Temporal Alignment

Anton Dignös
Department of Computer Science
University of Zürich, Switzerland
adignoes@ifi.uzh.ch

Michael H. Böhlen
Department of Computer Science
University of Zürich, Switzerland
boehlen@ifi.uzh.ch

Johann Gamper
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
gamper@inf.unibz.it

ABSTRACT

In order to process interval timestamped data, the sequenced semantics has been proposed. This paper presents a relational algebra solution that provides native support for the three properties of the sequenced semantics: snapshot reducibility, extended snapshot reducibility, and change preservation. We introduce two temporal primitives, *temporal splitter* and *temporal aligner*, and define rules that use these primitives to reduce the operators of a temporal algebra to their nontemporal counterparts. Our solution supports the three properties of the sequenced semantics through *interval adjustment* and *timestamp propagation*. We have implemented the temporal primitives and reduction rules in the kernel of PostgreSQL to get native database support for processing interval timestamped data. The support is comprehensive and includes outer joins, antijoins, and aggregations with predicates and functions over the time intervals of argument relations. The implementation and empirical evaluation confirms effectiveness and scalability of our solution that leverages existing database query optimization techniques.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Relational databases*

Keywords

Temporal databases, Sequenced semantics

1. INTRODUCTION

In order to query interval timestamped databases, temporal upward compatible, nonsequenced, and sequenced semantics exist [14, 7, 3]. Temporal upward compatible semantics [3][14, p.2936-2939] processes only the data that is valid at the current time, whereas nonsequenced semantics [14, p.1913-1915] treats time intervals as conventional attributes. For both semantics standard SQL can be used to query the database. Sequenced semantics is by far the most difficult to support. Various works have shown that the formulation of sequenced statements in standard SQL is complex and awkward [7, 13, 14, 21]. This paper proposes relational algebra primitives that provide support for the sequenced semantics, including outer joins, antijoins and aggregations with predicates and functions over the interval timestamps of argument relations.

Sequenced semantics comes with three properties: snapshot reducibility, which applies nontemporal statements to each snapshot of a temporal database; extended snapshot reducibility, which combines snapshot reducibility with the possibility to specify predicates and functions over the interval timestamps of argument relations; and change preservation, which preserves the changes defined by the start and end points of time intervals.

Example 1. Consider a hotel that has rooms to let. Room prices are fixed during winter and negotiated during summer. The hotel has three fixed-price categories: short term (1-2 months, high price), long term (3-7 months, lower price) and permanent (8-12 months, lowest price and no summer/winter fluctuation). Room prices are recorded in relation $\mathbf{P}$, where A is the daily price, Min and Max are minimum and maximum duration for the specific price category, and T is the time period during which the price is valid. For instance, tuple s_1 records that short term guests pay a price of 50 during the first 5 months of 2012. During the same period long terms guests pay a price of 40 (tuple s_2). Tuple s_3 is for permanent guests with a price of 30 that is valid for the entire year. Relation $\mathbf{R}$ records reservations, where N is the name of the guest

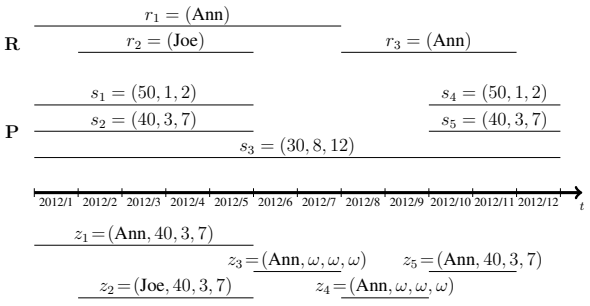
R

	N	T
r_1	Ann	$[2012/1, 2012/8)$
r_2	Joe	$[2012/2, 2012/6)$
r_3	Ann	$[2012/8, 2012/12)$

P

	A	Min	Max	T
s_1	50	1	2	$[2012/1, 2012/6)$
s_2	40	3	7	$[2012/1, 2012/6)$
s_3	30	8	12	$[2012/1, 2013/1)$
s_4	50	1	2	$[2012/10, 2013/1)$
s_5	40	3	7	$[2012/10, 2013/1)$

(a) Temporal Relations



(b) Result of Query $Q1$.

Figure 1: Sample Database.

and T is the time period of a reservation. For instance, r_1 records a reservation of Ann for the first 7 months of 2012. r_3 records a different reservation for Ann from August until November 2012.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

In order to determine periods with fixed prices and periods that need to be negotiated, the hotel computes a temporal left outer join: $Q1 = \mathbf{R} \mathbb{ⵠ}^{T}_{Min \leq DUR(\mathbf{R}.T) \leq Max} \mathbf{P}$. The result of query $Q1$ is shown in Fig. 1(b). We use a graphical representation, where timestamps are drawn as horizontal lines. For instance, $(\text{Ann}, 40, 3, 7, [2012/1, 2012/6))$ is produced from r_1 and s_1 over their common interval $[2012/1, 2012/6)$, and $(\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/8))$ from r_1 over the interval $[2012/6, 2012/8)$ for which the price must be negotiated (ω denotes a null value). Note that the join predicate references the timestamp attribute $\mathbf{R}.T$ (extended snapshot reducibility) and that z_3 and z_4 are not coalesced into a single result tuple since they are derived from different argument tuples (change preservation).

In order to satisfy the three properties of the sequenced semantics, we propose a solution that (1) adjusts timestamps by breaking them into pieces, (2) propagates timestamps as explicit attributes to support functions and predicates over these intervals, and (3) uses lineage information to preserve the changes defined by the interval timestamps of the argument tuples. It is easy to support each property individually. Supporting all three properties together, however, is difficult and is the goal pursued in this paper.

To adjust interval timestamps, we propose two primitives that transform each tuple of an argument relation into a set of tuples with adjusted timestamps. Based on the characteristics of how relational operators produce result tuples, we identify two classes of operators that have to be adjusted differently. For *group based* operators, $\{\pi, \vartheta, \cup, -, \cap\}$, all tuples in a group contribute to one result tuple. We define a *temporal splitter* to adjust interval timestamps for group based operators. For *tuple based* operators, $\{\sigma, \times, \bowtie, ⵠ, ⟕, ⟖, \triangleright\}$, at most one tuple of every argument relation contributes to a single result tuple. We define a *temporal aligner* to adjust intervals for tuple based operators. Once the argument tuples have been adjusted, the final result can be computed by comparing interval timestamps using equality and without further interval manipulations.

The purpose of the adjustment is to modify interval timestamps, so that all intervals that have to be compared are either identical or disjoint. After the adjustment the original interval timestamps are no longer available. To permit the use of predicates over the original interval timestamps, we provide the possibility to *propagate timestamps* as explicit attributes. Timestamp propagation is possible for *schema robust* operators, i.e., operators that are not affected if the schema of an argument relation is extended with additional attributes (cf. Section 3.3). Apart from the set operators, $\{\cup, -, \cap\}$, all relational algebra operators are schema robust. In relational algebra expressions interval timestamps can be propagated through sequences of schema robust operators and used in predicates and functions. They must be removed (using a projection) before operators that are not schema robust.

Interval adjustment, in combination with timestamp propagation, provides a uniform solution to reduce all operators of a temporal algebra with sequenced semantics to their nontemporal counterparts. With the adjustment primitives query processing becomes a two-step process: 1) propagate and adjust the interval timestamps of argument tuples; 2) apply the corresponding nontemporal operator on the interval-adjusted relations.

To summarize, we adopt an interval based temporal data model and propose an algebraic solution that provides support for the sequenced semantics with snapshot reducibility, extended snapshot reducibility, and change preservation. Our solution makes it unnecessary for applications to explicitly manipulate intervals: tuples that have to be compared by relational algebra operators have either equal or disjoint timestamps. The technical contributions are as follows:

- We introduce *timestamp propagation* as a mechanism to support extended snapshot reducibility for *schema robust* operators, i.e., operators that are not affected if the schema is extended.
- We define *lineage* for interval timestamped databases and show how to combine lineage with snapshot reducibility to define *change preservation*.
- We define a *temporal splitter* and a *temporal aligner* primitive to adjust the timestamp intervals of argument tuples. The temporal splitter adjusts the intervals for the group based operators, $\{\pi, \vartheta, \cup, -, \cap\}$, the temporal aligner for the tuple based operators, $\{\sigma, \times, \bowtie, ⵠ, ⟕, ⟖, \triangleright\}$.
- We define a temporal algebra with sequenced semantics by specifying a set of *reduction rules* that reduce temporal operators to the nontemporal counterparts. The reduction rules are comprehensive and cover all algebra operators, including outer joins, antijoins, and aggregations with predicates and functions over the timestamp attributes.
- We prove that the temporal algebra defined by the reduction rules is snapshot reducible, extended snapshot reducible, and change preserving.
- We describe an implementation of the temporal primitives and reduction rules in the kernel of PostgreSQL and conduct extensive experiments that show the effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the three properties of the sequenced semantics, including mechanisms and formalization to achieve them. In Sec. 4 we introduce temporal splitter and aligner, which in Sec. 5 are used to reduce the operators of a temporal algebra to their nontemporal counterparts. Section 6 describes the implementation of our solution in PostgreSQL. Section 7 reports the evaluation results. Section 8 concludes the paper and points to future work.

