Algebraic Operators for Processing Sets of Temporal Intervals in Relational Databases

Andreas Dohr
University of Bonn, Institute of Computer Science III, Germany
andreas.dohr@uni-bonn.de

Christiane Engels
University of Bonn, Institute of Computer Science III, Germany
engelsc@cs.uni-bonn.de

Andreas Behrend
University of Bonn, Institute of Computer Science III, Germany
behrend@cs.uni-bonn.de

Abstract

The efficient management of temporal data has become increasingly important for many database applications. Most commercial systems already allow the management of temporal data but the operational support for processing this data is still rather limited. One particular reason is that many extension proposals typically require considerable modifications of the underlying database engine. In this paper, we propose a lightweight solution where temporal operators are realized using a library of user-defined functions. This way the complexity of temporal queries can be drastically reduced leading to more readable and less error-prone code without touching the database system. Our experiments show that the proposed operators significantly outperform temporal queries formulated in pure SQL. In addition, we investigate the possibility to incorporate algebraic optimization strategies directly into our operator definitions which allow for further performance improvements.

2012 ACM Subject Classification Information systems → Database management system engines, Information systems → Information systems applications

Keywords and phrases Temporal Databases, Relational Operators, Situation Calculus

Digital Object Identifier 10.4230/LIPIcs.TIME.2018.11

1 Introduction

The support for temporal data is continuously extended by almost every commercial database system reflecting its importance for many practical applications. Today's systems typically provide an automated version control of the data and mechanisms for fast history access known as time travel. However, many temporal operations are still implemented within the application layer because database systems offer no comprehensive support for efficiently performing operations over temporal data, yet. In particular, the fact that temporal data is usually represented by means of temporal intervals rather than single time points is usually neglected and no specific data type like period nor operator support is provided in standard SQL. Consequently, the task of processing sets of temporal intervals has to be realized by complex low-level SQL queries or is simply left to tools or applications outside the database.

For querying temporal intervals, the sequenced semantics seems to be the most natural choice [6]. Formulating sequenced statements over this data in standard SQL, however, often leads to very complex expressions [9] which are difficult to maintain and can hardly be optimized by the database system.

© Andreas Dohr, Christiane Engels, and Andreas Behrend;
licensed under Creative Commons License CC-BY
25th International Symposium on Temporal Representation and Reasoning (TIME 2018).
Editors: Natasha Alechina, Kjetil Nørvåg, and Wojciech Penczek; Article No. 11; pp. 11:1–11:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As an example, we consider an air traffic scenario in which various events (e.g. changes of altitude, speed, and heading of an aircraft) are continuously monitored and stored in a database. If we are interested in the time interval during which a plane was in cruising mode but did not perform a maneuver, various subintervals have to be computed as a cruising period may coincide with several maneuver periods. The inherent complexity of the reasoning becomes even more apparent as soon as more than two conditions are compared this way. This is due to the fact that all possible sequences of time points have to be covered by the respective database query as the systems basically support an event-based calculus, only. Still missing is the possibility to generate sets of input intervals which are processed by set-oriented operators within the FROM clause. To this end, TABLE functions can be used which allow for modeling the required many-to-many mapping. As an example, the following query detects cruising phases without maneuvering for flights operated by Lufthansa using the table function TDIFF:

```
SELECT Start,End FROM TABLE (TDIFF(vw_LHcruising,vw_LHmaneuvering)) Period;
```

The complexity of the query is drastically reduced as all time point comparisons are hidden in the table function which accesses two views containing cruising and maneuvering phases of Lufthansa flights, respectively.

In this paper, we discuss the development of a library of such temporal operators which form the basis for realizing a situation calculus inside a temporal database system. In particular, our contributions are as follows:

- We introduce formal definitions of the most important operators like TUNION, TDIFF, TSECT, THULL, TCOMPL, or TCROP in a systematic way.
- We show the possibility of optimizing sequences of temporal operators similar to algebraic optimization rules for relational expressions.
- We provide an evaluation of various implementation alternatives for temporal operators showing the feasibility of our proposal.

The paper is organized as follows: After discussing related work in Section 2 we specify a set of temporal operators and corresponding algebraic optimization rules in Sections 3 and 4. Different implementations of two selected temporal operators are discussed in Section 5 and evaluated in Section 6. Finally, we draw a conclusions in Section 7.

2 Related Work

Our work is motivated by the general idea of supporting situation awareness inside a database system [13]. The workspace manager of Oracle provides the well-known relationship operators proposed by Allen [2] which can be employed for this purpose. When querying a history of situations, however, the comparison of sets of intervals is often needed, and Allen's operators are not suited anymore [4]. The lack of database support for monitoring the temporal development of data led to specialized systems such as moving object, data stream or complex event processing (CEP) systems. Especially CEP appeared to be well-suited for handling the dynamics of an application. While the employed event-based calculus is useful for general temporal reasoning over discrete changes, it is not well-suited for modeling the continuous changes between domain-specific situations represented as anchored temporal intervals. Even CEP systems with a richer data model for temporal abstractions such as Microsoft's StreamInsight [1], Naiad [17] or SEEP [12] are limited with respect to processing sets of temporal intervals.

As an alternative approach, operational support for processing sets of temporal intervals has been proposed in [9], [10]. In these works the authors propose to reduce a temporal operator to its nontemporal counterpart by using two primitives, the *temporal normalizer*

for group based operators (e.g. projection, aggregation, union) and the *temporal aligner* for tuple based operators (product, joins). This represents an elegant generic approach and even provides the possibility to use the DBMS optimizer for the resulting nontemporal expressions. However, rewriting temporal operators this way leads to unspecialized versions which perform mostly not as efficient as the more specialized table functions proposed in this work. In addition, several extensions to a relational DBMS kernel have to be implemented in order to use these temporal features which is problematic for proprietary DBMS. No support for temporal operations in non-sequenced semantics is provided.

3 Temporal Operators

In this section we present temporal operators for processing sets of temporal intervals which have been already mentioned in [4]. We extend this work by providing a formal basis for the specification of these operators from which suitable implementations are derived in Section 5. We first define temporal operators working on sets of intervals in Section 3.1. Afterwards, these operators are used to specify temporal operators for period-stamped temporal relations in Section 3.2. All our proposed operators act on table level, meaning that they have table-valued in- and/or output. For illustrating our approach, we exemplarily define

- three point-based binary operators TUNION, TDIFF, and TSECT,
- two point-based unary operators THULL and TCOMPL,
- four interval-based unary operators TMIN, TMAX, TFIRST, and TLAST,
- and a binary interval-based operator TCROP with its point-based analog TCROPC.

