Temporal Data Management – An Overview

Michael H. Böhlen[1], Anton Dignös[2], Johann Gamper[2(✉)], and Christian S. Jensen[3]

[1] University of Zurich, Zurich, Switzerland
[2] Free University of Bozen-Bolzano, Bolzano, Italy
gamper@inf.unibz.it
[3] Aalborg University, Aalborg, Denmark

Abstract. Despite the ubiquity of temporal data and considerable research on the effective and efficient processing of such data, database systems largely remain designed for processing the current state of some modeled reality. More recently, we have seen an increasing interest in the processing of temporal data that captures multiple states of reality. The SQL:2011 standard incorporates some temporal support, and commercial DBMSs have started to offer temporal functionality in a step-by-step manner, such as the representation of temporal intervals, temporal primary and foreign keys, and the support for so-called time-travel queries that enable access to past states.

This tutorial gives an overview of state-of-the-art research results and technologies for storing, managing, and processing temporal data in relational database management systems. Following an introduction that offers a historical perspective, we provide an overview of basic temporal database concepts. Then we survey the state-of-the-art in temporal database research, followed by a coverage of the support for temporal data in the current SQL standard and the extent to which the temporal aspects of the standard are supported by existing systems. The tutorial ends by covering a recently proposed framework that provides comprehensive support for processing temporal data and that has been implemented in PostgreSQL.

1 Introduction

The capture and processing of temporal data in database management systems (DBMS) has been an active research area since databases were invented. In the temporal database research history, four overlapping phases can be distinguished. First, the *concept development* phase (1956–1985) concentrated on the study of multiple kinds of time and temporal aspects of data and on temporal conceptual modeling. The following phase was dedicated to the *design of query languages* (1978–1994), including relational and object-oriented temporal query languages. Then the focus shifted to *implementation aspects* (1988–present), emphasis being on storage structures, algorithms for specific operators, and temporal indices.

E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 51–83, 2018.
https://doi.org/10.1007/978-3-319-96655-7_3

Finally, the *consolidation* phase (1993–present) produced a consensus glossary of temporal database concepts [47], a query language test suite [36], and TSQL2 [81] as an effort towards standardization of a temporal extension to SQL.

A number of events and activities involving the temporal database community have impacted the evolution of temporal database research significantly. The 1987 *IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems* [78] covered a wide range of topics, including requirements for temporal data models and information systems, temporal query languages, versioning, implementation techniques, temporal logic, constraints, and relations to natural language. The 1993 *ARPA/NSF International Workshop on an Infrastructure for Temporal Databases* [80] aimed at consolidating different temporal data models and query languages. In the same year, the collection *Temporal Databases: Theory, Design, and Implementation* [87] was published, which describes primarily a number of data models and query languages. Another influential book, *The TSQL2 Temporal Query Language* [81], was published in 1995. By leveraging many of the concepts that were proposed in previous research, TSQL2 aimed to be a consensus data model and query language. At the same time, a project with the ambition of creating a new part of the SQL standard dedicated to the support of temporal data started. Several proposals were submitted, e.g., [83], but were eventually not successful. The proposals proved controversial, and they were unable to achieve support from major database vendors. The last notable event dedicated to temporal database research was the 1997 *Dagstuhl Seminar on Temporal Databases* [37] that had as its aim to discuss future directions for temporal database management both in research as well as in system development.

The last several years have seen a renewed interest in temporal database management in both academia and industry. This interest is driven in part by the needs of new and emerging applications, such as versioning of web documents [33], social network analysis and communication networks [65,76], management of normative texts [46], air traffic monitoring and patient care [7], video surveillance [71], sales analysis [70], financial market analysis [45], and data warehousing and analytics [82], to name a few. These and other applications produce huge amounts of temporal data, including time series and streaming data, which are special forms of temporal data. It has been recognized [5] that analyses of historical data can reveal valuable information that cannot be found in only the current snapshot.

This tutorial provides an overview of temporal data management concepts and techniques, covering both research results and commercial database management systems. In Sect. 2, we summarize the most important concepts developed in temporal database research. In Sect. 3, we provide a brief overview of the state-of-the-art in temporal database research. Section 4 describes the most important temporal features of the SQL:2011 standard that introduces temporal support into SQL. In Sect. 5, we provide a brief overview of the support for the temporal SQL standard in commercial database management systems. Finally, in Sect. 6, we describe a recent framework that provides a comprehensive and native solution for processing temporal queries in relational database management systems.

2 Basic Concepts of Temporal Databases

In this section, we briefly summarize important concepts that have been developed in temporal database research.

2.1 Time Domain and Structure

The *time domain* (or ontology) specifies the basic building blocks of time [66]. It is generally modeled as a set of *time instants* (or points) with an imposed partial order, e.g., $(\mathbb{N}, <)$. Additional axioms impose more structure on the time domain, yielding more refined time domains. *Linear time* advances from past to future in a step-by-step fashion. This model of time is mainly used in the database area. In contrast, AI applications often used a *branching time* model, which has a tree-like structure, allowing for possible futures. Time is linear from the past to now, where it divides into several time lines; along any future path, additional branches may exist. This yields a tree-like structure rooted at now. *Now* marks the current time point and is constantly moving forward [32]. The time domain can be *bounded* in the past and/or in the future, i.e., a first and/or last time instant exists; otherwise, it is called *unbounded*.

The time domain can be dense, discrete, or continuous. In a *discrete* time domain, time instants are non-decomposable units of time with a positive duration, called *chronons* [31]. A chronon is the smallest duration of time that can be represented. This time model is isomorphic to the natural numbers. In contrast, in a *dense* time domain, between any two instants of time, there exists another instant; this model is isomorphic to the rational numbers. Finally, *continuous* time is dense and does not allow "gaps" between consecutive time instants. Time instants are durationless. The continuous time model is isomorphic to the real numbers.

While humans perceive time as continuous, a *discrete linear time model* is generally used in temporal databases for several practical reasons, e.g., measures of time are generally reported in terms of chronons, natural language references are compatible with chronons, and any practical implementation needs a discrete encoding of time. A limitation of a discrete time model is, for example, the inability to represent continuous phenomena [40].

A *time granularity* is a partitioning of the time domain into a finite set of segments, called *granules*, providing a particular discrete image of a (possibly continuous) timeline [9,10]. The main aim of granularities is to support user-friendly representations of time. For instance, birth dates are typically measured at the granularity of days, business appointments at the granularity of hours, and train schedules at the granularity of minutes. Multiple granularities are needed in many real-world applications.

2.2 Temporal Data Models

A *data model* is defined as $M = (DS, QL)$, where DS is a set of data structures and QL is a language for querying instances of the data structures. For instance,

the relational data model is composed of relations and, e.g., SQL. Many extensions of the relational data model to support time have been proposed in past research, e.g., IXSQL [63], TSQL2 [81], ATSQL [16] and SQL/TP [92]. When designing a temporal data model [49], several aspects have to be considered, such as

- different time dimensions, or temporal aspects,
- different timestamp types, and
- different forms of timestamping.

Time Dimensions. Different temporal aspects of data are of interest. Valid time and transaction time are the two aspects that have attracted the most attention by far in database research; other temporal aspects include publication time, efficacy time, assertion time, etc.

Valid time [54] is the time when a *fact was/is/will be true in the modeled reality*, e.g., John was hired from October 1, 2014 to May 31, 2016. Valid time captures the time-varying states of the modeled reality and is provided by the application or user. All facts have a valid time by definition, and it exists independently of whether the fact is recorded in a database or not. Valid time can be bounded or unbounded.

Transaction time [53] is the time when a *fact is current/present in the database* as stored data, e.g., the fact "John was hired from October 1, 2014 to May 31, 2016" was stored in the database on October 5, 2014, and was deleted on March 31, 2015. Transaction time captures the time-varying states of the database, and it is supplied automatically by the DBMS. Transaction time has a duration from the insertion of a fact to its deletion, with multiple insertions and deletions being possible for the same fact. Deletions of facts are purely logical: the fact remains in the database, but ceases to be part of the database's current state. Transaction time is always bounded on both ends. It starts when the database is created (nothing was stored before), and it does not extend past now (it is not known which facts are current in the future). Transaction time is the basis for supporting accountability and traceability requirements, e.g., in financial, medical, or legal applications.

A data model can support none, one, two, or more time dimensions. A so-called *snapshot* data model provides no support for time dimensions and records only a single snapshot of the modeled reality. A *valid time* data model supports only valid time, a *transaction time* data model only transaction time. A *bitemporal* data model supports both valid time and transaction time.

Timestamp Types. A *timestamp* is a time value that is associated with an attribute value (*attribute (value) timestamping*) or a tuple (*tuple timestamping*) in a database and captures some temporal aspect, e.g., valid time or transaction

time. It can be represented as one or more attributes or columns of a relation. Three different *types of timestamps* [50,51,62] have received particular attention:

- time points,
- time intervals, and
- temporal elements.

To illustrate different types of timestamps, we consider a car rental company, where customers, identified by a `CustID`, rent cars, identified by a `CarID`. Assume the following rentals during May 1997:

- On 3rd of May, customer `Sue` rents car `C1` for three days.
- On 5th of May, customer `Tim` rents car `C2` for 3 days.
- From 9th to 12th of May, customer `Tim` rents car `C1`.
- From 19th to 20th of May, and again from 21st to 22nd of May, customer `Tim` rents car `C2`.

These rentals are stored in a relation `Rental`, which is illustrated in Fig. 1.

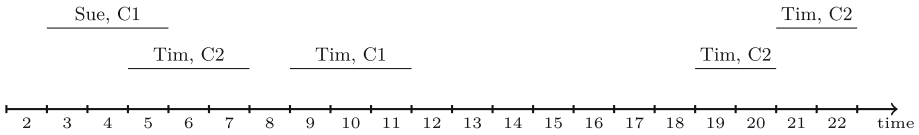


Fig. 1. Relation `Rental`.