2. RELATED WORK

The management of temporal data in DBMSs has been an active research area since several decades, focusing primarily on temporal data models and query languages (e.g., [1, 5, 10, 11, 20][14, p.2762-2768]) as well as efficient evaluation algorithms for specific operators (e.g., temporal join [19, 24] and temporal aggregation [4, 28][14, p.2924-2929]).

To make the formulation of temporal queries more convenient, various temporal query languages [6, 7, 23] have been proposed. The earliest approach to add support for the time 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. Extending an existing query language with new data types is fairly simple and facilitates the formulation of some temporal queries. However, it does not provide a systematic way to generalize nontemporal to temporal queries since it does not effectively support, e.g., temporal aggregation and temporal set difference. In response to this new keywords and clauses were added to SQL with the goal of expressing temporal queries similar to nontemporal ones. Below we discuss the languages and techniques that are directly relevant to our solution. Note that the goal of our approach is not an extension of SQL, but native database

support for temporal operators at the level of the relational algebra. Our solution is generic and provides built-in database support for implementing the proposed extensions of SQL.

The IXSQL language [10, 15] 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 operator is applied on 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 preserve changes and an efficient implementation has not been provided.

An approach based on point timestamped relations is SQL/TP [26, 27]. A temporal relation is a sequence of nontemporal relations (or snapshots), and the corresponding nontemporal operations are applied on each of the snapshots to answer temporal queries. To provide an efficient evaluation of SQL/TP an interval based 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 a temporal splitter for group based operators and we leverage the normalization for the splitting of interval timestamps of group based operators. We propose a database internal algorithm for the normalization function for which no implementation was provided. SQL/TP does not consider change preservation. Also not considered is extended snapshot reducibility, which is not relevant for point timestamped relations. Normalization is not applicable to tuple based operators, such as joins, outer joins, and antijoins, since for these operators it would not preserve changes.

Agesen et al. [2] introduce a *split* operator that extends the 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 nontemporal operations are applied to the split relation. Similar to our temporal splitter, changes are preserved. The focus of the split operator are aggregation and difference in now-relative bitemporal databases. It is limited to value-equivalent tuples, i.e., tuples with pairwise identical nontemporal attributes, and does not apply to change preserving joins, outer joins, and antijoins.

ATSQL [7] 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 control if the statement is evaluated with temporal or nontemporal semantics. In the context of ATSQL different desiderata for temporal languages were formulated, namely upward compatible, temporal upward compatible, sequenced, and nonsequenced semantics. No native database implementation of this approach has been provided.

In terms of query processing various query evaluation algorithms for specific operators have been proposed. Join algorithms are based on indexing techniques [22, 31] or well-known nested loop, sort merge and partitioning strategies [9]. Similarly, several solutions for the evaluation of various forms of temporal aggregation [4, 12, 16, 29, 30] were proposed. Instead of designing algorithms for specific operators, we adopt a generic approach and propose two primitives that allow to reduce all operators of a sequenced algebra to their nontemporal counterparts.

The support for temporal data in commercial DBMSs has been limited to new data types with associated predicates and functions. In *PostgreSQL*, a temporal module [18] adds the PERIOD datatype for anchored time intervals together with boolean predicates and functions, such as intersection, union and minus. Since not all of them are closed, the functions might throw a runtime error. While this module facilitates the formulation of some temporal queries, it does not conveniently support queries that need to adjust the timestamps of tuples, such as temporal difference, aggregation and outer joins. The *Oracle* database system [17] extends the capabilities of PostgreSQL by additionally supporting valid and transaction time (DBMS_WM package). Querying temporal relations, however, is only possible at a specific time point (snapshot). *Teradata* [25] 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` in queries. The support for `SEQUENCED` is limited to inner joins. The sequenced semantics for outer joins, set operations, duplicate elimination and aggregation is not supported.

3. SEQUENCED SEMANTICS

This section presents the three properties of the sequenced semantics [14, p.2619-2621]: snapshot reducibility, extended snapshot reducibility and change preservation. For each property we provide crisp definitions that will be used to prove that our solution supports the sequenced semantics.

3.1 Preliminaries

We assume a linearly ordered, discrete time domain, Ω^T. A time interval is a contiguous set of time points (or instants) and is represented as a pair $[T_s, T_e)$, where T_s is the inclusive start point and T_e the exclusive end point. We use tuple timestamping and associate each tuple with a single time interval that represents the tuple's valid time. A temporal relation schema is represented as $R = (A_1, \ldots, A_m, T)$, where $A_1, \ldots, A_m$ are the nontemporal attributes with domain Ω_i and T is a temporal attribute over $\Omega^T \times \Omega^T$. Similarly, we assume a temporal relation $\mathbf{s}$ with schema $S = (C_1, \ldots, C_k, T)$. A tuple, r, over schema R is a finite set that contains for every A_i a value $v_i \in \Omega_i$ and for T a time interval $[t_s, t_e) \in \Omega^T \times \Omega^T$. A temporal relation, $\mathbf{r}$, over schema R is a finite set of tuples over R. For a tuple r and an attribute A_i, $r.A_i$ denotes the value of the attribute A_i in r. As abbreviations we use $\mathbf{A} = \{A_1, \ldots, A_m\}$ and $r.\mathbf{A} = (r.A_1, \ldots, r.A_m)$. The operators of the temporal relational algebra are 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 antijoin $\triangleright^T$. For the set operators we assume union compatible argument relations, and for $\pi^T_{\mathbf{B}}(\mathbf{r})$ and ${}_{\mathbf{B}}\vartheta^T_F(\mathbf{r})$ we require $\mathbf{B} \subseteq \mathbf{A}$. $sch(\psi)$ denotes the schema of the relation defined by the relational algebra expression ψ. We assume set-based semantics with duplicate free temporal relations, i.e., there are no value-equivalent tuples over common timepoints. Formally, a temporal relation, $\mathbf{r}$, is *duplicate free* iff $\forall r \in \mathbf{r} \forall r' \in \mathbf{r}(r \neq r' \Rightarrow r.\mathbf{A} \neq r'.\mathbf{A} \vee r.T \cap r'.T = \emptyset)$. A *snapshot* of a temporal relation is a nontemporal relation that is valid at a specific time point t, and is defined in terms of the timeslice operator [14, p.3120-3121]: $\tau_t(\mathbf{r}) = \{r.\mathbf{A} \mid r \in \mathbf{r} \wedge t \in r.T\}$.

3.2 Snapshot Reducibility

Many temporal languages [15, 23] are based on the concept of snapshot reducibility. Snapshot reducibility ensures that each snapshot in the result of a temporal operator (e.g., $⟕^T_\theta$) is equal to the

result of the equivalent nontemporal operator (e.g., $⟕_\theta$) evaluated on the corresponding snapshots of the argument relations.

Definition 1. (*Snapshot Reducibility*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, ψ^T an n-ary temporal operator, ψ the corresponding nontemporal operator, Ω^T the time domain and $\tau_p(\mathbf{r})$ the timeslice operator. Operator ψ^T is *snapshot reducible* to ψ iff

$$\forall t \in \Omega^T(\tau_t(\psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)) \equiv \psi(\tau_t(\mathbf{r}_1), \ldots, \tau_t(\mathbf{r}_n))).$$

Snapshot reducibility constrains the result of a temporal operator. Note that it does not define how to group time points into intervals, and the timestamps of the argument relations $\mathbf{r}_1, ..., \mathbf{r}_n$ cannot be used in theta conditions of ψ since the timestamps are removed by the timeslice operator (for instance, snapshot reducibility does not apply to $\vartheta^T_{AVG(DUR(R.T))}(\mathbf{R})$, which determines the average duration of reservations at each point in time).

3.3 Extended Snapshot Reducibility

As illustrated above, snapshot reducibility does not apply to temporal operators with predicates and functions over the interval timestamps of argument relations. The sequenced semantics introduces the concept of extended snapshot reducibility, which requires that references to interval timestamps can be used along with snapshot reducibility. We support extended snapshot reducibility by propagating interval timestamps as nontemporal attributes. Since timestamp propagation adds attributes to the schema of argument relations of an operator ψ, the operator must be unaffected if its argument relations is extended by an additional attribute, i.e., the operator must be *schema robust*.

Definition 2. (*Schema Robust Operator*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be relations, where relation $\mathbf{r}_i$ has schema $R_i = (\mathbf{A}_i)$, and ψ be an n-ary operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathbf{r}_1, \ldots, \mathbf{r}_n$. Let $\mathbf{r}'_1, \ldots \mathbf{r}'_n$ be relations where $\mathbf{r}'_i$ has schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$ and let $\mathbf{r}_i \equiv \pi_{\mathbf{A}_i}(\mathbf{r}'_i)$. Operator ψ is schema robust iff for all $\mathbf{X}_i$ and $\mathbf{r}_1, \ldots, \mathbf{r}_n$ the following holds:

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

Definition 3. (*Extend Operator*) Let $\mathbf{r}$ be a temporal relation with schema $(A_1, \ldots, A_m, T)$. The *extend operator*, $\epsilon_U(\mathbf{r})$, yields a temporal relation with schema $(A_1, \ldots, A_m, U, T)$ and is defined as follows:

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

Definition 4. (*Extended Snapshot Reducibility*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, ψ^T an n-ary schema robust temporal operator, and ψ the corresponding n-ary nontemporal operator that yields a relation with schema $\mathbf{E}$. Let Ω^T be the time domain and $\tau_p(\mathbf{r})$ be the timeslice operator. Operator ψ^T is *extended snapshot reducible* to ψ iff

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

where in predicates and functions on the right-hand side $\mathbf{r}_i.T$ has been substituted with U_i.