3.1 Interval Operators

In the following we use half-open time intervals $I = [I.s, I.e)$ including the start point $I.s$ and excluding the end point $I.e$ of interval I. We denote the interval length by $I.\ell := I.e - I.s$ for $I.e \neq \infty$ and $I.\ell := \infty$ otherwise. For the definition of the point-based operators we use *coalescing* of a set of temporal intervals $\mathcal{I}$ specified as follows:

$$\begin{aligned} Coal(\mathcal{I}) := \{[I_1.s, I_k.e) \mid\ & \exists \text{ sequence } I_1 < \ldots < I_k \in \mathcal{I} \text{ s.t. } I_i.e \in I_{i+1}, 1 \leq i \leq k-1, \\ & \text{and } \nexists\, I \in \mathcal{I} \text{ s.t. } I_1.s \in \text{int}(I) \text{ or } I_k.e \in I\}, \end{aligned}$$

where $I_1 < I_2 :\Leftrightarrow I_1.s < I_2.s \wedge I_1.e < I_2.e$ and $\text{int}(I) := (I.s, I.e)$ denotes the interior of interval I. Because of the first condition we only coalesce intervals that are either overlapping or touching. The second condition ensures that the sequence and therefore the resulting interval cannot be further extended. An example is shown in Figure 1. Note that this definition of coalescing is equivalent to the standard definition of temporal coalescing given for example in [7].

For the binary operators union, difference and intersection we have chosen a point-based rather than interval-based model (cf. [5]). By using coalescing, we represent the required timestamps (i.e. minimum requirements) with a maximal interval partition.

▶ **Definition 1** (Operators TUNION, TDIFF, and TSECT). Let $\mathcal{I}$ and $\mathcal{I}'$ be finite sets of right-open intervals. Then the temporal union of the two interval sets $\mathcal{I}$ and $\mathcal{I}'$ is defined as $\text{TUNION}(\mathcal{I}, \mathcal{I}') := Coal(\mathcal{I} \cup \mathcal{I}')$. Their temporal difference is

$$\begin{aligned} \text{TDIFF}(\mathcal{I}, \mathcal{I}') := Coal(\{J \subseteq I \mid I \in \mathcal{I}, \forall I' \in \mathcal{I}' : J \cap I' = \emptyset, \\ \text{and J is maximal with this property}\}), \end{aligned}$$

and the temporal intersection is given by $\text{TSECT}(\mathcal{I}, \mathcal{I}') := Coal(\{I \cap I' \mid I \in \mathcal{I}, I' \in \mathcal{I}'\})$.

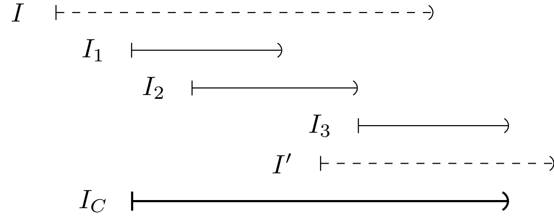


Figure 1 Example of the coalescing operator, $I_C = Coal(\{I_1, I_2, I_3\})$. If there were intervals I or I', the sequences $I < I_3$ and $I_1 < I_2 < I'$ respectively lead to an extended coalesced interval I_C.

Temporal union is defined as coalescing of the input interval sets whereas temporal difference comprises all subintervals of $\mathcal{I}$ not covered by $\mathcal{I}'$ in coalesced form. Similarly, temporal intersection is defined as the coalesced common time periods of both sets. Note that if both input sets are coalesced, i.e. $Coal(\mathcal{I}) = \mathcal{I}$ and $Coal(\mathcal{I}') = \mathcal{I}'$, then $\mathrm{TSECT}(\mathcal{I}, \mathcal{I}') = \{I \cap I' \,|\, I \in \mathcal{I}, I' \in \mathcal{I}'\}$, and if $\mathcal{I}$ is coalesced, then $\mathrm{TDIFF}(\mathcal{I}, \mathcal{I}') = \{J \subseteq I \,|\, I \in \mathcal{I}, \forall I' \in \mathcal{I}' : J \cap I' = \emptyset$, and J is maximal with this property$\}$ [7]. As a last example of point-based operators, the two unary operators THULL and TCOMPL are defined. A sample application of all introduced point-based operators is depicted in Figure 2.

▶ **Definition 2** (Operators THULL and TCOMPL). We define the hull of a finite set of right-open intervals $\mathcal{I}$ as the time period enclosed by the given intervals $\mathrm{THULL}(\mathcal{I}) := [\min_{I \in \mathcal{I}} I.s, \max_{I \in \mathcal{I}} I.e)$. The complement of $\mathcal{I}$ with respect to its hull is given by $\mathrm{TCOMPL}(\mathcal{I}) := \mathrm{TDIFF}(\{\mathrm{THULL}(\mathcal{I})\}, \mathcal{I})$.

The list of operators is complemented by four interval-based operators which return the shortest and longest as well as the first and last set of intervals from $\mathcal{I}$, respectively.

▶ **Definition 3** (Operators TMIN and TMAX). For a finite set of right-open intervals $\mathcal{I}$, we define the operators TMIN and TMAX returning the shortest respectively longest intervals of $\mathcal{I}$ as $\mathrm{TMIN}(\mathcal{I}) := \arg\min_{I \in \mathcal{I}} I.\ell$ and $\mathrm{TMAX}(\mathcal{I}) := \arg\max_{I \in \mathcal{I}} I.\ell$.

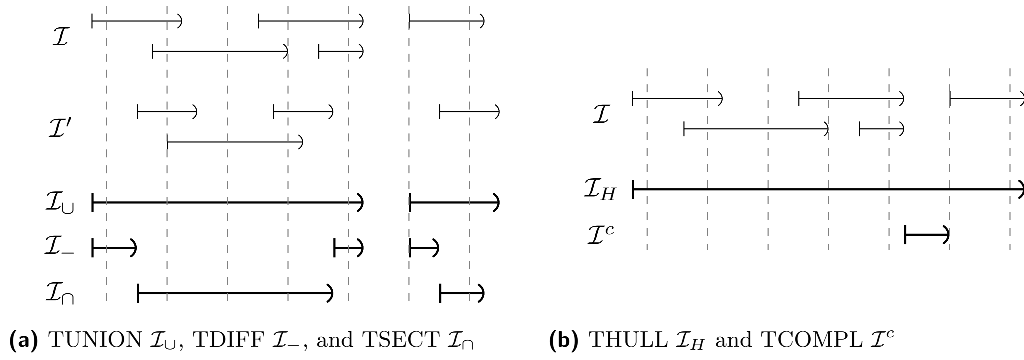
▶ **Definition 4** (Operators TFIRST and TLAST). For a finite set of right-open intervals $\mathcal{I}$, we define the operators TFIRST and TLAST returning the intervals of $\mathcal{I}$ having the earliest start (latest end) point as $\mathrm{TFIRST}(\mathcal{I}) := \arg\min_{I \in \mathcal{I}} I.s$ and $\mathrm{TLAST}(\mathcal{I}) := \arg\max_{I \in \mathcal{I}} I.e$.

