Extending the kernel of a relational dbms with comprehensive support for sequenced temporal queries

Dignös, Anton ; Böhlen, Michael Hanspeter ; Gamper, Johann ; Jensen, Christian S

Abstract: Many databases contain temporal, or time-referenced, data and use intervals to capture the temporal aspect. While SQL-based database management systems (DBMSs) are capable of supporting the management of interval data, the support they offer can be improved considerably. A range of proposed temporal data models and query languages offer ample evidence to this effect. Natural queries that are very difficult to formulate in SQL are easy to formulate in these temporal query languages. The increased focus on analytics over historical data where queries are generally more complex exacerbates the difficulties and thus the potential benefits of a temporal query language. Commercial DBMSs have recently started to offer limited temporal functionality in a step-by-step manner, focusing on the representation of intervals and neglecting the implementation of the query evaluation engine. This article demonstrates how it is possible to extend the relational database engine to achieve a full-fledged, industrial-strength implementation of sequenced temporal queries, which intuitively are queries that are evaluated at each time point. Our approach reduces temporal queries to nontemporal queries over data with adjusted intervals, and it leaves the processing of nontemporal queries unaffected. Specifically, the approach hinges on three concepts: interval adjustment, timestamp propagation, and attribute scaling. Interval adjustment is enabled by introducing two new relational operators, a temporal normalizer and a temporal aligner, and the latter two concepts are enabled by the replication of timestamp attributes and the use of so-called scaling functions. By providing a set of reduction rules, we can transform any temporal query, expressed in terms of temporal relational operators, to a query expressed in terms of relational operators and the two new operators. We prove that the size of a transformed query is linear in the number of temporal operators in the original query. An integration of the new operators and the transformation rules, along with query optimization rules, into the kernel of PostgreSQL is reported. Empirical studies with the resulting temporal DBMS are covered that offer insights into pertinent design properties of the article's proposal. The new system is available as open-source software.

DOI: https://doi.org/10.1145/2967608

Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries

ANTON DIGNÖS, Free University of Bozen-Bolzano
MICHAEL H. BÖHLEN, University of Zurich
JOHANN GAMPER, Free University of Bozen-Bolzano
CHRISTIAN S. JENSEN, Aalborg University

Many databases contain temporal, or time-referenced, data and use intervals to capture the temporal aspect. While SQL-based database management systems (DBMSs) are capable of supporting the management of interval data, the support they offer can be improved considerably. A range of proposed temporal data models and query languages offer ample evidence to this effect. Natural queries that are very difficult to formulate in SQL are easy to formulate in these temporal query languages. The increased focus on analytics over historical data where queries are generally more complex exacerbates the difficulties and thus the potential benefits of a temporal query language. Commercial DBMSs have recently started to offer limited temporal functionality in a step-by-step manner, focusing on the representation of intervals and neglecting the implementation of the query evaluation engine.

This paper demonstrates how it is possible to extend the relational database engine to achieve a full-fledged, industrial-strength implementation of sequenced temporal queries, which intuitively are queries that are evaluated at each time point. Our approach reduces temporal queries to nontemporal queries over data with adjusted intervals, and it leaves the processing of nontemporal queries unaffected. Specifically, the approach hinges on three concepts: *interval adjustment*, *timestamp propagation*, and *attribute scaling*. Interval adjustment is enabled by introducing two new relational operators, a temporal normalizer and a temporal aligner, and the latter two concepts are enabled by the replication of timestamp attributes and the use of so-called scaling functions. By providing a set of reduction rules, we can transform any temporal query, expressed in terms of temporal relational operators, to a query expressed in terms of relational operators and the two new operators. We prove that the size of a transformed query is linear in the number of temporal operators in the original query. An integration of the new operators and the transformation rules, along with query optimization rules, into the kernel of PostgreSQL is reported. Empirical studies with the resulting temporal DBMS are covered that offer insights into pertinent design properties of the paper's proposal. The new system is available as open source software.

Categories and Subject Descriptors: H.2 [**Database Management**]: Systems—*Relational databases*

Additional Key Words and Phrases: Attribute value scaling, Change preservation, Interval adjustment, Query processing, Sequenced semantics, Temporal databases, Timestamp propagation

ACM Reference Format:
Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen, 2016. Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries. *ACM Trans. Datab. Syst.* V, N, Article XXXX (January 2016), 45 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

This work was supported in part by the TPG project of the Free University of Bozen-Bolzano and by a grant from the Obel Family Foundation.
Author's addresses: A. Dignös (corresponding author) and J. Gamper, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy; email: dignoes@inf.unibz.it; M. H. Böhlen, Department of Computer Science, University of Zurich, Switzerland; C. S. Jensen, Department of Computer Science, Aalborg University, Denmark.

DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Intervals are used to capture temporal aspects of data, e.g., to capture when data is valid in the reality that the data models. We call data with such intervals temporal data. Database management systems (DBMS) readily support the storage of intervals, either using separate start and end time attributes or using a single interval-valued attribute. In contrast, the support for the formulation and processing of sequenced temporal queries against temporal data is very limited. The reason is that sequenced queries require groups of intervals that are aligned with respect to each other, i.e., that are either identical or disjoint, and the DBMS does not support such adjustments of intervals.

As a result, application code and SQL queries on temporal data are often intricate and difficult to formulate, understand, and prove correct [Snodgrass 2000]. The research community has responded by proposing dozens of temporal query languages that aim to make the querying of temporal data easier. More recently, DBMS vendors have also begun to offer limited temporal query language functionality [Al-Kateb et al. 2013; Kulkarni and Michels 2012].

While the querying of temporal data is quite well understood, the key remaining problem is how to achieve an industrial-strength and systematic DBMS implementation of a comprehensive temporal query language. Existing studies often simplify the problem, making assumptions that are not generally valid, e.g., assume that an interval has no meaning beyond a set of points, or consider only selected aspects of a temporal query language, e.g., so-called time travel queries or temporal joins.

We present an approach that does not make such assumptions and that builds comprehensive support for sequenced temporal querying into a relational engine. This approach integrates, in a systematic and wholesale manner, temporal support into an existing system without affecting the system's support for nontemporal queries. The key idea is to transform temporal queries into nontemporal queries with interval adjustment. This is achieved with the help of a four-step transformation of the relations that are used as arguments in temporal queries, encompassing timestamp propagation, interval adjustment, attribute value scaling, and operator transformation.

First, *timestamp propagation* replicates the timestamp attributes in argument relations. This is necessary because intervals will be adjusted and the original intervals are needed to scale attribute values and to evaluate query predicates and functions that reference the original timestamps. Second, *interval adjustment* replicates the input tuples and assigns different time intervals to them. The intervals are obtained by splitting the original intervals such that they match the intervals to be associated with the result tuples. Interval adjustment is achieved by introducing two new relational operators, a temporal normalizer and a temporal aligner, into the database engine. Third, *attribute value scaling* scales attribute values in a tuple to accord with the tuple's new time interval. Scaling relies on the original attribute value and the tuple's original and new intervals for its functioning. Our solution supports different scaling functions that can be provided as user-defined functions. Fourth, the above elements make it possible to transform a query with temporal operators into a query expressed in terms of the corresponding standard relational operators and the two new operators. This query is able to treat intervals as atomic values, since adjusted intervals can be compared using equality, and can be processed natively by the DBMS.

Example 1.1. Fig. 1 illustrates our approach. Relation p stores information about projects: the project number (P), the department a project is associated with (D), and the project budget (B). Attribute T is interval-valued and captures when the information recorded by the values of the other attributes is valid, or true. For instance, tuple r_1 captures the fact, that project P_1 belongs to the CS department and has a budget of 5K, is valid from January through May 2014. Attribute T thus plays a special role: its values are *about* the values of the other attributes. Consider Query Q_1 = *"At each time point, what is the*

The reduction of temporal queries to nontemporal operators works in four steps:

1) *Propagate the original intervals.* This is necessary since the original intervals are needed when scaling budget values to the adjusted intervals. To compute a scaled value, we need the original value and the original and adjusted intervals.

2) *Adjustment of intervals.* The DBMS kernel supports the adjustment with two new operators: a temporal normalizer $\mathcal{N}$ and a temporal aligner Φ. The adjusted interval attribute T in x_2 illustrates that the intervals have been adjusted with respect to each other. All intervals that later must be processed together are identical.

3) *Scaling of attribute values.* We use the values of attributes B, U, and T to scale the budget values. Note that each tuple in x_2 includes the information necessary for scaling. For instance, the first tuple in x_2 records that the 5K budget for 5 months is to be scaled to a 3 months budget.

4) *Evaluation of the corresponding nontemporal operator.* Instead of a temporal aggregation over the original argument relation, we perform a nontemporal aggregation over the relation with adjusted intervals and scaled budgets. The adjusted interval attribute T is used for grouping like other attributes.

p

	P	D	B	T
r_1	P_1	CS	5K	[2014-1, 2014-6)
r_2	P_2	CS	6K	[2014-4, 2014-7)
r_3	P_3	MA	2K	[2014-1, 2014-3)

1)↓

$\mathsf{x}_1 = \epsilon_U(\mathsf{p})$

P	D	B	U	T
P_1	CS	5K	[2014-1, 2014-6)	[2014-1, 2014-6)
P_2	CS	6K	[2014-4, 2014-7)	[2014-4, 2014-7)
P_3	MA	2K	[2014-1, 2014-3)	[2014-1, 2014-3)

2)↓

$\mathsf{x}_2 = \mathcal{N}_{\mathsf{r}.D=\mathsf{s}.D}(\mathsf{x}_1/\mathsf{r}, \mathsf{x}_1/\mathsf{s})$

P	D	B	U	T
P_1	CS	5K	[2014-1, 2014-6)	[2014-1, 2014-4)
P_1	CS	5K	[2014-1, 2014-6)	[2014-4, 2014-6)
P_2	CS	6K	[2014-4, 2014-7)	[2014-4, 2014-6)
P_2	CS	6K	[2014-4, 2014-7)	[2014-6, 2014-7)
P_3	MA	2K	[2014-1, 2014-3)	[2014-1, 2014-3)

3)↓

$\mathsf{x}_3 = \pi_{P,D,scaleU(B,T,U)/B,T}(\mathsf{x}_2)$

P	D	B	T
P_1	CS	3K	[2014-1, 2014-4)
P_1	CS	2K	[2014-4, 2014-6)
P_2	CS	4K	[2014-4, 2014-6)
P_2	CS	2K	[2014-6, 2014-7)
P_3	MA	2K	[2014-1, 2014-3)

4)↓

$\mathsf{z} = {}_{T,D}\vartheta_{SUM(B)/B}(\mathsf{x}_3)$

	D	B	T
z_1	CS	3K	[2014-1, 2014-4)
z_2	CS	6K	[2014-4, 2014-6)
z_3	CS	2K	[2014-6, 2014-7)
z_4	MA	2K	[2014-1, 2014-3)

Fig. 1. Reduction of temporal query Q_1 using timestamp propagation, interval adjustment, and scaling.

available project budget per department?". This is a sequenced temporal aggregation that asks for the available project budget per department and time point. Thus, for each department, the sum of the budgets over all projects that run concurrently for some time period must be computed. If a project extends beyond this time period, a part of the project's budget is not available during this time period, and the budget must be scaled. Relation z in Fig. 1 displays the desired result.

In temporal relational algebra query Q_1 is expressed as follows:

$$Q_1 = {}_D\vartheta^{T:B@scaleU}_{SUM(B)}(\mathsf{p})$$

The ^{T}-superscript indicates that the aggregation operator ϑ is sequenced such that aggregation is performed at each point in time. The $^{B@scaleU}$-superscript specifies that budget B is to be scaled using function $scaleU$. Thus, the aggregation function $SUM(B)$ is computed over scaled budgets.

Although Q_1 is simple to formulate and understand, the query is difficult to formulate in SQL. A first difficulty is that the time intervals are not equal but overlap. Because of the different running times of projects, we cannot simply sum the project budgets. Instead, the intervals must first be adjusted. Second, the budget applies to the entire project period. If intervals are adjusted, the budgets must be modified as well so that the modified budgets match the adjusted intervals. In addition to being difficult to formulate in SQL, existing DBMSs do not offer adequate support to process such queries.

Our approach advances the state-of-the-art in several respects. First, it is the first to support all operators of a comprehensive sequenced temporal algebra. In contrast, previous work focuses on efficient algorithms for specific operators, such as time-travel [Lomet et al. 2006; Rajamani 2007], temporal joins [Segev 1993; Gao et al. 2005], and temporal aggregation [Böhlen et al. 2006b; Vega Lopez et al. 2005; Gamper et al. 2009]. Second, instead of providing new evaluation algorithms for each temporal operator, the approach makes it possible to transform a query with sequenced temporal operators into a query that involves only the nontemporal counterparts and possibly the two adjustment operators. Although the size of the transformed expressions can be exponential in the number of temporal operators, we show that it is possible to guarantee linear complexity by employing common table expressions. Our approach is able to fully leverage an existing implementation of the relational algebra in a database engine, along with its existing query optimization and indexing techniques. Only two new operators need to be integrated into the engine. Third, the approach is able to support a sequenced temporal algebra with all features that have been identified in previous research as important for the processing of temporal data, namely snapshot reducibility, change preservation, extended snapshot reducibility, and attribute value scaling (these properties are jointly referred to as the sequenced semantics [Böhlen and Jensen 2009][1]).

The paper makes the following technical contributions:

— We introduce *interval adjustment* and *timestamp propagation* as key mechanisms for a database engine to natively support snapshot reducibility, change preservation, extended snapshot reducibility, and attribute value scaling.
— Our solution offers comprehensive support for the *scaling of attribute values* in sequenced temporal operators. The scaling is not limited to the pre- or post-processing of values, which limit expressiveness. Scaling is possible in grouping and join predicates and in aggregate functions.
— We define two new relational operators: a *temporal normalizer* and a *temporal aligner* for the adjustment of intervals. The former adjusts intervals for group based operators ($\{\pi, \vartheta, \cup, -, \cap\}$), and the latter adjusts intervals for tuple based operators ($\{\sigma, \times, \bowtie, \rtimes, \ltimes, \overline{\rtimes}, \triangleright\}$). We provide optimization rules for the new temporal operators.
— We define a set of *reduction rules* that reduce sequenced operators to their nontemporal counterparts. The reduction rules cover all algebra operators, including outer joins, anti joins, and aggregations. Predicates and functions can contain explicit references to interval attributes. We prove that the temporal algebra defined by the reduction rules satisfies the sequenced semantics.
— We show how to use the reduction rules together with SQL's common table expressions to get nontemporal relational algebra expressions with a number of operators that is linear in the number of operators of the original temporal relational algebra expressions.
— We describe an implementation of the sequenced temporal operators and the reduction rules in the kernel of PostgreSQL and report on extensive experiments that offer insight into the effectiveness and efficiency of the integration.

[1]Attribute value scaling was not mentioned explicitly in the original works on the sequenced semantics.

The rest of the paper is structured as follows. Section 2 introduces preliminary concepts and notation. Section 3 formalizes the sequenced semantics of the temporal relational algebra. Section 4 describes our approach to process sequenced temporal queries by transforming them to the nontemporal counterparts with the help of interval adjustment, timestamp propagation, and attribute scaling. Transformation rules for all algebra operators are given. Section 5 describes the implementation in the kernel of PostgreSQL. Section 6 reports on the experiments. Finally, Section 7 discusses related work, and Section 8 concludes the paper.

The paper integrates and extends two existing works on interval alignment [Dignös et al. 2012] and scaling of attribute values [Dignös et al. 2013]. The extensions over this past work are detailed in Section 7. The definition of the algebra and some detailed proofs can be found in electronic appendix accessible in the ACM Digital Library.

2. PRELIMINARIES

We assume a linearly ordered, discrete time domain, Ω^T. A time interval is a contiguous set of time points, and $[T_s, T_e)$ denotes the closed-open interval of points from T_s to T_e. A tuple in a temporal relation is timestamped with an interval that represents the tuple's valid time. The schema of a temporal relation is given by $R = (A_1, \ldots, A_m, T)$, where $A_1, \ldots, A_m$ are the nontemporal attributes with domains Ω_i and T is the timestamp attribute with domain $\Omega^T \times \Omega^T$. A temporal relation r with schema R is a finite set of tuples, where each tuple has a value in the appropriate domain for each attribute in the schema. We use $r.A_i$ to denote the value of attribute A_i in tuple r. We use the abbreviations $\mathbf{A} = \{A_1, \ldots, A_m\}$ and $r.\mathbf{A} = (r.A_1, \ldots, r.A_m)$, and we use r/s to denote the renaming of r to s.

We assume a relational algebra with the following temporal operators with sequenced semantics: selection σ^T, projection π^T, aggregation ϑ^T, difference $-^T$, union $\cup^T$, intersection $\cap^T$, Cartesian product $\times^T$, join $\bowtie^T$, left outer join $⟕^T$, right outer join $⟖^T$, full outer join $⟗^T$, and anti join $\triangleright^T$. Each of these temporal operators is a generalization of a standard relational operator that does not possess the ^{T}-superscript. The standard relational algebra is given in the electronic Appendix A. For set operators, we assume union compatible argument relations; and for $\pi^T_{\mathbf{B}}(\mathsf{r})$ and ${}_{\mathbf{B}}\vartheta^T_F(\mathsf{r})$, we require $\mathbf{B} \subseteq \mathbf{A}$. Next, $sch(\psi)$ denotes the schema of the relation defined by the relational algebra expression ψ. We assume duplicate free temporal relations, i.e., there are no value-equivalent tuples over common timepoints: r is *duplicate free* iff $\forall r \in \mathsf{r}\ (\forall r' \in \mathsf{r}\ (r \neq r' \Rightarrow r.\mathbf{A} \neq r'.\mathbf{A} \vee r.T \cap r'.T = \emptyset))$.

The *snapshot* of a temporal relation at a time point t is the nontemporal relation that is valid at t, and it is defined in terms of the *timeslice* operator τ [Jensen and Snodgrass 2009], i.e., $\tau_t(\mathsf{r}) = \{r.\mathbf{A} \mid r \in \mathsf{r} \wedge t \in r.T\}$.

Table I summarizes the most important notation used in this article.

Table I. Summary of notation.

Notation	Example	Description
$/$	r/s	Renaming of relations and attributes
sch	$sch(\mathsf{r})$	Schema of a relation
τ	$\tau_{2015\text{-}2}(\mathsf{r})$	Timeslice operator
ψ	$\psi(\mathsf{r}, \mathsf{s})$	Relational algebra operator
ψ^T	$\psi^T(\mathsf{r}, \mathsf{s})$	Sequenced temporal relational algebra operator
L	$\mathsf{L}[\mathsf{r} \times^T \mathsf{s}](z, 2014\text{-}3)$	Lineage set (Def. 3.2)
ϵ	$\epsilon_U(\mathsf{r})$	Extend operator (Def. 3.6)
$\mathcal{N}$	$\mathcal{N}_{\mathsf{r}.D=\mathsf{s}.D}(\mathsf{r}, \mathsf{s})$	Temporal normalization (Def. 4.3)
Φ	$\Phi_{\mathsf{r}.D=\mathsf{s}.D}(\mathsf{r}, \mathsf{s})$	Temporal alignment (Def. 4.8)
α	$\alpha(\mathsf{r})$	Absorb operator (Def. 4.15)

3. SEQUENCED TEMPORAL QUERY LANGUAGE SEMANTICS

A standard relational DBMS processes SQL queries by mapping them to relational algebra expressions. As we aim at supporting sequenced temporal queries, we proceed to define a temporal relational algebra to which queries in user-level temporal query languages are mapped and that plays the same role in the temporal setting as does the relational algebra in the standard setting. The semantics of the temporal algebra, called sequenced semantics [Böhlen and Jensen 2009], are defined in terms of four properties, namely

— snapshot reducibility,
— change preservation,
— extended snapshot reducibility, and
— scaling.

We purposely do not define a user-level temporal query language and its mapping to the temporal algebra, as these are orthogonal to the paper's focus on implementation. Indeed, different user-level temporal languages may be mapped to the temporal relational algebra.

Following an introductory section, we cover the four properties of the sequenced semantics in turn.

3.1. Temporal Data, Queries, and Semantics

The querying capabilities of temporal DBMSs can be partitioned into three modes [Snodgrass et al. 1997; Böhlen et al. 2000; Snodgrass 2010]: *nonsequenced*, *current*, and *sequenced semantics*.

The nonsequenced semantics is time agnostic, and applications must specify explicitly how to process the temporal information. DBMSs support the nonsequenced semantics [Böhlen et al. 2009] by extending SQL with new data types, predicates, and functions. Predicates such as `OVERLAPS`, `BEFORE`, and `CONTAINS` are now part of the SQL:2011 standard. Another approach to specify temporal relationships are the operators of temporal logic, which target the reasoning across different database states [Chomicki et al. 2001]. The nonsequenced semantics is the most flexible and expressive semantics since applications handle timestamps like all other attributes without any implicit meaning being enforced.

The current semantics [Böhlen et al. 2009; Bair et al. 1997] performs query processing on the snapshot at the current time and can be realized by restricting the data to the current time. The current semantics is present in the SQL:2011 standard, where standard SQL queries over system-versioned tables [Kulkarni and Michels 2012] evaluate queries on the current snapshot. As a simple extension to the current semantics, so-called time travel queries allow to specify any snapshot of interest. The integration of the current semantics into a database engine is usually done with the help of selection operations.

The sequenced semantics [Böhlen and Jensen 2009] is consistent with viewing a temporal database as a sequence of nontemporal databases and evaluates statements at each time point. It is difficult to support, and various studies have shown that the formulation of sequenced statements in standard SQL is complex and awkward [Snodgrass 2000; Böhlen et al. 2000; Li et al. 2001]. As evaluating queries at each time point is prohibitive in terms of performance, DBMSs must provide built-in support for the sequenced semantics.

We provide comprehensive support for the sequenced semantics of temporal queries without limiting the use of the nonsequenced semantics. The approach is systematic and separates the interval adjustment from the evaluation of the operators. This strategy renders it possible to fully leverage the DBMSs query optimization and evaluation engine for sequenced temporal query processing.

Temporal queries are expected to yield results that are consistent with the information recorded by the relations they take as arguments. Consider the temporal aggregation query Q'_1 = *"At each time point, what is the number, average duration, and available project*

budget per department?" The result of this query on relation p is shown in Fig. 2. The tuple labeled z'_2 reports the result for the time period [2014-4, 2014-6). This tuple derives from argument tuples r_1 and r_2. Specifically, P_1 lasts 5 months and has a budget of 5K, while P_2 lasts 3 months and has a budget of 3K. P_1 and P_2 overlap during [2014-4, 2014-6). This results in 2 projects for this interval (CNT), an average duration of $(5+3)/2 = 4$ months (AVG), and contributions of $5\text{K} \cdot 2/5$ and $6\text{K} \cdot 2/3$ in this interval. Thus, the amount of 6K is consistent with the amounts in r_1 and r_2. Similarly, the total budget across all times, departments, and projects is 5K + 6K + 2K = 13K according to p. This is identical to 3K + 6K + 2K + 2K = 13K, which is the total budget according to the result relation.

	D	CNT	AVG	SUM	T
z'_1	CS	1	5	3K	[2014-1, 2014-4)
z'_2	CS	2	4	6K	[2014-4, 2014-6)
z'_3	CS	1	3	2K	[2014-6, 2014-7)
z'_4	MA	1	2	2K	[2014-1, 2014-3)

Fig. 2. Number, average duration and budget of projects per department (result of query Q'_1).

3.2. Snapshot Reducibility

Snapshot reducibility [Lorentzos and Mitsopoulos 1997; Soo et al. 1995] is a fundamental concept in temporal query languages. Intuitively, snapshot reducibility corresponds to the use of "at each time point" in natural-language formulations of queries. It captures the property that the result of a temporal query on a temporal database must be consistent with the snapshots that are obtained by computing the corresponding nontemporal query on each snapshot of the temporal database.