Time Points. In a point-based data model, tuples or attribute values are timestamped with a *time point* (or instant) (cf. Fig. 2(a)). This is the most basic and simple data model. Timestamps are atomic values and can be compared easily with $=, \neq, <, >, \geq, \leq$. Multiple tuples are used if a fact is valid at several time points, e.g., four tuples for the two consecutive rentals from time 19 to time 22. Additional attributes are required to restore the original relation. In the `Rental` relation in Fig. 2(a), the `SeqNo` attribute is used to group tuples that constitute a rental. Without this attribute, it would be impossible to restore the two consecutive 2-day rentals, as they could be restored, e.g., as a single 4-day rental or four 1-day rentals. The point-based model is simple and provides an abstract view of a database, which makes it popular for theoretical studies, but inappropriate for physical implementation.

Time Intervals. In an interval-based data model, each tuple or attribute is timestamped with a *time interval, or period* (cf. Fig. 2(b)). Timestamps can be compared using Allen's 13 basic interval relationships (before, meets, during, etc.) [4], which is more convenient than comparing the endpoints of the intervals. Multiple tuples are used if a fact is valid during disjoint time intervals. The `SeqNo` attribute is not needed to distinguish among different tuples. This is the most popular model from an implementation perspective. Interval timestamps are *not closed* under set operations, e.g., subtracting the interval $[5, 7]$ from the interval $[1, 9]$ gives the set of intervals $\{[1, 4], [8, 9]\}$, not a single interval.

Rental

SeqNo	CustID	CarID	T
1	Sue	C1	3
1	Sue	C1	4
1	Sue	C1	5
2	Tim	C2	5
2	Tim	C2	6
2	Tim	C2	7
3	Tim	C1	9
3	Tim	C1	10
3	Tim	C1	11
3	Tim	C1	12
4	Tim	C2	19
4	Tim	C2	20
5	Tim	C2	21
5	Tim	C2	22

(a) Point-based model

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7]
Tim	C1	[9,12]
Tim	C2	[19,20]
Tim	C2	[21,22]

(b) Strong interval-based model

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7]
Tim	C1	[9,12]
Tim	C2	[19,22]

(c) Weak interval-based model

Fig. 2. Point- and interval-based data models for `Rental` relation.

Temporal Elements. In data models with temporal elements, each tuple or attribute is timestamped with a finite union of intervals, called a *temporal element* [38,39] (cf. Fig. 3). The full history of a fact is stored in a single tuple. For instance, the second tuple represents the fact that Tim rents car C1 from time 5 to 7 and from time 19 to 22. Temporal elements support only a point-based semantics, and an additional attribute would be necessary to distinguish between the two consecutive 2-day rentals (see also the discussion below).

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7] ∪ [19,22]
Tim	C1	[9,12]

Fig. 3. Data model with temporal elements.

Point-Based and Interval-Based Semantics. From a semantic viewpoint, two different types of models can be distinguished: models with point-based semantics and models with interval-based semantics. This distinction is orthogonal to the choice of the timestamp (i.e., time points, time intervals, or temporal elements) and focuses on the meaning of the timestamps. For instance, relation `Rental` in Fig. 2(a) uses time points as timestamps, but adopts an interval-based semantics, as information on the rental periods is preserved by using the additional `SeqNo` attribute. Similarly, a relation that uses interval timestamps may adopt either a point-based semantics or an interval-based semantics. The corresponding models are referred to as, respectively, weak interval-based model and strong interval-based model – see Figs. 2(b) and (c).

In the *weak interval-based data model*, intervals are only used as a compact and convenient representation of contiguous sets of time points. For instance, although (syntactically) different, the two relations in Figs. 2(b) and (c) are considered equivalent under point-based semantics since they are snapshot equivalent [48], i.e., they contain the same snapshots. More formally, let r and s be two temporal relations, Ω^T be the time domain, and $\tau_t(\mathsf{r})$ be the timeslice operator [52] with t being a time instant. The relations r and s are *snapshot equivalent* if and only if

$$\forall t \in \Omega^T : \tau_t(\mathsf{r}) \equiv \tau_t(\mathsf{s})$$

For the weak interval-based model, an important operation is *coalescing* [2,19,99]. Coalescing is the process of merging adjacent and overlapping interval timestamped tuples with identical nontemporal attribute values into tuples with maximal time intervals. For instance, the relation in Fig. 2(c) is the result of coalescing the relation in Fig. 2(b). Without an additional `SeqNo` attribute, the two consecutive 2-day rentals disappear: they are merged into a single 4-day rental.

In the *strong interval-based data model*, intervals are *atomic* units that carry meaning (and not just sets of time points). Thus, strong interval-based data models are more expressive. They can distinguish between a 4-day rental and two consecutive 2-day rentals without requiring an additional attribute. The relation in Fig. 2(b) is the appropriate representation of the `Rental` relation in our example since two 2-day rentals and one 4-day rental might impose different fees.

Timestamping. Timestamping denotes the association of a data element in a relation with a time value. In the above examples, we used *tuple timestamping*, which associates each tuple with a time value such as a time point, a time interval, or a temporal element.

In *attribute (value) timestamping*, each attribute value in a relation is associated with a timestamp (cf. Fig. 4). Relations are *grouped* by an attribute, and *all information* about that attribute (or real-world object) is captured in a single tuple. Information about other objects is spread across several tuples. In Fig. 4(a), all information about a customer is in one tuple, while the information about cars is spread across several tuples. A single tuple may record multiple facts. For instance, the second tuple records four different rentals involving customer `Tim` and the cars `C1` and `C2`. Different groupings of the information into tuples are possible. Figure 4(b) shows the same relation grouped on `CarID`. The two relations are snapshot-equivalent. Data models using attribute value timestamping are non-first-normal-form data models.

2.3 Query Language Semantics

The querying capabilities of temporal DBMSs can be partitioned into three modes [16,82,84]: nonsequenced, current, and sequenced semantics.

Rental

SeqNo	CustID	CarID
[3,5] 1	[3,5] Sue	[3,5] C1
[5,7] 2	[5,7] ∪ [9,12] ∪ [19,22] Tim	[5,7] ∪ [19,22] C2
[9,12] 3		[9,12] C1
[19,20] 4		
[21,22] 5		

(a) Grouped by `CustID`

Rental

SeqNo	CustID	CarID
[3,5] 1	[3,5] Sue	[3,5] ∪ [9,12] C1
[9,12] 3	[9,12] Tim	
[5,7] 2	[5,7] ∪ [19,22] Tim	[5,7] ∪ [19,22] C2
[19,20] 4		
[21,22] 5		

(b) Grouped by `CarID`

Fig. 4. Attribute value timestamping.

The *nonsequenced semantics* [18] is time agnostic, that is, the DBMS does not enforce any specific meaning on the timestamps, and applications must explicitly specify how to process the temporal information. The support for the nonsequenced semantics in DBMSs is limited to extending SQL with new data types, predicates, and functions. Predicates such as `OVERLAPS`, `BEFORE`, and `CONTAINS` are part of the SQL:2011 standard. Another approach to specify temporal relationships are to use the operators of temporal logic, which target the reasoning across different database states [23]. Nonsequenced semantics is the most flexible and expressive semantics since applications handle timestamps like all other attributes without any implicit meaning being enforced.

The *current semantics* [6,17] performs query processing on the database snapshot at the current time and can be realized by restricting the data to the current time. Current semantics is present in the SQL:2011 standard, where standard SQL queries over transaction time tables (in SQL:2011 called system-versioned tables) are evaluated on the current snapshot [58]. As a simple extension to current semantics, so-called time travel queries allow to specify any snapshot of interest. The integration of current semantics into a database engine is usually done with the help of selection operations.

The *sequenced semantics* [15,44] of a temporal query is defined by viewing a temporal database as a sequence of snapshot databases and evaluating the query at each of these snapshots. This concept is known as *snapshot reducibility* [63,86]. More formally, let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, ψ^T be an n-ary temporal operator, ψ be the corresponding nontemporal operator, Ω^T be the time domain, and $\tau_t(\mathsf{r})$ be the timeslice operator [52] with t being a time instant. Operator ψ^T is *snapshot reducible* to ψ if and only if

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

Snapshot reducibility provides a minimum requirement for sequenced semantics by constraining the result of a temporal query to be consistent with the snapshots that are obtained by computing the corresponding nontemporal query on each snapshot of the temporal database. This provides a clear semantics for theoretical studies, but a practical implementation needs additional constraints.

First, snapshot reducibility does not constrain the coalescing of consecutive tuples with identical nontemporal attribute values. For instance, the two relations in Figs. 2(b) and (c) are snapshot equivalent, yet they store different information. *Change preservation* [27,30] is a way to determine the time intervals of the result tuples, and thus control the coalescing of tuples. A new time interval is created when the argument tuples that contribute to a result tuple change (i.e., have different lineage or provenance) [20,24], yielding maximal time intervals for the result tuples over which the argument relations are constant.

Second, snapshot reducibility does not allow temporal operators to reference the timestamps of the argument relations since the intervals are removed by the timeslice operator. For example, computing the average duration of projects at each point is not possible. This problem can be tackled by propagating the original timestamp as additional attribute to relational algebra operators, yielding a concept known as *extended snapshot reducibility* [15].

Finally, sometimes attribute values need to be changed when the timestamp intervals of tuples change. For instance, if a project budget is 100,000 for a period of two years, then the corresponding budget for one year should be 50,000 (assuming a uniform distribution). This concept is called *scaling* of attribute values [11,28].

3 State-of-the-Art

In this section, we discuss the state-of-the-art in temporal database research, focusing on data models, SQL-based query languages, and evaluation algorithms for query processing.