Finally, the TCROP operator crops the input intervals with respect to a given interval CR. This operator is especially useful for viewing a snapshot of some relation, i.e. all Lufthansa flights performed today. TCROPC is its coalesced analog.

▶ **Definition 5** (TCROP and TCROPC). For a finite set of right-open intervals $\mathcal{I}$ and a non-empty, right-open interval CR we define $\mathrm{TCROP}(\mathcal{I}, CR) := \{I \cap CR \,|\, I \in \mathcal{I}\}$ and $\mathrm{TCROPC}(\mathcal{I}, CR) := Coal(\{I \cap CR \,|\, I \in \mathcal{I}\})$.

3.2 Operators for Period-Stamped Relations

In a temporal database tuple, temporal intervals are typically associated with other 'non-temporal' attribute values which have to be processed, too. Therefore, the introduced operators are extended to temporal relations with explicit, non-temporal attributes A and timestamp T. We consider $\pi_{B,T}(R)$ for $B \subseteq A$ a subset of the non-temporal attributes of the relation R. The proposed operators will act on all value equivalent tuples of $\pi_{B,T}(R)$, i.e.



(a) TUNION $\mathcal{I}_\cup$, TDIFF $\mathcal{I}_-$, and TSECT $\mathcal{I}_\cap$ **(b)** THULL $\mathcal{I}_H$ and TCOMPL $\mathcal{I}^c$

Figure 2 Sample applications of point-based operators for sets of right-open intervals $\mathcal{I}$ and $\mathcal{I}'$.

performing temporal aggregation with respect to B. Note that for $B = \emptyset$ the operations are directly performed on the temporal intervals and for $B = A$ value equivalence on relational level is considered. We denote by $R_T^{\bar{b}} := \pi_T(\sigma_{B=\bar{b}}(R))$ the timestamps of all value equivalent tuples w.r.t. B with value $\bar{b}$. First, we define the extended coalescing operator w.r.t $B \subseteq A$:

$$Coal(R, B) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R), t \in Coal(R_T^{\bar{b}})\} .$$

The three point-based operators TUNION, TDIFF, and TSECT for sets of intervals are extended in the following to relational operators having two period-stamped relations and a subset $B \subseteq A$ of the non-temporal attributes as input:

▶ **Definition 6.** For two period-stamped relations R and S with non-temporal attributes A, timestamp T, and $B \subseteq A$ we set

$$\text{TUNION}(R, S, B) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R \cup S), t \in \text{TUNION}(R_T^{\bar{b}}, S_T^{\bar{b}})\} ,$$

$$\text{TDIFF}(R, S, B) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R), t \in \text{TDIFF}(R_T^{\bar{b}}, S_T^{\bar{b}})\} ,$$

$$\text{TSECT}(R, S, B) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R) \cap \pi_B(S), t \in \text{TSECT}(R_T^{\bar{b}}, S_T^{\bar{b}})\} .$$

The remaining unary operators on relation level can be directly specified via their interval equivalent:

▶ **Definition 7.** For two period-stamped relations R and S with non-temporal attributes A, timestamp T, and $B \subseteq A$ we set $\text{TOP}(R, B) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R), t \in \text{TOP}(R_T^{\bar{b}})\}$ with TOP$\in$ {THULL, TCOMPL, TMIN, TMAX, TFIRST, TLAST}.

Similarly, the operators TCROP and TCROPC are extended:

▶ **Definition 8.** For a period-stamped relation R with non-temporal attributes A, timestamp T, $B \subseteq A$, CR a non-empty right-open interval, and TOP $\in$ {TCROP, TCROPC} we set $\text{TOP}(R, B, CR) := \{(\bar{b}, t) \,|\, \bar{b} \in \pi_B(R), t \in \text{TOP}(R_T^{\bar{b}}, CR)\}$.

4 Algebraic Optimization

Similar to the non-temporal relational operators, algebraic optimization can be performed for our temporal operators. Let's consider the following example. We are interested in all flight phases excluding landings performed yesterday by a Southwest Airlines' aircraft. The following SQL query employing the TDIFF- and TCROP-operators can be employed:

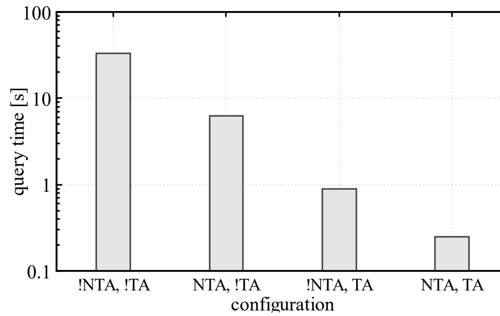


Figure 3 Performance of selection pushing for non-temporal attributes (NTA) and the timestamp attribute (TA).

```
SELECT aircraft, T FROM TABLE
  TCROP(TDIFF(flight,landing,{aircraft,airline})),
      {aircraft,airline}, yesterday) flight_phases
WHERE airline = 'Southwest Airlines'
```

Before calculating the temporal difference it makes sense to evaluate the selection to Southwest Airlines, i.e. to push the selection. Likewise we might also push the condition that the flight phase was performed yesterday. As this refers to the timestamp caution is advised. Pushing the temporal selection in the example using around 1.7 million entries from the dataset "Airline On-Time Statistics and Delay Causes" [18] considerably reduces the execution time as depicted in Figure 3. We focus in the following on the formal basis of these kinds of optimizations and introduce selection pushing as algebraic optimization rule for our temporal operators.

In [7] the following so called *conditional coalescing rule* states that selections σ^T can be pushed through the coalescing operator acting on value equivalent tuples (i.e. $Coal(R, A)$ for $A := attr(R)$ in our terminology) if the selection condition c does not contain any temporal attribute: $\sigma_c^T(Coal(R, A)) \equiv Coal(\sigma_c^T(R, A))$. This rule naturally extends to $B \subseteq A$ provided that the selection condition refers to attributes in B only. As our operators are based on coalescing and/or act on value equivalent tuples w.r.t. B this allows us to push selections – still provided that the selection condition refers to attributes in B only. Regarding pushing of selections referring to the timestamp T, we studied three kinds of selection criteria:

1. A restriction on the start point of the timestamp, i.e. T.s $\in$ Q for a given right-open interval Q.
2. A restriction on the end point of the timestamp, i.e. T.e $\in$ Q' for a given right-open interval Q'.
3. A restriction on the length of the timestamp, i.e. T.$\ell \in$ D for a given right-open interval D.