Definition 3.1. (*Snapshot Reducibility*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, ψ^T be an n-ary temporal operator, ψ be the corresponding nontemporal operator, Ω^T be the time domain, and $\tau_p(\mathsf{r})$ be the timeslice operator. Operator ψ^T is *snapshot reducible* to ψ iff for all $t \in \Omega^T$:

$$\tau_t(\psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)) \equiv \psi(\tau_t(\mathsf{r}_1), \ldots, \tau_t(\mathsf{r}_n)).$$

To see how this property is satisfied, consider the temporal aggregation query Q''_1 = *"At each time point, what is the number of projects per department?"* with the temporal algebra formulation ${}_D\vartheta^T_{CNT(*)}(\mathsf{p})$. The result is shown in Fig. 3. If we consider the snapshot of p at time 2014-4 and apply the corresponding nontemporal operator ${}_D\vartheta_{CNT(*)}(\tau_{\text{2014-4}}(\mathsf{p}))$ we get the tuple (CS, 2). This is the same as the snapshot at 2014-4 of the four tuples in Fig. 3.

	D	CNT	T
z''_1	CS	1	[2014-1, 2014-4)
z''_2	CS	2	[2014-4, 2014-6)
z''_3	CS	1	[2014-6, 2014-7)
z''_4	MA	1	[2014-1, 2014-3)

Fig. 3. Number of projects per department (result of query Q''_1).

3.3. Change Preservation

Snapshot reducibility only constrains the result of a temporal operator, but does not fully define the result. To see that, observe that if tuple z_2'' in Fig. 3 is replaced by the two tuples $(\text{CS}, 2, [\text{2014-4}, \text{2014-5}))$ and $(\text{CS}, 2, [\text{2014-5}, \text{2014-6}))$, the result still satisfies snapshot reducibility. The question that remains is which intervals to associate with the same non-temporal attribute values of a tuple. We proceed to provide a definition that creates a new interval exactly when the data lineage changes, i.e., when there is a change in the argument tuples that combine to yield a result tuple. This yields maximal time intervals for the result tuples over which the argument relations are constant.

Data lineage [Cui et al. 2000; Boulakia and Tan 2009] can be used to define the argument tuples that combine to produce a result tuple and thus shape the intervals of result tuples. Specifically, we adopt the influence contribution semantics [Glavic 2010; Glavic et al. 2010] to define the lineage of operators in temporal databases.

The *lineage set* of a result tuple z at time point t defines the argument tuples that are relevant for the computation of the result tuple at that time point.

Definition 3.2. (*Lineage Set*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, t be a time point, and $z \in \psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)$ be a result tuple of an n-ary snapshot reducible temporal operator ψ^T such that $t \in z.T$. The *lineage set*, $\mathsf{L}[\psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)](z,t)$, of tuple z at time t is the set of witness lists of argument tuples, $\{\langle r_1, \ldots, r_n\rangle\}$, $r_i \in \mathsf{r}_i$, from which z is derived:

$$
\begin{aligned}
\mathsf{L}[\sigma^T_\theta(\mathsf{r})](z,t) &= \{\langle r\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}\\
\mathsf{L}[\pi^T_{\mathbf{B}}(\mathsf{r})](z,t) &= \{\langle r\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{B} = r.\mathbf{B} \wedge t \in r.T\}\\
\mathsf{L}[{}_{\mathbf{B}}\vartheta^T_F(\mathsf{r})](z,t) &= \{\langle r\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{B} = r.\mathbf{B} \wedge t \in r.T\}\\
\mathsf{L}[\mathsf{r} -^T \mathsf{s}](z,t) &= \{\langle r, \bot\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}\\
\mathsf{L}[\mathsf{r} \cap^T \mathsf{s}](z,t) &= \{\langle r, s\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T \wedge s \in \mathsf{s} \wedge z.\mathbf{A} = s.\mathbf{A} \wedge t \in s.T\}\\
\mathsf{L}[\mathsf{r} \cup^T \mathsf{s}](z,t) &= \{\langle r, \bot\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\} \cup \{\langle \bot, s\rangle \mid s \in \mathsf{s} \wedge z.\mathbf{A} = s.\mathbf{A} \wedge t \in s.T\}\\
\mathsf{L}[\mathsf{r} \times^T \mathsf{s}](z,t) &= \{\langle r, s\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T \wedge s \in \mathsf{s} \wedge z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\}\\
\mathsf{L}[\mathsf{r} \bowtie^T_\theta \mathsf{s}](z,t) &= \{\langle r, s\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T \wedge s \in \mathsf{s} \wedge z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\}\\
\mathsf{L}[\mathsf{r} \rhd^T_\theta \mathsf{s}](z,t) &= \{\langle r, \bot\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}\\
\mathsf{L}[\mathsf{r} \rtimes^T_\theta \mathsf{s}](z,t) &= \begin{cases} \mathsf{L}[\mathsf{r} \rhd^T_\theta \mathsf{s}](z,t) & \text{if } \nexists s \in \mathsf{s}(\theta(z.\mathbf{A}, s) \wedge t \in s.T)\\ \mathsf{L}[\mathsf{r} \bowtie^T_\theta \mathsf{s}](z,t) & \text{otherwise}\end{cases}\\
\mathsf{L}[\mathsf{r} \ltimes^T_\theta \mathsf{s}](z,t) &= \begin{cases} \{\langle \bot, s\rangle \mid s \in \mathsf{s} \wedge z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\} & \text{if } \nexists r \in \mathsf{r}(\theta(r, z.\mathbf{C}) \wedge t \in r.T)\\ \mathsf{L}[\mathsf{r} \bowtie^T_\theta \mathsf{s}](z,t) & \text{otherwise}\end{cases}\\
\mathsf{L}[\mathsf{r} \rtimes\!\ltimes^T_\theta \mathsf{s}](z,t) &= \begin{cases} \mathsf{L}[\mathsf{r} \rhd^T_\theta \mathsf{s}](z,t) & \text{if } \nexists s \in \mathsf{s}(\theta(z.\mathbf{A}, s) \wedge t \in s.T)\\ \{\langle \bot, s\rangle \mid s \in \mathsf{s} \wedge z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\} & \text{if } \nexists r \in \mathsf{r}(\theta(r, z.\mathbf{C}) \wedge t \in r.T)\\ \mathsf{L}[\mathsf{r} \bowtie^T_\theta \mathsf{s}](z,t) & \text{otherwise}\end{cases}
\end{aligned}
$$

A single witness list w in a lineage set is an element of the Cartesian product of the argument relations and represents one combination of input tuples that are used together to derive a tuple. For instance, $w = \langle r, s\rangle$ is a witness list for a binary operator, where r is a tuple from the first input relation and s is a tuple from the second input relation. Evaluating a relational algebra operator over a lineage set $\{\langle r', s'\rangle, \langle r'', s''\rangle\}$ means to evaluate the operator using $\{r', r''\}$ as the first input relation and $\{s', s''\}$ as the second input relation (cf. Definition 3.3 in Glavic [2010]).

The lineage set of a result tuple z of the timeslice operator is $\mathsf{L}[\tau_t(\mathsf{r})](z) = \{\langle r\rangle \mid r \in \mathsf{r} \wedge z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}$.

Example 3.3. We continue the previous example and consider the result of query $Q_1'' = {}_D\vartheta^T_{CNT(*)}(\mathsf{p})$ in Fig. 3. For the result tuples z_1'', z_2'', and z_3'' and the time points 2014-3, 2014-4, 2014-5 and 2014-6, we get the following lineage sets (where $\psi^T = {}_D\vartheta^T_{CNT(*)}$):

— $\mathsf{L}[\psi^T(\mathsf{p})](z_1'', \text{2014-3}) = \{\langle r_1\rangle\}$
— $\mathsf{L}[\psi^T(\mathsf{p})](z_2'', \text{2014-4}) = \{\langle r_1\rangle, \langle r_2\rangle\}$
— $\mathsf{L}[\psi^T(\mathsf{p})](z_2'', \text{2014-5}) = \{\langle r_1\rangle, \langle r_2\rangle\}$
— $\mathsf{L}[\psi^T(\mathsf{p})](z_3'', \text{2014-6}) = \{\langle r_2\rangle\}$

For instance, $\mathsf{L}[\psi^T(\mathsf{p})](z_1'', \text{2014-3}) = \{\langle r_1\rangle\}$ states that result tuple z_1'' for $\psi^T = {}_D\vartheta^T_{CNT(*)}$ at time point 2014-3 exists because of input tuple r_1.

Data lineage captures the argument tuples that yield a result tuple, and we use it for defining the time intervals of the result tuples. Thus, when combined with snapshot reducibility, lineage defines the result of a temporal operator. By combining contiguous time points with identical lineage sets into the same interval, we get result tuples with maximal time intervals that preserve changes [Böhlen et al. 2000; Böhlen and Jensen 2009][2].

Definition 3.4. (*Change Preservation*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, $\mathsf{z} = \psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)$ be the result of an n-ary temporal operator ψ^T, and $\mathsf{L}(z,t) = \mathsf{L}[\psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)](z,t)$ be the lineage set of result tuple $z \in \mathsf{z}$ at time point t. Temporal operator ψ^T is *change preserving* iff for all $z \in \mathsf{z}$ and $z' \in \mathsf{z}$:

$$\forall t \in z.T(\mathsf{L}(z,t) = \mathsf{L}(z, z.T_s)) \wedge$$
$$\forall t \in \{z.T_s - 1, z.T_e\}(t \in z'.T \Rightarrow \mathsf{L}(z',t) \neq \mathsf{L}(z, z.T_s))$$

The first line ensures that the lineage set of a result tuple z is equal over all time points $t \in z.T$. The second line ensures that the time intervals are maximal, i.e., there is no tuple z' that is a direct predecessor or successor of z and has the same lineage set.

Example 3.5. Consider the lineage sets for the temporal aggregation query $Q_1'' = {}_D\vartheta^T_{CNT(*)}(\mathsf{p})$ from Example 3.3. The corresponding result relation in Fig. 3 is change preserving. For each result tuple $z_1, \ldots, z_4$, we have the same lineage set at each time point, and there are no directly preceding or succeeding tuples that have the same lineage set. If z_2'' were split into two tuples, the two tuples would have the same lineage set and hence violate the second condition in the above definition.

3.4. Extended Snapshot Reducibility

So far we have covered all temporal queries that can be formulated as "at each point in time" generalizations of standard relational algebra queries. However, some temporal queries cannot be formulated this way. Specifically, snapshot reducibility does not apply to temporal operators with predicates and functions that reference the timestamp intervals of the argument relations. This is because the intervals are removed by the timeslice operator. For instance, snapshot reducibility does not apply to a query that computes the average duration of projects at each point in time since the duration function refers to the original intervals of projects. *Extended snapshot reducibility* covers queries that reference the original timestamp intervals. This is supported by passing the original intervals as additional attributes to relational algebra operators.

[2]Originally the term *interval preservation* was used. We use the term *change preservation* instead, since all interval boundaries coincide with the start and end points of intervals where the lineage changes.

Definition 3.6. (*Extend Operator*) Let r with schema $(\mathbf{A}, T)$ be a temporal relation. The *extend operator*, $\epsilon_U(\mathsf{r})$, yields a temporal relation with schema $(\mathbf{A}, U, T)$ and is defined as follows:

$$z \in \epsilon_U(\mathsf{r}) \iff \exists r \in \mathsf{r}(z.\mathbf{A} = r.\mathbf{A} \wedge z.U = r.T \wedge z.T = r.T)$$

The extend operator adds an attribute to the schema of an argument relation. Assume a temporal relation r with schema $R = (A, T)$ containing one tuple $\mathsf{r} = \{(1, [\text{2014-01}, \text{2014-6}))\}$. The extend operator $\epsilon_U(\mathsf{r})$ produces the relation $\{(1, [\text{2014-01}, \text{2014-6}), [\text{2014-01}, \text{2014-6}))\}$ with schema (A, U, T).

The results of some algebra operators are affected by the introduction of additional attributes, while the results of other operators are not. The latter kind of operator is called *schema robust*.

Definition 3.7. (*Schema Robust Operator*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be relations with schema $R_i = (\mathbf{A}_i)$ and $\mathsf{r}'_1, \ldots \mathsf{r}'_n$ be relations with schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$ and $\mathsf{r}_i = \pi_{\mathbf{A}_i}(\mathsf{r}'_i)$. Furthermore, let ψ be an n-ary operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathsf{r}_1, \ldots, \mathsf{r}_n$. Operator ψ is *schema robust* iff for all $\mathbf{X}_i$ and $\mathsf{r}_1, \ldots, \mathsf{r}_n$, the following holds:

$$\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n) \equiv \pi_{\mathbf{E}}(\psi(\mathsf{r}'_1, \ldots, \mathsf{r}'_n))$$

The schema robust relational algebra operators are $\psi \in \{\pi, \vartheta, \sigma, \times, \bowtie, \ltimes, \rtimes, \ltimes, \rtimes, \triangleright\}$, whereas the set operations $\{-, \cap, \cup\}$ are not schema robust. The same holds for the corresponding temporal operators.

Example 3.8. Consider relations $\mathsf{r}(A) = \{(1)\}$, $\mathsf{s}(B) = \{(1)\}$ and relations $\mathsf{r}'(A, C) = \{(1, 2)\}$, $\mathsf{s}'(B, D) = \{(1, 3)\}$ that extend the former relations with an additional attribute. We have $\mathsf{r} \bowtie_{A=B} \mathsf{s} = \{(1, 1)\} = \pi_{A,B}(\mathsf{r}' \bowtie_{A=B} \mathsf{s}')$, but $\mathsf{r} - \mathsf{s} = \{\} \neq \pi_A(r' - s')$.

We exploit schema robustness in the temporal algebra to allow operators to access the original timestamps of tuples.

Definition 3.9. (*Extended Snapshot Reducibility*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, ψ^T be an n-ary schema robust temporal operator, and ψ be the corresponding n-ary nontemporal operator that yields a relation with schema $\mathbf{E}$. Let Ω^T be the time domain and τ be the timeslice operator. Operator ψ^T is *extended snapshot reducible* to ψ iff for all $t \in \Omega^T$:

$$\tau_t(\psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)) \equiv \pi_{\mathbf{E}}(\psi(\tau_t(\epsilon_{U_1}(\mathsf{r}_1)), \ldots, \tau_t(\epsilon_{U_n}(\mathsf{r}_n))))),$$

where, in predicates and functions on the right-hand side, references to $\mathsf{r}_i.T$ have been replaced by references to $\mathsf{r}_i.U_i$.

Note the difference to the definition of snapshot reducibility in Definition 3.1. The extend operator introduces a copy of the original timestamp, which is removed after executing the corresponding nontemporal operator. The essential property of extended snapshot reducibility is that it allows references to the original intervals, which is achieved by substituting them with references to additional attributes that have been propagated. Consider the query Q'''_1 = *"At each time point, what is the average duration of projects per department?"* that can be expressed as ${}_D\vartheta^T_{AVG(DUR(\mathsf{p}.T))}(\mathsf{p})$ in temporal relational algebra. The key point is that the aggregation operator (ϑ) is schema robust and thus extended snapshot reducibility can be used. For instance, for $t = \text{2014-04}$ we have:

$$\tau_{\text{2014-04}}({}_D\vartheta^T_{AVG(DUR(\mathsf{p}.T))}(\mathsf{p})) \equiv \pi_{D,AVG}({}_D\vartheta_{AVG(DUR(U_1))}(\tau_{\text{2014-04}}(\epsilon_{U_1}(\mathsf{p})))) = \{(\text{CS}, 4)\}$$

Corollary 3.10. *The temporal operators* $\{\pi^T, \vartheta^T, \sigma^T, \times^T, \bowtie^T, \rtimes^T, \ltimes^T, \rtimes^T, \triangleright^T\}$ *are extended snapshot reducible, whereas the set operations* $\{-^T, \cap^T, \cup^T\}$ *are not extended snapshot reducible.*

3.5. Scaling

When the timestamp intervals of tuples change, the values of some attributes of the tuples must also be changed [Böhlen et al. 2006a; Böhlen et al. 2006b]. For instance, tuple r_1 in Fig. 1 states that the external funding of project P_1 is 5K for five months. If the five-months interval is broken into a three-months interval and a two-months interval (e.g., as in Fig. 1), the original 5K budget must be split, too.

When splitting the allocation of values to different intervals is application dependent, for which reason we allow it to be handled by a generic scaling function. This function needs the original value, the original interval, and the new interval as arguments.

Definition 3.11 (*Scaling function*). Let x be an attribute value to be scaled and T_{NEW} and T_{OLD} be two interval timestamps such that $T_{NEW} \subseteq T_{OLD}$. Function *scale* is defined as follows:

$$scale(x, T_{NEW}, T_{OLD}) = x \cdot w(T_{NEW}, T_{OLD}), \text{ where } 0 < w(T_{NEW}, T_{OLD}) \leq 1$$

A scaling function uses the new interval timestamp T_{NEW} and the original timestamp T_{OLD} to define a weight $0 < w(T_{NEW}, T_{OLD}) \leq 1$ and scales x accordingly.

Scaling must be integrated into the temporal algebra, i.e., it must be part of a temporal operator, since a simple pre- or post-processing is not sufficient.

Lemma 3.12 (Scaling Required as Part of the Operators). *Scaling must be a parameter of temporal operators, since for some temporal operators (i.e., projection, aggregation, difference, intersection, and union), it cannot be performed in a pre- or post-processing step.*

Proof. Scaling cannot be performed as a pre- or post-processing step because the information required for the scaling is not available before or after the operation. Consider the aggregation query ${}_D\vartheta^T_{SUM(B)}(\mathsf{p})$, where the value of B must be scaled. Scaling requires three parameters: the original value to be scaled, the adjusted time interval, and the original time interval. Scaling cannot be performed in a pre-processing step, since relation p does not include the adjusted time interval, which is only determined as part of the aggregation. Similarly, it cannot be performed in a post-processing step since the original time intervals are no longer available after the operation. Note that an aggregation result tuple derives from more than one input tuple, and adding a single extra time interval attribute is not a viable approach. □

Example 3.13. Consider $Q_1 = {}_D\vartheta^{T:B@scaleU}_{SUM(B)}(\mathsf{p})$. The scaling must be performed after the adjustment of the intervals, but before the sum is computed. This is indicated in the algebra operation with the ${}^{B@scaleU}$-superscript, where *scaleU* is a scaling function and B is the attribute with the original amount that needs to be scaled. The details of the scaling function with the original and the adjusted interval as parameters are taken care of during the mapping of the temporal relational algebra to the relational algebra with interval adjustment (cf. Section 4).

4. TEMPORAL QUERY PROCESSING USING INTERVAL ADJUSTMENT

We proceed to present the systematic transformation of sequenced temporal algebra queries to queries expressed in the algebra of the underlying relational DBMS extended with only two operators for interval adjustment.

4.1. Solution Overview

The transformation replaces temporal operators with the corresponding nontemporal operators with the help of two operators for interval adjustment and timestamp propagation.

Specifically, a temporal query is processed in four steps, illustrated in Fig. 4 and described next.

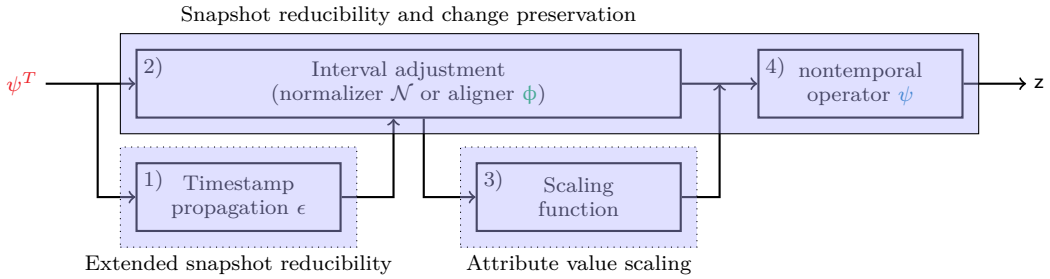


Fig. 4. Reduction of a temporal operator ψ^T to the corresponding nontemporal operator ψ using interval adjustment, timestamp propagation, and attribute value scaling.

(1) First, *timestamp propagation* replicates the original timestamps used in argument relations by introducing additional attributes. This step is only executed if the original timestamps are needed, either to scale attribute values in step 3 or to evaluate a predicate or a function that references the original timestamps in step 4 (cf. Section 4.4).
(2) Second, *interval adjustment* splits the timestamps of the input tuples such that they match the intervals to be used in the result tuples. All tuples that (in step 4) are processed together to produce a single result tuple now have identical timestamps. To achieve interval adjustment, we extend the DBMS kernel with two operators: a temporal normalizer, $\mathcal{N}$, used for so-called group based operators (π, ϑ, $-$, $\cap$, $\cup$), and a temporal aligner, Φ, used for so-called tuple based operators (σ, $\times$, $\bowtie$, $⟗$, $⟕$, $⟖$, $\triangleright$) (cf. Section 4.2).
(3) Third, *attribute value scaling* scales attribute values to the new, adjusted timestamps. For scaling, the DBMS needs the original and new timestamps in addition to the original value of the attribute to be scaled. Similar to timestamp propagation, this step is optional and is only executed if attribute values need to be scaled. As part of this step, propagated timestamps are removed if they are no longer needed by subsequent operators or if the subsequent operator is not schema robust (cf. Section 4.5).
(4) Finally, a *reduction* is applied such that the corresponding nontemporal operators are evaluated over the intermediate relations produced in the previous three steps. An additional equality constraint is imposed over the adjusted timestamps (e.g., as a grouping attribute for aggregation or an equality predicate in joins). This constraint guarantees that all tuples that produce a single result tuple are processed together. For each temporal operator we provide a reduction rule that introduces the corresponding nontemporal operator (cf. Section 4.3).

The timestamp adjustment (step 2) and the evaluation of the corresponding nontemporal operator (step 4) together guarantee snapshot reducibility and change preservation. In addition, we can propagate time intervals (step 1), thereby enabling extended snapshot reducibility (in step 4) and attribute value scaling (in step 3).

Example 4.1. Fig. 5 exemplifies our approach using the temporal aggregation query Q_1' = "*At each time point, what is the number, average duration and available budget of projects per department?*", which is expressed in temporal algebra as follows:

$$Q_1' = {}_D\vartheta^{T:B@scaleU}_{CNT(*),AVG(\mathsf{p}.T),SUM(B)}(\mathsf{p})$$

The left-hand query tree in Fig. 5 shows the temporal operator, and the right-hand query tree is the result of the transformation. The transformation proceeds as follows: (1) apply

timestamp propagation (ϵ) to enable the scaling of attribute B and the evaluation of the AVG aggregate; (2) adjust relation p using the temporal normalizer ($\mathcal{N}$); (3) scale attribute B and remove the propagated timestamp U; (4) introduce a nontemporal aggregation, where T is added as additional grouping attribute ($_T\vartheta$).

For clarity, the example uses two propagated timestamps, U for scaling and V for extended snapshot reducibility; however one propagated timestamp, e.g., U, that is not removed directly after scaling is sufficient.

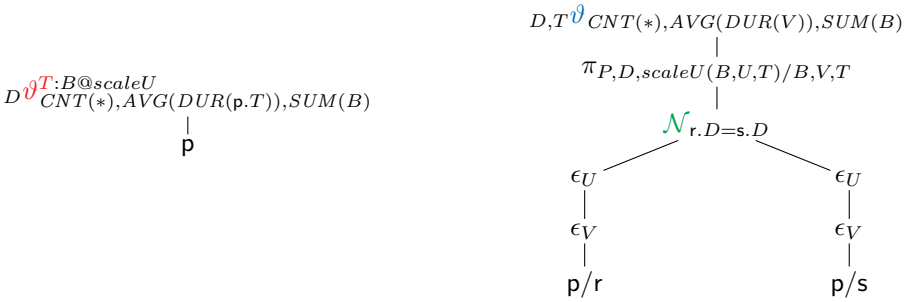


Fig. 5. Reduction of query Q_1'.

The proposed transformation requires minimal extensions of an existing DBMS and leverages built-in indexing and optimization techniques. The key extension of the DBMS is the integration of the normalizer $\mathcal{N}$ and aligner Φ operators into the DBMS kernel. Timestamp propagation (ϵ) and attribute value scaling can be achieved by means of generalized projections and user defined functions, respectively.