3.1 Data Models and SQL-Based Query Languages

To make the formulation of temporal queries more convenient, various temporal query languages [14,87] have been proposed. The earliest and simplest approach to add temporal support to SQL-based query languages was to introduce new data types with associated predicates and functions that were strongly influenced by Allen's interval relationships [4]. While this approach facilitates the formulation of some temporal queries, it falls short in the extent to which it makes it easier to formulate temporal queries. Therefore, new constructs were added to SQL with the goal of expressing temporal queries more easily. A representative query language following this approach is TSQL2 [81], which uses so-called syntactic defaults to facilitate query formulation. Challenges with this type of approach include to be "complete" in enabling easy formulation of temporal queries and to avoid unintentional interactions between the extensions.

A more systematic approach was adopted in IXSQL [25,63], which normalizes interval timestamped tuples for query processing and works as follows: (i) a function *unfold* transforms an interval timestamped relation into a point timestamped relation by splitting each tuple into a set of point timestamped tuples; (ii) the corresponding nontemporal operation is applied to the normalized relation; (iii) a function *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 does not respect lineage, and no efficient implementation exists.

SQL/TP [92,93] is an approach that is based on a point-based data model: a temporal relation is modeled as a sequence of nontemporal relations (or snapshots). To evaluate a temporal query, the corresponding nontemporal query is evaluated at each snapshot. For an efficient evaluation, an interval encoding of point timestamped relations was proposed together with a *normalization* function. The normalization splits overlapping value-equivalent input tuples into tuples with equal or disjoint timestamps, on which the corresponding nontemporal SQL statements are executed. SQL/TP considers neither lineage nor extended snapshot reducibility, which are not relevant for point timestamped relations. Moreover, the normalization function is not applicable to joins, outer joins, and anti joins.

Agesen et al. [1] extend normalization to bitemporal relations by means of a *split* operator. This operator splits input 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 split operator supports temporal aggregation and difference in now-relative bitemporal databases.

ATSQL [16] offers a systematic way to construct temporal SQL queries from nontemporal SQL queries. The main idea is to first formulate the nontemporal query and then prepend to this query a so-called *statement modifier* that specifies the intended semantics of the query evaluation, such as sequenced or nonsequenced semantics.

The *temporal alignment* approach [27,30] is a solution for computing temporal queries over interval timestamped relations using sequenced semantics. The key idea is to first adjust the timestamps of the input tuples and then to execute the corresponding nontemporal operator to obtain the intended result. While the adjustment of timestamps is similar to the normalization in SQL/TP [92], the temporal alignment approach is comprehensive and offers snapshot reducibility, extended snapshot reducibility, and attribute value scaling for all operators of a relational algebra. This approach provides a native database implementation for temporal query languages with sequenced semantics, such as ATSQL. More details are provided in Sect. 6.

The scaling of attribute values in response to the adjustment of interval timestamps has received little attention. Böhlen et al. [11] propose three different attribute characteristics: *constant* attributes that never change value during query processing, *malleable* attributes that require adjustment of the value when the timestamp changes, and *atomic* attributes that become undefined (invalid)

when the timestamp changes. For malleable attributes, an adjustment function is proposed. Terenziani and Snodgrass [91] distinguish between *atelic* facts that are valid for each point in time and *telic* facts that are only valid for one specific interval. That work focuses on the semantics of facts recorded in a database and proposes a three-sorted relational model (atelic, telic, nontemporal). Dignös et al. [28] show that scaling of attributes values is possible during query processing.

The focus of Dyreson et al. [34,35] is to provide a uniform framework for the evaluation of queries under different temporal semantics, including the two extremes of sequenced and nonsequenced semantics. Additional semantics can be realized in this framework, such as context, periodic, and preceding semantics. The framework uses lineage to track tuples through operations. The work is primarily at the conceptual level, the main goals being to unify and reconcile different temporal semantics.

3.2 Query Processing Algorithms

In terms of query processing, various query algorithms for selected operators have been studied, primarily for temporal aggregations (e.g., [13,57,67,95,97]) and temporal joins (e.g., [42,85,98]) over interval timestamped relations, which are arguably the most important and expensive operations.

Processing Temporal Aggregations. Aggregate functions enable the summarization of large volumes of data, and they were introduced in early relational DBMSs such as System R and INGRES. Various forms of temporal aggregation have been proposed since then. They differ in how the data is grouped along the time dimension [41]. In *instantaneous temporal aggregation*, an aggregate function is conceptually computed at each time point, followed by a subsequent coalescing step to merge contiguous tuples with the same aggregate value into a single interval timestamped tuple. *Moving-window temporal aggregation*, also termed cumulative temporal aggregation, works similarly, except that an aggregate at a time point is computed over all tuples that occur within a user-specified window. Finally, in *span temporal aggregation*, the aggregates are computed over sets of tuples that overlap with fixed time intervals specified by the user.

The earliest proposal aimed at the efficient processing of instantaneous temporal aggregates is by Tuma [94]. Following Tuma's work, research focused on the development of efficient main-memory algorithms for the evaluation of instantaneous temporal aggregates as the most important form of temporal aggregation. Key works in this direction include the aggregation tree algorithm [57] and the balanced tree algorithm [67].

With the diffusion of data warehouses and OLAP, disk-based index structures for incremental computation and maintenance of temporal aggregates were investigated. Notable works include the SB-tree by Yang and Widom [95], which was extended to the MVSB-tree by Zhang et al. [97] to include nontemporal range predicates. The high memory requirements of the MVSB-tree were addressed by Tao et al. [88], proposing two approximate solutions for temporal aggregation.

Vega Lopez et al. [61] formalized temporal aggregation in a unified framework that enables the comparison of the different forms of temporal aggregation based on various mechanisms for defining aggregation groups. In a similar vein, Böhlen et al. [13] propose a framework that generalizes existing forms of temporal aggregation by decoupling the partitioning of the time line from the specification of the aggregation groups.

The development of efficient temporal aggregation algorithms has recently received renewed interest. Kaufmann et al. [55,56] propose the *timeline index* to efficiently support query processing, including instantaneous temporal aggregation, in the main memory DBMS HANA. The timeline index is a general data structure that instead of intervals uses start and end points of the intervals. Query processing is performed by scanning sorted lists of endpoints. Piatov and Helmer [74] present a family of plane-sweeping algorithms that adopt the timeline index for other forms of temporal aggregation, such as aggregation over fixed intervals, sliding window aggregates, and MIN/MAX aggregates.

Temporal aggregation has been studied for different query languages and data models. Böhlen et al. [12] investigate how temporal aggregation is supported in different types of temporal extensions to SQL. Selected temporal aggregations are also found in non-relational query languages, such as XML, e.g., τXQuery [43].

Processing Temporal Joins. The overall efficiency of a query processor depends highly on its ability to evaluate joins efficiently, as joins occur frequently. Two classes of join algorithms can be distinguished: solutions that rely on indexing or secondary access paths, and solutions for ad-hoc join operations that operate on the original tables, but might take advantage of sorting the data.

Gao et al. [42] present a comprehensive and systematic study of join operations in temporal databases as of 2005, covering both semantics and implementation aspects. In addition to providing formal definitions of various join operations, the paper classifies existing evaluation algorithms along the following dimensions: nested-loop, partitioning, sort-merge, and index-based. The work includes also an experimental performance evaluation of 19 join algorithms.

Recently, a number of new studies on the efficient evaluation of temporal joins have been published. The *timeline index* by Kaufmann et al. [55,56] is a main memory index structure that supports also temporal joins where matching tuples must be overlapping.

The *overlap interval partition* join algorithm by Dignös et al. [29] partitions the input relations in such a way that the percentage of matching tuples in corresponding partitions is maximized. This yields a robust join algorithm that is not affected by the distribution of the data. The proposed partitioning works both in disk-based and main-memory settings.

The *lazy endpoint-based interval* join algorithm by Piatov et al. [75] adopts the timeline index. After creating a timeline index of the input relations, the two index structures are scanned in an interleaved fashion. Thereby, active tuples are managed by an in-memory hash map, called a *gapless hash map*, that is optimized

for sequential reads of the entire map. Additionally, a lazy evaluation technique is used to minimize the number of scans of the active tuple map.

The *disjoint interval partitioning* join algorithm by Cafagna and Böhlen [22] first creates so-called *disjoint* partitions for each relation, where all tuples in a partition are temporally disjoint. To compute a temporal join, all outer partitions are then sort-merge-joined with each inner partition to produce the final result. Since tuples within a partition are disjoint, the algorithm is able to avoid expensive backtracking.

Bouros and Mamoulis [21] implement a forward-scan based *plane sweep algorithm* for temporal joins and provide two optimizations. The first optimization groups consecutive tuples such that join results can be produced in batches in order to avoid redundant comparisons. The second optimization extends the grouping with a bucket index to further reduce the number of comparisons. A major contribution of this work is a parallel evaluation strategy based on a domain-based partitioning of the input relations.

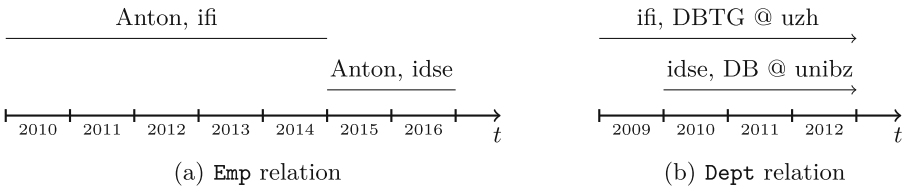
4 Temporal Support in the SQL:2011 Standard

This section summarizes the most important temporal features of the SQL:2011 standard, which is the first SQL standard with support for time.

The ability to create and manipulate tables whose rows are associated with one or two temporal periods, representing valid and transaction time, is the key extension in SQL:2011 [58,96]. A core concept of this extension is the *specification of time periods* associated with tables, bundled with support for updates, deletions, and integrity constraints. The support for querying temporal relations is limited to simple range restrictions and predicates.