Furthermore, we distinguished whether the input relations of the operators are in coalesced form and whether the output relation will be cropped to an interval CR. In the following we will focus on the three point-based binary operators TUNION, TDIFF and TSECT.

The example in Figure 4 illustrates that due to the coalescing applied in our temporal operators we cannot simply push selections like those specified above through the operators. Given a set of right-open intervals $\mathcal{I}$ we want to select all intervals of $Coal(\mathcal{I})$ starting in Q. Naively selecting all intervals of $\mathcal{I}$ starting in Q as depicted in $\mathcal{S}$tart does not lead to the solution $\mathcal{R}$es. This is because two of the intervals will be merged during coalescing with an interval starting before Q violating the selection condition, and the other interval will

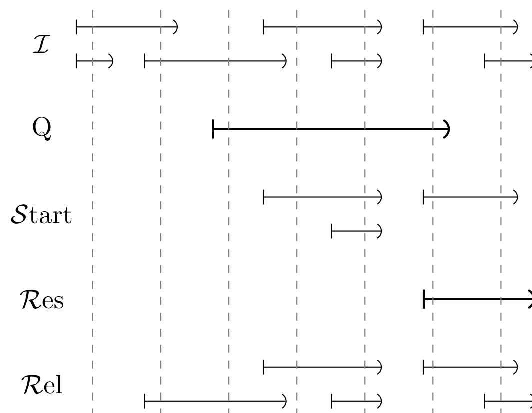


Figure 4 Example of selection pushing for temporal operators. $\mathcal{S}$tart comprises all intervals of $\mathcal{I}$ starting in Q, $\mathcal{R}$es contains the result set of $\sigma_{I.s \in Q}(Coal(\mathcal{I}))$, and $\mathcal{R}$el comprises the intervals of $\mathcal{I}$ relevant to compute $\mathcal{R}$es.

be extended during coalescing with an interval starting after Q, so that only the modified interval is contained in the result set $\mathcal{R}$es. But the first two intervals of $\mathcal{I}$ are not relevant at all to compute $\mathcal{R}$es because they end before Q starts. $\mathcal{R}$el comprises the intervals of $\mathcal{I}$ relevant to compute $\mathcal{R}$es.

Generalizing the idea of intervals relevant to compute the result of a selection referring to the timestamp, we cannot push the selection through the temporal operator, but we can introduce selections applied before the temporal operator on the basis of the selection condition in order to reduce the input size of the operator. For the question which intervals are starting during a specified interval Q over an uncoalesced input only those intervals are relevant that

- start in Q and are therefore possibly part of an interval of the result set,
- end in Q (or later) and may extend intervals starting in Q to violate the selection condition,
- or start after Q and may extend intervals starting in Q such that the extended, coalesced interval is part of the result set.

In particular, intervals ending before Q, i.e. satisfying I.e $<$ Q.s, are not relevant and can be discarded. This is true for all three operators TUNION, TDIFF, and TSECT giving the following rule:

In the case of uncoalesced input relations with selection condition $T.s \in Q$, a selection $\sigma_{T.e \geq Q.s}$ may be introduced and pushed through the temporal operator.

For TUNION, it does not make any difference whether the input relations are coalesced or not, as coalescing is involved anyway. But for TDIFF and TSECT, we can refine the above rule in cases with coalesced input relations as no further coalescing has to be performed (cf. Section 3.1). So if R is in coalesced form, we can introduce and push a stronger selection condition for R through TDIFF(R,S,B), namely $\sigma_{T \cap Q \neq \emptyset}$. Similarly, if R or S is coalesced, we can introduce and push the same selection condition for R or S respectively through TSECT(R,S,B).

If the output is cropped to an interval CR we can further reduce the input relations. Again considering the case with uncoalesced input first, we can introduce and push a new selection $\sigma_{T.s < CR.e}$. Everything happening after CR.e is irrelevant if the resulting intervals are cropped to CR even though there were intervals starting in Q. In addition, we can adjust

Table 1 Introduced selection conditions in the different settings for the operators TUNION, TDIFF, and TSECT. *Coalesced* only refers to relation R in TDIFF(R,S,B) and relations R and S in TSECT(R,S,B).

Setting	$T.s \in Q$	$T.e \in Q'$
uncoalesced	$\sigma_{T.e \geq Q.s}$	$\sigma_{T.s \leq Q'.e}$
coalesced	$\sigma_{T \cap Q \neq \emptyset}$	$\sigma_{T \cap Q' \neq \emptyset}$
cropped to CR + uncoalesced	$\sigma_{T.e \geq Q.s}$ $\sigma_{T.s < CR.e}$	$\sigma_{T.s \leq Q'.e}$ $\sigma_{T.e > CR.s}$
cropped to CR + coalesced	$\sigma_{T \cap Q \neq \emptyset}$ $\sigma_{T \cap CR \neq \emptyset}$	$\sigma_{T \cap Q' \neq \emptyset}$ $\sigma_{T \cap CR \neq \emptyset}$

the query interval Q and the crop interval CR according to the following formulas

$$CR^*.s := max(CR.s, Q.s) \quad \text{and} \quad Q^*.e := min(CR.e, Q.e)$$

as all resulting intervals start in Q and the output is cropped to CR. In the coalesced case we can again push a stronger selection condition, namely $\sigma_{T \cap CR \neq \emptyset}$ for coalesced R in TDIFF(R,S,B) and coalesced R or S in TSECT(R,S,B).

A summary of the introduced selection conditions in the different settings for $T.s \in Q$ – as well as those for $T.e \in Q'$ which are exactly inverted – is given in Table 1.

For queries referring to the timestamp length, i.e. $T.\ell \in D$, we can push selections only in the coalesced cases of TSECT and R in TDIFF(R,S,B). In these cases the input intervals can only get shorter when the temporal operator is applied. So only intervals longer than D.s are relevant and the condition $\sigma_{T.\ell \geq D.s}$ can be introduced and pushed. If in addition the output gets cropped the selection $\sigma_{T \cap CR \neq \emptyset}$ can be introduced again.

The presented rules for introducing and pushing selections are designed in such a way that for a combined selection $\sigma_{T.s \in Q \wedge T.e \in Q'}$ all suitable selections according to Table 1 can be introduced and pushed.