The crucial property of extended snapshot reducibility is that it allows references to timestamps by substituting them with references to explicit attributes that have been propagated for this purpose.

Example 2. Consider our running example in Fig. 1. We illustrate extended snapshot reducibility for timepoint 2012/1 and query $Q1 = \mathbf{R} ⟕^T_{Min \leq DUR(\mathbf{R}.T) \leq Max} \mathbf{P}$.

1. Propagate the timestamp of $\mathbf{R}$ by extending relation $\mathbf{R}$:

$\epsilon_U(\mathbf{R}) =$

	N	U	T
r_1	Ann	$[2012/1, 2012/8)$	$[2012/1, 2012/8)$
r_2	Joe	$[2012/2, 2012/6)$	$[2012/2, 2012/6)$
r_3	Ann	$[2012/8, 2012/12)$	$[2012/8, 2012/12)$

2. Determine snapshots at timepoint 2012/1:

$$\tau_{2012/1}(\epsilon_U(\mathbf{R})) = \{(\text{Ann}, [2012/1, 2012/8))\},$$
$$\tau_{2012/1}(\mathbf{P}) = \{(50, 1, 2), (40, 3, 7), (30, 8, 12)\}.$$

3. Substitute $\mathbf{R}.T$ in the condition of the left outer join with U and compute a nontemporal left outer join:

$$\tau_{2012/1}(\epsilon_U(\mathbf{R})) ⟕_{Min \leq DUR(U) \leq Max} \tau_{2012/1}(\mathbf{P}) = \\ \{(\text{Ann}, 40, 3, 7, [2012/1, 2012/8))\}.$$

4. Project on (N, A, Min, Max): $\{(\text{Ann}, 40, 3, 7)\}$.

For the construction of relational algebra expressions it is also relevant if an operator is *schema propagating* or not. For instance, all types of joins are schema robust as well as schema propagating. Temporal aggregation is schema robust but not timestamp propagating since a single result tuple is not derived from a fixed number of argument tuples.

Definition 5. (*Timestamp Propagating Operator*) Let $\mathbf{r}_1, \ldots,$ $\mathbf{r}_n$ be relations where relation $\mathbf{r}_i$ has schema $R_i = (\mathbf{A}_i)$, ψ an n-ary schema robust operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathbf{r}_1, \ldots, \mathbf{r}_n$, and $\mathbf{r}'_1, \ldots \mathbf{r}'_n$ relations where $\mathbf{r}'_i$ has schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$. Operator ψ is *timestamp propagating* iff

$$sch(\psi(\mathbf{r}_1, \ldots, \mathbf{r}_n)) = (\mathbf{E}) \\ \Rightarrow sch(\psi(\mathbf{r}'_1, \ldots, \mathbf{r}'_n)) = (\mathbf{E}, \mathbf{X}_1, \ldots, \mathbf{X}_n)$$

Table 1 summarizes schema robust and timestamp propagating operators, respectively.

Table 1: Properties of Operators.

Operators	Schema robust	Timestamp propagating
$\{\sigma, \times, ⋈, ⟕, ⟖, ⟗, \triangleright\}$	yes	yes
$\{\pi, \vartheta\}$	yes	no
$\{-, \cap, \cup\}$	no	no

3.4 Change Preservation

Data lineage [14, p.2202-2207][8] traces how result tuples are derived from argument tuples and has been studied in contexts where the relationship between argument and result tuples is relevant. We show that data lineage nicely complements snapshot reducibility and can be used to define a natural and unique grouping of time points into intervals that is change preserving.[1] For instance, given the result of query $Q1$ in Fig. 1(b) between 2012/6 and 2012/9, (extended) snapshot reducibility only defines that at each of these timepoints a tuple with values $(\text{Ann}, \omega, \omega, \omega)$ must exist. That means that replacing tuples z_3 and z_4 by a single tuple $z_{34} = (\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/10))$, or four tuples over one month each would not violate (extended) snapshot reducibility.

[1]Originally, when introduced in the context of the sequenced semantics [7][14, p.2619-2621], this property was termed interval preservation. We use the term change preservation, which better captures its nature.

Definition 6. (*Lineage Set*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations and $z \in \psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)$ be a result tuple at timepoint t of an n-ary (extended) snapshot reducible temporal operator ψ^T. The *lineage set*, $\mathsf{L}[\psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)](z, t)$, of result tuple z at time point t is the list of sets of argument tuples, $\langle \mathbf{r}'_1, \ldots, \mathbf{r}'_n \rangle$, $\mathbf{r}'_i \subseteq \mathbf{r}_i$ from which z is derived:

$$\mathsf{L}[\sigma^T_\theta(\mathbf{r})](z,t) = \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge \theta(r) \wedge t \in r.T\} \rangle$$

$$\mathsf{L}[\pi^T_{\mathbf{B}}(\mathbf{r})](z,t) = \langle \{r \in \mathbf{r} \mid z.\mathbf{B} = r.\mathbf{B} \wedge t \in r.T\} \rangle$$

$$\mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t) = \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \mathbf{s} \rangle$$

$$\mathsf{L}[\mathbf{r} \cup^T \mathbf{s}](z,t) = \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \{s \in \mathbf{s} \mid z.\mathbf{A} = s.\mathbf{C} \wedge t \in s.T\} \rangle$$

$$\mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t) = \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \{s \in \mathbf{s} \mid z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\} \rangle$$

$$\mathsf{L}[\mathbf{r} \ltimes^T_\theta \mathbf{s}](z,t) = \begin{cases} \mathsf{L}[\mathbf{r} \rhd^T_\theta \mathbf{s}](z,t) & \text{if } z.\mathbf{C} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases}$$

$$\mathsf{L}[\mathbf{r} \rtimes^T_\theta \mathbf{s}](z,t) = \begin{cases} \mathsf{L}[\mathbf{s} \rhd^T_\theta \mathbf{r}](z,t) & \text{if } z.\mathbf{A} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases}$$

$$\mathsf{L}[\mathbf{r} \ltimes\!\rtimes^T_\theta \mathbf{s}](z,t) = \begin{cases} \mathsf{L}[\mathbf{s} \rhd^T_\theta \mathbf{r}](z,t) & \text{if } z.\mathbf{A} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \rhd^T_\theta \mathbf{s}](z,t) & \text{if } z.\mathbf{C} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases}$$

Example 3. Consider the temporal left outer join in Example 1 with the result from Fig. 1(b):

- $\mathsf{L}[\mathbf{R} \ltimes^T_\theta \mathbf{P}](z_1, 2012/2) = \langle \{r_1\}, \{s_2\} \rangle$,
- $\mathsf{L}[\mathbf{R} \ltimes^T_\theta \mathbf{P}](z_3, 2012/6) = \langle \{r_1\}, \{s_1, s_2, s_3, s_4, s_5\} \rangle$.

Note that lineage sets for inner join, aggregation, intersection and antijoin are not listed explicitly in Def. 6 since they are identical to, respectively, Cartesian product, projection, union and difference (e.g., $\mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t)$). The definitions are identical since the specifics of the operators, e.g., theta conditions, are part of the definition of the result tuples z. The result tuples are defined through (extended) snapshot reducibility, which includes the theta conditions, and the result tuples are arguments in the definition of the lineage set. In the following we omit the operator and write $\mathsf{L}(z,t)$ if we discuss general properties of lineage sets.

Similar to nontemporal operators [8], the lineage set for temporal operators has three properties: (i) when an operator is applied to the lineage set, $\mathsf{L}(z,t)$, its result at time t is identical to the snapshot of tuple z at time t, (ii) each tuple in the lineage set contributes to the result tuple and (iii) the lineage set is maximal.

Lineage sets trace the result tuples at a time point back to the argument tuples. Together with (extended) snapshot reducibility they define the result of a temporal operator. By merging contiguous time points with identical lineage sets we obtain result tuples over maximal time intervals that preserve changes.

Definition 7. (*Change Preservation*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, $\mathbf{z} = \psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)$ be the result of an n-ary temporal operator ψ^T, and let $\mathsf{L}(z,p)$ be the lineage set of result tuple $z \in \mathbf{z}$ at timepoint t. The temporal operator, ψ^T, is *change preserving* iff for all $z \in \mathbf{z}$ and $z' \in \mathbf{z}$ the following holds:

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

Example 4. Consider the temporal left outer join in Fig. 1(b). For result tuples z_3 and z_4 we have the following lineage sets:

- $\forall p \in z_3.T$: $\mathsf{L}[\mathbf{R} \ltimes^T_\theta \mathbf{P}](z_3, p) = \langle \{r_1\}, \mathbf{P} \rangle$
- $\forall p' \in z_4.T$: $\mathsf{L}[\mathbf{R} \ltimes^T_\theta \mathbf{P}](z_4, p') = \langle \{r_3\}, \mathbf{P} \rangle$

The change at time $2012/8$ where one reservation of Ann ends and a different reservation of Ann starts is preserved. Any result relation with more tuples over smaller time intervals would not preserve changes. For example, replacing z_3 in Fig. 1(b) by $z'_3 = (\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/7))$ and $z''_3 = (\text{Ann}, \omega, \omega, \omega, [2012/7, 2012/8))$ violates the maximality constraint since their lineage sets would be equal and the two tuples are value-equivalent and adjacent.

