$ is correct.

Proof: We must prove that the relations produced by the left-hand side (r_{lhs}) and the right-hand side (r_{rhs}) of the transformation rule are list equivalent. Specifically, we prove that each tuple in position i in r_{lhs} exists in position i in r_{rhs}. The reverse implication may be proven analogously.

Assume that tuple t is in position i in r_{lhs}. It then follows that t is in r and satisfies $P_1 \wedge P_2$. Since t then also satisfies P_2 and P_1 independently, it follows that t is in r_{rhs}.

Next, let the number of tuples in r that satisfy $\neg(P_1 \wedge P_2)$ and occur before t be k. Similarly, let the number of tuples in r that satisfy $\neg P_2$ and occur before t be k_1 and the number of tuples in r that satisfy $\neg P_1 \wedge P_2$ and occur before t be k_2. Because the selection operator in Section 3.3.2 effects a linear scan over its input relation, it then follows that t occurs at position $i + k$ in r, at position $i + k - k_1$ in $\sigma_{P_2}(r)$, and at position $i + k - k_1 - k_2$ in r_{rhs}. The implication then follows if $k = k_1 + k_2$. This holds because any tuple that satisfies $\neg(P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$ satisfies exactly one of $\neg P_2$ or $\neg P_1 \wedge P_2$, which are exclusive, and because any tuple that satisfies one of the latter two also satisfies the former. □

Theorem B.3 The algorithm given in Figure 18 generates correct query plans.

Proof: To prove the theorem, we need to prove that the algorithm applies a transformation rule of some type only when the result produced by the new query plan is equivalent to the result produced by the original plan according to the top equivalence, which depends on the query language and the actual query statement. The proof is divided into six parts, one for each type of transformation rule. We provide a proof that $\equiv_M$ type rules are applied correctly if the top equivalence can be $\equiv_M$ or $\equiv_{L,A}$. The remaining parts of the proof can be worked out in similar fashion.

Consider the application of an $\equiv_M$ rule T. We denote the relation resulting from the left-side of T by r_T and the relation resulting from the right-side of T by r'_T. Consequently, we denote the result relation produced by the original query plan by $\mathit{result}(r_T)$ and the result relation produced by the new query plan by $\mathit{result}(r'_T)$. To prove that T is applied correctly, we have to prove that it does not violate any of the two possible top equivalences.

Case One First, we consider the case when the top equivalence is $\equiv_M$. From the definitions of the operations, we can derive the following implications. (We include all possible operations in this analysis.)

$$\forall op_u \in \{\sigma, \pi, \xi, \xi^T, \mathit{rdup}, \mathit{sort}\} \;\; r_1 \equiv_M r_2 \Rightarrow op_u(r_1) \equiv_M op_u(r_2)$$
$$\forall op_b \in \{\sqcup, \times, \times^T, \backslash, \cup\}, \forall r \;\; r_1 \equiv_M r_2 \Rightarrow op_b(r, r_1) \equiv_M op_b(r, r_2) \wedge op_b(r_1, r) \equiv_M op_b(r_2, r)$$
$$\forall op_u \in \{\mathit{rdup}^T, \mathit{coal}^T\} \;\; r_1 \equiv_M r_2 \Rightarrow op_u(r_1) \equiv_M op_u(r_2) \text{ if } \neg \mathit{MayHaveDupsInSn}(r_1)$$
$$r_1 \equiv_M r_2 \Rightarrow r_1 \backslash^T r \equiv_M r_2 \backslash^T r \text{ if } \neg \mathit{MayHaveDupsInSn}(r_1)$$
$$r_1 \equiv_M r_2 \Rightarrow r \cup^T r_1 \equiv_M r \cup^T r_2 \text{ if } \neg \mathit{MayHaveDupsInSn}(r_1)$$
$$\forall op_u \in \{\mathit{rdup}^T, \mathit{coal}^T\} \;\; r_1 \equiv_M r_2 \Rightarrow op_u(r_1) \equiv_M^S op_u(r_2) \text{ if } \mathit{MayHaveDupsInSn}(r_1)$$
$$r_1 \equiv_M r_2 \Rightarrow r_1 \backslash^T r \equiv_M^S r_1 \backslash^T r \text{ if } \mathit{MayHaveDupsInSn}(r_1)$$
$$r_1 \equiv_M r_2 \Rightarrow r \cup^T r_1 \equiv_M^S r \cup^T r_2 \text{ if } \mathit{MayHaveDupsInSn}(r_1)$$

Informally, this says that all operations preserve multiset equivalence, except coalescing, temporal duplicate elimination, temporal difference, and temporal union if their arguments may have duplicates in snapshots (the left argument for temporal difference and the right argument for temporal union count). In these offending cases, multiset snapshot equivalence is preserved. From the implications, it follows that each subsequent operation applied on r_T and r'_T produce relations that are $\equiv_M^S$ or $\equiv_M$ equivalent.