In the following discussion, we use two valid time relations, which are illustrated in Fig. 5 and initially contain the following data:

- an employee relation, named `Emp`, records that Anton was working in the `ifi` department from 2010 to 2014 and in the `idse` department from 2015 to 2016;
- a department relation, named `Dept`, contains descriptions of the `ifi` and `idse` departments.



(a) `Emp` relation (b) `Dept` relation

Fig. 5. Graphical illustration of the `Emp` and `Dept` relations.

4.1 Creation of Tables with Time Periods

SQL:2011 adopts an interval-based data model with tuple timestamping. Rather than introducing a new data type for a time interval, a *time period specification* is added as metadata to the table schema. A period specification combines a physical *start time* attribute and an *end time* attribute to form a period. Periods are specified with the `PERIOD FOR` clause:

```
PERIOD FOR <PeriodName> (<StartTime>, <EndTime>)
```

Here, `<StartTime>` and `<EndTime>` are of type `DATE` or `TIMESTAMP` and together form a time period, which can be referred to by the name `<PeriodName>`. A period is by default a *closed-open* interval, where `StartTime` is included and `EndTime` is excluded[1].

By designing periods as metadata, a minimally invasive approach was chosen that achieves *backward compatibility*, i.e., old schemas, queries, and tools still work. If start and end time points are already stored in a table (which is often the case), a time period can be added without the need to modify the physical table schema. A new data type would require more substantial changes across the DBMS.

SQL:2011 supports valid time and transaction time, which are called, respectively, *application time* and *system time*. At most one application time and one system time can be specified for a table. Tables that have an application time period are called *application time period tables*, and tables with a system time period are called *system time period tables*. Tables that support both time dimensions are usually called bitemporal tables, although SQL:2011 does not provide an explicit name.

To facilitate reading and to be consistent with the research literature, we will use the widely adopted terms valid time for application time and transaction time for system time.

Valid Time Tables. Valid time tables store for each tuple the time interval when the tuple is true in the modeled reality. In our example, the schema of the `Emp` relation with a valid time attribute can be created as follows:

```
CREATE TABLE Emp (
  EName VARCHAR,
  EDept VARCHAR,
  EStart DATE,
  EEnd DATE,
  PERIOD FOR EPeriod (EStart, EEnd)
);
```

The `PERIOD FOR` clause specifies a valid time interval named `EPeriod`, which is built from the physical attributes `EStart` and `EEnd`. Notice that `EPeriod` is not

[1] To comply with the SQL:2011 standard, in this section we use closed-open intervals, whereas in the other sections we use closed-closed intervals.

a physical column of the table, but it is stored as metadata and can be used to refer to the interval. A table with this schema is shown in Fig. 6(a).

When *inserting* tuples, the user has to specify the valid time period in addition to the nontemporal attribute values of the tuples. For instance, the following statements insert two tuples indicating that Anton was working in the `ifi` department from 2010 to 2014 and in the `idse` department from 2015 to 2016:[2]

```
INSERT INTO Emp VALUES (Anton, ifi, 2010, 2015);
INSERT INTO Emp VALUES (Anton, idse, 2015, 2017);
```

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2015
Anton	idse	2015	2017

(a) Valid time table

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DB @ unibz	2010	9999

(b) Transaction time table

Fig. 6. Tables with period timestamps.

Transaction Time Tables. In a transaction time table, each tuple stores an interval that records when the tuple was current in the database. Different from the valid time, the transaction time is set by the DBMS when a tuple is created, updated, or deleted; the user is not allowed to change the transaction time values. The following statement creates a transaction time table for the department relation:

```
CREATE TABLE Dept (
  DName VARCHAR,
  DDesc TEXT,
  DStart DATE GENERATED ALWAYS AS ROW START,
  DEnd DATE GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME (DStart, DEnd)
) WITH SYSTEM VERSIONING;
```

The `GENERATED ALWAYS` clause specifies that the two attributes `DStart` and `DEnd` are generated by the system when a tuple is, respectively, inserted, deleted, or modified. The `PERIOD FOR` clause in combination with the `WITH SYSTEM VERSIONING` clause define a system-versioned table. While the attribute names `DStart` and `DEnd` are user-specified, the name of the transaction time attribute must be `SYSTEM_TIME`.

The following SQL statements insert two tuples in the system-versioned department relation as shown in Fig. 6(b):

```
INSERT INTO DEPT (DName, DDesc) VALUES ('ifi', 'DBTG @ uzh');
INSERT INTO DEPT (DName, DDesc) VALUES ('idse', 'DB @ unibz');
```

[2] To keep the examples simple, we use only the year, not complete dates or timestamps.

The user specifies only the nontemporal attributes, whereas the transaction time is added automatically by the system. The value of `DStart` is set to the current transaction time when the tuple/row is created. Hence, the first tuple was inserted in 2009 and the second in 2010. The value `9999` of the `DEnd` attribute is the highest possible timestamp value and indicates that the tuple is current in the database.

A transaction time table conceptually distinguishes between current tuples and historical tuples. A tuple is considered a *current tuple* if its timestamp contains the current time (aka *now*). All other tuples are called *historical tuples*. Historical tuples are never modified and form *immutable snapshots* of the past.

4.2 Modification of Tables

The SQL:2011 standard specifies the behavior of temporal tables in the case of updates and deletions, which is different for valid time tables and transaction time tables.

Valid Time Tables. Conventional update and delete operations work in the same way as for nontemporal tables. That is, both nontemporal and temporal attributes can be modified using the known SQL syntax. In addition, there is enhanced support for modifying tuples over *parts* of the associated time periods by using the `FOR PORTION OF` clause. In this case, overlapping tuples are automatically split or cut. Consider the following statement:

```
DELETE Emp
FOR PORTION OF EPeriod FROM DATE '2011' TO DATE '2013'
WHERE EName = 'Anton';
```

This statement deletes a portion of the first tuple in the `Emp` relation in Fig. 7(a). As a consequence, the tuple is automatically split in two: one stating that Anton was employed at ifi in 2010, and the other that he was employed from 2013 to 2014, as shown in Fig. 7(b). Non-overlapping tuples are not affected.

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2015
Anton	idse	2015	2017

(a) Before DELETE statement

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2011
Anton	ifi	2013	2015
Anton	idse	2015	2017

(b) After DELETE statement

Fig. 7. Modifying a valid time table.

The behavior of the UPDATE operation is similar. For instance, the following statement would split the first tuple in Fig. 7(a) into three tuples:

```
UPDATE Emp
FOR PORTION OF EPeriod FROM DATE '2011' TO DATE '2013'
SET EDept = 'ai'
WHERE EName = 'Anton';
```

Transaction Time Tables. Any modification of a transaction time table operates only on the current tuples, and the user can only modify nontemporal attributes, not the timestamp attribute. The transaction time is automatically modified when nontemporal attributes of current tuples are modified. That is, if a current tuple is modified, a copy of that tuple is created with the end timestamp set to the current time. The tuple ceases to be current in the database and becomes a historical tuple. Then, the start time of the tuple is updated to the current timestamp, and the nontemporal attributes are changed accordingly. A DELETE statement creates only the historical tuple with end time equal to the current time. The following UPDATE statement changes the description of the IDSE department as of 2016 (cf. Fig. 8):

```
UPDATE Dept
SET DDesc = 'DBS @ unibz'
WHERE DName = 'idse';
```

This creates a historical tuple `(idse, 2010, 2016, DB @ unibz)`, which records the name of the `idse` department until `2015` (the gray tuple in Fig. 8). At the same time, the start time of the current tuple for the `idse` department is set to `2016` and the description is set to `DBS @ unibz`.

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DB @ unibz	2010	9999

(a) Before UPDATE statement

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DBS @ unibz	2016	9999
idse	DB @ unibz	2010	2016

(b) After UPDATE statement

Fig. 8. Modifying a transaction time table.

4.3 Integrity Constraints

Valid Time Tables. Primary keys enforce uniqueness of attribute values in a table. In a valid time table, the natural interpretation of a *primary key* is to require *uniqueness of attribute values at each time point*. To achieve this, the primary key specification includes, in addition to the nontemporal key attributes, also the valid time period together with the `WITHOUT OVERLAPS` constraint.

This ensures that only *one value at a time* exists for nontemporal key attributes (that is, the same values for nontemporal key attributes require *disjoint* periods). For instance, we can use the following primary key constraint to enforce that an employee is never in two different departments at the same time:

```
ALTER TABLE Emp
ADD PRIMARY KEY (EName, EPeriod WITHOUT OVERLAPS);
```

This primary key constraint would reject the table in Fig. 9 since the valid times of both tuples with `EName` equal to Anton include year 2014. Without the WITHOUT OVERLAPS clause, but including `EPeriod`, we would obtain a conventional primary key, which is satisfied by the table in Fig. 9 since the tuples have syntactically different values for these two attributes.

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2015
Anton	idse	2014	2017

Fig. 9. Primary key constraint is violated.

Valid time tables support also foreign keys to enforce the existence of certain tuples. A *foreign key constraint* in a valid time table guarantees that, at each point in time, for each tuple in the child table there exists a corresponding tuple in the referenced parent table. Consider Fig. 10 and assume that both `Emp` and `Dept` are valid time tables with valid time `EPeriod` and `DPeriod`, respectively. The following foreign key constraint achieves that, at any time, the department, in which an employee works, exists:

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2015
Anton	idse	2015	2017

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DBS @ unibz	2016	9999
idse	DB @ unibz	2010	2016

Fig. 10. Valid time foreign keys.

```
ALTER TABLE Emp
ADD FOREIGN KEY (EDept, PERIOD EPeriod)
REFERENCES (DName, PERIOD DPeriod);
```

It is not required that a single matching tuple exists in the referenced parent table that entirely covers the tuple in the child table. It is sufficient that the union of the timestamps of matching tuples in the parent table covers the timestamp

of the corresponding tuple in the child table. The tables in Fig. 10 satisfy the above constraint. The first tuple in the `Emp` table is covered by a single tuple in the `Dept` table, while the second tuple is covered by the union of the second and third tuples in `Dept`.