4.2. Interval Adjustment Operators

4.2.1. Group and Tuple based Operators. We distinguish between two classes of (temporal) operators, for which different adjustment operators are needed.

Definition 4.2 (*Group and Tuple Based Operators*). Let ψ be an n-ary operator with argument relations $\mathsf{r}_1, \ldots, \mathsf{r}_n$. Operator $\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)$ is

— *group based* iff more than one tuple of an argument relation r_i can contribute to a result tuple z, i.e., the lineage set of z can contain more than one witness list.
— *tuple based* iff at most one input tuple of each argument relation r_i can contribute to a result tuple z, i.e., the lineage set of z contains at most one witness list.

Table II classifies operators according to Definition 4.2. For instance, a full outer join is tuple based since for each result tuple $z \in \mathsf{r} \;⟗_\theta\; \mathsf{s}$, at most one tuple from r and one tuple from s contributes to z. In contrast, aggregation ϑ is a group based operator since more than one input tuple can (and typically does) contribute to a result tuple.

Table II. Tuple and group based operators.

	Operators
Group Based	π, ϑ, $-$, $\cap$, $\cup$
Tuple Based	σ, $\times$, $\bowtie$, $⟕$, $⟖$, $⟗$, $\triangleright$

For each operator class, we design an operator that allows to use equality predicates on the adjusted timestamps and ensures change preservation of the subsequent nontemporal operation. More specifically, we introduce

— a *temporal normalizer* for group-based operators and
— a *temporal aligner* for tuple-based operators.

4.2.2. Temporal Normalization. For the group based operators, we use a temporal normalization operator that adjusts the time interval of an argument tuple by splitting it at each start and end point of all tuples that are in the same group. The group of argument tuples is determined by a predicate θ. Condition θ is an equality condition on the projected attributes for a projection and an equality condition on the nontemporal attributes for set operations.

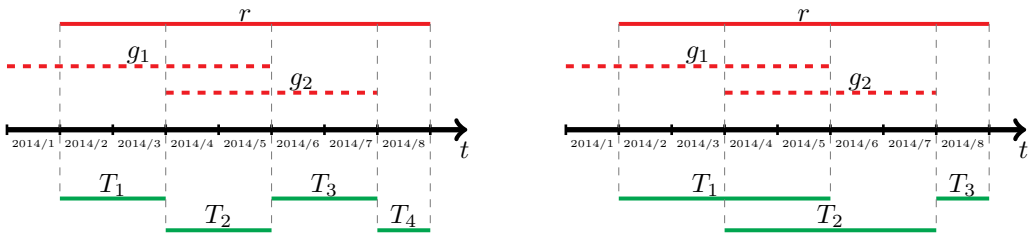
Definition 4.3. (*Temporal Normalization*) Let r and s be two temporal relations. The *normalization*, $\mathcal{N}_\theta(\mathsf{r},\mathsf{s})$, of r with respect to s and a predicate θ over attributes of r and s is defined as follows:

$$\tilde{r} \in \mathcal{N}_\theta(\mathsf{r},\mathsf{s}) \iff \exists r \in \mathsf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \tilde{r}.T \in \mathit{normalize}(r, \{s \in \mathsf{s} \mid \theta(r,s)\})),$$

where

$$\begin{aligned} &T \in \mathit{normalize}(r,\mathsf{g}) \iff \\ &\quad T \subseteq r.T \wedge \forall g \in \mathsf{g}\ (g.T \cap T = \emptyset \vee T \subseteq g.T) \wedge && (1)\\ &\quad \forall T' \supset T\ (\exists g \in \mathsf{g}(T' \cap g.T \neq \emptyset \wedge T' \not\subseteq g.T) \vee T' \not\subseteq r.T). && (2) \end{aligned}$$

The *normalize* function adjusts the intervals of the argument tuples by splitting them into sub-intervals. Condition (1) requires that an adjusted interval T is contained in r's timestamp and is either contained in or is disjoint from all intervals of tuples $g \in \mathsf{g}$. Condition (2) requires that T is maximal, i.e., it cannot be enlarged without violating the first condition. Fig. 6(a) illustrates $\mathit{normalize}(r, \{g_1, g_2\})$. The timestamp of tuple r is split at all start and end points of g_1 and g_2 that are covered by $r.T$. This yields four intervals. The temporal normalization produces tuples over the intervals produced by the *normalize* function, where the nontemporal attribute values are copied from r.



(a) $\mathit{normalize}(r, \{g_1, g_2\})$ (b) $\mathit{align}(r, \{g_1, g_2\})$

Fig. 6. Interval adjustment with temporal normalization and alignment.

Example 4.4. Fig. 7 illustrates temporal normalization $\mathcal{N}_{\mathsf{r}.D=\mathsf{s}.D}(\mathsf{p}/\mathsf{r}, \mathsf{p}/\mathsf{s})$ for our example relation p, where the grouping is on the department attribute D. For instance, tuples $\tilde{r}_{11}$ and $\tilde{r}_{12}$ are derived from argument tuple r_1, which is adjusted with respect to s_1 and s_2, whereas $\tilde{r}_{21}$ and $\tilde{r}_{22}$ are derived from r_2, which is adjusted with respect to s_1 and s_2.

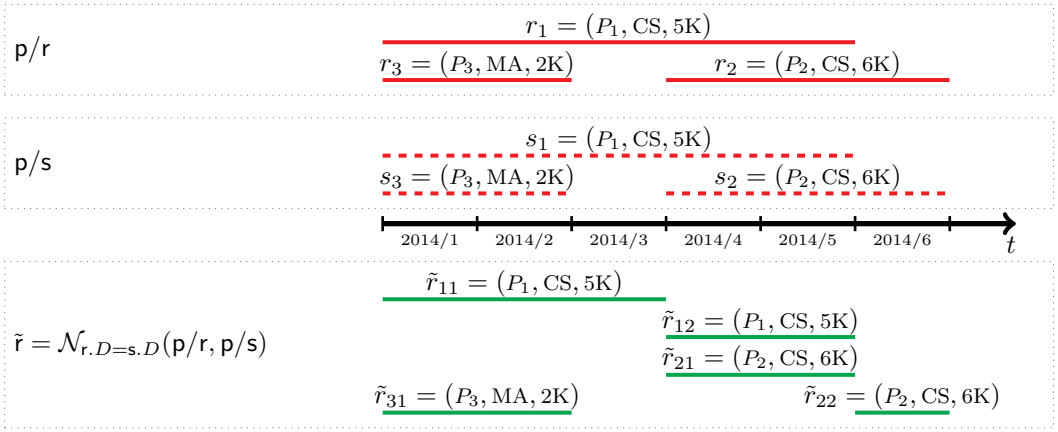


Fig. 7. Temporal normalization.

COROLLARY 4.5. *Assume a temporal relation* r *and the temporal normalization* $\tilde{\mathsf{r}} = \mathcal{N}_{\mathsf{r}.\mathbf{B}=\mathsf{s}.\mathbf{B}}(\mathsf{r}, \mathsf{r/s})$. *All tuples* $\tilde{r} \in \tilde{\mathsf{r}}$ *with the same* $\mathbf{B}$*-values have interval timestamps that are either equal or disjoint.*

COROLLARY 4.6. *Assume temporal relations* r *and* s. *Every tuple* $\tilde{r} \in \mathcal{N}_\theta(\mathsf{r}, \mathsf{s})$ *is derived from a tuple* $r \in \mathsf{r}$, *and the timestamp* $\tilde{r}.T$ *is either the intersection of* $r.T$ *with the timestamps of all tuples in* s *that share time points and satisfy* θ, *or it is a maximal sub-interval of* $r.T$ *that is not covered by any tuple in* s *that satisfies* θ.

LEMMA 4.7. *Let* r *be a temporal relation with* $|\mathsf{r}| = n$, s *be a temporal relation with* $|\mathsf{s}| = m$, *and* $\tilde{\mathsf{r}} = \mathcal{N}_\theta(\mathsf{r}, \mathsf{s})$ *be the result of temporal normalization with condition* θ. *The cardinality of the normalized relation* $\tilde{\mathsf{r}}$ *is bounded by* $n \leq |\tilde{\mathsf{r}}| \leq 2nm + n$.

PROOF. The lower bound n occurs when no matching tuple in s exists with start or end point within the interval of any tuple in r. In this case, $\tilde{\mathsf{r}} = \mathsf{r}$ since no tuple in r is split. The upper bound occurs when all start and end points of all tuples in s are split points of all tuples in r. Each of the m tuples in s can introduce at most two split points for each tuple in r, which gives $2m + 1$ normalized tuples for a single r-tuple. With n tuples in r, we get the upper bound. □

4.2.3. Temporal Alignment. For tuple based operators, we design a *temporal alignment* operator that adjusts an argument tuple according to each individual tuple of a group.

Definition 4.8. (*Temporal Alignment*) Let r and s be temporal relations and θ be a predicate over the nontemporal attributes of a tuple in r and a tuple in s. The *temporal alignment* operator, $\Phi_\theta(\mathsf{r}, \mathsf{s})$, of r with respect to s and condition θ is defined as follows:

$$\tilde{r} \in \Phi_\theta(\mathsf{r}, \mathsf{s}) \iff \exists r \in \mathsf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \tilde{r}.T \in \mathit{align}(r, \{s \in \mathsf{s} \mid \theta(r, s)\})),$$

where

$$\begin{aligned} &T \in \mathit{align}(r, \mathsf{g}) \iff \\ &\quad \exists g \in \mathsf{g}(T = r.T \cap g.T) \wedge T \neq \emptyset \vee && (1) \\ &\quad T \subseteq r.T \wedge \nexists g \in \mathsf{g}(g.T \cap T \neq \emptyset) \wedge \forall T' \supset T(\exists g \in \mathsf{g}(T' \cap g.T \neq \emptyset) \vee T' \not\subseteq r.T) && (2) \end{aligned}$$

Function $\mathit{align}(r, \mathsf{g})$ splits the interval timestamp of tuple r with respect to each individual tuple in g. Condition (1) handles all possible sub-intervals of $r.T$ for which a interval

timestamp in g exists, and the result timestamp T is their intersection. Condition (2) handles sub-intervals for which no covering interval in g exists, and the result timestamp T is a maximal non-covered part of $r.T$. Fig. 6(b) illustrates $align(r, \{g_1, g_2\})$. For instance, intervals T_1 and T_2 are derived from the intersections of r with g_1 and g_2, respectively. Interval T_3 is a sub-interval of $r.T$ that is not covered by any tuple in g.

Example 4.9. In order to illustrate temporal alignment, we additionally use the manager relation m in Fig. 8.

m

	M	D	T
s_1	Ann	CS	[2014-1, 2014-4)
s_2	Sam	MA	[2014-1, 2014-5)
s_3	Joe	CS	[2014-4, 2014-7)

Fig. 8. Project database with manager relation m.

Consider query Q_2 = *"At each time point, which projects is a manager responsible for and what is the available budget?"* Query Q_2 is a temporal natural left outer join between the manager relation m and the project relation p. The query requires a left outer join to also report periods when a manager was not responsible for a project. The project budget needs to be scaled since a manager might not be responsible for a project for its entire life time. The corresponding temporal relational algebra expression is:

$$Q_2 = \mathsf{m} \; ⟕^{T:B@scaleU} \; \mathsf{p}$$

Fig. 9 shows the aligned relations $\tilde{\mathsf{m}} = \Phi_{\mathsf{m}.D=\mathsf{p}.D}(\mathsf{m}, \mathsf{p})$ and $\tilde{\mathsf{p}} = \Phi_{\mathsf{m}.D=\mathsf{p}.D}(\mathsf{p}, \mathsf{m})$. For instance, tuples $\tilde{m}_{21}$ and $\tilde{m}_{22}$ are derived from tuple m_2, which overlaps with p_3 (producing $\tilde{m}_{21}$) and has a sub-interval that is not covered by any matching tuple in p (producing $\tilde{m}_{22}$). Tuple m_3 also produces two tuples, namely $\tilde{m}_{31}$, from the intersection with p_1, and $\tilde{m}_{32}$ from the intersection with p_2.

The essential property of the aligned relations $\tilde{\mathsf{m}}$ and $\tilde{\mathsf{p}}$ is that tuples that later must be joined have pairwise identical interval timestamps (or, in case of a left outer join, that no qualifying tuple from relation p overlaps with the tuple from $\tilde{\mathsf{m}}$). In our example, the tuple pairs that join and, either have identical timestamps or the second tuple is missing, are $(\tilde{m}_{11}, \tilde{p}_{11}), (\tilde{m}_{21}, \tilde{p}_{31}), (\tilde{m}_{31}, \tilde{p}_{12}), (\tilde{m}_{32}, \tilde{p}_{21})$, and $(\tilde{m}_{22}, -)$.

Corollary 4.10. *Given temporal relations* r *and* s *with alignments* $\tilde{\mathsf{r}} = \Phi_\theta(\mathsf{r}, \mathsf{s})$ *and* $\tilde{\mathsf{s}} = \Phi_\theta(\mathsf{s}, \mathsf{r})$*, for any pair of tuples* $r \in \mathsf{r}$ *and* $s \in \mathsf{s}$ *that satisfy* θ *and for which* $r.T \cap s.T \neq \emptyset$*, two tuples* $\tilde{r} \in \tilde{\mathsf{r}}$ *and* $\tilde{s} \in \tilde{\mathsf{s}}$ *exist that have matching nontemporal values for, respectively,* r *and* s *and that have the identical timestamp* $\tilde{r}.T = \tilde{s}.T = r.T \cap s.T$*.*

Corollary 4.11. *Given temporal relations* r *and* s*, every tuple* $\tilde{r} \in \Phi_\theta(\mathsf{r}, \mathsf{s})$ *is derived from a tuple* $r \in \mathsf{r}$*, and the timestamp of* $\tilde{r}$ *is either the intersection of* $r.T$ *with the timestamp of a tuple in* s *that satisfies* θ*, or it is a maximal sub-interval of* $r.T$ *that is not covered by the timestamp of any tuple in* s *satisfying* θ*.*

Lemma 4.12. *Let* r *be a temporal relation with* $|\mathsf{r}| = n$*,* s *be a temporal relation with* $|\mathsf{s}| = m$*, and* $\tilde{\mathsf{r}} = \Phi_\theta(\mathsf{r}, \mathsf{s})$ *be the result of temporal alignment with condition* θ*. The cardinality of the aligned relation is bounded by* $n \leq |\tilde{\mathsf{r}}| \leq 2nm + n$*.*

Proof. Recall from Corollary 4.11 that a tuple $\tilde{r} \in \tilde{\mathsf{r}}$ is either produced from an intersection with a matching tuple in s or by a maximal subinterval that is not covered by any matching tuple in s. The lower bound occurs when no tuple in r is split, and this is the case when no tuple in s overlaps any tuple in r. To determine the upper bound, note that the m tuples in s can have at most m intersections with a single tuple $r \in \mathsf{r}$ and at most $m + 1$

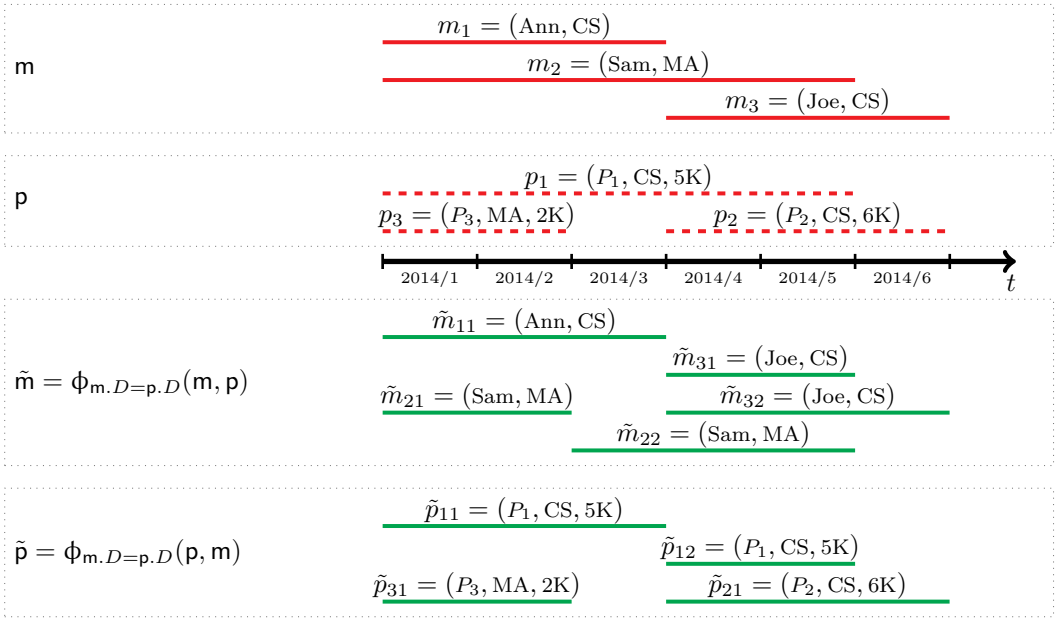


Fig. 9. Temporal alignment.

maximal uncovered subintervals. Hence, one tuple in r can be split into at most $2m + 1$ tuples. With n tuples in r, this yields the upper bound. □

4.2.4. New Algebraic Equivalences. The normalization and alignment operators interact with the other operators of the relational algebra. Here, we provide algebraic equivalences that involve these two operators.

LEMMA 4.13. *Let* r *and* s *be temporal relations with schemas* $(\mathbf{A}, T)$ *and* $(\mathbf{C}, T)$*, respectively;* $\phi_\theta(\mathsf{r}, \mathsf{s})$ *denote the temporal alignment of* r *and* s*;* $\mathcal{N}_\theta(\mathsf{r}, \mathsf{s})$ *denote the temporal normalization of* r *and* s*;* θ*,* θ_1*, and* θ_2 *be conditions; and* $attr(\theta)$ *be the attributes that occur in* θ*. The following equivalences hold.*

$$\sigma_{\theta_1 \wedge \theta_2}(\mathcal{N}_\theta(\mathsf{r}, \mathsf{s})) \equiv \sigma_{\theta_1}(\mathcal{N}_\theta(\sigma_{\theta_2}(\mathsf{r}), \mathsf{s})) \qquad T \notin attr(\theta_2) \qquad (E1)$$
$$\sigma_{\theta_1 \wedge \theta_2}(\phi_\theta(\mathsf{r}, \mathsf{s})) \equiv \sigma_{\theta_1}(\phi_\theta(\sigma_{\theta_2}(\mathsf{r}), \mathsf{s})) \qquad T \notin attr(\theta_2) \qquad (E2)$$
$$\pi_{\mathbf{B}}(\mathcal{N}_\theta(\mathsf{r}, \mathsf{s})) \equiv \pi_{\mathbf{B}}(\mathcal{N}_\theta(\pi_{\mathbf{B} \cup attr(\theta) \cup T}(\mathsf{r}), \mathsf{s})) \qquad \mathbf{B} \subseteq \mathbf{A} \cup T \qquad (E3)$$
$$\pi_{\mathbf{B}}(\phi_\theta(\mathsf{r}, \mathsf{s})) \equiv \pi_{\mathbf{B}}(\phi_\theta(\pi_{\mathbf{B} \cup attr(\theta) \cup T}(\mathsf{r}), \mathsf{s})) \qquad \mathbf{B} \subseteq \mathbf{A} \cup T \qquad (E4)$$
$$\mathcal{N}_\theta(\mathsf{r}, \mathsf{s}) \equiv \mathcal{N}_\theta(\mathsf{r}, \pi_{attr(\theta) \cup T}(\mathsf{s})) \qquad (E5)$$
$$\phi_\theta(\mathsf{r}, \mathsf{s}) \equiv \phi_\theta(\mathsf{r}, \pi_{attr(\theta) \cup T}(\mathsf{s})) \qquad (E6)$$
$$\mathcal{N}_\theta(\sigma_{\theta_1}(\mathsf{r}), \mathsf{s}) \equiv \mathcal{N}_\theta(\sigma_{\theta_1}(\mathsf{r}), \sigma_{\theta_2}(\mathsf{s})) \qquad \theta \wedge \theta_1 \Rightarrow \theta_2 \qquad (E7)$$
$$\phi_\theta(\sigma_{\theta_1}(\mathsf{r}), \mathsf{s}) \equiv \phi_\theta(\sigma_{\theta_1}(\mathsf{r}), \sigma_{\theta_2}(\mathsf{s})) \qquad \theta \wedge \theta_1 \Rightarrow \theta_2 \qquad (E8)$$

See the electronic Appendix B.1 for the proof of Lemma 4.13.

Equivalences E1 and E2 push a selection inside a normalization or alignment. Only the part of the selection that does not involve the timestamp attribute T can be pushed inside. The selection predicate θ_1 that involves T must be evaluated after a tuple has been adjusted since the adjustment might change the timestamp. Similarly, equivalences E3 and E4 show

how projection commutes with normalization and alignment. In order to reduce the size of the adjusted tuples in intermediate relations, we can project relation r to the projection attributes $\mathbf{B}$ and the attributes that are used in the alignment condition θ. Equivalences E5 and E6 show how a projection can be applied to relation s to retain only those attributes that are needed for the normalization or alignment, i.e., the attributes needed for evaluating condition θ and the interval timestamp T. Finally, equivalences E7 and E8 make it possible to reduce the cardinality of the second argument relation of the normalization and alignment operators by eliminating tuples that cannot contribute to the result. This is achieved by applying a selection to s that keeps only those tuples that can match a tuple in r and therefore can affect the alignment process. For instance, consider $\theta \equiv (\mathsf{r}.D = \mathsf{s}.D)$ for the normalization and a selection with $\theta_1 \equiv (\mathsf{r}.D = \text{'CS'})$ on relation r, i.e., only projects in the CS department are considered. By applying E7, we can push a selection with $\theta_2 \equiv (\mathsf{s}.D = \text{'CS'})$ inside the normalization.

4.3. Reduction of Temporal Operators to Nontemporal Operators with Interval Adjustment

The basic scheme for replacing temporal operators with sequenced semantics with the corresponding nontemporal operators is illustrated in Fig. 10. To keep the presentation simple, timestamp propagation and scaling are not included explicitly. Instead, we assume that all references to original timestamps have been substituted with references to the copy of the timestamps. Timestamp propagation and scaling are discussed in detail in subsequent sections.

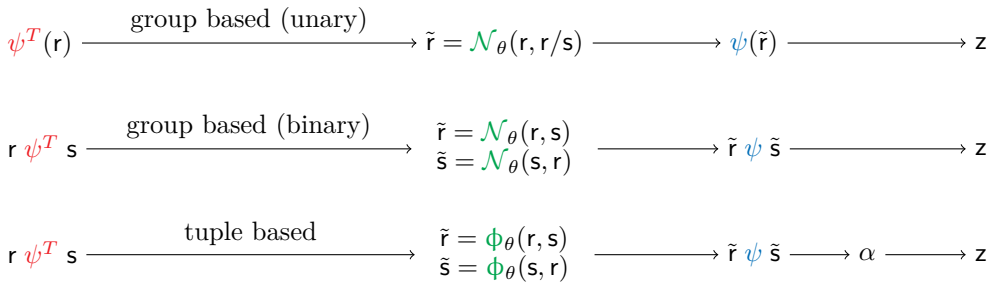


Fig. 10. Reduction of temporal operators.

Before giving the reduction rules, we need a final operator that eliminates temporal duplicates. The alignment operator produces all distinct intersections of matching tuples. Since the timestamps are adjusted independently for each tuple, the result might include intervals that are not maximal intersections of two tuples, as exemplified next.