If each subsequent operation produces $\equiv_M$ equivalent relations, it follows that $\mathit{result}(r_T) \equiv_M \mathit{result}(r'_T)$. Let us consider the cases if there is an operation producing only $\equiv_M^S$ equivalent relations. If such an operation exists in the original query plan above r_T and its relevant argument may have duplicate in snapshots, from the definition of the $\mathit{OrderRequired}$ property we can derive that one of the three cases is true: (1) the $\mathit{OrderRequired}$ property holds for the argument of the operation down to the bottom of the query plan, (2) the $\mathit{OrderRequired}$ property holds for the argument down to the first sort operation that sorts on all attributes, or (3) the $\mathit{OrderRequired}$ property does not hold for the argument because the $\mathit{PeriodPreserving}$ property does not hold for the operation. The first alternative is not possible because the $\mathit{OrderRequired}$ does not hold for r_T (otherwise we would not have applied T at all). The second alternative implies that the sort should be above r_T (otherwise the $\mathit{OrderRequired}$ property would have to hold for r_T) and its produced relations in both query plans will be equivalent as lists. All operations for list-equivalent arguments produce list-equivalent results, therefore the results of the original and new query plans will be equivalent as lists. The third alternative implies that there is another operation op' above the operation in question such that its $\mathit{PeriodPreserving}$ property holds, but the property of its parent does not hold (we know that the property holds for the root). According to the definition of how to propagate the $\mathit{PeriodPreserving}$ property, five cases are possible when the property holds for the parent operation but does not hold for the child operation. It can be proved using the definitions of operations that op' would return $\equiv_M$ equivalent results for $\equiv_M^S$ equivalent arguments.

Case Two The second case is when the top equivalence is $\equiv_{L,A}$. Then, we know that the $\mathit{OrderRequired}$ holds for the root operation and that there was a sort operation at the top of the initial query plan. Let us consider the highest operation op on the path from r_T to the root in the original plan for which the $\mathit{OrderRequired}$ does not hold. Due to the definition of the property (recall Table 8) and the sorting transformation rules (the sort operation cannot be pushed down below union ALL or regular and temporal union), only three instances of op are possible. From the first part of the present theorem, we know that arguments to such operation in both query plans will be $\equiv_M^S$ or $\equiv_M$ equivalent.

First, op can be the sort operation. From its definition, we can derive the following implications.

$$r_1 \equiv_M r_2 \Rightarrow \mathit{sort}_B(r_1) \equiv_{L,B} \mathit{sort}_B(r_2)$$
$$r_1 \equiv_M^S r_2 \Rightarrow \mathit{sort}_B(r_1) \equiv_{L,B}^S \mathit{sort}_B(r_2)$$

Second, op can be a Cartesian product or a difference, where r_T contributes to the right argument and the left argument has some order. From the definitions of $\times$ and $\backslash$, we derive the following implications.

$$r_1 \equiv_M r_2 \Rightarrow op(r, r_1) \equiv_{L,Order(r)} op(r, r_2), \text{ where } Order(r) \neq \perp \textit{ and } op \in \{\times, \backslash\}$$
$$r_1 \equiv_M^S r_2 \Rightarrow op(r, r_1) \equiv_{L,Order(r)}^S op(r, r_2), \text{ where } Order(r) \neq \perp \textit{ and } op \in \{\times, \backslash\}$$

Third, *op* can be a temporal Cartesian product or a temporal difference, where r_T contributes to the right argument and the left argument has some order. The following implications hold.

$$r_1 \equiv_M r_2 \Rightarrow op(r, r_1) \equiv_{L,Order(r)\backslash TimePairs} op(r, r_2), \text{ where } Order(r) \neq \perp \textit{ and } op \in \{\times^T, \backslash^T\}$$
$$r_1 \equiv_M^S r_2 \Rightarrow op(r, r_1) \equiv_{L,Order(r)\backslash TimePairs}^S op(r, r_2), \text{ where } Order(r) \neq \perp \textit{ and } op \in \{\times^T, \backslash^T\}$$

Thus, *op* can either produce $\equiv_{L,B}$ or $\equiv_{L,B}^S$ equivalent relations in both query plans, where B is some non-empty order.

First, we consider the case when the two relations in both query plans are $\equiv_{L,B}$ equivalent. Most operations preserve partial list equivalence, i.e., from the definitions of those operations, we can derive that results of any such operation applied to $result_{op}$ and $result'_{op}$ are $\equiv_{L,C}$ equivalent, where both results are sorted on C, which is the value of the Order column in Table 1 for the operation applied. Consequently, if only such operations are used, since the original query plan produces the final relation sorted on A, the new query plan also produces such relation, i.e., $result(r_T) \equiv_{L,A} result(r'_T)$. The only operations not preserving partial list equivalence (yet preserving partial list snapshot equivalence) are the four operations outlined in the first part of the proof, if their relevant arguments may have duplicates in snapshots. If such operations exist above *op*, we consider op' which is the highest among them, and, according to the definition of the $OrderRequired$ property, either (1) $PeriodPreserving(op')$ is False or (2) the $OrderRequired$ holds for the arguments of op' and other operations down until to the first $sort$ operation that sorts on all attributes (we know that such $sort$ operation exists because the $OrderRequired$ would have been True for all operations down to the bottom of the tree and we could not have applied T).

If we have the first case, since the $PeriodPreserving$ property holds for the root, we know that there is an operation, with this property set, above op' (which returns partial snapshot list-equivalent results in both plans). As shown in the first part of the proof, such an operation would return partial list-equivalent results for partial snapshot list-equivalent arguments. If we have the second case, we know that the $sort$ operation should be above r_T (otherwise the $OrderRequired$ property would have to hold for r_T), and above of or equal to *op* (which is the highest operation which changes the value of the $OrderRequired$ property from True to False). Its produced relations in both query plans will be equivalent as lists. Since all operations for list-equivalent arguments produce list-equivalent results, the results of both query plans will be equivalent as lists.

If the two relations in both query plans are $\equiv_{L,B}^S$ equivalent, then operation op', which returns $\equiv_M^S$ equivalent results when having $\equiv_M$ equivalent arguments, exists between r_T and op. Similarly as in the first part of the proof, there must exist an operation above op and op' that would return partial list-equivalent results for partial snapshot list-equivalent argument. $\square$