# Temporal Database Entries for the Springer Encyclopedia of Database Systems

Christian S. Jensen and Richard T. Snodgrass (Editors)

May 22, 2008

TR-90

A TIMECENTER Technical Report

| | |
|---|---|
| Title | Temporal Database Entries for the<br>Springer Encyclopedia of Database Systems |
| |  |
| Author(s) | Christian S. Jensen and Richard T. Snodgrass (Editors) |
| Publication History | May 2008, A TIMECENTER Technical Report |

## TIMECENTER Participants

**Aalborg University, Denmark**
Christian S. Jensen (codirector), Simonas Šaltenis, Kristian Torp

**University of Arizona, USA**
Richard T. Snodgrass (codirector), Sudha Ram

**Individual participants**
Yun Ae Ahn, Chungbuk National University, Korea; Michael H. Böhlen, Free University of Bolzano, Italy; Curtis E. Dyreson, Utah State University, USA; Dengfeng Gao, IBM Silicon Valley Lab, USA; Fabio Grandi, University of Bologna, Italy; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Paolo Terenziani, University of Piemonte Orientale "Amedeo Avogadro," Alessandria, Italy; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Siemens, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>



The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

# Preface

In January 2007 Ling Liu and Tamer Özsu started work on an Encyclopedia of Database Systems to be published by Springer. We were asked to edit the encyclopedia entries that relate to the area of temporal databases.

This report collects versions of the temporal database entries as of May, 2008. These entries are preliminary in several respects. First, the entries have not been subjected to Springer's final copyediting. Second, they are only approximately formatted: they will look much better in their final form. Third, in contrast to the entries in their final form, the entries in the technical report cannot be searched electronically (see below). And fourth, the reader does not get the benefit of the other entries available in the full encyclopedia. Nonetheless, the *content* appearing here is close to that which will appear in the final encyclopedia, and the entries included here provide a succinct and broad overview of the contributions and structure of the area of temporal databases.

The complete encyclopedia, in which the final entries will appear, will be in multiple volumes. It will constitute a comprehensive and authoritative reference on databases, data management, and database systems. Since it will be available in both print and online formats, researchers, students, and practitioners will benefit from advanced search functionality and convenient interlinking possibilities with related online content. The Encyclopedia's online version will be accessible on the SpringerLink platform (`http://www.springer.com/computer/database+management+%26+information+retrieval/book/978-0-387-49616-0`).

We thank the more than two dozen authors who contributed to these entries; some contributed to multiple entries. We list those authors here.

| | | |
|---|---|---|
| Claudio Bettini | Christian S. Jensen | Arie Shoshani |
| Michael H. Böhlen | James B. D. Joshi | Richard T. Snodgrass |
| Jan Chomicki | Vijay Khatri | V. S. Subrahmanian |
| Carlo Combi | David Lomet | Abdullah Uz Tansel |
| Curtis E. Dyreson | Nikos A. Lorentzos | Paolo Terenziani |
| Per F. V. Hasle | Nikos Mamoulis | David Toman |
| Johann Gamper | Angelo Montanari | Kristian Torp |
| Dengfeng Gao | Mirella M. Moro | Vassilis J. Tsotras |
| Like Gao | Peter Øhrstrøm | X. Sean Wang |
| Fabio Grandi | Peter Revesz | Jef Wijsen |
| Sushil Jajodia | John F. Roddick | Yue Zhang |

All entries were reviewed by several experts, underwent one or several revisions, and were eventually accepted by an Associate Editor of the encyclopedia. The authors represent in concert over three *centuries* of innovative research in temporal databases, experience that informs the content of these entries.

We thank Springer for providing useful online tools for managing the logistics of this large project and for investing heavily to ensure a highly useful and authoritative resource for the database community and for others interested in this technology. Finally, we thank Jennifer Carlson and Simone Tavenrath, who so effectively and cheerfully managed the process at Springer; Jennifer Evans, also at Springer, who consistently supported this effort; and Ling and Tamer, for heading up this effort and insisting on the highest quality from the very beginning.

Richard Snodgrass and Christian S. Jensen
May 2008

# Contents

# ABSOLUTE TIME

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

none

## DEFINITION

A temporal database contains time-referenced, or timestamped, facts. A time reference in such a database is *absolute* if its value is independent of the context, including the current time, *now*.

## MAIN TEXT

An example is "Mary's salary was raised on March 30, 2007." The fact here is that Mary's salary was raised. The absolute time reference is March 30, 2007, which is a time instant at the granularity of day.
Another example is "Mary's monthly salary was $ 15,000 from January 1, 2006 to November 30, 2007." In this example, the absolute time reference is the time period $[\text{January}1, 2006 - \text{November}30, 2007]$.
Absolute time can be contrasted with *relative time*.

## CROSS REFERENCE*

Now in Temporal Databases, Relative Time, Temporal Database, Temporal Granularity

## REFERENCES*

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.
C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# Abstract Versus Concrete Temporal Query Languages

Jan Chomicki, University at Buffalo, USA, http://www.cse.buffalo.edu/~chomicki
David Toman, University of Waterloo, Canada, http://www.cs.uwaterloo.ca/~david

## SYNONYMS

historical query languages

## DEFINITION

*Temporal query languages* are a family of query languages designed to query (and access in general) time-dependent information stored in temporal databases. The languages are commonly defined as extensions of standard query languages for non-temporal databases with *temporal features*. The additional features reflect the way dependencies of data on time are captured by and represented in the underlying temporal data model.

## HISTORICAL BACKGROUND

Most databases store time-varying information. On the other hand, SQL is often the language of choice for developing applications that utilize the information in these databases. Plain SQL, however, does not seem to provide adequate support for temporal applications.

**Example**. To represent the *employment histories* of persons, a common relational design would use a schema

$$\texttt{Employment}(\texttt{FromDate}, \texttt{ToDate}, \texttt{EID}, \texttt{Company}),$$

with the intended meaning that a person identified by `EID` worked for `Company` continuously from `FromDate` to `ToDate`. Note that while the above schema is a standard relational schema, the additional assumption that the values of the attributes `FromDate` and `ToDate` represent *continuous periods* of time is itself *not* a part of the relational model.

Formulating even simple queries over such a schema is non-trivial: for example the query GAPS: *"List all persons with gaps in their employment history, together with the gaps"* leads to a rather complex formulation in, e.g., SQL over the above schema (this is left as a challenge to readers who consider themselves SQL experts; for a list of appealing, but incorrect solutions, including the reasons why, see [9]). The difficulty arises because a single tuple in the relation is conceptually a *compact representation of a set of tuples*, each tuple stating that an employment fact was true on a particular day.

The tension between the conceptual abstract temporal data model (in the example, the property that employment facts are associated with individual *time instants*) and the need for an efficient and compact representation of temporal data (in the example, the representation of continuous periods by their start and end instants) has been reflected in the development of numerous temporal data models and temporal query languages [3].

## SCIENTIFIC FUNDAMENTALS

Temporal query languages are commonly defined using *temporal extensions* of existing non-temporal query languages, such as relational calculus, relational algebra, or SQL. The temporal extensions can be categorized in two, mostly orthogonal, ways:

*The choice of the actual temporal values manipulated by the language.* This choice is primarily determined by the underlying temporal data model. The model also determines the associated operations on these values. The meaning of temporal queries is then defined in terms of temporal values and operations on them, and their interactions with *data* (non-temporal) values in a temporal database.

*The choice of syntactic constructs to manipulate temporal values in the language.* This distinction determines whether the temporal values in the language are accessed and manipulated *explicitly*, in a way similar to other values stored in the database, or whether the access is *implicit*, based primarily on *temporally extending* the meaning of constructs that already exist in the underlying non-temporal language (while still using the operations defined by the temporal data model).

Additional design considerations relate to *compatibility* with existing query languages, e.g., the notion of temporal upward compatibility.

However, as illustrated above, an additional hurdle stems from the fact that many (early) temporal query languages allowed the users to manipulate a *finite underlying representation* of temporal databases rather than the actual temporal values/objects in the associated temporal data model. A typical example of this situation would be an approach in which the temporal data model is based on time instants, while the query language introduces interval-valued attributes. Such a discrepancy often leads to a complex and unintuitive semantics of queries.

In order to clarify this issue, Chomicki has introduced the notions of *abstract* and *concrete* temporal databases and query languages [2]. Intuitively, *abstract temporal query languages* are defined at the conceptual level of the temporal data model, while their *concrete* counterparts operate directly on an actual *compact encoding* of temporal databases. The relationship between abstract and concrete temporal query languages is also implicitly present in the notion of snapshot equivalence [7]. Moreover, Bettini *et al.* [1] proposed to distinguish between *explicit* and *implicit* information in a temporal database. The explicit information is stored in the database and used to derive the implicit information through *semantic assumptions*. Semantic assumptions about fact persistence play a role similar to mappings between concrete and abstract databases, while other assumptions are used to address time-granularity issues.

## Abstract Temporal Query Languages

Most temporal query languages derived by temporally extending the relational calculus can be classified as abstract temporal query languages. Their semantics is defined in terms of abstract temporal databases which, in turn, are typically defined within the point-stamped temporal data model, in particular *without* any additional hidden assumptions about the meaning of tuples in instances of temporal relations.

**Example**. The *employment histories* in an abstract temporal data model would most likely be captured by a simpler schema "$\texttt{Employment}(\texttt{Date}, \texttt{EID}, \texttt{Company})$", with the intended meaning that a person identified by `EID` was working for `Company` on a particular `Date`. While instances of such a schema can be potentially very large (especially when a fine granularity of time is used), formulating queries is now much more natural.

Choosing abstract temporal query languages over concrete ones resolves the first design issue: the temporal values used by the former languages are time instants equipped with an appropriate temporal ordering (which is typically a linear order over the instants), and possibly other predicates such as temporal distance. The second design issue—access to temporal values—may be resolved in two different ways, as exemplified by the following two different query languages:

- Temporal Relational Calculus (TRC): a two-sorted first-order logic with variables and quantifiers explicitly ranging over the time and data domains (see the entry Temporal Relational Calculus).
- First-order Temporal Logic (FOTL): a language with an implicit access to timestamps using temporal connectives (see the entry Temporal Logic in Database Query Languages).

**Example**. The GAPS query is formulated as follows:

**TRC:** $\exists t_1, t_3.t_1 < t_2 < t_3 \wedge \exists c.\texttt{Employment}(t_1, x, c) \wedge (\neg\exists c.\texttt{Employment}(t_2, x, c)) \wedge \exists c.\texttt{Employment}(t_3, x, c)$

**FOTL:** $\blacklozenge\exists c.\texttt{Employment}(x, c) \wedge (\neg\exists c.\texttt{Employment}(x, c)) \wedge \Diamond\exists c.\texttt{Employment}(x, c)$

Here, the explicit access to temporal values (in TRC) using the variables $t_1$, $t_2$, and $t_3$ can be contrasted with the implicit access (in FOTL) using the temporal operators $\blacklozenge$ (read "sometime in the past") and $\Diamond$ (read "sometime in the future"). The conjunction in the FOTL query represents an implicit temporal join. The formulation in

TRC leads immediately to an equivalent way of expressing the query in SQL/TP [9], an extension of SQL based on TRC (see the entry SQL-based Temporal Query Languages).

**Example**. The above query can be formulated in SQL/TP as follows:

```
SELECT  t.Date, e1.EID
FROM    Employment e1, Time t, Employment e2
WHERE   e1.EID = e2.EID AND e1.Date < e2.Date
  AND NOT EXISTS ( SELECT *
                   FROM   Employment e3
                   WHERE  e1.EID = e3.EID   AND t.Date = e3.Date
                     AND  e1.Date < e3.Date AND e3.Date < e2.Date )
```

The unary constant relation `Time` contains all time instants in the time domain (in our case, all `Dates`) and is only needed to fulfill syntactic SQL-style requirements on attribute ranges. However, despite of the fact that the instance of this relation is not finite, the query can be efficiently evaluated [9].

Note also that in all the above cases, the formulation is *exactly the same* as if the underlying temporal database used the *plain* relational model (allowing for attributes ranging over time instants).

The two languages, FOTL and TRC, are the counterparts of the snapshot and timestamp models (cf. the entry Point-stamped Data Models) and are the roots of many other temporal query languages, ranging from the more TRC-like temporal extensions of SQL, to more FOTL-like temporal relational algebras (e.g., the conjunction in temporal logic directly corresponds to a temporal join in a temporal relational algebra, as both of them induce an *implicit equality* on the associated time attributes). The precise relationship between these two groups of languages is investigated in the entry Temporal Logic in Database Query Languages.

Temporal integrity constraints over point-stamped temporal databases can also be conveniently expressed in TRC or FOTL (see the entry Temporal Integrity Constraints).

**Multiple Temporal Dimensions and Complex Values.** While the abstract temporal query languages are typically defined in terms of the point-based temporal data model, they can similarly be defined with respect to complex temporal values, e.g., pairs (or tuples) of time instants or even sets of time instants. In these cases, in particular in the case of set-valued attributes, it is important to remember that the set values are treated as *indivisible objects*, and hence truth (i.e., query semantics) is associated with the entire objects, but not necessarily with their components/subparts. For a detailed discussion of this issue, see the entry Telic Distinction in Temporal Databases.

## Concrete Temporal Query Languages

Although abstract temporal query languages provide a convenient and clean way of specifying queries, they are not immediately amenable to implementation: the main problem is that, in practice, in temporal databases facts persist over periods of time. Storing all true facts individually *for every time instant* during a period would be prohibitively expensive or, in the case of infinite time domains such as *dense time*, even impossible.

Concrete temporal query languages avoid these problems by operating directly on the compact encodings of temporal databases (see the discussion of compact encodings in the entry on Point-stamped Temporal Models). The most commonly used encoding is the one that uses *intervals*. However, in this setting, a tuple that associates a fact with such an interval is a compact representation of the association between the same fact and *all the time instants that belong to this interval*. This observation leads to the design choices that are commonly present in such languages:

- Coalescing is used, explicitly or implicitly, to consolidate representations of (sets of) time instants associated *with the same fact*. In the case of interval-based encodings, this leads to coalescing adjoining or overlapping intervals into a single interval (see the entry Temporal Coalescing). Note that coalescing only changes the *concrete representation* of a temporal relation, not its meaning (i.e., the abstract temporal relation); hence it has no counterpart in abstract temporal query languages.

- Implicit *set operations* on time values are used in relational operations. For example, conjunction (join)

typically uses set intersection to generate a compact representation of the time instants attached to the facts in the result of such an operation.

**Example**. For the running example, a concrete schema for the employment histories would typically be defined as "`Employment(VT, EID, Company)`", where `VT` is a valid time attribute ranging over periods (intervals). The GAPS query can be formulated in a calculus-style language corresponding to TSQL2 (see the entry on TSQL2) along the following lines:

$$\exists I_1, I_2.\, [\exists c.\texttt{Employment}(I_1, x, c)] \wedge [\exists c.\texttt{Employment}(I_2, x, c)] \wedge I_1 \texttt{ precedes } I_2 \wedge I = [\mathsf{end}(I_1) + 1, \mathsf{begin}(I_2) - 1].$$

In particular, the variables $I_1$ and $I_2$ range over periods and the `precedes` relationship is one of Allen's interval relationships. The final conjunct, $I = [\mathsf{end}(I_1) + 1, \mathsf{begin}(I_2) - 1]$, creates a new period corresponding to the time instants related to a person's *gap in employment*; this interval value is explicitly constructed from the end and start points of $I_1$ and $I_2$, respectively. For the query to be correct, however, the results of evaluating the bracketed subexpressions, e.g., "$[\exists c.\texttt{Employmeent}(I_1, x, c)]$," have to to be *coalesced*. Without the insertion of the explicit coalescing operators, the query is *incorrect*. To see that, consider a situation in which a person $p_0$ is first employed by a company $c_1$, then by $c_2$, and finally by $c_3$, without any gaps in employment. Then without coalescing of the bracketed subexpressions of the above query, $p_0$ will be returned as a part of the result of the query, which is incorrect. Note also that it is not enough for the underlying (concrete) database to be coalesced.

The need for an explicit use of coalescing makes often the formulation of queries in some concrete SQL-based temporal query languages cumbersome and error-prone.

An orthogonal issue is the difference between explicit and implicit access to temporal values: this distinction carries over to the concrete temporal languages as well. Typically, the various temporal extensions of SQL are based on the assumption of an explicit access to temporal values (often employing a built-in *valid time* attribute ranging over intervals or temporal elements), while many temporal relational algebras have chosen to use the implicit access based on temporally extending standard relational operators such as temporal join or temporal projection.

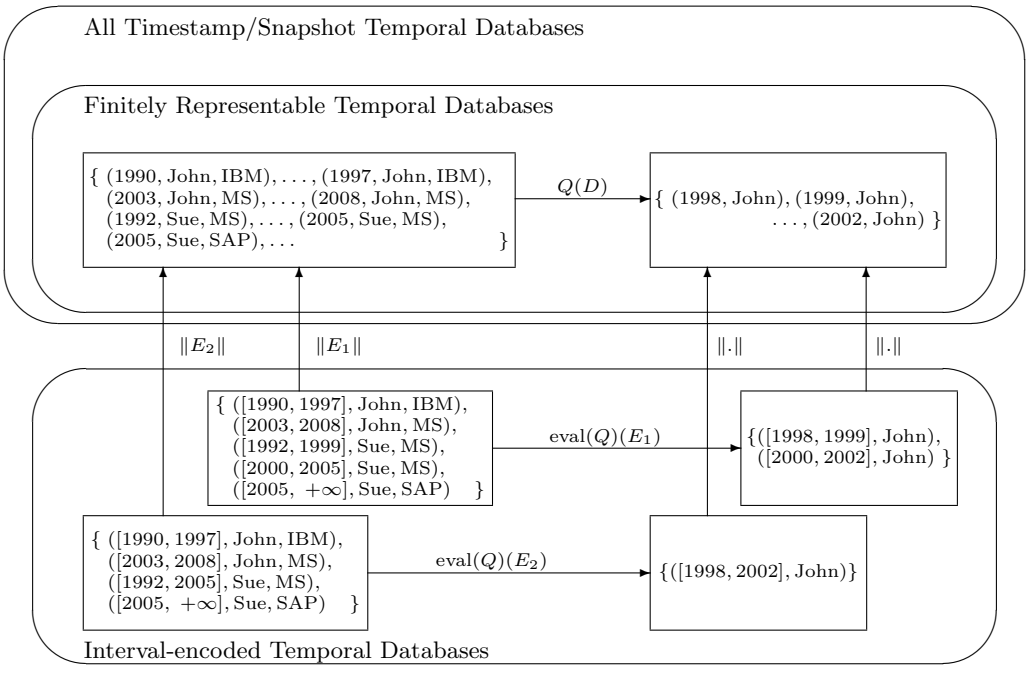



Figure 1: Query Evaluation over Interval Encodings of Point-stamped Temporal Databases

**Compilation and Query Evaluation.** An alternative to allowing users direct access to the encodings of temporal databases is to develop techniques that allow the evaluation of *abstract temporal queries* over these encodings. The main approaches are based on *query compilation* techniques that map abstract queries to concrete queries, while preserving query answers. More formally:

$$Q(\|E\|) = \|\mathsf{eval}(Q)(E)\|,$$

where $Q$ an abstract query, $\mathsf{eval}(Q)$ the corresponding concrete query, $E$ is a concrete temporal database, and $\|.\|$ a mapping that associates encodings (concrete temporal databases) with their abstract counterparts (cf. Figure 1). Note that a single abstract temporal database, $D$, can be encoded using several *different* instances of the corresponding concrete database, e.g., $E_1$ and $E_2$ in Figure 1.

Most of the practical temporal data models adopt a common approach to physical representation of temporal databases: with every fact (usually represented as a tuple), a *concise encoding* of the set of time points at which the fact holds is associated. The encoding is commonly realized by *intervals* [6, 7] or temporal elements (finite unions of intervals). For such an encoding it has been shown that both First-Order Temporal Logic [4] and Temporal Relational Calculus [8] queries can be *compiled* to first-order queries over a natural relational representation of the interval encoding of the database. Evaluating the resulting queries yields the interval encodings of the answers to the original queries, as if the queries were directly evaluated on the point-stamped temporal database. Similar results can be obtained for more complex encodings, e.g., periodic sets, and for abstract temporal query languages that adopt the duplicate semantics matching the SQL standard, such as SQL/TP [9].

## KEY APPLICATIONS

Temporal query languages are primarily used for querying temporal databases. However, because of their generality they can be applied in other contexts as well, e.g., as an underlying conceptual foundation for querying sequences and data streams [5].

## CROSS REFERENCE

Allen's relations, bitemporal relation, constraint databases, key, nested relational model, non first normal form (N1NF), point-stamped temporal models, relational model, snapshot equivalence, SQL, telic distinction in temporal databases, temporal coalescing, temporal data models, temporal element, temporal granularity, temporal integrity constraints, temporal join, temporal logic in database query languages, temporal relational calculus and algebra, time domain, time instant, TSQL2, transaction time, valid time.

## RECOMMENDED READING


[1] C. Bettini, X. S. Wang, and S. Jajodia. Temporal Semantic Assumptions and Their Use in Databases. *Knowledge and Data Engineering*, 10(2):277–296, 1998.

[2] J. Chomicki. Temporal Query Languages: A Survey. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic, First International Conference*, pages 506–534. Springer-Verlag, LNAI 827, 1994.

[3] J. Chomicki and D. Toman. Temporal Databases. In M. Fischer, D. Gabbay, and L. Villa, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier *Foundations of Artificial Intelligence*, 2005.

[4] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL Databases with Temporal Logic. *ACM Transactions on Database Systems*, 26(2):145–178, 2001.

[5] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *International Conference on Very Large Data Bases*, pages 492–503, 2004.

[6] S. B. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings, 1993.

[7] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.

[8] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *ACM Symposium on Principles of Database Systems*, pages 58–67, 1996.

[9] D. Toman. Point-based Temporal Extensions of SQL. In *International Conference on Deductive and Object-Oriented Databases*, pages 103–121, 1997.

## ALLEN'S RELATIONS

**Peter Revesz, University of Nebraska-Lincoln**, http://www.cse.unl.edu/~revesz/
**Paolo Terenziani Universita' del Piemonte Orientale "Amedeo Avogadro",**
http://www.di.unito.it/~terenz/

### SYNONYMS
Qualitative relations between time intervals, qualitative temporal constraints between time intervals

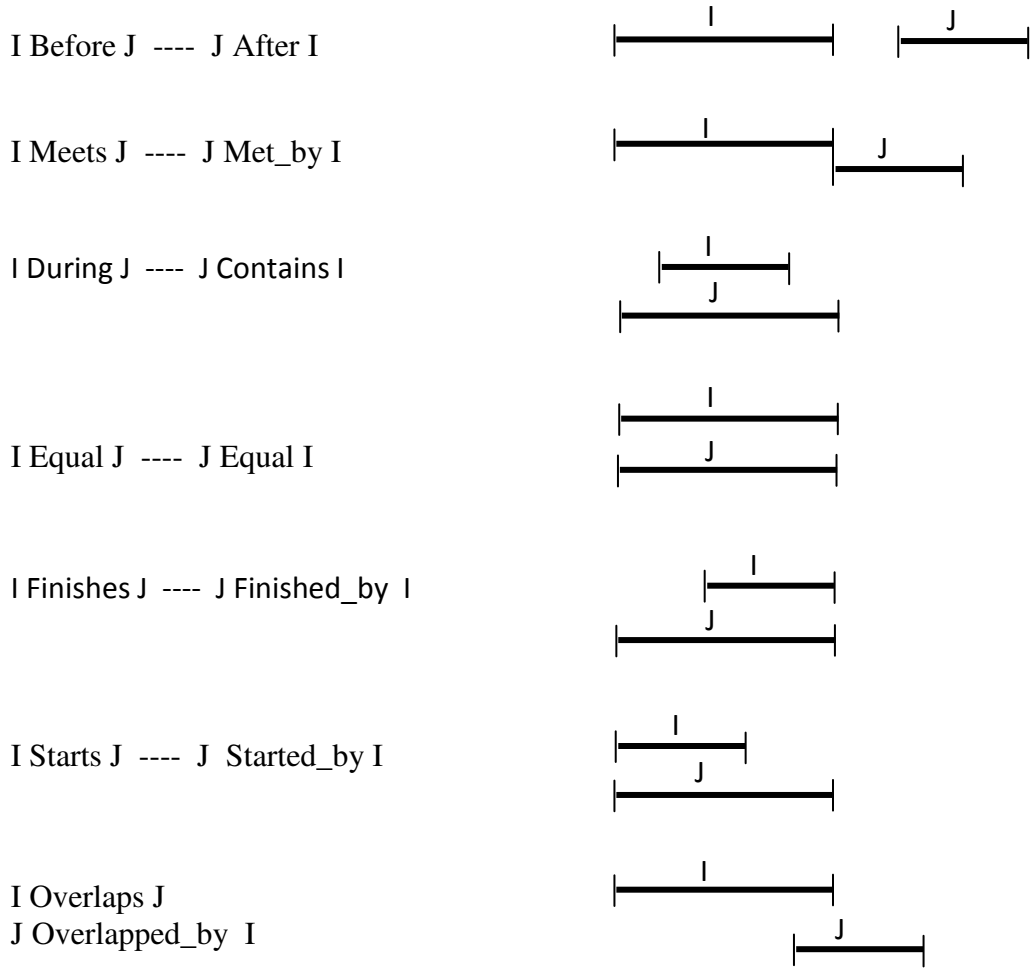
### DEFINITION
A (convex) *time interval* I is the set of all time points between a starting point (usually denoted by $I^-$) and an ending point ($I^+$). Allen's relations model all possible relative positions between two time intervals [1]. There are 13 different possibilities, depending on the relative positions of the endpoints of the intervals.

| | $I^- \; J^-$ | $I^- \; J^+$ | $I^+ \; J^-$ | $I^+ \; J^+$ |
|---|---|---|---|---|
| **After** | | > | | |
| **Before** | | | < | |
| **Meets** | | | = | |
| **Met_by** | | = | | |
| **During** | > | | | < |
| **Contains** | < | | | > |
| **Equal** | = | | | = |
| **Finishes** | > | | | = |
| **Finished_by** | < | | | = |
| **Starts** | = | | | < |
| **Started_by** | = | | | > |
| **Overlaps** | < | | > | < |
| **Overlapped_by** | > | < | | > |

For example, "There will be a guest speaker during the Database System class" can be represented by Allen's relation $I_{Guest}$ *During* $I_{Database}$ (or by $I^-_{Guest} > I^-_{Database} \wedge I^+_{Guest} < I^+_{Database}$ considering the relative position of the endpoints; *point relations* are discussed in the entry "Temporal Constraints" of this Encyclopedia). Moreover, any subset of the 13 relations, excluding the empty subset, is a relation in Allen's Interval Algebra (therefore, there are $2^{13}$-1 relations in Allen's Algebra). Such subsets are used in order to denote ambiguous cases, in which the relative position of two intervals is only partially known. For instance, $I_1$ (Before, Meets, Overvaps) $I_2$ represents the fact that $I_1$ is before *or* meets *or* overlaps $I_2$.

### MAIN TEXT
In many cases, the exact time interval when facts occur is not known, but (possibly imprecise) information on the relative temporal location of facts is available. Allen's relations allow one to represent such cases of *temporal indeterminacy*. For instance, planning in Artificial Intelligence

has been the first application of Allen's relations. A graphical representation of the basic 13 Allen's relations is shown in the following figure.




Allen's relations are specific cases of *temporal constraints* (see the entry "Temporal Constraints" of this Encyclopedia): namely, they are qualitative temporal constraints between time intervals. Given a set of such constraints, *qualitative temporal reasoning* can be used in order to make inferences (e.g., to check whether the set of constraints is consistent; see in the entry "Qualitative temporal reasoning" of this Encyclopedia).
Finally, notice that, in many entries of this Encyclopedia, the term *(time) period* has been used with the same meaning of *(time) interval* in this entry.

**CROSS REFERENCES**
Temporal constraints
Qualitative temporal reasoning
Temporal indeterminacy

**REFERENCES* (optional)**
[1] Allen, J.F., Maintaining knowledge about temporal intervals, C*ommunications of the ACM 26(11)*: 832-843, 1983.

# APPLICABILITY PERIOD

Christian S. Jensen
Aalborg University, Denmark
http://www.cs.aau.dk/~csj
Richard T. Snodgrass
University of Arizona
http://www.cs.arizona.edu/people/rts/

**SYNONYMS**

none

**DEFINITION**

The *applicability period* (or *period of applicability*) for a modification (generally an insertion, deletion, or update) is the time period for which that modification is to apply to. Generally the modification is a sequenced modification and the period applies to valid time. This period should be distinguished from *lifespan*.

**MAIN TEXT**

The applicability period is specified within a modification statement. In constrast, the lifespan is an aspect of a stored fact.

This illustration uses the TSQL2 language, which has an explicit `VALID` clause to specify the applicability period within an `INSERT`, `DELETE`, or `UPDATE` statement.

For insertions, the applicability period is the valid time of the fact being inserted. The following states that Ben is in the book department for one month in 2007.

```
INSERT INTO EMPLOYEE
VALUES ('Ben', 'Book')
VALID PERIOD '[15 Feb 2007, 15 Mar 2007]'
```

For a deletion, the applicability period states for what period of time the deletion is to apply. The following modification states that Ben in fact wasn't in the book department during March.

```
DELETE FROM EMPLOYEE
WHERE Name = 'Ben'
VALID PERIOD '[1 Mar 2007, 31 Mar 2007]'
```

After this modification, the lifespan would be February 15 through February 28.

Similarly, the applicability period for an `UPDATE` statement would affect the stored state just for the applicability period.

A *current modification* has a default applicability period that either extends from the time the statement is executed to forever or, when now-relative time is supported, from the time of execution to the ever-increasing current time for insertions.

**CROSS REFERENCE***

Current Semantics, Lifespan, Now in Temporal Databases, Sequenced Semantics, Temporal Database, Time Period, TSQL2, Valid Time

**REFERENCES***

R. T. Snodgrass (editor), **The TSQL2 Temporal Query Language**. Kluwer Academic Publishers, 674+xxiv pages, 1995.

R. T. Snodgrass, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, July 1999, 504+xxiv pages.

# Atelic Data (DEFINITIONAL ENTRY)

Vijay Khatri
*Operations and Decision Technologies Department*
*Kelley School of Business*
*Indiana University*
*Bloomington, Indiana, USA*
*vkhatri@indiana.edu*
URL: http://mypage.iu.edu/~vkhatri/

Richard T. Snodgrass
*Department of Computer Science*
*711 Gould Simpson*
*University of Arizona*
*P.O. Box 210077*
*Tucson, Arizona, USA*
*rts@cs.arizona.edu*
URL: http://www.cs.arizona.edu/people/rts/

Paolo Terenziani
*Universita' del Piemonte Orientale "Amedeo Avogadro"*
*Via Bellini 25, 15100 Alessandria, Italy*
*terenz@di.unito.it*
URL: http://www.di.unito.it/~terenz/

## TITLE
Atelic data

## SYNONYMS
Snapshot data, point-based temporal data

## DEFINITION
Atelic data is temporal data describing facts that do not involve a goal or culmination; in ancient Greek, *telos* means 'goal,' and $\alpha$ is used as prefix to denote negation. In the context of temporal databases, atelic data is that data for which both *upward* and *downward* (temporal) *inheritance* holds. Specifically,

- **Downward inheritance**. The *downward inheritance* property implies that one can infer from temporal data *d* that holds at valid time *t* (where t is a time period) that *d* holds in any sub-period (and sub-point) of *t* can be inferred from temporal data *d* that holds at valid time *t* (where t is a time period).
- **Upward inheritance**. The *upward inheritance* property implies that one can infer from temporal data *d* that holds at two consecutive or overlapping time periods $t_1$ and $t_2$ that d holds in the union time period $t_1 \cup t_2$.

Atelic data is differentiated from telic data, in which neither upward nor downward inheritance holds.

## MAIN TEXT

Starting from Aristotle, researchers in areas such as philosophy, linguistics, cognitive science and computer science have noticed that different types of facts can be distinguished according to their temporal behavior. Specifically, since atelic facts do not have any specific goal or culmination, they can be seen as "temporally homogeneous" facts, so that both upward and downward inheritance holds for them. For example, the fact that an employee (say, John) works for a company (say, ACME) would be considered atelic because of lack of goal or accomplishment. As a consequence, if John has worked for ACME from January 20 2007 to September 23 2007, it can be correctly inferred that John was working for ACME in May 2007 (or at any specific time point in May; therefore, downward inheritance does hold); furthermore, from the additional fact that John has also worked for ACME from September 23 2007 to February 2 2008, it can be correctly inferred that John worked for ACME from January 20 2007 to February 2 2008 (therefore, upward inheritance does hold).

In Aristotle's categorization, all possible facts are divided into two categories, telic and atelic; a telic fact, e.g., "John built a house" has goal or culmination.

Since both upward and downward inheritance properties hold for atelic data, such data supports the conventional "snapshot-by-snapshot" (i.e., point-based) viewpoint: the intended semantics of a temporal database is the set of conventional (atemporal) databases holding at each time point. As a consequence, most approaches to temporal databases support (only) atelic facts even if, in several cases, the representation allows, as a syntactic sugar, the use of periods to denote convex sets of time points. An integrated temporal database model that supports both telic and atelic data semantics has been developed [2].

## CROSS REFERENCES

Period-Stamped Data Models, Point-Stamped Data Models, Telic Distinction in Temporal Databases

## REFERENCES

[1] Aristotle, The Categories, on Interpretation. Prior Analytics. Cambridge, MA, Harvard University Press.

[2] P. Terenziani and R. T. Snodgrass, "Reconciling Point-based and Interval-based Semantics in Temporal Relational Databases: A Proper Treatment of the Telic/Atelic Distinction," IEEE *Transactions on Knowledge and Data Engineering*, 16(4):540-551, 2004.

# BI-TEMPORAL INDEXING

Mirella M. Moro
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
http://www.inf.ufrgs.br/~mirella/

Vassilis J. Tsotras
University of California, Riverside
Riverside, CA 92521, USA
http://www.cs.ucr.edu/~tsotras

## SYNONYMS

Bi-temporal Access Methods

## DEFINITION

A bi-temporal index is a data structure that supports both temporal time dimensions, namely, transaction-time (the time when a fact is stored in the database) and valid-time (the time when a fact becomes valid in reality). The characteristics of the time dimensions supported imply various properties that the bi-temporal index should have to be efficient. As traditional indices, the performance of a temporal index is described by three costs: (i) storage cost (i.e., the number of pages the index occupies on the disk), (ii) update cost (the number of pages accessed to perform an update on the index; for example when adding, deleting or updating a record), and (iii) query cost (the number of pages accessed for the index to answer a query).

## HISTORICAL BACKGROUND

Most of the early work on temporal indexing has concentrated on providing solutions for transaction-time databases. A basic property of transaction-time is that it always *increases*. Each newly recorded piece of data is time-stamped with a new, larger, transaction time. The immediate implication of this property is that previous transaction times *cannot* be changed. Hence, a transaction-time database can “rollback” to, or answer queries for, any of its previous states.

On the other hand, a valid-time database maintains the entire temporal behavior of an enterprise as best known now. It stores the current knowledge about the enterprise's past, current or even future behavior. If errors are discovered about this temporal behavior, they are corrected by modifying the database. In general, if the knowledge about the enterprise is updated, the new knowledge modifies the existing one. When a correction or an update is applied, previous values are not retained. It is thus not possible to view the database as it was before the correction/update.

By supporting both valid and transaction time, a bi-temporal database combines the features of the other temporal database types. While it keeps its past states, it also supports changes anywhere in the valid time domain. Hence, the overlapping and persistent methodologies proposed for transaction-time indexing (for details see chapter on Transaction-Time Indexing) can be applied [6, 9, 5]. The difference with transaction-time indexing is that the underlying access method should be able to dynamically manage intervals (like an R-tree, a quad-tree etc.). For a worst-case comparison of temporal access methods, the reader is referred to [7].

## SCIENTIFIC FUNDAMENTALS

When considering temporal indexing, it is important to realize that the valid and transaction time dimensions are *orthogonal* [8]. While in various scenarios it may be assumed that data about a fact is entered in the database at the same time as when it happens in the real world (i.e., valid and transaction time coincide), in practice, there are many applications where this assumption does not hold. For example, data records about the sales that occurred during a given day are recorded in the database at the end of the day (when batch processing of all data collected during the day is performed). Moreover, a recorded valid time may represent a later time instant than the transaction time when it was recorded. For example, a contract may be valid for an interval that is later than the (transaction) time when this information was entered in the database. The above properties are critical in the design of a bi-temporal access method since the support of both valid and transaction time affects directly the way records are created or updated. Note that the term “interval” is used here to mean a “convex subset of the time domain" (and not a "directed duration"). This concept has also been named a “period”; in this discussion however, only the term “interval” is used.

The reader is referred to the entry on Transaction-Time Indexing, in which a transaction time database was abstracted as an evolving collection of objects; updates arrive in increasing transaction-time order and are always applied on the latest state of this set. In other words, previous states cannot be changed. Thus a transaction-time database represents and stores the database activity; objects are associated with intervals based on this database activity. In contrast, in the chapter on Valid-Time Indexing, a valid-time database was abstracted as an evolving collection of interval-objects, where each interval represents the validity interval of an object. The allowable changes in this environment are the addition/deletion/ modification of an interval-object. A difference with the transaction-time abstraction is that the collection's evolution (past states) is *not* kept. Note that when considering the valid time dimension, changes do not necessarily come in increasing time order; rather they can affect *any* interval in the collection. This implies that a valid-time database can correct errors in previously recorded data. However, only a single data state is kept, the one resulting after the correction is applied.

A bi-temporal database has the characteristics of both approaches. Its abstraction maintains the evolution (through the support of transaction-time) of a dynamic collection of (valid-time) interval-objects. Figure 1 offers a conceptual view of a bi-temporal database. Instead of maintaining a single collection of interval-objects (as a valid-time database does) a bi-temporal database maintains a sequence of such collections $C(t_i)$ indexed by transaction-time. Assume that each interval $I$ represents the validity interval of a contract in a company. In this environment, the user can represent how the knowledge about company contracts evolved. In Figure 1, the $t$-axis ($v$-axis) corresponds to transaction (valid) times. At transaction time $t_1$, the database starts with interval-objects $I_x$ and $I_y$. At $t_2$, a new interval-object $I_z$ is recorded, etc. At $t_5$ the valid-time interval of object $I_x$ is modified to a new length.

When an interval-object $I_j$ is inserted in the database at transaction-time $t$, a record is created with the object's surrogate (contract_no $I_j$), a valid-time interval (contract duration), and an initial transaction-time interval [$t$, $UC$). When an object is inserted, it is not yet known if it (ever) will be updated. Therefore, the right endpoint of the transaction-time interval is filled with the variable $UC$ (Until Changed), which will be changed to another transaction time if this object is later updated. For example, the record for interval-object $I_z$ has transaction-time interval [$t_2$, $t_4$), because it was inserted in the database at transaction-time $t_2$ and was “deleted” at $t_4$. Note that the collections $C(t_3)$ and $C(t_4)$ correspond to the collections $C_a$ and $C_b$ of Figure 1 in the Valid-Time Indexing chapter, assuming that at transaction-time $t_4$ the erroneous contract $I_z$ was deleted from the database.

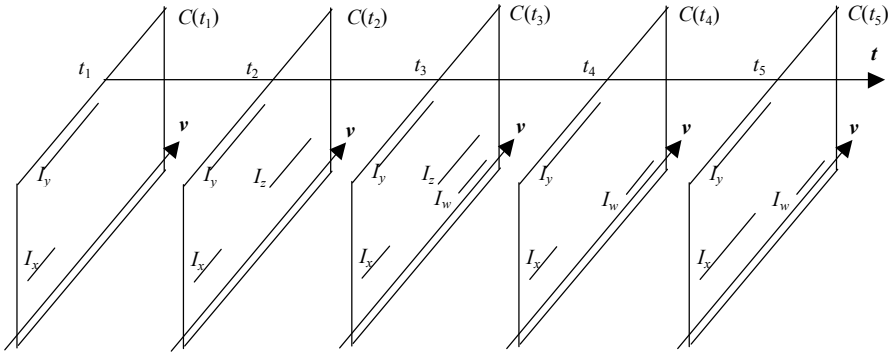



Figure 1. A bi-temporal database.

Based on the above discussion, an index for a bi-temporal database should: (i) store past states, (ii) support addition/deletion/modification changes on the interval-objects of its current logical state, and (iii) efficiently access and query the interval-objects on any state.

Figure 1 summarizes the differences among the various database types. Each collection $C(t_i)$ can be thought of on its own, as a separate valid-time database. A valid-time database differs from a bi-temporal

database since it keeps *only one* collection of interval-objects (the latest). A transaction-time database differs from a bi-temporal database in that it maintains the history of an evolving set of *plain*-objects instead of *interval*-objects. A transaction-time database differs from a conventional (non-temporal) database in that it also keeps its *past* states instead of only the latest state. Finally, the difference between a valid-time and a conventional database is that the former keeps *interval*-objects (and these intervals can be queried).

There are three approaches that can be used for indexing bi-temporal databases.