Example 4.14. Consider the Cartesian product of relations $\mathsf{r} = \{(a, [1, 9)), (b, [3, 7))\}$ and $\mathsf{s} = \{(c, [1, 9)), (d, [3, 7))\}$. Temporal alignment yields $\tilde{\mathsf{r}} = \Phi_{true}(\mathsf{r}, \mathsf{s}) = \{(a, [1, 9)), (a, [3, 7)), (b, [3, 7))\}$ and $\tilde{\mathsf{s}} = \Phi_{true}(\mathsf{s}, \mathsf{r}) = \{(c, [1, 9)), (c, [3, 7)), (d, [3, 7))\}$. The subsequent equality join of $\tilde{\mathsf{r}}$ and $\tilde{\mathsf{s}}$ on the adjusted timestamp attributes yields:

z_1	a	c	$[1, 9)$
z_2	a	c	$[3, 7)$
z_3	a	d	$[3, 7)$
z_4	b	c	$[3, 7)$
z_5	b	d	$[3, 7)$

Tuple z_2 is produced by joining $\tilde{r}_2 = (a, [3, 7))$ and $\tilde{s}_2 = (c, [3, 7))$, and it is a temporal duplicate of z_1. Note that neither $\tilde{r}_2$ nor $\tilde{s}_2$ can be removed before the join, since these

tuples are required to produce z_3 and z_4, respectively. Instead, we apply an absorb operator to remove temporal duplicates in a post-processing step.

Definition 4.15. (*Absorb Operator*) Let r be a temporal relation. The *absorb* operator, α, eliminates all tuples $r \in \mathsf{r}$ for which another value-equivalent tuple $r' \in \mathsf{r}$ exists with $r.T \subset r'.T$:

$$\alpha(\mathsf{r}) = \{r \in \mathsf{r} \mid \nexists r' \in \mathsf{r}\ (r.\mathbf{A} = r'.\mathbf{A} \wedge r.T \subset r'.T)\}$$

The following theorem defines the reduction rules for a temporal algebra with sequenced semantics.

THEOREM 4.16. *Let* r *and* s *be temporal relations,* θ *be a predicate,* F *be a set of aggregation functions over* $\mathsf{r}.\mathbf{A}$*,* $\mathbf{B} \subseteq \mathbf{A}$ *be a set of attributes, and* α *be the absorb operator. The temporal algebra reduction rules in Table III satisfy the sequenced semantics*[3]*.*

Table III. Reduction rules.

Operator			**Reduction**
Selection	$\sigma^T_\theta(\mathsf{r})$	$=$	$\sigma_\theta(\mathsf{r})$
Projection	$\pi^T_{\mathbf{B}}(\mathsf{r})$	$=$	$\pi_{\mathbf{B},T}(\mathcal{N}_{\mathsf{r}.\mathbf{B}=\mathsf{s}.\mathbf{B}}(\mathsf{r}/\mathsf{r},\mathsf{r}/\mathsf{s}))$
Aggregation	${}_{\mathbf{B}}\vartheta^T_F(\mathsf{r})$	$=$	${}_{\mathbf{B},T}\vartheta_F(\mathcal{N}_{\mathsf{r}.\mathbf{B}=\mathsf{s}.\mathbf{B}}(\mathsf{r}/\mathsf{r},\mathsf{r}/\mathsf{s}))$
Difference	$\mathsf{r} -^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{r},\mathsf{s}) - \mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{s},\mathsf{r})$
Union	$\mathsf{r} \cup^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{r},\mathsf{s}) \cup \mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{s},\mathsf{r})$
Intersection	$\mathsf{r} \cap^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{r},\mathsf{s}) \cap \mathcal{N}_{\mathsf{r}.\mathbf{A}=\mathsf{s}.\mathbf{A}}(\mathsf{s},\mathsf{r})$
Cartesian Product	$\mathsf{r} \times^T \mathsf{s}$	$=$	$\alpha(\phi_{true}(\mathsf{r},\mathsf{s}) \bowtie_{\mathsf{r}.T=\mathsf{s}.T} \phi_{true}(\mathsf{s},\mathsf{r}))$
Inner Join	$\mathsf{r} \bowtie^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) \bowtie_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Left Outer Join	$\mathsf{r} ⟕^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) ⟕_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Right Outer Join	$\mathsf{r} ⟖^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) ⟖_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Full Outer Join	$\mathsf{r} ⟗^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) ⟗_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Anti Join	$\mathsf{r} \triangleright^T_\theta \mathsf{s}$	$=$	$\phi_\theta(\mathsf{r},\mathsf{s}) \triangleright_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r})$

PROOF. The proof is given in the electronic Appendix B.2. □

In terms of implementation, the absorb operator can be expressed as a selection that checks whether a result tuple was created from a maximum intersection of the original intervals; hence, no new operator needs to be implemented.

PROPOSITION 4.17 (ABSORB OPERATOR AS A SELECTION). *Let* $\psi \in \{\times, \bowtie, ⟕, ⟖, ⟗\}$*. The absorb operator* α *in Table III can be expressed as a selection followed by a projection as follows:*

$$\alpha(\phi_\theta(\mathsf{r},\mathsf{s})\ \psi_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T}\ \phi_\theta(\mathsf{s},\mathsf{r})) \equiv \pi_X(\sigma_\Theta(\phi_\theta(\epsilon_U(\mathsf{r}),\mathsf{s})\ \psi_\theta\ \phi_\theta(\epsilon_V(\mathsf{s}),\mathsf{r}))),$$

where $\Theta = (T_s = U_s \vee T_s = V_s) \wedge (T_e = U_e \vee T_e = V_e) \vee U_s = \omega \vee V_s = \omega$ *and* $X = sch(\phi_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T}(\mathsf{r},\mathsf{s})\ \psi_\theta\ \phi_\theta(\mathsf{s},\mathsf{r}))$*.*

[3]To shorten expressions, we assume that the name (or alias) of the relation resulting from an alignment operation is the the same as for the first input relation, i.e., r for $\phi_\theta(\mathsf{r},\mathsf{s})$. Otherwise renaming, such as $\phi_\theta(\mathsf{r},\mathsf{s})/\mathsf{r}$, has to be used.

PROOF. The proof is given in the electronic Appendix B.3 □

All operators that call for use of the absorb operator are schema robust, meaning that absorb can always be expressed as a selection according to Proposition 4.17.

When applying the transformation rules to a temporal relational algebra expression, the number of (nontemporal) algebra operator occurrences in the resulting expression is exponential in the number of operator occurrences in the temporal algebra expression. This is due to the fact that each reduction rule doubles the number of arguments.

LEMMA 4.18. *Given a temporal relational algebra expression with n temporal operators (except temporal selections), the size of the transformed algebra expression is $\mathcal{O}(2^n)$.*

PROOF. We show that n temporal operators in a relational algebra expression can be transformed into at most $2^n - 1$ nontemporal operators. In the proof, we only count the nontemporal operators. The total size of the transformed expression is within a constant factor c. For instance, according to Table III for a temporal left outer join we have one nontemporal operator, one absorb operator, two adjustment operators, and four relations. We get $\mathcal{O}(c \cdot (2^n - 1)) = \mathcal{O}(2^n)$. We do the proof by induction.
Base case ($n = 1$): 1 temporal operator transforms into $2^1 - 1 = 1$ nontemporal operator. This is evident from Table III, where each reduction rule transforms one temporal operator (red) into exactly one nontemporal operator (blue).
Inductive step ($n + 1$): An additional temporal operator transforms into one additional nontemporal operator that uses the transformations of the previous n temporal operators as input twice, yielding $2 \cdot (2^n - 1) + 1 = 2 \cdot 2^n - 2 + 1 = 2^{n+1} - 1$. □

The following example illustrates the exponential growth of the transformed expression and how it can be reduced to linear complexity with the help of common table expressions (CTE[4]).

Example 4.19. Consider the expression $((\mathsf{r} \mathbin{⟗}^T_{\theta_1} \mathsf{s}) \mathbin{⟗}^T_{\theta_2} \mathsf{t}) \mathbin{⟗}^T_{\theta_3} \mathsf{u}$. The expression and its reduction using the rules in Table III are shown as a tree structure in Fig. 11. The exponential size of the reduced expression is evident.

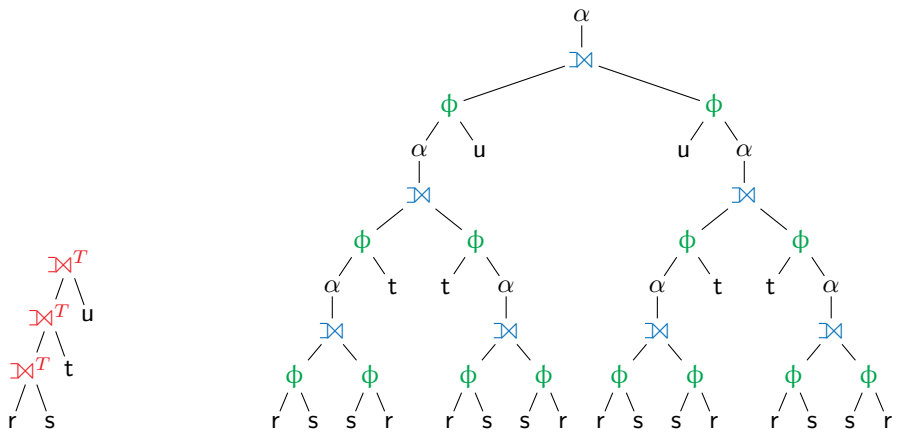


Fig. 11. Temporal expression $((\mathsf{r} \mathbin{⟗}^T_{\theta_1} \mathsf{s}) \mathbin{⟗}^T_{\theta_2} \mathsf{t}) \mathbin{⟗}^T_{\theta_3} \mathsf{u}$ and its transformed expression tree.

An equivalent expression using common table expressions, where the final result is in v_3, is shown in Fig. 12. Instead of repeating the sub-expression that involves relations r and s,

[4]https://www.postgresql.org/docs/current/static/queries-with.html

a CTE v_1 is introduced and used for the next sub-expression that in turn introduces the CTE v_2, and so on.

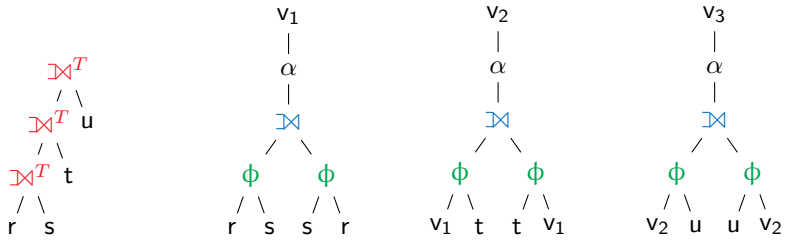


Fig. 12. Temporal expression $((\mathsf{r} \bowtie^T_{\theta_1} \mathsf{s}) \bowtie^T_{\theta_2} \mathsf{t}) \bowtie^T_{\theta_3} \mathsf{u}$ and its optimized, reduced expression tree.

To avoid long transformed expressions that may be difficult to optimize and potentially expensive to compute, we replace reoccurring sub-expressions with CTEs. The effect is that the exponential growth is reduced to a linear growth.

LEMMA 4.20 (REDUCTION OF QUERY EXPONENTIALLY). *Given a temporal relational algebra expression with n temporal operators, the size of the transformed algebra expression can be reduced to $\mathcal{O}(n)$ by using CTEs.*

PROOF. Consider a temporal relational algebra expression with n temporal operators, that are transformed in depth-first order. After each applied reduction rule a CTE for it is created and used as the input (instead of its defining expression) for the subsequent temporal operator. Thus, whenever a temporal operator is transformed, its inputs are either temporal relations or CTEs and its transformed expression contains exactly one nontemporal operator. Similar to the proof of Lemma 4.18 we only count the number of nontemporal operators, since the size of the transformed expression is within a constant factor c. Hence, given n temporal operators, the transformation results in n CTEs and $\mathcal{O}(c \cdot n) = \mathcal{O}(n)$. □

4.4. Timestamp Propagation

Timestamp propagation makes a copy of the original interval timestamps of tuples available to subsequent operations, which is needed in two cases: when the query references the original timestamps, as in the case of extended snapshot reducibility (cf. Section 3.4), and when attribute value scaling is needed (cf. Section 3.5). In both cases, the original timestamps are needed after temporal adjustment has been applied.

Example 4.21. Fig. 13 illustrates the reduction of the temporal aggregation query $Q_1''' =$ *"At each time point, what is the average duration of projects per department?"* Since the duration function refers to the original timestamps, the query is governed by extended snapshot reducibility. We first propagate the timestamps of p by means of $\epsilon_U(\mathsf{p})$. Next, references to T are substituted with references to U, yielding $Q_1''' = {}_D\vartheta^T_{AVG(DUR(U))}(\epsilon_U(\mathsf{p}))$. For the reduction, we apply the temporal normalizer to $\epsilon_U(\mathsf{p})$ (cf. Table III) to get groups of tuples with timestamps that are identical or disjoint. The normalized result is identical to the one shown in Fig. 7, except that the original intervals are present as well. Finally, we execute the reduced query to get the result.

We proceed to explain how timestamp propagation interacts with the reduction rules in Table III, showing that the propagated timestamps are not affected by the adjustment and that the adjustment is not affected by propagated timestamps.

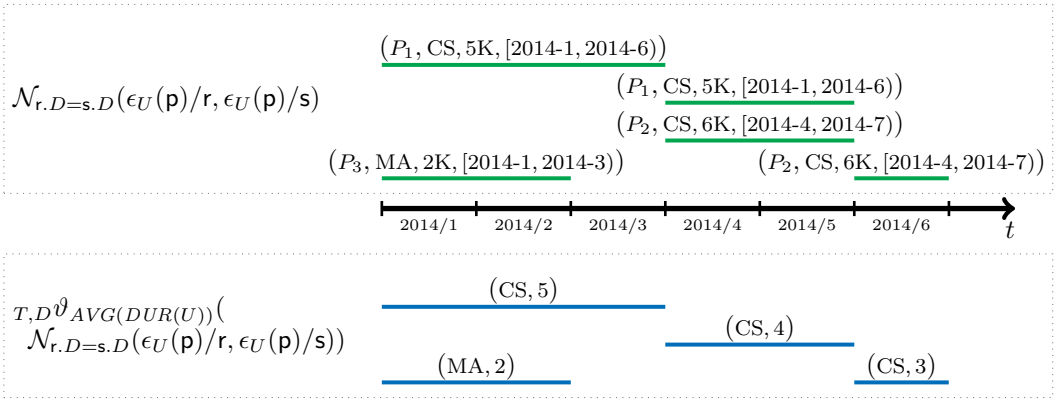


Fig. 13. Reduction of Query Q_1'''.

LEMMA 4.22. *Apart from the added propagated timestamp attribute, propagated timestamps do not change the result of the temporal adjustment, i.e., temporal adjustment is schema robust. For a temporal relation* r *with schema* $(\mathbf{E}, T)$ *and* $U \notin attr(\theta)$*, the following hold:*

$$\mathcal{N}_\theta(\mathsf{r}, \mathsf{s}) \equiv \pi_{\mathbf{E},T}(\mathcal{N}_\theta(\epsilon_U(\mathsf{r}), \mathsf{s}))$$
$$\mathcal{N}_\theta(\mathsf{r}, \mathsf{s}) \equiv \mathcal{N}_\theta(\mathsf{r}, \epsilon_U(\mathsf{s}))$$
$$\phi_\theta(\mathsf{r}, \mathsf{s}) \equiv \pi_{\mathbf{E},T}(\phi_\theta(\epsilon_U(\mathsf{r}), \mathsf{s}))$$
$$\phi_\theta(\mathsf{r}, \mathsf{s}) \equiv \phi_\theta(\mathsf{r}, \epsilon_U(\mathsf{s}))$$

PROOF. The proof is given in the electronic Appendix B.4. □

LEMMA 4.23. *Temporal adjustment does not change the values of the propagated timestamp attributes. Given a temporal relation* r *with schema* $(\mathbf{E}, T)$ *and* $U \notin attr(\theta)$*, the following hold:*

$$\pi_{\mathbf{E},U/T}(\mathcal{N}_\theta(\epsilon_U(\mathsf{r}), \mathsf{s}) \equiv \mathsf{r}$$
$$\pi_{\mathbf{E},U/T}(\phi_\theta(\epsilon_U(\mathsf{r}), \mathsf{s}) \equiv \mathsf{r}$$

PROOF. The proof is given in the electronic Appendix B.5. □

Timestamp propagation enables the information-preserving Cartesian product [Sarda 1993; Böhlen et al. 2000]. This is important since it allows to express an n-ary temporal join by means of binary joins that are commonly available in relational systems. The information-preserving Cartesian product is supported by retaining original interval timestamps through sequences of Cartesian products or joins such that predicates on original interval timestamps can be evaluated on intermediate binary join results.

Our timestamp propagation approach goes beyond the information-preserving Cartesian product, since it allows propagated timestamps for all schema robust operators (cf. Definition 3.7), including temporal outer joins and temporal aggregation.

In the context of timestamp propagation, it is important whether or not an operator is *schema preserving*, as this property characterizes the operators for which original timestamps can be preserved for subsequent operators. For instance, the Cartesian product and all types of joins are schema robust as well as schema preserving. In contrast, temporal aggregation is schema robust, but not schema preserving since a single result tuple is not derived from a fixed number of argument tuples. Thus, after an aggregation, the original timestamps are no longer available.

Definition 4.24. (*Schema Preserving Operator*) Let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be relations where relation r_i has schema $R_i = (\mathbf{A}_i)$, ψ be an n-ary operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathsf{r}_1, \ldots, \mathsf{r}_n$, and $\mathsf{r}'_1, \ldots \mathsf{r}'_n$ be relations where r'_i has schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$ where $\mathbf{X}_i$ is a set of attributes. Operator ψ is *schema preserving* iff

$$sch(\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)) = \mathbf{E} \Rightarrow sch(\psi(\mathsf{r}'_1, \ldots, \mathsf{r}'_n)) = \mathbf{E} \cup \bigcup_{i=1}^{n} \mathbf{X}_i$$

Table IV categorizes operators according to the properties of *schema robustness* and *schema preservation*. As the temporal operators are defined in terms of their corresponding nontemporal operators (cf. Section 3), they inherit the same properties.

Table IV. Properties of operators.

Operators	Schema robust	Schema preserving
σ, $\times$, $\bowtie$, $\rtimes$, $\ltimes$, $\overline{\ltimes}$, $\triangleright$	yes	yes
π, ϑ	yes	no
$-$, $\cap$, $\cup$	no	yes

In summary, timestamp propagation enables support for extended snapshot reducibility by allowing access to original interval timestamps in predicates and functions of temporal operators. Due to the different properties of relational algebra operators, propagated timestamps need to be removed before the application of operators that are not schema robust, but can be preserved through sequences of schema preserving operators so that they are available to subsequent operators.

4.5. Attribute Value Scaling

We proceed to describe how to combine the scaling of attribute values with the reduction rules in Table III. Attribute value scaling relies on the original attribute value, the adjusted interval timestamp produced by a temporal adjustment operator, and the original interval timestamp. Hence, scaling must be done after temporal adjustment and before the execution of the nontemporal operator. In order to scale attribute values, we thus first employ timestamp propagation to retain a copy of the original intervals during the temporal adjustment process; then scaling is applied, the original timestamp is removed, and the nontemporal operator is applied.

Example 4.25. Consider Example 4.9 with query $Q_2 = \mathsf{m} \; \rtimes^{T:B@scaleU} \; \mathsf{p}$. The result of query Q_2 is shown in Fig. 14. For instance, tuple y_1 records that manager Ann was responsible for project P_1 for the time period [2014-1, 2014-4), for which the available budget was 3K. Tuple y_3 records that Sam was not responsible for any project for the time period [2014-3, 2014-5), thus the project attributes are ω (NULL) values. Note that the budget of project P_1 in relation p and the total budgets associated to the managers supervising project P_1 in Fig. 14 is the same, i.e., 5K = 3K + 2K, as required.

	M	D	P	B	T
y_1	Ann	CS	P_1	3K	[2014-1, 2014-4)
y_2	Sam	MA	P_3	2K	[2014-1, 2014-3)
y_3	Sam	MA	ω	ω	[2014-3, 2014-5)
y_4	Joe	CS	P_1	2K	[2014-4, 2014-6)
y_5	Joe	CS	P_2	6K	[2014-4, 2014-7)

Fig. 14. Projects and budgets each manager is responsible for (result of query Q_2).

We continue with Example 4.9 and Fig. 9. First, since we have to scale the budget, and the reduction of the temporal left outer join requires an absorb, we propagate the timestamp of m and p before doing the alignment. Since timestamp propagation does not change the adjustment, we get adjusted relations that are identical to the ones in Fig. 9 apart from the propagated timestamp. Figure 15 illustrates the aligned relations at the top. Next, the budget of the adjusted relation p is scaled. This is done with a generalized projection. The propagated timestamp attributes are not removed yet, since they are needed by absorb. The final left outer join and absorb operator produces the intended temporal result with scaled budgets.

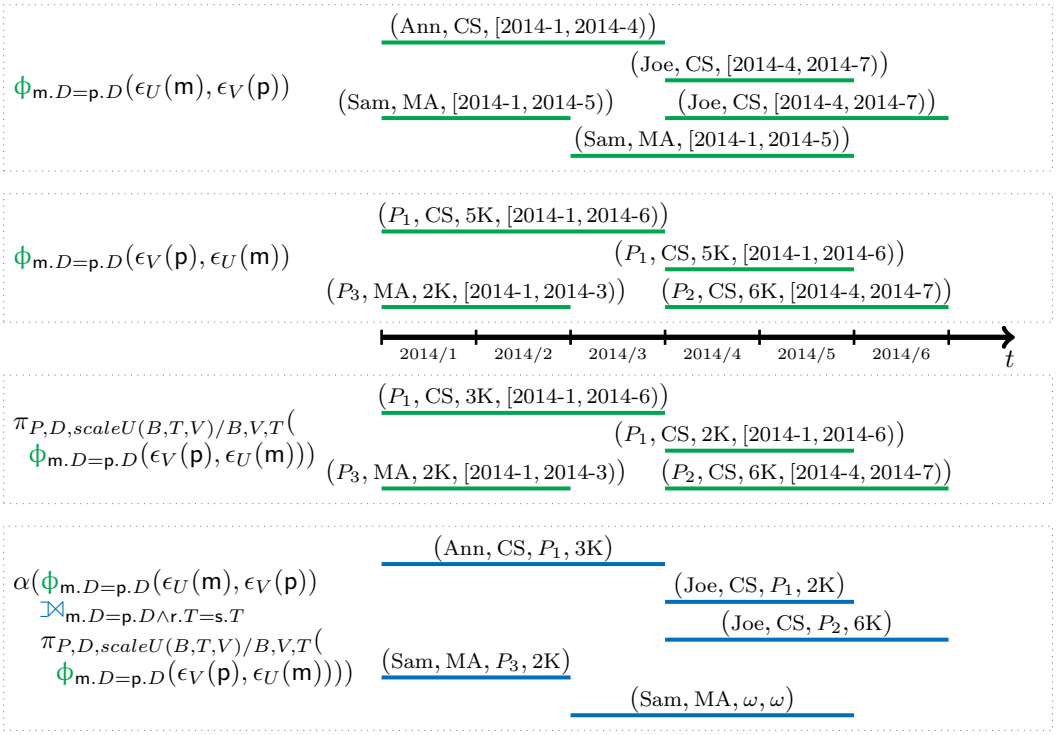


Fig. 15. Reduction of $\mathsf{m}\ \mathbb{N}^{T:B@scaleU}\ \mathsf{p}$.

To support scaling on grouping attributes or on attributes in set operations we need an equality condition with scaling functions as arguments. This can be achieved by substituting in the normalization condition the attribute that must be scaled with a scaling function. For instance, consider temporal difference $\mathsf{r}\ -^{T:B@scaleU}\ \mathsf{s}$ with schemas $R = S = (A, B, T)$. If the difference shall be on the scaled attribute B this can be achieved as follows: $\mathcal{N}_{\mathsf{r}.A=\mathsf{s}.A\wedge scaleU(\mathsf{r}.B,U\cap V,U)=scaleU(\mathsf{s}.B,U\cap V,V)}(\epsilon_U(\mathsf{r}),\epsilon_V(\mathsf{s})) - \mathcal{N}_{\mathsf{r}.A=\mathsf{s}.A\wedge scaleU(\mathsf{r}.B,U\cap V,U)=scaleU(\mathsf{s}.B,U\cap V,V)}(\epsilon_V(\mathsf{s}),\epsilon_U(\mathsf{r}))$.