4. TEMPORAL PRIMITIVES

This section introduces two temporal primitives that will be used in Sec. 5 to reduce the operators of a temporal algebra to operators of the nontemporal relational algebra, while preserving the three properties of the sequenced semantics.

Based on the characteristics of how operators produce result tuples, we identify group and tuple based operators. *Group based* operators are $\{\pi, \vartheta, \cup, -, \cap\}$. All tuples with identical values for the (grouping) attributes contribute to one result tuple. *Tuple based* operators are $\{\sigma, \times, \bowtie, \ltimes, \ltimes\!\rtimes, \rtimes, \rhd\}$. For these operators at most one tuple of every argument relation contributes to the values of a single result tuple. For each operator class we design a temporal primitive that provides both equality on the adjusted timestamps and change preservation for the subsequent nontemporal operation.

4.1 Temporal Splitter

For group based operators, $\{\pi, \vartheta, \cup, -, \cap\}$, we propose a temporal splitter primitive that adjusts the time interval of an argument tuple by splitting it at each start and end point of all tuples in the same group.

Definition 8. (Temporal Splitter) Let r be a tuple and $\mathbf{g}$ a set of tuples. A *temporal splitter* produces a set of tuples with the nontemporal attributes of r over the following adjusted intervals:

$$T \in \mathit{split}(r, \mathbf{g}) \iff$$
$$T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset \vee T \subseteq g.T) \wedge$$
$$\forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset \wedge T' \not\subseteq g.T) \vee T' \not\subseteq r.T).$$

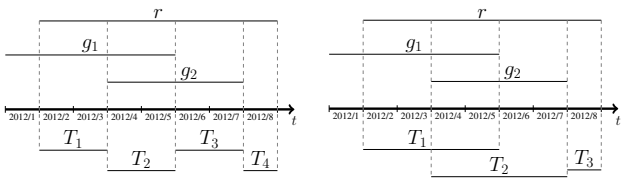
The second line requires that an adjusted interval, T, is contained in r's timestamp and either contained or disjoint from all timestamp intervals of tuples $g \in \mathbf{g}$. The third line requires that T is maximal, i.e., it cannot be enlarged without violating the first condition.

Example 5. Figure 2(a) illustrates the temporal splitter with $\mathbf{g} = \{g_1, g_2\}$. The temporal splitter produces maximal sub-intervals of r's timestamp that are contained in the intervals of all overlapping tuples.

A temporal primitive that satisfies the properties of a temporal splitter is the normalization function of Toman [27].

Definition 9. (*Normalization [27, Sec. 4]*) Let $\mathbf{r}$ be a temporal relation. The *normalization*, $\mathcal{N}_{\mathbf{B}}(\mathbf{r}; \mathbf{s})$, of $\mathbf{r}$ with respect to $\mathbf{s}$ and attributes $\mathbf{B} \subseteq \mathbf{r}.\mathbf{A}$ is defined as follows:

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



(a) Temporal Splitter. (b) Temporal Aligner.

Figure 2: Temporal Splitter and Aligner.

PROPOSITION 1. *Assume a temporal relation* $\mathbf{r}$ *and the temporal normalization* $\tilde{\mathbf{r}} = \mathcal{N}_{\mathbf{B}}(\mathbf{r};\mathbf{r})$. *All tuples* $\tilde{r} \in \tilde{\mathbf{r}}$ *with the same* $\mathbf{B}$*-values have interval timestamps that are either equal or disjoint.*

PROPOSITION 2. *Assume temporal relations* $\mathbf{r}$ *and* $\mathbf{s}$ *with schema* $(\mathbf{A}, T)$ *and the temporal normalizations* $\tilde{\mathbf{r}} = \mathcal{N}_{\mathbf{A}}(\mathbf{r};\mathbf{s})$ *and* $\tilde{\mathbf{s}} = \mathcal{N}_{\mathbf{A}}(\mathbf{s};\mathbf{r})$. *Any two tuples* $\tilde{r} \in \tilde{\mathbf{r}}$ *and* $\tilde{s} \in \tilde{\mathbf{s}}$ *with the same* $\mathbf{A}$*-values have interval timestamps that are either equal or disjoint.*

Example 6. Figure 3 illustrates the temporal normalization $\mathcal{N}_{\{\}}(\mathbf{R};\mathbf{R})$ for relation $\mathbf{R}$ from Example 1. For instance, tuple $(\text{Ann}, [2012/2, 2012/6))$ is derived from r_1 over a maximal sub-interval that is either identical or disjoint from the intervals of all other result tuples.

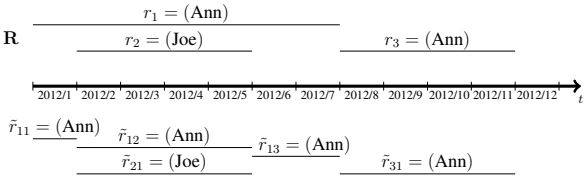


Figure 3: Temporal Normalization.

4.2 Temporal Aligner

For tuple based operators, $\{\sigma, \times, \bowtie, ⟕, ⟖, ⟗, \triangleright\}$, we propose a temporal aligner primitive that adjusts an argument tuple according to each tuple of a group.

Definition 10. (Temporal Aligner) Let r be a tuple and $\mathbf{g}$ be a set of tuples. A *temporal aligner* produces a set of tuples with the nontemporal attributes of r over the following adjusted intervals:

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

The second line handles all possible sub-intervals of $r.T$ for which a timestamp interval in $\mathbf{g}$ exists: in this case T is their intersection. The third and fourth lines handle sub-intervals for which no covering interval in $\mathbf{g}$ exists: in this case T is a maximal non-covered part of $r.T$.

Example 7. Figure 2(b) illustrates the temporal aligner with $\mathbf{g} = \{g_1, g_2\}$. The time intervals T_1 and T_2 are derived from the intersection of r with g_1 and g_2, respectively. T_3 is a sub-interval of $r.T$ that is not covered by any tuple in $\mathbf{g}$.

Definition 11. (*Temporal Alignment*) Let $\mathbf{r}$ and $\mathbf{s}$ be two temporal relations and θ be a predicate over the nontemporal attributes of a tuple in $\mathbf{r}$ and a tuple in $\mathbf{s}$. The *temporal alignment* operator, $\mathbf{r}\Phi_\theta\mathbf{s}$, of $\mathbf{r}$ with respect to $\mathbf{s}$ and condition θ is defined as follows:

$$
\begin{aligned}
&\tilde{r} \in \mathbf{r}\Phi_\theta\mathbf{s} \iff \\
&\quad \exists r \in \mathbf{r}(\tilde{r}.A = r.A \wedge \tilde{r}.T \in align(r, \{s \in \mathbf{s} \mid \theta(r,s)\})).
\end{aligned}
$$

Example 8. Figure 4 shows the alignment of $\mathbf{P}$ with respect to $\epsilon_U(\mathbf{R})$ using condition $\theta \equiv (Min \leq DUR(U) \leq Max)$. For instance, the first result tuple, $(50, 1, 2, [2012/1, 2012/6))$, is derived from s_1 over the interval $[2012/1, 2012/6)$ for which no tuple in the other relation exists that satisfies θ. The second result tuple, $(40, 3, 7, [2012/1, 2012/6))$, is derived from s_2 and r_1 over their common interval, and the third result tuple, $(40, 3, 7, [2012/2, 2012/6))$, from s_2 and r_2 over their common interval. Notice that the second and third tuple are value-equivalent over overlapping timepoints and are both derived from tuple s_2.

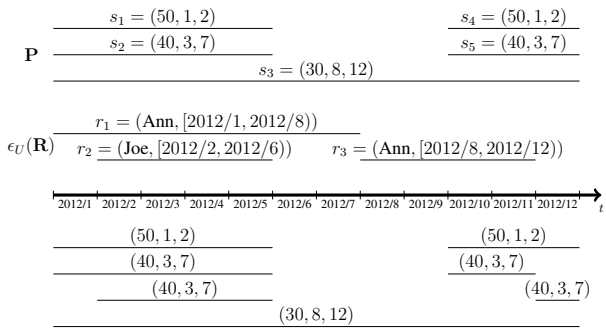


Figure 4: Temporal Alignment.

LEMMA 1. *Let* $\mathbf{r}$ *be a temporal relation with* $|\mathbf{r}| = n$, $\mathbf{s}$ *be a temporal relation with* $|\mathbf{s}| = m$, *and* $\tilde{\mathbf{r}} = \mathbf{r}\Phi_\theta\mathbf{s}$ *be the result of temporal alignment with condition* θ. *The upper bound of the cardinality of the aligned relation is* $|\tilde{\mathbf{r}}| \leq 2nm + n$.