5 Implementation

In this section we discuss the implementations of two selected temporal operators. As examples we choose the temporal union operator (TUNION) and the temporal difference operator (TDIFF). We present several algorithms to realize the proposed semantics of those two operators. We implement the TUNION operator using *Single Scan*, *Index Based* and *Sorting Based* algorithms. The TDIFF operator is implemented by *Union First* and *Sorting Once* algorithms.

To use these implementations, there is no need to modify database systems regarding parser trees, cost functions, etc. To make the application of temporal operators transparent to the user in a high degree, we are using Oracle User-Defined Functions [20] for implementation. The implementation is not limited to Oracle Database products. Using primarily user defined table functions and basic concepts it is relatively easy to implement the operators in any common database system. For instance, in Section 6 an implementation in PostgreSQL was used to compare TUNION and TDIFF with existing temporal operators using Temporal Postgres [19].

5.1 Temporal Union

The temporal union operator (TUNION) merges right-open intervals using the concept of coalescing introduced decades ago ([7], [11], [15]) that do have overlapping relationships as defined by James F. Allen (i.e. *contains*, *equals*, *starts*, *finishes*, *meets*, *overlaps*) [2]. While the input sets can contain those overlapping relationships the output sets only include relationships of the kind *before* or *after* due to the construction of this operator.

Using an algorithm described in [22] as well as two basic algorithms we present three implementations for temporal union designed as Oracle User-Defined Functions. Pure SQL:1992 queries [21] are neither used nor evaluated due to their already shown poor performance [22].

In contrast to the definition from Section 3, the following implementations use one interval set as input instead of two sets due to simplicity reasons. The implementation with two input sets is analogue.

5.1.1 Single Scan

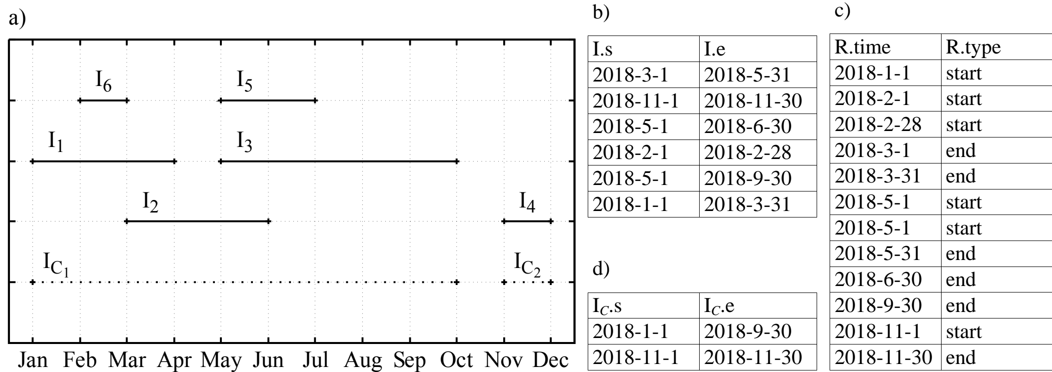
The most efficient algorithm proposed in [22] uses partitioning of data and windowing functions to coalesce time intervals of equal, nontemporal attributes. The concepts of windowing functions were introduced in the SQL:2003 standard.

For a table R, containing start and end points of intervals and additional nontemporal columns, the temporal columns are queried into a temporary table that holds the timestamps ts, two additional columns named $Start_ts$ and End_ts and columns for the nontemporal data. For a start point set $Start_ts = 1$ and $End_ts = 0$, for an end point set $Start_ts = 0$ and $End_ts = 1$.

The temporary table is queried to build differences of the sums of the start and end points assigned to a certain timestamp. Compute the difference of sums for ts at time t_i and denote it as $Current_Total_ts$. Compute the difference of sums for ts at time t_{i-1} and denote this as $Previous_Total_ts$. For efficient querying order the timestamps and limit the scope of the analytic windows function. If $Current_Total_ts = 0$ the number of start and end points at time t_i are equal and ts is an end point of a result interval. If $Previous_Total_ts = 0$ the number of start and end points at time t_{i-1} were equal and ts is an start point of a result interval. In a last step all those identified timestamps are combined to create the final result set of coalesced intervals.

5.1.2 Index Based

The index based implementation of the temporal union operator is using built-in indexes on the start points and end points of given intervals in order to enhance searching for large intervals overlapping many small intervals. Finding these large intervals the algorithm can skip processing the smaller intervals contained. Starting with the lowest start point $I_i.s$, the algorithm searches for the maximum end point $I_k.e$ that has an start point $I_k.s$ smaller than the given end point $I_i.e$. The interval is extended by this endpoint and this extended interval is used for further exploration of new end points. If none is found, $[I_i.s, I_k.e)$ is stored as new interval and the start point of a new interval is searched with $I_l.s > I_k.e$. With this new start point another iteration starts finding a maximum end point. Indexing in this context is used to support finding the minimal start points and maximal end points. These can be computed in almost constant time. To keep the implementation simple and compatible with existing systems, we decided to use traditional b-trees, so that only little changes have to be applied to an existing database schema.



b)

I.s	I.e
2018-3-1	2018-5-31
2018-11-1	2018-11-30
2018-5-1	2018-6-30
2018-2-1	2018-2-28
2018-5-1	2018-9-30
2018-1-1	2018-3-31

c)

R.time	R.type
2018-1-1	start
2018-2-1	start
2018-2-28	start
2018-3-1	end
2018-3-31	end
2018-5-1	start
2018-5-1	start
2018-5-31	end
2018-6-30	end
2018-9-30	end
2018-11-1	start
2018-11-30	end

d)

I_C.s	I_C.e
2018-1-1	2018-9-30
2018-11-1	2018-11-30

Figure 5 TUNION Sorting Algorithm.

5.1.3 Sorting Based

The sorting based algorithm is using a fairly simple approach. It is somehow similar to the single scan algorithm but does not need SQL:2003 windowing functions. Let $\mathcal{I}$ be a set of right-open intervals (see Figure 5 a). Sort the start and end points of these intervals (b), hold them in a result set $\mathcal{R}$ and assign a type to them (*start* and *end*, (c)). After that, sweep over the sorted result set with a single running counter. By increasing or decreasing the counter depending on the type of time point one can create all resulting intervals in a single run (d). The counter is increased if a point with type *start* is recognized and decreased if point of type *end* is recognized. If the counter equals 0, a new result interval can be added to the results $\mathcal{I}_C$. The sorting based TUNION is by default almost independent of the dataset's structure. The sorting is applied to all starting and all end points of every interval. Therefore by having n intervals we have to order $2n$ time points in logarithmic complexity which results in a runtime complexity of $O(n \log(n))$.