When changing the timestamp interval T_{OLD} associated with an attribute value x to an adjusted timestamp interval T_{NEW}, a scaling function can be used to modify the value of x to correspond to the adjusted timestamp (cf. Definition 3.11). Below we describe two example scaling functions: uniform `scaleU` and trend `scaleT` scaling. The PostgreSQL PL/pgSQL code of the scaling functions is given in the electronic Appendix C.

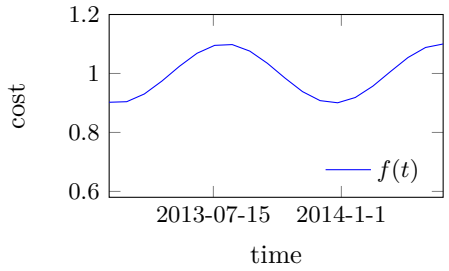
Uniform Scaling. The most common and simplest scaling functions assume a uniform distribution of a value over time so that a scaled value for a new timestamp is the old value multiplied by the duration of the new timestamp divided by that of the old timestamp:

$$scaleU(x, T_{NEW}, T_{OLD}) = x \cdot \frac{te_{new} - ts_{new}}{te_{old} - ts_{old}}$$

Trend Scaling. Attribute values are not always uniformly distributed over time, but may follow a *trend* represented by a function $f(t)$. Given an attribute value x and two time intervals T_{OLD} and T_{NEW}, we can define a scaling function over the integrals of the trend function:

$$scaleT(x, T_{NEW}, T_{OLD}) = x \cdot \frac{\int_{T_{NEW_s}}^{T_{NEW_e}} f(t) \cdot dt}{\int_{T_{OLD_s}}^{T_{OLD_e}} f(t) \cdot dt}$$

Consider an application, where we want to scale according to the cost of power consumption, which fluctuates by 20% due to cooling. Assume that the temperature over a year follows a cosine trend with peaks during summer, as shown in Fig. 16. This can be modeled by the trend function $f(t) = 1 + \frac{\cos(2*\pi*off/365)}{10}$, where *off* is the offset between t and the peak, i.e., $off = t -$ '2013-7-15'.



$$\int_{off_s}^{off_e} f(t) \cdot dt = (off_e - off_s) + \frac{365}{20 \cdot \pi} \cdot (\sin(2 \cdot \pi \cdot \frac{off_e}{365}) - \sin(2 \cdot \pi \cdot \frac{off_s}{365}))$$

Fig. 16. Trend function $f(t)$.

Example 4.26. Fig. 17 shows the result of query Q_1 for the two different scaling functions defined above. In both cases, the result of the grouping is the same (values of attributes D and T), since scaling is used only for the aggregation.

`scaleU`

D	SUM	T
CS	3K	[2014-1, 2014-4)
CS	6K	[2014-4, 2014-6)
CS	2K	[2014-6, 2014-7)
MA	2K	[2014-1, 2014-3)

`scaleT`

D	SUM	T
CS	2.85K	[2014-1, 2014-4)
CS	6.10K	[2014-4, 2014-6)
CS	2.05K	[2014-6, 2014-7)
MA	2K	[2014-1, 2014-3)

Fig. 17. Query Q_1 with different scaling functions.

5. IMPLEMENTATION

In this section, we describe how to integrate the two new temporal operators into the kernel of PostgreSQL. The goal is to show how to enable comprehensive support for sequenced semantics in a cost effective manner. We leverage the capabilities of an existing DBMS and provide an extension that does not interfere with existing functionality.

5.1. Overview

The integration of the temporal normalizer and the temporal aligner into PostgreSQL requires modification of four modules and data structures: the parser and the parse tree, the analyzer and the query tree, the optimizer and the plan tree, and the executor and the execution tree. For each type of tree, a new custom node is introduced that stores information required for the processing of the new operators.

Next the query processing workflow needs to be extended to support the transformations between the new nodes: SQL query $\overset{\text{parser}}{\longrightarrow}$ parse tree $\overset{\text{analyzer}}{\longrightarrow}$ query tree $\overset{\text{optimizer}}{\longrightarrow}$ plan tree $\overset{\text{executor}}{\longrightarrow}$ execution tree.

Finally, the executor module of the query optimizer is extended with three new functions that estimate the cost for the new operators, namely `ExecInit`⟨`Operator`⟩, `Exec`⟨`Operator`⟩, and `ExecEnd`⟨`Operator`⟩ for, respectively, the initialization, the execution, and the finalization of the evaluation algorithms; Here, ⟨`Operator`⟩ is the name of the actual execution algorithm.

For the purpose of illustrating and empirically evaluating the reduction rules, we also extended SQL with the temporal operators. We integrate the algebraic operators directly into SQL to demonstrate that the operators are useful building blocks that support the implementation of a variety of temporal SQL extensions, including existing proposals based on snapshot reducibility (e.g., [Snodgrass 1995; Lorentzos and Mitsopoulos 1997; Toman 1998; Böhlen et al. 2000]). The extended SQL is not intended as a proposal for a new temporal query language.

We proceed to give a detailed description of the integration of the temporal alignment operator, covering first the execution algorithm for alignment and then the extensions to the parser, analyzer, and optimizer. As the extensions that are needed to accommodate the normalization operator are similar, we describe only the differences.

5.2. Execution Algorithm for Temporal Alignment

The implementation of the temporal alignment operator, $\phi_\theta(\mathsf{r},\mathsf{s})$, is a two-step process. First, for each tuple $r_i \in \mathsf{r}$, the group $\mathsf{g}_i \subseteq \mathsf{s}$ of s-tuples that satisfy θ and have overlapping interval timestamps is retrieved. Then, a plane sweep algorithm is applied to each sorted group g_i to produce the aligned relation.

5.2.1. Group Construction. To construct for each tuple r_i the corresponding group g_i of matching s-tuples, we use a system-internal left outer join. To illustrate, we assume two relations r and s and the predicate $(B = D \wedge \mathsf{r}.T \cap \mathsf{s}.T \neq \emptyset)$ as shown in Fig. 18. Tuple r_1 matches two s-tuples, r_2 matches three s-tuples, and r_3 matches no s-tuple; hence, the s-part in the result for r_3 is substituted by ω values. The result of this system-internal left outer join has two timestamp attributes, one from the r-tuple and one from the s-tuple.

r

	RN	A	B	T
r_1	1	a	β	$[1,7)$
r_2	2	b	β	$[3,9)$
r_3	3	c	γ	$[8,10)$

s

	C	D	T
s_1	1	β	$[2,5)$
s_2	2	β	$[3,4)$
s_3	3	β	$[7,9)$

r $\mathbb{N}_{B=D\wedge \mathsf{r}.T\cap \mathsf{s}.T\neq\emptyset}$ s

	RN	A	B	T	C	D	T
$r_1 \circ s_1$	1	a	β	$[1,7)$	1	β	$[2,5)$
$r_2 \circ s_3$	2	b	β	$[3,9)$	3	β	$[7,9)$
$r_1 \circ s_2$	1	a	β	$[1,7)$	2	β	$[3,4)$
$r_2 \circ s_2$	2	b	β	$[3,9)$	2	β	$[3,4)$
$r_3 \circ \omega$	3	c	γ	$[8,10)$	ω	ω	ω
$r_2 \circ s_1$	2	b	β	$[3,9)$	1	β	$[2,5)$

Fig. 18. System-internal left outer join of r-tuples with s-tuples.

Our implementation, as do all other execution functions in PostgreSQL, supports pipelining such that intermediate results do not need to be materialized. To enable this, the join

result is partitioned according to the groups and, within each group, sorted according to the intersection timestamp of the r- and s-tuple. By doing so, tuples with equal intersection timestamps are consecutive, which allows to identify (and remove) duplicate timestamps during the subsequent plane sweep algorithm in step 2. We implement partitioning and sorting by first adding a row number to each r-tuple (attribute RN in Fig. 18) using the PostgreSQL function `row_number()`[5] and then sorting the join result according to the row number and the start and end time points of the intersection timestamps.

Fig. 19 illustrates the group construction for the example in Fig. 18. After partitioning and sorting, all tuples in a group have identical row numbers, namely 1 for $\mathtt{g}_1$, 2 for $\mathtt{g}_2$, and 3 for $\mathtt{g}_3$. The sorting in each group is from top to bottom. The two nearby lines for each tuple in the illustration indicate the two timestamps of the joining tuples.

	RN	A	B	T	C	D	T	
$r_1 \circ s_1$	1	a	β	$[1,7)$	1	β	$[2,5)$	$\mathtt{g}_1$
$r_1 \circ s_2$	1	a	β	$[1,7)$	2	β	$[3,4)$	
$r_2 \circ s_2$	2	b	β	$[3,9)$	2	β	$[3,4)$	$\mathtt{g}_2$
$r_2 \circ s_1$	2	b	β	$[3,9)$	1	β	$[2,5)$	
$r_2 \circ s_3$	2	b	β	$[3,9)$	3	β	$[7,9)$	
$r_3 \circ \omega$	3	c	γ	$[8,10)$	ω	ω	ω	$\mathtt{g}_3$

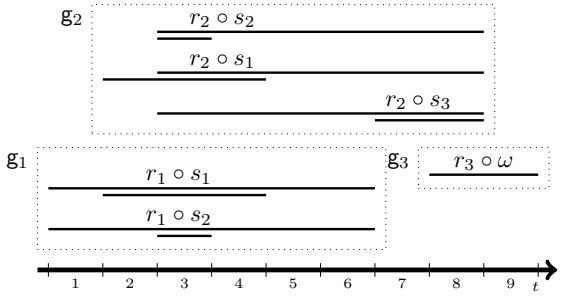


Fig. 19. Partitioning and sorting of groups.

5.2.2. Planesweep Algorithm. The algorithm that computes the aligned relation is shown in Fig. 20. It is implemented in PostgreSQL as an executor function, `ExecAdjustment`, and works for both the alignment operator and the normalization operator (with different input). The function is integrated into the pipelining architecture of PostgreSQL, so on each invocation, it either emits a single result tuple or emits ω to indicate the end of the operation.

The input to the `ExecAdjustment` function is a context node n that stores a number of variables that must be passed between consecutive invocations: a reference to its input (*subnode*), the previous and current tuples from the input (*prev*, *curr*), the sweep-line status (*sweepline*), an output tuple (*out*), a row number (*outrn*) that identifies the group from which the last output tuple was generated, a Boolean *isalign* that indicates alignment versus normalization, and a Boolean (*samegroup*) that is *true* when *prev* and *curr* contain tuples from the same group. Finally, $[P_1, P_2)$ denotes the already computed intersection of the r- and s-tuple.

Fig. 21 illustrates four invocations of `ExecAdjustment` that yield the result tuples $\tilde{r_1}$, $\tilde{r_2}$, $\tilde{r_3}$, and $\tilde{r_4}$. When a new group starts, i.e., when we encounter a new row number rn of the outer tuple in the input, *curr* and *prev* store the same input tuple, *samegroup* is set to *true*, and *sweepline* stores the r-tuple's starting time point. On the first invocation, x_1 is fetched. Here, *samegroup* = *true* and $P_1 = 2012/2$ exceeds *sweepline* $= 2012/1$. Thus, tuple $\tilde{r_1}$ is produced, the row number of the group from which it was derived is stored in *outrn*, and the *sweepline* is advanced to P_1 (first block of the function, lines 10–14).

On the second invocation, *samegroup* = *true* and *sweepline* $= P_1$; hence, the second block (lines 15–23) of the function is entered. We check if the same intersection has already been produced by comparing group and time interval. Since this is not the case, $\tilde{r_2}$ is produced,

[5]https://www.postgresql.org/docs/current/static/functions-window.html

Function: `ExecAdjustment`(n)
Input: Node n in execution tree.
Output: A single output tuple or ω.

```
1  Copy variables of n to local variables;
2  if first call then
3      prev ← next tuple from subnode;
4      curr ← prev;
5      samegroup ← true;
6      sweepline ← curr.T_s;
7      outrn ← −1;
8  produced ← false;
9  while produced = false ∧ prev ≠ ω do
10     if samegroup ∧ curr.P_1 ≠ ω ∧ sweepline < curr.P_1 then
11         out ← (curr.A, [sweepline, curr.P_1));
12         outrn ← curr.rn;
13         produced ← true;
14         sweepline ← curr.P_1;
15     else if samegroup ∧ (curr.P_1 = ω ∨ sweepline ≥ curr.P_1) then
16         if isalign∧(outrn, out.T) ≠ (curr.rn, [curr.P_1, curr.P_2)) then
17             out ← (curr.A, [curr.P_1, curr.P_2));
18             outrn ← curr.rn;
19             sweepline ← max(sweepline, curr.P_2);
20             produced ← true;
21         prev ← curr;
22         curr ← next tuple from subnode;
23         samegroup ← curr ≠ ω ∧ prev.rn = curr.rn;
24     else
25         if sweepline < prev.T_e then
26             out ← (prev.A, [sweepline, prev.T_e));
27             outrn ← prev.rn;
28             produced ← true;
29         prev ← curr;
30         samegroup ← true;
31         if curr ≠ ω then sweepline ← curr.T_s;
32 if produced = false then
33     out ← ω;
34 Copy local variables to n;
35 return out;
```

Fig. 20. Executor function.

the sweepline is advanced to 2012/4, *curr* is copied to *prev*, and the next tuple, x_2, is fetched into *curr*. Because x_2 belongs to the same group as x_1, *samegroup* is set to *true*.

On the third invocation, *samegroup* = *true* and *sweepline* $> P_1$ (= 2012/3). The execution enters again the second block and produces $\tilde{r}_3$. After updating *prev*, the next tuple, x_3, is fetched into *curr*. Since x_3 belongs to a new group, i.e., has a different row number *rn* as it was produced from r_2, *samegroup* is set to *false*.

On the fourth invocation, *samegroup* = *false*, and the execution enters the third block (lines 24–31) of the function. We check if *sweepline* $<$ *prev*.T_e, i.e., if the timestamp of the r-tuple of the previous group is completely covered. This is not so, and a result tuple over the remaining part of the timestamp is produced ($\tilde{r}_4$). The variables are reset for processing the next group.

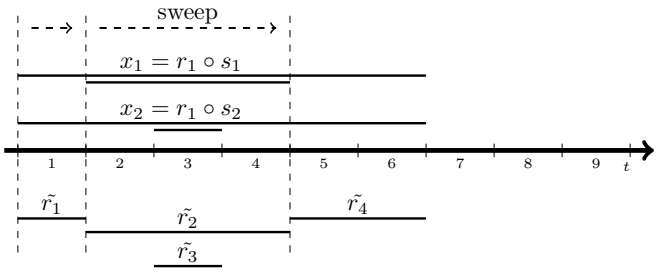


Fig. 21. Plane sweep algorithm for group g_1.

Fig. 22 shows a query execution plan for the query expression $\phi_{\mathsf{r}.B=\mathsf{s}.D}(\mathsf{r},\mathsf{s})/\mathsf{r}$. For the argument relations (80k tuples each), the query optimizer chooses a hash right outer join (with switched argument relations) to build the group (cf. Fig. 18) for each r-tuple (line 5), and it estimates using statistics the join result to hold 94,210 tuples and a given cost. The result is sorted (cf. Fig. 19) by row_number and intersection timestamp (lines 3 and 4). `Adjustment(for ALIGN)` corresponds to `ExecAdjustment` with the flag *isalign* set to *true* (cf. Fig. 20). The cardinality of the output of this step, as we will see in the next section, is estimated to be three times the cardinality of the previous join.

	QUERY PLAN text
1	Subquery Scan on r (cost=41645.33..49653.18 rows=282630 width=16)
2	-> Adjustment(for ALIGN) (cost=41645.33..46826.88 rows=282630 width=32)
3	-> Sort (cost=41645.33..41880.86 rows=94210 width=32)
4	Sort Key: (row_number() OVER (?)), (GREATEST(r_1.ts, s.ts)), (LEAST(r_1.te, s.te))
5	-> Hash Right Join (cost=4502.00..31604.40 rows=94210 width=32)
6	Hash Cond: (s.d = r_1.b)
7	Join Filter: ((r_1.ts < s.te) AND (r_1.te > s.ts))
8	-> Seq Scan on s (cost=0.00..1233.00 rows=80000 width=12)
9	-> Hash (cost=3033.00..3033.00 rows=80000 width=24)
10	-> WindowAgg (cost=0.00..2233.00 rows=80000 width=16)
11	-> Seq Scan on r r_1 (cost=0.00..1233.00 rows=80000 width=16)

Fig. 22. Query plan for $\phi_{\mathsf{r}.B=\mathsf{s}.D}(\mathsf{r},\mathsf{s})/\mathsf{r}$ (screenshot from the PostgreSQL client pgAdmin3).

5.3. Parser, Analyzer, and Optimizer Extensions for Temporal Alignment

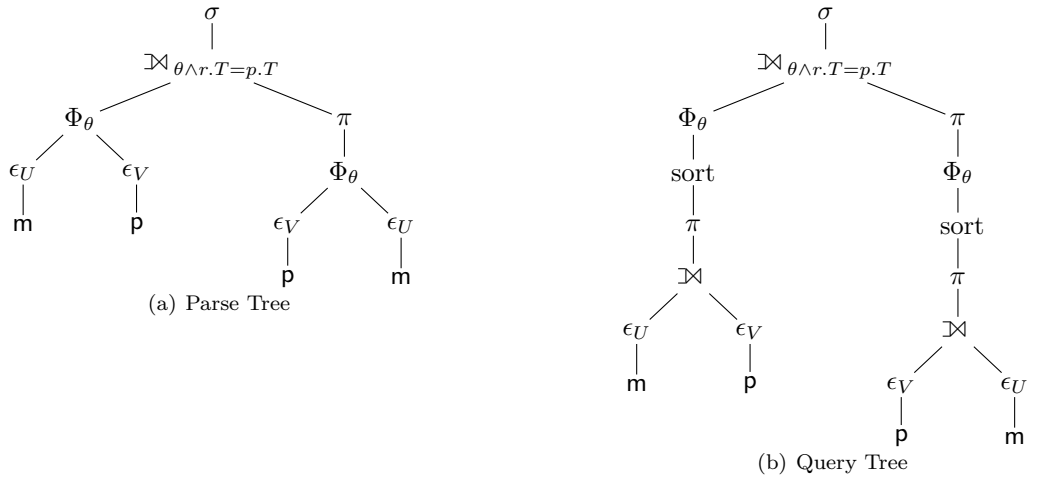
Here, we describe the extensions of the three modules that precede the executor, namely, parser, analyzer, and optimizer. First, we add a new SQL keyword `ALIGN` and extend the grammar of the parser as follows:

```
aligned_table:
  table_ref ALIGN table_ref ON a_expr;
table_ref: ...
  '(' aligned_table ')' alias_clause
```

The alignment statement consists of two `table_ref`s and can be used similarly to any other item in SQL's FROM clause. The first `table_ref` is the relation to align, the second is the reference relation, and `a_expr` is the θ condition. For instance, query $Q_2 = \mathsf{m}\ \tilde{\bowtie}^{T:B@scaleU}\ \mathsf{p}$ can be formulated in the extended SQL as follows:

```
WITH   m AS (SELECT Ts Us, Te Ue, * FROM m),
       p AS (SELECT Ts Vs, Te Ve, * FROM p)
SELECT M, D, P, B, Ts, Te
FROM   (m ALIGN p ON m.D = p.D) m
       LEFT OUTER JOIN
       (SELECT P, D, scaleU(B, Ts, Te, Vs, Ve) B , Ts, Te
        FROM (p ALIGN m ON m.D = p.D) p ) p
       USING (D, Ts, Te)
WHERE (Ts = Us OR Ts = Vs) AND (Te = Ue OR Te = Ve) OR
      Us IS NULL OR Vs IS NULL;
```

Alignment is implemented as defined in Section 4. The statements in the `WITH` clause do timestamp propagation, and the `SELECT` statement implements the sequence of alignment, scaling, and the invocation of the nontemporal operator according to the reduction rule for the temporal left outer join (cf. Table III). The condition in the `WHERE` clause implements the absorb operator as a selection (cf. Proposition 4.17). The corresponding parse tree is shown in Fig. 23(a).



(a) Parse Tree

(b) Query Tree

Fig. 23. Parse tree and query tree of query $Q_2 = \mathsf{m} \overset{T:B@scaleU}{\bowtie} \mathsf{p}$.

In the analyzer, we extend the query tree with the partitioning and sorting of the groups. The resulting query tree for our example is shown in Fig. 23(b).

The optimizer is the last module before the executor. Here, the DBMS chooses among different execution strategies. The cost estimates for the temporal alignment node, where x is the direct subnode, are computed as follows:

$$numRows = 3 \cdot \mathsf{x}.numRows$$
$$cost = \mathsf{x}.cost + (cpu\_tup\_cost + 3 \cdot cpu\_op\_cost) \cdot numRows$$

The cardinality of the output can be up to three times the cardinality of the subnode because the algorithm can produce up to three tuples for every tuple in the input. Note that the subnode is the join as described in the previous section and that the number of rows, $\mathsf{x}.numRows$, has already been estimated by the query optimizer. The total cost is estimated as the cost of the subnode plus, for each result tuple, the cost to produce it and

the cost of up to three attribute comparisons in the executor function. The estimates we provide in this step are used by the optimizer to determine the execution plan with the smallest estimated cost. The estimates derive from the input cardinalities and ensure that the alignment operator is properly integrated into the query optimization process of the DBMS. To further improve the accuracy of the estimation, the catalog with statistics about the data distribution can be used.

5.4. Extensions Needed to Accommodate Temporal Normalization

The integration of temporal normalization is similar to the integration of temporal alignment. One difference relates to the construction of the groups. Temporal normalization splits a tuple's interval according to all start and end time points in its group. To build such groups, we use a system-internal nontemporal left outer join. We impose a total order on split points to be able to use a plane sweep algorithm with constant memory complexity. Therefore, we do not join with the `s` relation directly, but with the union of its start and end points, i.e., $\pi_{\mathbf{A},T_s/P_1}(\mathsf{s}) \cup \pi_{\mathbf{A},T_e/P_1}(\mathsf{s})$. We build the groups as for alignment, sort on the split point P_1, and use the same plane sweep algorithm (Fig. 20) as for temporal alignment, but without the intersection part, i.e., `ExecAdjustment` with *isalign* = *false*. As a result, the sweepline moves from split point to split point to produce the result.

Fig. 24 shows a query execution plan for $\mathcal{N}_{\mathsf{r}.B=\mathsf{s}.D}(\mathsf{r},\mathsf{s})/\mathsf{r}$. The query optimizer chooses a sort-merge join to build the groups for each r-tuple. The result is sorted by row_number and split point. `Adjustment(for NORMALIZE)` corresponds to `ExecAdjustment` with the flag *isalign* set to *false* (cf. Fig. 20).

	QUERY PLAN text
1	Subquery Scan on r (cost=47449.58..51249.58 rows=160000 width=16)
2	-> Adjustment(for NORMALIZE) (cost=47449.58..49649.58 rows=160000 width=28)
3	-> Sort (cost=47449.58..47649.58 rows=80000 width=28)
4	Sort Key: (row_number() OVER (?)), s.ts
5	-> Merge Left Join (cost=28031.75..39019.99 rows=80000 width=28)
6	Merge Cond: (r_1.b = s.d)
7	Join Filter: ((s.ts >= r_1.ts) AND (s.ts < r_1.te))
8	-> Sort (cost=9548.08..9748.08 rows=80000 width=24)
9	Sort Key: r_1.b
10	-> WindowAgg (cost=0.00..2233.00 rows=80000 width=16)
11	-> Seq Scan on r r_1 (cost=0.00..1233.00 rows=80000 width=16)
12	-> Materialize (cost=18483.67..19283.67 rows=160000 width=8)
13	-> Sort (cost=18483.67..18883.67 rows=160000 width=8)
14	Sort Key: s.d
15	-> Append (cost=0.00..2466.00 rows=160000 width=8)
16	-> Seq Scan on s (cost=0.00..1233.00 rows=80000 width=8)
17	-> Seq Scan on s s_1 (cost=0.00..1233.00 rows=80000 width=8)

Fig. 24. Query plan for $\mathcal{N}_{\mathsf{r}.B=\mathsf{s}.D}(\mathsf{r},\mathsf{s})/\mathsf{r}$ (screenshot from the PostgreSQL client pgAdmin3).