PROOF. By induction. Base case: $n = 1$. The result of unifying a relation $\mathbf{r} = \{r_1\}$ with a relation $\mathbf{s} = \{s_1, \ldots, s_m\}$ generates at most $2m + 1$ result tuples. There exist at most m sub-intervals of $r_1.T$ that overlap with a tuple in $\mathbf{s}$ and at most $m + 1$ sub-intervals of $r_1.T$ that do not overlap with any tuple in $\mathbf{s}$. This is illustrated in Fig. 5 for $n = 1$ and $m = 2$, where r_1 is split into $2 * 2 + 1 = 5$ result tuples. Inductive case: $n > 1$. Assume an argument relation $\mathbf{r}$ with n tuples that can have up to $2nm + n$ output tuples. Then $n+1$ tuples in the argument relation can produce $2(n+1)m+(n+1)$ tuples. This holds since $2mn + n$ tuples can be produced by n argument tuples and an additional tuple can yield up to $2m+1$ new result tuples. □

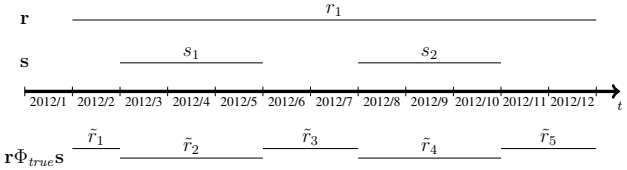


Figure 5: Base Case (for $n = 1$ and $m = 2$).

PROPOSITION 3. *Assume temporal relations* $\mathbf{r}$ *and* $\mathbf{s}$ *with alignments* $\tilde{\mathbf{r}} = \mathbf{r}\Phi_\theta\mathbf{s}$ *and* $\tilde{\mathbf{s}} = \mathbf{s}\Phi_\theta\mathbf{r}$. *For any two tuples* $r \in \mathbf{r}$ *and*

$s \in \mathbf{s}$ that satisfy θ and $r.T \cap s.T \neq \emptyset$, there are two tuples $\tilde{r} \in \tilde{\mathbf{r}}$ and $\tilde{s} \in \tilde{\mathbf{s}}$ with matching nontemporal values for r and s, respectively, and with identical timestamps $\tilde{r}.T = \tilde{s}.T = r.T \cap s.T$.

PROPOSITION 4. *Assume temporal relations $\mathbf{r}$ and $\mathbf{s}$. Every tuple $\tilde{r} \in \mathbf{r}\Phi_\theta\mathbf{s}$ is derived from a tuple $r \in \mathbf{r}$, and the timestamp of $\tilde{r}$ is either the intersection of $r.T$ with the timestamp of a tuple $s \in \mathbf{s}$ satisfying θ, or a maximal sub-interval of $r.T$ that is not covered by the interval timestamp of a tuple $s \in \mathbf{s}$ satisfying θ.*

5. REDUCING TEMPORAL OPERATORS

This section uses temporal splitter and aligner to reduce operators with sequenced semantics to their nontemporal counterparts.

5.1 Overview

Figure 6 illustrates the basic scheme for reducing temporal operators with sequenced semantics. We assume extended relations (cf. Sec. 3.3). Thus, all references to timestamps have been substituted with references to explicit attributes with propagated timestamps.

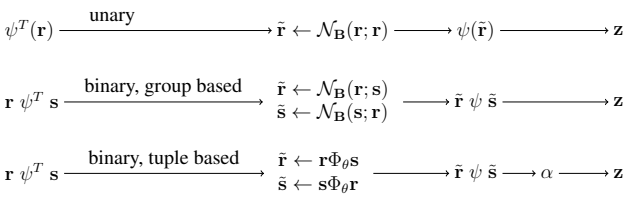


Figure 6: Reduction of Temporal Operators.

The normalization or alignment primitive transforms argument relation(s) with overlapping timestamps into temporal relations where the timestamps of tuples have been adjusted. Only equality is required to compare such timestamps. This allows to replace the temporal operator by the corresponding nontemporal operator on adjusted relations and an equality on the timestamps.

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

Example 9. Consider the Cartesian product of relations $\mathbf{r} = \{(a,[1,9)),(b,[3,7))\}$ and $\mathbf{s} = \{(c,[1,9)),(d,[3,7))\}$. The temporal alignment produces $\tilde{\mathbf{r}} = \mathbf{r}\Phi_{true}\mathbf{s} = \{(a,[1,9)),(a,[3,7)),(b,[3,7))\}$. Similar for $\mathbf{s}$ we get $\tilde{\mathbf{s}} = \{(c,[1,9)),(c,[3,7)),(d,[3,7))\}$. The subsequent equality join of $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{s}}$ on the adjusted timestamp attributes (cf. reduction rule for the Cartesian product in Table 2) gives:

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 is a temporal duplicate of z_1. Note that we cannot remove $\tilde{r}_2$ or $\tilde{s}_2$ before the join, since these tuples are required to produce tuples z_3 and z_4, respectively. Instead, the absorb operator removes temporal duplicates in a post-processing step.

Definition 12. (*Absorb Operator*) Let $\mathbf{r}$ be a temporal relation with timestamp attribute T. The *absorb* operator, α, eliminates all tuples $r \in \mathbf{r}$ for which another value-equivalent tuple $r' \in \mathbf{r}$ exists such that $r.T \subset r'.T$:

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

5.2 Reduction Rules

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

THEOREM 1. *Let $\mathbf{r}$ and $\mathbf{s}$ be temporal relations, θ be a predicate, F be a set of aggregation functions over $\mathbf{r}.\mathbf{A}$, $\mathbf{B} \subseteq \mathbf{A}$ be a set of attributes and α be the absorb operator. The reduction rules in Table 2 define a temporal algebra with sequenced semantics.*

Table 2: Reduction Rules.

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

See Appendix for the proof of Theorem 1.

Example 10. Figure 7 illustrates the reduction of the temporal aggregation query $Q2 = \vartheta^T_{AVG(DUR(\mathbf{R}.T))}(\mathbf{R})$. The query determines the average duration of reservations at each timepoint. Since there is a function with a reference to a timestamp the query is governed by extended snapshot reducibility and we first extend $\mathbf{R}$ to $\epsilon_U(\mathbf{R})$ and substitute $\mathbf{R}.T$ in $Q2$ with U. Next we normalize $\epsilon_U(\mathbf{R})$ to get tuples with timestamps that are identical or disjoint. Finally, we apply the reduced query to get the desired result.

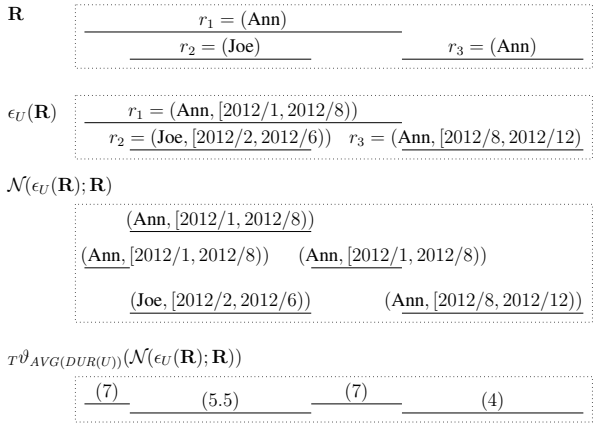


Figure 7: Reduction of Query Q2.

6. IMPLEMENTATION

This section describes the implementation of the temporal primitives in the kernel of the PostgreSQL database system.[2] We

[2] `http://www.ifi.uzh.ch/dbtg/research/align.html`.

modified parser and parse tree, analyzer and query tree, optimizer and plan tree, and executor and execution tree. For each tree a new custom node was defined that stores information for processing the new operator. In the query processing sequence transformations between these nodes were implemented: 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. The optimizer needs cost estimations for the new operator, and in the executor module three functions were implemented: `ExecInit⟨Operator⟩`, `Exec⟨Operator⟩` and `ExecEnd⟨Operator⟩` for initialization, execution and finalization of the evaluation algorithm, respectively, where `⟨Operator⟩` is the name of the actual execution algorithm.

To illustrate and evaluate the reduction rules, we extended SQL with the two temporal primitives. Note that this is just for illustration purposes and we do not propose a new temporal SQL. Instead, our primitives are useful building blocks that support the implementation of the temporal SQL extensions that have been proposed in the past.

6.1 Execution Algorithm for Temporal Alignment

The implementation of temporal alignment is a two step process: (1) we retrieve for each tuple $r_i \in \mathbf{r}$ the group $\mathbf{g}_i \subseteq \mathbf{s}$ of s-tuples that satisfy θ and (2) we apply a plane sweep algorithm on each sorted group $\mathbf{g}_i$ to produce the aligned relation.

First, we construct for each r-tuple the group $\mathbf{g}_i$ of matching s-tuples using a database internal left outer join. To illustrate our implementation, we assume two relations $\mathbf{r}$ and $\mathbf{s}$ with three tuples each and $\theta \equiv (B = D \wedge \mathbf{r}.T \cap \mathbf{s}.T \neq \emptyset)$ as illustrated in Fig. 8. r_1 matches two, and r_2 three s-tuples; r_3 does not match any s-tuple, hence the s-part is filled with ω values. Note that the join tuples have two timestamp attributes, from the r-tuple and the s-tuple, respectively.

r

	A	B	T
r_1	a	β	$[1,7)$
r_2	b	β	$[3,9)$
r_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 $\mathbb{M}_\theta$ **s**

r_1	s_1
r_2	s_3
r_1	s_2
r_2	s_2
r_3	ω
r_2	s_1

Figure 8: Join of r-tuples with s-tuples.