**Approach 1:** The first one is to have each bi-temporal object represented by a "bounding rectangle" created by the object's valid and transaction-time intervals, and to store it in a conventional multi-dimensional structure like the R-tree. While this approach has the advantage of using a single index to support both time dimensions, the characteristics of transaction-time create a serious overlapping problem [5]. A bi-temporal object with valid-time interval *I* that is inserted in the database at transaction time *t*, is represented by a rectangle with a transaction-time interval of the form [*t*, *UC*). All bi-temporal objects that have not been deleted (in the transaction sense) will share the common transaction-time endpoint *UC* (which in a typical implementation, could be represented by the largest possible transaction time). Furthermore, intervals that remain unchanged will create long (in the transaction-time axis) rectangles, a reason for further overlapping. A simple bi-temporal query that asks for all valid time intervals that at transaction time $t_i$ contained valid time $v_j$, corresponds to finding all rectangles that contain point ($t_i$, $v_j$).

Figure 2 illustrates the bounding-rectangle approach; only the valid and transaction axis are shown. At $t_5$, the valid-time interval $I_1$ is modified (enlarged). As a result, the initial rectangle for $I_1$ ends at $t_5$, and a new enlarged rectangle is inserted ranging from $t_5$ to *UC*.

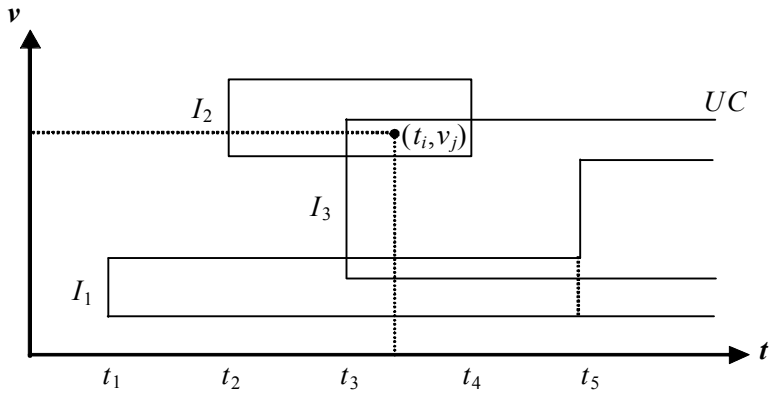



Figure 2. The bounding-rectangle approach for bi-temporal objects.

**Approach 2:** To avoid overlapping, the use of two R-trees has also been proposed [5]. When a bi-temporal object with valid-time interval *I* is added in the database at transaction-time *t*, it is inserted at the *front* R-tree. This tree keeps bi-temporal objects whose right transaction endpoint is unknown. If a bi-temporal object is later deleted at some time $t' > t$, it is physically deleted from the front R-tree and inserted as a rectangle of height *I* and width from *t* to *t'* in the *back* R-tree. The back R-tree keeps bi-temporal objects with known transaction-time interval. At any given time, all bi-temporal objects stored in the front R-tree share the property that they are alive in the transaction-time sense. The temporal information of every such object is thus represented simply by a vertical (valid-time) interval that "cuts" the transaction axis at the transaction-time when this object was inserted in the database. Insertions in the front R-tree objects are in increasing transaction time while physical deletions can happen anywhere on the transaction axis.

In Figure 3, the two R-trees methodology for bi-temporal data is divided according to whether their right transaction endpoint is known. The scenario of Figure 2 is presented here (i.e., after time $t_5$ has elapsed). The query is then translated into an interval intersection and a point enclosure problem. A simple bi-temporal query that asks for all valid time intervals which contained valid time $v_j$ at transaction time $t_i$, is answered with two searches. The back R-tree is searched for all rectangles that contain point ($t_i$, $v_j$). The front R-tree is searched for all vertical intervals that intersect a horizontal interval $H$ that starts from the beginning of transaction time and extends until point $t_i$ at height $v_j$.

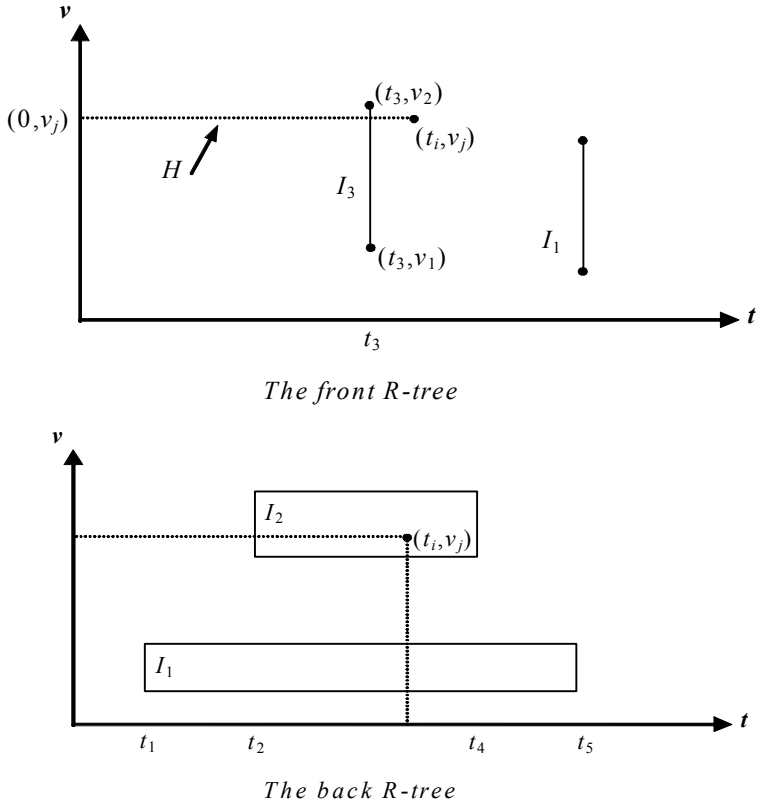



Figure 3. The two R-tree methodology for bi-temporal data.

When an R-tree is used to index bi-temporal data, overlapping may also incur if the valid-time intervals extend to the ever-increasing *now*. One approach could be to use the largest possible valid-time timestamp to represent the variable *now*. In [2] the problem of addressing both the *now* and *UC* variables is addressed by using bounding rectangles/regions that increase as the time proceeds. A variation of the R-tree, the GR-tree is presented. The index leaf nodes capture the exact geometry of the bi-temporal regions of data. Bi-temporal regions can be static or growing, rectangles or stair-shapes. Two versions of the GR-tree are explored, one using minimum bounding rectangles in non-leaf nodes, and one using minimum bounding regions in non-leaf nodes. Details appear in [2].

**Approach 3:** Another approach to address bi-temporal problems is to use the notion of partial persistence [4,1]. This solution emanates from the abstraction of a bi-temporal database as a sequence of collections $C(t)$ (in Figure 1) and has two steps. First, a good index is chosen to represent each $C(t)$. This index must support dynamic addition/deletion of (valid-time) interval-objects. Second, this index is made partially persistent. The collection of queries supported by the interval index structure implies which queries are answered by the bi-temporal structure. Using this approach, the Bi-temporal R-tree that takes an R-tree and makes it partially persistent was introduced in [5].

Similar to the transaction-time databases, one can use the “overlapping” approach [3] to create an index for bi-temporal databases. It is necessary to use an index that can handle the valid-time intervals and an overlapping approach to provide the transaction-time support. Multi-dimensional indexes can be used for supporting intervals. For example, an R-tree or a quad-tree. The Overlapping-R-tree was proposed in [6],

where an R-tree maintains the valid time intervals at each transaction time instant. As intervals are added/deleted or updated, overlapping is used to share common paths in the relevant R-trees. Likewise, [9] proposes the use of quad-trees (which can also be used for spatiotemporal queries).

There are two advantages in "viewing" a bi-temporal query as a "partial persistence" or "overlapping" problem. First, the valid-time requirements are disassociated from the transaction-time ones. More specifically, the valid time support is provided from the properties of the R-tree while the transaction time support is achieved by making this structure "partially persistent" or "overlapping". Conceptually, this methodology provides fast access to the $C(t)$ of interest on which the valid-time query is then performed. Second, changes are always applied to the most current state of the structure and last until updated (if ever) at a later transaction time, thus avoiding the explicit representation of variable *UC*. Considering the two approaches, overlapping has the advantage of simpler implementation, while the partial-persistence approach avoids the possible logarithmic space overhead.

## KEY APPLICATIONS

The importance of temporal indexing emanates from the many applications that maintain temporal data. The ever increasing nature of time imposes the need for many applications to store large amounts of temporal data. Accessing such data specialized indexing techniques is necessary. Temporal indexing has offered many such solutions that enable fast access.

## CROSS REFERENCES

Temporal Databases, Transaction-Time Indexing, Valid-Time Indexing, B+-tree, R-tree

## RECOMMENDED READING

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer (1996). An Asymptotically Optimal Multiversion B-Tree. VLDB J. 5(4): 264-275.

[2] R. Bliujute, C.S. Jensen, S. Saltenis and G. Slivinskas (1998). R-Tree Based Indexing of Now-Relative Bitemporal Data. VLDB Conference, pp: 345-356.

[3] F. W. Burton, M.M. Huntbach, and J.G. Kollias (1985). Multiple Generation Text Files Using Overlapping Tree Structures. Comput. J. 28(4): 414-416.

[4] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan (1989). Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124.

[5] A. Kumar, V.J. Tsotras, and C. Faloutsos (1998). Designing Access Methods for Bitemporal Databases. IEEE Transactions on Knowledge Data Engineering 10(1): 1-20.

[6] M.A. Nascimento and J.R.O. Silva (1998). Towards historical R-trees. Proceedings of SAC, pages 235-240

[7] B. Salzberg and V.J. Tsotras (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2): 158-221.

[8] R.T. Snodgrass and I. Ahn (1986). Temporal Databases. IEEE Computer 19(9): 35-42.

[9] T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos (2000). Overlapping Linear Quadtrees and Spatio-Temporal Query Processing. Comput. J. 43(4): 325-343.

# BITEMPORAL INTERVAL

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

none

## DEFINITION

Facts in a bitemporal database may be timestamped by time values that are products of time intervals drawn from two orthogonal time domains that model valid time and transaction time, respectively. A *bitemporal interval* then is given by an interval from the valid-time domain and an interval from the transaction-time domain, and denotes a rectangle in the two-dimensional space spanned by valid and transaction time.

When associated with a fact, a bitemporal interval then identifies an interval (valid time) during which that fact held (or holds or will hold) true in reality, as well as identifies an interval (transaction time) when that belief (that the fact was true during the specified valid-time interval) was held, i.e., was part of the current database state.

## MAIN TEXT

In this definition, a time interval denotes a convex subset of the time domain. Assuming a discrete time domain, a bitemporal interval can be represented with a non-empty set of bitemporal chronons or granules.

## CROSS REFERENCE*

Bitemporal Relation, Chronon, Temporal Database, Temporal Granularity, Time Interval, Transaction Time, Valid Time

## REFERENCES*

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# Bitemporal Relation

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Temporal relation; Fully temporal relation; Valid-time and transaction-time relation

## DEFINITION

A *bitemporal relation* captures exactly one valid time aspect and one transaction time aspect of the data it contains. This relation inherits its properties from valid-time relations and transaction-time relations. There are no restrictions as to how either of these temporal aspects may be incorporated into the tuples.

## MAIN TEXT

In this definition, "bi" refers to the capture of exactly two temporal aspects. An alternative definition states that a bitemporal relation captures one or more valid times and one or more transaction times. In this definition, "bi" refers to the existence of exactly two types of times.

One may adopt the view that the data in a relation represents a collection of logical statements, i.e., statements that can be assigned a truth values. The valid times of these so-called facts are the times when these are true in the reality modeled by the relation. In cases where multiple realities are perceived, a single fact may have multiple, different valid times. This might occur in a relation capturing archaeological facts for which there no agreements among the archaeologists. In effect, different archaeologists perceive different realities.

Transaction times capture when database objects are current in a database. In case an object migrates from one database to another, the object may carry along its transaction times from the predecessor databases. This then calls for relations that capture multiple transaction times.

The definition of bitemporal is used as the basis for applying bitemporal as a modifier to other concepts such as "query language." A query language is bitemporal if and only if it supports any bitemporal relation. Hence, most query languages involving both valid and transaction time may be characterized as bitemporal.

Relations are named as opposed to databases because a database may contain several types of relations. Most relations involving both valid and transaction time are bitemporal according to both definitions.

Concerning synonyms, the term "temporal relation" is commonly used. However, it is also used in a generic and less strict sense, simply meaning any relation with time-referenced data.

Next, the term "fully temporal relation" was originally proposed because a bitemporal relation is capable of modeling both the intrinsic and the extrinsic time aspects of facts, thus providing the "full story." However, this term is no longer used.

The term "valid-time and transaction-time relation" is precise and consistent with the other terms, but is also lengthy.

## CROSS REFERENCE*

Temporal Database, Transaction Time, Valid Time

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag,

pp. 367–405, 1998.

# CALENDAR

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

None

## DEFINITION

A *calendar* provides a human interpretation of time. As such, calendars ascribe meaning to temporal values such that the particular meaning or interpretation provided is relevant to its users. In particular, calendars determine the mapping between human-meaningful time values and an underlying time line.

## MAIN TEXT

Calendars are most often cyclic, allowing human-meaningful time values to be expressed succinctly. For example, dates in the common Gregorian calendar may be expressed in the form <*month day*, *year*> where the month and day fields cycle as time passes.

The concept of calendar defined here subsumes commonly used calendars such as the Gregorian calendar, the Hebrew calendar, and the Lunar calendar, though the given definition is much more general. This usage is consistent with the conventional English meaning of the word.

Dershowitz and Reingold's book presents complete algorithms for fourteen prominent calendars: the present civil calendar (Gregorian), the recent ISO commercial calendar, the old civil calendar (Julian), the Coptic an Ethiopic calendars, the Islamic (Muslim) calendar, the modern Persian (solar) calendar, the Bahá'í calendar, the Hebrew (Jewish) calendar, the Mayan calendars, the French Revolutionary calendar, the Chinese calendar, and both the old (mean) and new (true) Hindu (Indian) calendars. One could also envision more specific calendars, such as an academic calendar particular to a school, or a fiscal calendar particular to a company.

## CROSS REFERENCE*

Calendric System, SQL, Temporal Database

## REFERENCES*

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

N. Dershowitz and E. M. Reingold, **Calendrical Calculations**, Cambridge, 1977.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

B. Urgun, C. E. Dyreson, R. T. Snodgrass, J. K. Miller, N. Kline, M. D. Soo, and C. S. Jensen, "Integrating Multiple Calendars using $\tau$Zaman," *Software—Practice and Experience* 37(3):267-308, 2007.

# CALENDRIC SYSTEM

Curtis E. Dyreson, Christian S. Jensen and Richard T. Snodgrass
Utah State University, USA, Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

none

**DEFINITION**

A calendric system is a collection of calendars. The calendars in a calendric system are defined over contiguous and non-overlapping intervals of an underlying time-line. Calendric systems define the human interpretation of time for a particular locale as different calendars may be employed during different intervals.

**MAIN TEXT**

A calendric system is the abstraction of time available at the conceptual and logical (query language) levels. As an example, a Russian calendric system could be constructed by considering the sequence of five different calendars used in that region of the world. In prehistoric epochs, the Geologic calendar and Carbon-14 dating (another form of calendar) are used to measure time. Later, during the Roman empire, the lunar calendar developed by the Roman republic was used. Pope Julius, in the first Century B.C., introduced a solar calendar, the Julian calendar. This calendar was in use until the 1917 Bolshevik revolution when the Gregorian calendar, first introduced by Pope Gregory XIII in 1572, was adopted. In 1929, the Soviets introduced a continuous schedule work week based on four days of work followed by one day of rest, in an attempt to break tradition with the seven-day week. This new calendar, the Communist calendar, had the failing that only eighty percent of the work force was active on any day, and it was abandoned after only two years in favor of the Gregorian calendar, which is still in use today. The term "calendric system" has been used to describe the calculation of events within a single calendar. However, the given definition generalizes that usage to multiple calendars in a very natural way.

**CROSS REFERENCE***

Calendar, Temporal Database, Time Interval

**REFERENCES***

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TITLE

chronon

# BYLINE

Curtis Dyreson
Utah State University
Curtis.Dyreson@usu.edu
http://www.cs.usu.edu/~cdyreson

# SYNONYMS

instant, moment, time quantum, time unit

# DEFINITION

A chronon is the smallest, discrete, non-decomposable unit of time in a temporal data model. In a one-dimensional model, a chronon is a *time interval* or *period*, while in an *n*-dimensional model it is a non-decomposable region in *n*-dimensional time. Important special types of chronons include valid-time, transaction-time, and bitemporal chronons.

# MAIN TEXT

Data models often represent a time line by a sequence of non-decomposable, consecutive time periods of identical duration. These periods are termed chronons. A data model will typically leave the size of each particular chronon unspecified. The size (e.g., one microsecond) will be fixed later by an individual application or by a database management system, within the restrictions posed by the implementation of the data model. The number of chronons is finite in a bounded model (i.e., a model with a minimum and maximum chronon), or countably infinite otherwise. Consecutive chronons may be grouped into larger intervals or segments, termed *granules*; a chronon is a granule at the lowest possible granularity.

# CROSS REFERENCES

Temporal Granularity, Time Domain, Time Instant, Time Interval

# Current Semantics

Michael H. Böhlen, Christian S. Jensen and Richard T. Snodgrass
Free Univerity of Bozen-Bolzano, Italy, Aalborg University, Denmark and
University of Arizona, USA

## SYNONYMS

Temporal upward compatibility

## DEFINITION

Current semantics constrains the semantics of non-temporal statements applied to temporal databases. Specifically, current semantics requires that non-temporal statements on a temporal database behave as if applied to the non-temporal database that is the result of taking the timeslice of the temporal database as of the current time.

## MAIN TEXT

Current semantics [3] requires that queries and views on a temporal database consider the current information only and work exactly as if applied to a non-temporal database. For example, a query to determine who manages the high-salaried employees should consider the current database state only. Constraints and assertions also work exactly as before: they are applied to the current state and checked on database modification.

Database modifications are subject to the same constraint as queries: they should work exactly as if applied to a non-temporal database. Database modifications, however, also have to take into consideration that the current time is constantly moving forward. Therefore, the effects of modifications must persist into the future (until overwritten by a subsequent modification).

The definition of current semantics assumes a timeslice operator $\tau[t](D^t)$ that takes the snapshot of a temporal database $D^t$ at time $t$. The timeslice operator takes the snapshot of all temporal relations in $D^t$ and returns the set of resulting non-temporal relations.

Let *now* be the current time [2] and let $t$ be a time point that does not exceed *now*. Let $D^t$ be a temporal database instance at time $t$. Let $M_1, \ldots, M_n$, $n \geq 0$ be a sequence of non-temporal database modifications.

Let $Q$ be a non-temporal query. Current semantics requires that for all $Q$, $t$, $D^t$, and $M_1, \ldots, M_n$ the following equivalence holds:

$$Q(M_n(M_{n-1}(\ldots(M_1(D^t)\ldots)))) = Q(M_n(M_{n-1}(\ldots(M_1(\tau[now](D^t)))\ldots)))$$

Note that for $n = 0$ there are no modifications, and the equivalence becomes $Q(D^t) = Q(\tau[now](D^t))$, i.e., a non-temporal query applied to a temporal database must consider the current database state only.

An unfortunate ramification of the above equivalence is that temporal query languages that introduce new reserved keywords not used in the non-temporal languages they extend will violate current semantics. The reason is that the user may have previously used such a keyword as an identifier (e.g., a table name) in the database. To avoid being overly restrictive, it is reasonable to consider current semantics satisfied even when reserved words are added, as long as the semantics of all statements that do not use the new reserved words is retained by the temporal query language.

Temporal upward compatibility [1] is a synonym that focuses on settings where the original temporal database is the result of rendering a non-temporal database temporal.

## CROSS REFERENCE

Nonsequenced Semantics, Now in Temporal Databases, Sequenced Semantics, Snapshot equivalence, Temporal Database, Temporal Data Model, Temporal Query Languages, Timeslice Operator

## REFERENCES

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

[2] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "NOW" in Databases. *ACM Transactions on Database Systems*, 22:171–214, June 1997.

[3] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann, 1999.

# EVENT

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Event relation; Instant relation

**DEFINITION**

An *event* is an instantaneous fact, i.e., something occurring at an instant.

**MAIN TEXT**

Some temporal data models support valid-time relations where tuples represent events and where the tuples are thus timestamped with time values that represent instants. When granules are used as timestamps, an event timestamped with granule $t$ occurs, or is valid, at some instant during $t$.

It may be observed that more general kinds of events have also been considered in the literature, including events with duration and complex and composite events.

**CROSS REFERENCE***

Temporal Database, Temporal Granularity, Time Instant

**REFERENCES***

[1] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

[2] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

[3] Events. The Internet Encyclopedia of Philosophy. http://www.iep.utm.edu/e/events.htm

# FIXED TIME SPAN

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Constant span

**DEFINITION**

A time span is *fixed* if it possesses the special property that its duration is independent of the assumed context.

**MAIN TEXT**

As an example of a fixed span, "one hour" always, assuming a setting without leap seconds, has a duration of 60 minutes. To see that not all spans are fixed, consider "one month," which is a prime example of a variable span in the Gregorian calendar. The duration of this span may be any of 28, 29, 30, and 31 days, depending on the context, i.e., the specific month.

**CROSS REFERENCE***

Calendar, Temporal Database, Time Span, Variable Span

**REFERENCES***

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# FOREVER

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Infinity; Positive Infinity

**DEFINITION**

The distinguished value *forever* is a special valid-time instant following the largest granule on the valid-time domain. Forever is specific to valid time and has no transaction-time semantics.

**MAIN TEXT**

The value *forever* is often used as the valid end time for currently-valid facts. However, this practice is semantically incorrect, as such an end time implies that the facts are true during all future times. This usage occurs because database management systems do not offer means of storing the ever-changing current time, *now*, as an attribute value. To fix the incorrect semantics, the applications that manipulate and query the facts may interpret the semantics specially. However, the better solution is to extend the database management system to support the use of the variable *now* as an attribute value.

Concerning the synonyms, "infinity" and "positive infinity" both appear to be more straightforward, but have conflicting mathematical meanings. In addition, the time domain used for valid time may be finite, making the use of "infinity" inappropriate. Furthermore, the term positive infinity is longer and would imply the use of "negative infinity" for its opposite. Forever is intuitive and does not have conflicting meanings.

**CROSS REFERENCE***

Now in Temporal Databases, Temporal Database, Time Instant, Until Changed, Valid Time

**REFERENCES***

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

K. Torp, C. S. Jensen, and R. T. Snodgrass, "Effective Timestamping in Databases," *International Journal on Very Large Databases*, Vol. 8, Issue 3+4, February 2000, pp. 267–288.

# History in Temporal Databases

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Time Sequence; Time Series; Temporal Value; Temporal Evolution

## DEFINITION

The *history* of an "object" of the real world or of a database is the temporal representation of that object. Depending on the specific object, one can have *attribute histories, entity histories, relationship histories, schema histories, transaction histories*, etc.

## MAIN TEXT

"History" is a general concept, intended in the sense of "train of events connected with a person or thing."
In the realm of temporal databases, the concept of history is intended to include multiple time dimensions as well as data models. Thus one can have, e.g., valid-time histories, transaction-time histories, bitemporal histories, and user-defined time histories. However, multi-dimensional histories can be defined from mono-dimensional ones (e.g., a bitemporal history can be seen as the transaction-time history of a valid-time history).
The term "history," defined formally or informally, has been used in many temporal database papers, also to explain other terms. For instance, salary history, object history, and transaction history are all expressions used in this respect.
Although "history" usually has to do with the past events, its use for the future—as introduced by, e.g., prophecies, science fiction, and scientific forecasts—does not seem to present comprehension difficulties. (The adjective "historical" seems more problematic for some.) Talking about future history requires the same extension of meaning as required by talking about future data.
The synonym "temporal value" appears less general than "history," since it applies when "history" specializes into attribute history (value history), and it suggests a single value rather than a succession of values across time.
The concept of a history is a slightly more general than the concept of a time sequence. Therefore the definition of "history" does not prevent defining "time sequence."
Since "history" in itself implies the idea of time, the use of "history" does not require further qualifications as is needed in the case of "sequence" or "series.". In particular, "history" lends itself well to be used as modifier, even though "time sequence" is an alternative consolidated term.

## CROSS REFERENCE*

Bitemporal Relation, Event, Temporal Database, Temporal Data Models, Temporal Element, Transaction Time, User-Defined Time, Valid Time

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# LIFESPAN

Christian S. Jensen
Aalborg University, Denmark

## SYNONYMS

Existence time; Temporal domain

## DEFINITION

The *lifespan* of a database object is the time during which the corresponding real-world object exists in the modeled reality.

## MAIN TEXT

Some temporal data models, e.g., conceptual models, provide built-in support for the capture of lifespans, while other models do not. It may be observed that lifespans can be reduced to valid times, in the sense that the lifespan of an object $o$ is the valid time of the fact "$o$ exists."

In the general case, the lifespan of an object is an arbitrary subset of the time domain and so is naturally captured by a temporal element timestamp.

The synonym "existence time" is also used for this concept.

## CROSS REFERENCE*

Temporal Conceptual Data Models, Temporal Data Models, Temporal Database, Temporal Element, Time Domain, Time in Philosophical Logic, Valid Time

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# Nonsequenced Semantics

Michael H. Böhlen, Christian S. Jensen and Richard T. Snodgrass
Free Univerity of Bozen-Bolzano, Italy, Aalborg University, Denmark and
University of Arizona, USA

## SYNONYMS

Nontemporal semantics

## DEFINITION

Nonsequenced semantics guarantees that query language statements can reference and manipulate the timestamps that capture the valid and transaction of data in temporal databases as regular attribute values, with no built-in temporal semantics being enforced by the query language.

## MAIN TEXT

A temporal database generalizes a non-temporal database and associates one or more timestamps with database entities. Different communities have suggested temporal query languages that provide advanced support for formulating temporal statements. Results of these efforts include a variety of temporal extensions of SQL, temporal algebras, and temporal logics that simplify the management of temporal data.

Languages with built-in temporal support are attractive because they offer convenient support for formulating a wide range of common temporal statements. The classical example of advanced built-in temporal support is sequenced semantics, which makes it possible to conveniently interpret a temporal database as a sequence of non-temporal databases. To achieve built-in temporal support, the timestamps are viewed as implicit attributes that are given special semantics.

Built-in temporal support, however, may also limit the expressiveness of the language when compared to the original non-temporal language where timestamps are explicit attributes. Nonsequenced semantics guarantees that statements can manipulate timestamps as regular attribute values with no built-in temporal semantics being enforced. This ensures that the expressiveness of the original language is preserved.

The availability of legacy statements with the standard non-temporal semantics is also important in the context of migration where users can be expected to be well-acquainted with the semantics of their non-temporal language. Nonsequenced semantics ensures that users are able to keep using the paradigm they are familiar with and to incrementally adopt the new features. Moreover, from a theoretical perspective, any variant of temporal logic, a well-understood language that only provides built-in temporal semantics, is strictly less expressive than a first order logic language with explicit references to time [Abiteboul 1996, Toman 1996].

Each statement of the original language has the potential to either be evaluated with a temporal or a non-temporal semantics. For example, a count query can count the tuples at each time instant (this would be temporal, i.e., sequenced semantics) or count the tuples actually stored in a relation instance (this would be non-temporal, i.e., nonsequenced semantics).

To distinguish the two semantics, different approaches have been suggested. For instance, TempSQL distinguishes between so-called current and classical users. ATSQL and SQL/Temporal offer so-called statement modifiers that enable the users to choose between the two semantics at the granularity of statements. Below, statement modifiers are used for illustration. Specifically the ATSQL modifier `NSEQ VT` signals standard SQL semantics with full control over the timestamp attributes of a valid-time database.

The illustrations assume a database instance with three relations:

`Employee`

| ID | Name | VTIME |
|---|---|---|
| 1 | Bob | 5−8 |
| 3 | Pam | 1−3 |
| 3 | Pam | 4−12 |
| 4 | Sarah | 1−5 |

`Salary`

| ID | Amt | VTIME |
|---|---|---|
| 1 | 20 | 4−10 |
| 3 | 20 | 6−9 |
| 4 | 20 | 6−9 |

`Bonus`

| ID | Amt | VTIME |
|---|---|---|
| 1 | 20 | 1−6 |
| 1 | 20 | 7−12 |
| 3 | 20 | 1−12 |

and the following queries:

```
NSEQ VT
  SELECT COUNT(*) FROM Bonus;
NSEQ VT
  SELECT E.ID FROM Employee AS E, Salary AS S
  WHERE VTIME(E) PRECEDES VTIME(S) AND E.ID = S.ID;
```

Both queries are nonsequenced, i.e., the valid time is treated as a regular attribute without any special processing going on. The first query determines the number of bonuses that have been paid. It returns the number of tuples in the `Bonus` relation, which is equal to three. Note that if the sequenced modifier was used (cf. sequenced semantics) then the time-varying count had been computed. With the given example, the count at each point in time would be two. The second query joins `Employee` and `Salary`. The join is not performed at each snapshot (cf. sequenced semantics). Instead it requires that the valid time of `Employee` precedes the valid time of `Salary`. The result is a non-temporal table.

Nonsequenced statements offer no built-in temporal support, but instead offer complete control. This is akin to programming in assembly language, where one can do everything, but everything is hard to do. The query language must provide a set of functions and predicates for expressing temporal relationships (e.g., `PRECEDES` [Allen 1983]) and performing manipulations and computations on timestamps (e.g., `VTIME`). The resulting new query-language constructs are relatively easy to implement because they only require changes at the level of built-in predicates and functions. Instead of using functions and predicates on timestamps, the use of temporal logic with temporal connectives has also been suggested.

**CROSS REFERENCE***

Allen's Relation, Sequenced Semantics, SQL-Based Temporal Query Languages, Temporal Database, Valid Time

**REFERENCES***

S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal Versus First-Order Logic to Query Temporal Databases. In *Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems*, pages 49–57, Montreal, Canada, June 1996. ACM Press.

J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal Statement Modifiers. *ACM Transactions on Database Systems*, 25(4):48, December 2000.

D. Toman and D. Niwiński. First-Order Queries over Temporal Databases Inexpressible in Temporal Logic. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology — EDBT'96, $5^{th}$ International Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 307–324. Springer, March 1996.

# Now in Temporal Databases

Curtis E. Dyreson, Christian S. Jensen, and Richard T. Snodgrass
Utah State University, Aalborg University, and University of Arizona

## SYNONYMS

Current time; Current date; Current timestamp; Until changed

## DEFINITION

The word *now* is a noun in the English language that means "at the present time." This notion appears in databases in three guises. The first use of *now* is as a function within queries, views, assertions, etc. For instance, in SQL, `CURRENT_DATE` within queries, etc., returns the current date as an SQL `DATE` value; `CURRENT_TIME` and `CURRENT_TIMESTAMP` are also available. These constructs are nullary functions.

In the context of a transaction that contains more than one occurrence of these functions, the issue of which time value(s) to return when these functions are invoked becomes important. When having these functions return the same (or consistent) value, it becomes a challenge to select this time and to synchronize it with the serialization time of the transaction containing the query.

The second use is as a database variable used extensively in temporal data model proposals, primarily as timestamp values associated with tuples or attribute values in temporal database instances. As an example, within transaction-time databases, the stop time of data that have not been logically deleted and thus are current is termed "until changed." A challenging aspect of supporting this notion of *now* has been to contend with instances that contain this variable when defining the semantics of queries and modification and when supporting queries and updates efficiently, e.g., with the aid of indices.

The third use of *now* is as a database variable with a specified offset (a "now-relative value") that can be stored within an implicit timestamp or as the value of an explicit attribute. Challenges include the specification of precise semantics for database instances that contain these variables and the indexing of such instances.

## HISTORICAL BACKGROUND

Time variables such as *now* are of interest and indeed are quite useful in databases, including databases managed by a conventional DBMS, that record time-varying information, the validity of which often depends on the current-time value. Such databases may be found in many application areas, such as banking, inventory management, and medical and personnel records.

The SQL nullary functions `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` have been present since SQL's precursor, SEQUEL 2 [3]. The transaction stop time of *until changed* has been present since the initial definition of transaction-time databases (e.g., in the early work of Ben-Zvi [1]).

The notion of *now* and the surprisingly subtle concerns with this ostensibly simple concept permeate the literature, and yield quite interesting solutions.

## SCIENTIFIC FUNDAMENTALS

Three notions of "now" in temporal databases are explored in more detail below.

## SQL Nullary Functions

The following example illustrates the uses and limitations of *now*, specifically the SQL `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP`, in conventional databases. Banking applications record when account balances for customers are valid. Consider the relation `AccountBalance` with attributes `AccountNumber`, `Balance`, `FromDate`, and `ToDate`. To determine the balance of account 12345 on January 1, 2007, one could use a simple SQL query.

```
SELECT Balance
FROM AccountBalance
WHERE Account = 12345 AND FromDate <= DATE '2007-01-01' AND DATE '2007-01-01' < ToDate
```

To determine the balance of that account *today*, the nullary function `CURRENT_DATE` is available.

```
SELECT Balance
FROM AccountBalance
WHERE Account = 12345 AND FromDate <= CURRENT_DATE AND CURRENT_DATE < ToDate
```

Interestingly, in the SQL standard the semantics of the function are implementation-dependent which opens it to various interpretations: "If an SQL-statement generally contains more than one reference to one or more <datetime value functions>s then all such references are effectively evaluated simultaneously. The time of evaluation of the <datetime value function> during the execution of the SQL-statement is implementation-dependent." (This is from the SQL:1999 standard.) So an implementation is afforded considerable freedom in choosing a definition of "current," including perhaps when the statement was presented to the system, or perhaps when the database was first defined.

Transactions take time to complete. If a transaction needs to insert or modify many tuples, it may take minutes. However, from the point of view of the user, the transaction is atomic (all or nothing) and serializable (placed in a total ordering). Ideally "current" should mean "when the transaction executed," that is, the instantaneous time the transaction logically executed, consistent with the serialization order.

Say a customer opens an account and deposits $200. This transaction results in a tuple being inserted into the `AccountBalance` table. The transaction started on January 14 at 11:49 p.m. and committed at 12:07 a.m. (that is, starting before midnight and completing afterwards). The tuple was inserted on 11:52 p.m. Another user also created a second account with an initial balance of $500 in a transaction that started on January 14 at 11:51 p.m. and committed on 11:59 p.m., inserting a tuple into the `AccountBalance` relation at 11:52 p.m. If the system uses the transaction start time, the following two tuples will be in the relation.

| AccountNumber | Balance | FromDate | ToDate |
|---|---|---|---|
| 121345 | 200 | 2007-01-14 | ... |
| 543121 | 500 | 2007-01-14 | ... |

According to these two tuples, the sum of balances on January 14 is $700. But note that though both transactions began on January 14, only the second transaction committed by midnight (also note that the second transaction is earlier in the serialization order since it commits first). Hence, the actual aggregate balance on January 14 was never $700: the balance started at $0 (assuming there were initially no other tuples), then changed to $500. Only on January 15 did it increase to $700.

Suppose that instead of using the time at which the entire transaction began, the time of the actual insert statement (e.g., for the first transaction, January 14, 11:52 p.m.) is used; then the same problem occurs.

What is desired is for a time returned by `CURRENT_DATE` to be consistent with the serialization order and with the commit time, which unfortunately is not known when `CURRENT_DATE` is executed. Lomet et al. [7] showed how to utilize the lock manager to assign a commit time in such a way that it was consistent with the serialization order as well as with dates assigned as values to prior `CURRENT_DATE` invocations. Specifically, each use of `CURRENT_DATE`, etc. defines or narrows a *possible period* during which the commit time of the transaction must reside. For

CURRENT_DATE this period is the particular twenty-four hours of the day returned by this function. Read-write conflicts with other transactions further narrow the possible period. For example, if a particular transaction reads values written by another transaction, the transaction time of the reader must be later than that of the writer. At commit time, it is attractive to assign the earliest instant in the possible period to the transaction. Alternatively, if the possible period ever becomes empty, the transaction must be aborted. Lomet et al. outline important optimizations to render such calculations efficient.

## Now in End or Stop Columns

While the rationale for using *now* as the start time of a tuple (e.g., in the FromDate column for the two example tuples in table displayed earlier) is not clear, so is the use of *now* as the end time of a tuple (e.g., in the ToDate column). The validity of a tuple then starts when a deposit is made and extends until the current time, assuming no update transactions are committed. Thus, on January 16, the balance is valid from January 14 until January 16; on January 17, the balance is valid from January 14 until January 17; etc.

It is impractical to update the database each day (or millisecond) to correctly reflect the valid time of the balance. A more promising approach is to store a variable, such as *now*, in the ToDate field of a tuple, to indicate that the time when a balance is valid depends on the current time. In the example, it would be recorded on January 14 that the customer's balance of $200 is valid from January 14 through *now*. The CURRENT_DATE construct used in the above query cannot be stored in a column of an SQL table. All major commercial DBMSs have similar constructs, and impose this same restriction. The user is forced instead to store a specific time, which is cumbersome and inaccurate. (Clifford et al. explain these difficulties in great detail [4]).

The solution is to allow one or more free, current-time variables to be stored in the database. Chief among these current-time variables is "*now*" (e.g., [5]), but a variety of other symbols have been used, including "–" [1], "∞" [11], "@" [8], and "*until-changed*" [14]. Such stored variables have advantages at both the semantic and implementation levels. They are expressive and space efficient and avoid the need for updates at every moment in time.

As an example, consider a variable database with the tuple $\langle$*Jane*, *Assistant*, [*June 1*, *now*]$\rangle$, with *now* being a variable. The query "List the faculty on June 15," evaluated on June 27, results in $\{\langle$*Jane*, *Assistant*$\rangle\}$.

## Now-Relative Values

A now-relative instant generalizes and adds flexibility to the variable *now* by allowing an offset from this variable to be specified. Now-relative instants can be used to more accurately record the knowledge of Jane's employment. For example, it may be that hiring changes are recorded in the database only three days after they take effect. Assuming that Jane was hired on June 1, the definite knowledge of her employment is accurately captured in the tuple $\langle$*Jane*, *Assistant*, [*June 1*, *now* − 3 *days*]$\rangle$. This tuple states that Jane was an Assistant Professor from June 1 and until three days ago, but it contains no information about her employment as of, e.g., yesterday.

A now-relative instant thus includes a displacement, which is a signed duration, also termed a span, from *now*. In the example given above, the displacement is minus three days. Now-relative variables can be extended to be indeterminate [4], as can regular instants and the variable *now*.

The semantics of now variables has been formalized with an *extensionalization* that maps from a *variable database level* containing variables as values to an *extensional database level*. The extensional database exhibits three key differences when compared to the variable database level. First, no variables are allowed—the extensional level is fully ground. Second, timestamps are instants rather than intervals. Third, an extensional tuple has one additional temporal attribute, called a *reference time attribute*, which may be thought of as representing the time at which a meaning was given to the temporal variables in the original tuple. Whereas the variable-database level offers a convenient representation that end-users can understand and that is amenable to implementation, the mathematical simplicity of the extensional level supports a rigorous treatment of temporal databases in terms of first order logic.

When a variable database is queried, an additional problem surfaces: what to do when a variable is encountered during query evaluation. When evaluating a user-level query, e.g., written in some dialect of SQL, it is common to transform it into an internal algebraic form that is suitable for subsequent rule or cost-based query optimization. As the query processor and optimizer are among the most complex components of a database management system, it is attractive if the added functionality of current-time-related timestamps necessitates only minimal changes to

these components.
Perhaps the simplest approach to supporting querying is that, when a timestamp that contains a variable is used during query processing (e.g., in a test for overlap with another timestamp), a ground version of that timestamp is created and used instead. With this approach, only a new component that substitutes variable timestamps with ground timestamps has to be added, while existing components remain unchanged.
Put differently, a *bind* operator can be added to the set of operators already present. This operator is then utilized when user-level queries are mapped to the internal representation. The operator accepts any tuple with variables. It substitutes a ground value for each variable and thus returns a ground (but still variable-level) tuple. Intuitively, the *bind* operator sets the perspective of the observer, i.e., it sets the reference time. Existing query languages generally assume that the temporal perspective of an observer querying a database is the time when the observer initiates the query.
Torp et al. has shown how to implement such variables within a database system [13].
With *now* as a database variable, the temporal extent of a tuple becomes a non-constant function of time. As most indices assume that the extents being indexed are constant in-between updates, the indexing of the temporal extents of now-relative data poses new challenges.
For now-relative transaction-time data, one may index all data that is not now-relative (i.e., has a fixed end time) in one index, e.g., an R-tree, and all data that is now-relative (i.e., the end time is *now*) in another index where only the start time is indexed.
For bitemporal data, this approach can be generalized to one where tuples are distributed among four R-trees. The idea is again to overcome the inabilities of indices to cope with continuously evolving regions by applying transformations to the growing bitemporal extents that render them stationary and thus amenable to indexing. Growing regions come in three shapes, each with its own transformation. These transformations are accompanied by matching query transformations [2].
In another approach, the R-tree is extended to store *now* for both valid and transaction time in the index. The resulting index, termed the GR-tree, thus accommodates bitemporal regions and uses minimum bounding regions that can be either static or growing and either rectangles or stair shapes. This approach has been extended to accommodate bitemporal data that also have spatial extents [10].