The rules for the parser are similar to those introduced for temporal alignment, but we use the keyword `NORMALIZE` and allow `USING (att_list)` as a shorthand for `ON` θ, where θ contains equality conditions over the attributes `att_list`, i.e., `USING (D)` corresponds to `ON r.D=s.D`. For instance, the temporal aggregation $Q_1' = {}_{D}\vartheta^{T:B@scaleU}_{CNT(*),AVG(DUR(V)),SUM(B)}(\epsilon_V(\mathsf{p}))$ is formulated in the extended SQL as follows.

```
WITH p AS (SELECT Ts Vs, Te Ve, * FROM p)
SELECT D, Count(*), AVG(DUR(Vs, Ve)), SUM(B) , Ts, Te
FROM (SELECT N, D, scaleU(B, Ts, Te, Vs, Ve) B , Vs, Ve, Ts, Te
      FROM (p r NORMALIZE p s USING (D)) r) r
GROUP BY D, Ts, Te ;
```

The parse tree of this expression corresponds to Fig. 5 (right side). The optimizer uses the following cost estimations:

$$numRows = 2 \cdot \mathtt{x}.numRows$$
$$cost = \mathtt{x}.cost + (cpu\_tup\_cost + cpu\_op\_cost) \cdot numRows$$

For each split point in the subnode, we can have up to two result tuples, which is less than for the alignment since no intersections are produced (cf. Fig. 20; in the second block only alignment can produce a tuple). Although alignment and normalize have the same upper bound for the output cardinality, the difference is the construction of the subnode. The total cost is the cost of the subnode plus, for each result tuple, the cost to produce it and one attribute comparison (also different from alignment since we omit the intersection part).

6. EMPIRICAL EVALUATION

6.1. Objectives

To evaluate the implementation of the proposed temporal query language, we conduct a series of experiments that offer insights into pertinent aspects of the implementation. First, we explore the degree of integration. Our goal is a tight integration that ensures that the new operators benefit from, and contribute to, e.g., cardinality estimation and cost-based query optimization. Second, we study the performance of temporal adjustment. Specifically, we want to understand the costs of considering lineage information for the adjustment, and compare it with alternatives that do not do this. Third, we investigate the stability of temporal adjustment for different data distributions. The performance of the adjustment should be robust and should not deteriorate if the data distribution changes. Fourth, we quantify the effect of the use of row numbers for constructing groups. Fifth, we analyze the costs of different elements in temporal queries. Without built-in support for time intervals, we end up with syntactically complex queries that are inefficient to evaluate. Adjustment effectively separates the nontemporal and temporal parts of the query processing, and we want to understand the relative performances of these parts. Finally, we compare the performance of our solution with evaluation techniques that rewrite queries and use no or only one adjustment operator.

Note that our goal is not to provide new specialized high-performance algorithms or indexing structures. Likewise, we do not consider main memory processing of queries. While these aspects are important, the focus of our empirical evaluation is to evaluate the integration into the DBMS. How to improve the performance of single components in PostgreSQL is a separate and orthogonal topic that may be studied in follow-on research.

6.2. Experimental Setting

For the experiments, we use a 2.7 GHz Intel Core i7 machine with 16 GB main memory and 750 GB flash storage running Mac OS X. The client and the database server run on the same machine. We use the PostgreSQL server 9.5 extended with our implementation of the normalize and alignment operators. All parameters of the PostgreSQL server, such as the maximum memory for sorting, are kept to default values, and no indexes are used.

We use two real-world datasets. The *Incumbent* dataset [Gendrano et al. 1998] from the University of Arizona has 83,857 tuples. Each tuple records a job assignment (*pcn*)

for an employee (*ssn*) over a specific time interval. The data ranges over 16 years and contains 49,195 employees assigned to 38,178 jobs. The interval timestamps are recorded at the granularity of days and have durations from 1 to 573 days, with an average of approximately 180 days. Next, the *Flight* dataset [Behrend and Schüller 2014] contains 55,072 tuples. Each tuple records the actual time interval of a flight from a departure airport (*fap*) to a destination airport (*dap*). The data ranges over 10 days and contains 559 different departure and 578 different destination airports. The interval timestamps are recorded at the granularity of minutes and have durations from 25 to 915 minutes, with an average of approximately 128 minutes. Synthetic datasets used in the evaluation are described where they are first used.

6.3. DBMS Integration

We want the new functionality to be integrated seamlessly into PostgreSQL. This means that PostgreSQL offers the same support (e.g., tree nodes with cardinality and cost estimation) for the new operators as it offers for the existing operators; and it means that the two new operators interact with other operators to support equivalence-based transformations. As an example, consider the query $\sigma_{T_s>\text{'2014-1-1'}\wedge pcn=1234}(\phi_{\mathsf{r}.pcn=\mathsf{s}.pcn}(\mathsf{r},\mathsf{s}))$ on the *Incumbent* dataset. The query plan from PostgreSQL is shown in Fig. 25.

	QUERY PLAN text
1	Subquery Scan on r (cost=4666.21..4666.49 rows=1 width=16)
2	Filter: (r.ts > '2014-01-01'::date)
3	-> Adjustment(for ALIGN) (cost=4666.21..4666.38 rows=9 width=32)
4	-> Sort (cost=4666.21..4666.22 rows=3 width=32)
5	Sort Key: r_1."RN", (GREATEST(r_1.ts, s.ts)), (LEAST(r_1.te, s.te))
6	-> Nested Loop Left Join (cost=0.00..4666.19 rows=3 width=32)
7	Join Filter: ((r_1.ts < s.te) AND (r_1.te > s.ts) AND (r_1.pcn = s.pcn))
8	-> Subquery Scan on r_1 (cost=0.00..3233.00 rows=3 width=24)
9	Filter: (r_1.pcn = 1234)
10	-> WindowAgg (cost=0.00..2233.00 rows=80000 width=16)
11	-> Seq Scan on r r_2 (cost=0.00..1233.00 rows=80000 width=16)
12	-> Materialize (cost=0.00..1433.02 rows=3 width=12)
13	-> Seq Scan on s (cost=0.00..1433.00 rows=3 width=12)
14	Filter: (pcn = 1234)

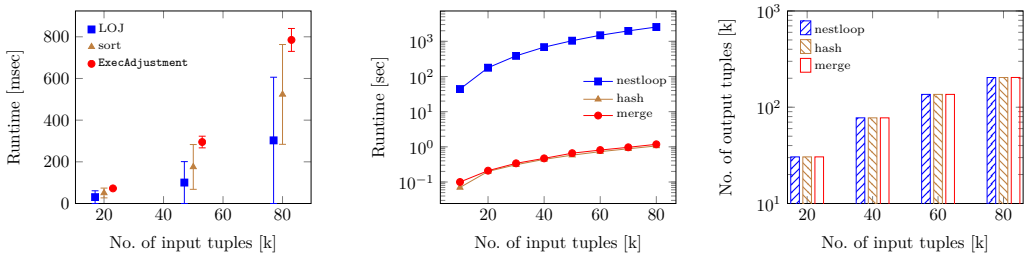
Fig. 25. Query plan for $\sigma_{T_s>\text{'2014-1-1'}\wedge pcn=1234}(\phi_{\mathsf{r}.pcn=\mathsf{s}.pcn}(\mathsf{r},\mathsf{s}))$ on the *Incumbent* dataset (screenshot from the PostgreSQL client pgAdmin3).

Several elements of the query plan relate to the integration. First, equivalence rule E2 is applied, i.e., the selection $pcn = 1234$ is pushed down to the first input relation (line 9), while the selection $T_s >$ '2014-1-1' with timestamp T_s, which is computed as part of the alignment, cannot be pushed down (line 2). Second, equivalence rule E8 is applied, i.e., $\mathsf{r}.pcn = 1234 \wedge \mathsf{r}.pcn = \mathsf{s}.pcn \Rightarrow \mathsf{s}.pcn = 1234$, and condition $\mathsf{s}.pcn = 1234$ is pushed down to the second argument relation (line 14). Third, E5 is applied, i.e., only the required attributes (pcn, T_s, T_e) of the second argument relation are included in the processing (line 12: width is 12 Bytes while the size of an original tuple is 16 Bytes).

In Section 5 we saw that the cost of temporal adjustment is dominated by functionality that we have been able to support by means of a nontemporal left outer join. Thus, the algorithms for temporal adjustment that contain the group construction are capable of leveraging the capabilities of the underlying engine (optimization, different join algorithm

implementations). Fig. 26 investigates this for the temporal normalization $\mathcal{N}_{\mathsf{r}.pcn=\mathsf{s}.pcn}$ on the *Incumbent* dataset.

Fig. 26(a) shows a breakdown of the runtime cost of the normalization operator for, respectively, 20k, 50k, and 80k tuples. LOJ is the system-internal left outer join (cf. Fig. 18), sort is the partitioning and sorting of groups (cf. Fig. 19), and `ExecAdjustment` is the execution algorithm (cf. Fig. 20). The vertical lines in the graph indicate the time span in msec during which the respective parts of the normalization operator were running. The time spans overlap since the parts are connected through pipelining. We can observe that the runtime is dominated by the nontemporal left outer join. We also see that pipelining is enabled, i.e., sorting starts as soon as the LOJ produces the first tuples, and `ExecAdjustment` starts as soon as the first tuple of sorting is computed but not all tuples may be read yet from the sorted runs.



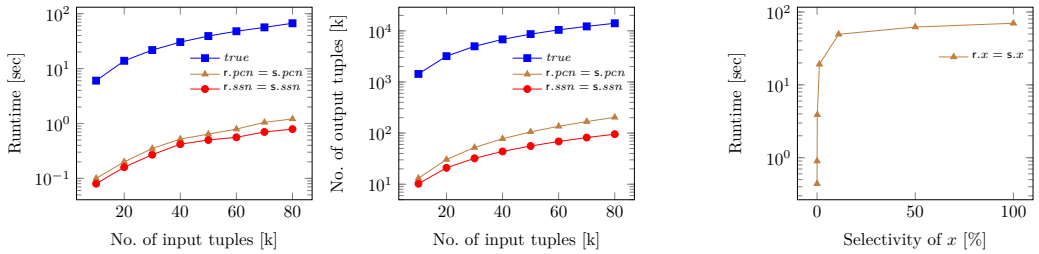
(a) Runtime Composition (b) Runtime (c) Output Size

Fig. 26. Normalization $\mathcal{N}_{\mathsf{r}.pcn=\mathsf{s}.pcn}$ on the *Incumbent* dataset.

We then used the DBMS with three different settings: (a) all join methods enabled, (b) merge join disabled (i.e., SET enable_mergejoin=false), and (c) merge and hash join disabled. For each of the three settings the database chooses the best available join strategy for the left outer join in the normalization operator: in (a), a sorted merge join; in (b), a hash join; and in (c), a nested loop join. Fig. 26(b) shows the runtime of the normalization, which is dominated by the nontemporal left outer join for the group construction and for which the DBMS chooses the best available join algorithm. The same observation holds for the temporal alignment. Hence, the runtimes of the normalization and alignment operators are proportional to the runtime of a nontemporal left outer join. The output cardinality of the normalization is shown in Fig. 26(c), which is obviously the same for all settings.

6.4. Cost of Temporal Adjustment

In this experiment, we evaluate the performance impact of considering change preservation. Applications that only want snapshot reducibility, but not change preservation, are free to ignore the lineage information and do more splits (fewer splits would violate snapshot reducibility and is not an option). The experiment evaluates the costs of additional splits. We do so by measuring the performance of temporal normalization with different normalization conditions. Fig. 27(a) shows the runtimes and output cardinalities for varying input cardinalities for the normalization of the *Incumbent* dataset using three conditions: *true* (i.e., split at each start and end point), $\mathsf{r}.pcn = \mathsf{s}.pcn$ (i.e., group by *pcn* and split within groups), and $\mathsf{r}.ssn = \mathsf{s}.ssn$ (i.e., group by *ssn* and split within groups). Condition $\mathsf{r}.ssn = \mathsf{s}.ssn$ is more selective since there are more distinct values of *ssn* than of *pcn* in the dataset. Thus, there are more splits for the equality on *ssn* than for the equality on *pcn*. Clearly, the number of splits influences performance noticeably, and there is a strong correlation between the runtime and the number of splits.



(a) *Incumbent* Dataset (b) Random Dataset

Fig. 27. Runtime and output size of normalization operator.

In Fig. 27(b), we use 80k tuples from the *Incumbent* dataset and add an additional attribute x for which we vary the selectivity, i.e., the average number of occurrences of a value x divided by the number of tuples (in percent). Thus, a selectivity of 100% means all x values are equal. The more selective the condition of the normalization, i.e., the lower the selectivity, the more efficient the normalization is, since it produces fewer splits and the internal left outer join with condition $\mathsf{r}.x = \mathsf{s}.x$ is evaluated more efficiently. In Fig. 27(a) the selectivities are 100% for condition *true*, 0.003% for *pcn*, and 0.002% for *ssn*.

There are mainly two implications from this experiment. First, more splits decrease performance, since they produce a larger output of the adjustment operators. Second, the performance of the adjustment operators is correlated with the selectivity of condition θ. This is because current left outer join implementations, used in the adjustment operators, are not able to further optimize the condition on the timestamp attributes (cf. Fig. 22 line 7), but thanks to the tight integration of our adjustment operators, they will immediately take advantage of future optimizations of join algorithms in this direction.

6.5. Impact of Row Numbers for Group Building

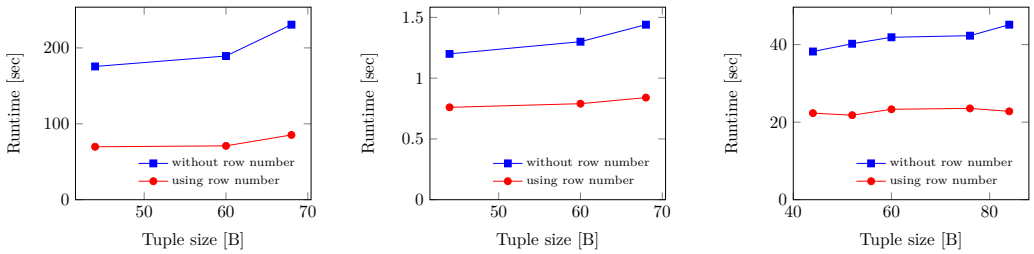
In this experiment, we study the effect of using row numbers for group building as described in Section 5. We do this by comparing with an implementation [Dignös et al. 2012] that has been made available to us and that uses attribute values to build groups. As an input for the `ExecAdjustment` function, both approaches use the internal sorting of the DBMS to first partition groups and then sort them by split point (for the normalizer) or intersection timestamp (for the aligner). The advantage of using row numbers is that a single attribute identifies a group.

Fig. 28 compares the two implementation approaches for different normalization conditions on the *Incumbent* and *Flight* datasets. The size of input tuples is varied by adding attributes.

The approach using row numbers is faster since it allows the `ExecAdjustment` function to identify if two consecutive tuples belong to the same group by looking at the row number only, i.e., one numeric value. Note that both approaches get slower if the tuple size increases, since more data needs to be read. However, when using row numbers, the performance is more stable since the sort comparisons are always performed on the same number of attributes, independently of the tuple size.

6.6. Reduction using Adjustment Operators and Timestamp Propagation

In this experiment, we analyze the costs of the different elements involved in computing a temporal operator. Recall that we transform a temporal operator to a nontemporal operator over an argument relation with adjusted intervals. The aim is to study if the adjustment operators enable an efficient processing of the subsequent nontemporal operator. A direct

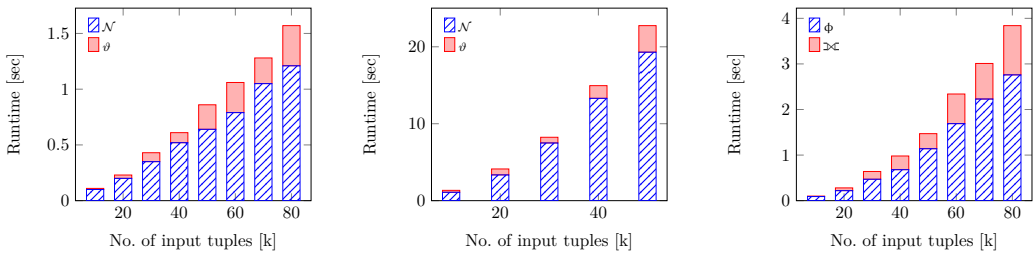


(a) *Incumbent*, $\mathcal{N}_{true}$ (b) *Incumbent*, $\mathcal{N}_{\mathsf{r}.ssn=\mathsf{s}.ssn}$ (c) *Flight*, $\mathcal{N}_{\mathsf{r}.fap=\mathsf{s}.fap}$

Fig. 28. Implementation with or without row numbers for group building, normalization.

formulation of the temporal query yields a syntactically complex query with many inequalities that is difficult to optimize. The adjustment operators, instead, first split the timestamp intervals, upon which equality can be used to efficiently compare the split intervals. Since the individual parts of the query evaluation are connected via pipelining, we determine the runtime of the adjustment operation in isolation and show its impact on the overall runtime of the reduction of a temporal operator.

Fig. 29 shows the runtime breakdown for the reduction of a temporal aggregation and a temporal full outer join. Fig. 29(a) shows the result for temporal aggregation on the *Incumbent* dataset, i.e., ${}_{pcn}\vartheta^{T}_{CNT(*)}(\mathsf{r}) = {}_{pcn,T}\vartheta_{CNT(*)}(\mathcal{N}_{\mathsf{r}.pcn=\mathsf{s}.pcn}(\mathsf{r},\mathsf{r}/\mathsf{s}))$. We observe that the normalization enables an efficient nontemporal aggregation. Fig. 29(b) shows a similar picture for a temporal aggregation on the *Flight* dataset (grouping by *fap*). The nontemporal aggregation after the normalization is more efficient for the *Flight* dataset since it contains shorter intervals (cf. Section 6.2) and, as a consequence, produces fewer splits than the *Incumbent* dataset. This yields a smaller output from the normalization, which is the input to the nontemporal aggregation. Fig. 29(c) shows the same experiment for the reduction of a temporal full outer join. We see the same pattern as for aggregation: the alignment takes care of the complex adjustment of intervals and enables an efficient equality-based nontemporal full outer join.

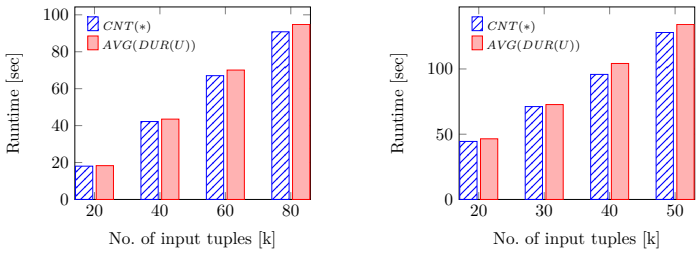


(a) *Incumbent*, ${}_{pcn}\vartheta^{T}$ (b) *Flight*, ${}_{fap}\vartheta^{T}$ (c) *Incumbent*, $\mathbb{X}\!\!\subset^{T}_{\mathsf{r}.pcn=\mathsf{s}.pcn}$

Fig. 29. Runtime breakdown for aggregation and full outer join.

The next experiment analyzes the impact of timestamp propagation on query performance. We consider temporal aggregation on our real-world datasets and compute ${}_{T}\vartheta_{CNT(*)}(\mathcal{N}_{true}(\mathsf{r},\mathsf{r}/\mathsf{s}))$, which does not require timestamp propagation and calculates at each time point the number of project assignments (for the *Incumbent* dataset) and the number of flights (for the *Flight* dataset). We compare the result with the computation of ${}_{T}\vartheta_{AVG(DUR(U))}(\mathcal{N}_{true}(\epsilon_U(\mathsf{r}),\epsilon_U(\mathsf{r})/\mathsf{s}))$, which uses timestamp propagation and calculates

at each time point the average duration of project assignment and the average duration of flights for the two datasets, respectively. The results of this comparison are shown in Fig. 30. We observe that timestamp propagation adds only little overhead since the propagated timestamps increase the size of the input tuples by a small amount only.



(a) *Incumbent* dataset (b) *Flight* dataset

Fig. 30. Runtime comparison between aggregation with and without timestamp propagation (no grouping).

To summarize, the adjustment is beneficial for the subsequent operators since they do not have to deal with overlapping intervals and only need to evaluate a few additional equality conditions over the adjusted intervals, which the DBMS does very efficiently. Similarly, timestamp propagation, which is needed for extended snapshot reducibility and scaling, adds very little overhead.

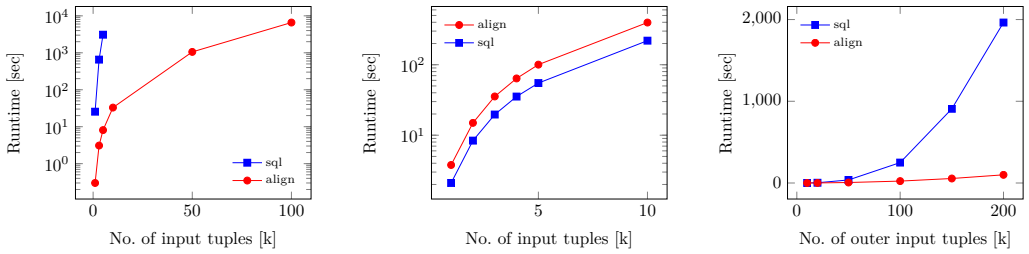
6.7. Expressing Temporal Outer Joins in SQL

If no built-in temporal support is available in the DBMS, queries must be expressed in SQL. We proceed to compare the performance of using the built-in temporal support to that of custom crafted SQL queries that give the same result. Specifically, we compare the computation of different temporal outer joins using temporal alignment (*align*) with the computation of temporal outer joins expressed in standard SQL (*sql*). In order to express a temporal outer join in SQL, we express the join part with the overlap predicate on the timestamps, and we express the negative part of the temporal outer join with joins and NOT EXISTS statements [Snodgrass 2000, pp. 154–156]. The union of the two parts gives the final result.

For the comparison we use three different settings. First, a left outer join, $O_1 = \mathsf{r} \,⟕^T_{true}\, \mathsf{s}$, without a join predicate, and evaluated on two synthetic datasets, namely $\mathbf{D}^{disj}$, where the intervals in both relations are disjoint, and $\mathbf{D}^{eq}$, where the intervals in both relations are equal. Second, a left outer join, $O_2 = \epsilon_U(\mathsf{r}) \,⟕^T_{Min \leq DUR(U) \leq Max}\, \mathsf{s}$, with a predicate over the interval timestamp of the left argument relation r, and evaluated on a synthetic dataset, $\mathbf{D}^{rand}$, with uniformly distributed interval timestamps, where relation r represents hotel reservations and relation s, of size 0.5% of the size of r, represents prices for reservations with a duration in a specified range (Min, Max) and in a given season. Third, a full outer join with an equality predicate on our real-word datasets, i.e., $O_3 = \mathsf{r} \,⟗^T_{\mathsf{r}.pcn=\mathsf{s}.pcn}\, \mathsf{s}$ on the *Incumbent* dataset and $O_3 = \mathsf{r} \,⟗^T_{\mathsf{r}.fap=\mathsf{s}.fap}\, \mathsf{s}$ on the *Flight* dataset.