Our implementation supports pipelining such that intermediate results do not have to be materialized. To make this possible the join is partitioned according to the groups and within each group sorted according to the intersection timestamp of the r and s-tuple. This ensures that tuples with equal intersection timestamps are consecutive and allows to identify (and remove) duplicate timestamps during the plane sweep. Figure 9 illustrates the group construction after partitioning and sorting (the sorting in each group is displayed top down; the nearby lines show the two timestamps of join tuples).

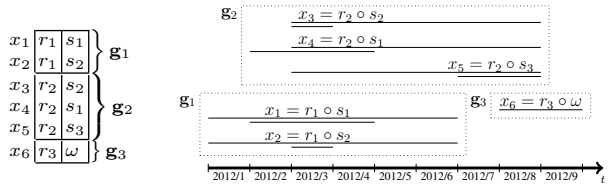


Figure 9: Partitioning and Sorting of Groups.

The plane sweep algorithm in Fig. 10 is implemented in PostgreSQL as executor function `ExecAdjustment`. The function is integrated into the pipelining architecture of PostgreSQL and on each invocation either a single result tuple is returned, or ω to indicate the end of the operation. The input is a context node, n, that keeps variables between consecutive invocations. Node n stores the following information: the reference to its input ($subnode$), the previous and current tuples from the input ($prev$, $curr$), the sweepline status ($sweepline$), an output tuple (out), the boolean *isalign* to distinguish alignment and normalization, and a boolean variable ($sameleft$) that is $true$ whenever $prev$ and $curr$ contain tuples from the same group and $false$ otherwise. $[P_1, P_2)$ denotes the already computed intersection of the r- and s-tuple.

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

Copy variables of n to local;
if *first call* **then**
  $prev \leftarrow curr \leftarrow$ next tuple from $subnode$;
  $sameleft \leftarrow true$;
  $sweepline \leftarrow curr.T_s$;
$produced = false$;
while $produced = false \wedge prev \neq \omega$ **do**
  if $sameleft \wedge sweepline < curr.P_1$ **then**
    $out \leftarrow (curr.\mathbf{A}, [sweepline, curr.P_1))$;
    $produced \leftarrow true$;
    $sweepline \leftarrow curr.P_1$;
  else if $sameleft \wedge sweepline \geq curr.P_1$ **then**
    if $isalign \wedge out \neq (curr.\mathbf{A}, curr.P_1, curr.P_2)$ **then**
      $out \leftarrow (curr.\mathbf{A}, [curr.P_1, curr.P_2))$;
      $sweepline \leftarrow max(sweepline, curr.P_2)$;
      $produced \leftarrow true$;
    $prev \leftarrow curr$;
    $curr \leftarrow$ next tuple from $subnode$;
    $sameleft \leftarrow prev.\mathbf{A} = curr.\mathbf{A} \wedge prev.T = curr.T$;
  else
    if $sweepline < prev.T_e$ **then**
      $out \leftarrow (prev.\mathbf{A}, [sweepline, prev.T_e))$;
      $produced \leftarrow true$;
    $prev \leftarrow curr$;
    $sweepline \leftarrow curr.T_s$;
    $sameleft \leftarrow true$;
if $produced = false$ **then** $out \leftarrow \omega$;
Copy local variables to n;
return out;

Figure 10: Executor Function.

Figure 11 illustrates four invocations of `ExecAdjustment` with four result tuples. Whenever a new group starts, $curr$ and

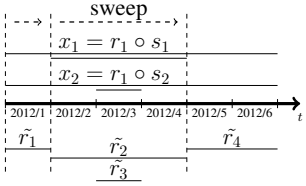


Figure 11: Plane Sweep Algorithm for Group $\mathbf{g}_1$.

$prev$ store the same input tuple, $sameleft$ is set to $true$ and $sweepline$ stores the r-tuple's starting timepoint. On the first invocation x_1 is fetched. $sameleft = true$ and $P_1 = 2012/2$ is larger than the $sweepline = 2012/1$. Thus, tuple $\tilde{r}_1$ is produced and the $sweepline$ is advanced to P_1 (first block of the function). On the second invocation, $sameleft = true$ and $sweepline = P_1$, hence the second block of the function is entered. We check if the

same intersection has already been produced before. Since this is not the case, $\tilde{r}_2$ is produced, the sweepline is advanced to 2012/4, $curr$ is copied to $prev$, and the next tuple x_2 is fetched into $curr$. Since x_2 belongs to the same group as x_1, $sameleft$ is set to $true$. On the third invocation, $sameleft = true$ and $sweepline > P_1$ (= 2012/3). The execution enters again in 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, $sameleft$ is set to $false$. On the fourth invocation, $sameleft = false$ and the execution enters the third block 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. Since this is not the case, a result tuple over the remaining part of the timestamp is produced ($\tilde{r}_4$). The variables are reset for processing the next group.

6.2 Extensions to Parser, Analyzer and Optimizer for Temporal Alignment

This section describes the extensions that are required in the three modules that precede the executor. First, we add a new SQL keyword ALIGN and extend the grammar of the parser:

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

The alignment statement consists of two `table_ref` and can be used similar to any other item in the FROM clause. The first `table_ref` argument is the relation to align, the second one is the reference relation; `a_expr` is the θ condition. In the select clause `ABSORB` can be specified instead of `DISTINCT` to eliminate temporal duplicates. For instance, query $Q1$ can be formulated in SQL as:

```
WITH R AS (SELECT Ts Us, Te Ue, * FROM R)
SELECT ABSORB n, a, min, max, r.Ts, r.Te
FROM (R ALIGN P ON DUR(Us,Ue) BETWEEN Min AND Max) r
     LEFT OUTER JOIN
     (P ALIGN R ON DUR(Us,Ue) BETWEEN Min AND Max) p
     ON DUR(Us,Ue) BETWEEN Min AND Max AND
        r.Ts=p.Ts AND r.Te=p.Te
```

The `WITH` statement does the timestamp propagation and the `SELECT` statement implements the reduction rule for the temporal left outer join (cf. Table 2). `DUR` is a user defined function (UDF) that evaluates the duration of the period defined by the two parameters. The corresponding RA expression and parse tree are shown in Fig. 12(a).

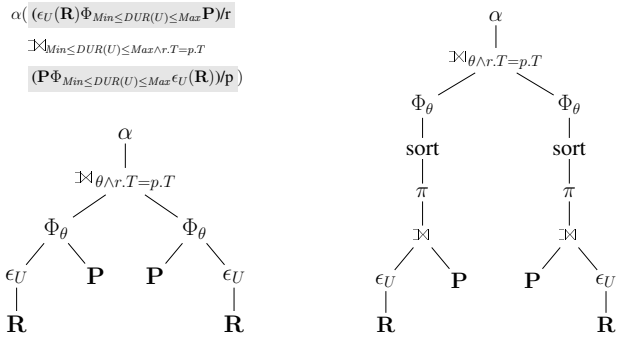
In the analyzer we extend the query tree with the partitioning and sorting of the groups. The query tree for our example is shown in Fig. 12(b). The optimizer is the final state before the executor. Here the database system chooses among different execution strategies. The cost estimations for our temporal alignment node, where $\mathbf{x}$ is the direct subnode, are as follows:

$$numRows = 3 * \mathbf{x}.numRows$$
$$cost = \mathbf{x}.cost + 2 * cpu\_op\_cost * \mathbf{x}.numRows * numCols$$
$$sortOrder = (\mathbf{A}, T)$$

The cardinality of the output can be up to three times the cardinality of the subnode, that is every tuple in the input can produce up to three tuples in the algorithm. The total cost is estimated by the cost of the subnode, plus two tuple comparison for each result tuple in the executor function. The result is sorted on all attributes.

6.3 Temporal Normalization

The approach to implement temporal normalization is similar to the implementation of temporal alignment. It differs in the construction of the groups. Temporal normalization splits a tuple's interval according to all start and end timepoints in its group. To build the group we use a database internal nontemporal left outer join. We impose a total order on split points to get a plane sweep algorithm with constant memory complexity. Therefore we do not join with the $\mathbf{s}$ relation directly but with the union of its start- and endpoints, i.e., $\pi_{B,T_s/P_1}(\mathbf{s}) \cup \pi_{B,T_e/P_1}(\mathbf{s})$. We build the groups as for alignment, sort on the split point P_1, and use the same plane sweep algorithm 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 final result.



(a) Parse Tree (b) Query Tree

Figure 12: Parse Tree and Query Tree.