## KEY APPLICATIONS*

The notion of *now* as a nullary function representing the current time is common in database applications. For relational database management, SQL offers `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` functions that return an appropriate SQL time value. In other kinds of database management systems, such as native XML database management systems, similar constructs can be found. For example, XQuery has a `fn:current-time()` function that returns a value of XML Schema's `xs:time` type. XQuery also has a `fn:current-date()` function. Less common in existing database applications are the other notions of now: as a variable stored in a database to represent the ever-changing current time or as a time related to, but displaced from, the current time.

## FUTURE DIRECTIONS

The convenience of using now variables poses challenges to the designers of database query languages. The user-defined time types available in SQL-92 can be extended to store now-relative variables as values in columns. The TSQL2 language [12] does so, and also supports those variables for valid and transaction time. In TSQL2 the "bind" operation is implicit; `NOBIND` is provided to store variables in the database. But such variables have yet to be supported by commercial DBMSs. It may also be expected that at least one of the three uses of now will re-emerge as part of a temporal extension of XQuery or a language associated with the Semantic Web.
The impact of stored variables on database storage structures and access methods is a relatively unexplored area. Such stored variables may present optimization opportunities. For example, if the optimizer knows (through attribute statistics) that a large proportion of tuples has a "to" time of *now*, it may then decide that a sort-merge temporal join will be less effective. Finally, new kinds of variables, such as *here* for spatial and spatio-temporal databases, might offer an interesting extension of the framework discussed here.

## CROSS REFERENCE*

Bitemporal Indexing, Supporting Transaction-Time Databases, Temporal Concepts in Philosophy, Temporal Query Languages, Temporal Strata, Temporal Support in XML, Time Period, Transaction Time, TSQL2, Valid Time

## RECOMMENDED READING


[1] J. Ben-Zvi, **The Time Relational Model**. Ph.D. Dissertation, University of California at Los Angeles, 1982.

[2] R. Bliujūtė, C. S. Jensen, S. Šaltenis, and G. Slivinskas, Light-Weight Indexing of Bitemporal Data. In *Proceedings of the Twelfth International Conference on Scientific and Statistical Database Management*, (Berlin, Germany, July), 125–138, 2000.

[3] D. D. Chamberlin, M. M. Astraham, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM Journal of Research and Development* 20(6):560–575, 1976.

[4] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On The Semantics of "Now" *ACM Transactions on Database Systems* 22(2):171–214, June 1997.

[5] J. Clifford and A. U. Tansel, On an algebra for historical relational databases: Two views. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (Austin, TX, May), S. Navathe, editor, 247–265, 1985.

[6] M. Finger, M., Handling Database Updates in Two-dimensional Temporal Logic. *Journal of Applied Non-Classical Logics 2*, 2, 201–224, 1992.

[7] D. Lomet, R. T. Snodgrass, and C. S. Jensen, Exploiting the Lock Manager for Timestamping. In *Proceedings of the Ninth International Database Engineering and Applications Symposium*, Montreal, Canada, July 2005.

[8] N. A. Lorentzos and R. G. Johnson, Extending Relational Algebra to Manipulate Temporal Data. *Information Systems* 13(3):289–296, 1988.

[9] R. Montague, **Formal Philosophy: Selected Papers of Richard Montague**. Yale University Press, New Haven, 1974.

[10] S. Šaltenis, and C. S. Jensen, Indexing of Now-Relative Spatio-Bitemporal Data. *The VLDB Journal* 11(1):1–16, August, 2002.

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

[12] R. T. Snodgrass (editor), **The TSQL2 Temporal Query Language**. Kluwer Academic Publishers, 674+xxiv pages, 1995.

[13] Kristian Torp, Christian S. Jensen and Richard T. Snodgrass, "Modification Semantics in Now-Relative Databases," *Information Systems* 29(78):653–683, December 2004.

[14] G. Wiederhold, S. Jajodia, and W. Litwin, Integrating Temporal Data in a Heterogeneous Environment. Chapter 22 of **Temporal Databases: Theory, Design, and Implementation**. A. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass (editors), Benjamin/Cummings, 1993, pp. 563–579, 1993.

# PERIOD-STAMPED TEMPORAL MODELS

Nikos A. Lorentzos
Informatics Laboratory
Science Department
Agricultural University of Athens
Iera Odos 75, 11855 Athens, Greece
http://www.aua.gr/~lorentzos
lorentzos@aua.gr

## SYNONYMS

Interval-based Temporal Models

## DEFINITION

A period-stamped temporal model is a temporal data model for the management of data in which time has the form of a time period.

## HISTORICAL BACKGROUND

There are applications that require the recording and management not only of data but also of the time during which this data is valid (valid time). A typical example is the data maintained by pension and life insurance organizations in order to determine the benefits for which a person qualifies. Similarly, such organizations have to record their financial obligations at various times in the future.

There are also sensitive applications, in which it is important to record not only the data but also the time at which this data was either recorded in the database or deleted or updated (transaction time). This time is recorded automatically by the Database Management System (DBMS) and it cannot be modified by the user.

Finally, there are applications in which both data, valid time, and transaction time need be recorded.

A data model for the management of data and also of either valid time or transaction time is a temporal data model. If time has the form of a *time period* the model is a *period-stamped data model*.

In the case of the relational model, data and time are recorded in a relation. A relation to record data and valid time is a valid time relation. A relation to record data and transaction time is a transaction time relation. Finally, a relation to record data, valid time and transaction time is a bitermporal relation.

In spite of the interest for the management of temporal data, there are many practical problems, which cannot be supported directly by the use of a conventional DBMS. As a consequence, much programming is required in order to support such applications. Hence, the satisfaction of actual user requirements necessitated the definition of a period-stamped temporal model. The bulk of research, on the definition of such a model, appeared in the 80's, at a time when computers became powerful enough to process large volumes of data.

## SCIENTIFIC FUNDAMENTALS

Problems related to the management of temporal data are depicted next. The illustration is restricted to the relational model, in particular to the management of valid time relations. Note, however, that relevant problems can be identified in the management of transaction time relations whereas the management of bitermporal relations is much more complicated.

**Examples Illustrating Inadequacy of Conventional Models to Support Time Period**

Figure 1 shows SALARY, a valid time relation that records salary histories. The notation 'd_number', for values recorded in attributes Begin and End, represents a point in time, which is of a date-time data type. For example, d100 could represent "January 1, 2007". The interpretation of the first tuple is that Alex earned 100 on each of the dates in the time period [d100, d299], i.e. on each of the dates d100, d101, …, d299. Then the amount earned was 150 for each of the dates in [d300, d499].

**SALARY**

| Name | Amount | Begin | End |
|---|---|---|---|
| Alex | 100 | d100 | d299 |
| Alex | 150 | d300 | d499 |
| Alex | 150 | d800 | d999 |
| John | 100 | d200 | d899 |

**Figure 1** Example of a valid time relation.

Contrary to the simple representation of a valid time relation, a conventional DBMS lacks the functionality required for the management of such relations. This is illustrated below by a number of examples.

***Data Insertion*:** If the tuple (Alex, 150, d400, d799) is inserted into SALARY, it will be recorded in addition to the other tuples. In this case, however, SALARY will contain duplicate data for Alex's payment on the dates d400, d401, …, d499, since this data is already recorded in the second tuple of the relation. Moreover, the query, to return *the time at which Alex's salary became 150*, if not formulated carefully, will return three dates, d300, d400 and d800 whereas the correct answer should be d300. To avoid such problems, it would be appropriate for the newly inserted tuple to be combined with the second and third tuple for Alex into a single one, (Alex, 150, d300, d999), i.e. tuples with identical data for attributes Name and Amount (value equivalence), which have either *overlapping* or *adjacent* time periods, to coalesce into a single time period (temporal *coalescing*).

***Key*:** Declaring the primary key of SALARY, to be a multi-attribute key of Name and Amount, will not disallow the recording of the tuple (Alex, 200, d400, d799). As a result, SALARY will contain conflicting data, since its second tuple will be showing that Alex's payment is 150 for each of the dates d400, d401, …, d499 whereas, from the newly inserted tuple, it will also be shown that, for these dates, this payment is 200. Hence, in the case of a conventional DBMS, special integrity constraints have to be defined for every relation like SALARY.

***Data Deletion*:** Assuming that John's payment for the dates d400, d401, …, d799 was recorded by mistake and has to be deleted, the last tuple of SALARY has to be replaced by two tuples, (John, 100, d200, d399) and (John, 100, d800, d899).

***Data Update*:** Assuming that John's payment for dates d400, d401, …, d799 has to be corrected to be 150, the last tuple of SALARY has to be replaced by three tuples, (John, 100, d200, d399), (John, 150, d400, d799) and (John, 100, d800, d899).

***Data Projection*:** The query '*for every employee, list the time during which he is paid*', requires a projection on attributes Name, Begin, End. This will generate four tuples. In practice, however, a temporal coalescing of the time periods recorded in SALARY is also required in order to obtain the correct result, consisting of only three tuples, (Alex, d100, d499), (Alex, d800, d999) and (John, d200, d899).

Since a conventional DBMS lacks the functionality illustrated above, special code has to be written to handle correctly all the cases described. This limitation gave rise to research on the definition of a temporal data model. Many of the research efforts led to proposals for the definition of various period-stamped data models.

**Common Characteristics of Period-Stamped Data Models**

The majority of researchers adopted the principle that a temporal data model should be a minimal extension of the conventional relational data model. The most common characteristics of these approaches are outlined below.

A period-stamped, valid time relation always has two types of attributes: The first type consists of one or more ordinary attributes of a conventional relation. These attributes are termed *explicit* by some authors. Valid time itself is not considered to be data, it is considered as being *orthogonal* to data. Hence, the second type consists of special attributes to record valid time. These attributes are often termed *implicit* and can be system-assigned by default. Some approaches consider one pair of such attributes, to record the Begin and End of valid time periods. The valid time recorded in them is usually assumed to represent a time period of the form [Begin, End], i.e. an interval closed on both sides. It is said that the data recorded in the explicit attributes is *stamped* by the time recorded in the implicit attributes.

According to the concepts discussed above, the explicit attributes of the relation in Figure 1 are Name and Amount and the implicit attributes are Begin and End. The first tuple of the relation is (Alex, 100) and it is *stamped* by the time period [d100, d299].

Some approaches define a time period data type; therefore they consider only one implicit attribute. In such approaches, the equivalent scheme for the relation in Figure 1 is SALARY(Name, Amount, Period).

Since valid time is not considered to be data, it cannot be part of the primary key of a relation. Hence, Name is designated as the primary key of SALARY, i.e. it matches the primary key of an ordinary relation.

In all the approaches, valid time is considered to be discrete. Hence, the time period [d100, d299] is interpreted as consisting of the dates d100, d101, …, d299. Various temporal granularities can be declared by the user, depending on the application.

Temporal Algebras, Temporal Query Languages or Temporal Calculi are proposed in various approaches. The functionality of the operations of the conventional relational model is revised accordingly, so as to overcome the problems illustrated earlier. Appropriate predicates are also defined, applicable to time periods, which can be used in selection operations. In some approaches additional operations are defined to capture temporal functionality that cannot be achieved by simply extending existing languages. In general, an implicit temporal coalescing takes place in all the temporal operations introduced.

Most of the proposed models in the literature support the valid time property over past, present and future time periods.

**Desired Behavior of Period-Stamped Data Models**

Given the above limitations, new problems have to be faced by the proposed period-stamped models. The most interesting of them are illustrated next by making use of relation SALARY in Figure 1.

***Data Projection*:** The query '*list all the employees ever paid*' requires a projection of SALARY on Name. Such an operation normally yields a relation R with tuples (Alex), (John). Given however that R lacks implicit attributes, it gives rise to the question whether it is an appropriate response. To overcome this problem, in some approaches projection is defined to yield, after a temporal coalescing, the rows (Alex, d100, d499), (Alex, d800, d999) and (John, d200, d899). Notice however that, in this case, the result matches exactly that of another query, '*for every employee, list the time during which he is paid*'.

***Stamp Projection*:** Since time is not data, some special operation, to project only on time, may be necessary. For example, the answer to the query '*give the time during which at least one employee was paid*' should consist of one row, (d100, d999). Given however that this result lacks explicit attributes, it is necessary to determine whether it is appropriate to allow time stamp projection without explicit attributes. Similar questions arise in the case of projections of only one of the two implicit attributes.

**ASSIGNMENT**

| Name | Department | Begin | End |
|---|---|---|---|
| Alex | Shoe | d100 | d199 |
| Alex | Food | d300 | d800 |
| John | Food | d200 | d899 |

**Figure 2** Another valid time relation.

**R**

| Name | Amount | Department | Begin | End |
|---|---|---|---|---|
| Alex | 100 | Shoe | d100 | d199 |
| Alex | 150 | Food | d300 | d499 |
| Alex | 150 | Food | d800 | d800 |
| John | 100 | Food | d200 | d899 |

**Figure 3** Result of a temporal join operation.

***Association of data in distinct relations*:** Such an association requires the use of the Cartesian product operation. For an illustration, consider also the relation in Figure 2, which is used to record the history of employee assignments to departments. Then a Cartesian product of SALARY with ASSIGNMENT seems to yield a relation with two pairs of Begin and End time stamps, one pair from each relation. Clearly this is not a correct period-stamped relation. The same is also true for a join operation. As a consequence, some researchers avoid defining Cartesian product and restrict to the definition of a temporal join operation. This operation yields relation R, in Figure 3, with a single pair of implicit attributes.

**Literature Overview**

The characteristics of individual approaches are outlined below. Unless otherwise specified, a point-stamped valid time relation, in the approach under discussion, is that of Figure 1.

Jones and Mason [1] define LEGOL, a language for the management of period-stamped, valid time relations. It is an early, yet incomplete piece of work.

Ben-Zvi defines a model for the management of period-stamped, bitermporal relations ([2], Chapter 8). One more attribute is used, to record the time at which a tuple is deleted. Sets of time periods are also considered, consisting of mutually disjoint periods. Some SQL extension is provided, too. It is also an early and incomplete piece of work.

Snodgrass defines a QUEL extension, for the management of valid time, transaction time and bitemporal relations [3]. Data may alternatively be stamped by time points (point-stamped temporal models). Time may not be specified for the primary key. Formal semantics are provided. A tuple calculus and a relational algebra are defined in [2], Chapter 6.

Navathe and Ahmed propose an SQL extension for period-stamped valid time relations ([2], Chapter 4, [4, 5]). In this approach, the primary key of the relation in Figure 1 is <Name, Begin>. The definition of the primary key of a relation, in conjunction with the definition of a *Time Normal Form*, enables temporal coalescing. One variation of the select operation takes as argument a time period value. A *moving window* operation enables a temporary *split* of the time stamps of all the tuples into time periods of a fixed size, and the subsequent application of aggregate functions on the data that are associated with these fixed size periods.

Lorentzos and Johnson define a relational algebra for the management of either period-stamped or point-stamped valid time data [6 7]. It is also shown that there are practical considerations for having relations with more than one valid time. A period data type is later defined in [2], Chapter 3. A generic period data type is defined in [2], Chapter 3, and in [8], enabling the use of periods of numbers and strings. Such periods can be used for the management of non-valid time relations which, however, require a functionality identical with that of period-stamped, valid time relations. The operations of the conventional relational model are not revised but two new algebraic operations are defined. Time is treated as data. As such, it may participate in the primary key [8]. In [2], Chapter 3, and in [8], it is shown that there are applications in which a temporal coalescing should be disallowed. An SQL extension is also defined in [8].

Sarda defines a relational algebra in [9] and an SQL extension for valid time relations in [2], Chapter 5, and in [10]. A time period data type is also defined in [9]. Data may alternatively be stamped by time points. Time may not be specified for the primary key. The operations of the conventional relational model are not revised but, as reported in [2], if time is projected out, the result relation is not a correct valid time relation. Similarly, the result returned by the Cartesian product operation is considered not to be a correct period-stamped relation. Two additional relational algebra operations are also defined, functionally equivalent with those defined by Lorentzos.

TSQL2 is a consensus temporal SQL2 extension for the management of either period or point-stamped valid time relations, transaction time relations and bitermporal relations. It is complemented by the definition of a relational algebra.

**EMPLOYEE**

| Name | Salary | Department |
|---|---|---|
| Alex | {<[d100, d300), 100><br><[d300, d500), 150><br><[d800, d1000), 150>} | {<[d100, d200), Shoe><br><[d300, now], Food>} |
| John | {<[d200, d900), 100>} | {<[d200, d900), Food>} |

**Figure 4** A relation in Tansel's approach

All the previous approaches incorporate time at the tuple level (*tuple time-stamping*). Contrary to these, Tansel incorporates time at the attribute level (*attribute time-stamping*) [11]. Hence, to record the history of salaries and of assignment to departments, it suffices to consider a relation like that shown in Figure 4, in place of the two distinct relations in Figures 1 and 2. The relation consists of two tuples, one for Alex and another for John. Time periods are open for the End time. One exception is the case where the end point of a time period equals *now*, in which case the period is closed (see for example Figure 4, the assignment of Alex to the Food department). Therefore, data valid in the future is not supported.

The approach considers four types of attributes, *atomic* (Name, in Figure 4), *set valued* (e.g. to record data such as {Alex, Tom}), *triplet valued* (e.g. to record data such as <[d100, d300), 100>) and *set triplet valued* (Salary and Department, in Figure 4). The operations of the relevant conventional model are revised accordingly. Four additional operations are defined, which enable transformations between relations with different types of attributes. A QUEL extension is defined in [12]. The approach in [11] is extended in [2], Chapter 7, and in [13], to a nested model, incorporating the well-known nest and unnest operations. A Calculus and Algebra are also defined. The approach in [13] is extended in [14], in order to support nested bitemporal relations, in which valid and transaction time are incorporated at the attribute level.

## KEY APPLICATIONS

There are many applications that necessitate the management of period-stamped and, more generally, temporal data. Some examples are the following.

Period-stamped, valid time relations can be used by pension and life insurance organizations, in order to determine the benefits a person qualifies for. Such relations are also needed for these organizations to record their financial obligations at various times in the future.

Similarly, period-stamped, transaction time relations can be used in sensitive applications, to enable tracing the content of the database and evaluate some decision taken in the past, with respect to the content of the database at that time.

## FUTURE DIRECTIONS

Given the practical interest of period-stamped data, and given the many differences between different approaches, it will be useful to agree on and develop a standard model by an international organization, such as ISO.

Further research is necessary for the management of period-stamped geographical data and for the definition of a nested period-stamped data model.

## CROSS REFERENCES

Temporal Database, Temporal data models, Temporal Logical Models, Point-stamped Temporal Models, Temporal Algebras, Temporal Query Languages, Temporal Object-Oriented Databases, Supporting Transaction Time

**RECOMMENDED READING**

1. Jones S. and Mason P. S. (1980): Handling the Time Dimension in a Database. S. M. Deen and P. Hammersley (eds.) Int. Conf. on Data Bases. British Computer Society 1980: 65-83.

2. Tansel A., Clifford J., Gadia J., Segev A., Snodgrass R. (eds.) (1993): Temporal Databases: Theory, Design and Implementation. Benjamin/Cummings 1993.

3. Snodgrass S. (1987): The Temporal Query Language TQUEL. ACM Trans. Database Systems 12(2): 247-298.

4. Navathe S. B. and Ahmed R. (1987): TSQL: A Language Interface for History Databases. Conf. Temporal Aspects in Information Systems, Sophia-Antipolis, France, 13-15 May 1987 (in Temporal Aspects in Information Systems, C. Rolland, F. Bodart, M. Leonard (eds), North-Holland), 1988: 109-122.

5. Navathe S. B. and Ahmed R. (1989): A Temporal Relational Model and a Query Language. Information Sciences, an International Journal. 47(2): 147-175.

6. Lorentzos N. A. and Johnson R. G. (1987): TRA A Model for a Temporal Relational Algebra. Conf. Temporal Aspects in Information Systems, Sophia-Antipolis, France, 13-15 May 1987 (in Temporal Aspects in Information Systems, C. Rolland, F. Bodart, M. Leonard (eds), North-Holland), 1988: 203-215.

7. Lorentzos N. A. and Johnson R. G. (1988): Extending Relational Algebra to Manipulate Temporal Data. Information Systems 13(3): 289-296.

8. Lorentzos N. A, and Mitsopoulos Y. G. (1997): SQL Extension for Interval Data. IEEE Transactions on Knowledge and Data Engineering 9(3): 480-499.

9. Sarda N. L. (1990): Algebra and Query Language for a Historical Data Model. Computer J. 33(1): 11-18.

10. Sarda N. L. (1990): Extensions to SQL for Historical Databases. IEEE Trans. Knowledge and Data Engineering 2(2): 220-230.

11. Tansel A. U. (1986): Adding Time Dimension to Relational Model and Extending Relational Algebra. Information Systems 11(4): 343-355.

12. Tansel A. U. (1991): A Historical query language. Inf. Sci. 53(1-2): 101-133.

13 Tansel A. U. (1997): Temporal Relational Data Model. IEEE Trans. Knowl. Data Eng. 9(3): 464-479.

14. Tansel A. U. (2006): Canan Eren Atay: Nested Bitemporal Relational Algebra. ISCIS 2006: 622-633.

# TITLE

physical clock

# BYLINE

Curtis Dyreson
Utah State University
Curtis.Dyreson@usu.edu
http://www.cs.usu.edu/~cdyreson

# SYNONYMS

Clock

# DEFINITION

A *physical clock* is a physical process coupled with a method of measuring that process to record the passage of time. For instance, the rotation of the Earth measured in *solar days* is a physical clock. Most physical clocks are based on cyclic processes (such as a celestial rotation). One or more physical clocks are used to establish a *time-line clock* for a temporal database.

# MAIN TEXT

Every concrete time is a measurement of some physical clock. For instance a wind-up watch provides the time "now" by measuring the rate at which a coiled, wound spring unwinds. A physical clock is limited by the durability and regularity of the underlying physical process that it measures, so to provide a clock for every instant in a time-line for a temporal database several physical clocks might be needed. For instance a clock based on the rotation of the Earth will be (likely be) limited as current models predict the Earth will eventually be gravitationally drawn into the Sun. The modifier "physical" distinguishes this kind of clock from other kinds of clocks in an application, in particular a "time-line" clock. Atomic clocks, which are a kind of physical clock, provide the measurements for International Atomic Time (TAI).

# CROSS REFERENCES

Chronon, Time-line Clock, Time Instant

# REFERENCES

J. T. Fraser. Time: The Familiar Stranger. The University of Massachusetts Press, 1987. 408 pages.

Curtis E. Dyreson and Richard T. Snodgrass. The Baseline Clock. *The TSQL2 Temporal Query Language*, Kluwer, 1995: 73-92.

# Point-stamped Temporal Models

David Toman, University of Waterloo, Canada, http://www.cs.uwaterloo.ca/~david

**SYNONYMS**

Point Based Temporal Models

**DEFINITION**

Point-stamped temporal data models associate database objects with time instants, *indivisible* elements drawn from an underlying time domain. Such an association usually indicates that the information represented by the database object in question is *valid* (i.e., believed to be true by the database) at that particular time instant. The time instant is then called the *timestamp* of the object.

**HISTORICAL BACKGROUND**

Associating time-dependent data with time instants has been used in sciences, in particular in physics, at least since Isaac Newton's development of classical mechanics: time is commonly modeled as a two-way infinite and continuous linear order of indivisible time instants (often with distance defined as well). Similarly, in many areas of computer science, ranging from protocol specifications to program verification to model checking, discrete point-based timestamps play an essential role. In database systems (and in AI), however, the requirement of *finite* and compact representation has often lead to the use of more complex timestamps, such as intervals. This is particularly common when information about *durations* is stored in a database (in AI, such timestamps are called *fluents*). However, Chomicki [3] has shown that many of these approaches are simply compact representations of large and potentially infinite sets of time instants associated with a particular fact and that, at the conceptual level, the underlying information is often better understood as being timestamped by time instants.

**SCIENTIFIC FUNDAMENTALS**

Temporal data models are typically defined as extensions of standard non-temporal data models that provide means for representation and manipulation of time-dependent data. Temporal extensions of the relational model accomplish this by relativising *truth*, i.e., the sets of facts the database believes to be true in the modeled reality, with respect to elements of the time domain. These elements are then called the timestamps of the facts and, intuitively, specify *at which times* the associated facts are true. In the case of point-stamped temporal models, these elements are indivisible time instants drawn from an appropriate time domain.

The main ideas are outlined in a setting in which the time domain is an unbounded countably infinite linear order of *time instants*. Moreover, while the main focus is on *temporal extensions of the relational model*, most of the issues arise in other data models as well, and admit similar solutions.

## Timestamps and Database Objects

The choice of timestamps is orthogonal to other choices that define the flavor and properties of the resulting temporal data model, in particular

1. to the decision of *what database objects the timestamps are attached to*, and
2. to the decision of *how many timestamps* (per such object) are used.

First to explore is the choice of objects to be timestamped. Intuitively, timestamps should only be attached to objects that actually *capture* information in the underlying data model, indicating that the particular (piece of) information is valid for that timestamp. In the relational model these are *tuples* and their membership in *relations*. Thus, the two principal choices are as follows:

**The Snapshot Model.** Temporal databases in the *snapshot model* are formally defined as mappings from the time domain $T$ to (the class of) standard relational databases with a particular fixed schema $\rho$. In the case

| Year | Database Snapshot (as a set of facts) |
|---|---|
| ... | |
| 1997 | {**Degree**(John,BS,MIT) } |
| 1998 | { } |
| 1999 | {**Degree**(John,MS,UofT) } |
| 2000 | { } |
| 2001 | {**WorksFor**(John,IBM) } |
| 2002 | {**WorksFor**(John,IBM) } |
| 2003 | { } |
| 2004 | {**Degree**(John,PhD,UW) , **WorksFor**(John,Microsoft) } |
| 2005 | {**WorksFor**(John,Microsoft) } |
| ... | {**WorksFor**(John,Microsoft) } |

| **WorksFor** | | |
|---|---|---|
| Year | Name | Company |
| 2001 | John | IBM |
| 2002 | John | IBM |
| 2004 | John | Microsoft |
| 2005 | John | Microsoft |
| ... | John | Microsoft |

| **Degree** | | | |
|---|---|---|---|
| Year | Name | Degree | School |
| 1997 | John | BS | MIT |
| 1999 | John | MS | UofT |
| 2004 | John | PhD | UW |

Figure 1: Instances of matching Snapshot and Timestamp Temporal Database.

of a linearly ordered time domain, such a temporal database can be viewed as a time-indexed sequence of standard relational databases, commonly called *snapshots*. Note that such a sequence is not necessarily discrete or finite: that depends on the choice of the underlying time domain. Intuitively, to capture the fact that, in a snapshot temporal database $D$, a relationship $r$ holds among uninterpreted constants $a_1, \ldots, a_k$ at time $t$ it must be the case that $r(a_1, \ldots, a_k)$ holds in $D(t)$, where $D(t)$ is a standard relational instance, the snapshot of $D$ at time $t$; this statement is denoted by writing $r^{D(t)}(a_1, \ldots, a_k)$.

These structures, in the area of *modal logics*, are called *Kripke structures* in which worlds are described by relational databases and where the time domain serves as the accessibility relation.

**The Timestamp Model.** Temporal databases in the *timestamp model* are defined in terms of temporal relations, relations whose schemas are extended with an additional attribute ranging over the time domain. This attribute is commonly referred to as the *timestamp attribute* or simply *timestamp*.

More formally, a relational symbol $R$ is a *timestamped extension* of a symbol $r \in \rho$ if it contains all attributes of $r$ and a single additional attribute $t$ of the temporal sort (without loss of generality, assuming that it is always the first attribute). A *timestamp temporal database* $D$ is then a first-order structure $D = \{R_1^D \ldots, R_k^D\}$ consisting of the interpretations (instances) for all the temporal extensions $R_i$ of $r_i$ in $\rho$. The instances $R_i^D$ are called *temporal relations*. Similarly to the snapshot case, there is no restriction on the number of timestamps (i.e., the cardinality of the set of timestamps) in such instances; issues connected with the actual finite representation of these relations are addressed below. However, the relation $\{a_1, \ldots, a_k : (t, a_1, \ldots, a_k) \in R_i^D\}$, a snapshot of $R$ at $t$, must be finite for every timestamp $t \in T$.

It is easy to see that snapshot and timestamp temporal models are simply different views of the same (isomorphic) sets of facts and thus represent the same class of temporal databases. Formally, a snapshot temporal database $D$ corresponds to a timestamp temporal database $D'$ (and vice versa) as follows:

$$\forall t. \forall x_1, \ldots, x_k. r^{D(t)}(x_1, \ldots, x_k) \iff R^{D'}(t, x_1, \ldots, x_k),$$

where $r$ and $R$ are a relation in the schema of $D$ and its temporal extension in the schema of $D'$, respectively. This correspondence makes the two models interchangeable. Hence, for temporal queries formulated in query languages such as Temporal Relational Calculus (TRC) or First-order Temporal Logic (FOTL), these two models of point-stamped temporal databases can be used interchangeably.

**The Parametric Model.** Several temporal data models propose to use time-dependent *unary functions* as attribute values of tuples in temporal relations. These models are called *parametric models* or *attribute timestamped models*. As unary functions can be represented by binary relations (e.g., with the first attribute being the timestamp and the second attribute the value of the function for that timestamp), these models are examples of nested, non-first normal form models [5, 6]. Moreover, if the implicit grouping of tuples in such relations is accidental or in the presence of appropriate keys, a first-normal form representation can be obtained by unfolding, similarly to the

case of the nested relational model. The transformation is then defined by:

$$R := \{(t, f_1(t), \ldots, f_k(t)) \mid (f_1, \ldots, f_k) \in R^P, t \in T\},$$

where $R$ is a timestamp relation corresponding to the parametric relation $R^P$. Note also that if a varying number of tuples is to be modeled in this model, *partial functions* must be used in $R^P$; then, however, tuples can become only partially defined for a given time instant, and therefore additional conditions must be enforced. This makes the model quite cumbersome to use, in particular when compared to the snapshot and timestamp models. Many of these difficulties originate from associating timestamps with *uninterpreted constants* that, in the relational model, *do not* carry any information on their own.

## Multiple Atomic Timestamps

The second issue that arises is the question of *how many* timestamps are attached to database objects. So far single-dimensional temporal databases were considered, i.e., where each database object was associated with a single timestamp. The intuition behind such models is that *validity* of data is determined by a single time instant. In the case of the timestamp model, this translates to the fact that temporal relations were allowed only a single (often *distinguished*) temporal attribute. However, there are two natural reasons to relax this requirement and to allow multiple timestamps to be associated with database objects (i.e., multiple timestamp attributes to appear in schemas of relations, either explicitly or implicitly). There are two cases to consider:

**Models with a fixed number of timestamp attributes.** In many cases data modeling requires attaching several timestamps to database objects. Intuitively, a particular piece of information can be associated with, e.g., a timestamp for which the information is *valid* in the modeled world and another timestamp that states when the information is *recorded* in the database. Hence, every object has two timestamps. The resulting data model is called the bitemporal model [7] and the timestamps are called valid time and transaction time, respectively. Similarly, one can envision temporal models with three (or any fixed number) of *distinguished temporal attributes*—attributes with a predetermined interpretation—that are *common to all temporal relation schemas*. Temporal data models with an apriori *fixed* number of timestamp attributes can still be equivalently represented using the snapshot and the timestamp approaches. In the snapshot case, database instances are indexed by fixed tuples of time instants. Hence the bitemporal model can be seen as a two-dimensional plane of relational instances. The timestamp model simply adds an appropriate number of (distinguished) attributes to the timestamp extensions of relational schemes.

**Models with a varying number of timestamp attributes.** While most of the data modeling techniques require only a fixed number of timestamp attributes in schemas of temporal relations, it is often convenient to allow arbitrary number of timestamp attributes to be associated with a database object. This leads to allowing temporal data models without limits on the number of timestamp attributes in schemas of temporal relations. In such models, time dependencies are captured by explicit (user-defined) attributes ranging over the time domain. For a varying number of timestamps, only the *timestamp* view of the temporal data model makes sense.

Fixed-dimensional temporal data models are appealing as they commonly provide an additional built-in *interpretation* for timestamps, e.g., the valid time timestamp always states that the information is *true* in the world at that particular time. Temporal data models with varying, user-defined timestamps do not posses this additional interpretation, and the exact meaning of the timestamps depends on how the world is modeled by the associated attributes (similarly to the standard relational case). However, there is another need for models with varying and unbounded number of timestamp attributes: for temporal query languages based on temporal relational calculus, there cannot be an equivalent temporal relational algebra defined over any of the fixed-dimensional temporal data models (see the entry on Temporal Logic in Database Query Languages or [8, 9] for details).

## Sets of Timestamps: Compact Representation

The point-stamped temporal data models and the associated temporal query languages provide an excellent vehicle for defining a precise semantics of queries and for studying their properties. However, in *practical applications* an additional hurdle has to be overcome: a naive storage of point-stamped temporal relations,

either in the timestamp or in the snapshot model, is often not possible (as the instances of the relations can be infinite, e.g., sets of time instants that represent bounded durations are infinite when a *dense time domain* is used) or impractical (the number of instants is very large). For these and other reasons, many temporal data models associate database objects with *sets* of time instants rather than with single individual time instants.

Temporal models that use complex timestamps, such as intervals, no longer appear to be point-stamped—or in the first normal form (1NF). However, Chomicki [3] has shown that in many cases these complex objects are merely compact representations of a possibly large or even infinite number of time instants associated with a particular fact. The most common approach along these lines is to attach timestamps in the form of *intervals* to facts that persist over time. For example, the temporal relation **WorksFor** in Figure 1 can be compactly (and finitely) represented by the relation:

$$\mathbf{WorksFor}' = \{([2001, 2002], \text{John}, \text{IBM}), ([2004, \infty], \text{John}, \text{Microsoft})\}.$$

Note that such an *interval encoding* is not unique and multiple snapshot equivalent representations can exist. For single-dimensional temporal relations, uniqueness of the representation can be achieved using temporal coalescing. However, it is not clear that meaningful, canonical ways of defining coalescing for multi-dimensional temporal relations, including bitemporal relations, exist [4].

On the other hand, the use of temporal coalescing and other set-oriented operations on interval timestamps, such as intersection of timestamps in temporal joins, indicates that the interval timestamps are indeed solely a representation tool for point-stamped relations: these operations cannot be used in a model that associates truth with the intervals themselves as there is no clear way to do so for the intervals that result from such operations, e.g., it is unclear—from a conceptual point of view—what facts should be associated with an intersection of two intervals *without* implicitly assuming that facts associated with the original two intervals hold *for all time points contained in those two intervals*, or at least for certain sub-intervals.

The choice of *intervals* to compactly represent adjacent timestamps originates from the structure of the temporal domain: intervals are the only (convex) one-dimensional sets that can be defined by (first-order) formulas in the language of linear order. For more structured time domains, however, the repertoire of encodings for sets of timestamps can be richer, e.g., using formulas describing *periodic sets*, etc., as compact timestamps.

## Query Languages and Integrity Constraints

Point-stamped data models support point-based temporal query languages that are commonly based on extensions of the relational calculus (first-order logic). The two principal extensions are as follows:

1. First-order Temporal Logic: a language with an implicit access to timestamps using temporal connectives (see the entry Temporal Logic in Database Query Languages), and
2. Temporal Relational Calculus: a two-sorted logic with variables and quantifiers explicitly ranging over the time domain. (see the entry Temporal Relational Calculus).

These languages are the counterparts of the snapshot and timestamp models. However, unlike the data models that are equivalent in their expressiveness, the second language is strictly more expressive than the first [1, 9] (see the entry Temporal Logic in Database Query Languages for details). Temporal integrity constraints can also be conveniently expressed in these languages (see Temporal Integrity Constraints or [4]).

### KEY APPLICATIONS

Point-based temporal data models serve as the underlying data model for most abstract temporal query languages.

### CROSS REFERENCE

bitemporal relation, constraint data model, data domain, duplicate semantics, First-order Temporal Logic, key, nested relational model, non first normal form (N1NF), point-stamped temporal data models, relational model, snapshot equivalence, temporal data models, temporal coalescing, temporal element, temporal granularity, temporal integrity constraints, temporal joins, temporal logic in database query languages, temporal query languages, temporal relational calculus, time domain, time instant, transaction time, valid time.

### RECOMMENDED READING

[1] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal Versus First-Order Logic to Query Temporal Databases. In *ACM Symposium on Principles of Database Systems*, pages 49–57, 1996.

[2] C. Bettini, S. Jajodia, and X. S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer, 2000.

[3] J. Chomicki. Temporal Query Languages: A Survey. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic, First International Conference*, pages 506–534. Springer-Verlag, LNAI 827, 1994.

[4] J. Chomicki and D. Toman. Temporal Databases. In M. Fischer, D. Gabbay, and L. Villa, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier *Foundations of Artificial Intelligence*, 2005.

[5] J. Clifford, A. Croker, and A. Tuzhilin. On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 496–533. Benjamin/Cummings, 1993.

[6] J. Clifford, A. Croker, and A. Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, 19(1):64–116, 1994.

[7] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unification of Temporal Data Models. In *18th International Conference on Data Engineering*, 1993.

[8] D. Toman. On Incompleteness of Multi-dimensional First-order Temporal Logics. In *International Symposium on Temporal Representation and Reasoning and International Conference on Temporal Logic*, pages 99–106, 2003.

[9] D. Toman and D. Niwinski. First-Order Queries over Temporal Databases Inexpressible in Temporal Logic. In *International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, 1996.

[10] J. Wijsen. Temporal FDs on Complex Objects. *ACM Transactions on Database Systems*, 24(1):127–176, 1999.

# PROBABILISTIC TEMPORAL DATABASES

V.S. Subrahmanian
University of Maryland, College Park.
vs@cs.umd.edu
www.cs.umd.edu/~vs/

## SYNONYMS

Temporally uncertain databases, temporally indeterminate databases

## DEFINITION

There are many applications where the fact that a given event occurred is known, but where there is uncertainty about exactly when that event occurred. Such events are called temporally indeterminate events. Probabilistic temporal databases attempt to store information about events that are both temporally determinate and temporally indeterminate. For the latter, they specify a set of time points (often an interval) – it is known that the event occurred at some time point in this set. The probability that the event occurred at a specific time point is given by a probability distribution or by one of a set of probability distributions.

## HISTORICAL BACKGROUND

There is no shortage of events that are known to have certainly occurred, but where the exact dates are not known with certainty. For instance, the exact date of the extinction of dinosaurs is unknown – nor does the historical record show the precise date when Cyrus the Great of Persia was born. The latter, estimated by scholars to be between 590 and 576 BC, is an excellent example of a temporally indeterminate event.

Techniques to study such temporally indeterminate events in databases started with information about *partial null values* [1,2]. Meanwhile, in a largely separate community, researchers focused on the problem of incorporating probabilistic information within a relational database. One of the earliest efforts was due to Cavallo and Pittarelli [3] who proposed an extension of the relational data model to include probabilities – they proposed a partial algebra consisting of projection and join operators. Much early work in this field, such as that of [4,5] made important advances under some restrictions Lakshmanan et. al.'s ProbView system [6] proposed two representations of probabilistic data: a *probabilistic tuple* has the form $((t_1,V_1),\ldots,(t_n,V_n))$ where each $t_i$ is a set of possible values and $V_i$ is a probability distribution over the set. They then showed that each probabilistic tuple could be "flattened" into an "annotated" representation. Annotated tuples look like ordinary tuples except that a probability is attached to the tuple as a whole (and a special "path" field was introduced to handle tuples with identical data). ProbView introduced the important concept of conjunction and disjunction strategies that allowed users to specify – in their query – their knowledge about the dependencies between events when posing the query. Thus, the barrier of the independence assumption in previous works was overcome. They proposed a query algebra for annotated tuples, developed query rewrite rules, and developed view maintenance algorithms.

Much work has subsequently been done on temporal probabilistic data. Despite a long history of reasoning about time and uncertainty in AI [7], a major advance in temporal probabilistic databases occurred when Dyreson and Snodgrass [8] proposed the notion of an indeterminate instant. This is just like one of the $(t_i, V_i)$ pairs in the annotated representation of [6] with the exception that the $t_i$ component represents a set of time points. They then extended the SQL query language in several ways. First, they added constructs to indicate that a temporal attribute in indeterminate. Second, they added the concept of "correlation credibility" which allows a query to modify indeterminate temporal data (e.g. by choosing a max temporal value or an expected temporal value). Third, they introduced the concept of "ordering plausibility" which specifies an ordering about the plausibility levels of different conditions in the WHERE clause of an SQL query. The next major breakthrough in temporal probabilistic databases came when Dekhtyar et. al. [9] proposed a temporal probabilistic relational database model that added "tp-cases" to ordinary relational tuples. A tp-case contained two constraints (see also the related article in this encyclopedia on "Temporal Constraints") one of which was used to denote valid time points as solutions of the constraints, and the other was used in conjunction with a distribution function to infer a probability distribution on the solutions of the first constraint. They developed an extension of the relational algebra that directly manipulated tp-tuples and used the conjunction and disjunction strategies of ProbView [6] to avoid making independence assumptions. TP-databases were one of the first temporal probabilistic DBMSs to report real world applications for the US Navy [10]. It was later used to build similar applications for the US Army as well. Later, Biazzo et. al. [11] extended this work to the case of object bases containing temporal indeterminacy. The Trio system [12] extends temporal indeterminacy models to include lineage information as well. More recently, SPOT databases [14] allow reasoning in the presence of space, time and uncertainty.