5.2 Temporal Difference

The temporal difference operator (TDIFF) is used to compute the difference of two sets of right-open intervals $\mathcal{I}$, $\mathcal{I}'$. The resulting set of intervals $\mathcal{R}$ consists of all intervals created by the points of the intervals of the first set which are not elements of the intervals of the second set. We will present two strategies for implementing the temporal difference in this section.

5.2.1 Union First

To implement a *Union First* algorithm we utilize the already proposed temporal union operator by applying it on the two input sets $\mathcal{I}$, $\mathcal{I}'$ separately. Reducing the input sets with this operator ensures all remaining intervals to be distinct and ordered in their respective result tables. Now we can move two nested cursors over these remaining intervals and compare them to get the resulting subintervals $\mathcal{R}$. Comparing the intervals $[I_i.s, I_i.e)$ and $[I'_j.s, I'_j.e)$, three different relationships can hold.

- $I_i.e < I'_j.s$
  The first interval $[I_i.s, I_i.e)$ is before the second $[I'_j.s, I'_j.e)$ not overlapping with it. A new start point $R_k.s = I_i.s$ in the result intervals of R is created if $R_k.s = null$ holds. Set $R_k.e = I_i.e$ and move to $[I_{i+1}.s, I_{i+1}.e)$.

- $I_i.s < I'_j.s =< I_i.e$
  The two intervals are overlapping or adjacent and the first interval starts before the second interval start. If $R_k.s = null$ add the start point of the first interval a new result interval start point, $R_k.s = I_j.s$, and set $R_k.e = I'_j.s$. If the end point of the first interval is after the end point of the second interval, move to the next interval $[R_{k+1}.s, R_{k+1}.e) = [null, null)$. Set $[R_{k+1}.s, R_{k+1}.e) = [I'_j.e, I_i.e)$. Move to $[I'_{j+1}.s, I'_{j+1}.e)$.
- $I_i.s >= I'_j.s$
  If the end point of the first interval is before or at the end point of the second interval, no results are created or updated. Move to $[I_{i+1}.s, I_{i+1}.e)$. If the end point of the second interval is after or at the start point of the first interval, the result can be updated: $[R_k.s, R_k.e) = [I'_j.e, I_i.e)$. If the end point of the second is before the start point of the first interval, set $[R_k.s, R_k.e) = [I_i.e, I_i.s)$.

5.2.2 Sorting Once

Implementing a *Sorting Once* algorithm is similar to the TUNION operator based on sorting: querying the start and end points of the input interval sets, assigning types to the points, ordering them and sweeping over the ordered points one time only. We query the start and end points of the two sets of right-open intervals $\mathcal{I}$, $\mathcal{I}'$ and assign those points to four different types: $start1$, $start2$, $end1$, $end2$. To log the occurrence of those types we use two counters: $counter1$ is increased if a type $start1$ is recognized and decreased if the type is $end1$; $counter2$ is increased if a type $start2$ is recognized and decreased if the type is $end2$. Initial values are 0. To avoid creating false intervals when start or end points in $\mathcal{I}$ and $\mathcal{I}'$ are identical, ordering of types must follow the order $I.s$, $I'.s$, $I.e$, $I'.e$, which can be accomplished by simply adjusting names of the assigned types: $1start2$, $2start1$, $3end1$, $4end2$. In the following, the ordered list of points of $\mathcal{I}$ and $\mathcal{I}'$ is denoted $\mathcal{L}$, the types for entries are $L.t$ and time points $L.p$, the resulting interval set is denoted as $\mathcal{R}$. After ordering $\mathcal{L}$ by $L.p$ and $L.t$, we sweep over $\mathcal{L}$ and evaluate by type:

- $L_i.t = 2start1$
  Start point of an interval from $\mathcal{I}$ can be a start point in result interval $R_k.s$. If $counter1 = 0$ and $counter2 = 0$ set $R_k.s = L_i.p$, increment $counter1$ and i.
- $L_i.t = 1start2$
  Start point of an interval from $\mathcal{I}'$ can be an end point in the result interval. If $counter2 = 0$, set $T_k.e = L_i.p$. If $R_k.s \neq 0$, increment k, (create a result) and set $R_k.s = 0$ and $T_k.e = 0$. Finish by incrementing $counter2$ and i.
- $L_i.t = 4end2$
  An end point of an interval from $\mathcal{I}'$ can be start point in the result interval. If $counter1 > 0$ and $counter2 = 1$, there is a period in the first input relation that is ended by the end point of the second. $L_i.p$ will be start point of the result interval. Decrement $counter2$, increment i.
- $L_i.t = 3end1$
  An end point of an interval from the $\mathcal{I}$ can be an end point in the result interval. If $counter1 = 1$, this time points closes a period in $\mathcal{I}$. $L_i.p$ will be end point in result interval, $T_k.e = L_i.p.\ If R_k.s \neq 0$ increment k, (create a result) and set $R_k.s = 0$ and $T_k.e = 0$. Finish by decrementing $counter$, incrementing i.

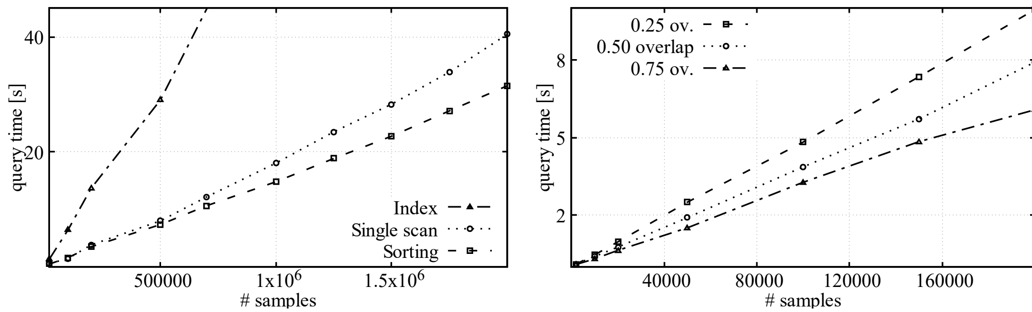


Figure 6 a) TUNION Comparison. b) TUNION Indexing on Statistical Data.

6 Evaluation

In this section we evaluate the implementations of the two selected temporal operators discussed in Section 5, TUNION and TDIFF. We use two kinds of datasets, statistically generated datasets and a real world dataset. In generated data the properties of overlapping temporal intervals can be adjusted by changing the length or position of intervals. This enables us to adapt the relationships of the generated intervals, i.e. change the ratio of overlapping, starting or containing intervals.