The rules for the parser are similar to temporal alignment, but we use the keyword `NORMALIZE` with a list of grouping attributes instead of a θ condition. For instance, the temporal aggregation ${}_T\vartheta_{AVG(DUR(U))}(\mathcal{N}_{\{\}}(\epsilon_U(\mathbf{R}); \epsilon_U(\mathbf{R})))$ is formulated in SQL as:

```
WITH R AS (SELECT Ts Us, Te Ue, * FROM R)
SELECT AVG(DUR(Us,Ue)), Ts, Te
FROM (R R1 NORMALIZE R R2 USING()) r
GROUP BY Ts, Te
```

In the `USING` clause the grouping attributes are specified (empty in this example). In the optimizer we use the following cost estimations:

$$numRows = 2 * \mathbf{x}.numRows$$
$$cost = \mathbf{x}.cost + cpu\_op\_cost * \mathbf{x}.numRows * numCols$$
$$sortOrder = (\mathbf{A}, T)$$

For each split point in the subnode we can have up to two result tuples. The total cost is the cost of the subnode plus one tuple comparison for every output tuple (different from alignment since we omit the intersection part).

7. EMPIRICAL EVALUATION

In this section we evaluate our implementation of temporal normalization and temporal alignment, by showing that (1) our implementation is tightly integrated into the database kernel and leverages existing database optimization techniques; (2) temporal normalization with change preservation minimizes the number of splits, which keeps intermediate results small; and (3) temporal alignment remains stable for datasets that are inefficient to process with other approaches.

7.1 Setup

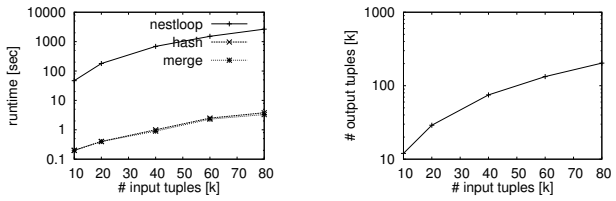
For the experiments the client and the database server run on the same 2.6 GHz machine with 4 GB RAM and a hard disk rotational

rate of 5400 rpm. We use the PostgreSQL server 9.0, extended with our implementation of normalize and alignment. All parameters of the PostgreSQL server, such as maximum memory for sorting, were kept to default values, and no indexes were used.

We use the real world dataset $Incumben$ of the University of Arizona with 83,857 entries. Each entry records a job assignment (pcn) for an employee (ssn) over a specific time interval. The data ranges over 16 years and contains 49,195 different employees. The timestamps are recorded at the granularity of days and range from 1 to 573 days with an average of approximately 180 days. Synthetic datasets used in the evaluation are described below.

7.2 Database System Integration

Temporal normalization and temporal alignment fully leverage existing database optimization strategies and algorithms. The nontemporal left outer join used for the group construction in our implementation is optimized by the database system. This applies to both temporal normalization and temporal alignment. We illustrate this for temporal normalization $\mathcal{N}_{\{ssn\}}$ of the $Incumben$ dataset, running the database in 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 suited 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 is used. Figure 13(a) shows the runtime of the normalization, which



(a) Runtime (b) Output Size

Figure 13: Normalization $\mathcal{N}_{\{ssn\}}$ ($Incumben$).

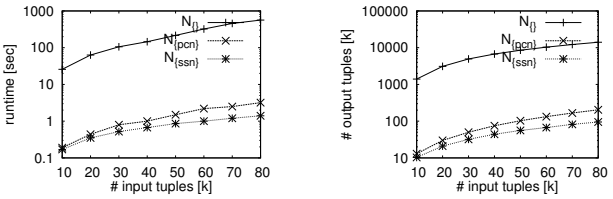
is dominated by the join, for which the DBMS chooses the best available join algorithm. The same holds for temporal alignment. Hence, the runtime of normalization and alignment is proportional to the runtime of a join. The output cardinality of the normalization is shown in Fig. 13(b), which is obviously the same in all settings.

7.3 Normalization Attributes

In this experiment we evaluate the performance of temporal normalization with different normalization attributes. Splitting data across all start and end points independent of the normalization attributes not only violates change preservation for group based operators, but dramatically decreases the performance. We show this on the $Incumben$ dataset and the following three normalization operations: $\mathcal{N}_{\{\}}$, $\mathcal{N}_{\{pcn\}}$ and $\mathcal{N}_{\{ssn\}}$. The runtime results and the output cardinality of these operations are shown in Fig. 14. There is a strong correlation between the normalization attributes and the performance. $\mathcal{N}_{\{\}}$ requires that all overlapping tuples are split, whereas $\mathcal{N}_{\{pcn\}}$ and $\mathcal{N}_{\{ssn\}}$ only require a split when the tuples match on the corresponding attribute values.

7.4 Expressing Temporal Outer Joins in SQL

We compare the computation of temporal outer joins using temporal alignment ($align$) with the computation of temporal outer joins expressed in standard SQL (sql). To express a temporal outer join in SQL we have to express the join part using overlap predi-



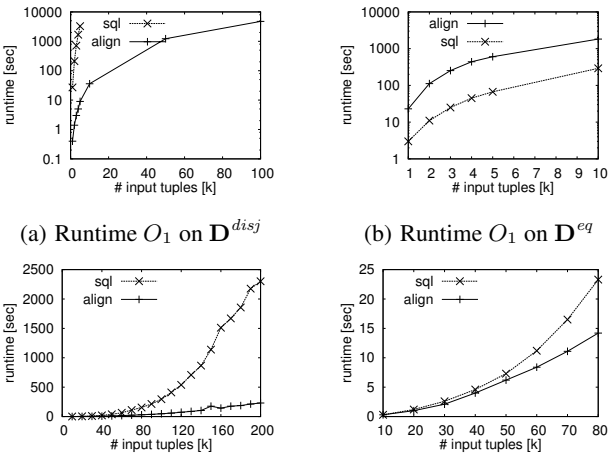
(a) Runtime (b) Output Size

Figure 14: Normalizations ($Incumben$).

cates on timestamps and evaluate the negative part of the temporal outer join using joins and NOT EXISTS statements [21]. The final result is the union of the two parts.

For the comparison we use three queries: $O_1 = \mathbf{r} \mathbb{X}^T_{true} \mathbf{s}$, $O_2 = \mathbf{r} \mathbb{X}^T_{Min \leq DUR(\mathbf{r}.T) \leq Max} \mathbf{s}$, and $O_3 = \mathbf{r} \mathbb{X}^T_{\mathbf{r}.pcn=\mathbf{s}.pcn} \mathbf{s}$, and three synthetic datasets: $\mathbf{D}^{disj}$ where the intervals in both relations are disjoint, $\mathbf{D}^{eq}$ where the intervals in both relations are equal, and $\mathbf{D}^{rand}$ where we have random intervals and categories.

Figure 15(a) shows the runtime of query O_1 on $\mathbf{D}^{disj}$. As expected, $align$ performs much faster than sql because of the NOT EXISTS used by SQL. The NOT EXISTS predicate is only efficient if a match can be found as fast as possible, so that the evaluation can terminate and return $false$. Since there are only few overlapping intervals in both relations, the NOT EXISTS has to scan almost the entire relation, which yields a quadratic complexity. The best setting for SQL for this is shown in Fig. 15(b) with the same query O_1 on dataset $\mathbf{D}^{eq}$. All timestamps of $\mathbf{D}^{eq}$ are equal, and thus the NOT EXISTS can be evaluated efficiently. For the $\mathbf{D}^{eq}$ dataset sql is more efficient than $align$ as it does not require any adjustment and the overhead is less than for alignment.



(a) Runtime O_1 on $\mathbf{D}^{disj}$ (b) Runtime O_1 on $\mathbf{D}^{eq}$

(c) Runtime O_2 on $\mathbf{D}^{rand}$ (d) Runtime O_3 on $Incumben$

Figure 15: Outer Joins (Real World and Synthetic Data Sets).

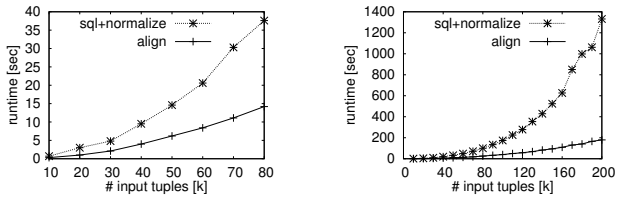
Figure 15(c) shows the runtime of query O_2 on dataset $\mathbf{D}^{rand}$. The θ condition of the outer join does not allow efficient NOT EXISTS statements using antijoins, resulting in a high runtime of the SQL approach. The approach using temporal alignment performs much faster as it is more efficient for timestamp adjustment.

Finally, we run query O_3 on the real world $Incumben$ dataset (Figure 15(d)). Both approaches are much faster than for the other datasets since the equality condition in the case of temporal alignment allows the database system to choose a fast nontemporal hash

or merge join, and in the case of SQL to speed up the NOT EXISTS statements.

7.5 Expressing Temporal Outer Joins with SQL and Normalize

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+normalize*). The joined part of the temporal outer join is computed with SQL, and temporal normalization is used for the temporal difference. Expressing outer joins with difference requires to compute the difference between an argument relation and the intermediate join result to determine all tuples that are not joining. Figure 16(a) shows the runtime behavior of query O_3 on the real world dataset $Incumben$. *align* performs much faster than *sql+normalize* due to the expensive normalization steps that *sql+normalize* is required to perform on the intermediate join result. In the last experiment in Figure 16(b) we compare the runtime



(a) Runtime O_3 on $Incumben$

(b) Runtime O_3 on Random Dataset

Figure 16: Outer Joins (Real World and Synthetic Data Sets).

of the same query O_3 on a random dataset. The interval timestamps have on average the same duration as in the $Incumben$ dataset, but start and end timestamps are randomly distributed. This yields a larger temporal join result and more distinct splitting points than for the real world dataset. With a larger temporal join result and more candidate splitting points, the performance of *sql+normalize* decreases compared to *align*.