Transaction Time Tables. The enforcement of primary and foreign key constraints in transaction time tables is much simpler since only current tuples need to be considered. Historical data continue to satisfy the constraints as they are never changed. Therefore, the time periods need not to be included in the definition of the key constraints.

A *primary key* on an attribute in a transaction time table enforces that *at most one current tuple* exists with a given value for that attribute. Note that there might be several historical tuples with the same key attribute value. Consider now that `Emp` and `Dept` are transaction time tables. Then, the following primary key constraint ensures that there exists at most one current tuple with a given `DName` value:

```
ALTER TABLE Dept
ADD PRIMARY KEY (DName);
```

In a similar way, also *foreign keys* need only be verified among the current tuples of the two tables. That is, for each *current tuple* in the child table, there exists a matching *current tuple* in the parent table. For instance, the following constraint enforces that for each current `Emp` tuple there exists *now* a tuple in the `Dept` table with `DName = EDept`:

```
ALTER TABLE Dept
ADD FOREIGN KEY (EDept) REFERENCES (DName);
```

4.4 Querying Temporal Tables

Valid Time Tables. SQL:2011 provides limited support for querying temporal tables, in particular for valid time tables. The usual SQL syntax can be used to specify constraints on the period end points. For instance, the following query retrieves all departments that existed in 2012:

```
SELECT DName, DDesc
FROM Emp
WHERE DStart <= '2012' AND DEnd > '2012';
```

To facilitate the formulation of queries, so-called *period predicates* are introduced, such as `OVERLAPS`, `BEFORE`, `AFTER`, etc. Although similar, they do not correspond exactly to Allen's interval relations [4]. With these predicates, the selection predicate in the above statement can be specified as `WHERE EPeriod CONTAINS DATE '2011'`.

The temporal predicates can also be used in the `FROM` clause. For instance, the `OVERLAPS` predicate allows to formulate a temporal join, which requires that

matching result tuples are temporally overlapping. The following query is a temporal join on the department name of the `Emp` and `Dept` tables:

```
SELECT *
FROM Emp
JOIN Dept ON EDept = DName AND EPeriod OVERLAPS DPeriod;
```

Transaction Time Tables. To facilitate the retrieval of data from transaction time tables, three new SQL extensions are provided. First, the `FOR SYSTEM_TIME AS OF` extension retrieves tuples *as of a given time point*, i.e., tuples with start time less than or equal to and end time larger than a user-specified time point. The following statement retrieves all employee tuples that were current in the database in 2010:

```
SELECT *
FROM Emp FOR SYSTEM_TIME AS OF DATE '2010';
```

The second extension, `FOR SYSTEM_TIME FROM TO`, retrieves tuples *between any two time points*, where the start time is included and the end time is excluded, corresponding to a closed-open interval model. The following statement retrieves all tuples that were current from 2011 (including) up to 2013 (excluding):

```
SELECT *
FROM Dept FOR SYSTEM_TIME FROM DATE '2011' TO DATE '2013';
```

The third extension is `FOR SYSTEM_TIME BETWEEN` and is similar to the previous one, except that the end time point is also included, corresponding to a closed-closed interval model:

```
SELECT *
FROM Dept FOR SYSTEM_TIME BETWEEN DATE '2011' AND DATE '2012';
```

If none of the above extensions are specified in the `FROM` clause, only the *current* tuples are considered. This corresponds to `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP`. This feature facilitates the migration to system-versioned tables, as old queries would continue to produce correct results by considering only current tuples.

5 Temporal Data Support in Commercial DBMSs

Since the introduction of the temporal features in the SQL:2011 standard [58,96], major database vendors have started to implement temporal support in their database management systems [59,72,73]. Some companies realized the need for supporting temporal data earlier, and they extended their database systems with basic temporal features, such as data types, functions, and time travel queries, that make past states of a database available for querying.

IBM was the first vendor to integrate the temporal features from SQL:2011 into their DB2 database system, which occurred in version 10 [79]. DB2 supports both valid time and transaction time tables, which are called business time and system time tables, respectively. Transaction time tables are implemented by means of two distinct tables: a current table and a history table. The current table stores the current snapshot of the data, i.e., all tuples whose timestamp contains the current time (now). The history table stores all previously current data, i.e., all tuples that were modified or deleted in the past. Queries on transaction time tables are automatically rewritten into queries over one or both of these two tables.

The Oracle DBMS supports temporal features from SQL:2011 as of version 12c. The temporal features are implemented using the Oracle flashback technology [69]. The syntax employed differs slightly from that of the SQL standard, e.g., `AS OF PERIOD FOR` is used instead of `FOR` to retrieve data in a certain time period. Earlier versions of Oracle offered similar support for temporal data through the Oracle Workspace Manager [68]. The workspace manager (DBMS_WM package) offered a PERIOD data type with associated predicates and functions as well as additional support for valid and transaction time. Querying temporal relations was possible at a specific time point (snapshot) or for a specific period.

PostgreSQL originally provided an external module [26] that introduced a PERIOD data type for anchored time intervals together with Boolean predicates and functions, such as intersection, union, and minus. Most of the functionality of this module was subsequently integrated into the core of PostgreSQL version 9.2 using range types [77]. Unlike the period specification that is metadata in the SQL standard, a range type in PostgreSQL is a new data type in the query language that was introduced to represent generic intervals, and it comes with associated predicates, functions, and indices. Indices on range types are based on the extendible index infrastructure GiST (Generalized Search Tree) and SP-GiST (space-partitioned Generalized Search Tree). These indices support efficient querying when predicates involve range types as well as support efficient constraint implementation, such as uniqueness for overlapping intervals in a table.

The Teradata DBMS as of version 15.00, supports derived periods and temporal features from the SQL:2011 standard. Version 13.10 already integrated temporal features, including a PERIOD data type with associated predicates and functions as well as support for valid and transaction time [3,89]. The querying of valid and transaction time tables is achieved by means of so-called temporal statement modifiers such as `SEQUENCED` and `NONSEQUENCED` [16]. The implementation of the sequenced semantics is based on query rewriting, where a temporal query is rewritten into a standard SQL query [2,3]. The support for temporal features in Teradata has been enhanced gradually. As of version 14.10 [90], support for sequenced aggregation and coalescing (using the syntax `NORMALIZE ON`) was added. Sequenced outer joins and set operations are not yet supported.

Since 2016, Microsoft's SQL Server [64] has supported temporal features from SQL:2011. The support is limited to transaction time tables, called system time tables. To achieve general temporal support for querying, users have to write user-defined functions [8] that implement the fold and unfold functions of IXSQL.

6 Native Support for Managing Temporal Data in RDBMSs

In this section, we describe a recent approach [30] to extending a relational database engine to achieve full-fledged, industrial-strength, and comprehensive support for sequenced temporal queries. The key idea is to reduce temporal queries to nontemporal queries by first adjusting the timestamps of the input tuples, which produces intermediate relations on which the corresponding nontemporal operators are applied. This solution provides comprehensive support for temporal queries with sequenced semantics without limiting the use of queries with nonsequenced semantics. The approach is systematic and separates interval adjustment from the evaluation of the operators. This strategy renders it possible to fully leverage the query optimization and evaluation engine of a DBMS for sequenced temporal query processing.

6.1 Requirements for Sequenced Temporal Queries

The evaluation of sequenced temporal queries over a temporal database has to satisfy four properties: *snapshot reducibility*, *change preservation*, *extended snapshot reducibility*, and *scaling* (cf. Sect. 2.3). Two important ingredients are needed for the query execution in order to achieve these properties:

- timestamps must be *adjusted* for the result, and
- some values might have to be *scaled* to the adjusted timestamps.

This is illustrated in the example in Fig. 11, which computes the budget for each department in the `Dept` relation. In a nontemporal context, the result would be 380 K for the DB department and 150 K for the AI department. In a temporal context, we want to obtain the time-varying budget shown in Fig. 11(b). We observe that there are two result tuples for the `DB` department. Result tuple z_1 is over the time period [Feb, Apr], where only one project is running. Result tuple z_2 is over the time period [May, Jul] with two contributing input tuples, namely r_1 and r_2. A second observation is that the total budget of 200 K of the input tuple r_1 is distributed over (or *scaled* to) the two sub-periods [Feb, Apr] and [May, Sep], i.e., 100 K is assigned to each of the two periods.

A major limitation of SQL that renders it difficult to process interval timestamped data, such as in the above example, is that periods are considered as atomic units. Comparing interval timestamped tuples in SQL yields the following results:

Dept

	Name	Dept	Budget	Time
r_1	Sue	DB	200K	[Feb, Jul]
r_2	Tim	DB	180K	[May, Jul]
r_3	Joe	AI	150K	[Apr, Aug]

(a) Department relation

Result

	Dept	SUM	Time
z_1	DB	100K	[Feb, Apr]
z_2	DB	280K	[May, Jul]
z_3	AI	150K	[Apr, Aug]

(b) Budget per department

Fig. 11. Compute budget for each department.

- $(\text{DB}, [\text{May}, \text{Jul}]) = (\text{DB}, [\text{May}, \text{Jul}]) \rightarrow \textit{true}$
- $(\text{DB}, [\text{Feb}, \text{Apr}]) = (\text{DB}, [\text{May}, \text{Jul}]) \rightarrow \textit{false}$
- $(\text{DB}, [\text{Feb}, \text{Jul}]) = (\text{DB}, [\text{May}, \text{Jul}]) \rightarrow \textit{false}$

The first two comparisons are ok, since the two tuples are, respectively, identical in the first case and have disjoint timestamps (and hence are syntactically different) in the second case. The result of the last comparison is problematic in a temporal context. Since the two timestamps are overlapping, the two tuples are equal over the common part of the timestamps.