The real world dataset “Airline On-Time Statistics and Delay Causes” is provided by the Bureau of Transportation Statistics [18] and specifies the departure and landing timestamps of flights in the USA. It consists of roughly eight million entries per year and the flight times can be observed at a granularity of minutes.

For benchmark purposes we mainly use the commercial product Oracle Database 12c Standard Edition running on a Windows 7 Professional operating system. In order to compare our operators to the difference and union described in [9], we reimplemented TUNION and TDIFF with the open source product PostgreSQL 9.4 on Ubuntu 16.04 operating system (see Section 6.1.2 and 6.2.3). The software components are all executed on virtual hardware that has 8 GB of RAM and 3x3.4 GHz CPU cores assigned to and is running on solid state discs.

6.1 Temporal Union

Besides the real world dataset, different generated datasets with single nontemporal attributes are employed for the evaluation as well. These are comparable to the dataset of flights where a flight has an assigned flight number as nontemporal attribute and two timestamps for departure and arrival. The maximum size of input tables is two million samples. The overall result of comparing the different implementations of the TUNION operator is that sorting and single scan algorithms have the best overall performance (see Figure 6 a)) whereas the index-based implementation using basic b-trees performs very poorly.

Using Solid State Discs for evaluation does not affect the measurements significantly. This is due to the implementation of TUNION using mostly sequential disc reads.

6.1.1 Index vs. Sorting

Obviously, the index-based algorithm is highly dependent on the dataset’s structure by design. If there are large intervals overlapping smaller ones, the loop searching for maximum end points of merged intervals is executed fewer times. This is tested by creating statistical datasets that have different amounts of large intervals compared to the sample size. Figure 6

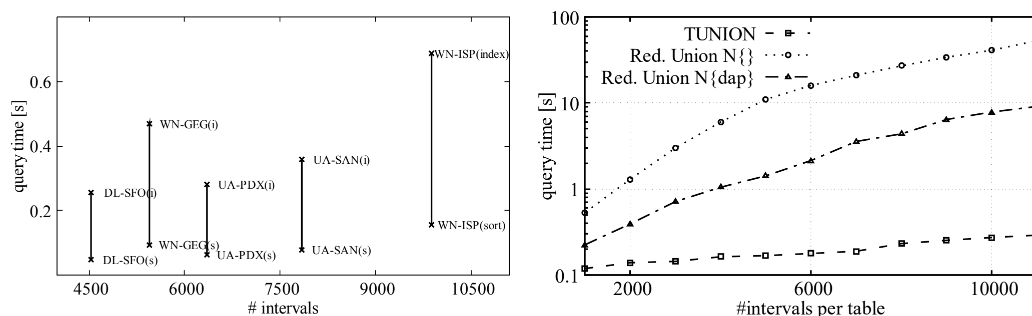


Figure 7 a) Index vs. Sorting on Flight Data. b) TUNION vs. Reduced Union.

b) shows the query over this dataset instanced with 200,000 intervals with no nontemporal attributes and data having 25%, 50% and 75% overlapping intervals.

Additionally, we take a look at the real life dataset ([18]). Airports departing flights mostly to airports located in short distances have less large flight time intervals than airports, which are in central regions or hubs with long routes to fly. This results in less overlaps and thus in higher execution times of indexed TUNION.

Figure 7 a) shows some combinations of airports and airlines queried with indexing and sorting. In this figure the upper points represent the query time using indexing, and the lower points show the query times using sorting. The lesser the distance between upper and lower points is, the more intervals are overlapped by large intervals and the more indexing approximates sorting.

6.1.2 TUNION vs. Reduced Union

To compare the execution times of [9] where temporal union is reduced to its nontemporal counterpart by temporal primitives (i.e. split, align, normalize) with our approach we took the proposed temporal extension for PostgreSQL [19] and reimplemented the TUNION-operator in this environment [14] as UDFs. Using the known flights dataset, Figure 7 b) shows the comparison of the sorting-based TUNION and the reduced union based on the normalizer function. We compared two scenarios $\mathcal{N}\{\}$ and $\mathcal{N}\{dap\}$, where in the first setting without a normalization attribute all overlapping intervals have to be split. In the latter the destination airport is used as normalization attribute which partitions the dataset by ten airports.

The reduced union produces a much larger set of results due to the temporal normalizers nature, which breaks every interval into disjoint subintervals in order to use standard SQL union on these subintervals and thus decreasing performance in comparison to sorting TUNION.

The results are only comparable to a certain degree, since reducing using temporal primitives and creating a set of very specialized operators do not follow the same approach. The first approach shows more aspects of generality, given that temporal selection, projection, aggregation, difference, union, and intersection can be reduced to nontemporal operators using a small set of temporal primitives. Our approach concedes this generality and focuses on very special operators with efficient implementations. Regarding the TUNION implementation, the use of a temporal splitter is not necessary.

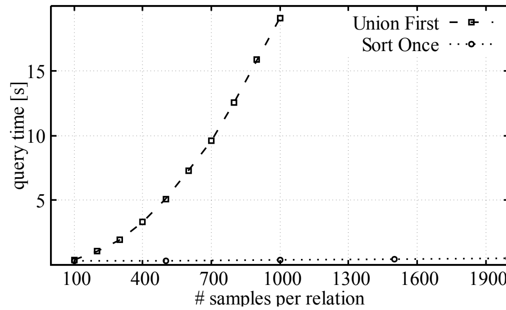


Figure 8 TDIFF - Union First vs. Sort Once.

6.2 Temporal Difference

Similar to Section 6.1 we use a real world dataset as well as generated datasets with a sample size of 500,000 in order to evaluate TDIFF operator variants.

6.2.1 Statistical Data

For the evaluation of TDIFF we first generate a dataset in a way that for every interval in the first interval set $\mathcal{I}$ we assign an interval in the second interval set $\mathcal{I}'$ that is neither adjacent nor overlapping. Modulating the position of the intervals of $\mathcal{I}'$ we then can change the relationship from 1) I'_i *after* I_i to 2) I_i *overlaps* I'_i to 3) I'_i *finishes* I_i to 4) I'_i *during* I_i, and use these scenarios to compute averages for various sample sizes.

Because of the structure of the *Union First* algorithm and the nested loop to be executed, the query time has to be in $O(n \log(n) + m \log(m) + nm)$ where n is the size of $\mathcal{I}$ and m is the size of $\mathcal{I}'$. Assuming that both input sets have similar sizes and few overlapping intervals that can be coalesced in the first step, this implementation performs rather poorly.