8. CONCLUSION AND FUTURE WORK

In this paper we describe a relational algebra solution that provides native support for processing interval timestamped data with the sequenced semantics. We support the three properties of the sequenced semantics (snapshot reducibility, extended snapshot reducibility and change preservation) through timestamp propagation and two temporal primitives, a temporal splitter and temporal aligner. With these primitives query processing becomes a two-step process: (1) propagate and adjust the interval timestamps of argument tuples such that changes are preserved and predicates and functions over the original timestamps remain possible, and (2) apply the corresponding nontemporal operator on the adjusted relations. We defined rules to reduce the operators of a temporal algebra to their nontemporal counterparts. We have implemented the temporal primitives and reduction rules in the kernel of PostgreSQL to get native database support for all operations of a temporal algebra, including outer joins, antijoins, and aggregations with predicates and functions over the original timestamps.

Future work includes the following directions: investigate indexing or merge sort techniques to improve the performance of the temporal primitives for cases when conventional join techniques cannot be evaluated efficiently; customize the temporal primitives for specific temporal operators to not produce adjusted tuples that do not contribute to the result for that operator (the current temporal primitives are generic and work for tuple and group based operators, respectively); extend the temporal primitives for a bag based temporal algebra.

9. REFERENCES

[1] S. Abiteboul, L. Herr, and J. V. den Bussche. Temporal versus first-order logic to query temporal databases. In *Proc. of PODS*, pages 49–57, 1996.

[2] M. Agesen, M. H. Böhlen, L. O. Poulsen, and K. Torp. A split operator for now-relative bitemporal databases. In *Proc. of ICDE*, page 10, 2001.

[3] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, 39(1):25–34, 1997.

[4] M. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *Proc. of EDBT*, pages 257–275, 2006.

[5] M. H. Böhlen and C. S. Jensen. *Encyclopedia of Information Systems*, chapter Temporal Data Model and Query Language Concepts. Academic Press, 2002.

[6] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating the completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pages 153–172, 1995.

[7] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):48, 2000.

[8] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems*, 25(2):54, 2000.

[9] D. Gao, S. Jensen, T. Snodgrass, and D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005.

[10] H. D. J. Date and N. Lorentzos. *Temporal Data and the Relational Model*. Morgen Kaufmann Publisher, 2002.

[11] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying temporal models via a conceptual model. *Information Systems*, 19(7):513–547, 1994.

[12] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proc. of ICDE*, pages 222–231, 1995.

[13] W. Li, R. T. Snodgrass, S. Deng, V. K. Gattu, and A. Kasthurirangan. Efficient sequenced integrity constraint checking. In *ICDE*, pages 131–140, 2001.

[14] L. Liu and T. M. Özsu, editors. *Encyclopedia of Database Systems*. Springer Verlag, 2009.

[15] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.

[16] B. Moon, I. F. Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):744–759, 2003.

[17] C. Murray. Oracle database workspace manager developer's guide. `http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf`, 2008.

[18] PostgreSQL. Online temporal PostgreSQL reference. `http://temporal.projects.postgresql.org/reference.html`.

[19] A. Segev. Join Processing and Optimization in Temporal Relational Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 15,

pages 356–387. Benjamin/Cummings Publishing Company, 1993.

[20] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[21] R. T. Snodgrass. *Developing Time-Oriented Database Application in SQL*. Morgen Kaufmann Publisher, 1999.

[22] D. Son and R. Elmasri. Efficient temporal join processing using time index. In *Proc. of SSDBM*, pages 252–261, 1996.

[23] M. D. Soo, C. J. Jensen, and R. T. Snodgrass. An Algebra for TSQL2. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 27, pages 505–546. Kluwer Academic Publishers, 1995.

[24] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *Proc. of ICDE*, pages 282–292, 1994.

[25] Teradata. Teradata database temporal table support. `http://www.info.teradata.com/edownload.cfm?itemid=102320064`, 2010.

[26] D. Toman. Point-based vs. interval-based temporal query languages. In *Proc. of PODS*, pages 58–67, 1996.

[27] D. Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal Databases, Dagstuhl*, pages 211–237, 1997.

[28] I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 17(2):271–286, 2005.

[29] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB Journal*, 12(3):262–283, 2003.

[30] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. of PODS*, pages 237–245, 2001.

[31] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Proc. of ICDE*, pages 103–113, 2002.

APPENDIX

A. PROOF OF THEOREM 1

PROOF. We prove the reduction rule for the temporal left outer join, $\mathbf{r}\mathbb{X}\!\!\bowtie_\theta^T \mathbf{s}$, by showing that the operator satisfies the three properties of the sequenced semantics.

Snapshot reducibility (cf. Def. 1): We have to show two cases. Case 1: For each pair of matching and intersecting tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$ (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 $r \in \mathbf{r}$ and interval $T' \subseteq r.T$, for which no matching and intersecting $s \in \mathbf{s}$ 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. 10 (temporal alignment) and Proposition 4 we know that aligned tuples $\tilde{r} \in \mathbf{r}\Phi_\theta\mathbf{s}$ are derived from an $r \in \mathbf{r}$ as follows: (i) for each matching and intersecting $s \in \mathbf{s}$ 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 $s \in \mathbf{s}$ we get $\tilde{r} = (r.\mathbf{A}, T)$. The same holds for the aligned tuples $\tilde{s} \in \mathbf{s}\Phi_\theta\mathbf{r}$.

From (i) we conclude that for any two matching and intersecting tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$, there exists an aligned tuple $\tilde{r} = (r.\mathbf{A}, r.T \cap s.T)$ 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 $r \in \mathbf{r}$ and maximal subinterval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathbf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, T)$ but no matching $\tilde{s} \in \mathbf{s}\Phi_\theta\mathbf{r}$ 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. 4): 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 $r \in \mathbf{r}$ ($s \in \mathbf{s}$) 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 $r \in \mathbf{r}$ and $s \in \mathbf{s}$, 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 $r \in \mathbf{r}$ and maximal sub-interval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathbf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, r.U, T)$ but no matching $\tilde{s} \in \mathbf{s}\Phi_\theta\mathbf{r}$ 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. 7): From Def. 10 (temporal alignment) and Proposition 4 we know that the timestamp of each result tuple is (case 1) either an intersection of two argument tuples, $r \in \mathbf{r}$ and $s \in \mathbf{s}$, or (case 2) a maximal subinterval $T \in r.T$ for which no matching and intersecting $s \in \mathbf{s}$ exists. Furthermore, the α-operator ensures that all result tuples have maximal timestamps.

Case 1: We show that for each result tuple $z = (r.\mathbf{A}, z.\mathbf{C}, r.T \cap s.T)$, the lineage set $\mathsf{L}[\mathbf{r}\mathbb{X}\!\!\bowtie_\theta^T \mathbf{s}](z,t)$ is equal for each $t \in z.T$ and that adjacent value-equivalent tuples have different lineage sets. From Def. 6 (lineage sets) we get $\mathsf{L}[\mathbf{r}\mathbb{X}\!\!\bowtie_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \bowtie_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t)$ for case (1). The lineage set of the temporal Cartesian product contains all $r \in \mathbf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t and all $s \in \mathbf{s}$ that are value-equivalent to $z.\mathbf{C}$ and cover t. Since relations are duplicate free, the lineage set contains exactly one $r \in \mathbf{r}$ and one $s \in \mathbf{s}$, i.e., $\mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t) = \langle\{r\},\{s\}\rangle$. This holds for all $t \in r.T \cap s.T$. To show that the lineage set at timepoint $z.T_s{-}1$ is different for value-equivalent 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 timepoint $z.T_e$.

Case 2: We show that for each result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$, the lineage set $\mathsf{L}[\mathbf{r}\mathbb{X}\!\!\bowtie_\theta^T \mathbf{s}](z,t)$ is equal for all $t \in z.T$ and that adjacent value-equivalent tuples have different lineage sets. From Def. 6 we get $\mathsf{L}[\mathbf{r}\mathbb{X}\!\!\bowtie_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \rhd_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t)$. The lineage set of the temporal difference contains all $r \in \mathbf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t as well as $\mathbf{s}$. Since relations are duplicate free, we get $\mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t) = \langle\{r\},\mathbf{s}\rangle$. This holds for all $t \in z.T$ since $z.T = \tilde{r}.T \subseteq r.T$. To show that the lineage set of value-equivalent tuples is different at timepoint $z.T_s{-}1$, recall that $z.T$ is maximal. Either $z.T_s{-}1 \notin r.T$ and therefore r is not in the lineage set, or there exists a matching $s \in \mathbf{s}$ with $z.T_s{-}1 \in s.T$ that would produce a join with $r.\mathbf{A}$, and thus no value-equivalent tuple to $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ can exist. The same reasoning applies for the timepoint $z.T_e$. □