Fig. 31(a) shows the runtime of query O_1 on $\mathbf{D}^{disj}$, showing that *align* performs much faster than *sql*. The reason is the NOT EXISTS predicates that are evaluated using anti joins and that are only efficient if a match is found at an early stage so that the evaluation can terminate and return *false*. Since there are no overlapping intervals in both relations, the NOT EXISTS clause has to scan almost the entire relation, which yields a quadratic runtime complexity. PostgreSQL provides GiST indices for range types, but multikey GiST indices with other attributes are not yet supported. Since we do not have other equality

attributes for O_1, we also measured the performance after creating a GiST index. The runtime for 100k input tuples dropped (from 6590 sec) to 13.1 sec for *align* and to 26.6 sec for *sql*. Next, we increased the input relations to 10M tuples each, upon which the result was 1163 sec for *align* and 3653 sec for *sql*. The runtime improvement using the GiST index is only due to a more efficient execution of the alignment operators (cf. blue shaded area in Fig. 29), as the subsequent join has the same number of tuples to process. This shows that our solution is integrated tightly and, similar to any other DBMS operator, takes advantage of new DBMS functionalities without any additional implementation effort. The best setting for *sql* for query O_1 is on the dataset $\mathbf{D}^{eq}$, shown in Fig. 31(b). All interval timestamps of $\mathbf{D}^{eq}$ are equal, meaning that the NOT EXISTS clause can be evaluated efficiently, and terminates immediately after checking the first tuple. The experiment reveals that *sql* is extremely sensitive to the dataset in contrast to *align*, even if the join condition is *true*, which is efficient for a NOT EXISTS evaluation. In this setting, a GiST index does not improve performance since the optimizer correctly chooses to not use the index because of the high selectivity.



(a) O_1 on $\mathbf{D}^{disj}$ dataset (b) O_1 on $\mathbf{D}^{eq}$ dataset (c) O_2 on $\mathbf{D}^{rand}$ dataset

Fig. 31. Query O_1 and O_2 on synthetic datasets.

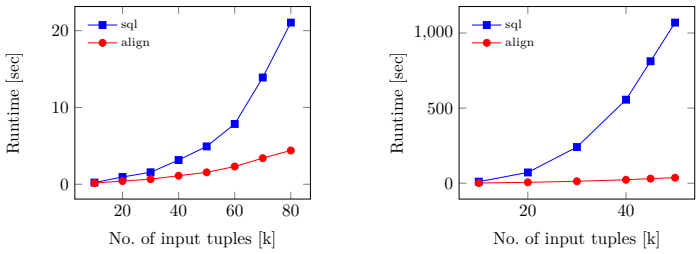
In the next experiment, we compare the runtimes for query O_2, which contains a complex join condition with inequalities. The result is shown in Fig. 31(c). Also for this case, *align* performs much better than *sql* since the complex join condition prevents an efficient evaluation of the NOT EXISTS clauses, even though the inner relation's cardinality containing the ranges (Min, Max) is small, i.e., only 0.5% of the outer relation's cardinality. We repeated this experiment after creating GiST indices. For 200k outer tuples, the result was 52.74 sec for *align* and 728.31 sec for *sql*.

Finally, we run query O_3 on the *Incumbent* and *Flight* datasets (Fig. 32). Both approaches run much faster for the real-world datasets than for the synthetic datasets since the equality condition allows the DBMS to choose a fast nontemporal hash join or merge join in the case of temporal alignment, and to speed up the NOT EXISTS statements using for instance a hash anti join in the case of SQL. Again, *align* performs much faster than the SQL approach, especially for the *Flight* dataset, where the join condition is less selective than for the *Incumbent* dataset.

Remember that there are also very substantial benefits to using a temporal SQL in terms of the ease of formulating queries and proving them correct.

6.8. Expressing Temporal Outer Joins with SQL and Normalize

Here, we compare the computation of temporal outer joins using temporal alignment (*align*) with an approach that expresses temporal outer joins using standard SQL plus temporal normalization for the negative part (*sql+*$\mathcal{N}$). We do this experiment to analyze the relevance of providing a temporal alignment operator not only as a logical but also as a physical algebra operator. The joined part of the temporal outer join is computed with SQL, and

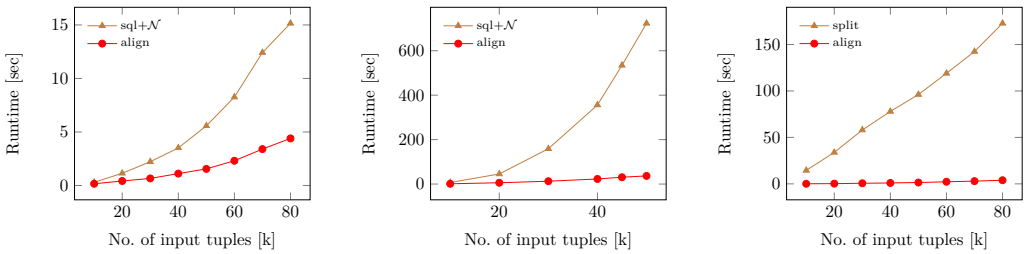


(a) *Incumbent* Dataset (b) *Flight* Dataset

Fig. 32. Runtime of query O_3 on real-world datasets.

temporal normalization is used for the temporal difference. Expressing outer joins with difference requires the computation of the difference between an argument relation and the intermediate join result to determine all tuples that are not joining so that they can be concatenated with ω-values and included in the result. We use query O_3 from the previous section.

Figs. 33(a) and 33(b) show the runtime behavior of query O_3 on the real-world datasets. We can observe that in both experiments, *align* performs much faster than *sql+*$\mathcal{N}$ due to the expensive normalization steps that *sql+*$\mathcal{N}$ is required to perform on the intermediate join result. We can also see that *sql+*$\mathcal{N}$ has a much higher runtime for the *Flight* dataset, since the predicate for this outer join is less selective than the one for the *Incumbent* dataset. As a consequence, the intermediate join result, on which the normalization is applied, is much larger.



(a) *Incumbent* dataset (b) *Flight* dataset (c) *Incumbent* dataset

Fig. 33. Runtime of query O_3 on real-world datasets.

Finally, Fig. 33(c) considers an approach (split) that first splits the timestamp intervals of the tuples at all start and end time points of all other tuples an then performs a nontemporal full outer join. Note that this approach is not change preserving. Its performance is poor since many splits lead to a high number of intermediate tuples that are passed to the subsequent nontemporal full outer join. In this example, the cardinality of the final result is approximately 100 times larger than the change preserving result of the alignment.

7. RELATED WORK

The management of temporal data in DBMSs has been an active research area for several decades, focusing primarily on temporal data models and query languages (e.g., [Abiteboul et al. 1996; Böhlen and Jensen 2003; Date and Darwen 2002; Jensen et al. 1994; Snodgrass 1995; Böhlen et al. 2009]) as well as efficient algorithms for specific operators (e.g., temporal

join [Segev 1993; Soo et al. 1994; Dignös et al. 2014; Piatov et al. 2016] and temporal aggregation [Böhlen et al. 2006b; Vega Lopez et al. 2005; Gamper et al. 2009]).

To make the formulation of temporal queries more convenient, various temporal query languages [Böhlen and Jensen 2003; Tansel et al. 1993] have been proposed. The earliest approach to add temporal support to relational query languages, such as SQL, was to introduce new data types with associated predicates and functions that were strongly influenced by Allen's interval relationships [Allen 1983]. Extending an existing query language with new data types is simple and facilitates the formulation of some temporal queries. However, this approach does not provide a systematic way to generalize nontemporal queries to temporal queries since it does not effectively support, e.g., temporal aggregation and temporal set difference. Therefore, new constructs were added to SQL with the goal of expressing temporal queries more easily. Below we discuss the languages and techniques that are directly relevant to our solution. Note that this article's goal is not to provide an extension of SQL, but to provide native temporal database support that is generic and can be used for implementing different existing temporal extensions to SQL.

IXSQL [Date and Darwen 2002; Lorentzos and Mitsopoulos 1997] normalizes timestamps and provides two functions, *unfold* and *fold*, that are used as follows: (i) unfold transforms an interval timestamped relation into a point timestamped relation by splitting each interval timestamped tuple into a set of point timestamped tuples; (ii) the corresponding nontemporal operation is applied to the normalized relation; (iii) fold collapses value-equivalent tuples over consecutive time points into interval timestamped tuples over maximal time intervals. The approach is conceptually simple, but timestamp normalization using fold and unfold does not respect lineage, and no efficient implementation has been provided.

Next, SQL/TP is an approach based on point timestamped relations [Toman 1996; Toman 1998]. A temporal relation is modeled as a sequence of nontemporal relations (or snapshots), and the corresponding nontemporal operations are applied to each of the snapshots to answer temporal queries. To provide an efficient evaluation, an interval encoding of point timestamped relations was proposed together with a *normalization* function. The normalization splits overlapping value-equivalent argument tuples into tuples with equal or disjoint timestamps, and SQL/TP queries are then mapped to standard SQL statements with equality predicates. Toman's normalization function satisfies the properties of our temporal normalizer for group based operators, and we leverage the normalization for the splitting of interval timestamps of group based operators. SQL/TP considers neither lineage nor extended snapshot reducibility, which are not relevant for point timestamped relations. Normalization is not applicable to tuple based operators, such as joins, outer joins, and anti joins, since for these operators, it would not respect lineage.

Agesen et al. [2001] introduce a *split* operator that extends normalization to bitemporal relations. The operator splits argument tuples that are value-equivalent over nontemporal attributes into tuples over smaller, yet maximal timestamps such that the new timestamps are either equal or disjoint. The focus of this work is to support temporal aggregation and difference in now-relative bitemporal databases. This study is limited to value-equivalent tuples, i.e., tuples with pairwise identical nontemporal attributes, and it does not apply to change preserving joins, outer joins, and anti joins.

ATSQL [Böhlen et al. 2000] offers a systematic way to construct temporal SQL queries from nontemporal SQL queries. The main idea is to formulate the nontemporal query and use *statement modifiers* to specify whether the statement is to be evaluated with sequenced or nonsequenced semantics. In the context of ATSQL, different desiderata for temporal languages were formulated, including the sequenced semantics. A native database implementation of this approach has yet to be provided.

In terms of query processing, various query algorithms for selected operators have been proposed. Join algorithms are based on indexing [Son and Elmasri 1996; Zhang et al. 2002] or well-known nested loop, sort merge, and partitioning strategies [Gao et al. 2005]. Similarly,

several solutions for the evaluation of various forms of temporal aggregation [Böhlen et al. 2006b; Kline and Snodgrass 1995; Moon et al. 2003; Yang and Widom 2003; Zhang et al. 2001] exist.

The support for temporal data in commercial DBMSs has focused on new data types with associated predicates and functions. A temporal PostgreSQL module [Davis 2009] introduces a PERIOD datatype for anchored time intervals together with Boolean predicates and functions, such as intersection, union, and minus. Most of the functionality of this module was integrated into the core of PostgreSQL version 9.2 using range types [PostgreSQL Global Development Group 2012]. Range types are generic interval datatypes with associated predicates, functions, and indices. They facilitate the formulation of some temporal queries, but they do not conveniently support queries that need to adjust the timestamps of tuples, such as difference, aggregation, and outer joins.

Oracle [Murray 2008] provides a PERIOD datatype with predicates and functions, and additionally supports valid and transaction time (DBMS_WM package). Querying temporal relations is only possible at a specific time point (snapshot). Teradata [Teradata 2014; Al-Kateb et al. 2013] provides similar temporal support as Oracle, i.e., the PERIOD datatype with associated predicates and functions as well as valid time and transaction time. As of release 13.10, Teradata supports the temporal statement modifiers `SEQUENCED` and `NONSEQUENCED` [Böhlen et al. 2000] in queries. Teradata implements sequenced queries using query rewriting, i.e., a temporal query is rewritten as a standard SQL query [Al-Kateb et al. 2013]. The support for sequenced queries is limited to inner joins. Sequenced outer joins, set operations, duplicate elimination, and aggregation are not supported.

The main memory database system SAP HANA [Kaufmann et al. 2013b; Kaufmann et al. 2013a; Kaufmann et al. 2015] is currently being extended to support specific temporal operators, such as time travel, temporal aggregation, and temporal join on top of a temporal index. Operations such as outer joins and scaling are not considered.

The SQL:2011 [Kulkarni and Michels 2012] standard supports period specifications over two attributes that should be considered as an interval timestamp. This approach has been implemented in IBM DB2 [Saracco et al. 2012] and consists of support for application-time (valid-time) and system-time (transaction-time) period tables bundled with support for temporal insertion, update, and deletion. The support for querying is limited to simple range restrictions and predicates.

The scaling of attribute values in response to the adjustment of interval timestamps has received little attention and no general implementations of scaling have been provided. Böhlen et al. [2006a] propose three different attribute characteristics: *constant* attributes that never change value during query processing, *malleable* attributes that require an adjustment of the value when the timestamp changes, and *atomic* attributes that become undefined (invalid) when the timestamp changes. For malleable attributes, an adjustment function is proposed. We use the terminology from this work, propose an implementation, and extend the work to scale attribute values in aggregate functions, grouping, set operations, and join conditions. Terenziani and Snodgrass [2004] distinguish between atelic facts that are valid for each point in time and telic facts that are only valid for one specific interval. That work focuses on the semantics of facts recorded in a database and proposes a three-sorted relational model (atelic, telic, nontemporal). We provide a solution that allows applications to flexibly scale attribute values at query time, and we integrate support for scaling into a query language that adjusts intervals and allows to propagate the original intervals.

Dignös et al. [2012] introduced temporal adjustment operators and timestamp propagation as a solution for computing temporal queries over interval timestamped relations using sequenced semantics. They show in a demonstration [Dignös et al. 2013] that scaling of attributes values is possible during query processing. The present study builds on this previous work to provide a comprehensive foundation for and build an industrial-strength

implementation of a complete sequenced temporal query language. Besides the foundation and implementation the following parts are new. We prove that the adjustment operators are not affected by timestamp propagation, and vice versa that timestamp propagation is not affected by the adjustment operators. We prove that scaling must be a parameter of the temporal algebra operators and cannot be performed as a simple pre- or post-processing step of the operators. We prove that the size of the transformed algebra expressions in Dignös et al. [2012] can be exponential in the number of temporal operators in the original query expression and show how this exponential growth can be reduced to linear complexity with the help of common table expressions. We provide and prove equivalence rules that work out the interaction of the adjustment operators with the selection and projection operators. We provide novel implementation techniques that yield significant performance improvements, e.g., by using row numbers instead of all attributes of a tuple for the grouping step inside the adjustment operators, and by showing how to implement the absorb operator by a generalized selection, which permits a single uniform scaling procedure for all operators of the temporal relational algebra.

The focus of Dyreson et al. [2015] is to provide a uniform framework for the evaluation of queries under different temporal semantics, including the two extremes of sequenced and nonsequenced semantics. Additional semantics can be realized in this framework, such as context, periodic, and preceding semantics. The framework uses lineage to track tuples through operations. The work is primarily at the conceptual level, trying to unify and reconcile different temporal semantics.

8. CONCLUSION

We present a principled solution for querying interval-timestamped data together with a full-fledged, industrial-strength implementation in the kernel of a relational DBMS that does not affect the processing of nontemporal queries while offering effective support for sequenced temporal queries. The solution transforms temporal operators into the corresponding nontemporal operators with the help of two key concepts: interval adjustment and timestamp propagation. For the interval adjustment, two new temporal operators are introduced, namely a temporal normalizer for group-based operators and a temporal aligner for tuple-based operators. Timestamp propagation makes it possible to use timestamp attributes in query conditions. With the help of these novel concepts, a set of reduction rules is provided that transform any temporal query with relational operators into queries that use the corresponding nontemporal operators.

The processing of sequenced temporal queries works in four steps. The first three steps, respectively, propagate the timestamp interval attribute, adjust the argument tuples in such a way that the timestamps correspond to the timestamps of the result tuples, and scale the attribute values to the new timestamp. In the fourth step, the adjusted argument relations are processed using the corresponding nontemporal operators, which yields the final result of the temporal operator. The adjustment operators guarantee snapshot reducibility and change preservation by splitting the argument tuples according to the operator. Timestamp propagation enables extended snapshot reducibility as well as attribute value scaling. We show how to integrate the new operators and the transformation rules into the kernel of PostgreSQL to build the first industrial-strength open-source DBMS with full-fledged built-in temporal support, which is currently not offered by any commercial DBMS. We also cover optimizations at the implementation level together with equivalence rules that show how the new adjustment operators commute with the nontemporal relational algebra operators and can be used by the query optimizer. The results of detailed empirical studies with the proposed temporal DBMS, which is available as open source software, offer insights into pertinent design properties of the new framework.

Given our insights, future work points in several directions. First, it would be interesting to propose a user-level temporal language and its mapping to our temporal algebra. In

contrast to our current prototype, where queries explicitly incorporate the transformation rules, such a user-level temporal query language would allow to directly use the temporal operators in the query formulation, and the transformation to the nontemporal operators would be done by the DBMS while parsing the query. Second, opportunities exist for improving the efficiency of query processing beyond the optimizations presented in this paper. Currently, our implementation focuses on achieving a cost effective (i.e., minimal changes to the host DBMS) and tightly integrated solution that leverages the services of an existing DBMS. Opportunities include (a) implementing new special-purpose algorithms for specific sequenced temporal algebra operators that bypass our reduction rules, (b) improving the adjustment operators for specific scenarios and offering additional adjustment operators, and (c) developing new DBMS algorithms, such as an interval merge-join and indexing techniques that are directly beneficial for our solution. Finally, it is of interest to apply the proposed temporal DBMS in various application contexts as well as to other types of interval data than temporal data.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank Peter Moser for his help in implementing the extensions in PostgreSQL.

REFERENCES

Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. 1996. Temporal versus first-order logic to query temporal databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '96)*. 49–57.

Mikkel Agesen, Michael H. Böhlen, Lasse Poulsen, and Kristian Torp. 2001. A split operator for now-relative bitemporal databases. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*. 41–50.

Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. 573–578.

James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (1983), 832–843.

John Bair, Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 1997. Notions of upward compatibility of temporal query languages. *Wirtschaftsinformatik* 39, 1 (1997), 25–34.

Andreas Behrend and Gereon Schüller. 2014. A case study in optimizing continuous queries using the magic update technique. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM '14)*. 31:1–31:4.

Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2006a. An algebraic framework for temporal attribute characteristics. *Ann. Math. Artif. Intell.* 46, 3 (2006), 349–374.

Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2006b. Multi-dimensional aggregation for temporal data. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT '06)*. 257–275.

Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Richard T. Snodgrass. 2009. SQL-based temporal query languages. See Liu and Özsu [2009], 2762–2768.

Michael H. Böhlen and Christian S. Jensen. 2003. Temporal data model and query language concepts. In *Encyclopedia of Information Systems*. 437 – 453.

Michael H. Böhlen and Christian S. Jensen. 2009. Sequenced semantics. See Liu and Özsu [2009], 2619–2621.

Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 2000. Temporal statement modifiers. *ACM Trans. Database Syst.* 25, 4 (2000), 407–456.

Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 2009. Current semantics. See Liu and Özsu [2009], 544–545.

Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 2009. Nonsequenced Semantics. See Liu and Özsu [2009], 1913–1915.

Sarah Cohen Boulakia and Wang Chiew Tan. 2009. Provenance in scientific databases. See Liu and Özsu [2009], 2202–2207.

Jan Chomicki, David Toman, and Michael H. Böhlen. 2001. Querying ATSQL databases with temporal logic. *ACM Trans. Database Syst.* 26, 2 (2001), 145–178.

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227.

Chris Date and Hugn Darwen. 2002. *Temporal data and the relational model*. Morgan Kaufmann Publishers Inc.

Jeff Davis. 2009. Online temporal PostgreSQL reference. http://temporal.projects.postgresql.org/reference.html. (2009).

Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2012. Temporal alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. 433–444.

Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2013. Query time scaling of attribute values in interval timestamped databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE '13)*. 1304–1307.

Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 1459–1470.

Curtis E. Dyreson, Venkata A. Rani, and Amani Shatnawi. 2015. Unifying sequenced and non-sequenced semantics. In *Proceedings of the 22nd International Symposium on Temporal Representation and Reasoning (TIME '15)*. 38–46.

Johann Gamper, Michael H. Böhlen, and Christian S. Jensen. 2009. Temporal aggregation. See Liu and Özsu [2009], 2924–2929.

Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. 2005. Join operations in temporal databases. *The VLDB Journal* 14, 1 (2005), 2–29.

Jose A. G. Gendrano, R. Shah, Richard T. Snodgrass, and Jun Yang. 1998. University information system (UIS) dataset. TimeCenter CD-1. (1998).

Boris Glavic. 2010. *Perm: Efficient provenance support for relational databases*. Ph.D. Dissertation. University of Zurich. http://cs.iit.edu/~dbgroup/pdfpubls/G10a.pdf

Boris Glavic, Gustavo Alonso, Renée J. Miller, and Laura M. Haas. 2010. TRAMP: Understanding the behavior of schema mappings through provenance. *PVLDB* 3, 1 (2010), 1314–1325.

Christian S. Jensen and Richard T. Snodgrass. 2009. Timeslice operator. See Liu and Özsu [2009], 3120–3121.

Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. 1994. Unifying temporal data models via a conceptual model. *Inf. Syst.* 19, 7 (1994), 513–547.

Martin Kaufmann, Peter M. Fischer, Norman May, Chang Ge, Anil K. Goel, and Donald Kossmann. 2015. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE '15)*. 471–482.

Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013a. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 1173–1184.

Martin Kaufmann, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, and Franz Färber. 2013b. Comprehensive and interactive temporal query processing with SAP HANA. *PVLDB* 6, 12 (2013), 1210–1213.

Nick Kline and Richard T. Snodgrass. 1995. Computing temporal aggregates. In *Proceedings of the 11th International Conference on Data Engineering (ICDE '95)*. 222–231.

Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Record* 41, 3 (2012), 34–43.

Wei Li, Richard T. Snodgrass, Shiyan Deng, Vineel Kumar Gattu, and Aravindan Kasthurirangan. 2001. Efficient sequenced integrity constraint checking. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*. 131–140.

Ling Liu and M. Tamer Özsu (Eds.). 2009. *Encyclopedia of Database Systems*. Springer US.

David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2006. Transaction time support inside a database engine. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. 35.

Nikos A. Lorentzos and Yannis G. Mitsopoulos. 1997. SQL extension for interval data. *IEEE Trans. on Knowl. and Data Eng.* 9, 3 (1997), 480–499.

Bongki Moon, Ines Fernando Vega Lopez, and Vijaykumar Immanuel. 2003. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. on Knowl. and Data Eng.* 15, 3 (2003), 744–759.

Chuck Murray. 2008. Oracle database workspace manager developer's guide. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf. (2008).

Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE '16)*. 1098–1109.

PostgreSQL Global Development Group. 2012. Documentation manual PostgreSQL - range types. http://www.postgresql.org/docs/9.2/static/rangetypes.html. (2012).

Ravi Rajamani. 2007. *Oracle total recall/flashback data archive*. Technical Report. Oracle.

Cynthia Saracco, Matthias Nicola, and Lenisha Gandhi. 2012. A matter of time: Temporal data management in DB2 10. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf. (2012).

Nandlal L. Sarda. 1993. HSQL: A historical query language. In *Temporal Databases*. Benjamin/Cummings Publishing Company, 110–140.

Arie Segev. 1993. Join processing and optimization in temporal relational databases. See Tansel et al. [1993], Chapter 15, 356–387.

Richard T. Snodgrass (Ed.). 1995. *The TSQL2 temporal query language*. Kluwer Academic Publishers.

Richard T. Snodgrass. 2000. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc.

Richard T. Snodgrass. 2010. *A case study of temporal data*. Teradata Corporation.

Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. 1997. Transitioning temporal support in TSQL2 to SQL3. In *Temporal Databases, Dagstuhl*. 150–194.

Daeweon Son and Ramez Elmasri. 1996. Efficient temporal join processing using time index. In *Proceedings of the 8th International Conference on Scientific and Statistical Database Management (SSDBM '96)*. 252–261.

Michael D. Soo, Christian S. Jensen, and Richard T. Snodgrass. 1995. An algebra for TSQL2. See Snodgrass [1995], Chapter 27, 505–546.

Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. 1994. Efficient evaluation of the valid-time natural join. In *Proceedings of the 10th International Conference on Data Engineering (ICDE '94)*. 282–292.

Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass (Eds.). 1993. *Temporal databases: Theory, design, and implementation*. Benjamin/Cummings.