In comparison the *Sort Once* strategy has a much better performance. This is due to the fact that after the sorting in $O(n \log(n) + m \log(m))$ time the algorithm sweeps in linear $O(n + m)$ time over the ordered time points.

Figure 8 shows the query times of *Union First* and *Sort Once*. We are using only 0.5% of entries of the 500k dataset for visualization purposes. We are discarding the *Union First* algorithm further on and evaluate with *Sort Once* only in the next section.

6.2.2 Real World Dataset

Suppose an airline company wants to get an overview of the landing delays of its flights. The company is especially interested in when they are happening, and how long those delays are. The desired delay intervals can be elegantly computed by applying TDIFF($\mathcal{I}'$,$\mathcal{I}$) to interval sets, where $\mathcal{I}$ are all flights assigned to that airline from the flight plan and $\mathcal{I}'$ are the time intervals of planned departure and and actual landing data. Using this scenario, Figure 9 a) shows the query times of different airlines in the first quarter of the year 2008 based on the *Sort Once* algorithm. The query times on a real world dataset show similarity to those of generated averaged data we used in the previous section.

6.2.3 TDIFF vs. Reduced Difference

In [9] it is shown how the temporal difference can be reduced to its nontemporal difference operator. Figure 9 b) shows the comparison of the TDIFF operator and the nontemporal

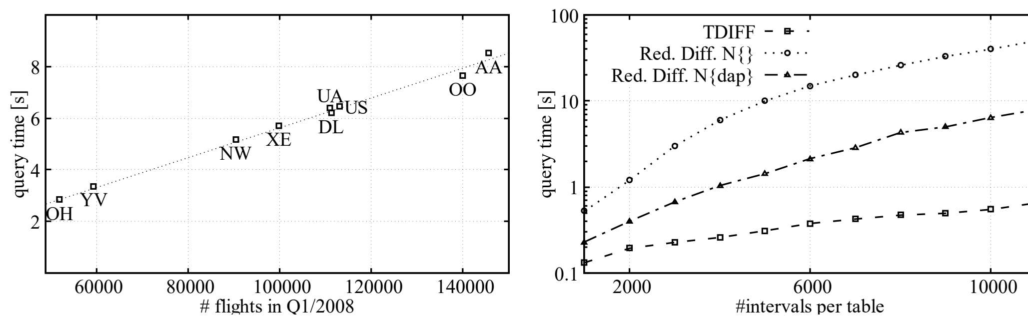


Figure 9 a) TDIFF - Sort Once with Flight Data. b) TDIFF vs. Reduced Difference.

difference with normalizer function using the combined generated datasets as well as the known flights dataset analogue to Section 6.1.2. Likewise, the reduced difference produces a much larger set of results due to the temporal normalizers nature, which breaks every interval into disjoint subintervals in order to use standard SQL minus operator on these subintervals and thus decreasing performance in comparison to TDIFF.

7 Conclusions

In this work we discussed the development of a library of temporal operators for processing sets of temporal intervals. To this end, we provided the formal description of various operators and discussed their implementation by means of table functions. The resulting operators allow to formulate complex temporal queries in an elegant way and open the possibility for further algebraic optimization. In fact, the provided operators can be seen as a foundation of a situation calculus necessary for achieving situation-awareness inside a database system. We have studied and evaluated different algorithms for implementing the operators TUNION and TDIFF without the need for considerable changes to an existing database schema or even database kernel. One particular interesting finding is the fact that the index-based algorithm performed poorly in comparison with the other implementations. However, we assume that performance can be increased dramatically by using specialized index structures as proposed in ([3], [8], [16]). Improving existing algorithms of the proposed temporal operators by implementing specific data structures as well as extending the library of table functions is subject of future work.

References

1 Mohamed H. Ali, Badrish Chandramouli, Balan Sethu Raman, and Ed Katibah. Spatio-temporal stream processing in microsoft streaminsight. *IEEE Data Eng. Bull.*, 33(2):69–74, 2010.

2 James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

3 Chuan-Heng Ang and Kok-Phuang Tan. The interval b-tree. *Information Processing Letters*, 53(2):85–89, 1995.

4 Andreas Behrend, Philip Schmiegelt, and Andreas Dohr. Supporting situation awareness in spatio-temporal databases. *Datenbank-Spektrum*, 16(3):207–218, 2016.

5 Michael H. Böhlen, Renato Busatto, and Christian S. Jensen. Point-versus interval-based temporal data models. In *Proc. of ICDE*, pages 192–200, 1998.

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

7 Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. Coalescing in temporal databases. In *VLDB 1996*, pages 180–191, 1996.

8 Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing valid time intervals. In *DEXA 1998*, pages 541–550, 1998.

9 Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *SIGMOD*, pages 433–444, 2012.

10 Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. *ACM Trans. Database Syst.*, 41(4):26:1–26:46, 2016.

11 Curtis E. Dyreson. Temporal coalescing with now, granularity, and incomplete information. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 169–180. ACM, 2003. doi:10.1145/872757.872779.

12 Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 725–736. ACM, 2013. URL: http://dl.acm.org/citation.cfm?id=2463676, doi:10.1145/2463676.2465282.

13 Dieter Gawlick, Eric S. Chan, Adel Ghoneimy, and Zhen Hua Liu. Mastering situation awareness: The next big challenge? *SIGMOD Record*, 44(3):19–24, 2015.

14 The PostgreSQL Global Development Group. PostgreSQL 9.4.12 Documentation, 2017. [Online; accessed 28-May-2017]. URL: https://www.postgresql.org/docs/9.4/static/index.html.

15 Christian S. Jensen, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3):35–43, 1992.

16 Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *VLDB 2000*, pages 407–418, 2000.

17 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 2013*, pages 439–455, 2013.

18 Bureau of Transportation Statistics. Airline On-Time Statistics and Delay Causes, 2017. [Online; accessed 7-February-2017]. URL: https://www.transtats.bts.gov.

19 University of Zurich. Temporal PostgreSQL, 2017. [Online; accessed 28-May-2017]. URL: http://tpg.inf.unibz.it/downloads/postgresql-9.6beta3-temporal.tar.gz.

20 Oracle. Oracle Database 12c Release 2 Online Documentation, 2017. [Online; accessed 7-February-2017]. URL: http://docs.oracle.com/database/122/index.htm.

21 Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

22 Xin Zhou, Fusheng Wang, and Carlo Zaniolo. Efficient temporal coalescing query support in relational database systems. In *DEXA 2006*, pages 676–686, 2006.