6.2 Reducing Temporal Operators to Nontemporal Operators via Temporal Alignment

Based on the above requirements and observations, the core of the temporal alignment approach [30] is to adjust the timestamps of input tuples such that all tuples that contribute to a single result tuple have identical timestamps. The adjusted timestamps can then be treated as atomic units, and the corresponding nontemporal operator with SQL's notion of equality produces the expected result. Additionally, for some queries, the original timestamp needs to be preserved, and the attribute values need to be scaled. This reduction of a temporal operator ψ^T to the corresponding nontemporal operator ψ is a four-step process (cf. Fig. 12):

1. *Timestamp propagation* replicates the original timestamps in the argument relations as additional attributes. This step is optional and is only executed if the original timestamps are needed, either to scale attribute values in step 3 or to evaluate a predicate or a function that references the original timestamps, in step 4.
2. *Interval adjustment* splits the overlapping timestamps of the input tuples such that they are aligned. This yields an intermediate relation, where all tuples that (in step 4) are processed together to produce a result tuple have the same timestamp. This intermediate relation can conceptually be considered as a sequence of snapshots, each of which lasts for one or more time points. Two interval adjustment primitives are needed: a *temporal normalizer*, $\mathcal{N}$, for the operators π, ϑ, $-$, $\cap$, and $\cup$, where for each time point, one input tuple can contribute to at most one result tuple; and a *temporal aligner*, ϕ, for the operators $\times$, $\bowtie$, $\rtimes$, $\ltimes$, $\bowtie$, and $\triangleright$, where for each time point, one input tuple can contribute to more than one result tuple.

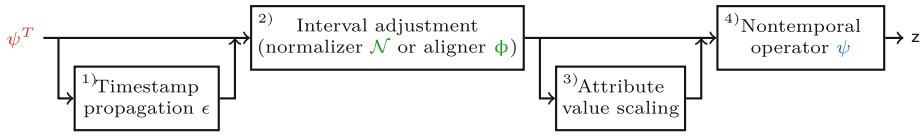


Fig. 12. Reduction of a temporal operator ψ^T to the corresponding nontemporal operator ψ using interval adjustment, timestamp propagation, and attribute value scaling.

3. *Attribute value scaling* is optional and scales, if required, the attribute values of intermediate tuples to the adjusted timestamps. For this, the original and new timestamps in addition to the original value of the attribute to be scaled are needed. As part of this step, the propagated timestamps are removed if they are no longer needed by subsequent operators.
4. The *nontemporal operator evaluation* applies the corresponding nontemporal operator ψ to the intermediate relations. An additional equality constraint over the adjusted timestamps (e.g., as a grouping attribute for aggregation or an equality predicate in joins) guarantees that all tuples that produce a single result tuple are processed together. For join operations, a post-processing step α is required to remove non-maximal duplicates.

The interval adjustment (step 2) and the evaluation of the corresponding nontemporal operator (step 4) form the core of the temporal alignment approach and guarantee snapshot reducibility and change preservation. In addition, the propagation of the timestamp intervals (step 1) enables attribute value scaling (step 3) and the access to the original timestamp in step 4 (needed for extended snapshot reducibility).

Table 1 provides a summary of the reduction rules, following the above strategy, for all operators of the relational algebra. Here, r and s are temporal relations, $\mathbf{A}$ and $\mathbf{B}$ are sets of attributes, T is the timestamp attribute, θ is a condition, and α is a post-processing operator that removes duplicates. For instance, the temporal aggregation operator ${}_{\mathbf{B}}\vartheta^T_F(\mathsf{r}) = {}_{\mathbf{B},T}\ \vartheta_F(\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r}))$ with grouping attributes $\mathbf{B}$ can be computed as follows: First, input relation r is aligned by calling the temporal normalizer $\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r})$, which yields an intermediate relation of aligned tuples. Then, nontemporal aggregation is applied to the intermediate result. By adding the timestamp attribute T as an (additional) grouping attribute, the intermediate tuples are grouped by snapshot, and the nontemporal aggregation operator is applied to each snapshot.

Figure 13 illustrates the temporal alignment approach by using a temporal aggregation query to compute the number of projects for each department: ${}_{Dept}\vartheta^T_{Count(Proj)}(\mathsf{Proj})$. Since timestamp propagation and attribute value scaling are not needed for this query, steps 1 and 3 are skipped. During the adjustment of the timestamps using the temporal normalizer (step 2), the first input tuple r_1 is split into tuples r'_1 and r''_1. The split point is determined by tuple r_2, which belongs to the same department. There is no need to split tuples r_2 and r_3, yielding an intermediate relation with four tuples. Then, the intermediate

Table 1. Reduction rules $\psi^T \longrightarrow \{\mathcal{N},\phi\} + \psi$ (from [27,30])

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

relation is aggregated (step 4) by using the nontemporal aggregation, where the timestamp attribute T is added to the grouping attributes. Hence, all tuples with the same adjusted timestamp and the same department are processed together.

Figure 14 illustrates a temporal left outer join using the temporal aligner primitive. Given a manager relation Mgr and a project relation Proj, we want

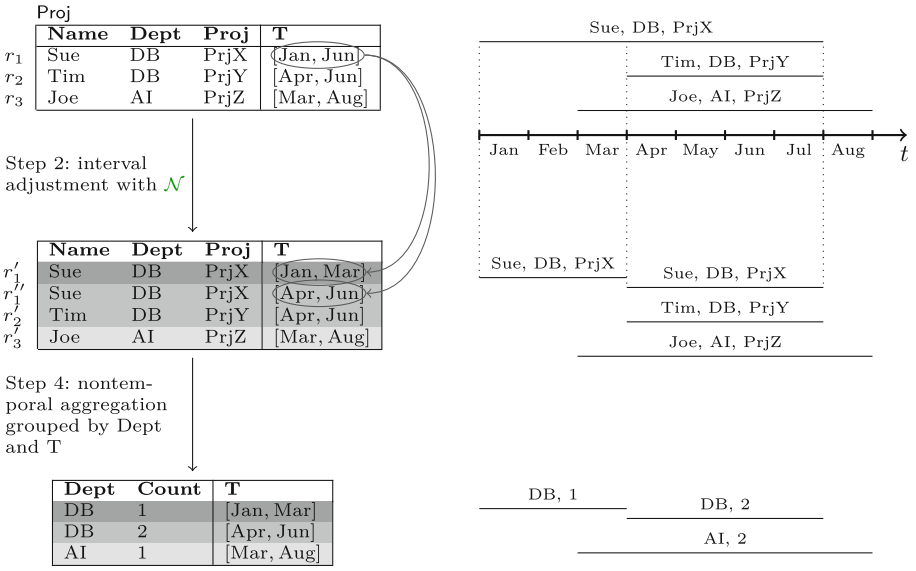


Fig. 13. Illustration of temporal normalizer for a temporal aggregation query.

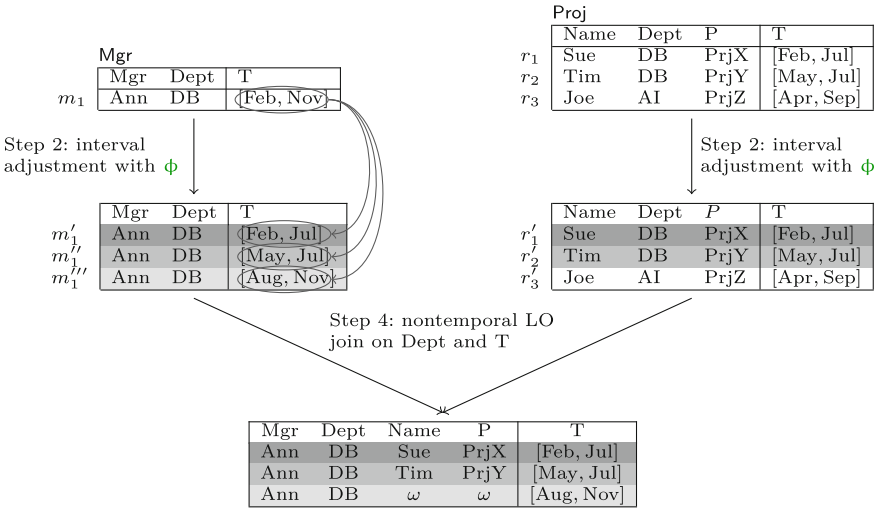


Fig. 14. Illustration of temporal aligner for a temporal left-outer join query.

to determine a manager's budget: $\mathsf{Mgr} \overset{T}{\bowtie}_{\mathsf{Mgr}.Dept=\mathsf{Proj}.Dept} \mathsf{Proj}$. Again, to keep the example simple, timestamp propagation and attribute value scaling are not involved. Hence, we first align the timestamps of the two input relations. In the manager relation, the only tuple m_1 is split into three intermediate tuples. The first two, m_1' and m_1'', are generated from the intersection of m_1's timestamp and the timestamp of the joining tuples r_1 and r_2 of Proj, respectively. The third intermediate tuple, m_1''', covers the part of m_1's timestamp not covered by any matching tuple in the project relation; this tuple is needed for the outer join. Similarly, the tuples in the project relation are adjusted. The first two tuples are completely covered by matching tuple m_1, so no split is required. The third tuple need not to be split since it has no matching tuple in Mgr. After adjusting the timestamps of the two input relations, the two intermediate tables are joined by using the nontemporal left outer join, where the timestamp attribute T is added to the join condition in order to join only tuples that have identical timestamps.

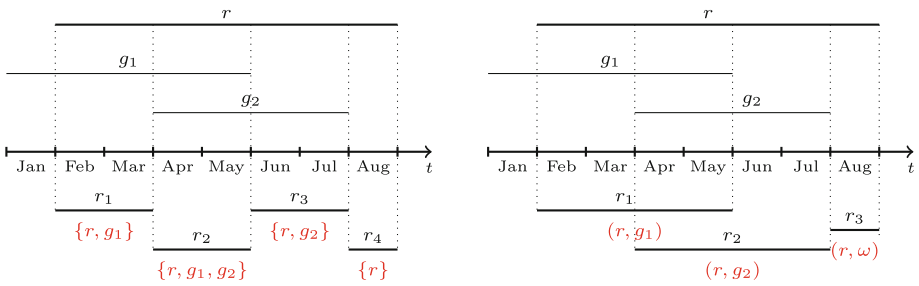
6.3 Temporal Primitives

The temporal alignment approach requires two new temporal primitives, a temporal normalizer and a temporal aligner, to break the timestamps of the input tuples into aligned pieces.