## SCIENTIFIC FUNDAMENTALS

Consider a relation $R(A_1,\ldots,A_n)$ which describes events whose temporal validity is not precisely known. In order to identify when the tuples in such a relation are valid, *constraints* can be to describe the period of validity of the tuple. The table below (ignoring the shaded column for now) is a temporally indeterminate database about vehicle locations.

The first row in this table says that the event "Vehicle V1 was at location *a*" occurred at some time during the closed interval [11,20]. This is a form of temporal indeterminacy with no probabilities. Suppose there is a probability distribution over the last column. A common temptation is to add an additional column specifying the "name" of the distribution (e.g. "u" might be the uniform distribution, $g_r$ may be a geometric distribution with a parameter r, and so forth). Such a table might now look like this:

| ***VehicleID*** | ***VehicleType*** | ***Location*** | ***Time*** | ***PDF*** |
|---|---|---|---|---|
| V1 | T72 | *a* | 11 <= t <= 20 | u |
| V1 | T72 | *b* | 18 <= t <= 25 | $\gamma_{0.5}$ |
| V1 | T72 | *c* | 24 <= t <= 27 | $\gamma_{0.2}$ |

Now consider the table above with the shaded column included. The first row in the table now says that there is a 10% probability that vehicle V1 will be at location *a* at time 11 (and the same for times 12, 13, and so on till time 20) and that the probability is uniformly distributed (probability distribution function or pdf "u"). In contrast, the second row says something different because the pdf $\gamma$ treats time intervals differently. It says, implicitly, that the probability that vehicle V1 will be at location *b* at time 18 is 0.15, at time 19 is 0.25, at time 20 is 0.125, and so on.

How should queries over this representation of the database be answered? To see this, consider a temporal query which says "Select all tuples where $20 < Time < 23$." The only tuple that has any chance of satisfying this constraint is the second one. However, there is no *guarantee* that the second tuple actually is valid during this interval. Fortunately, there is a 9.375% chance that this is the case – so the system could return the probabilistically valid response saying that the second tuple is valid with probability 9.375%. Unfortunately, there is no way of returning this answer to the user *unless the implicit assumption in all relational databases that the output schema for selection queries should match the input schema is sacrificed.* It is clearly inappropriate to just return the table

| ***VehicleID*** | ***VehicleType*** | ***Location*** | ***Time*** | ***PDF*** |
|---|---|---|---|---|
| V1 | T72 | *b* | 20 < t < 23 | $\gamma_{0.5}$ |

The ***Time*** field here is computed merely by solving the constraint in the second tuple of the input relation in conjunction with the constraint in the query. Unfortunately, the distribution in the ***PDF*** field is incorrect, and would need to be recomputed. This is further complicated by the fact that the shape of the original geometric distribution does not look the same when restricted to a portion (of interest) of the original distribution.

There have been two attempts to solve the problem of dealing with what happens when a distribution is manipulated. Dyreson and Snodgrass [8] develop a "rod and point" method to store distributions and infer new distributions when selection operations of the kind above are performed. They approximate a probability mass by splitting it into chunks called "rods". However, each chunk can have a different length. An approximate representation of the original distribution is obtained through this rod mechanism.

Dekhtyar et. al. [9] solve this problem by using some extra space. They require that the "base relation" contains *two constraints,* both of which are identical initially. They would represent the original relation as follows:

| ***VehicleID*** | ***VehicleType*** | ***Location*** | ***Time*** | ***Time 2*** | ***PDF*** |
|---|---|---|---|---|---|
| V1 | T72 | *a* | 11 <= t <= 20 | 11 <= t <= 20 | U |
| V1 | T72 | *b* | 18 <= t <= 25 | 18 <= t <= 25 | $G_{0.5}$ |
| V1 | T72 | *c* | 24 <= t <= 27 | 24 <= t <= 27 | $G_{0.2}$ |

In base relations, the ***Time*** and ***Time2*** fields have exactly the same constraints in them. When queries are executed, the ***Time*** field ends up denoting valid time and changes

based on the query – however, the ***Time2*** field rarely changes. When the selection query mentioned above is execution, they would return the answer:

| ***VehicleID*** | ***VehicleType*** | ***Location*** | ***Time*** | ***Time 2*** | ***PDF*** |
|---|---|---|---|---|---|
| V1 | T72 | *b* | 20 < t < 23 | 18 <= t <= 25 | $G_{0.5}$ |

This is very subtle. The ***Time*** attribute here specifies the valid time (in this case, time points 21 and 22). The ***Time2*** attribute is a system-maintained attribute that need not be shown to the user which says "apply the PDF mentioned in the ***PDF*** field to the solutions of the ***Time2*** constraint – but only show the probability values for the solutions of the ***Time*** constraint). Thus, ***Time2*** is used to derive the probabilities for each valid time point. In the above case, the valid time points are 21 and 22, and their probabilities are derived – using the distribution in the ***PDF*** column applied to the constraint in the ***Time2*** column – to get probabilities of 0.0625 and 0.03125, respectively. [9] goes on and specifies how to add additional "low" and "high" probability fields to such relations and provides normalization methods.

Cartesian products between two relations are more complex. When concatentating tuple s*t1* and *t2* from two different relations, it is important to consider the probability that both tuples will be valid at a given time point. This requires knowing the relationship between the events being denoted by these two tuples: are they independent? Are they correlated somehow? Is there no information about the relationship? Thus, a *conjunction strategy* must be specified in the query when a Cartesian product (and hence a join operation) is being performed. Dekhtyar et. al. [9] show how Cartesian products and joins can be computed under any assumption specified in a user query.

Another recent effort is the one on Trio [12] which attempts to deal with time, uncertainty, and lineage. They address the fact that in many applications involving uncertainty (such as crime-fighting applications), any "final" answer needs to be explainable. As a consequence, they introduce "lineage" parameters when answering a query – informally speaking, the lineage parameter associated with a tuple in an answer ( similar to the "path" parameter in [6]) associates a "justification" for each tuple. This justification references the set of base tuples that caused the derived tuple to be placed in an answer.

## KEY APPLICATIONS

Temporal probabilistic databases have already been used in defense applications. For instance, [10] describes work in which temporal probabilistic databases are used to store the results of predictions about where enemy submarines will be in the future, when they will be there, and with what probability. In fact, this raises a large set of possible applications based on reasoning about moving objects. For defense applications, cell phone applications, logistics applications, and many other application domains, there is interest in knowing where a moving object will be in the future, when it is expected to be there, and with what probability. Cell phone companies can use such data to understand and better handle load on cell towers.

Moreover, as the world is becoming increasing "geo-location aware" through the use of devices like RFID tags and GPS locators, it is clear that reasoning about where vehicles will be in the future and with what probability will be important in a wide range of applications such as traffic light settings to ease road congestion, recommending detours on highway signs, and more effectively directing 911 traffic in congested situations. These would not be possible without a good estimate of when and where and with what probabilities vehicles will be in the future.

Logistics applications are another important class of applications where companies need to plan activities in the presence of uncertainty about when various supply items will arrive. Corporations today use complex prediction models to learn about suppliers' performance.

Financial applications are another major source of temporal uncertainty. Banks need to have a good idea of their incoming funds. For instance, a credit card provider deals with constant uncertainty about when people will pay their credit card bills, how much of the bills they will pay, and how much they will carry forward as debt. Such applications embody a mix of data uncertainty and temporal uncertainty.

## FUTURE DIRECTIONS

Three major areas of expansion include:

1. Temporal probabilistic aggregates. To date, there are almost no techniques to manage aggregates efficiently in temporal probabilistic databases. Most methods would compute aggregates by first answering a non-aggregate query and then deriving aggregates from there: however, techniques to scalably answer aggregate queries are required.
2. Probabilistic spatio-temporal reasoning. Moving objects clearly have a spatial component – hence, reasoning about them involves a neat fusion of temporal reasoning, spatial reasoning, and probabilistic reasoning. Some work on indexing in such domains has recently been proposed. However, much future work is needed, especially in understanding the correlations that exist between the presence of a vehicle at time $t$ and its presence at another location at time $t+1$.
3. Query optimization. Recent work [13] has made a good start on query optimization in probabilistic databases – however, query optimization in databases involving time and uncertainty has a ways to go. Such methods are critical for scaling applications.

## CROSS REFERENCES

Temporal constraints, Qualitative Temporal Reasoning

## RECOMMENDED READING

1. John Grant: Partial Values in a Tabular Database Model. Inf. Process. Lett. 9(2): 97-99 (1979).
2. A. Ola: Relational Databases with Exclusive Disjunctions, Proc. IEEE International Conf. on Data Engineering, pages 328-336, Tempe, AZ, Feb. 1992.
3. Roger Cavallo, Michael Pittarelli: The Theory of Probabilistic Databases. VLDB 1987: 71-81.
4. Daniel Barbará, Hector Garcia-Molina, Daryl Porter: The Management of Probabilistic Data. IEEE Trans. Knowl. Data Eng. 4(5): 487-502 (1992)
5. Debabrata Dey, Sumit Sarkar: A Probabilistic Relational Model and Algebra. ACM Trans. Database Syst. 21(3): 339-369 (1996)
6. Laks V. S. Lakshmanan, Nicola Leone, Robert B. Ross, V. S. Subrahmanian: ProbView: A Flexible Probabilistic Database System. ACM Trans. Database Syst. 22(3): 419-469 (1997)
7. S. Kraus and V.S. Subrahmanian. Multiagent reasoning with probability, time and beliefs, Intl. Journal of Intelligent Systems, 10, 5, pages 459-499, 1994.
8. Curtis E. Dyreson, Richard T. Snodgrass: Supporting Valid-Time Indeterminacy. ACM Trans. Database Syst. 23(1): 1-57 (1998)
9. Alex Dekhtyar, Robert Ross, V. S. Subrahmanian: Probabilistic temporal databases, I: algebra. ACM Trans. Database Syst. 26(1): 41-95 (2001)
10. Ranjeev Mittu, Robert Ross: Building upon the Coalitions Agent Experiment (COAX) - Integration of Multimedia Information in GCCS-M using IMPACT. Multimedia Information Systems 2003: 35-44
11. Veronica Biazzo, Rosalba Giugno, Thomas Lukasiewicz, V. S. Subrahmanian: Temporal Probabilistic Object Bases. IEEE Trans. Knowl. Data Eng. 15(4): 921-939 (2003)
12. Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, Jennifer Widom: Trio: A System for Data, Uncertainty, and Lineage. VLDB 2006: 1151-1154.
13. N. Dalvi and D. Suciu. Answering queries from Statistics and Probabilistic Views, Proc. VLDB 2004.
14. A. Parker, V.S. Subrahmanian and J. Grant. A Logical Formulation of Probabilistic Spatial Databases, IEEE Trans. Knowl. Data Eng. 19(11): 1541-1556.

# QUALITATIVE TEMPORAL REASONING

**Paolo Terenziani**
**Universita' del Piemonte Orientale "Amedeo Avogadro",** http://www.di.unito.it/~terenz

## SYNONYMS

Non-metric temporal reasoning, reasoning with qualitative temporal constraints, qualitative temporal deduction

## DEFINITION

*Qualitative temporal constraints* are non-metric *temporal constraints* stating the *relative* temporal position of facts that happen in time (e.g., fact $F_1$ is *before* or *during* fact $F_2$). Different types of qualitative constraints can be defined, depending on whether facts are instantaneous, durative, and\or repeated. *Qualitative temporal reasoning* is the process of reasoning with such temporal constraints. Given a set of qualitative temporal constraints, qualitative temporal reasoning can be used for different purposes, including checking their *consistency*, determining the *strictest constraints* between pairs of facts (e.g., for query answering purposes), pointing out a *consistent scenario* (i.e., a possible instantiation of all the facts on the timeline in such a way that all temporal constraints are satisfied).

## HISTORICAL BACKGROUND

In several domains and\or application areas, *temporal indeterminacy* has to be coped with. In such domains, the *absolute time* when facts hold (i.e., the exact temporal location of facts) is generally unknown. On the other hand, in many of such domains, *qualitative* temporal constraints about the *relative* temporal location of facts are available, and reasoning about such constraints is an important task. As a consequence, there is a long tradition for qualitative temporal reasoning within *philosophical logic* (consider, e.g., Walker [15], who first formulated a kind of logical calculus about durative periods, and Prior's seminal branching time logics [11]; the interested reader is referred to the encyclopedia entry 'Time in Philosophical Logic').

In particular, qualitative temporal reasoning is important in several artificial intelligence areas, including planning, scheduling, and natural language understanding. Therefore, starting from the beginning of the 1980's, several *specialized* approaches (as opposed to *logical* approaches, which are usually general-purpose) to the *representation* of *qualitative* temporal constraints and to temporal *reasoning* about them have been developed, especially within the artificial intelligence area.

The first milestone in the specialized approaches to qualitative temporal reasoning dates back to Allen's Interval Algebra [1], coping with qualitative temporal constraints between intervals (called *periods* by the database community, and henceforth in this entry), to cope with durative facts. Further on, several other algebrae of qualitative temporal constraints have been developed, to cope with instantaneous [14] or repeated\periodic [9, 12] facts, and several temporal reasoning systems have been implemented and used in practical applications (see, e.g., some comparisons in Delgrande et al. [4]).

Significant effort in the area has been devoted to the analysis of the *trade-off* between the *expressiveness* of the representation formalisms and the *complexity* of *correct* and *complete* temporal reasoning algorithms operating on them (see, e.g., the survey by Van Beek [13]). Since consistency checking in Allen's algebra is NP-complete, several approaches, starting

from Nebel and Burkert [10], have focused on the identification of *tractable fragments* of it. The *integration* of qualitative and metric constraints has also been analyzed (see, e.g., Jonsson and Backstrom [7]). Recent developments also include *incremental* [6] and *fuzzy* [2] qualitative temporal reasoning. Moreover, starting from the 1990's, some approaches have also started to investigate the adoption of qualitative temporal constraints and temporal reasoning within the temporal database context [8,3].

## SCIENTIFIC FUNDAMENTALS

*Qualitative* temporal constraints concern the relative temporal location of facts on the timeline. A significant and milestone example is Allen's Interval Algebra (henceforth: IA). Allen pointed out the 13 primitive qualitative relations between time periods: before, after, meets, met-by, overlaps, overlapped-by, starts, started-by, during, contains, ends, ended-by, equal (see the encyclopedia entry 'Temporal constraint' for a detailed description of such relations). These relations are *exhaustive* and *mutually exclusive*, and can be combined in order to represent disjunctive relations. For example, the constraints in (Ex.1) and (Ex.2) state that $F_1$ is before or during $F_2$, which, in turn, is before $F_3$.

(Ex.1) $F_1$ (BEFORE,DURING) $F_2$

(Ex.2) $F_2$ (BEFORE) $F_3$

In Allen's approach qualitative temporal reasoning is based on two algebraic operations over relations on time periods: intersection and composition. Given two possibly disjunctive relations $R_1$ and $R_2$ between two facts $F_1$ and $F_2$, temporal intersection (henceforth $\cap$) determines the most constraining relation R between $F_1$ and $F_2$. For example, the temporal intersection between (Ex.2) and (Ex.3) is (Ex.4). On the other hand, given a relation $R_1$ between $F_1$ and $F_2$ and a relation $R_2$ between $F_2$ and $F_3$, composition (@) gives the resulting relation between $F_1$ and $F_3$. For example, (Ex.5) is the composition of (Ex.1) and (Ex.2) above.

(Ex.3) $F_2$ (BEFORE,MEETS,OVERLAPS) $F_3$

(Ex.4) $F_2$ (BEFORE) $F_3$

(Ex.5) $F_1$ (BEFORE) $F_3$

In Allen's approach, temporal reasoning is performed by a *path consistency* algorithm that basically computes the *transitive closure* of the constraints by repeatedly applying intersection and composition. Abstracting from many optimisations, such an algorithm can be schematized as follows:

**Repeat**
  **For** all triples of facts $<F_i,F_k,F_j>$
    **Let** $R_{ij}$ denote the (possibly ambiguous) relation between $F_i$ and $F_j$
    $R_{ij} \leftarrow R_{ij} \cap (R_{ik} @ R_{kj})$
**Until** quiescence

Allen's algorithm operates in a time *cubic* in the number of periods. However, such an algorithm is *not complete* for the Interval Algebra (in fact, checking the consistency of a set of temporal constraints in the Interval Algebra is NP-hard [14]).

While in many approaches researchers chose to adopt Allen's algorithm, in other approaches they tried to design less expressive but *tractable* formalisms. For example, the Point Algebra is

defined in the same way as the Interval Algebra, but the temporal elements are time points. Thus, there are only three primitive relations between time points (i.e., <,=, and >), and four disjunctive relations (i.e., (<,=), (>,=), (<,>), and (<,=,>); see the entry 'Temporal constraints' for more details). In the Point Algebra, sound and complete constraint propagation algorithms operate in polynomial time (namely, in $O(n^4)$, where n is the number of points [13]). Obviously, the price to be paid for tractability is expressive power: not all (disjunctive) relations between periods can be mapped onto relations between their endpoints. For instance $F_1$ *(BEFORE,AFTER)* $F_2$ cannot be mapped into a set of (possibly disjunctive) pairwise relations between time points; indeed an explicit disjunction between two different pairs of time points is needed (i.e., $(end(F_1) < start(F_2))$ *or* $(end(F_2) < start(F_1))$). The Continuous Point Algebra restricts the Point Algebra excluding inequality (i.e., (<,>)). Allen's path consistency algorithm is both sound and complete for such an algebra, and operates in $O(n^3)$ time, where n is the number of time points (for more details, see, e.g., the survey by Van Beek [13]).

A different simplification of Allen's Algebra has been provided by Freksa [5]. Freska has identified coarser qualitative temporal relations than Allen's ones, based on the notion of *semi-intervals* (i.e., beginnings and ending points of durative events). As an example of relation on semi-intervals, Freksa has introduced the "older" relation ($F_1$ is *older* than $F_2$ if $F_1$'s starting point is before $F_2$'s starting point, with no constraint on the ending points); notice that Freska's *older* relation corresponds to a disjunction of five Allen's relations (i.e., *before*, *meets*, *overlaps*, *finished-by*, *contains*). Freska has also shown that relations between semi-intervals result in a possible more compact notation and more efficient reasoning mechanisms, in particular if the initial knowledge is, at least in part, coarse knowledge.

Another mainstream of research about qualitative temporal reasoning focused on the identification of *tractable fragments* of Allen's algebra. The milestone work by Nebel and Burkert [10] first pointed out the "*ORD-Horn subclass*", showing that reasoning in such a class is a polynomial time problem and that it constitutes a maximal tractable subclass of Allen's algebra.

Starting from the beginning of 1990's, some *integrated* temporal reasoning approaches have been devised in order to deal with both qualitative and quantitative (i.e., metric) temporal constraints. For instance, Jonsson and Backstrom [7] have proposed a framework, based on *linear programming*, that deals with both qualitative and metric constraints, and that also allows one to express constraints on the relative duration of events (see, e.g., (Ex.6)).

(Ex.6) John drives to work at least 30 minutes more than Fred does.

Many other important issues have to be taken into account when considering qualitative temporal reasoning. For example, starting from Ladkin's seminal work [9], qualitative constraints between *repeated* facts have been considered. In the same mainstream, Terenziani has proposed an extension of Allen's algebra to consider qualitative relations between periodic facts [12]. Terenziani's approach deals with constraints such as (Ex.7):

(Ex.7) Between January 1, 1999 and December 31, 1999 on the first Monday of each month, Andrea went to the post office *before* going to work.

In Terenziani's approach, temporal reasoning over such constraints is performed by a path consistency algorithm which extends Allen's one. Such an algorithm is sound but not complete and operates in cubic time with respect to the number of periodic facts.

As concerns more strictly the area of (temporal) databases, starting from the middle of the 1990's, some researchers have started to investigate the treatment of qualitative temporal constraints within temporal (relational) databases (see, e.g., [8,3]). In such approaches, the

*valid time* of facts (tuples) is represented by symbols denoting time periods, and *qualitative* and *quantitative temporal constraints* are used in order to express constraints on their relative location in time, on their duration, and so on.

Koubarakis [8] first extended the constraint database model to include indefinite (or uncertain) temporal information (including qualitative temporal constraints). Koubarakis proposed an explicit representation of temporal constraints on data; moreover, the local temporal constraints on tuples are stored into a dedicated attribute. He also defined the algebraic operators, and theoretically analysed their complexity. On the other hand, the work by Brusoni et al., [3] mainly focused on defining an integrated approach in which "standard" artificial intelligence temporal reasoning capabilities (such as the ones sketched above in this entry) are suitably extended and paired with an (extended) relational temporal model. First, the data model is extended in such a way that each temporal tuple can be associated with a set of identifiers, each one referring to a time period. A separate relation is used in order to store the qualitative (and quantitative) temporal constraints about such periods. The algebraic operations of intersection, union and difference are defined over such sets of periods, and *indeterminacy* (e.g., about the existence of the intersection between two periods) is coped with through the adoption of *conditional intervals*. Algebraic relational operators are defined on such a data model, and their complexity analysed. Finally, an integrated and modular architecture combining a temporal reasoner with an extended temporal database is described, as well as a practical application to the management of temporal constraints in clinical protocols and guidelines.

## KEY APPLICATIONS

Qualitative temporal constraints are pervasive in many application domains, in which the absolute and exact time when facts occur is generally unknown, while there are constraints on their relative order (or temporal location). Such domains include the "classical" domains of planning and scheduling, but also more recent ones such as managing multimedia presentations or clinical guidelines.
As a consequence, temporal reasoning is already a well-consolidated area of research, especially within the artificial intelligence community, in which a large deal of works aimed at building *application-independent* and *domain-independent* managers of temporal constraints. Such managers are intended to be *specialised knowledge servers* that represent and reason with temporal constraints, and that *co-operate* with other software modules in order to solve problems in different applications. For instance, in planning problems, a temporal manager could co-operate with a planner, in order to check incrementally the temporal consistency of the plan being built. In general, the adoption of a specialised temporal manager is advantageous from the computational point of view (e.g., with respect to general logical approaches based on theorem proving), and it allows programmers to focus on their domain-specific and application-specific problems and to design modular architectures for their systems.

On the other hand, the impact and potentiality of extensively exploiting qualitative temporal reasoning in temporal databases have only been minimally explored by the database community, possibly due to the computational complexity that it necessarily involves. However, in the last years (temporal) databases are increasingly going to be applied to new applications domains, in which the structure and the inter-dependencies of facts (including the temporal dependencies) play a major role, while the assumption that the absolute temporal location of facts is known does no longer hold. Significant application areas include database applications to store workflows, protocols, guidelines (see, e.g., the example in [3]), and so on. To cope with such applications, "hybrid" approaches in which qualitative (and\or quantitative) temporal reasoning mechanisms are paired with classical temporal database frameworks (e.g., along the lines suggested in [3]) are likely to play a significant role in a near future. The role of qualitative

temporal constraints (and temporal reasoning) may be even more relevant at the *conceptual* level. Several temporal extensions to conceptual formalisms (such as the Entity-Relationship one) have been proposed in the last years, and there is an increasing awareness that, in many domains, qualitative (and\or quantitative) temporal constraints between conceptual objects are an intrinsic part of the conceptual model. As a consequence, qualitative temporal reasoning techniques such as the ones discussed above, can, in a near future, play a relevant role also at the conceptual modelling level.

## FUTURE DIRECTIONS

One of several possible future research directions of qualitative temporal reasoning, which may be particularly interesting for the database community, is its application to "Active Conceptual Modeling". In his keynote talk at ER'2007, Prof. P. Chen, the creator of the Entity-Relationship model, has stressed the importance of extending traditional conceptual modeling to "Active Conceptual Modeling". Roughly speaking, the term 'active' denotes the need for coping with evolving models having learning and prediction capabilities. Such an extension is needed in order to adequately cope with a new range of phenomena, including disaster prevention and management. Chen has stressed that "Active Conceptual Modeling" requires, besides the others, an explicit treatment of time. The extension and integration of qualitative temporal reasoning techniques into the "Active Conceptual Modeling" context is likely to give a major contribution to the achievement of predictive and learning capabilities, and to become a potentially fruitful line of research.

## CROSS REFERENCES

(other topics in the Encyclopedia which may be of interest to the reader of this entry)
Allen's Relations
Temporal constraints
Time in Philosophical Logic
Absolute time
Relative time
Temporal Indeterminacy
Temporal Integrity Constraints
Temporal periodicity
Time period
Valid time

## RECOMMENDED READING


[1] Allen, J.F., Maintaining knowledge about temporal intervals, C*ommunications of the ACM 26(11)*: 832-843, 1983.

[2] Badaloni, S., Giacomin, M., The algebra $IA^{fuz}$: a framework for qualitative fuzzy temporal reasoning, Artificial Intelligence 170(10), 872-908, 2006.

[3] Brusoni, V., Console, L., Pernici, B., Terenziani, P., Qualitative and Quantitative Temporal Constraints and Relational Databases: Theory, Architecture, and Applications, IEEE Transactions on Knowledge and Data Engineering 11(6), 948-968, 1999.

[4] Delgrande, J., Gupta, A., and Van Allen, T., A comparison of point-based approaches to qualitative temporal reasoning. Artificial Intelligence 131 (1-2), 135-170, 2001.

[5] Freksa, C.,Temporal reasoning based on semi-intervals. Artificial Intelligence 54(1-2), 199-227, 1992.

[6] Gerevini, A., Incremental qualitative temporal reasoning: algorithms for the point algebra and the ORD-Horn class, Artificial Intelligence 166(1-2), 37-80, 2005.

[7] Jonsson, P., and Backstrom, C., A unifying approach to temporal constraint reasoning, *Artificial Intelligence 102*, 143-155, 1998.

[8] Koubarakis, M., Database models for infinite and indefinite temporal information, Information Systems 19(2), 141-173, 1994.

[9] Ladkin, P., Time Representation: A Taxonomy of Interval Relations, Proc. fifth National Conf. on Artificial Intelligence, pp. 360-366, Philadelphia, PA, August 1986.

[10] Nebel, B., and Burkert, H.J., Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra, Journal of the ACM 42(1), 43-66, 1995.

[11] Prior, A.N., *Past, Present and Future*. Oxford University Press, Oxford, 1967.

[12] Terenziani, P., Integrating calendar-dates and qualitative temporal constraints in the treatment of periodic events, *IEEE Transactions on Knowledge and Data Engeneering 9(5)*, 763-783, 1997.

[13] Van Beek, P., Reasoning about Qualitative Temporal Information, Artificial Intelligence 58(1-3), 297-326, 1992

[14] Vilain, M., and Kautz, H., (1986) Constraint propagation algorithms for temporal reasoning, In: Kehler T, Rosenschein S, Filman r, and Patel-Schneider P (eds), *Proc. Fifth National Conference on Artificial Intelligence (AAAI'86)*, 377-382, Philadelphia, PA, 1986.

[15] Walker A.G., Durèes et instants. La Revue Scientifique, (3266), p. 131 ff. 1947.

# RELATIVE TIME

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

none

**DEFINITION**

Viewing a temporal database as a collection of time-referenced, or timestamped, facts, a time reference in such a database is called *relative* if its value is dependent of the context, e.g., the current time, *now*.

**MAIN TEXT**

The relationship between times can be qualitative (before, after, etc.) as well as quantitative (3 days before, 397 years after, etc.). If quantitative, the relationship is specified using a time span.
Examples: "Mary's salary was raised yesterday," "it happened sometime last week," "it happened within 3 days of Easter," "the Jurassic is sometime after the Triassic," and "the French revolution occurred 397 years after the discovery of America."

**CROSS REFERENCE***

Absolute Time, Now in Temporal Databases, Qualitative Temporal Reasoning, Time Span, Temporal Database

**REFERENCES***

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# SCHEMA EVOLUTION

John F. Roddick, Flinders University, Australia

## DEFINITION

Schema evolution deals with the need to retain current data when database schema changes are performed. Formally, **Schema Evolution** is accommodated when a database system facilitates database schema modification without the loss of existing data, (q.v. the stronger concept of **Schema Versioning**)[1].

## HISTORICAL BACKGROUND

Since schemata change and/or multiple schemata are often required, there is a need to ensure that extant data either stays consistent with the revised schema or is explicitly deleted as part of the change process. A database that supports schema evolution supports this transformation process.

The first schema evolutioning proposals discussed database conversion primarily in terms of a set of transformations from one schema to another [10]. These transformations focussed on the relational structure of the database and included name changing, changing the membership of keys, composing and decomposing relations both vertically and horizontally and so on. In all cases only one schema remained and all data (that still remained) was coerced (ie. copied from one type to another) to the new structure.

Schema evolution has also been covered in the proposals to manage issues such as data coercion [6, 14], authority control [2] and query language support [9].

## SCIENTIFIC FUNDAMENTALS

Schema evolution is related to the view-update problem, discussed in-depth when the relational model was introduced [1], and is strongly linked to the notion of information capacity [5, 8]. Specifically, non-loss evolution can only be guaranteed when the information capacity of the new schema exceeds that of the existing schema. Formally, if $I(S)$ is the set of all valid instances of $S$, then for non-loss evolution $I(S_{new}) \supseteq I(S_{old})$. One novel solution is the integration of schema evolution with the database view facilities. When new requirements demand schema updates for a particular user, then the user specifies schema changes to a personal view, rather than to the shared base schema [**?**].

Once a schema change is accepted, the common procedure is for the underlying instances to be coerced to the new structure. Since the old schema is obsolete this presents few problems and is conceptually simple. However results in an inability to reverse schema amendments. Schema versioning support provides two other options (q.v.).

Four classes of schema evolution can be envisaged. Each type brings different problems.

**Attribute Evolution** occurs when attributes are added to, deleted from or renamed in a relation. Issues here include the values to be ascribed to attributes in tuples stored under a new version that does not possess the attribute.

- **Domain Evolution** occurs when the domain over which an attribute is defined is altered. Issues here include implying accuracy that does not exist in existing data when, for example, attributes defined as integers are converted to reals, and in truncation when character fields are shortened.
- **Relation Evolution** occurs when the relational structure is altered through the definition, deletion, decomposition or merging of a relation. Such changes are almost always irreversible.
- **Key Evolution** occurs when the structure of a primary key is altered or when foreign keys are added or removed. The issues here can be quite complex. For example, removing an attribute from a primary key may not violate the primary key uniqueness constraint for current data (the amendment can be rejected if

[1] Schema evolution and schema versioning has been conflated in the literature with the two terms occasionally being used interchangeably. Readers are thus also encouraged to read also the entry for **Schema Versioning**.

it does) but in a temporal database may still do so for historical information.

Note that one change may involve more than one type of evolution, such as changing the domain of a key attribute. These changes may also be reflected in the conceptual model of the system. For example, the addition of an entity in an EER diagram would result in the addition of a relation in the underlying relational model; deleting a 1-to-many relationship would remove a foreign key constraint, and so on.

## KEY APPLICATIONS*

Schema changes are linked to either error correction or design change. It is therefore useful if the design decisions can be consulted and the users can interact with schema changes at a high level. One way is to propagate requirements changes to database schemas [4] or provide better support for metadata management by providing a higher level view in which models can be mapped to each other [7].

In order to quantify the types of schema evolution, Sjøberg [11] investigated change to a database system over eighteen months, covering six months of development and twelve months of field trials. A more recent study complements this by following the changes in an established database system over many years [3].

## FUTURE DIRECTIONS

The major directions for schema versioning research have moved from low-level handling of syntactic elemental changes (such as adding an attribute or demoting an index attribute) to more model-directed semantic handling of change (such as propagating changes in a conceptual model to a database schema) [4]. Research has also moved from schema evolution to the more complex problem of providing versions of schema.

## CROSS REFERENCE*

- •schema versioning
- •temporal algebras
- •conceptual modelling
- •temporal query languages

## RECOMMENDED READING

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[2] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The gemstone data management system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, New York, 1989.

[3] D. de Vries and J. F. Roddick. The case for mesodata: An empirical investigation of an evolving database system. *Information and Software Technology*, 49(9-10):1061–1072, 2007.

[4] J.-M. Hick and J.-L. Hainaut. Database application evolution: A transformational approach. *Data and Knowledge Engineering*, Article in Press, Preprint.

[5] R. Hull. Relative information capacity of simple relational database schemata. *Society for Industrial and Applied Mathematics*, 15(3):856–886, 1986.

[6] W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In F. Bancilhon and D. DeWitt, editors, *14th International Conference on Very Large Data Bases, VLDB'88*, pages 148–159, Los Angeles, CA, 1988. Morgan Kaufmann.

[7] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: a programming platform for generic model management. In *2003 ACM SIGMOD International Conference on Management of data*, pages 193–204, San Diego, California, 2003. ACM Press.

[8] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In R. Agrawal, S. Baker, and D. Bell, editors, *19th International Conference on Very Large Data Bases, VLDB'93*, pages 120–133, Dublin, Ireland, 1993. Morgan Kaufmann.

[9] Y. G. Ra and E. A. Rundensteiner. A transparent schema-evolution system based on object-oriented viewtechnology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, 1997.

[10] J. F. Roddick. SQL/SE - a query language extension for databases supporting schema evolution. *SIGMOD Record*, 21(3):10–16, 1992.

[11] B. Shneiderman and G. Thomas. An architecture for automatic relational database system conversion. *ACM Transactions on Database Systems*, 7(2):235–257, 1982.

[12] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.

[13] L. Tan and T. Katayama. Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *1st International Conference on Deductive and Object-Oriented Databases, DOOD'89*, pages 241–258, Kyoto, Japan, 1989. North-Holland.

# SCHEMA VERSIONING

John F. Roddick, Flinders University, Australia

## DEFINITION

Schema versioning deals with the need to retain current data, and the ability to query and update it, through alternate database structures[1]. **Schema Versioning** requires not only that data is not lost in schema transformation but also requires that all data is able to be queried, both retrospectively and prospectively, through user-definable version interfaces. **Partial schema versioning** is supported when data stored under any historical schema may be viewed through any other schema but may only be updated through one specified schema version - normally the current or active schema[2].

## HISTORICAL BACKGROUND

Multiple versions of a database schema may exist for a number of reasons. First, as a result of changes in system functionality and the external environment, the structure of a database system might change over time but the historical shape of the database might need to be retained. Second, future versions might be created to develop and test later versions of a system. Third, more than one schema may be required in parallel to access the same data in a number of ways. Temporal databases, because of their requirement to maintain the context of historical information, are particularly affected by schema change.

The idea of schema versioning was introduced in the context of OODBs with a number of systems implementing techniques to handle multiple schema (such as Encore [12], Gemstone [7] and Orion [1]), including those that might be required for reasons other than simple historical succession. For example, parallel, alternate schema might be required to conceptualise an idea from a number of semantically consistent but different perspectives. In particular, polymorphism was suggested as a mechanism for providing some stability when faced with changing schema [6].

In order to maintain long-established concepts such as soundness and completeness, algebraic extensions have also been discussed [3]. More recently, schema versioning has also been considered in the context of spatio-temporal databases [9] and meta-data management [2, 4].

## SCIENTIFIC FUNDAMENTALS

Schema versioning is closely related to the concepts of schema integration and data integration – all deal with the problems of accessing data through schema that were not used when the data were originally stored. However, the idea of maintaining multiple schemata, and allowing data to be accessed through them, raises a number of issues.

What is the significance of a difference between two schema (or two databases) and therefore what is the informational cost of the change?

- What are the atomic operations of schema translation or transformation and what happens to the data during these operations?
- Are there any modelling techniques that can be used?
- Are there any other side-effects or opportunities (for instance in query language support)?

[1] The structure of a database is held in a *schema* (pl. *schemata* or *schemas*). Commonly, particularly in temporal databases, these schemata represent the historical structure of a database but this may not always be the case.

[2] Schema evolution and schema versioning has been conflated in the literature with the two terms occasionally being used interchangeably. Readers are thus also encouraged to read also the entry for **Schema Evolution**.

## Types of schema evolution

As outlined elsewhere (qv. schema evolution) four forms of schema evolution can be envisaged - attribute, domain, relation and key evolution. Moreover, one change may involve more than one type of evolution, such as changing the domain of a key attribute and may also be reflected in the conceptual model of the system. Importantly for schema versioning, the inverse function for each of these must be considered. For example, when a schema (merely) evolves by vertically splitting a relation in two with data is suitably transformed, for schema versioning to be allowed, active transformation functions must be provided if the old schema is still to be utilised.

## Practical and Theoretical Limits of Schema Versioning

It has been shown that in order to update data stored under two different schemata using the opposite schemata, they must have equivalent information capacity – all valid instances of some schema $S_1$ must be able to be stored under $S_2$ and vice-versa [5]. Specifically, $S_1 \equiv S_2$ iff $I(S_1) \rightarrow I(S_2)$ is bijective where $I(S)$ is the set of all valid instances of $S$. This means that, in theory, *full* schema versioning across nonequivalent versions of a schema is unattainable and much research in the area adopts the weaker concept of *partial schema versioning* in which data stored under any historical schema may be viewed through any other schema but may only be updated through one specified schema version - normally the current or active schema.

However, in practice, many schema changes that expand or reduce the information capacity of a schema can be done without loss of information. This is the case, for example, for domains defined too large for any of the data, for the creation of subclass relations from a single relation where the subclass type attribute already exists. It is a common practice, where there is some ambiguity in the requirements definition of a system, to allow for a larger schema capacity - some of which may never materialise and, as a result, changes to schemata to adhere to the data actually collected are not uncommon. For example, allowing for time and date when only date is recorded in practice.

Thus the limits for *practical schema versioning* in a database $\mathfrak{D}$ are that $S_1 \stackrel{p}{\equiv} S_2$ ($S_1$ and $S_2$ have practical equivalent information capacity) iff $I'(\mathfrak{D}|S_1) \rightarrow I'(\mathfrak{D}|S_2)$ is bijective where $I'(\mathfrak{D}|S_n)$ is the set of all instances of $S_n$ inferrable from $\mathfrak{D}$ given the constraints of $S_n$. This means that whether the integration of two schema is possible is dependent on the data held as well as the schema definition and while this makes the ability to undertake wholesale change less predictable, it may provide an acceptable level of support in many practical situations.

## Completed Schemas

In order to make all data for a relation available without the need to issue multiple queries, each targetting different time periods, the concept of a *completed schema*, $C$, can be employed that includes all attributes that have ever been defined over the life of a relation. The domain of each attribute in $C$ is considered syntactically general enough to hold all data stored under every version of the relation and the implicit primary key of $C$ is defined as the maximal set of key attributes for the relation over time. Depending on the mechanism used to implement schema versioning, the completed schema can then be used by a series of view functions. For example, in Figure 1, $V_{t_5}$ maps the completed schema $C$ to a subset of the attributes in a schema $S_{t_5}$ active during $t_5$. A converse view function $W_{t_2}$ maps from $S_{t_2}$ to $C$.

Thus the data stored during $t_2$ may be mapped to the format specified during $t_5$ through invocation of $V_{t_5}(W_{t_2}(S_{t_2}))$.

## Query language support

Support for schema versioning does not yet exist in commercially available query languages. However, the TSQL2 proposal [10] and an earlier SQL/SE proposal [8] outlined some parts of the solution. As examples of such extensions:

reference to the *completed schema* can be included to provide access to all data;

- the specification of the schema could be done either through the specification of a global *schema-time* as in TSQL2, which would be useful for SQL embedded in a program with the schema-time set to compile time, or explicitly as part of the query;
- attribute definition might be able to be tested by adding a test to see if a value was missing because it was

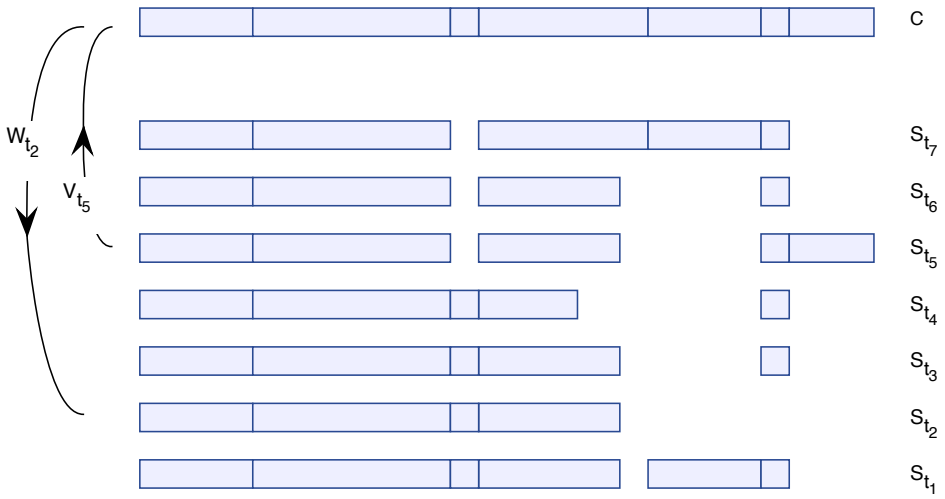



Figure 1: Versions of Schemata over time

*not defined* rather than being merely *null*;

- the language may also include meta-data queries such as the ability to ask what version of the schema a given piece of data adheres to.