Teradata. 2014. Teradata database temporal table support. http://www.info.teradata.com/eDownload.cfm?itemid=131540028. (2014).

Paolo Terenziani and Richard T. Snodgrass. 2004. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Trans. Knowl. Data Eng.* 16, 5 (2004), 540–551.

David Toman. 1996. Point vs. interval-based query languages for temporal databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '96)*. 58–67.

David Toman. 1998. Point-based temporal extensions of SQL and their efficient implementation. In *Temporal databases: Research and practice*. LNCS, Vol. 1399. 211–237.

Ines Fernando Vega Lopez, Richard T. Snodgrass, and Bongki Moon. 2005. Spatiotemporal aggregate computation: A survey. *IEEE Trans. on Knowl. and Data Eng.* 17, 2 (2005), 271–286.

Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal* 12, 3 (2003), 262–283.

Donhui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. 2001. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '01)*. 237–245.

Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. 2002. Efficient temporal join processing using indices. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. 103–113.

Online Appendix to: Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries

ANTON DIGNÖS, Free University of Bozen-Bolzano
MICHAEL H. BÖHLEN, University of Zurich
JOHANN GAMPER, Free University of Bozen-Bolzano
CHRISTIAN S. JENSEN, Aalborg University

A. DEFINITION OF RELATIONAL ALGEBRA

Definition A.1 (*Relational Algebra*). Let r and s be relations, θ be a predicate, $\mathbf{B}$ be a subset of the attributes of the schema of r, i.e., $\mathbf{B} \subseteq sch(\mathsf{r})$, $\mathbf{F} = \{f_1, \ldots, f_k\}$ be a set of aggregation functions, where f_i takes a relation as argument and applies aggregation to the values of one of the relation's attributes. The resulting value is stored as the value of an attribute named f_i. The relational algebra is defined as follows:

$$
\begin{aligned}
\sigma_\theta(\mathsf{r}) &= \{r \mid r \in \mathsf{r} \wedge \theta(r)\} \\
\pi_{\mathbf{B}}(\mathsf{r}) &= \{r.\mathbf{B} \mid r \in \mathsf{r}\} \\
{}_{\mathbf{B}}\vartheta_{\mathbf{F}}(\mathsf{r}) &= \{r.\mathbf{B} \circ \mathbf{X} \mid r \in \mathsf{r} \wedge \mathsf{r}_g = \{r' \mid r' \in \mathsf{r} \wedge r'.\mathbf{B} = r.\mathbf{B}\} \wedge \mathbf{X} = f_1(\mathsf{r}_g), \ldots, f_k(\mathsf{r}_g)\} \\
\mathsf{r} - \mathsf{s} &= \{r \mid r \in \mathsf{r} \wedge r \notin \mathsf{s}\}, \text{ where } sch(\mathsf{s}) = sch(\mathsf{r}) \\
\mathsf{r} \cup \mathsf{s} &= \{r \mid r \in \mathsf{r} \vee r \in \mathsf{s}\}, \text{ where } sch(\mathsf{s}) = sch(\mathsf{r}) \\
\mathsf{r} \cap \mathsf{s} &= \{r \mid r \in \mathsf{r} \wedge r \in \mathsf{s}\}, \text{ where } sch(\mathsf{s}) = sch(\mathsf{r}) \\
\mathsf{r} \times \mathsf{s} &= \{r \circ s \mid r \in \mathsf{r} \wedge s \in \mathsf{s}\} \\
\mathsf{r} \bowtie_\theta \mathsf{s} &= \{r \circ s \mid r \in \mathsf{r} \wedge s \in \mathsf{s} \wedge \theta(r, s)\} \\
\mathsf{r} \mathbin{⟕}_\theta \mathsf{s} &= \{r \circ s \mid r \in \mathsf{r} \wedge s \in \mathsf{s} \wedge \theta(r, s)\} \cup \\
&\quad \{r \circ (\omega, \ldots, \omega) \mid r \in \mathsf{r} \wedge \nexists s \in \mathsf{s}(\theta(r, s))\} \\
\mathsf{r} \mathbin{⟗}_\theta \mathsf{s} &= \{r \circ s \mid r \in \mathsf{r} \wedge s \in \mathsf{s} \wedge \theta(r, s)\} \cup \\
&\quad \{(\omega, \ldots, \omega) \circ s \mid s \in \mathsf{s} \wedge \nexists r \in \mathsf{r}(\theta(r, s))\} \\
\mathsf{r} \mathbin{⟗}_\theta \mathsf{s} &= \{r \circ s \mid r \in \mathsf{r} \wedge s \in \mathsf{s} \wedge \theta(r, s)\} \cup \\
&\quad \{r \circ (\omega, \ldots, \omega) \mid r \in \mathsf{r} \wedge \nexists s \in \mathsf{s}(\theta(r, s))\} \cup \\
&\quad \{(\omega, \ldots, \omega) \circ s \mid s \in \mathsf{s} \wedge \nexists r \in \mathsf{r}(\theta(r, s))\} \\
\mathsf{r} \triangleright_\theta \mathsf{s} &= \{r \mid r \in \mathsf{r} \wedge \nexists s \in \mathsf{s}(\theta(r, s))\}
\end{aligned}
$$

In Cartesian product and joins, we assume that the concatenation operator $\circ$ disambiguates each attribute X with the same name in r and s as follows: if $\theta \Rightarrow \mathsf{r}.X = \mathsf{s}.X$, we apply the standard approach used by the natural join, i.e., X appears in the result schema only once and the first non-null value in either $r.X$ or $s.X$ is used. Otherwise, attribute X is prefixed by the relation name and a dot, i.e., $\mathsf{r}.X$ and $\mathsf{s}.X$.

B. PROOFS

B.1. Proof of Lemma 4.13

Proof Rule E1. We show that the left-hand side of rule E1 is equivalent to its right-hand side. The left-hand side $\sigma_{\theta_1 \wedge \theta_2}(\mathcal{N}_\theta(\mathsf{r}, \mathsf{s}))$ is expressed using Definition 4.8 as follows:

 0362-5915/2016/01-ARTXXXX $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

$$\{\tilde{r} \mid \exists r \in \mathsf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \mathsf{s}' = \{s \in \mathsf{s} \mid \theta(r,s)\} \wedge \tilde{r}.T \in normalize(r,\mathsf{s}')) \wedge \theta_1(\tilde{r}) \wedge \theta_2(\tilde{r})\},$$

which is equivalent to:

$$\{\tilde{r} \mid \exists r \in \mathsf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \mathsf{s}' = \{s \in \mathsf{s} \mid \theta(r,s)\} \wedge \tilde{r}.T \in normalize(r,\mathsf{s}') \wedge \theta_1(\tilde{r}) \wedge \theta_2(\tilde{r}))\}$$

Since θ_2 only contains attributes of $\tilde{r}.\mathbf{A}$, i.e., $T \notin attr(\theta_2)$, by transitivity from $\tilde{r}.\mathbf{A} = r.\mathbf{A}$ and $\theta_2(\tilde{r})$, we have that $\theta_2(\tilde{r}) \equiv \theta_2(r)$, and we get:

$$\{\tilde{r} \mid \exists r \in \mathsf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \mathsf{s}' = \{s \in \mathsf{s} \mid \theta(r,s)\} \wedge \tilde{r}.T \in normalize(r,\mathsf{s}') \wedge \theta_1(\tilde{r}) \wedge \theta_2(r))\}$$

Thus corresponds to the right-hand side $\sigma_{\theta_1}(\mathcal{N}_\theta(\sigma_{\theta_2}(\mathsf{r}),\mathsf{s}))$. □

The proof of rule E2 is identical to the proof of E1. The proofs of rules E3 to E8 follow the same structure and reasoning as the proof of E1.

B.2. Proof of Theorem 4.16

Proof. We prove the reduction rule for the temporal left outer join, $\mathsf{r} \mathbin{⟕}^T_\theta \mathsf{s}$, by showing that the operator satisfies the three properties of the sequenced semantics.

Snapshot reducibility (cf. Def. 3.1): We have to show two cases. Case 1: For each pair of matching and intersecting tuples $r \in \mathsf{r}$ and $s \in \mathsf{s}$ (i.e., $\theta(r,s)$ is true and $r.T \cap s.T \neq \emptyset$) the following holds: for each $t \in r.T \cap s.T$ there exists a result tuple $z = (r.\mathbf{A}, s.\mathbf{C}, T)$ such that $t \in T$. Case 2: For each $r \in \mathsf{r}$ and interval $T' \subseteq r.T$, for which no matching and intersecting $s \in \mathsf{s}$ exists, the following holds: for each $t \in T'$ there exists a result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ such that $t \in T$.

From Def. 4.8 (temporal alignment) and Proposition 4.11, we know that aligned tuples $\tilde{r} \in \Phi_\theta(\mathsf{r},\mathsf{s})$ are derived from an $r \in \mathsf{r}$ as follows: (i) for each matching and intersecting $s \in \mathsf{s}$, we get $\tilde{r} = (r.\mathbf{A}, r.T \cap s.T)$; and (ii) for each maximal subinterval $T \subseteq r.T$ that is not covered by any matching $s \in \mathsf{s}$, we get $\tilde{r} = (r.\mathbf{A}, T)$. The same holds for the aligned tuples $\tilde{s} \in \Phi_\theta(\mathsf{s},\mathsf{r})$.

From (i), we conclude that for any two matching and intersecting tuples $r \in \mathsf{r}$ and $s \in \mathsf{s}$, there exist aligned tuples $\tilde{r} = (r.\mathbf{A}, r.T \cap s.T)$ and $\tilde{s} = (s.\mathbf{C}, s.T \cap r.T)$. Since intersection is commutative, $r.T \cap s.T = s.T \cap r.T$, and the nontemporal left outer join yields a result tuple $z = (r.\mathbf{A}, s.\mathbf{C}, r.T \cap s.T)$ that covers each $t \in r.T \cap s.T$ (proves case 1). From (ii), we conclude that for each $r \in \mathsf{r}$ and maximal subinterval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathsf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, T)$ but no matching $\tilde{s} \in \Phi_\theta(\mathsf{s},\mathsf{r})$ that intersects T. Thus, the nontemporal left outer join yields a result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ that covers each $t \in T$ (proves case 2).

The final absorb operator, α, removes tuples that are covered by a value-equivalent tuple. Thus, if a tuple z is removed, each $t \in z.T$ is covered by another value-equivalent result tuple z'.

Extended snapshot reducibility (Def. 3.9): To prove extended snapshot reducibility, we show that propagated timestamps do not interfere with the alignment of the argument relations and hence with the production of result tuples. Recall that relations are extended, i.e., each $r \in \mathsf{r}$ ($s \in \mathsf{s}$) has a nontemporal attribute $r.U$ ($s.U$) that is a copy of $r.T$ ($s.T$), and in θ all references to timestamps have been substituted with $r.U$ and $s.U$, respectively. Since θ is independent of the timestamp attributes, alignment and nontemporal left outer join work exactly in the same way as for snapshot reducibility.

From (i), we conclude that for any two matching and intersecting tuples $r \in \mathsf{r}$ and $s \in \mathsf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, r.U, r.T \cap s.T)$ and an $\tilde{s} = (s.\mathbf{C}, s.U, s.T \cap r.T)$ that yield a result tuple $z = (r.\mathbf{A}, r.U, s.\mathbf{C}, s.U, r.T \cap s.T)$ that covers each $t \in r.T \cap s.T$ (proves case 1). From

(ii), we conclude that for each $r \in \mathsf{r}$ and maximal sub-interval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathsf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, r.U, T)$ but no matching $\tilde{s} \in \phi_\theta(\mathsf{s}, \mathsf{r})$ that intersects T. This yields a result tuple $z = (r.\mathbf{A}, r.U, \omega, \ldots, \omega, T)$ that covers each $t \in T$ (proves case 2).

Change preservation (Def. 3.4): From Def. 4.8 (temporal alignment) and Proposition 4.11, we know that the timestamp of each result tuple is (case 1) either an intersection of two argument tuples, $r \in \mathsf{r}$ and $s \in \mathsf{s}$, or (case 2) a maximal subinterval $T \in r.T$ for which no matching and intersecting tuple $s \in \mathsf{s}$ exists. Furthermore, the α-operator ensures that all result tuples have maximal timestamps.

Case 1: We show that for each result tuple $z = (r.\mathbf{A} \circ z.\mathbf{C}, r.T \cap s.T)$, the lineage set $\mathsf{L}[\mathsf{r} \rtimes_\theta^T \mathsf{s}](z,t)$ is equal for each $t \in z.T$ and that adjacent tuples have different lineage sets. From Def. 3.2 (lineage set), we get $\mathsf{L}[\mathsf{r} \rtimes_\theta^T \mathsf{s}](z,t) = \mathsf{L}[\mathsf{r} \bowtie_\theta^T \mathsf{s}](z,t)$ for case (1). The lineage set of the temporal join contains all $r \in \mathsf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t and all $s \in \mathsf{s}$ that are value-equivalent to $z.\mathbf{C}$ and cover t. Since relations are duplicate free, the lineage set contains exactly one $r \in \mathsf{r}$ and one $s \in \mathsf{s}$, i.e., $\mathsf{L}[\mathsf{r} \bowtie_\theta^T \mathsf{s}](z,t) = \{\langle r, s\rangle\}$. This holds for all $t \in r.T \cap s.T$. To show that the lineage set at time point $z.T_s - 1$ is different for other tuples, recall that either $z.T_s - 1 \notin r.T$ or $z.T_s - 1 \notin s.T$ since $z.T = r.T \cap z.T$. Hence, at least one of r and s is not in the lineage set. The same reasoning applies for time point $z.T_e$.

Case 2: We show that for each result tuple $z = (r.\mathbf{A} \circ (\omega, \ldots, \omega), T)$, the lineage set $\mathsf{L}[\mathsf{r} \rtimes_\theta^T \mathsf{s}](z,t)$ is equal for all $t \in z.T$ and that adjacent tuples have different lineage sets. From Def. 3.2, we get $\mathsf{L}[\mathsf{r} \rtimes_\theta^T \mathsf{s}](z,t) = \mathsf{L}[\mathsf{r} \rhd_\theta^T \mathsf{s}](z,t)$. The lineage set of the temporal anti-join contains all $r \in \mathsf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t. Since relations are duplicate free, we get $\mathsf{L}[\mathsf{r} \rhd_\theta^T \mathsf{s}](z,t) = \{\langle r, \perp\rangle\}$. This holds for all $t \in z.T$ since $z.T = \tilde{r}.T \subseteq r.T$. To show that the lineage set of adjacent tuples is different at time point $z.T_s - 1$, recall that $z.T$ is maximal. Either $z.T_s - 1 \notin r.T$ meaning that r is not in the lineage set, or there exists a matching $s \in \mathsf{s}$ with $z.T_s - 1 \in s.T$ that would produce a join with $r.\mathbf{A}$, and thus the lineage set would be different. The same reasoning applies for the time point $z.T_e$. □

B.3. Proof of Proposition 4.17

Proof. We show that selection σ_Θ removes all tuples that are completely covered by another value-equivalent tuple. It follows that after applying the selection, all tuples in the result have maximal timestamps.

Recall from Corollary 4.11 that each tuple $\tilde{r}$ in the alignment $\phi_\theta(\mathsf{r}, \mathsf{s})$ has an interval timestamp that is either the intersection of two matching tuples from r and s or a maximal sub-interval of a tuple in r that is not covered by any matching tuple in s. The symmetric case holds for each tuple $\tilde{s}$ in $\phi_\theta(\mathsf{s}, \mathsf{r})$. The nontemporal operators $\times$, $\bowtie$, $\rtimes$, $\ltimes$, and $\rhd$ with condition $\theta \wedge \tilde{\mathsf{r}}.T = \tilde{\mathsf{s}}.T$ then produce a result tuple z by either (a) a join match or (b) a result over the negation.

If z is produced by a join match of $\tilde{r}$ and $\tilde{s}$, we need to ensure that $z.T$ is the intersection of the interval timestamps of the two tuples r and s from which $\tilde{r}$ and $\tilde{s}$, respectively, were derived (which ensures a maximal interval timestamp). We have $z.T = \tilde{r}.T = \tilde{s}.T$ and $z.T \subseteq r.T \cap s.T$, and we need to ensure $z.T = r.T \cap s.T$ (because $\tilde{r}.T$ may be the intersection of two tuples r and s', and $\tilde{s}.T$ may be the intersection of two tuples r' and s, but not r and s). Given the original interval timestamps U and V of tuples r and s, respectively, we can ensure that $z.T$ is the intersection of r and s by checking that both $z.T_s$ and $z.T_e$ come from at least one tuple as follows: $(T_s = U_s \vee T_s = V_s) \wedge (T_e = U_e \vee T_e = V_e)$.

If z is produced by a negation from $\tilde{r}$ then V is ω. Similarly, if z is produced by a negation from $\tilde{s}$, we have that U is ω. It follows from Corollary 4.11 that for these cases, the sub-intervals are maximal and that directly before (or after), either a match exists that produces

a tuple that is not value-equivalent, or the tuple from which z is produced does not exist (at that time). □

B.4. Proof of Lemma 4.22

PROOF. To prove the lemma, we use intermediate relations $\mathsf{r}' = \epsilon_U(\mathsf{r})$, $\mathsf{r}'' = \mathcal{N}_\theta(\mathsf{r}', \mathsf{s})$, and $\mathsf{r}''' = \pi_{E,T}(\mathsf{r}'')$, and we need to show that $\mathsf{r}''' \equiv \mathcal{N}_\theta(\mathsf{r}, \mathsf{s})$. We have that $(\mathbf{E}, T) = sch(\mathsf{r})$ and $U \notin attr(\theta)$.

Then from Definition 3.6 we have:

$$\mathsf{r}' = \{r' \mid \exists r \in \mathsf{r}(r'.\mathbf{E} = r.\mathbf{E} \wedge r'.U = r.T \wedge r'.T = r.T)\}$$

Applying Definition 4.3, we get:

$$\mathsf{r}'' = \{r'' \mid \exists r \in \mathsf{r}(r'.\mathbf{E} = r.\mathbf{E} \wedge r'.U = r.T \wedge r'.T = r.T \wedge r''.\mathbf{E} = r'.\mathbf{E} \wedge r''.U = r'.U \wedge r''.T \in normalize(r', \{s \in \mathsf{s} \mid \theta(r', s)\}))\}$$

Since $U \notin attr(\theta)$ and $r'.\mathbf{E} = r.\mathbf{E} \wedge r'.T = r.T$ holds, we have that $normalize(r', \{s \in \mathsf{s} \mid \theta(r', s)\}) \equiv normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\})$. By using transitivity of equality, we get:

$$\mathsf{r}'' = \{r'' \mid \exists r \in \mathsf{r}(r''.\mathbf{E} = r.\mathbf{E} \wedge r''.U = r.T \wedge r''.T \in normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\}))\}$$

We now apply the projection and get:

$$\mathsf{r}''' = \{r''' \mid \exists r \in \mathsf{r}(r''.\mathbf{E} = r.\mathbf{E} \wedge r''.U = r.T \wedge r''.T \in normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\}) \wedge r'''.\mathbf{E} = r''.\mathbf{E} \wedge r'''.T = r''.T)\}$$

By applying transitive rules of equality, we have:

$$\mathsf{r}''' = \{r''' \mid \exists r \in \mathsf{r}(r'''.\mathbf{E} = r.\mathbf{E} \wedge r'''.T \in normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\})\} \equiv \mathcal{N}_\theta(\mathsf{r}, \mathsf{s})$$

The proofs of the other equivalences of Lemma 4.22 follow the same reasoning. □

B.5. Proof of Lemma 4.23

PROOF. To prove the lemma, we use intermediate relations $\mathsf{r}' = \epsilon_U(\mathsf{r})$, $\mathsf{r}'' = \mathcal{N}_\theta(\mathsf{r}', \mathsf{s})$, and $\mathsf{r}''' = \pi_{E,U/T}(\mathsf{r}'')$, and we need to show that $\mathsf{r}''' \equiv \mathsf{r}$. We have that $(\mathbf{E}, T) = sch(\mathsf{r})$ and $U \notin attr(\theta)$.

Then from Definition 3.6, we have:

$$\mathsf{r}' = \{r' \mid \exists r \in \mathsf{r}(r'.\mathbf{E} = r.\mathbf{E} \wedge r'.U = r.T \wedge r'.T = r.T)\}$$

Applying Definition 4.3, we get:

$$\mathsf{r}'' = \{r'' \mid \exists r \in \mathsf{r}(r'.\mathbf{E} = r.\mathbf{E} \wedge r'.U = r.T \wedge r'.T = r.T \wedge r''.\mathbf{E} = r'.\mathbf{E} \wedge r''.U = r'.U \wedge r''.T \in normalize(r', \{s \in \mathsf{s} \mid \theta(r', s)\}))\}$$

Since $U \notin attr(\theta)$ and $r'.\mathbf{E} = r.\mathbf{E} \wedge r'.T = r.T$ holds, we have that $normalize(r', \{s \in \mathsf{s} \mid \theta(r', s)\}) \equiv normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\})$, and by using transitivity of equality, we get:

$$\mathsf{r}'' = \{r'' \mid \exists r \in \mathsf{r}(r''.\mathbf{E} = r.\mathbf{E} \wedge r''.U = r.T \wedge r''.T \in normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\}))\}$$

We now apply the projection and get:

$$\mathsf{r}''' = \{r''' \mid \exists r \in \mathsf{r}(r''.\mathbf{E} = r.\mathbf{E} \wedge r''.U = r.T \wedge r''.T \in normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\}) \wedge r'''.\mathbf{E} = r''.\mathbf{E} \wedge r'''.T = r''.U)\}$$

Since $normalize(r, \{s \in \mathsf{s} \mid \theta(r, s)\})$ always returns at least one result but is not part of the final result, we can replace it by true and again apply transitivity of equality as follows:

$$\mathsf{r}''' = \{r''' \mid \exists r \in \mathsf{r}(r'''.\mathbf{E} = r.\mathbf{E} \wedge r'''.T = r.T\} = \mathsf{r}$$

The proof of $\pi_{r.\mathbf{A},U/T}(\phi_\theta(\epsilon_U(\mathsf{r}), \mathsf{s}) \equiv \mathsf{r}$ follows the same reasoning. □

C. EXAMPLE SCALING FUNCTIONS IN PL/PGSQL

C.1. Uniform Scaling

```
CREATE OR REPLACE FUNCTION
    scaleU(x FLOAT, ts_new DATE, te_new DATE, ts_old DATE, te_old DATE)
    RETURNS FLOAT AS
$$
DECLARE
    w FLOAT;
BEGIN
    w := (EXTRACT(YEAR FROM AGE(te_new, ts_new)) * 12 +
          EXTRACT(MONTH FROM AGE(te_new, ts_new)))
         /
         (EXTRACT(YEAR FROM AGE(te_old, ts_old)) * 12 +
          EXTRACT(MONTH FROM AGE(te_old, ts_old)));
    RETURN x * w;
END;
$$ LANGUAGE PLPGSQL;
```

Function `AGE(DATE, DATE)`[6] is used to determine the difference between two time points in years and months.

C.2. Trend Scaling

```
CREATE OR REPLACE FUNCTION
    scaleT(x FLOAT, ts_new DATE, te_new DATE, ts_old DATE, te_old DATE)
    RETURNS FLOAT AS
$$
DECLARE
    w FLOAT; w_old FLOAT; w_new FLOAT; off_s INT; off_e INT;
BEGIN
    off_s := ts_old-'2013/7/15';
    off_e := te_old-'2013/7/15';
    w_old := (off_e - off_s) + 365 / (20 * pi()) *
               (sin((2*pi()*off_e) / 365) - sin((2*pi()*off_s) / 365));
    off_s := ts_new-'2013/7/15';
    off_e := te_new-'2013/7/15';
    w_new := (off_e - off_s) + 365 / (20 * pi()) *
               (sin((2*pi()*off_e) / 365) - sin((2*pi()*off_s) / 365));
        w := w_new / w_old;
    RETURN x * w;
END;
$$ LANGUAGE PLPGSQL;
```

[6] https://www.postgresql.org/docs/current/static/functions-datetime.html