The *temporal normalizer* is used for operators, $\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)$, for which more than one tuple of each argument relation r_i can contribute to a result tuple z (i.e., the lineage set of z can contain more than one tuple from each input relation). This holds for the following operators: aggregation (ϑ), projection (π), difference ($-$), intersection ($\cap$), and union ($\cup$). The temporal normalizer splits each input tuple into temporally *disjoint* pieces, where groups of matching tuples define the

split points. This is illustrated in Fig. 15(a) for an input tuple r and two other input tuples g_1 and g_2 in the same group. Tuple r is split whenever another tuple in the same group starts or finishes, producing r_1, r_2 and r_3. Moreover, all parts of r that are not covered by another tuple in the group are reported, i.e., r_4. Notice that the intermediate tuples are disjoint.

The *temporal aligner* is used for operators, $\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)$, for which at most one input tuple from each argument relation r_i can contribute to a result tuple z (i.e., the lineage set of z contains at most one tuple from each input relation). This holds for the following operators: Cartesian product ($\times$) and all forms of joins ($\bowtie$, $\rtimes$, $\ltimes$, $\rtimes\!\!\ltimes$, $\triangleright$). The temporal aligner considers pairs of matching tuples and determines the *intersections* of their timestamps; the resulting intermediate relation might contain temporally overlapping tuples. Figure 15(b) illustrates the temporal aligner for an input tuple r and two other matching input tuples g_1 and g_2. Tuple r produces three intermediate tuples: one as the intersection with tuple g_1, one as the intersection with tuple g_2, and one for the part of the timestamp that is not covered by any matching tuple (r_3).



(a) Temporal normalizer (b) Temporal aligner

Fig. 15. Temporal normalizer vs. aligner (from [27,30]).

6.4 Implementation

The temporal alignment approach to transform temporal queries to the corresponding nontemporal queries with the help of two adjustment primitives requires minimal extensions of an existing DBMS. Moreover, this strategy renders it possible to fully leverage the query optimization and evaluation engine of a DBMS for sequenced temporal query processing, and it does not affect the use of nonsequenced queries. The key extension is the integration of the normalizer $\mathcal{N}$ and aligner ϕ operators into the DBMS kernel. Timestamp propagation (ϵ) and attribute value scaling can be achieved, respectively, by means of generalized projections and user defined functions.

The temporal alignment approach has been implemented in the kernel of the PostgreSQL database system and is available at tpg.inf.unibz.it [27,30].

7 Conclusion

In this tutorial, we provided an overview of temporal data management, covering both research results and commercial database management systems. Following a brief summary of important concepts that have been developed and used in temporal database research, we discussed the state-of-the-art in temporal database research, focusing on query languages and evaluation algorithms. We then described the most important temporal features in SQL:2011, which is the first SQL standard to introduce temporal support in SQL. Next, we briefly discussed the degree to which temporal features of the SQL:2011 standard have been adopted by commercial database management systems. The tutorial ends with a description of a recent framework that provides a comprehensive and native solution to the processing of so-called sequenced temporal queries in relational database management systems.

Future work in temporal databases points in various directions. While temporal alignment provides a solid and systematic framework for implementing temporal query support in relational database systems, a number of open issues require further investigation. First, it would be interesting to extend the framework to multisets that allow duplicates, as well as to support two or more time dimensions. Second, for some operators, a significant boost in efficiency is needed to scale for very large datasets. Ideas for performance optimizations range from additional and more targeted alignment primitives over more precise cost estimates to specialized query algorithms and equivalence rules. Third, support for user-friendly formulation of complex temporal queries is needed, including a SQL-based temporal query language.

References

1. Agesen, M., Böhlen, M.H., Poulsen, L., Torp, K.: A split operator for now-relative bitemporal databases. In: Proceedings of the 17th International Conference on Data Engineering, ICDE 2001, pp. 41–50 (2001)
2. Al-Kateb, M., Ghazal, A., Crolotte, A.: An efficient SQL rewrite approach for temporal coalescing in the teradata RDBMS. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012. LNCS, vol. 7447, pp. 375–383. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32597-7_32
3. Al-Kateb, M., Ghazal, A., Crolotte, A., Bhashyam, R., Chimanchode, J., Pakala, S.P.: Temporal query processing in teradata. In: Proceedings of the 16th International Conference on Extending Database Technology, EDBT 2013, pp. 573–578 (2013)
4. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
5. Arbesman, S.: Stop hyping big data and start paying attention to 'long data'. Wired.com (2013). https://www.wired.com/2013/01/forget-big-data-think-long-data/
6. Bair, J., Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Notions of upward compatibility of temporal query languages. Wirtschaftsinformatik **39**(1), 25–34 (1997)

7. Behrend, A., et al.: Temporal state management for supporting the real-time analysis of clinical data. In: Bassiliades, N., et al. (eds.) New Trends in Database and Information Systems II. AISC, vol. 312, pp. 159–170. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-10518-5_13
8. Ben-Gan, I., Sarka, D., Wolter, R., Low, G., Katibah, E., Kunen, I.: Inside Microsoft SQL Server 2008 T-SQL programming, Chap. 12. In: Temporal Support in the Relational Model. Microsoft Press (2008)
9. Bettini, C., Jajodia, S., Wang, S.: Time Granularities in Databases, Data Mining, and Temporal Reasoning. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04228-1
10. Bettini, C., Sean Wang, X., Jajodia, S.: Temporal granularity. In: Liu and Özsu [60], pp. 2968–2973
11. Böhlen, M.H., Gamper, J., Jensen, C.S.: An algebraic framework for temporal attribute characteristics. Ann. Math. Artif. Intell. **46**(3), 349–374 (2006)
12. Böhlen, M.H., Gamper, J., Jensen, C.S.: How would you like to aggregate your temporal data? In: Proceedings of the 13th International Symposium on Temporal Representation and Reasoning, TIME 2006, pp. 121–136 (2006)
13. Böhlen, M., Gamper, J., Jensen, C.S.: Multi-dimensional aggregation for temporal data. In: Ioannidis, Y., et al. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 257–275. Springer, Heidelberg (2006). https://doi.org/10.1007/11687238_18
14. Böhlen, M.H., Jensen, C.S.: Temporal data model and query language concepts. In: Encyclopedia of Information Systems, pp. 437–453. Elsevier (2003)
15. Böhlen, M.H., Jensen, C.S.: Sequenced semantics. In: Liu and Özsu [60], pp. 2619–2621
16. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Temporal statement modifiers. ACM Trans. Database Syst. **25**(4), 407–456 (2000)
17. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Current semantics. In: Liu and Özsu [60], pp. 544–545
18. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Nonsequenced semantics. In: Liu and Özsu [60], pp. 1913–1915
19. Böhlen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in temporal databases. In: Proceedings of 22th International Conference on Very Large Data Bases, VLDB 1996, pp. 180–191 (1996)
20. Cohen Boulakia, S., Tan, W.C.: Provenance in scientific databases. In: Liu and Özsu [60], pp. 2202–2207
21. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. PVLDB **10**(11), 1346–1357 (2017)
22. Cafagna, F., Böhlen, M.H.: Disjoint interval partitioning. VLDB J. **26**(3), 447–466 (2017)
23. Chomicki, J., Toman, D., Böhlen, M.H.: Querying ATSQL databases with temporal logic. ACM Trans. Database Syst. **26**(2), 145–178 (2001)
24. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. **25**(2), 179–227 (2000)
25. Date, C.J., Darwen, H., Lorentzos, N.A.: Temporal Data and the Relational Model. Elsevier (2002)
26. Davis, J.: Online temporal PostgreSQL reference (2009). http://temporal.projects.postgresql.org/reference.html
27. Dignös, A., Böhlen, M.H., Gamper, J.: Temporal alignment. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, pp. 433–444 (2012)

28. Dignös, A., Böhlen, M.H., Gamper, J.: Query time scaling of attribute values in interval timestamped databases. In: Proceedings of the 29th International Conference on Data Engineering, ICDE 2013, pp. 1304–1307 (2013)
29. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2014, pp. 1459–1470 (2014)
30. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S.: 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)
31. Dyreson, C.E.: Chronon. In: Liu and Özsu [60], p. 329
32. Dyreson, C.E., Jensen, C.S., Snodgrass, R.T.: Now in temporal databases. In: Liu and Özsu [60], pp. 1920–1924
33. Dyreson, C.E., Lin, H., Wang, Y.: Managing versions of web documents in a transaction-time web server. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 422–432 (2004)
34. Dyreson, C.E., Rani, V.A.: Translating temporal SQL to nested SQL. In: Proceedings of the 23rd International Symposium on Temporal Representation and Reasoning, TIME 2016, pp. 157–166 (2016)
35. Dyreson, C.E., Rani, V.A., Shatnawi, A.: Unifying sequenced and non-sequenced semantics. In: Proceedings of the 22nd International Symposium on Temporal Representation and Reasoning, TIME 2015, pp. 38–46 (2015)
36. Jensen, C.S., Clifford, J., Gadia, S.K., Grandi, F., Kalua, P.P., Kline, N., Lorentzos, N., Mitsopoulos, Y., Montanari, A., Nair, S.S., Peressi, E., Pernici, B., Robertson, E.L., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Tansel, A., Tiberio, P., Tuzhilin, A., Wuu, G.T.J.: A consensus test suite of temporal database queries. Technical report R 93–2034, Aalborg University, Department of Mathematics and Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, November 1993
37. Etzion, O., Jajodia, S., Sripada, S. (eds.): Temporal Databases: Research and Practice. LNCS, vol. 1399. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053695
38. Gadia, S.K.: A homogeneous relational model and query languages for temporal databases. ACM Trans. Database Syst. **13**(4), 418–448 (1988)
39. Gadia, S.K., Yeung, C.-S.: A generalized model for a relational temporal database. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD 1988, pp. 251–259 (1988)
40. Galton, A.: A critical examination of Allen's theory of action and time. Artif. Intell. **42**(2–3), 159–188 (1990)
41. Gamper, J., Böhlen, M.H., Jensen, C.S.: Temporal aggregation. In: Liu and Özsu [60], pp. 2924–2929
42. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), 2–29 (2005)
43. Gao, D., Snodgrass, R.T.: Temporal slicing in the evaluation of XML queries. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, pp. 632–643 (2003)
44. Grandi, F.: Temporal databases. In: Encyclopedia of Information Science and Technology, 3rd edn., pp. 1914–1922. IGI Global (2015)
45. Grandi, F., Mandreoli, F., Martoglia, R., Penzo, W.: A relational algebra for streaming tables living in a temporal database world. In: Proceedings of the 24th International Symposium on Temporal Representation and Reasoning, TIME 2017, pp. 15:1–15:17 (2017)