## Instance Amendment

For schema versioning in which the old schemata are still considered valuable, once a schema change is accepted, there are three options regarding the change to existing data. First, the underlying instances may be coerced to the new structure. While conceptually simple, this may result in lost information and an inability to reverse schema amendments. Secondly, data is retained in the format in which it was originally stored. This retains information content at the expense of more complex (and slower) translation of data when needed. Thirdly, data is initially retained in the format in which it was originally stored but is converted when amended. While the most complex option, it has the advantage of identifying data that has not been amended since the schema change.

## FUTURE DIRECTIONS

Schema versioning research has moved from low-level handling of syntactic elemental changes to more model-directed semantic handling of change. There are a number of other issues that make schema versioning non-trivial. Some of these represent future issues to be investigated.

Many schema change requirements involve composite operations and thus a mechanism for schema level commit and rollback functions could be envisaged which could operate at a higher level to the data level commit and rollback operations.

- Access rights considerations are particularly a problem in object-oriented database systems. Consider, for example, a change to a class (eg. Employees) from which attributes are inherited to a sub-class (eg. Engineers) for which the modifying user has no legitimate access. Any change to the definition of attributes inherited from the superclass can be considered to violate the access rights of the subclass. Moreover, in some systems ownership of a class does not imply ownership of all instances of that class.
- In temporal databases the concept of vacuuming (q.v.) allows for the physical deletion of temporal data in cases where the utility of holding the data is outweighed by the cost of doing so [11]. Similar consideration must be given to the deletion of obsolete schema definitions, especially in cases where no data exists adhering to either that version (physically) or referring, through its transaction-time values, to the period in which the definition was active.

## CROSS REFERENCE*

- •temporal evolution
- •temporal algebras
- •conceptual modelling
- •temporal query languages

## RECOMMENDED READING

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] W. Kim, N. Ballou, H.-T. Chou, J. F. Garza, and D. Woelk. Features of the orion object-oriented database system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. ACM Press, New York, 1989.

[2] J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In J. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, editors, *29th International Conference on Very Large Data Bases (VLDB)*, pages 572–583, Berlin, Germany, 2003. Morgan Kaufmann.

[3] L. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[4] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: a programming platform for generic model management. In *2003 ACM SIGMOD International Conference on Management of data*, pages 193–204, San Diego, California, 2003. ACM Press.

[5] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In R. Agrawal, S. Baker, and D. Bell, editors, *19th International Conference on Very Large Data Bases, VLDB'93*, pages 120–133, Dublin, Ireland, 1993. Morgan Kaufmann.

[6] S. Osborn. The role of polymorphism in schema evolution in an object-oriented database. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):310–317, 1989.

[7] D. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. *OOPSLA '87 (SIGPLAN Notices)*, 22(12):111–117, 1987.

[8] J. F. Roddick. SQL/SE - a query language extension for databases supporting schema evolution. *SIGMOD Record*, 21(3):10–16, 1992.

[9] J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas. Beyond schema versioning: A flexible model for spatio-temporal schema selection. *Geoinformatica*, 5(1):33–50, 2001.

[10] J. F. Roddick and R. Snodgrass. Schema versioning support. In R. Snodgrass, editor, *The TSQL2 Temporal Query Language*, pages Ch. 22. 427–449. Kluwer Academic Publishing, Boston, 1995.

[11] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Data and Knowledge Engineering*, 44(1):1–29, 2003.

[12] S. Zdonik. Version management in an object-oriented database. volume 244 of *LNCS*, pages 405–422. Springer, Berlin, 1986.

# Sequenced Semantics

Michael H. Böhlen and Christian S. Jensen
Free Univerity of Bozen-Bolzano, Italy and Aalborg University, Denmark

## SYNONYMS

none

## DEFINITION

Sequenced semantics makes it possible to generalize a query language statement on a non-temporal database to a temporal query on a corresponding temporal, interval-timestamped database by applying minor syntactic modifications to the statement that are independent of the particular statement. The semantics of such a generalized statement is consistent with considering the temporal database as being composed of a sequence of non-temporal database states. Sequenced semantics takes into account the interval timestamps of the argument tuples when forming the interval timestamps associated with result tuples, as well as permits the use of additional timestamp-related predicates in statements.

## MAIN TEXT

A question that has intrigued temporal database researchers for years is how to systematically generalize non-temporal query language statements, i.e., queries on non-temporal databases, to apply to corresponding temporal databases. A prominent approach is to view a temporal database as a sequence of non-temporal databases. Then a non-temporal statement is rendered temporal by applying it to each non-temporal database, followed by integration of the non-temporal results into a temporal result. Sequenced semantics formalizes this approach and is based on three concepts: S-reducibility, extended S-reducibility, and interval preservation. These topics are discussed in turn.

Ensuing examples assume a database instance with three relations:

`Employee`

| ID | Name | VTIME |
|---|---|---|
| 1 | Bob | 5−8 |
| 3 | Pam | 1−3 |
| 3 | Pam | 4−12 |
| 4 | Sarah | 1−5 |

`Salary`

| ID | Amt | VTIME |
|---|---|---|
| 1 | 20 | 4−10 |
| 3 | 20 | 6−9 |
| 4 | 20 | 6−9 |

`Bonus`

| ID | Amt | VTIME |
|---|---|---|
| 1 | 20 | 1−6 |
| 1 | 20 | 7−12 |
| 3 | 20 | 1−12 |

**S-Reducibility** S-reducibility states that the query language of the temporally extended data model must offer, for each query $q$ in the non-temporal query language, a *syntactically similar temporal query* $q^t$ that is its natural generalization, i.e., $q^t$ is snapshot reducible to $q$, and $q^t$ is syntactically identical to $S_1 q S_2$. The goal is to make the semantics of temporal queries easily understandable in terms of the semantics of the corresponding non-temporal queries. The strings $S_1$ and $S_2$ are independent of $q$ and are termed *statement modifiers* because they change the semantics of the entire statement $q$ that they enclose.

In the following examples, statements are prefixed with the modifier `SEQ VT` [2]. This modifier tells the temporal DBMS to evaluate statements with sequenced semantics in the valid-time dimension. These examples illustrate that S-reducible statements are easy to write and understand because they are simply conventional SQL statements with the additional prefix `SQL VT`. Writing statements that compute the same results, but without using statement modifiers, can be very difficult [3].

```
SEQ VT SELECT * FROM Employee;
SEQ VT
  SELECT ID FROM Employee AS E
  WHERE NOT EXISTS (SELECT * FROM Salary AS S WHERE E.ID = S.ID);
```

The first query returns all `Employee` tuples together with their valid time—this corresponds to returning the

content of `Employee` at each state. The second query determines the time periods when an employee did not get a salary. It returns $\{\langle 3, 1-3\rangle, \langle 3, 4-5\rangle, \langle 3, 10-12\rangle, \langle 4, 1-5\rangle\}$. Conceptually the enclosed statement is evaluated on each state of the database. Computationally, the interval $6-9$ is subtracted from the interval $4-12$ to get the intervals $4-5$ and $10-12$.

**Extended S-Reducibility** S-reducibility is applicable only to queries of the underlying non-temporal query language and does not extend to queries explicit references to time. Consider the following queries.

```
SEQ VT
  SELECT E.ID
  FROM Employee AS E, Salary AS S
  WHERE E.ID = S.ID
  AND DURATION(VTIME(E)) > DURATION(VTIME(S))
```

```
SEQ VT
  SELECT E.ID, VTIME(S), VTIME(E)
  FROM  Employee AS E, Salary AS S
  WHERE E.ID = S.ID
```

The query to the left constrains the temporal join to tuples in `Employee` with a valid time that is longer than the valid time of the salary tuple it shall be joined with. This condition cannot be evaluated on individual non-temporal relation states because the timestamp is not present in these states. Nevertheless, the temporal join itself can still be conceptualized as a non-temporal join evaluated on each snapshot, with an additional predicate. The query to the right computes a temporal join as well, but also returns the original valid times. Again the semantics of this query fall outside of snapshot reducibility because the original valid times are not present in the non-temporal relation states.

DBMSs generally provide predicates and functions on time attributes, which may be applied to, e.g., valid time, and queries such as these arise naturally. Applying sequenced semantics to statements that include predicates and functions on time offers a higher degree of orthogonality and wider ranging temporal support.

**Interval Preservation** Coupling snapshot reducibility with syntactical similarity and using this property as a guideline for how to semantically and syntactically embed temporal functionality in a language is attractive. However, S-reducibility does not distinguish between different relations if they are snapshot equivalent. This means that different results of an S-reducible query are possible: the results will be snapshot equivalent, but will differ in how the result tuples are timestamped. As an example, consider a query that fetches and displays the content of the `Bonus` relation. An S-reducible query may return the result $\{\langle 1, 20, 1\text{–}6\rangle, \langle 1, 20, 7\text{–}12\rangle, \langle 3, 20, 1\text{–}12\rangle\}$. If Bob received a 20K bonus for his performance during the first half of the year and another 20K bonus for his performance during the second half of the year and Pam received a 20K bonus for her performance during the entire year, this is the expected result. This is also the result supported by the three tuples in the example instance displayed above. However, S-reducibility does not distinguish this result from any other snapshot equivalent result. With S-reducibility a perfectly equivalent result would be $\{\langle 1, 20, 1-12\rangle, \langle 3, 20, 1-6\rangle, \langle 3, 20, 7-12\rangle\}$.

Interval preservation settles the issue of which result should be favored out of the many possible results permitted by S-reducibility. When defining how to timestamp tuples of query results, two possibilities come to mind. Results can be coalesced. This solution is attractive because it defines a canonical representation for temporal relations. A second possibility is to consider lineage and preserve, or respect, the timestamps as originally entered into the database [1]. Sequenced semantics requires that the default is to preserve the timestamps—being irreversible, coalescing cannot be the default.

## CROSS REFERENCE

Non-Sequenced Semantics, Snapshot Equivalence, Temporal Coalescing, Time Interval, Valid Time

## REFERENCES

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.


[1] M. H. Böhlen, R. Busatto, and C. S. Jensen. Point-Versus Interval-Based Temporal Data Models. In *Proceedings of the 14th International Conference on Data Engineering*, pages 192–200, Orlando, Florida, February 23–27 1998. IEEE Computer Society.

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

[3] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

# SNAPSHOT EQUIVALENCE

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Temporally Weak; Weak Equivalence

## DEFINITION

Informally, two tuples are *snapshot equivalent* or *weakly equivalent* if all pairs of timeslices with the same time instant parameter of the tuples are identical.

Let temporal relation schema $R$ have $n$ time dimensions, $D_i$, $i = 1, \ldots, n$, and let $\tau^i$, $i = 1, \ldots, n$ be corresponding timeslice operators, e.g., the valid timeslice and transaction timeslice operators. Then, formally, tuples $x$ and $y$ are snapshot equivalent if

$$\forall t_1 \in D_1 \ldots \forall t_n \in D_n(\tau^n_{t_n}(\ldots(\tau^1_{t_1}(x))\ldots) = \tau^n_{t_n}(\ldots(\tau^1_{t_1}(y))\ldots))$$

Similarly, two relations are snapshot equivalent or weakly equivalent if at every instant their snapshots are equal. Snapshot equivalence, or weak equivalence, is a binary relation that can be applied to tuples and to relations.

## MAIN TEXT

The notion of weak equivalence captures the information content of a temporal relation in a point-based sense, where the actual timestamps used are not important as long as the same timeslices result. For example, consider the two relations with just a single attribute: {(a, [3, 9]} and {(a, [3, 5]), (a, [6, 9])}. These relations are different, but snapshot equivalent.

Both "snapshot equivalent" and "weakly equivalent" are being used in the temporal database community. "Weak equivalence" was originally introduced by Aho et al. in 1979 to relate two algebraic expressions [1,2]. This concept has subsequently been covered in several textbooks. One must rely on the context to disambiguate this usage from the usage specific to temporal databases. The synonym "temporally weak" does not seem intuitive—in what sense are tuples or relations weak?

## CROSS REFERENCE*

Temporal Database, Time Instant, Timeslice Operator, Transaction Time, Valid Time, Weak Equivalence

## REFERENCES*

[1] Alfred V. Aho, Yehoshua Sagiv, Jeffrey D. Ullman: Efficient Optimization of a Class of Relational Expressions. ACM Trans. Database Syst. 4(4): 435-454 (1979)

[2] Alfred V. Aho, Yehoshua Sagiv, Jeffrey D. Ullman: Equivalences Among Relational Expressions. SIAM J. Comput. 8(2): 218-246 (1979)

[3] S. K. Gadia, "Weak Temporal Relations," in *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, MA, 1985, pp. 70-77.

[4] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# SQL-Based Temporal Query Languages

Michael Böhlen, Johann Gamper
Free University of Bozen-Bolzano, Italy
www.inf.unibz.it/~{boehlen,gamper}

Christian S. Jensen
Aalborg University, Denmark
www.cs.aau.dk/~csj

Richard T. Snodgrass
University of Arizona
www.cs.arizona.edu/people/rts

## SYNONYMS

none

## DEFINITION

More than two dozen extensions to the relational data model have been proposed that support the storage and retrieval of time-referenced data. These models timestamp tuples or attribute values, and the timestamps used include time points, time periods, and finite unions of time periods, termed temporal elements.

A temporal query language is defined in the context of a specific data model. Most notably, it supports the specification of queries on the specific form of time-referenced data provided by its data model. More generally, it enables the management of time-referenced data.

Different approaches to the design of a temporal extension to the Structured Query Language (SQL) have emerged that yield temporal query languages with quite different design properties.

## HISTORICAL BACKGROUND

A number of past events and activities that included the temporal database community at large had a significant impact on the evolution of temporal query languages. The 1987 *IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems* [7] covered topics such as requirements for temporal data models and information systems, temporal query languages, versioning, implementation techniques, as well as temporal logic, constraints, and relations to natural language.

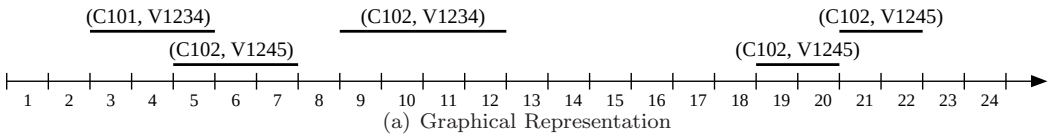
The 1993 *ARPA/NSF International Workshop on an Infrastructure for Temporal Databases* [8] gathered researchers in temporal databases with the goal of consolidating the different approaches to temporal data models and query languages. In 1993, the influential collection *Temporal Databases: Theory, Design, and Implementation* [11] was also published. This collection describes a number of data models and query languages produced during the previous ten years of temporal database research.

Year 1995 saw the publication of the book *The TSQL2 Temporal Query Language* [9]. TSQL2 represents an effort to design a consensus data model and query language, and it includes many of the concepts that were proposed by earlier temporal data models and query languages. In 1995, the *International Workshop on Temporal Databases* [3] was co-located with the VLDB conference.

Then, in 1996, *SQL/Temporal: Part 7 of SQL3* was accepted. This was the result of an effort aimed at transferring results of temporal database research into SQL3. The first step was a proposal of a new part to SQL3, termed SQL/Temporal, which included the PERIOD data type. In 1997, the *Dagstuhl seminar on Temporal Databases* took place [5]. Its goal was to discuss future directions for temporal database management, with respect to both research issues and the means to incorporate temporal databases into mainstream application development.

## SCIENTIFIC FUNDAMENTALS

A discrete and totally ordered time domain is assumed that consists of time instants/points. The term (time) "period" is used to denote a convex subset of the time domain. The term (time) "interval" then denotes a duration of time, which coincides with its definition in SQL. As a running example, the temporal relation *Rental* in Figure 1(a) is used, which records car rentals, e.g., customer C101 rents vehicle V1234 from time 1 to time 3. Figures 1(b)–(e) show different representations of this relation, using the strong period-based, weak period-based, point-based, and parametric model, respectively. (In all the example relation instances, the conventional attribute(s) are separated from the timestamp attribute(s) with a vertical line.) The following queries together with their intended results build on the car rental example. These will serve for illustration.



(a) Graphical Representation

| CustID | VID | T |
|---|---|---|
| C101 | V1234 | [3,5] |
| C102 | V1245 | [5,7] |
| C102 | V1234 | [9,12] |
| C102 | V1245 | [19,20] |
| C102 | V1245 | [21,22] |

(b) Strong Period Model

| CustID | VID | T |
|---|---|---|
| C101 | V1234 | [3,5] |
| C102 | V1245 | [5,7] |
| C102 | V1234 | [9,12] |
| C102 | V1245 | [19,22] |

(c) Weak Period Model

| SeqNo | CustID | VID | T |
|---|---|---|---|
| 1 | C101 | V1234 | 3 |
| 1 | C101 | V1234 | 4 |
| 1 | C101 | V1234 | 5 |
| 2 | C102 | V1245 | 5 |
| | ... | | |

(d) Point Model

| SeqNo | | CustID | | VID | |
|---|---|---|---|---|---|
| [3, 5] | 1 | [3, 5] | C101 | [3, 5] | V1234 |
| [5, 7] | 2 | [5, 7] | C102 | [5, 7] | V1234 |
| [9, 12] | 3 | [9, 12] | C102 | [9, 12] | V1245 |
| [19, 20] | 4 | [19, 20] | C102 | [19, 20] | V1245 |
| [21, 22] | 5 | [21, 22] | C102 | [21, 22] | V1245 |

(e) Parametric Model: Grouping on `SeqNo`

Figure 1: Temporal Relation *Rental*

Q1: *All rentals that overlap the time period* $[7, 9]$. Query $Q1$ asks for all available information about rentals that overlap the period $[7, 9]$.

| CustID | VID | T |
|---|---|---|
| C102 | V1245 | [5,7] |
| C102 | V1234 | [9,12] |

Q2: *All 2-day rentals.* This query constrains the number of time points included in a time period and teases out the difference between the use of time points versus periods.

| VID | T |
|---|---|
| V1245 | [19,20] |
| V1245 | [21,22] |

Q3: *How many vehicles have been rented?* This is an example of an ordinary query that must be applied to each state of a temporal database. The non-temporal query is an aggregation. Thus, the result at a specific time point is computed over all tuples that are valid at that time point. (Note that some query languages don't return tuples when there is no data, e.g., when the `Cnt` is 0.)

| Cnt | T |
|---|---|
| 1 | [3,4] |
| 2 | [5,5] |
| 1 | [6,7] |
| 0 | [8,8] |
| 1 | [9,12] |
| 0 | [13,18] |
| 1 | [19,20] |
| 1 | [21,22] |

Q4: *How many rentals were made in total?* This is another aggregation query; however, the aggregation is to be applied independently of any temporal information.

| Cnt |
|---|
| 5 |

Q5: *List all (*current*) rentals.* This query refers to the (constantly moving) current time. It is assumed that the current time is 5.

| CustID | VID |
|---|---|
| C101 | V1234 |
| C102 | V1245 |

**Approach I: Abstract Data Types — SQL/ATD.** The earliest and, from a language design perspective, simplest approaches to improving the temporal data management capabilities of SQL have simply introduced time data types and associated predicates and functions. This approach is illustrated on the `Rental` instance in Figure 1(b).

$Q1^{\text{SQL/ATD}}$ : `select * from Rental where T overlaps [7,9]`
$Q2^{\text{SQL/ATD}}$ : `select VID, T from Rental where duration(T) = 2`
$Q4^{\text{SQL/ATD}}$ : `select count(*) as Cnt from Rental`
$Q5^{\text{SQL/ATD}}$ : `select CustID, VID from Rental where T overlaps [now,now]`

The predicates on time-period data types available in query languages have been influenced by Allen's 13 period relationships [1], and different practical proposals for collections of predicates exist. For example, the overlaps predicate (as defined in the TSQL2 language) can be used to formulate Query Q1. Predicates that limit the duration of a period (Q2) and retrieve current data (Q5) follow the same approach.

Expressing the time-varying aggregation of Q3 in SQL is possible, but exceedingly complicated and inefficient. The hard part is that of expressing the computation of the periods during which the aggregate values remain constant. (This requires about two dozen lines of SQL with nested `NOT EXISTS` subqueries [10, pp. 165–6].) In contrast, counting the rentals independently of the time references is easy, as shown in Q4.

Adding a new ADT to SQL has limited impact on the language design, and extending SQL with new data types with accompanying predicates and functions is relatively simple and fairly well understood. The approach falls short in offering means of conveniently formulating a wide range of queries on period timestamped data, including temporal aggregation. It also offers no systematic way of generalizing a simple snapshot query to becoming time-

varying. Shortcomings such as these motivate the consideration of other approaches.

**Approach II: Folding and Unfolding — IXSQL.** Another approach is to equip SQL with the ability to normalize timestamps. Advanced most prominently by Lorentzos [4, 6] in the IXSQL language, the earliest and most radical approach is to introduce two functions: `unfold`, which decomposes a period timestamped tuple into a set of point-timestamped tuples, and `fold`, which "collapses" a set of point-timestamped tuples into value-equivalent tuples timestamped with maximum periods. The general pattern for queries is then: (i) construct the point-based representation by unfolding the argument relation(s), (ii) compute the query on the period-free representation, and (iii) fold the result to obtain a period-based representation. The `Rental` relation in Figure 1(b) is assumed.

$Q3^{\text{IXSQL}}$ :
```
select count(*) as Cnt, T
from (select * from Rental reformat as unfold T)
group by T reformat as fold T
```

The IXSQL formulations of Q1, Q2, Q4, and Q5 are essentially those of the ADT approach (modulo minor syntactic differences); specifically, normalization is not needed. The `fold` and `unfold` functions become useful for the temporal aggregation in Q3. The inner query unfolds the argument relation, yielding the point-based representation in Figure 1(d), on which the aggregation is computed. The `fold` function then transforms the result back into a period-stamped relation, which, however, is different from the intended result because the last two tuples are merged into a single tuple $(1, [19, 22])$. The combination of unfolding and folding yields maximal periods of snapshot equivalent tuples and does not carry over any lineage information.

SQL with folding and unfolding is conceptually simple and offers a systematic approach to formulating at least some temporal queries, including temporal queries that generalize non-temporal queries. It obtains the representational benefits of periods while avoiding the potential problems they pose in query formulation, since the temporal data is manipulated in point-stamped form. The `fold` and `unfold` functions preserve the information content in a relation only up to that captured by the point-based perspective; thus, lineage information is lost. This leaves some "technicalities" (which are tricky at times) to be addressed by the application programmer.

**Approach III: Point Timestamps — SQL/TP.** A more radical approach to designing a temporal query language is to simply assume that temporal relations use point timestamps. The temporal query language SQL/TP advanced by Toman [12] takes this approach to generalizing queries on non-temporal relations to apply to temporal relations. The point timestamped `Rental` relation in Figure 1(d) is assumed in the following.

$Q1^{\text{SQL/TP}}$ :
```
select distinct a.* from Rental a, Rental b
where a.SeqNo = b.SeqNo and (b.T = 7 or b.T = 8 or b.T = 9)
```
$Q2^{\text{SQL/TP}}$ :
```
select SeqNo, VID, T from Rental group by SeqNo having count(T) = 2
```
$Q3^{\text{SQL/TP}}$ :
```
select count(*) as Cnt, T from Rental group by T
```
$Q4^{\text{SQL/TP}}$ :
```
select count(distinct SeqNo) as Cnt from Rental
```
$Q5^{\text{SQL/TP}}$ :
```
select CustID, VID from Rental where T = now
```

Q1 calls for a comparison of neighboring database states. The point-based perspective, which separates the database states, does not easily support such queries, and a join is needed to report the original rental periods. The `distinct` keyword removes duplicates that are introduced if a tuple shares more than one time point with the period $[7, 9]$.

Duration queries, such as Q2, are formulated as aggregations and require an attribute, in this case `SeqNo`, that distinguishes the individual rentals. The strength of SQL/TP is in its generalization of queries on snapshot relations to queries on temporal relations, as exemplified by Q3. The general principle is to extend the snapshot query to separate database snapshots, which here is done by the grouping clause. SQL/TP and SQL are opposites when it comes to the handling of temporal information. In SQL, *time-varying* aggregation is poorly supported, while SQL/TP needs an additional attribute that identifies the real-world facts in the argument relation to support *time-invariant* aggregation (Q4).

The restriction to time points ensures a simple and well-defined semantics that avoids many of the pitfalls that can be attributed to period timestamps. As periods are still to be used in the physical representation and user interaction, one may think of SQL/TP as a variant of IXSQL where, conceptually, queries must always apply `unfold` as the first operation and `fold` as the last. To express the desired queries, an identifying attribute (e.g., `SeqNo`) is often needed. Such identifiers do not offer a systematic way of obtaining point-based semantics *and*

a semantics that preserves the periods of the argument relations. The query *"When was vehicle V1245, but not vehicle V1234, rented?"* illustrates this point. A formulation using the temporal difference between the timestamp attributes does not give the expected answer $\{[6,7],[19,20],[21,22]\}$ because the sequence number is not included. If the sequence number is included, the difference is effectively disabled. This issue is not only germane to SQL/TP, but applies equally to all approaches that use a point-based data model.

**Approach IV: Syntactic Defaults — TSQL2.** What may be viewed as syntactic defaults have been introduced to make the formulation of common temporal queries more convenient. The most comprehensive approach based on syntactic defaults is TSQL2 [9]. As TSQL2 adopts a point-based perspective, the `Rental` instance in Figure 1(c) is assumed, where the periods are a shorthand representation of time points.

$Q1^{\text{TSQL2}}$ : `select * from Rental where valid(Rental) overlaps period '7-9'`
$Q2^{\text{TSQL2}}$ : `select SeqNo, VID from Rental where cast(valid(Rental) as interval) = 2`
$Q3^{\text{TSQL2}}$ : `select count(*) as Cnt from Rental group by valid(Rental) using instant`
$Q4^{\text{TSQL2}}$ : `select snapshot count(*) as Cnt from Rental`
$Q5^{\text{TSQL2}}$ : `select snapshot * valid(date 'now') from Rental`

In TSQL2, a `valid` clause, which by default is present implicitly after the `select` clause, computes the intersection of the valid times of the relations in the `from` clause, which is then returned in the result. With only one relation in the `from` clause, this default clause yields the original timestamps as exemplified in Q1 and Q2. The cast function in Q2 maps between periods (e.g., $[7-9]$) and intervals (e.g., 4 days). The argument relation must be augmented by the `SeqNo` attribute (thus obtaining a relation with 5 tuples, as in Figure 1(b)) for this query to properly return the 2-day rentals.
The default behavior of the implicit `valid` clause was designed with snapshot reducibility in mind, which shows nicely in the instant temporal aggregation query Q3. The grouping is performed according to the time points, not the original timestamps returned by `valid(Rental)`. The `using instant` is in fact the default and could be omitted (added for clarity). As TSQL2 returns temporal relations by default, the `snapshot` keyword is used in queries Q4 and Q5 to retrieve non-temporal relations.
Well-chosen syntactic defaults yield a language that enables succinct formulation of common temporal queries. However, adding temporal support to SQL in this manner is difficult since the non-temporal constructs do not permit a systematic and easy way to express the defaults. It is challenging to be comprehensive in the specification of such defaults and to ensure that they do not interact in unattractive ways. Thus, syntactic defaults lack "scalability" over language constructs.

**Approach V: Statement Modifiers — ATSQL.** ATSQL [2] introduces temporal statement modifiers to offer a systematic means of constructing temporal queries from non-temporal queries. A temporal query is formulated by first formulating the corresponding non-temporal query and then prepending this query with a statement modifier that tells the database system to use temporal semantics. In contrast to syntactic defaults, statement modifiers are semantic in that they apply in the same manner to any statement they modify. The strong period-timestamped `Rental` instance in Figure 1(b) is assumed in the following.

$Q1^{\text{ATSQL}}$ : `seq vt select * from Rental where T overlaps [7,9]`
$Q2^{\text{ATSQL}}$ : `seq vt select VID from Rental where duration(T) = 2`
$Q3^{\text{ATSQL}}$ : `seq vt select count(*) as Cnt from Rental`
$Q4^{\text{ATSQL}}$ : `nseq vt select count(*) as Cnt from Rental`
$Q5^{\text{ATSQL}}$ : `select * from Rental`

Queries Q1 and Q2 can be formulated almost as in SQL. The `seq vt` ("sequenced valid time") modifier indicates that the semantics is consistent with evaluating the non-temporal query on a sequence of non-temporal relations and ensures that the original timestamps are returned. Modifiers also work for queries that use period predicates, such as, e.g., Allen's relations, which cannot be used in languages of point-timestamped data models.
Query Q3 is a temporal generalization of a non-temporal query and can be formulated by prepending the non-temporal SQL query with the `seq vt` modifier. The modifier ensures that at each time point the aggregates are evaluated over all tuples that overlap with that time point. Query Q4 is to be evaluated independently of the time attribute values of the tuples. This is achieved by using the `nseq vt` ("non-sequenced valid time") modifier, which indicates that what follows should be treated as a regular SQL query.

A query without any modifiers considers only the current states of the argument relations, as exemplified by Query Q5. This ensures that legacy queries on non-temporal relations are unaffected if the non-temporal relations are made temporal.
Statement modifiers are orthogonal to SQL and adding them to SQL represents a much more fundamental change to the language than, e.g., adding a new ADT or syntactic defaults. The notion of statement modifiers offers a wholesale approach to rendering a query language temporal: modifiers control the semantics of any query language statement. This language mechanism is independent of the syntactic complexity of the queries that the modifiers are applied to. It becomes easy to construct temporal queries that generalize snapshot queries.

**Approach VI: Temporal Expressions — TempSQL.** The notion of temporal expression was originally advocated by Gadia and is supported in the TempSQL language [11, p. 28ff], which is based on the parametric data model (see Figure 1(e)). Relations in TempSQL consist of tuples with attribute values that are functions from a subset of the time domain to some value domain (specified as a pair of a temporal element, a finite union of time periods, and a value). The functions in the same tuple must have the same domain. The relations are keyed. If a set of attributes is a key, then no two tuples are allowed to exist in the relation that have the same range values for those attributes. Figure 1(e) with the key `SeqNo` is assumed in the following.

$Q1^{\text{TempSQL}}$ : `select * from Rental where ⟦ VID ⟧ ∩ [7,9] ≠ ∅`
$Q2^{\text{TempSQL}}$ : `select VID from Rental where duration(⟦ VID ⟧) = 2`
$Q3^{\text{TempSQL}}$ : `select count(*) as Cnt from Rental`
$Q5^{\text{TempSQL}}$ : `select * from Rental`

Query Q1 and Q2 can be formulated using temporal expressions. If $X$ is an expression that returns a function from time to some value domain then $[\![X]\!]$ is a temporal expression which returns the domain of $X$, i.e., the time when $X$ is true. The result of Q2 is the relation $\{\langle[19, 22]\ \text{V1245}\rangle\}$. For the aggregation query Q3, TempSQL automatically performs an instant temporal aggregation [11, p. 42]. A different query must be used to determine the time-invariant count in Q4. One possibility would be to formulate a query that first drops or equalizes all timestamps and then performs the above aggregation. For so-called current users, TempSQL offers built-in support for accessing the current state of a database, by assuming that the argument relations are the ordinary snapshot relations that contain the current states of the temporal relations. This is exemplified in Q5.
Temporal expressions as used in TempSQL, which return the temporal elements during which a logical expression is true, are convenient and often enable the elegant formulation of queries. Temporal expressions along with temporal elements, fit well into the point-based framework. However, as of yet little research has been done to further explore temporal expressions and to include them into query languages.

## KEY APPLICATIONS

SQL-based temporal query languages are intended for use in database applications that involve the management of time-referenced data. Such applications are found literally in all data management application areas—in fact, virtually all real-world databases contain time-referenced data. SQL-based languages are attractive in comparison to other types of languages because SQL is used by existing database management systems.

## FUTURE DIRECTIONS

While temporal query language support appears to be emerging in commercial systems, comprehensive temporal support is still not available in products.
Much research in temporal query languages has implicitly or explicitly assumed a traditional administrative data management setting, as exemplified by the car rental example. The design of temporal query languages for other kinds of data and applications, e.g., continuous sensor data, has received little attention.

## CROSS REFERENCE

Allen's Relations, Now in Temporal Databases, Period-Stamped Data Models, Point-Stamped Data Models, Temporal Data Model, Temporal Database, Temporal Element, Time Period, Time Interval, Temporal Query Languages, TSQL2, Valid Time

## RECOMMENDED READING


[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
[2] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.

[3] J. Clifford and A. Tuzhilin, editors. *Recent Advances in Temporal Databases, Proceedings of the International Workshop on Temporal Databases*, Workshops in Computing, 1995.

[4] C. J. Date, H. Darwen, and N. Lorentzos, editors. *Temporal Data & the Relational Model*. Morgan Kaufmann Publishers, 2002.

[5] O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[6] N. A. Lorentzos and R. G. Johnson. Extending relational algebra to manipulate temporal data. *Information Systems*, 13(3):289–296, 1988.

[7] C. Rolland, F. Bodart, and M. Lèonard, editors. *Temporal Aspects in Information Systems, Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems*, 1987.

[8] R. T. Snodgrass, editor. *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, 1993.

[9] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[10] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, July 1999.

[11] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, Inc., 1993.

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

**TEMPLATE FOR REGULAR ENTRY**
**(ENCYCLOPEDIA OF DATABASE SYSTEMS)**

## TITLE OF ENTRY
**Supporting Transaction Time Databases**

## BYLINE
David Lomet
Microsoft Research
http://www.research.microsoft.com/~lomet

## SYNONYMS
Temporal database, multi-version database

## DEFINITION
The temporal concepts glossary maintained at http://www.cs.aau.dk/~csj/Glossary/ defines transaction time as: "The *transaction time* of a database fact is the time when the fact is current in the database and may be retrieved." A transaction time database thus stores versions of database records or tuples, each of which has a start time and an end time, delimiting the time range during which they represent the current versions of database facts. Because each version is the result of transactions, the times associated with the version are the times for the transaction starting the version (the start time) and for the transaction ending the version (the end time). These transaction times are required to agree with the serialization order of the transaction so that the database can present a transaction consistent view of the facts being stored.

## HISTORICAL BACKGROUND
Postgres was the first database system that supported transaction time databases [13]. It implemented a prototype relational database system using a versioning approach for recovery. This was then augmented with a version store based on the R-tree [4]. Subsequently, the TSB-tree was introduced [7], based on the WOB-tree time splitting [3], that provided an integrated approach to storing both current and historical data.

Commercially, Oracle and Rdb [5] database systems both support multi-version concurrency control, now called snapshot isolation. Oracle subsequently added a Flashback [11] feature that permitted access to historical versions, based on saving a linear history of their recovery versions. While this permits access to historical versions, it is mostly intended as an efficient means of providing an online backup for "point-in-time" recovery, a form of "media" recovery in which the database is recovered to a point just preceding a bad user transaction.

Temporal databases have been extensively studied [14], including not only transaction time but also valid time. Included as well are bi-temporal databases, where both valid and transaction time are supported. This work has clarified the conceptual issues and provided a common vocabulary of terms for describing work in the field.

## SCIENTIFIC FUNDAMENTALS

Transaction time databases usually offer the full range of database functionality that one would expect of a database storing only current facts, which is called a current time database. In addition, a transaction time database provides access to all prior facts as represented by versions of records that existed at some prior time. Several types of functionality can be envisioned.

1. Access to database facts "as of" some past time, which are called "as of" queries.
2. Access to versions of a database fact in some time range, which are called "time travel" queries.
3. Access to collections of database facts in some time range, i.e., "general transaction time queries".

The functionality provided by transaction time databases is important for applications such as time series data, regulatory compliance, repeatability of scientific experiments, etc. Further, a transaction time database can provide valuable system capabilities such as snapshot isolation concurrency control, recovery from bad user transactions, and recovery from media failure.

To support transaction time functionality, one needs to change the semantics of the data manipulation operations "update" and "delete". An update creates an additional version instead of overwriting the prior state, retaining both new and old versions of the data. A delete is strictly logical, providing an end time for the previously current version. In this way, the prior version persists so that queries about the database while this version was alive can be answered.

Database systems supporting transaction time data have been built that are based on the relational data model, though this is not a fundamental limitation, simply a pragmatic choice.

### Implementation Approaches

There are two generic approaches to implementing transaction time databases, which are described here.

**Layered Approach:** The premise of the layered approach is that application programmers cannot wait for vendors to build transaction time functionality into database products [15]. Rather, a middleware layer (MWL) is implemented that provides this functionality. The MWL processes data definition statements, adding timestamp fields to each record, processes data manipulation statements (queries plus updates) written in a language that exposes temporal functionality and translates them into equivalent ordinary SQL queries. Typically, start time and end time are added to each record of a transaction time table. The table may be organized by a clustering key that includes (user defined primary key, start time, end time). Thus, a record with a given user defined primary key is clustered next to its earlier versions, which makes "time travel" queries efficient, while implying that "as of" queries will have relatively poor performance. Grouping by time does not help much with "as of" performance, and usually compromises "time travel" performance.

**Built-in Approach:** The built-in approach requires the ability to modify the database engine to provide transaction time support. While a significant barrier, building transaction time support into a database system can greatly improve performance, bringing it close to current time database performance. This improved performance is the result of optimizations that are possible (1) for update, when the timestamps need to be added to versions, (2) for storage with simple forms of version compression, and (3) for query, because specialized indexing is

possible that improves data clustering. The rest of this article discusses the issues of built-in support and how to make this support perform well.

## Managing Versions

Transaction time functionality requires dealing with the multiple versions of database facts that exist to express the states of the database over time. The built-in approach has more freedom than the MWL approach in how to achieve efficiency and performance. There are a number of issues, the more important ones being:

**Timestamps:** Each historical version stored in a transaction time database has a begin time, at which it first became the current version, and an end time, at which it was either replaced by another version or deleted. Current versions have the usual start time, but have a special end time called "now" [2]. An MWL approach usually includes both start and end times with each version so that the SQL queries resulting from translating temporal queries are simple and efficient. With built-in support, most systems [13, 6] store only the start time with each version, the end time being derived from the start time of the subsequent version. For a deletion, this next version can be a special "delete stub" version. For a query "as of" time T, the system then looks for the version of each record with the largest start time ≤ T.

**Storing Versions on a Page:** The common approach for organizing pages for current time databases is called a slotted array. Each array element points to a record on the page. For B-trees, these records are maintained in B-tree key order. When adding temporal support to a current time database system, it is convenient to minimize the change required for current time functionality. This argues for retaining the slotted array, and back-linking versions where each version is augmented with a pointer to its preceding version. Then for each record accessed in a query, the system follows this backward chain to the first version with a timestamp ≤ T, the "as of" time requested.

**Indexing Versions:** Being able to index historical versions by time is essential to avoid increasing costs for ever earlier query times [12]. (See the section on transaction time indexing.) Postgres [13] used the R-tree [4] for this. The TSB-tree [7] is a more specialized index that can, with an appropriate page splitting policy [1] provide guarantees about the performance of "as of" queries. Its special feature is the introduction of a time split [3] where the time interval of a full database page is partitioned, with record versions being assigned to the resulting pages whenever their lifetimes intersect the time interval of a resulting page. Thus, a version whose lifetime intersects both resulting pages will be replicated in both pages. The result is that, when combined with ordinary B-tree key splitting, each TSB-tree page contains all versions within a key-time rectangle of the search space. This enables identifying exactly which pages can contain answers to a temporal query. Despite the need for replicating versions, the space required for the versions remains linear in the number of unique versions.

**Compressing Versions:** The way that versions are stored on a page and indexed makes compressing versions simple. Usually, an update changes only a small part of a record, perhaps only a single attribute. Thus, delta compression, where the compressed version represents the difference between one version and another that is adjacent in time order, can be very effective. Only the updated attribute together with location information and timestamp needs to appear in the compressed version. Backward delta's are to be preferred because this leaves the current time data uncompressed and hence unchanged, important both for compatibility and current time performance. Because time splits in the TSB-tree always replicate versions spanning the split time, and because splitting at current time is convenient, the last version in each page is always uncompressed, and this is preserved during a time split.

Decompressing a version never needs information from any other page than the page upon which the version is stored.

## Dealing with Timestamps

In a transaction time database, for a record identified in some way, its versions are distinguished by timestamps. The nature of timestamps, when they are chosen and included, how to optimize this process, and how to deal with user requests for transaction time are discussed below.

**Nature of timestamps:** Several forms of timestamp have been used for temporal support. Some systems use transaction identifiers (XIDs) instead of time, sometimes maintaining a separate table that maps XID to time. When versioning is limited, e.g. to only support multiversion concurrency control, an active transaction's XID may be mapped to a list of transactions (their XIDs) that committed before them, hence determining which transactions have updates that should be visible to the active transaction [5]. However, for more general functionality, system time is usually used. It may need to be augmented with a sequence number because its granularity may not be sufficient to completely distinguish every transaction's updates.

**When to Timestamp:** A timestamp for a version must enable "as of" queries to always see a transaction consistent view of the data. This can be achieved when timestamp order agrees with the serialization order of transactions. If one chooses timestamps prior to updating, the timestamp can be added immediately to versions generated by the updating. However, early timestamp choice means that transactions that serialize differently must be aborted. Most implementations of transaction time functionality thus choose timestamps at commit, where the commit order is the same as the serialization order for the transactions. This means, however, that the timestamp is not available at time of update, and must be added later.

**Lazy Timestamping:** When a transaction's timestamp is determined late in the transaction, e.g., at commit time, preceding updated records need to be revisited to add the timestamp. Typically, an XID is placed in an updated record at update time, to be replaced later by the system time. Eager timestamping replaces XID with time prior to transaction commit, logging this activity as another update. This can be costly, so a lazy approach is generally preferred in which XID is replaced by time after the transaction commits. The mapping from XID to system time must be maintained persistently, at least until the timestamping is complete, to ensure that replacing XID with time can continue after a possible system crash.

**Impact of User Requested Time:** The SQL language supports a user's request for current (transaction) time within a query. It is essential that the user see a time that is consistent with the transaction timestamp used for updates of the transaction. Providing the user with a time for the transaction while the transaction is executing constrains the choice of timestamp when the transaction is committed [9]. A transaction must be aborted if a timestamp cannot be chosen that is consistent with the time provided to the user. To provide this, the system can exploit the fact that the user time request is usually not for the full precision of the timestamp, e.g. SQL DATE constrains only the date part of the timestamp. Further, remembering the largest timestamp on data that is seen by the transaction provides a lower bound for a possibly non-empty interval for timestamp choice.

## Additional Uses

The versions maintained by a transaction time database can support a variety of other system uses.

**Snapshot Isolation:** Recent versions can be used to provide snapshot isolation. With snapshot isolation (the default concurrency provided by Oracle), a transaction reads not the current data (which would be used for a serializable transaction) but a snapshot (version) current as of the start of the transaction or as of the first data read by the transaction. A transaction time database keeps these versions as well as possibly older versions. Thus efficiently providing transaction time support also provides efficient snapshot isolation. The system may choose to garbage collect older versions more quickly when they are only used for concurrency control purposes.

**Online Backup:** A database backup simply an earlier state of the database. Transaction time databases make all earlier states accessible and queryable. To use an earlier transaction time database state as a backup for, e.g. media recovery for the current state, requires two things: (1) the earlier state needs to be on a separate medium than the current state; and (2) the media recovery log needs include all updates from the earlier state forward to the current state and itself be on a separate device. A transaction time database system can use time splits (which it would use in a TSB-tree) to move versions to separate backup media, and it can do this incrementally as well [8].

**Bad User Transactions:** Occasionally, erroneous transactions commit, compromising the correctness of a database. Point-in-time recovery, where the database state is reset to an earlier time, just prior to the bad transaction, is the usual way of dealing with this. Conventional database backups used for this incur a long restore time followed by a roll forward to the just earlier time. A transaction time database lends itself greatly shortening the outage caused by this problem because earlier versions of the database are already maintained online. Oracle Flashback [11] implements point-in-time recovery in this way. One can further limit the outage by identifying exactly which transactions should be removed from the database by tracking transaction read dependencies [10].

## KEY APPLICATIONS

In addition to the system uses just described, transaction time databases are valuable for several applications, e.g. time series analysis, repeating experiments or analysis on historical data, and auditing and legal compliance.

## FUTURE DIRECTIONS

Data stream processing is a new functionality that has several important applications requiring fast reaction to sequences of events, e.g., stock market data. This data may also be stored and more carefully analyzed. It is quite natural to think about stream data as transaction time data, and ask temporal queries of it.

## CROSS REFERENCES

Temporal databases, transactions, concurrency control, multiversion data, transaction time indexing, temporal strata

## RECOMMENDED READING


1. B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB J.* 5(4), 264--275, 1996.
2. Clifford, J., C. Dyreson, T. Isakowitz, C. S. Jensen, R. T. Snodgrass, "On the Semantics of "Now" in Databases," *ACM TODS* 22, 2, 171--214, 1997.

3. M. Easton. Key-Sequence Data Sets on Inedible Storage. *IBM J. R & D* 30(3), 230--241, 1986.
4. A. Guttman, "R-trees: a dynamic index structure for spatial searching", *SIGMOD*, pp. 47--57, 1984
5. L. Hobbs, K. England. *Rdb: A Comprehensive Guide*. Digital Press 1995.
6. D. B. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction Time Support Inside a Database Engine. *ICDE*, 35, 2006.
7. D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. *SIGMOD*, 315--324, 1989.
8. D. B. Lomet and B. Salzberg. Exploiting a History Database for Backup. *VLDB*, 380--390, 1993.
9. D. B. Lomet, R. T. Snodgrass, and C. S. Jensen. Using the Lock Manager to Choose Timestamps. *IDEAS*, 357--368, 2005.
10. D. B. Lomet, Z. Vagena, and R. Barga. Recovery from "Bad" User Transactions. *SIGMOD*, 337--346, 2006.
11. Oracle. Oracle Flashback Technology. (2005) http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm
12. B. Salzberg and V.J. Tsotras, A Comparison of Access Methods for Time-Evolving Data. ACM Computing Surveys (1999), 31(2):158-221.
13. M. Stonebraker. The Design of the POSTGRES Storage System. *VLDB*, 289--300, 1987.
14. U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993
15. K. Torp, R. T. Snodgrass, C. S. Jensen. Effective Timestamping in Databases. *VLDB Journal*, 8, 4, 267--288, 2000.

# Telic Distinction in Temporal Databases (REGULAR ENTRY)

Vijay Khatri
*Operations and Decision Technologies Department*
*Kelley School of Business*
*Indiana University*
*Bloomington, Indiana, USA*
*vkhatri@indiana.edu*
URL: http://mypage.iu.edu/~vkhatri/

Richard T. Snodgrass
*Department of Computer Science*
*711 Gould Simpson*
*University of Arizona*
*P.O. Box 210077*
*Tucson, Arizona, USA*
*rts@cs.arizona.edu*
URL: http://www.cs.arizona.edu/people/rts/

Paolo Terenziani
*Universita' del Piemonte Orientale "Amedeo Avogadro"*
*Via Bellini 25, 15100 Alessandria, Italy*
*terenz@di.unito.it*
URL: http://www.di.unito.it/~terenz/

## TITLE
Telic Distinction in Temporal Databases

## SYNONYM
Point- Versus Period-Based Semantics

## DEFINITION
In the context of temporal databases, telic (atelic) data is used to store telic (atelic) facts, and the distinction between telic and atelic data is drawn using the properties of downward and upward inheritance.

- **Downward inheritance**. The *downward inheritance* property implies that one can infer from temporal data *d* that holds at valid time *t* (where t is a time period) that *d* holds in any sub-period (and sub-point) of *t*.
- **Upward inheritance**. The *upward inheritance* property implies that one can infer from temporal data *d* that holds at two consecutive or overlapping time periods $t_1$ and $t_2$ that d holds in the union time period $t_1 \cup t_2$.

In temporal databases, the semantics of atelic data implies that both downward and upward inheritance holds; on the other hand, neither downward nor upward inheritance holds for telic data.

## HISTORICAL BACKGROUND

The distinction between telic and atelic facts dates back to Aristotle's categories [1] and has had a deep influence in the Western philosophical and linguistic tradition. In particular, the distinction

between different classes of sentences (called aktionsart classes) according to their linguistic behavior and temporal properties is at the core of the modern linguistic tradition (consider, e.g., the milestone categorization by Vendler [11]). For instance, the upward and downward inheritance properties were used (without adopting terminology, which is imported from Shoham [7] and, more generally, from the artificial intelligence tradition) by Dowty [4] in order to distinguish between Vendler's accomplishments (telic facts) and states along with processes (atelic facts). Starting from the pioneering work by Bennet and Partee [3], several linguistic approaches have pointed out that the traditional point-based semantics, in which facts can be evaluated at each time point, properly applies only to atelic facts, while a period-based semantics is needed in order to properly cope with telic facts. Starting from Allen's milestone approach [2], the telic/atelic dichotomy has played a major role in the area of artificial intelligence also. In the field of temporal databases, the point-based vs. period-based dichotomy was initially related to representation and query evaluation issues (rather than to data semantics); later the connection was made between Aristotle's categories and the telic/atelic data semantics [9,10]. It is the emphasis on data semantics that renders "telic" and "atelic" the preferred term.

## SCIENTIFIC FUNDAMENTALS

The distinction between telic and atelic data regards the time when facts hold or occur, i.e., their valid time. The following discussion focuses on the temporal *semantics* of data and queries, independent of the *representation* that is used for time. Moreover, while in several database approaches the semantics of data is not distinguished from the semantics of the query, this presentation follows the logical tradition, stating that data has its own semantics independently of any query language and operators just in the same way in which a knowledge base of logical formulæ have their own semantics – usually expressed in model-theoretic terms.

### Data semantics

In the linguistic literature, most approaches classify facts (or sentences describing facts) according to their temporal properties. In particular, most approaches distinguish between telic and atelic facts, and prior research (see, e.g., [3]) points out that, while the point-based semantics is useful to cope with atelic facts, it is not suitable for telic ones, for which a period-based semantics is needed.

*Point-based semantics of data:* The data in a temporal relation is interpreted as a sequence of states (with each state a conventional relation, i.e., a set of tuples) indexed by points in time. Each state is independent of every other state.

Such temporal relations can be encoded in many different ways (data language). For example the following are three different encodings of the same information, within a point-based semantics, of John being married to Mary in the states indexed by the times 1, 2, 7, 8, and 9:

(i) <John, Mary || {1,2,7,8,9}> $\in$ R
(ii) < John, Mary || {[1–2],[7–9]}> $\in$ R
(iii) < John, Mary || [1–2]> $\in$ R and <John, Mary || [7–9]> $\in$ R

Independently of the representation, the point-based semantics implies that the fact denoted by < John, Mary > includes 5 individual states as follows:

$1 \rightarrow$ {< John, Mary >} $2 \rightarrow$ {< John, Mary >} $7 \rightarrow$ {< John, Mary >}

$8 \rightarrow$ {< John, Mary >}  $9 \rightarrow$ {< John, Mary >}

Notice that the point-based semantics naturally applies to atelic facts, since both downward and upward inheritance are naturally supported.

*Period-based semantics of data:* Each tuple in a temporal relation is associated with a multiset of time periods, which are the temporal extents in which the fact described by the tuple occur. In this case, time periods are atomic primitive entities in the sense that they cannot be decomposed. Note, however, that time periods can overlap and that there is no total order on time periods, unlike time points.

For example, let <John || {[10–20]}> represent the fact that John started to build a house at time 10 and finished at time 20. If a period-based semantics is adopted, the period [10–20] is interpreted as an atomic (indivisible) one.

(i) $[10,20] \rightarrow$ {<John>}

Note that a period-based semantics does not imply that John built the house in [12–15], or at the time point 12, or at any other time period other than [10–20]. As a consequence, the period-based semantics is naturally suited to cope with telic facts, for which (by definition) neither downward nor upward inheritance holds.

Although several query and data representation languages include time periods, most temporal database approaches adopt, explicitly or implicitly, the point-based semantics, interpreting a temporal database as a set of conventional databases, each one holding at a specific snapshot of time. This is the approach followed, e.g., by the Bitemporal Conceptual Data Model (BCDM) [5], a model that has been proven to capture the semantic core of many prior approaches in the temporal database literature, including the TSQL2 “consensus” approach [8]. While point-based semantics perfectly works when coping with atelic data, the problem with using it to cope with telic fact is illustrated by the following example.

**Example**
Phone calls are durative telic facts. For instance, if John made a call to Mary from time 10 to time 12, he didn’t make it from 10 to 11. Similarly, two consecutive calls, one from 10 to 12 (inclusive) and the other from 13 to 15 (inclusive), are clearly different from a single call from 10 to 15. However, such a distinction cannot be captured at the semantic level, if the point-based semantics is used. In fact, the point-based semantics for the two phone calls of John is as follows:

$10 \rightarrow$ {<John, Mary>}  $11 \rightarrow$ {<John, Mary>}  $12 \rightarrow$ {<John, Mary>}
$13 \rightarrow$ {<John, Mary>}  $14 \rightarrow$ {<John, Mary>}  $15 \rightarrow$ {<John, Mary>}

Based on point-semantics, there is no way to of grasping that two different calls were made. In other words, there is a loss of information. Note that such a loss of information is completely independent of the representation language used to model data. For instance, the above example could be represented as

(i) <John, Mary || {10,11,12,13,14,15}> $\in$ R
(ii) < John, Mary || {[10–12],[13–15]}}> $\in$ R
(iii) < John, Mary || [10–12]> $\in$ R and <John, Mary || [13–15]> $\in$ R

But, as long as the point-based semantics is used, the data semantics is the one elicited above.

On the other hand, independently of the representation formalism being chosen, the semantics of telic facts such as phone calls is properly coped with if the period-based semantics is used. For instance, in the phone example, the semantics

[10-12] $\rightarrow$ {<John, Mary>}  [13-15] $\rightarrow$ {<John, Mary>}

correctly expresses the distinction between the two consecutive phone calls.

In an analogous way, period-based semantics is not suitable to model atelic facts. In short, both upward and downward inheritance holds for them, and the period-based semantics does not support such properties.

Terenziani and Snodgrass have proposed a two-sorted data model, in which telic data can be stored in telic relations (i.e., relations to be interpreted using a period-based semantics) and atelic data in atelic relations (i.e., relations to be interpreted using a point-based semantics) [9].

**Query Semantics**

It is interesting that the loss of information due to the treatment of telic data in a point-based (atelic) framework is even more evident when queries are considered. Results of queries should depend only on the data semantics, not on the data representation. For instance, considering the phone example above (and independently of the chosen representation), queries about the number or duration of phone calls would not provide the desired answers. For instance, the number of calls from John to Mary would be one, and a call from John to Mary would (incorrectly) be provided to a query asking for calls lasting for at least five consecutive units.

In order to cope with a data model supporting both telic and atelic relations, temporal query languages must be extended. Specifically, queries must cope with atelic relations, telic relations, or a combination of both.

Furthermore, linguistic research suggests a further requirement for telic/atelic query languages: *flexibility*. It is widely accepted within the linguistic community that while basic facts can be classified as telic or atelic, natural languages provides several ways to switch between the two classes. For instance, given a telic fact (such as “John built a house”), the progressive form (e.g., “John *was* build*ing* a house”) coerces it into an atelic one, stripping away the culmination (and, in fact, “John was building a house” does not imply that “John built a house”, i.e., that he finished it) [6]. For the sake of expressiveness, it is desirable that a database query language provide the same flexibility.

**Queries about atelic data**

As already mentioned above, most database approaches are interpreted (implicitly or explicitly) on the basis of the point-based semantics. Therefore, the corresponding algebraic operators already cope with atelic data. As an example, in BCDM, the union of two relations is simply obtained by taking the tuples of both relations, and “merging” the valid time of value equivalent tuples performing the union of the time points in their valid time. This definition is perfectly consistent with the “snapshot-by-snapshot” view enforced by the underlying point-based (atelic) semantics.

However, the algebræ in the literature also contain operators which contrast with such a “snapshot-by-snapshot” underlying semantics. Typical examples are temporal selection

operators. For instance, whenever a duration is asked for (e.g., “retrieve all persons married for at least *n* consecutive time units”), the query implicitly relies on a telic view of data, in which snapshots are not taken into account independently of each others.

**Queries about telic data**
Algebraic operators on telic data can be easily defined by paralleling the atelic definitions, and considering that, in the telic case, the basic temporal primitives are not time points, but time periods [9]. For instance, telic union is similar to atelic one, except that the merging of valid times of value-equivalent tuples is performed by making the union of multisets of time periods, considered as primitive entities, e.g.,

{[10–12],[ 13–15]} ∪ {[10–14],[ 13–18]} = {[10–12],[ 13–15],[10–14],[ 13–18]}

Note that such temporal selection operators perfectly fit with the telic environment. On the other hand, algebraic operators that intuitively involve a snapshot-by-snapshot view of data (e.g., Cartesian product, involving a snapshot-by-snapshot intersection between valid times) have an awkward interpretation in the telic context; for this reason, difference and “standard” Cartesian product have not been defined in this telic algebra.

**Queries combining telic and atelic data**
In general, if a two-sorted data model is used, queries combining relations of both kinds are needed. In general, such queries involve the (explicit or implicit) coercion of some of the relations, to make the sort of the relations consistent with the types of the operators being used. For instance, the following query utilizes the example atelic relation modeling marriages and the atelic one considering phone calls: Who were being married when John was calling Mary?
In such a case, the English clause “when” demands for an atelic interpretation: the result can be obtained by first coercing the relation about phone calls into an atelic relation, and then by getting the temporal intersection through the application of the atelic Cartesian product.

On the other hand, the query “list the marriage ceremonies that had a duration of more than 3 hours” requires a coercion of marriages into a telic relation, so that the whole valid time is considered as (a set of) time periods (instead as a set of independent points, as in the atelic interpretation), and the telic temporal selection operator can be applied.

In general, two coercion operators need to be provided [9]. Coercion from telic to atelic is easy: each time period constituting the (semantics of the) valid time is converted into the set of time points it contains, e.g., *to-atelic*({[10–12],[13–15]}) = {10,11,12,13,14, 15}. Of course, since multisets of time periods are more expressive than sets of time points, such a conversion causes a loss of information. On the other hand, coercion from atelic to telic demands the formation of time periods out of sets of points: the output is the set of maximal convex time periods exactly covering the input set of time points, e.g., to-telic({10,11,12,13,14, 15}) = {[10–15]}.

## KEY APPLICATIONS
Most applications involve differentiating between telic and atelic data.

## CROSS REFERENCES
Atelic Data, Period-Stamped Temporal Models, Point-Stamped Temporal Models, Temporal Query Languages

**RECOMMENDED READING**


[1] Aristotle, The Categories, on Interpretation. Prior Analytics. Cambridge, MA, Harvard University Press.

[2] J.F. Allen, “Towards a General Theory of Action and Time,” *Artificial Intelligence* 23:123–154, 1984.

[3] M. Bennet and B. Partee, “Tense and Discourse Location In Situation Semantics,” distributed 1978 by the Indiana University Linguistics Club, Bloomington.

[4] D. Dowty, “The Effects of the Aspectual Class on the Temporal Structure of Discourse,” *Tense and Aspect in Discourse, Linguistics and Philosophy* 9(1), 37–61, 1986.

[5] C. S. Jensen and R. T. Snodgrass, “Semantics of Time-Varying Information,” *Information Systems*, 21(4):311–352, 1996.

[6] M. Moens and M. Steedman, “Temporal Ontology and Temporal Reference,” *Computational Linguistics 14*(2):15–28, 1998.

[7] Y. Shoham, “Temporal Logics in AI: Semantical and Ontological Considerations,” *Artificial Intelligence* 33:89–104, 1987.

[8] R.T. Snodgrass (editor), The Temporal Query Language TSQL2, Kluwer Academic Pub., 1995

[9] P. Terenziani and R. T. Snodgrass, “Reconciling Point-based and Interval-based Semantics in Temporal Relational Databases: A Proper Treatment of the Telic/Atelic Distinction.” IEEE *Transactions on Knowledge and Data Engineering*, 16(4), 540–551, 2004.

[10] P. Terenziani, R. T. Snodgrass, A. Bottrighi, M. Torchio, and G. Molino, “Extending Temporal Databases to Deal with Telic/Atelic Medical Data,” *Artificial Intelligence in Medicine* 39(2):113–126, 2007.

[11] Z. Vendler, “Verbs and times,” in Linguistics in Philosophy, Cornell University Press, New York NY, 97–121, 1967.

# Temporal Access Control

*Yue Zhang*,
*James B.D. Joshi*,
School of Information Sciences, University of Pittsburgh,
{yuz20+, jjoshi}@ pitt.edu

## SYNONYMS

Time-based access control

## DEFINITION

Temporal access control refers to access control service that restricts granting of authorization based on time. The authorization may be given to a subject for a particular interval or duration of time or based on the temporal characteristics of the objects being accessed. Such a need arises from the fact that a subject's need to access a resource and the sensitivity (and hence the protection requirement) of the objects being accessed may change with time.

## HISTORICAL BACKGROUND

Work related to temporal access control has had only a brief history and goes back to early 90s. In many real-world situations, access to information and resources may have to be restricted based on time as the subject and object characteristics may change and so can the need for the subject to access the object. For example, in a hospital organization, the head of the hospital may need to grant the permissions related to a part-time doctor only during certain time intervals, e.g. everyday between 10am and 3pm. Similarly, an external auditor may need to be given access to sensitive company data for a specific duration. Need for temporal access control has been emphasized by Thomas *et al.* [5].

Bertino *et al.*'s *Temporal Authorization Model* (TAM) is the first known access control model to support the time-based access control requirements in a discretionary access control (DAC) model [1]. TAM associates a periodicity constraint (see definitional entry for TAM) with each authorization indicating the valid time instants of the authorization. TAM also defines four derivation rules that allow an authorization to be derived based on another authorization. The TAM model, however, is limited to specifying the temporal interval for an entire authorization and does not consider the temporal characteristics of data/objects [2]. For example, there could be an authorization like "a subject *s* is allowed to read object *o* one month after it has been created/written". Furthermore, the states of the object could change with time and access to such different object-states may need to be carefully specified. Atluri *et al*. propose a Temporal and Derived data Authorization Model (TDAM) for a Web information portal [2]. An information portal mainly aims to provide access to data from different sources and hence the temporal characteristics of such data need to be properly captured in an access control policy [2]. TDAM uses a logic formula to capture time-related conditions to specify authorization rules. TDAM also discusses the consistency problem for the authorization of derived data.

Other significant work related to temporal access control can be seen within the context of the Role Based Access Control (RBAC) model. Research on RBAC gathered significant momentum in the 90s. Bertino *et al.* proposed the Temporal Role Based Access Control Model (TRBAC) model by extending the existing RBAC model [3]. The TRBAC model supports the periodicity/interval constraints on the role enabling/disabling and uses triggers to define dependencies that may exist among basic role related events such as "*enable role*" and "*disable role*". Bertino *et al.* also provided formal analysis of consistency and safety issues for TRBAC policies.

One limitation of the TRBAC model is that it only supports specification of a set of temporal intervals on the role enabling/disabling events. For example, the user-role and the role-permission assignments are not time-constrained in TRBAC, neither are the role activations. Joshi *et al.* proposed a General Temporal RBAC (GTRBAC) model to support more time-based constraints to allow specification of more fine-

grained access control policies [4]. The GTRBAC model allows the interval and duration constraints on user-role assignment, role-permission assignment, and role enabling events. It also defines the duration and cardinality constraints on role activation. GTRBAC uses constraint enabling events to trigger the duration constraints and uses run-time requests to allow dynamic changes in the authorization states by administrators and users. GTRBAC uses triggers, first introduced in the TRBAC model, to support the dependencies among events. The other features of the GTRBAC include the hybrid hierarchy and temporal Separation of Duty (SoD) constraints.

Along with work on temporal access control, many researchers have also started working on access control approaches based on the location context and the integration of temporal and location based access control. Such work includes the GEO-RBAC model, the spatial-temporal access control model [6], the spatial-temporal RBAC model [7], the LRBAC model [8], and the location and time-based RBAC model (LoT-RBAC). These efforts are geared towards addressing the generic need for context based access control for emerging systems and applications.

## SCIENTIFIC FUNDAMENTALS

In this section, we introduce the existing time-based access control models mentioned above, namely TAM, TDAM, the TRBAC model, and the GTRBAC model. Definitional entries for these as well as GEO-RBAC and LoTRBAC models have been included in this volume. The major features of each model are discussed below with examples.

### TAM: Temporal Authorization Model

TAM models the temporal context as a periodicity constraint. A periodicity constraint is of the form <[*begin*, *end*], *P*> every time instant in *P* between "begin" and "end", where *P* indicates a recurring set of intervals indicating. For example, the periodical constraint "[1/1/1994, ∞], Monday" indicates every Monday starting 1/1/1994. The model simply uses the symbol $\perp$ in place of *P* to indicate all time instants. The temporal authorization in TAM associates such a periodicity constraint with a normal discretionary authorization. The authorization is valid at any time instant specified in the periodicity constraint. At any given time instant, if there are two authorization rules that try to grant and deny an operation on the same object, a conflict is said to occur. In such a case, the model uses the "*denials-take-precedence*" principle to favor negative authorization.

It is possible that several authorizations have temporal dependencies among them. For example, suppose user $u_1$ grants a permission $p$ to $u_2$ (authorization $A_1$), and $u_2$ wants to further grant $p$ to $u_3$ (authorization $A_2$). It is easy to see that $A_2$ can only be valid after $A_1$ becomes valid. Therefore, a requirement such as "$u_2$ is allowed to grant $p$ to $u_2$ whenever he/she acquires $p$ from $u_1$," can be specified as a derivation rule "$A_2$ WHENEVER $A_1$". TAM uses three such derivation rules to capture various relationships among different authorizations.

The derivation rule is of the form "[*begin*, *end*], *P*, *A* <*op*>, $\mathcal{A}$", where, *A* is an authorization, $\mathcal{A}$ is a boolean expression on authorizations, and <*op*> is one of the following: WHENEVER, ASLONGAS, UPON. $\mathcal{A}$ is true at time instant *t*, if $\mathcal{A}$ evaluates true by substituting each authorization in it by the value "true" if it is valid at *t*, and by the value 'false" if not valid. For example, $\mathcal{A} = \neg$ ($A_1$ and $A_2$) is true at time *t* if one or both of $A_1$ and $A_2$ is not true at time *t*. ([*begin, end*], P, *A* WHENEVER $\mathcal{A}$) specifies that we can derive *A* for each instant in $\prod(P) \cap \{[t_b, t_e]\}$ (here, $\prod(P)$ is the set of all time instant in *P*) for which $\mathcal{A}$ is valid. ([*begin, end*], P, *A* ASLONGAS $\mathcal{A}$) specifies that we can derive *A* for each instant in $\prod(P) \cap \{[t_b, t_e]\}$ such that $\mathcal{A}$ is valid for each time instant in II(*P*) that is greater than or equal to $t_b$ and lesser than or equal to $t_e$. ([*begin, end*], *P*, *A* UPON $\mathcal{A}$) specifies that we can derive A for each instant in $\prod (P) \cap \{[t_b, t_e]\}$ if there exists an instant $t' \in \prod(P)$ that is greater than or equal to $t_b$ and lesser than or equal to $t_e$ such that $\mathcal{A}$ is valid at time *t*'. The difference between WHENEVER and ASLONGAS is that the former only evaluates the authorization state for a time instant while the latter evaluates the history of the authorization states in a given time interval. The difference between ASLONGAS and UPON is that the former evaluates the

authorization state for the entire time interval in the history while the latter only evaluates the authorization states at a time instant in the history. The following example, taken from [1], illustrates the use of these four derivation rules:

*Example*: Consider the following authorizations.

- ($A_1$) ([1995, 5/20/1995], $\perp$ (manager, guidelines, write, +, Sam))
- ($A_2$) ([10/1/1995, $\infty$], Working-days, (technical-staff, guidelines, read, +, Sam))
- ($R_1$) ([1996, 1998], Working-days, (temporary-staff, document, read, +, Sam) ASLONGAS
  $\neg$ (summer-staff, document, read, +, Sam))
- ($R_2$) ([1995, $\infty$], Mondays and Fridays, (technical-staff, report, write, +, Sam) UPON
  $\neg$ ((manager, guidelines, write, +, Sam) $\vee$ (staff, guidelines, write, +, Sam))
- ($R_3$) ([1995, $\infty$], $\perp$, (technical-staff, report, write, -, Sam) WHENEVER
  $\neg$ (technical-staff, guidelines, read, +, Sam))

From these, the following temporal authorizations can be derived using the five rules:

- (technical-staff, report, write, +, Sam) for each Monday and Friday from 5/22/1995 to $\infty$, from rule $R_2$, using authorization $A_1$.
- (technical-staff, report, write, -, Sam) for each day in [1/1/1995, 9/30/1995] and for each Saturday and Sunday from 10/1/1995 to $\infty$, from rule $R_3$ using authorization $A_2$.

Given a set of initial authorizations and rules, the derived authorizations may depend on the order in which the rules are evaluated. This is due to the existence of both positive and negative authorizations and the "denials-take-precedence" rule. To analyze this problem, the author defines the unique set of valid authorizations using the notion of critical set. Moreover, the authors also propose a Critical Set Detection (CSD) algorithm to verify whether a given set is critical [1].

**TDAM: Temporal and Derived data Authorization Model**

The TDAM model, as mentioned earlier, focuses on providing authorization specification by considering temporal characteristics of data/objects. This is achieved by associating a complex formula $\tau$ instead of a simple temporal interval to each authorization. At any time instant, if $\tau$ is evaluated to be true, then the corresponding authorization is valid. Otherwise, the corresponding authorization is not valid. By carefully designing the formula $\tau$, the model can support specification of a very fine-grained temporal access control policy. If we want to use the temporal context of a single data item, we can make it the variable in $\tau$. For example, assume a company maintains a database including the current and future predicted price for some goods, and let each price of a data item $d$ be associated with a time interval [$t_b$, $t_e$] indicating the temporal interval when the specified price is predicted for. If we restrict a subject to read the current price only, we can specify $\tau$ as "$t_b \leq t \leq t_e$" where $t$ is the current time.

Similarly, the temporal dependency, instead of specifying absolute time interval, can also be indicated by $\tau$. For example, consider the policy "a subject $s$ is allowed to read object $o$ one month after it has been created/written." Such a requirement can be specified by including $\tau$ as "$t \geq t_w$+1 month" where $t_w$ indicates the time when the object $o$ is created/written and $t$ is the current time.

In summary, the logic formula $\tau$ associated with each authorization gives TDAM the ability to specify very fine-grained temporal authorization policies based on the temporal properties of the data. The following example, taken from [2], illustrates the use of $\tau$ for temporal authorizations.

*Example*: Consider a stock exchange and assume that subjects belonging to a specific privileged group (say *pg*) are allowed access to a certain object (*o*), which is, say, the last trade size of various companies,

only five minutes after it has been written into the database. The authorization can be specified as: (*pg*, *o*, *read*, +, $t_x$ + 5 minutes $\leq t_{req}$). The formula "$t_x$+5 minutes$\leq t_{req}$" establishes the 5 minute delay as intended. For example, consider an element with two states $se_1$ and $se_2$ of a specific company that is traded on the stock exchange, with transaction times $ct_x$ = 58, and $se_2.t_x$ = 64. If $t_{req}$=63 is the access request time, $se_1$ is selected, since $se_1.t_x$=58 and thus $t_x$+5 minutes=58 + 5 = 63 $\leq t_{req}$. $se_1$ and $se_2$ are both selected for $t_{req}$ =69, but not for earlier times than that as only at time instant 69 the assignment $se_2.t_x$ = 64 becomes valid. It is worth noting that in the absence of any valid time constraint, a state-element will be presented to the user even if its valid time has already expired. For example, assuming $se_1.t_v$ = [57, 63), the object *o* remains accessible for a subject from the *pg* group even after time 63.

**TRBAC: Temporal Role Based Access Control Model**

The TRBAC model allows the specification of temporal constraints to specify when a role can be enabled or disabled, and triggers to capture possible temporal dependencies among role events. The TRBAC model also uses periodic time expression instead of simple time interval to represent the periodicity constraint. In RBAC, the users can acquire the permissions only by activating enabled roles within a session and hence the TRBAC model associates the periodic time expressions with role enabling/disabling events to define periodicity constraints.

Triggers constitute an important part of the TRBAC model, and they are used to capture temporal dependencies among RBAC authorization states. A trigger simply specifies that if some events occur and/or some role status predicates are evaluated to be true then another event can occur, with a possible time delay. For example, assume that a hospital requires that when a part-time doctor is in duty a part-time nurse can also be allowed to log in to help the part-time doctor. Here the trigger "enable *part-time doctor* → enable *part-time nurse*" can be used to capture such a requirement. An example of a TRBAC policy, taken from [3], for a medical domain is as follows;

*Example*: ($PE_1$). ([1/1/2000, ∞], *Day-time*, *VH*: *enable doctor-on-day-duty*)
($PE_2$). ([1/1/2000, ∞], *Night-time*, *VH*: *disable doctor-on-day-duty*)
($RT_1$). *enable doctor-on-day-duty → H: enable nurse-on-day-duty*
($RT_2$). *disable doctor-on-day-duty → H: disable nurse-on-day-duty*

Bertino *et al.* introduces a soft notion of *safety* to formally characterize the efficiency and practicality of the model. In essence, because of triggers and other periodicity constraints, ambiguous semantics can be generated. They propose an efficient graph based analysis technique to identify such ambiguous policies so that a unique execution model can be guaranteed by eliminating them. Once such a unique execution model is ensured by eliminating the triggers that give rise to ambiguity, the policy base is said to be *safe*. Conflicts could also occur because of the opposing *enable* and *disable* role events. The TRBAC model uses *denial- takes-precedence* in conjunction with *priority* to resolve such conflicts.

**GTRBAC : Generalized Temporal Role Based Access Control Model**

Joshi *et al.* have proposed the GTRBAC model by extending the TRBAC model to incorporate more comprehensive set of periodicity and duration constraints and time-based activation constraints. The GTRBAC model introduces the separate notion of role enabling and role activation, and introduces role states. An *enabled* role indicates that a valid user can activate it, whereas a *disabled* role indicates that the role is not available for use. A role in *active* state indicates that at least one user has activated the role. Such a distinction makes the semantics and implementation of any RBAC compatible system much clearer. Besides events related to user-role assignment, permission-role assignment and role enabling, the GTRBAC model also includes role activation/deactivation events. The GTRBAC model associates the temporal constraints with every possible event in the system. In particular, GTRBAC uses the periodicity constraint, introduced in the TRBAC model, to constrain the validity of role enabling events, user-role assignment events as well as the permission-role assignment events. As the role activation events are initiated by the users at their discretion, GTRBAC does not associate temporal constraints with activation events. Besides the periodic temporal constraints, GTRBAC also supports duration constraints, unlike the TRBAC model. A duration constraint specifies how long should an event be valid for once it occurs. For

example, one duration constraint could specify that once a role *r* has been enabled it should be in the enabled state for 2 hours. Note that the duration constraint has a non-deterministic start time and requires some other actions to initiate the start of the duration, i.e., it is important to define at which time instant the duration constraint starts to count. For example, consider a duration constraint (2 hours, enable *r*). Here, when the "enable *r*" event occurs because of a trigger, the event becomes valid for 2 hours because of this constraint. At times, one may need to also enable duration or activation constraints - GTRBAC uses constraint enabling events to facilitate that.

The GTRBAC model also supports the temporal and cardinality constraints on role activation. Cardinality constraints have been often mentioned in the literature but have not been addressed much in the existing models. For example, the ANSI RBAC standard (see the definitional entry) does not include cardinality constraints. A cardinality constraint simply limits the number of activations (applies for assignments as well) within a given period of time. For example, GTRBAC supports limiting the total number of activations of a role or the maximum concurrent number of activations of a role in a given interval or duration. Furthermore, GTRBAC allows the activation constraint to be applied to all the activations of a role (per-role constraint) or applied to each activation of a role by a particular user (per-user constraint). The GTRBAC model uses the trigger framework introduced in the TRBAC model. In addition to these, the GTRBAC model also extends work on role hierarchy and constraints, and introduces hybrid hierarchy and time-based SoD constraints.

The GTRBAC model uses three conflict types (Type-1, Type-2 and Type-3) to categorize the different types of conflicting situations that may arise and provides a resolution technique. In particular, the conflict resolution technique uses a combination of the following approaches: (i) *priority-based*, (ii) *denials-takes precedence* and (iii) *more specific constraint takes precedence*. The GTRBAC framework also extends the notion of safety introduced in the TRBAC model. The following example, taken from [4], illustrates a GTRBAC policy for a medical information system [4].

Table 1. Example GTRBAC access policy for a medical information System

| | | |
|---|---|---|
| 1 | a | (*DayTime*, `enable` DayDoctor), (*NightTime*, `enable` NightDoctor) |
| | b | ((M, W, F), $\texttt{assign}_U$ *Adams* `to` DayDoctor), ((T, Th, S, Su), $\texttt{assign}_U$ *Bill* `to` DayDoctor), |
| | c | (*Everyday between* 10am - 3pm, $\texttt{assign}_U$ *Carol* `to` DayDoctor) |
| 2 | a | ($\texttt{assign}_U$ *Ami* `to` NurseInTraining); ($\texttt{assign}_U$ *Elizabeth* `to` DayNurse) |
| | b | *c*1 = (6 *hours*, 2 *hours*, `enable` NurseInTraining) |
| 3 | a | (`enable` DayNurse → `enable` *c*1) |
| | b | (`activate` DayNurse `for` *Elizabeth* → `enable` NurseInTraining `after` 10 *min*) |
| | c | (`enable` NightDoctor → `enable` NightNurse after 10 *min*); (`disable` NightDoctor → `disable` NightNurse after 10 *min*) |
| 4 | (a) (10, $\texttt{active}_{R\_n}$ DayNurse); (b) (5, $\texttt{active}_{R\_n}$ NightNurse); (c) (2 *hours*, $\texttt{active}_{R\_total}$ NurseInTraining) | |

*Example*: In row 1a, the enabling times of DayDoctor and NightDoctor roles are specified as a periodicity constraint. The (*I, P*) forms for *DayTime* (9am-9pm) and *NightTime* (9pm-9am) are as follows: *DayTime* = ([12/1/2003, ∞], *all.Da*ys, + 10.*Hours* ▷ 12.*Hours*), and *NightTime* = ([12/1/2003, ∞], *all.Da*ys, + 12.*Hours* ▷ 12.*Hours*). In constraint 1b, *Adams* is assigned to the role of DayDoctor on *Mondays*, *Wednesdays* and *Fridays*, whereas *Bill* is assigned to this role on *Tuesdays*, *Thursdays, Saturdays* and *Sundays*. The assignment in constraint 1c indicates that *Carol* can assume the DayDoctor role everyday between 10 am and 3pm. In constraint 2a, users *Ami* and *Elizabeth* are assigned to the roles of NurseInTraining and DayNurse respectively, without any periodicity or duration constraints. In other words, their assignments are valid at all the times. Constraint 2b specifies a duration constraint of 2 *hours* for the enabling time of the NurseInTraining role, but this constraint is valid only for 6 *hours* after the

constraint $c1$ is enabled. Consequently, once the NurseInTraining role is enabled, *Ami* can activate the NurseInTraining role at the most for two hours.

Trigger 3a indicates that the constraint $c_1$ in row 2b is enabled once the DayNurse is enabled. As a result, the NurseInTraining role can be enabled within 6 *hours*. Trigger 3b indicates that 10 *min* after *Elizabeth* activates the DayNurse role, the NurseInTraining role is enabled for a period of 2 *hours*. As a result, a nurse-in-training can have access to the system only if *Elizabeth* is present in the system. In other words, once the roles are assumed, *Elizabeth* acts as a training supervisor for a nurse-in-training. Note that *Elizabeth* can activate the DayNurse role multiple times within a duration of 6 hours after the DayNurse role is enabled. The activation constraint 4c limits the total activation time associated with the NurseInTraining role to 2 *hours*. The constraint set 4 shows additional activation constraints. For example, constraint 4a indicates that there can be at most 10 users activating DayDoctor role at a time, whereas 4b shows that there can be at most 5 users activating the NightDoctor role at a time.

Suroop *et al*. have recently proposed the *Location and Time based RBAC* (LoTRBAC) model (please see definitional entry for more detail) by extending the GTRBAC model to address both time and location context for security mobile applications..

## KEY APPLICATIONS

Temporal access control models are suitable for the applications where temporal constraints and temporal dependencies among authorizations are important protection requirements. One example is workflow systems that often have timing constraints on tasks and their dependencies. In particular, TAM is suitable for the system implementing the discretionary access control policies. TDAM is suitable for access control for data dissemination systems such as a web information portal where data have different states at different times. TRBAC and GTRBAC are suitable for large scale systems that have very fine-grained time-based access requirements.

## FUTURE DIRECTIONS

With the development of networking and mobile technologies, context based access control is becoming very crucial. Time is very crucial context information, as is location. Hence, the work on temporal access control have provided a basis for looking at the intricacies of the context parameters that need to be used to restrict authorization decisions. In addition to context-based access, content-based access control is also becoming significant issues because of the growing need to dynamically identify content and make authorization decisions based on its semantics. Again, the work on temporal access control have provided a basis for capture the dynamic feature of the object content. Authorization models that capture context and content parameters using temporal access control are being pursued to develop more fine-grained access control models.

## CROSS REFERENCES

Role Based Access Control


## ACKNOWLEDGEMENTS

This work has been supported by the US National Science Foundation award IIS-0545912.


## REFERENCES


[1] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, “An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning”, ACM Transactions on Database Systems, 23(3):231--285, 1998

[2] V. Atluri, A. Gal, “An authorization model for temporal and derived data: securing information portals,” ACM Transactions on Information and System Security, Vol 5, Issues 1 (Feb. 2002), pp 62-94.

[3] E. Bertino, P.A. Bonatti, E. Ferrari, “TRBAC: A temporal role-based access control model”, ACM Transactions on Information and System Security, Vol 4. Issue 3. pp. 191-233, 2001

[4] James B. D. Joshi , Elisa Bertino , Usman Latif , Arif Ghafoor, A Generalized Temporal Role-Based Access Control Model, IEEE Transactions on Knowledge and Data Engineering, v.17 n.1, p.4-23, January 2005

[5] Thomas, R.K. and Sandhu, R, “Discretionary access control in object-oriented databases: Issues and research directions”, Proceedings of the Sixteenth National Computer Security Conference 1993, 63--74.

[6] Song Fu, Cheng-Zhong Xu, "A Coordinated Spatio-Temporal Access Control Model for Mobile Computing in Coalition Environments”, 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 17, 2005

[7] Indrakshi Ray and Manachai Toahchoodee, "A Spatio-Temporal Role-Based Access Control Model", Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Redondo Beach, California, July 2007.