46. Grandi, F., Mandreoli, F., Tiberio, P.: Temporal modelling and management of normative documents in XML format. Data Knowl. Eng. **54**(3), 327–354 (2005)
47. Jensen, C.S., Dyreson, C.E., Böhlen, M.H., Clifford, J., Elmasri, R., Gadia, S.K., Grandi, F., Hayes, P.J., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N.A., Mitsopoulos, Y.G., Montanari, A., Nonen, D.A., Peressi, E., Pernici, B., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Soo, M.D., Uz Tansel, A., Tiberio, P., Wiederhold, G.: The consensus glossary of temporal database concepts. In Temporal Databases, Dagstuhl, pp. 367–405 (1997)
48. Jensen, C.S., Snodgrass, R.T.: Snapshot equivalence. In: Liu and Özsu [60], p. 2659
49. Jensen, C.S., Snodgrass, R.T.: Temporal data models. In: Liu and Özsu [60], pp. 2952–2957
50. Jensen, C.S., Snodgrass, R.T.: Temporal element. In: Liu and Özsu [60], p. 2966
51. Jensen, C.S., Snodgrass, R.T.: Time instant. In: Liu and Özsu [60], p. 3112
52. Jensen, C.S., Snodgrass, R.T.: Timeslice operator. In: Liu and Özsu [60], pp. 3120–3121
53. Jensen, C.S., Snodgrass, R.T.: Transaction time. In: Liu and Özsu [60], pp. 3162–3163
54. Jensen, C.S., Snodgrass, R.T.: Valid time. In: Liu and Özsu [60], pp. 3253–3254
55. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1173–1184 (2013)
56. Kaufmann, M., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F.: Comprehensive and interactive temporal query processing with SAP HANA. PVLDB **6**(12), 1210–1213 (2013)
57. Kline, N., Snodgrass, R.T.: Computing temporal aggregates. In: Proceedings of the 11th International Conference on Data Engineering, ICDE 1995, pp. 222–231 (1995)
58. Kulkarni, K.G., Michels, J.-E.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
59. Künzner, F., Petković, D.: A comparison of different forms of temporal data management. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015. CCIS, vol. 521, pp. 92–106. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18422-7_8
60. Liu, L., Tamer Özsu, M. (eds.): Encyclopedia of Database Systems. Springer, Boston (2009)
61. López, I.F.V., Snodgrass, R.T., Moon, B.: Spatiotemporal aggregate computation: a survey. IEEE Trans. Knowl. Data Eng. **17**(2), 271–286 (2005)
62. Lorentzos, N.A.: Time period. In: Liu and Özsu [60], p. 3113
63. Lorentzos, N.A., Mitsopoulos, Y.G.: SQL extension for interval data. IEEE Trans. Knowl. Data Eng. **9**(3), 480–499 (1997)
64. Microsoft. SQL Server 2016 - temporal tables (2016). https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables
65. Moffitt, V.Z., Stoyanovich, J.: Towards sequenced semantics for evolving graphs. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, pp. 446–449 (2017)
66. Montanari, A., Chomicki, J.: Time domain. In: Liu and Özsu [60], pp. 3103–3107
67. Moon, B., López, I.F.V., Immanuel, V.: Efficient algorithms for large-scale temporal aggregation. IEEE Trans. Knowl. Data Eng. **15**(3), 744–759 (2003)
68. Murray, C.: Oracle database workspace manager developer's guide (2008). http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf

69. Oracle. Database development guide - temporal validity support (2016). https://docs.oracle.com/database/121/ADFNS/adfns_design.htm#ADFNS967
70. Papaioannou, K., Böhlen, M.H.: TemProRA: top-k temporal-probabilistic results analysis. In: Proceedings of the 32nd IEEE International Conference on Data Engineering, ICDE 2016, pp. 1382–1385 (2016)
71. Persia, F., Bettini, F., Helmer, S.: An interactive framework for video surveillance event detection and modeling. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, pp. 2515–2518 (2017)
72. Petković, D.: Modern temporal data models: strengths and weaknesses. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015. CCIS, vol. 521, pp. 136–146. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18422-7_12
73. Petkovic, Dušan: Temporal data in relational database systems: a comparison. In: Rocha, Á., Correia, A.M., Adeli, H., Teixeira, M.M., Reis, L.P. (eds.) New Advances in Information Systems and Technologies. AISC, vol. 444, pp. 13–23. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31232-3_2
74. Piatov, D., Helmer, S.: Sweeping-based temporal aggregation. In: Gertz, M., et al. (eds.) SSTD 2017. LNCS, vol. 10411, pp. 125–144. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64367-0_7
75. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: Proceedings of the 32nd International Conference on Data Engineering, ICDE 2016, pp. 1098–1109 (2016)
76. Pitoura, E.: Historical graphs: models, storage, processing. In: Zimányi, E. (ed.) eBISS 2017. LNBIP, vol. 324, pp. 84–111. Springer, Cham (2017)
77. PostgreSQL Global Development Group. Documentation manual PostgreSQL - range types (2012). http://www.postgresql.org/docs/9.2/static/rangetypes.html
78. Rolland, C., Bodart, F., Léonard, M. (eds.) Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems (1988)
79. Saracco, C., Nicola, M., Gandhi, L.: A matter of time: Temporal data management in DB2 10 (2012). http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf
80. Snodgrass, R.T. (ed.): Proceedings of the International Workshop on an Infrastructure for Temporal Databases (1993)
81. Snodgrass, R.T. (ed.): The TSQL2 Temporal Query Language. Kluwer (1995)
82. Snodgrass, R.T. (ed.): A Case Study of Temporal Data. Teradata Corporation (2010)
83. Snodgrass, R.T., Böhlen, M.H., Jensen, C.S., Steiner, A.: Adding valid time to SQL/temporal. Technical report ANSI-96-501r2, October 1996
84. Snodgrass, R.T., Böhlen, M.H., Jensen, C.S., Steiner, A.: Transitioning temporal support in TSQL2 to SQL3. In: Temporal Databases, Dagstuhl, pp. 150–194 (1997)
85. Son, D., Elmasri, R.: Efficient temporal join processing using time index. In: Proceedings of the 8th International Conference on Scientific and Statistical Database Management, SSDBM 1996, pp. 252–261 (1996)
86. Soo, M.D., Jensen, C.S., Snodgrass, R.T.: An algebra for TSQL2. In: The TSQL2 Temporal Query Language, Chap. 27, pp. 501–544. Kluwer (1995)
87. Uz Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., Snodgrass, R.T. (eds.): Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings (1993)
88. Tao, Y., Papadias, D., Faloutsos, C.: Approximate temporal aggregation. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, pp. 190–201 (2004)

89. Teradata. Teradata database 13.10 - temporal table support (2010). http://www.info.teradata.com/download.cfm?ItemID=1005295
90. Teradata. Teradata database 14.10 - temporal table support (2014). http://www.info.teradata.com/eDownload.cfm?itemid=131540028
91. Terenziani, P., Snodgrass, R.T.: Reconciling point-based and interval-based semantics in temporal relational databases: a treatment of the telic/atelic distinction. IEEE Trans. Knowl. Data Eng. **16**(5), 540–551 (2004)
92. Toman, D.: Point vs. interval-based query languages for temporal databases. In: Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996, pp. 58–67 (1996)
93. Toman, D.: Point-based temporal extensions of SQL and their efficient implementation. In: Etzion, O., Jajodia, S., Sripada, S. (eds.) Temporal Databases: Research and Practice. LNCS, vol. 1399, pp. 211–237. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053704
94. Tuma, P.A.: Implementing Historical Aggregates in TempIS. Ph.D. thesis, Wayne State University (1992)
95. Yang, J., Widom, J.: Incremental computation and maintenance of temporal aggregates. VLDB J. **12**(3), 262–283 (2003)
96. Zemke, F.: Whats new in SQL: 2011. SIGMOD Rec. **41**(1), 67–73 (2012)
97. Zhang, D., Markowetz, A., Tsotras, V.J., Gunopulos, D., Seeger, B.: Efficient computation of temporal aggregates with range predicates. In: Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2001 (2001)
98. Zhang, D., Tsotras, V.J., Seeger, B.: Efficient temporal join processing using indices. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, pp. 103–113 (2002)
99. Zhou, X., Wang, F., Zaniolo, C.: Efficient temporal coalescing query support in relational database systems. In: Bressan, S., Küng, J., Wagner, R. (eds.) DEXA 2006. LNCS, vol. 4080, pp. 676–686. Springer, Heidelberg (2006). https://doi.org/10.1007/11827405_66