[8] Indrakshi Ray, Mahendra Kumar, and Lijun Yu, "LRBAC: A Location-Aware Role-Based Access Control Model", Proceedings of the 2nd International Conference on Information Systems Security, Kolkata, India, December 2006

# Temporal Aggregation

Johann Gamper Michael Böhlen Christian S. Jensen
Free University of Bozen-Bolzano, Italy Aalborg University, Denmark
http://www.inf.unibz.it/~{gamper,boehlen} http://www.cs.aau.dk/~csj

**SYNONYMS**

none

**DEFINITION**

In database management, aggregation denotes the process of consolidating or summarizing a database instance; this is typically done by creating so-called aggregation groups of elements in the argument database instance and then applying an aggregate function to each group, thus obtaining an aggregate value for each group that is then associated with each element in the group. In a relational database context, the instances are relations and the elements are tuples. Aggregation groups are then typically formed by partitioning the tuples based on the values of one or more attributes so that tuples with identical values for these attributes are assigned to the same group. An aggregate function, e.g., *sum*, *avg*, or *min*, is then applied to another attribute to obtain a single value for each group that is assigned to each tuple in the group as a value of a new attribute. Relational projection is used for eliminating detail from aggregation results.

In temporal relational aggregation, the arguments are temporal relations, and the tuples can also be grouped according to their timestamp values. In temporal grouping, groups of values from the time domain are formed. Then an argument tuple is assigned to each group that overlaps with the tuple's timestamp, this way obtaining groups of tuples. When aggregate functions are applied to the groups of tuples, a temporal relation results. Different kinds of temporal groupings are possible: instantaneous temporal aggregation where the time line is partitioned into time instants/points; moving-window (or cumulative) temporal aggregation where additionally a time period is place around a time instant to determine the aggregation groups; and span aggregation where the time line is partitioned into user-defined time periods.

**HISTORICAL BACKGROUND**

Aggregate functions assist with the summarization of large volumes of data, and they were introduced in early relational database management systems such as System R and INGRES. During the intensive research activities in temporal databases in the 1980's, aggregates were incorporated in temporal query languages, e.g., the Time Relational model [1], TSQL [8], TQuel [9], and a proposal by Tansel [11]. The earliest proposal aimed at the efficient processing of (instantaneous) temporal aggregates is due to Tuma [12]. Following Tuma's pioneering work, research concentrated on the development of efficient main-memory algorithms for the evaluation of instantaneous temporal aggregates as the most important form of temporal aggregation [6, 7].

With the diffusion of data warehouses and OLAP, disk-based index structures for the incremental computation and maintenance of temporal aggregates were investigated by Yang and Widom [14] and extended by Zhang et al. [15] to include non-temporal range predicates. The high memory requirements of the latter approach were addressed by Tao et al. [10], and approximate solutions for temporal aggregation were proposed. More recently, Vega Lopez et al. [13] formalized temporal aggregation in a uniform framework that enables the analysis and comparison of the different forms of temporal aggregation based on various mechanisms for defining aggregation groups. In a similar vein, Böhlen et al. [2] develop a new framework that generalizes existing forms of temporal aggregation by decoupling the partitioning of the time line from the specification of the aggregation groups.

It has been observed that expressing queries on temporal databases is often difficult with SQL, in particular for aggregation. As a result, temporal query languages often include support for temporal aggregation. A recent paper [3] studies the support for temporal aggregation in different types of temporal extensions to SQL. A subset of the temporal aggregates considered in the entry are also found in non-relational query languages, e.g., $\tau$XQuery [5].

**SCIENTIFIC FUNDAMENTALS**

Without loss of generality, a discrete time domain consisting of a totally ordered set of time instants/points is assumed together with an interval-based, valid-time data model, i.e., an interval timestamp is assigned to each tuple that captures the time when the corresponding fact is true in the modeled reality. As a running example, the temporal relation *CheckOut* in Figure 1 is used, which records rentals of video tapes, e.g., customer C101 rent tape T1234 from time 1 to 3 at cost 4.

| *CustID* | *TapeNum* | *Cost* | *T* |
|---|---|---|---|
| C101 | T1234 | 4 | [1,3] |
| C102 | T1245 | 2 | [3,5] |
| C102 | T1234 | 4 | [7,10] |
| C102 | T1245 | 2 | [17,18] |
| C102 | T1245 | 2 | [19,20] |

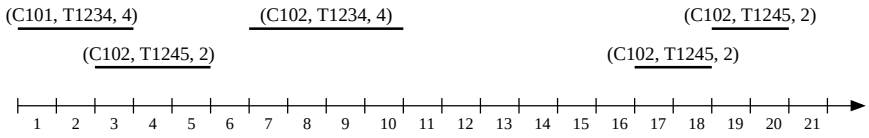



Figure 1: Tabular Representation and Graphical Representation of Temporal Relation *CheckOut*

## Defining Temporal Aggregation

Various forms of temporal aggregation that differ in how the temporal grouping is accomplished have been studied. In *instantaneous temporal aggregation (ITA)* the time line is partitioned into time instants and an aggregation group is associated with each time instant $t$ that contains all tuples with a timestamp that intersects with $t$. Then the aggregate functions are evaluated on each group, producing a single aggregate value at each time $t$. Finally, identical aggregate results for consecutive time instants are coalesced into so-called constant intervals that are maximal intervals over which all result values remain constant. In some approaches, the aggregate results in the same constant interval must also have the same lineage, meaning that they are produced from the same set of argument tuples. The following query, $Q_1$, and its result in Figure 2(a) illustrate ITA: *What is the number of tapes that have been checked out?* Without the lineage requirement, the result tuples $(1, [17, 18])$ and $(1, [19, 20])$ would have been coalesced into $(1, [17, 20])$. While, conceptually, the time line is partitioned into time instants, which yields the most detailed result, the result tuples are consolidated so that only one tuple is reported for each constant interval. A main drawback is that the result relation is typically larger than the argument relation and can be up to twice the size of the argument relation.

With *moving-window temporal aggregation (MWTA)* (first introduced in TSQL [8] and later also termed *cumulative temporal aggregation* [9, 14]), a time window is used to determine the aggregation groups. For each time instant $t$, an aggregation group is defined as the set of argument tuples that hold in the interval $[t-w, t]$, where $w \geq 0$ is called a window offset. In some work [13], a pair of offsets $w$ and $w'$ is used, yielding a window $[t-w, t+w']$ for determining the aggregation groups. After computing the aggregate functions for each aggregation group, coalescing is applied similarly to how it is done for ITA to obtain result tuples over maximal time intervals. The following query, $Q_2$, and its result in Figure 2(b) illustrate MWTA: *What is the number of tapes that have been checked out in the last three days?* To answer this query, a window is moved along the time line, computing at each time point an aggregate value over the set of tuples that are valid at some point during the last three days. While both ITA and MWTA partition the time line into time instants, the important difference is in how the aggregation groups for each time instant are defined.

Next, for *span temporal aggregation (STA)*, the time line is first partitioned into predefined intervals that are defined independently of the argument relation. For each such interval, an aggregation group is then given as the set of all argument tuples that overlap the interval. A result tuple is produced for each interval by evaluating an aggregate function over the corresponding aggregation group. The following query, $Q_3$, and its result in Figure 2(c) illustrate STA: *What is the weekly number of tapes that have been checked out?* The time span is here defined as a period of seven days. In contrast to in ITA and MWTA, in STA the timestamps of the result tuples are specified by the application and are independent of the argument data. Most approaches consider only regular time spans expressed in terms of granularities, e.g., years, months, and days.

The *multi-dimensional temporal aggregation (MDTA)* [2] extends existing approaches to temporal aggregation, by decoupling the definition of result groups and aggregation groups. A result group specifies the part of a result tuple that is independent of the actual aggregation (corresponds to the group by attributes in SQL). Each result group has an associated aggregation group, namely the set of tuples from which the aggregated value

| *Cnt* | *T* |
|---|---|
| 1 | [1,2] |
| 2 | [3,3] |
| 1 | [4,5] |
| 0 | [6,6] |
| 1 | [7,10] |
| 0 | [11,16] |
| 1 | [17,18] |
| 1 | [19,20] |

(a) $Q_1$: ITA

| *Cnt* | *T* |
|---|---|
| 1 | [1,2] |
| 2 | [3,5] |
| 1 | [6,6] |
| 2 | [7,7] |
| 1 | [8,12] |
| 0 | [13,16] |
| 1 | [17,18] |
| 2 | [19,20] |
| 1 | [21,22] |

(b) $Q_2$: MWTA

| *Cnt* | *T* |
|---|---|
| 3 | [1,7] |
| 1 | [8,14] |
| 2 | [15,21] |

(c) $Q_3$: STA

| *CntE* | *CntC* | *T* |
|---|---|---|
| 0 | 0 | [1,7] |
| 2 | 1 | [8,14] |
| 1 | 0 | [15,21] |
| 0 | 2 | [15,21] |

(d) $Q_4$: MDTA

Figure 2: Results of Different Forms of Temporal Aggregation

is computed. In general, the grouping attributes of the tuples in an aggregation group might differ from the grouping attributes of the result group. For the specification of the result groups, two different semantics are supported: constant-interval semantics that covers ITA and MWTA and fixed-interval semantics that covers STA. The fixed-interval semantics supports the partitioning of the time line into arbitrary, possibly overlapping time intervals. The following query, $Q_4$, and its result in Figure 2(d) illustrate some of the new features of MDTA: *For each week, list the number of expensive and the number of cheap checkouts during the preceding week?* (expensive being defined as a cost equal or greater than 4 and cheap as a cost equal or smaller than 2). The result groups are composed of a single temporal attribute that partitions the time line, the tuples in the associated aggregation groups do not have to overlap the timestamp of the result group, and two aggregates over different aggregation groups are computed for each result group.

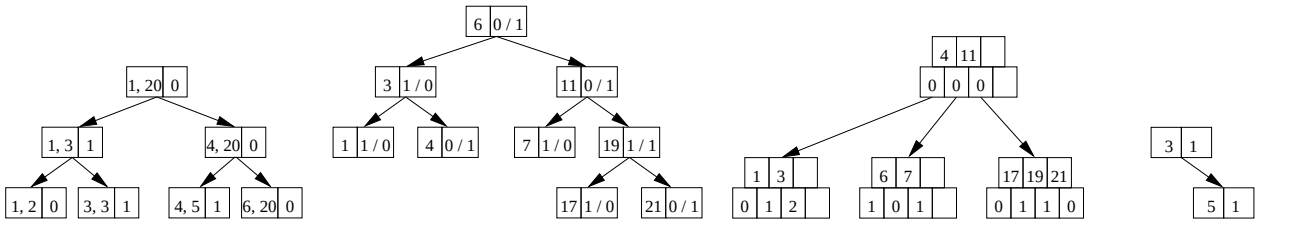
## Temporal Aggregation Processing Techniques

The efficient computation of temporal aggregation poses new challenges, most importantly the computation of the time intervals of the result tuples that depend on the argument tuples and thus are not known in advance.

**Two Scans.** The earliest proposal for computing ITA was presented by Tuma [12] and requires two scans of the argument relation—one for computing the constant intervals and one for computing the aggregate values over these intervals. The algorithm has a worst case running time of $O(mn)$ for $m$ result tuples and $n$ argument tuples. Following Tuma's pioneering work, research concentrated on algorithms that construct main-memory data structures that allow to perform both steps at once, thus requiring only one scan of the argument relation.

**Aggregation Tree.** The *aggregation tree* algorithm for ITA by Kline and Snodgrass [6] incrementally constructs a tree structure in main memory while scanning the argument relation. The tree stores a hierarchy of intervals and partial aggregation results. The intervals at the leaf nodes encode the constant intervals. Accumulating the partial results in a depth-first traversal of the tree yields the result tuples in chronological order. Figure 3(a) shows the tree for Query $Q_1$ after scanning the first two argument tuples. The path from the root to the leaf with time interval $[3, 3]$ yields the result tuple $(2, [3, 3])$. The algorithm is constrained by the size of the available main memory, and it has a worst case time complexity of $O(n^2)$ for $n$ argument tuples since the tree is not balanced. An improvement, although with the same worst case complexity, is the $k$-ordered aggregation tree [6], which requires the argument tuples to be chronologically ordered to some degree. This allows to reduce the memory requirements by garbage collecting old nodes that will not be affected by any future tuples. Gao et al. [4] describe a number of parallel temporal aggregation algorithms that are all based on the aggregation tree.

**Balanced Tree.** Moon et al. [7] propose the *balanced tree* algorithm for the main memory evaluation of ITA queries involving *sum*, *count*, and *avg*. As the argument tuples are scanned, their start and end times are stored in a balanced tree together with two values for each aggregate function being computed, namely the partial aggregate result over all tuples that start and end here, respectively. An in-order traversal of the tree combines these values to compute the result relation. Whenever a node, $v$, is visited, a result tuple is produced over the interval that is formed by the time point of the previously visited node and the time point immediately




(a) Aggregation Tree (b) Balanced Tree (c) SB-Tree (d) Endpoint Tree

Figure 3: Different Forms of Tree Structures for Temporal Aggregation

preceding $v$. Figure 3(b) shows the balanced tree for Query $Q_1$. The aggregate value of the result tuple $(2, [3, 3])$ is determined as $1 + (1 - 0) = 2$. Although the balanced tree requires less memory than the aggregation tree, it is constrained by the amount of available memory. For the *min* and *max* functions a merge-sort like algorithm is proposed. Both algorithms have $O(n \log n)$ time complexity for $n$ argument tuples. To overcome the memory limitation, a bucket algorithm is proposed, which partitiones the argument relation along the time line and keeps long-lived tuples in a meta-array. Aggregation is then performed on each bucket in isolation.

**SB-Tree.** Yang and Widom [14] propose a disk-based index structure, the *SB-tree*, together with algorithms for the incremental computation and maintenance of ITA and MWTA queries. It combines features from the segment tree and the B-tree and stores a hierarchy of time intervals associated with partially computed aggregates. To find the value of an aggregate at a time instant $t$, the tree is traversed from the root to the leaf that contains $t$ and the partial aggregate values associated with the time intervals that contain $t$ are combined. Figure 3(c) shows the SB-tree for Query $Q_1$. The value at time 8 results from adding 0 and 1 (associated with $[4, 10]$ and $[4, 10]$, respectively). The time complexity of answering an ITA query at a single time point is $O(h)$, where $h$ is the height of the tree, and $O(h + r)$ for retrieving the result over a time interval, where $r$ is the number of leaves that intersect with the given time interval. The same paper extends the basic SB-tree to compute MWTA. For a fixed window offset $w$, the timestamps of the argument tuples are extended by $w$ to account for the tuples' contributions to the results at later time points. For arbitrary window offsets, a pair of SB-trees is required.

**MVSB-Tree.** With the SB-tree, aggregate queries are always applied to an entire argument relation. The *multi-version SB-tree* (MVSB-tree) by Zhang et al. [15] tackles this problem and supports temporal aggregation coupled with non-temporal range predicates that select the tuples over which an aggregate is computed. The MVSB-tree is logically a sequence of SB-trees, one for each timestamp. The main drawbacks of this approach are: the tree might be larger than the argument relation, the range restriction is limited to a single non-timestamp attribute, and the temporal evolution of the aggregate values cannot be computed. Tao et al. [10] present two approximate solutions that address the high memory requirements of the MVSB-tree. They use an MVB-tree and a combination of B- and R-trees, respectively. These achieve linear space complexity in the size of the argument relation and logarithmic query time complexity.

**MDTA.** Böhlen et al. [2] provide two memory-based algorithms for the evaluation of MDTA queries. The algorithm for fixed-interval semantics keeps in a *group table* the result groups that are extended with an additional column for each aggregate being computed. As the argument relation is scanned, all aggregate values to which a tuple contributes are updated. The group table contains then the result relation. The memory requirements only depend on the size of the result relation. With an index on the group table, the average runtime is $n \log m$ for $n$ argument tuples and $m$ result groups, the worst case being $O(nm)$ when each argument tuple contributes to each result tuple. The algorithm for constant-interval semantics processes the argument tuples in chronological order and computes the result tuples as time proceeds. An *endpoint tree* maintains partial aggregate results that are computed over all argument tuples that are currently valid, and they are indexed by the tuples' end points. Figure 3(d) shows the endpoint tree for Query $Q_1$ after processing the first two argument tuples. When the third argument tuple is read, the result tuples $(2, [3, 3])$ and $(1, [4, 5])$ are generated by accumulating all partial aggregate

values; the nodes are then removed from the tree. The size of the tree is determined by the maximal number of overlapping tuples, $n_o$. The average time complexity of the algorithm is $n * n_o$. The worst-case complexity is $O(n^2)$, when the start and end points of all argument tuples are different and all tuples overlap.

## KEY APPLICATIONS

Temporal aggregation is used widely in different data-intensive applications, which become more and more important with the increasing availability of huge volumes of data in many application domains, e.g., medical, environmental, scientific, or financial applications. Prominent examples of specific applications include data warehousing and stream processing. Time variance is one of four salient characteristics of a data warehouse, and there is general consensus that a data warehouse is likely to contain several years of time-referenced data. Temporal aggregation is a key operation for the analysis of such data repositories. Similarly, data streams are inherently temporal, and the computation of aggregation functions is by far the most important operation on such data. Many of the ideas, methods, and technologies from temporal aggregation have been and will be adopted for stream processing.

## FUTURE DIRECTIONS

Future research work is possible in various directions. First, it may be of interest to study new forms of temporal aggregation. For example, a temporal aggregation operator that combines the best features of ITA and STA may be attractive. This operator should follow a data-driven approach that approximates the precision of ITA while allowing to limit the size of the result. Second, it is relevant to study efficient evaluation algorithms for more complex aggregate functions beyond the five standard functions for which most research has been done so far. Third, the results obtained so far can be adapted for and applied in related fields, including spatio-temporal databases where uncertainty is inherent as well as data streaming applications.

## CROSS REFERENCE

Temporal Database, Temporal Data Mining, Temporal Coalescing, Temporal Query Languages, Temporal Indexing, Temporal Query Processing

## RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.


[1] J. Ben-Zvi. *The Time Relational Model*. Ph.D. thesis, Computer Science Department, UCLA, 1982.

[2] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *International Conference on Extending Database Technology*, pages 257–275, 2006.

[3] M. H. Böhlen, J. Gamper, and C. S. Jensen. How would you like to aggregate your temporal data? In *Thirteenth International Symposium on Temporal Representation and Reasoning*, pages 121–136, 2006.

[4] D. Gao, J. A. G. Gendrano, B. Moon, R. T. Snodgrass, M. Park, B. C. Huang, and J. M. Rodrigue. Main memory-based algorithms for efficient parallel aggregation for temporal databases. *Distributed and Parallel Databases*, 16(2):123–163, 2004.

[5] D. Gao and R. T. Snodgrass. Temporal slicing in the evaluation of XML queries. In *International Conference on Very Large Databases*, pages 632–643, 2003.

[6] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *International Conference on Data Engineering*, pages 222–231, 1995.

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

[8] S. B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1-3):147–175, 1989.

[9] R. T. Snodgrass, S. Gomez, and L. E. McKenzie. Aggregates in the temporal query language TQuel. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):826–842, 1993.

[10] Y. Tao, D. Papadias, and C. Faloutsos. Approximate temporal aggregation. In *International Conference on Data Engineering*, pages 190–201, 2004.

[11] A. U. Tansel. A statistical interface to historical relational databases. In *International Conference on Data Engineering*, pages 538–546, 1987.

[12] P. A. Tuma. *Implementing Historical Aggregates in TempIS*. Ph.D. thesis, Wayne State University, 1992.

[13] I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.

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

[15] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Symposium on Principles of Database Systems*, pages 237–245, 2001.

# Temporal Algebras

**Abdullah Uz Tansel**
**Baruch College & The Graduate Center – CUNY**
**55 Lexington Avenue, Box 11-220**
**New York, NY 10010**
**Abdullah_tansel@baruch.cuny.edu**

SYNONYMS

Historical algebras, Valid-time algebras, Transaction-time algebras, Bitemporal algebras

DEFINITION

Temporal algebra is a generic term for an algebra defined for a data model that organizes temporal data. A temporal data model may support Valid-time (the time over which a data value is valid), Transaction-time (time when a data value is recorded in the database), or both (Bitemporal). So an algebra can be defined for each case, a Valid-time relational algebra, a Transaction-time relational algebra, or a Bitemporal relational algebra, respectively. Temporal algebras include the temporal versions of relational algebra operations in addition to new operations for manipulating temporal data like Time-slice, Rollback, Temporal Coalesce, temporal restructuring operations, and others. For a Temporal algebra, it is desirable to be closed (common algebras are closed), a consistent extension of the relational algebra and to reduce to relational algebra when only current data is considered.

HISTORICAL BACKGROUND

Temporal algebraic languages first appeared as extensions to the relational algebra in early 1980's, mostly Valid –time or Transaction-time algebras, followed by Bitemporal algebras. These extensions differed according to their operations and how temporal data is represented. McKenzie and Snodgrass surveyed temporal algebras and identified desirable criteria that are needed in a temporal algebra [10]. However, some of these criteria are conflicting.

SCIENTIFIC FUNDEMENTALS

**Temporal Algebra Basics**

Let *T* represent the time domain which has a linear order under "≤". A time point (instant) is any element of *T*. A period is a consecutive sequence of time points. A temporal element is a set of disjoint maximal periods [6], and a temporal set is any set of time points. Any of these time constructs can be used to timestamp data values. A Bitemporal atom, <*Valid-Time*, *Transaction-time*, *Data*> asserts that *data* is valid during *Valid-time* and is recorded during *Transaction-time*. Either Valid-time or Transaction-time may be omitted and the result is a Valid-time Atom, or a Transaction-time atom, respectively.

A temporal relational algebra is closely related to how the temporal data (temporal atoms) are

represented, i.e. the type of timestamps used, where they are attached (relations, tuples, or attribute values), and whether temporal atoms are kept atomic or broken into their components. In other words, time specification may be explicit or implicit (see the entry on temporal data models). This in turn determines possible evaluation (semantics) of temporal algebra expressions. There are two commonly adopted approaches: 1) Snapshot (Point or Sequenced [2]) evaluation that manipulates the snapshot relation at each time point, like Temporal Logic [6, 11, 17]; 2) Traditional (Nonsequenced [2]) evaluation that manipulates the entire temporal relation much like the traditional relational algebra. It is also possible to mix these approaches.The syntax and the operations of a temporal algebra are designed to accommodate a desired type of evaluation. Moreover, specifying temporal algebra operations at an attribute level, instead of tuples keeps the tuple structure intact after the operation is executed, so preserving the rest of a temporal relation that is beyond the scope of operation applied.

Let Q(A, B, C), R(A, D) and S(A, D) be temporal relations in attribute timestamping, whose attribute values are sets of temporal atoms except attribute A which has constant values. It is possibly a temporal grouping identifier [5] (or a temporal key). For the sake of simplicity, attributes B, C, and D are assumed to have one temporal atom in each tuple, i.e., they are already flattened. $Q_t$ stands for the snapshot of temporal relation Q at time t. Temporal Algebras generally include temporal equivalents of traditional relational algebra operations. The five basic Temporal Algebra operations, $\cup^t$, $-^t$, $\pi^t$, $\sigma^t$, and $\times^t$ in snapshot evaluation are [6, 11, 17]:

- $R \cup^t S_t$ is $R_t \cup S_t$ for all t in T
- $R -^t S_t$ is $R_t - S_t$ for all t in T
- $\pi^t_{A1,A2,\ldots,An}(R)$ is $\pi_{A1,A2,\ldots,An}(R_t)$ for all t in T
- $\sigma^t_F(R)$ is $\sigma_F(R_t)$ for all t in T; Formula F includes traditional predicates and temporal predicates like *Before*, *After*, *Overlaps*, etc
- $R \times^t Q$ is $R_t \times Q_t$ for all t in T

An important issue in Snapshot Evaluation is the homogeneity of the temporal relations. A temporal relation is homogenous if each tuple is defined on the same time, i.e., in a tuple, all the attributes have the same time reference [6]. If attributes in a tuple have different time references then a snapshot may have null values leading to complications. Thus, a Temporal Algebra may be homogenous or not depending on the relations scheme on which it is defined.

In Temporal Algebras that use traditional evaluation, $\cup^t$, $-^t$, $\pi^t$, $\sigma^t$, and $\times^t$ may be defined exactly the same as the relational algebra operations or they have temporal semantics incorporated in their definitions. The temporal set operations may specially be defined by considering the overlapping timestamps in tuples. Two temporal tuples are value equivalent if their value components are the same, but their timestamps may be different. Let {(a1,<[2/07,11/07), d1>)} and {(a1, <[6/07,8/07), d1>)} be tuples in R and S, respectively. These two tuples are value equivalent. In case of $R \cup^t S$, value equivalent tuples are combined into one if their timestamps overlap. Considering the former tuples the result is {(a1,<[2/07,11/07), d1>)}. In case of $R -^t S$, the common portion of the timestamps for the value equivalent tuples is removed from the tuples of R. For the above tuples the result is {a1, <[2/07,6/07), d1>), (a1, <[8/07,11/07), d1>)}. Existence of value equivalent tuples makes query specification more complex but, query evaluation is less costly. On the other hand eliminating them is also more costly. Temporal

Coalescing operation combines value equivalent tuples into one tuple [1]. Temporal algebras may include aggregates (see the entry on temporal aggregates).

The definition of Temporal Projection ($\pi^t$) is straight forward. However, it may generate value equivalent tuples much like the traditional projection operation creates duplicate tuples. Moreover, the projection operation may also be used to discard the time of a relation if it is explicitly specified. In the case of implicit time specification, it needs to be converted to an explicit specification before applying the projection operation. The formula F in the Selection operation ($\sigma^t_F(Q)$) may include time points, the end points of periods, and periods in temporal elements, or temporal predicates like *Before*, *After*, *Overlaps*, etc. It is possible to simulate the temporal predicates by conditions referring to time points or end points of periods.

Other temporal algebra operations such as Temporal Set Intersection or Temporal Join are similarly defined. There are different versions of Temporal Join. Intersection join is computed over the common time of operand relations (see the entry on temporal joins). For instance, if {(a1, <[1/07, 5/07), b1>, <[1/07, 4/07), c1>)} is a tuple in Q, the natural join ($Q \bowtie R$) contains the tuple {(a1, <[1/07, 5/07), b1>, <[1/07, 4/07), c1>, <[2/07,11/07), d1>)}. If this were an intersection natural join, times of the attributes in this tuple would be restricted to their common time period [2/07, 4/07). It is also possible to define temporal outer joins [11].
.

Temporal algebra operations are defined independent of time granularities. However, if operand relations are defined on different time granularities, a granularity conversion is required as part of processing the operation.

**Algebras for Tuple Timestamping**

In tuple timestamping relations are augmented with one column to represent time points, periods or temporal elements, or two columns to represent periods. Relation Q is represented as Q1(A, B, From, To) and Q2(A, C, From, To) where From and To are the end points of periods. Similarly, R(A, D, From, To) and S(A, D, From, To) correspond to the relations R and S, respectively. The tuple of Q given above is represented as the following tuples: (a1, b1, 1/07, 5/07) in Q1 and (a1, c1, 1/07, 4/07) in Q2. For accessing time points within a period snapshot evaluation may be used [6, 17] or in case of Traditional Evaluation, attributes representing the end points of periods may be specified in operations. Another path followed is to define temporal expansion and contraction operations [9]. A period is expanded to all the time points included in it by temporal expansion and temporal contraction does the opposite, converts a sequence of time points to a period. Relation instances indexed by time points are used to define a temporal algebra by Clifford, Croker, and Tuzhilin [17].

**Algebras for Attribute Timestamping**

Timestamps are attached to attributes and N1NF relations are used and the entire history of an object is represented as a set of temporal atoms in one tuple. These temporal relations are called temporally grouped in contrast to temporally ungrouped relations that are based on tuple timestamping [17]. Naturally, temporally grouped algebras are more expressive than temporally ungrouped algebras and the former is more complex than the latter [17]. A temporal algebra that is based on snapshot evaluation and allows set theoretic operations on temporal elements is given

in [6]. For the algebra expression e, the construct [[e]] returns the time over which e's result is defined [6] and it can further be used in algebraic expressions. An algebra, based on time points and lifespans, that uses snapshot evaluation is proposed in [3, 4]. The nest and unnest operations for the transformations between 1NF and N1NF relations and operations which form and break temporal atoms are included in a temporal algebra [3, 12, 13]. N1NF Temporal relations and their algebras may or may not be homogonous [6, 12, 13].

**Valid-time and Transaction-time Algebras**

Most of the algebras mentioned above are Valid- time Relational Algebras. A Valid-time Relational Algebra includes additionally a Slice operation ($\varsigma$) that is a redundant, but very useful operation. Let R be a Valid-time Relation and t be a time point (period, temporal element, or temporal set). Then, $\varsigma_t$ (R) cuts a slice from R, the values that are valid over time t and returns them as a relation [2, 3, 12, 13]. Common usage of the Slice operation is to express the "when" predicate in natural languages. Slice may also be incorporated into the selection operation or the specification of a relation to restrict the time of a temporal relation by a temporal element, i.e, R[t] [6]. Slice may be applied at an attribute level to synchronize time of one attribute by another attribute [3, 12, 13]. For instance, $\varsigma_{B,C}$ (Q) restricts the time of attribute B by the time of attribute C. Applying the Slice operation on all the attributes by the same time specification returns a snapshot at the specified time.

A Transaction-time relational algebra includes a Rollback operation ($\tau$), another form of Slice for rolling back to the values recorded at a designated time. Let R be a Transaction-time relation and t be a time point (period, temporal element, or temporal set). Then, $\tau_t$ (R) cuts a slice from R, the values that were recorded over time t and returns them as a relation [2, 3, 8]. Note the duality between the Valid-time relational algebra and the Transaction-time relational algebra; each has the same set of operations and an appropriate version of the slice operation.

**BiTemporal Relational Algebras**

Bitemporal algebra operations are more complicated than temporal algebras that support one time dimension only [2, 8, 16]. A Bitemporal algebra includes both forms of the Slice operation in addition to other algebraic operations. A Bitemporal query has a context that may or may not imply a Rollback operation [16]. However, once a Rollback operation is applied on a Bitemporal relation; Valid-time algebra operations can be applied on the result. It is also possible to apply Bitemporal Algebra operations on a bitemporal relation before applying a Rollback operation. In this case, the entire temporal relation is the context of the algebraic operations. In coalescing Bitemporal tuples, starting with Valid-time or Transaction-time may result in different coalesced tuples [8]. For the data maintenance queries, Valid-time needs to be coalesced within the Transaction-time.

KEY APPLICATIONS

Use of temporal algebra includes query language design [14], temporal relational completeness [15, 17], and query optimization [14]. Some Relational algebra identities directly apply to temporal relational algebra whereas other identities do not hold due to the composite representation or semantics of temporal data [10]. Naturally, identities for the new temporal algebra operations and their interaction with the operations borrowed from the relational algebra

need to be explored as well [3, 7].

## CROSS REFERENCES

Bitemporal interval, Bitemporal relation, Interval-based temporal models, Nonsequenced semantics, Period Stamped Models, Point-stamped temporal models, Relational algebra, Relational data model, Sequenced semantics, Snapshot equivalence, Temporal aggregates, Temporal coalescing, Temporal conceptual data model, Temporal data models, Temporal element, Temporal query languages, Temporal expression, Temporal homogeneity, Temporal joins, Temporal natural join, Temporal object oriented databases, Temporal projection, Temporal query optimization, Temporal query processing, Time domain, Time interval, Time period, Time slice, Transaction time, Value equivalence, Valid time.

## RECOMMENDED READING


1. Böhlen, M.H., Snodgrass, R.T. and Soo, M.D. (1996): Coalescing in Temporal Databases. *Proceedings of International Conference on Very Large Databases*.

2. Böhlen, M. H., C. S. Jensen, R. T. Snodgrass (2000) Temporal Statement Modifiers. *ACM Transactions on Database Systems, Vol. 25, No. 4, pp. 407-456.*

3. Clifford J. and Tansel A.U. (1985): On an Algebra for Historical Relational Databases: Two Views. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 247-265.

4. Clifford, J., Croker A. (1987): The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. *ICDE 1987*, 528-537.

5. Clifford, J., Croker, A. and Tuzhilin A. (1993): On Completeness of Historical Data Models. *ACM Transactions on Database Systems*, 19(1), 64-116.

6. Gadia, S.K (1988): A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4), 418-448.

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

8. Jensen, C. S., Soo, M. D., and Snodgrass, R. T. (1994): Unifying Temporal Data Models via a Conceptual Model. *Information Systems* 19(7): 513-547.

9. Lorentzos, N. A Johnson, R. G. (1988): Extending Relational Algebra to Manipulate Temporal Data. *Information Systems* 13(3), 289-296.

10. McKenzie, E., Snodgrass, R.T. (1991): Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. ACM Computing Surveys. 23(4), 501-543.
11. Soo, M. D, Jensen, C., and Snodgrass R. T. (1995): An Algebra for TSQL2 in TSQL2 Temporal Query Language. R. T. Snodgrass, editor, Kluwer Academic Publishers, 505–546.
12. Tansel, A.U (1986): Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems* 11(4), 343-355.
13. Tansel, A.U (1997): Temporal Relational Data Model. *IEEE Transactions on Knowledge and Database Engineering*, 9(3), 464-479.
14. Tansel, A.U., Arkun, M.E. and Ozsoyoglu, G. (1989): Time-by-Example Query Language for Historical Databases. *IEEE Transactions on Software Engineering* 15(4), 464-478.
15. Tansel, A.U. and Tin E. (1997): Expressive Power of Temporal Relational Query Languages. *IEEE Transactions on Knowledge and Data Engineering*, 9(1), 120-134.
16. Tansel, A.U, Eren-Atay, C., (2006) Nested Bitemporal Relational Algebra. *ISCIS 2006*, 622-633.
17. Tuzhilin, A., Clifford, J. (1990): A Temporal Relational Algebra as Basis for Temporal Relational Completeness. *VLDB 1990*, 13-23.

# Temporal Coalescing

Michael Böhlen, Free University of Bozen-Bolzano, Italy

## SYNONYMS

none

## DEFINITION

Temporal coalescing is a unary operator applicable to temporal databases that is similar to duplicate elimination in conventional databases. Temporal coalescing merges value-equivalent tuples, i.e., tuples with overlapping or adjacent timestamps and matching explicit attribute values. Tuples in a temporal relation that agree on the explicit attribute values and that have adjacent or overlapping timestamps are candidates for temporal coalescing. The result of operators may change if a relation is coalesced before applying the operator. For instance an operator that counts the number of tuples in a relation or an operator that selects all tuples with a timestamp spanning at least 3 months are sensitive to temporal coalescing.

## HISTORICAL BACKGROUND

Early temporal relational models implicitly assumed that the relations were coalesced. Ben Zvi's Time Relational Model [12, chap.8], Clifford and Croker's Historical Relational Data Model (HRDM) [12, chap.1], Navathe's Temporal Relational Model (TRM) [12, chap.4], and the data models defined by Gadia [12, p.28-66], Sadeghi [9] and Tansel [12, chap.7] all have this property. The term *coalesced* was coined by Snodgrass in his description of the data model underlying TQuel, which also requires temporal coalescing [10]. Later data models, such as those associated with HSQL [12, chap.5] and TSQL2 [11], explicitly required coalesced relations. The query languages associated with these data models generally did not include explicit constructs for temporal coalescing. HSQL is the exception; it includes a `COALESCE ON` clause within the select statement, and a `COALESCED` optional modifier immediately following `SELECT` [12, chap.5]. Some query languages that do not require coalesced relations provide constructs to explicitly specify temporal coalescing; VT-SQL [8] and ATSQL [2] are examples.

Navathe and Ahmed defined the first temporal coalescing algebraic operator; they called this COMPRESS [12, chap.4]. Sarda defined an operator called COALESCE [12, chap.5], Lorentzos' FOLD operator includes temporal coalescing [12, chap.3], Leung's second variant of a temporal select join operator $TSJ_2$ [12, chap.14] can be used to effect temporal coalescing, and TSQL2's representational algebra also included a coalesce operator [11].

In terms of performance and expressiveness Leung and Pirahesh provided a mapping of the coalesce operation into *recursive* SQL [6, p. 329]. Lorentzos and Johnson provided a translation of his FOLD operator into Quel [7, p. 295]. Böhlen et al. [3] show how to express temporal coalescing in terms of standard SQL and compare different implementations.

## SCIENTIFIC FUNDAMENTALS

Temporal databases support the recording and retrieval of time-varying information [12] and associate with each tuple in a temporal relation one or more timestamps that denote some time periods. The discussion assumes that each tuple is associated with a valid time attribute VT. This attribute is called the timestamp of the tuple. The timestamps are half open time periods: the start point is included but the end point is not. The non-timestamp attributes are referred to as the explicit attributes.

In a temporal database, tuples are uncoalesced when they have identical attribute values and their timestamps are either adjacent in time ("meet" in Allen's taxonomy [1]) or have some time in common. Consider the relations in Figure 1. The relation records bonus payments that have been given to employees. Ron received two 2K bonuses: one for his performance from January 1981 to April 1981 and another one for his performance from May 1981 to September 1981. Pam received a 3K bonus for her performance from April 1981 to May 1981. Bonus1 is uncoalesced since the tuples for Ron have adjacent timestamps and can be coalesced. Bonus2 is coalesced. Coalescing Bonus1 yields Bonus2.

Bonus1

| Name | Amount | VT |
|---|---|---|
| Ron | 2K | [1981/01-1981/06) |
| Ron | 2K | [1981/06-1981/10) |
| Pam | 3K | [1981/04-1981/06) |

Bonus2

| Name | Amount | VT |
|---|---|---|
| Ron | 2K | [1981/01-1981/10) |
| Pam | 3K | [1981/04-1981/06) |

Figure 1: Uncoalesced (Bonus1) and Coalesced (Bonus2) Valid Time Relations

As with duplicate elimination in nontemporal databases, the result of some operators in temporal databases changes if the argument relation is coalesced before applying the operator [11]. For instance an operator that counts the number of tuples in a relation or an operator that selects all tuples with a timestamp spanning at least 3 months are sensitive to temporal coalescing.
In general, two tuples in a valid time relation are candidates for temporal coalescing if they have identical explicit attribute values (see value equivalence [10]) and have adjacent or overlapping timestamps. Such tuples can arise in many ways. For example, a projection of a coalesced temporal relation may produce an uncoalesced result, much as duplicate tuples may be produced by a duplicate preserving projection on a duplicate-free nontemporal relation. In addition, update and insertion operations may not enforce temporal coalescing, possibly due to efficiency concerns.
Thus, whether a relation is coalesced or not makes a semantic difference. In general, it is not possible to switch between a coalesced and an uncoalesced representation without changing the semantics of programs. Moreover, as frequently used database operations (projection, union, insertion, and update) may lead to potentially uncoalesced relations and because many (but not all) real world queries require coalesced relations, a fast implementation is imperative.
Temporal coalescing is potentially more expensive than duplicate elimination, which relies on an equality predicate over the attributes. Temporal coalescing also requires detecting if the timestamps of tuples overlap, which is an inequality predicate over the timestamp attribute. Most conventional DBMSs handle inequality predicates poorly; the typical strategy is to resort to exhaustive comparison when confronted with such predicates [5], yielding quadratic complexity (or worse) for this operation.

**Implementing Temporal Coalescing** Temporal coalescing does not add expressive power to SQL. Assuming that time is linear, i.e., totally ordered, it is possible to compute a coalesced relation instance with a single SQL statement (see also [4, p. 291]). The basic idea is to use a join to determine the first (f) and last (l) time period of a sequence of value equivalent tuples with adjacent or overlapping timestamps as illustrated in Figure 2.

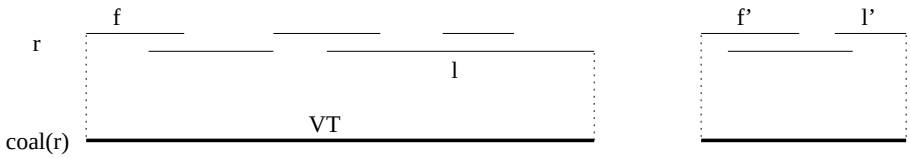



Figure 2: Illustration of Temporal Coalescing

The SQL code assumes that the time period is represented by start ($S$) and end ($E$) point, respectively. Besides start and end point there is an explicit attribute $c$. This yields a relation with schema $R(S, E, c)$. Two subqueries are used to ensure that there are no temporal gaps (for example between l and f' is a temporal gap) and that the sequence is maximal (there is no tuple with a time period that starts before the start point of f and that temporally overlaps with f; there is no tuple with a time period that ends after the end point of l and that temporally overlaps with l), respectively.

```
SELECT DISTINCT f.S, l.E, f.c
FROM r AS f, r AS l
WHERE f.S < l.E
AND f.c = l.c
```

```
AND NOT EXISTS (SELECT *
                FROM r AS m
                WHERE m.c = f.c
                AND f.S < m.S AND m.S < l.E
                AND NOT EXISTS (SELECT *
                                FROM r AS a1
                                WHERE a1.c = f.c
                                AND a1.S < m.S AND m.S <= a1.E))
AND NOT EXISTS (SELECT *
                FROM r AS a2
                WHERE a2.c = f.c
                AND (a2.S < f.S AND f.S <= a2.E OR
                     a2.S <= l.E AND l.E < a2.E))
```

The above SQL statement effectively coalesces a relation. However, current database systems cannot evaluate this statement efficiently. It is possible to exploit the fact that only the maximal time periods are relevant. Rather than inserting a new tuple (and retaining the old ones) it is possible to update one of the tuples that was used to derive the new one. This approach can be implemented by iterating an update statement. The statement is repeated until the relation does not change anymore, i.e., until the fixpoint with respect to temporal coalescing is reached.

*repeat*

```
  UPDATE r l
  SET (l.E) = (
    SELECT MAX(h.E)
    FROM r h
    WHERE l.c = h.c
    AND l.S < h.S AND l.E >= h.S AND l.E < h.E)
  WHERE EXISTS (
    SELECT *
    FROM r h
    WHERE l.c = h.c
    AND l.S < h.S AND l.E >= h.S AND l.E < h.E)
```

*until fixpoint(r)*

One means to further improve the performance is to use the DBMS as an enhanced storage manager and to develop main memory algorithms on top of it. Essentially, this means to load the relation into main memory, coalesce it manually, and then store it back in the database. If tuples are fetched ordered primarily by explicit attribute values and secondarily by start points it is possible to coalesce a relation with just a single tuple in main memory. The core of the C code of the temporal coalescing algorithm is displayed below. It uses ODBC to access the database.

```
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
SQLConnect(hdbc,"Ora10g",SQL_NTS,"scott",SQL_NTS,"tiger",SQL_NTS);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt1)
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2)

/* initialize buffer curr_tpl (a one tuple buffer) */
SQLExecDirect(hstmt1, "SELECT S, E, c FROM r ORDER BY c, S", SQL_NTS))
curr_tpl.S = next_tpl.S;
curr_tpl.E = next_tpl.E;
curr_tpl.c = next_tpl.c;

/* open a cursor to store tuples back in the DB */
SQLPrepare(hstmt2, "INSERT INTO r_coal VALUES (?,?,?)", SQL_NTS))
```

```
/* main memory temporal coalescing */
while (SQLFetch(hstmt1) != SQL_NO_DATA) { /* fetch all tuples */
  if (curr_tpl.c == next_tpl.c && next_tpl.S <= curr_tpl.E) {
    /* value-equivalent and overlapping */
    if (next_tpl.E > curr_tpl.E) curr_tpl.E = next_tpl.E;
  } else {
    /* not value-equivalent or non-overlapping */
    SQLExecute(hstmt2)) /* store back current tuple */
    curr_tpl.S = next_tpl.S;
    curr_tpl.E = next_tpl.E;
    curr_tpl.c = next_tpl.c;
  }
}
SQLExecute(hstmt2)) /* store back current tuple */
```

## KEY APPLICATIONS*

Temporal coalescing defines a normal form for temporal relations and is a crucial and frequently used operator for applications that do not want to distinguish between snapshot equivalent relations. Applications that allow to distinguish between snapshot equivalent relations have temporal coalescing as an explicit operator similar to duplicate elimination in existing database systems.

## CROSS REFERENCE*

Temporal Data Model, Temporal Database, Time Domain, Time Interval, Time Period, Valid Time, Snapshot equivalence

## RECOMMENDED READING

~~**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**~~


[1] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

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

[3] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Mumbai (Bombay), India, September 1996.

[4] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann Publishers, 1995.

[5] C. Leung and R. Muntz. Query Processing for Temporal Databases. In *Proceedings of the International Conference on Data Engineering*, February 1990.

[6] T. Y. C. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing, Zürich, Switzerland, September 1995. Springer Verlag.

[7] N. Lorentzos and R. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 15(3), 1988.

[8] N. A. Lorentzos and Y. G.Mitsopoulos. Sql extension for interval data. *IEEE Trans. Knowl. Data Eng.*, 9(3):480–499, 1997.

[9] R. Sadeghi, W. B. Samson, and S. M. Deen. HQL – A Historical Query Language. Technical report, Dundee College of Technology, Dundee, Scotland, September 1987.

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

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

[12] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1993.

# Temporal Compatibility

Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass
Free University of Bozen-Bolzano, Italy, University of Aalborg, Denmark,
and University of Arizona, USA

**SYNONYMS**

none

**DEFINITION**

*Temporal compatibility* captures properties of temporal languages with respect to the nontemporal languages that they extend. Temporal compatibility, when satisfied, ensures a smooth migration of legacy applications from a non-temporal system to a temporal system. Temporal compatibility dictates the semantics of legacy statements and constrains the semantics of temporal extensions to these statements, as well as the language design.

**HISTORICAL BACKGROUND**

Since the very early days of temporal database research, the compatibility with legacy languages and systems have been considered, but the first comprehensive investigation was reported by Bair et al. [2]. Compatibility issues are common for work done in the context of systems and commercial languages, such as SQL or Quel. Theoretical or logic-based approaches usually do not explore compatibility notions since they tend to strictly separate temporal from nontemporal structures.

**SCIENTIFIC FUNDAMENTALS**

## Motivation

Most data management applications manage time-referenced, or temporal, data. However, these applications typically run on top of relational or object-relational database management systems (DBMSs), such as DB2, Oracle, SQL Server, and MySQL, that offer only little built-in support for temporal data management. Organizations that manage temporal data may benefit from doing so using a DBMS with built-in temporal support. Indeed, it has been shown that using a temporal DBSM in place of a non-temporal DBMS may reduce the number of lines of query language code by a factor of three, with the conceptual complexity of application development decreasing even further [13].

Then, what hinders an organization from adopting a temporal DBMS? A key observation is that an organization is likely to already have a portfolio of data management applications that run against a non-temporal DBMS. In fact, the organization is likely to have made very large investments in its legacy systems, and it depends on the functioning of these systems for the day-to-day operation of its business.

It should be as easy as possible for the organization to migrate to a temporal DBMS. It would be attractive if all existing applications would simply work without modification on the temporal DBMS. This would help protect the organization's investment in its legacy applications. The opposite, that of having to rewrite all legacy applications, is a daunting proposition.

However, this type of compatibility is only the first step. The next step is to make sure that legacy applications can coexist with new applications that actually exploit the enhanced temporal support of the new DBMS. These applications may query and modify the same (legacy) tables. It should thus be possible to add a new temporal dimension to existing tables, without this affecting the legacy applications that use these tables.

Next, the organization maintains a large investment in the skill set of its IT staff. In particular, the staff is

skilled at using the legacy query language, typically SQL. The new, temporal query language should leverage this investment, by making it easy for the application programmers to write temporal queries against temporal relations.
Below, four specific compatibility properties that aim to facilitate the migration from a non-temporal DBMS to a temporal DBMS are considered.

## Upward Compatibility

The property of upward compatibility states that all language statements expressible in the underlying non-temporal query language must evaluate to the same result in the temporal query language, when evaluated on non-temporal data.
Figure 1 illustrates this property. In the figure, a conventional table is denoted with a rectangle. The current state of this table is the rectangle in the upper-right corner. Whenever a modification is made to this table, the previous state is discarded; hence, at any time, only the current state is available. The discarded prior states are denoted with dashed rectangles; the right-pointing arrows denote the modifications that took the table from one state to the next.

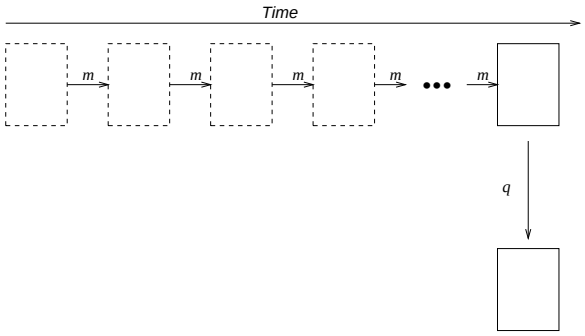



Figure 1: Upward Compatible Queries

When a query $q$ is applied to the current state of a table, a resulting table is computed, shown as the rectangle in the bottom right corner. While this figure only concerns queries over single tables, the extension to queries over multiple tables is clear.
As an example, consider a hypothetical temporal extension of the conventional query language SQL [9]. Upward compatibility states that (1) all instances of tables in SQL are instances of tables in our extension, (2) all SQL modifications to tables in SQL result in the same tables when the modifications are evaluated according to the semantics of the extension, and (3) all SQL queries result in the same tables when the queries are evaluated according to the extension.
By requiring that a temporal extension to SQL is a strict superset (i.e., only *adding* constructs and semantics), it is relatively easy to ensure that the extension is upward compatible with SQL. TOSQL [1], TSQL [10], HSQL [11], IXSQL [7], TempSQL [5], and TSQL2 [12] were designed to satisfy upward compatibility.
While upward compatibility is essential in ensuring a smooth transition to a new temporal DBMS, it does not address all aspects of migration. It only ensures the operation of existing legacy applications and does not address the coexistence of these with new applications that exploit the improved temporal support of the DBMS.

## Temporal Upward Compatibility

The property of temporal upward compatibility (TUC) addresses the coexistence of legacy and new applications. Assume an existing or new application needs support for the temporal dimension of the data in one or more of the existing tables that record only the current state. This is best achieved by changing the snapshot table to become a temporal table. It is undesirable to be forced to change the application code that accesses the snapshot table when that table is made temporal. TUC states that conventional queries on temporal data yield the same results as do the same queries on a conventional database formed by taking a timeslice at "now." TUC applies also to modifications, views, assertions, and constraints [2].

Temporal upward compatibility is illustrated in Figure 2. When temporal support is added to a table, the history is preserved and modifications over time are retained. In the figure, the rightmost dashed state was the current state when the table was made temporal. All subsequent modifications, denoted again by arrows, result in states that are retained, and thus are represented by solid rectangles. Temporal upward compatibility ensures that the states will have identical contents to those states resulting from modifications of the snapshot table. The query $q$ is a conventional SQL query. Due to temporal upward compatibility, the semantics of this query must not change if it is applied to a temporal table. Hence, the query only applies to the current state, and a snapshot table results.

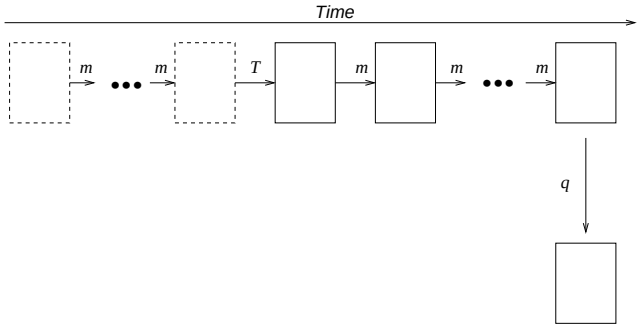



Figure 2: Temporal Upward Compatibility

Most temporal languages were not designed with TUC in mind, and TOSQL [1], TSQL [10], HSQL [11], IXSQL [7], and TSQL2 [12] do not satisfy TUC. The same holds for temporal logics [4]. TempSQL [5] introduces a concept of different types of users, classical and system user. TempSQL satisfies TUC for classical users. ATSQL [3] has been designed to satisfy TUC.

## Snapshot Reducibility

This third property states that for each conventional query, there is a corresponding temporal query that, when applied to a temporal relation, yields the same result as the original snapshot query when applied separately to every snapshot state of the temporal relation.

Graphically, snapshot reducibility implies that for all conventional query expressions $q$ in the snapshot model, there must exist a temporal query $q^t$ in the temporal model so that for all $db^t$ and for all $c$, the commutativity diagram shown in Figure 3 holds.

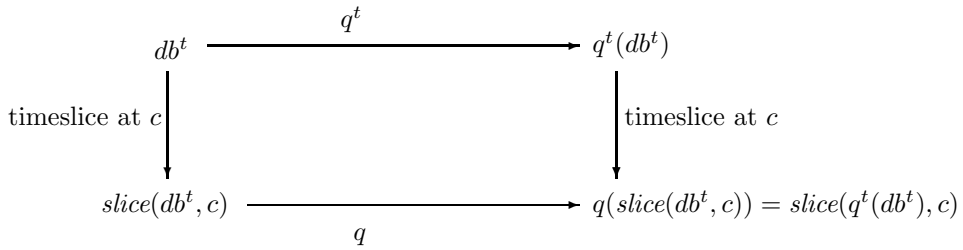



Figure 3: Snapshot Reducibility

This property requires that each query $q$ (or operator) in the snapshot model has a counterpart $q^t$ in the temporal model that is snapshot reducible with respect to the original query $q$. Observe that $q^t$ being snapshot reducible with respect to $q$ poses no syntactical restrictions on $q^t$. It is thus possible for $q^t$ to be quite different from $q$, and $q^t$ might be very involved. This is undesirable, the temporal model should be a straightforward extension of the snapshot model.

Most languages satisfy snapshot reducibility, but only because corresponding non-temporal and temporal statements do not have to be syntactically similar. This allows the languages to formulate for each nontemporal statement a snapshot reducible temporal statement, possibly a very different and complex statement.

## Sequenced Semantics

This property addresses the shortcoming of snapshot reducibility: it requires that $q^t$ and $q$ be syntactically identical, modulo an added string.

Figure 4 illustrates this property. This figure depicts a temporal query, $q'$, that, when applied to a temporal table (the sequence of values across the top of the figure), results in a temporal table, which is the sequence of values across the bottom.

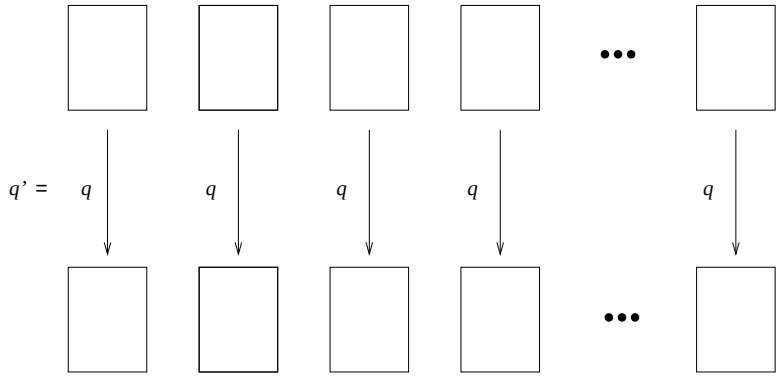



Figure 4: Sequenced Semantics

The goal is to ensure that an application programmer who is familiar with the conventional query language is able to easily formulate temporal generalizations of conventional queries using the temporal query language. This is achieved if a query $q$ can be made temporal by simply adding a string to it. The syntactical similarity requirement of sequenced semantics makes this possible. Specifically, the meaning of $q'$ is precisely that of applying the analogous *non-temporal* query $q$ on each value of the argument table (which must be temporal), producing a state of the result table for each such application.

Most temporal languages do not offer sequenced semantics. As an exception, ATSQL [3] prepends the modifier SEQUENCED, together with the time dimension (valid time or transaction time, or both), to nontemporal statements to obtain their snapshot reducible generalizations. Temporal Logic [4] satisfies sequenced semantics as well: the original nontemporal statement yields sequenced semantics when evaluated over a corresponding temporal relation.

### KEY APPLICATIONS*

Temporal compatibility properties such as those covered here are important for the adoption of temporal database technology in practice. The properties are important because temporal technology is likely to most often be applied in settings where substantial investments have already been made in database management staff and applications. The properties aim at facilitating the introduction of temporal database technology in such settings.

Properties such as these are easily as crucial for the successful adoption of temporal database technology as is highly sophisticated support for the querying of time-referenced data.

Given the very significant decrease in code size and complexity for temporal applications that temporal database technology offers, it is hoped that other DBMS vendors will take Oracle's lead and incorporate support for temporal databases into their products.

### FUTURE DIRECTIONS

Further studies of compatibility properties are in order. For example, note that temporal upward compatibility addresses the case where existing tables are snapshot tables that record only the current state. However, in many cases, the existing tables may already record temporal data using a variety of ad-hoc formats. The challenge is then how to migrate such tables to real temporal tables while maintaining compatibilities.

Next, it is felt that much could be learned from conducting actual case studies of the migration of real-world legacy applications to a temporal DBMS.

### CROSS REFERENCE*

Period-Stamped Data Models, Point-Stamped Data Models, Temporal Data Model, Temporal Database,

Temporal Element, Time Domain, Time Interval, Time Period, Temporal Query Languages, Valid Time, Snapshot Equivalence, Sequenced Semantics

## RECOMMENDED READING


[1] Gadi Ariav, "A Temporally Oriented Data Model", *ACM Transactions on Database Systems* 11(4):499-527, December, 1986.

[2] John Bair, Michael Böhlen, Christian S. Jensen, and Richard T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)*, Vol. 39, No. 1, February 1997, pp. 25–34.

[3] Michael H. Böhlen, Christian S. Jensen and Richard T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems* 25(4), December 2000.

[4] Jan Chomicki, David Toman, and Michael H. Böhlen, "Querying ATSQL Databases with Temporal Logic," *ACM Transactions on Database Systems* 34, June 2001.

[5] Shashi K. Gadia and Sunil S. Nair, "Temporal Databases: A Prelude to Parametric Data," in **Temporal Databases: Theory, Design, and Implementation**, Abdullah Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass (eds), pp. 28–66, Benjamin/Cummings Publishing Company, 1993.

[6] Christian S. Jensen, Michael D. Soo and Richard T. Snodgrass, "Unifying Temporal Data Models via a Conceptual Model," *Information Systems*, Vol. 19, No. 7, December 1994, pp. 513–547.

[7] Nikos A. Lorentzos and Yannis G.Mitsopoulos, "SQL Extension for Interval Data," *IEEE Trans. Knowl. Data Eng.* 9(3):480–499, 1997.

[8] McKenzie, E. and R. T. Snodgrass, "An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases," *ACM Computing Surveys*, Vol. 23, No. 4, December 1991, pp. 501–543.

[9] Jim Melton and Alan R. Simon, **Understanding the New SQL: A Complete Guide**. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[10] Shamkant Navathe and Rafi Ahmed, "Temporal Extensions to the Relational Model and SQL," in **Temporal Databases: Theory, Design, and Implementation**, Abdullah Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass (eds), pp. 92–109, Benjamin/Cummings Publishing Company, 1993.

[11] Nandlal Sarda, "HSQL: A Historical Query Language," in **Temporal Databases: Theory, Design, and Implementation**, Abdullah Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass (eds), Benjamin/Cummings Publishing Company, 1993.

[12] Richard T. Snodgrass, **The TSQL2 Temporal Query Language**, Kluwer Academic Publishers, Boston, 1995.

[13] Richard T. Snodgrass, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, July 1999.

# TEMPORAL CONCEPTUAL MODELS

Vijay Khatri
*Operations and Decision Technologies Department*
*Kelley School of Business*
*Indiana University*
*Bloomington, Indiana, USA*
*vkhatri@indiana.edu*
URL: http://mypage.iu.edu/~vkhatri/

## SYNONYMS
None

## DEFINITION
A *conceptual model* provides a notation and formalism that can be used to construct a high-level, implementation-independent description of selected aspects of the "real world," termed a miniworld. This process is called conceptual modeling, and the resulting description is referred to as a *conceptual schema.* Conceptual modeling is an important part of systems analysis and design. A *temporal conceptual model* provides a notation and formalism with built-in support for capturing temporal aspects of a miniworld during conceptual design.

## HISTORICAL BACKGROUND
*Temporal applications* need to represent data semantics not only related to "what" is important for the application, but also related to "when" it is important. The history of temporal conceptual models can be viewed in terms of two generations. The *first generation temporal conceptual models*, e.g., [2, 14], provide support for only *user-defined time*; see Figure 1a for an example. In contrast to the first generation, the *second generation temporal conceptual models*, e.g., [5, 9, 16], provide varying degree of support for temporal aspects; see Figure 1b for an example. Because the first generation temporal conceptual models provide support for representation of only user-defined time, they may be thought of as "almost" time-agnostic conceptual models; on the other hand, second generation temporal conceptual models that support temporal semantics, e.g., event, state, valid time, transaction time, may be construed as time-aware conceptual models.

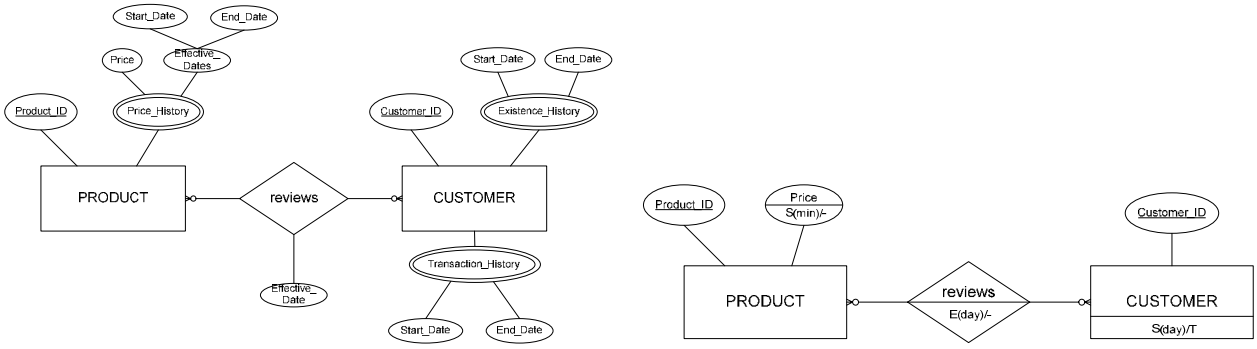
## SCIENTIFIC FUNDAMENTALS
To highlight core concepts developed in the research related to temporal conceptual models, a framework of linguists is adopted that studies symbols with respect to three dimensions: syntactics, semantics and pragmatics [13]. The *syntactics* dimension includes formal relation between symbols, the *semantics* dimension involves the study of symbols in relation to the designatum (i.e., what the sign refers to) and the *pragmatics* dimension includes the relation between symbols and the interpreter.

In the following, a motivating example is employed to differentiate between first and second generation temporal conceptual models. Core concepts related to temporal conceptual models are described using syntactics, semantics and pragmatics.

### Motivating Example
Figure 1a (first generation temporal conceptual model) provides an example of an ER schema that requires preserving the history of "prices" (of PRODUCT). Additionally, there is another entity type, CUSTOMER, whose existence needs to be modeled; the existence history needs to take into account both when the customer exists in the real world (Existence_History) and when

the customer was added to the database (Transaction_History). Further, a CUSTOMER "reviews" PRODUCTs and the Effective_Date on which the PRODUCT was reviewed needs to be captured as well. Because the first generation conceptual models do not provide a mechanism to represent temporal concepts, e.g., valid time, transaction time, event and state, these are all represented using only user-defined time. For example, the schema cannot differentiate Existence_History from Transaction_History, which are both represented simply as multi-valued attributes (double-lined ellipse). Additionally, the database analyst needs to make ad-hoc decisions related to granularity of a user-defined attribute such as Transaction_History. Start_Date during implementation. As a result of the lack of a mechanism for directly mapping the miniworld to its representation, database designers are left to discover, design, and implement the temporal concepts in an ad-hoc manner.

The second generation temporal conceptual schema, referred to as the ST USM (geoSpatio-Temporal Unifying Semantic Model) schema [9] shown in Figure 1b employs a textual string to represent temporal semantics. For example, "Price" is associated with valid time, which is represented as state ("S") with granularity of "min"(ute); further, the transaction time related to price is not relevant ("-"). The temporal semantics associated with "Price" are therefore represented by a textual string of valid time state ("S"), followed by a slash ("/"), followed by the specification of transaction time ("-"): "S (min)/- ". Because both the existence (or valid) time and transaction time need to be recorded for the entity type, CUSTOMER, the annotation string for CUSTOMER is specified as "S(day)/T". Note that the granularity of transaction time is not specified because it is system-defined. A CUSTOMER "reviews" a PRODUCT at a certain point in time (event, E), captured to the granularity of day ("E(day)/-").




a) First generation b) Second generation

Figure 1: Examples of the two generations of temporal conceptual models

In summary, while the first generation temporal conceptual models provide a mechanism to represent only user-defined time, the second generation temporal conceptual models provide a mechanism to represent temporal data semantics. The readers are referred to Gregersen and Jensen [6] for a survey on various temporal conceptual models of the second generation.

### Syntactics

Prior research has employed both graphical (see, for example, the TimeER Model [6]) and textual (see, for example, the Temporal Entity Relationship Model, TERM [11]) syntax to represent the temporal data semantics.

While some of graphical temporal conceptual models have changed the semantics of the constructs of conventional conceptual models (see, for example, [2]), others have proposed a new formalism for representing temporal aspects. The Temporal EER (TEER) Model [3] gave

*new* meaning to *extant* ER modeling constructs such as the entity type, the attribute and the relationship; for example, each entity of an entity type is associated with a temporal element that represents the lifespan of the entity, i.e., the semantics of an entity type in a conventional conceptual model was changed. On the other hand, most of the graphical temporal conceptual models propose *new* constructs that represent the temporal aspects.

Prior research has employed two ways to graphically represent temporal aspects using new constructs; they are referred to as augmented (see, for example, the TimeER Model [6] and ST USM [9]) and standalone (see, for example, the Relationships, Attributes, Keys and Entities (RAKE) Model [4]). The *augmented approaches* construe second generation conceptual schemas as "constrained" first generation schemas. For example, ST USM employs a "shorthand" for temporal semantics that is represented as annotations (see, for example, Figure 1); however, the semantics of a second generation schema (ST USM schema) can be "unwrapped" using a first generation schema (USM schema) and a set of constraints. The readers are referred to [9] for examples of and procedure for "unwrapping" of the semantics of an annotated schema. In contrast, the *standalone approaches* suggest *new* constructs for representing the temporal aspects. The *augmented approaches* provide a mechanism for capturing temporal data semantics at the second level of abstraction; such approaches *deliberately* defer elicitation of the temporal data semantics ("when") from the first level of abstraction that focuses on "what" is important for the application. In contrast, the *standalone approaches* provide a single level of abstraction for representing both "what" and "when."

Having outlined different syntax adopted by various conceptual models, the temporal semantics that need to be captured in a temporal conceptual model are described next.

## Semantics

Based on [8], definitions of different temporal aspects that need to be represented in a temporal conceptual model are outlined below.

An *event* occurs at a point in time, that is, it has no duration (for example, a special promotion for a product is scheduled on Christmas Eve this year (2007-12-24)), while a *state* has duration (for example, a certain price for a product is valid from 5:07 PM on 2005-11-11 to 5:46 PM on 2007-1-11).

Facts can interact with time in two orthogonal ways, resulting in transaction time and valid time. *Transaction time* links a fact to the time that it becomes current in the database, and implies the storage of versions of data. The data semantics of transaction time associated with a fact require that the fact can exist in certain time periods in the past until *now* (state). *Valid time* is used to record the time at which a particular fact is true in the real world and implies the storage of histories related to facts. The data semantics of valid time associated with a fact imply that the fact can exist at certain points in time (events) or in certain time periods (states), in the past, the present, or the future.

*Granularities*, which are intrinsic to temporal data, provide a mechanism to hide details that are not known or not pertinent to an application. Day, minute, and second are examples of temporal granularities related to the Gregorian calendar. The price history for a manufacturing application may, for example, be associated with a temporal granularity of "day," while the representation of price history for a stock market application may require a temporal granularity of "minute" or even "second."

## Pragmatics

Prior research suggests that "effective exchange of information between people and machines is easier if the data structures that are used to organize the information in the machine correspond in a natural way to the conceptual structures people use to organize the same information" [12]. Three criteria play a role in how an "interpreter" (users) interacts with "symbols" (conceptual schema): 1) "internal" representation; 2) snapshot reducibility; 3) upward compatibility. While snapshot reducibility and upward compatibility may be rooted in syntactics and semantics, they affect the pragmatic goal, comprehension.

### *Internal Representation*

All human knowledge is stored as abstract conceptual propositions. Based on propositions, Anderson and Bower's [1] Human Associative Model (HAM) represents information in the long-term memory as shown in Figure 2. A *proposition* is an assertion about the real world that is composed of a *fact* and *context* (associated with the fact). A *subject* and *predicate* correspond with a topic and a comment about the topic. For some applications, the context in which the fact is true can be the key to reasoning about the miniworld. This context in turn is composed of *time* and *location* associated with the fact. Note that the "context" element is orthogonal to the "fact" element and specifies the temporal reality for which the fact is true.[1] An augmented approach that segregates "what" from "when" corresponds with the way humans naturally organize temporal information and should, thus, support effective exchange of information.

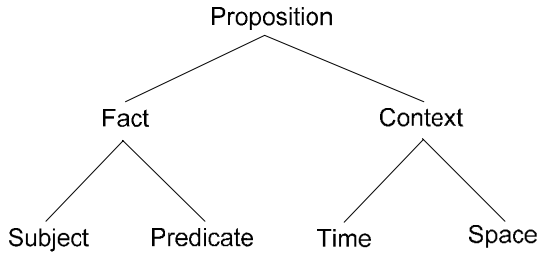



Figure 2: Internal representation of human knowledge; adapted from HAM Model [1]

### *Snapshot reducibility*

Snapshot reducibility implies a "natural" generalization of the syntax and semantics of extant conventional conceptual models, e.g., ER Model [2], for incorporating the temporal extension. Snapshot reducibility ensures that the semantics of a temporal model are understandable "in terms of" the semantics of the conventional conceptual model. Here, the overall objective is to help ensure minimum additional investment in a database analyst training.

For example, in a "conventional" conceptual model a *key attribute* uniquely identifies an entity (at a point in time). A *temporal key* implies uniqueness at *each point in time*. As may be evident, the semantics of a temporal key here are implied by the semantics of a key in a "conventional" conceptual model.

### *Upward compatibility*

Upward compatibility refers to the ability to render a conventional conceptual schema temporal without impacting or negating that legacy schema, thus, protecting investments in the existing schemas. It also implies that both the legacy schemas and the temporal schemas can co-exist.

[1] This paper does not cover spatial aspects; see [9] for details on a spatio-temporal conceptual model.

Upward compatibility requires that the syntax and semantics of the traditional conceptual model remain unaltered. An augmented approach that extends conventional conceptual models would ensure upward compatibility.

## Summary

Because one of the important roles of conceptual modeling is to support user-database analyst interaction, the linguistics-based framework of evaluation is broader: it not only includes syntactics and semantics but also includes cognitive aspects in conceptual design (pragmatics). Table 1 summarizes the evaluation of a first generation and a few second generation temporal conceptual models.

| | Syntactics | Semantics | | | | Pragmatics | | |
|---|---|---|---|---|---|---|---|---|
| | *Syntax* | *User-defined time* | *Valid time and transaction time* | *Event and state* | *Granularity* | *Consideration of internal representation* | *Upward compatibility* | *Snapshot reducibility* |
| *First Generation* | | | | | | | | |
| ER Model [2] | • "What": Graphical<br>• "When": NA | Yes | No | No | No | NA | NA | NA |
| *Second Generation* | | | | | | | | |
| TERM [11] | • "What": Textual<br>• "When": Textual | Yes | Valid time only | Both | Yes | No | No | No |
| TERC+ [16] | • "What": Graphical<br>• "When": Graphical | Yes | Valid time only | Both | No | No | Yes | Yes |
| TimeER [5] | • "What": Graphical<br>• "When": Textual | Yes | Both | Both | No | No | Yes | Yes |
| ST USM [9] | • "What": Graphical<br>• "When": Textual | Yes | Both | Both | Yes | Yes | Yes | Yes |

Table 1: Summary of a sample of first and second generation temporal conceptual models

## KEY APPLICATIONS

There are several applications of this research, both for researchers and practitioners. 1) A temporal conceptual model can help support elicitation and representation of temporal data semantics during conceptual design. 2) A temporal conceptual schema can, thus, be the basis for the logical schema and the database. 3) A temporal conceptual modeling approach can be used as the basis for developing a design-support environment. Such a design support environment can be integrated with tools such as ERWin.[2]

## FUTURE DIRECTIONS* (optional)

Future research should explore how the temporal schema can be used as the canonical model for information integration of distributed temporal databases. A temporal conceptual model should also be extended to incorporate schema versioning.

[2] http://www.ca.com/us/products/product.aspx?id=260

While an initial user study has been conducted [10], future research should further evaluate temporal conceptual modeling using, e.g., protocol analysis. Studies that address *how* problem solving occurs focus on "opening up the black box" that lies between problem-solving inputs and outputs; that is, such studies investigate what happens during individual problem solving (*isomorphic approach*) rather than simply observing the effects of certain stimuli averaged over a number of cases, as in traditional studies (*paramorphic approach*) [7]. The most common approach to opening up the black box is to examine the characteristics of the problem-solving process using protocol analysis.

## EXPERIMENTAL RESULTS* (optional)

To evaluate the augmented temporal conceptual design approach, a user experiment [10] views conceptual schema comprehension in terms of matching the *external problem representation* (i.e., conceptual schema) with *internal task representation*, based on the theory of HAM and the theory of cognitive fit [15]. The study suggests that the similarity between annotated schemas (external representation) and the HAM model of internal memory results in cognitive fit, thus, facilitating comprehension of the schemas.

## DATA SETS* (optional)

## URL to CODE* (optional)

## CROSS REFERENCES

Temporal granularity, temporal data model, schema versioning, supporting transaction time, now in temporal databases, sequenced semantics

## RECOMMENDED READING


[1] J. R. Anderson and G. H. Bower, *Human Associative Memory*. Washington, D.C.: V. H. Winston & Sons, 1973.

[2] P. P. Chen, "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions of Database Systems*, vol. 1, no. 1, pp. 9-36, 1976.

[3] R. Elmasri, G. Wuu, and V. Kouramajian, "A Temporal Model and Query Language for EER Databases," in *Temporal Databases: Theory, Design and Implementation*, A. Tansel, Ed.: Benjamin/ Cummings, 1993, pp. 212-229.

[4] S. Ferg, "Modeling the Time Dimension in an Entity-Relationship Diagram," presented at 4th International Conference on the Entity-Relationship Approach, 1985.

[5] H. Gregersen and C. Jensen, "Conceptual Modeling of Time-Varying Information," TIMECENTER Technical Report TR-35, September 10 1998.

[6] H. Gregersen and C. S. Jensen, "Temporal Entity-Relationship Models-A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 3, pp. 464-497, 1999.

[7] P. J. Hoffman, "The paramorphic representation of clinical judgment" *Psychological Bulletin*, vol. 57, no. 2, pp. 116-131, 1960.

[8] C. S. Jensen, C. E. Dyreson, M. Bohlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Kafer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peresi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio, and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts-February 1998 Version," in *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada, Eds.: Springer-Verlag, 1998.

[9] V. Khatri, S. Ram, and R. T. Snodgrass, "Augmenting a Conceptual Model with Geospatiotemporal Annotations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 11, pp. 1324-1338, 2004.

[10] V. Khatri, I. Vessey, S. Ram, and V. Ramesh, "Cognitive Fit between Conceptual Schemas and Internal Problem Representations: The Case of Geospatio-Temporal Conceptual Schema Comprehension," *IEEE Transactions on Professional Communication*, vol. 49, no. 2, pp. 109-127, 2006.
[11] M. R. Klopprogge, "TERM: An Approach to include the Time Dimension in the Entity Relationship Model," presented at 2nd International Conference Entity Relationship Approach, 1981.
[12] M. Moens and M. Steedman, "Temporal Ontology and Temporal Reference," *Computational Linguistics*, vol. 14, no. 2, pp. 15-28, 1988.
[13] C. W. Morris, "Foundations of the Theory of Signs," in *International Encyclopedia of Unified Science*, vol. 1, Second ed: University of Chicago Press, 1955.
[14] S. Ram, "Intelligent Database Design using the Unifying Semantic Model," *Information and Management*, vol. 29, no. 4, pp. 191-206, 1995.
[15] I. Vessey, "Cognitive Fit: A Theory-based Analysis of Graphs Vs. Tables Literature," *Decision Sciences*, vol. 22, no. 2, pp. 219-240, 1991.
[16] E. Zimanyi, C. Parent, S. Spaccapietra, and A. Pirotte, "TERC+: A Temporal Conceptual Model," presented at International Symposium Digital Media Information Base, 1997.

# Temporal Constraints

Peter Revesz
University of Nebraska-Lincoln
revesz@cse.unl.edu

## SYNONYMS

temporal constraints on time points; temporal constraints on sets of time points; temporal constraints on time intervals

## DEFINITION

*Temporal Constraints* describe relationships among variables that refer somehow to time. A set of temporal constraints can be stored in a temporal database, which is queried by temporal queries during problem solving. For example, a set of temporal constraints may form some requirements that must be all satisfied during some scheduling problem.

Most interesting temporal constraints derive from references to time in natural language. Such references typically compare two *time points*, two *sets of time points*, or two *time intervals*. The literature on temporal constraints and this entry focuses on the study of these types of comparative or *binary* constraints.

## HISTORICAL BACKGROUND

The seminal work on temporal intervals is by Allen [1]. Difference Bounded Matrices (see the Section on Scientific Fundamentals) were introduced by Dill [3]. A graph representation of difference constraints and efficient constraint satisfaction problem-based solutions for consistency of difference constraints were presented by Dechter et al. [2]. A graph representation of gap-order constraints and an efficient algebra on them is presented by Revesz [11]. A graph representation of set order constraints and algebra on them is described in [12]. Addition constraints are considered in [12]. The complexity of deciding the consistency of conjunctions of integer addition constraints in Table 1 is from [9].

Periodicity constraint within query languages are considered by Kabanza et al. [4] and Toman and Chomicki [15]. Constraint databases [8, 12] were introduced by Kanellakis et al. [5] with a general framework for constraints that includes temporal constraints. Indefinite temporal constraint databases were introduced by Koubarakis [6]. Linear cardinality constraints on sets were considered by Kuncak et al. [7] and Revesz [13].

Deciding the consistency of conjunctions of rational (or real) difference and inequality constraints was proven tractable by Koubarakis, but the $O(v^3)$ complexity result in Table 1 is from Péron and Halbwachs [10]. The NP-completeness result in Table 1 follows from [14].

## SCIENTIFIC FUNDAMENTALS

*Temporal constraints on time points* express temporal relationships between two time points, which are called also *time instances*. More precisely, let $x$ and $y$ be integer or rational variables or constants representing time points,

and let $b$ be an integer constant. Then some common temporal constraints on time points include the following:

| | | | | |
|---|---|---|---|---|
| **After** : | $x$ | $>$ | $y$ | |
| **Before** : | $x$ | $<$ | $y$ | |
| **Equal** : | $x$ | $=$ | $y$ | |
| **Inequal** : | $x$ | $\neq$ | $y$ | |
| **After or Equal (AoE)** : | $x$ | $\geq$ | $y$ | |
| **Before or Equal (BoE)** : | $x$ | $\leq$ | $y$ | |
| **After by at least b (Gap-Order)** : | $x - y$ | $\geq$ | $b$ | where $b \geq 0$ |
| **Difference** : | $x - y$ | $\geq$ | $b$ | |
| **Potential** : | $x - y$ | $\leq$ | $b$ | |
| **Addition** : | $\pm x \pm y$ | $\geq$ | $b$ | |

In the above table the first six constraints are called *qualitative* constraints, and the last four constraints are called *metric* constraints because they involve a measure $b$ of time units. For example, "a copyright form needs to be filled out before publication" can be expressed as $t_{copyright} < t_{publication}$, where $t_{copyright}$ is the time point when the copyright form is filled out and $t_{publication}$ is the time point when the paper is printed. This constraint could be just one of the requirements in a complex scheduling problem, for example, the process of publishing in a book a collection of research papers. Another temporal constraint may express that "the publication must be at least 30 days after the time of submission," which can be expressed by the constraint $t_{publication} - t_{submission} \geq 30$.

*Temporal constraints on sets of time points* express temporal relationships between two sets of time points. The domain of a set $X$ is usually assumed to be the finite and infinite subsets of the integers. Common temporal constraints between sets of time points include the following:

| | | | | |
|---|---|---|---|---|
| **Equal** : | $X$ | $=$ | $Y$ | |
| **Inequal** : | $X$ | $\neq$ | $Y$ | |
| **Contains (Set Order)** : | $X$ | $\subseteq$ | $Y$ | |
| **Disjoint** : | $X$ | $\cap$ | $Y = \emptyset$ | |
| **Overlap with b elements** : | $\lvert X$ | $\cap$ | $Y\rvert = b$ | |

where $X$ and $Y$ are set variables or constants, $\emptyset$ is the empty set, $b$ is an integer constant, and $|\ |$ is the *cardinality* operator. For example, "The Database Systems class and the Geographic Information Systems class cannot be at the same time" can be expressed as $T_{Database} \cap T_{GIS} = \emptyset$, where $T_{Database}$ is the set of time points (measured in hour units) the Database System class meets, and $T_{GIS}$ is the set of time points the Geographic Information Systems class meets.

*Temporal constraints on time intervals* express temporal relationships between two time intervals. These types of temporal constraints are known as *Allen's Relations* [Allen's Relations] because they were studied first by J.F. Allen [1].

*Other types of temporal constraints:* The various other types of temporal constraints can be grouped as follows: (1) *n-ary temporal constraints.* These generalize the binary temporal constraints to $n$ number of variables that refer to time. While *temporal databases* typically use binary constraints, *constraint databases* [5] use n-ary constraints [Constraint Databases], for example linear and polynomial constraints on $n$ time point variables.

(2) *Temporal periodicity constraints.* Periodicity constraints [4, 15] occur in natural language in phrases such as "every Monday." These constraints are discussed separately in [Temporal Periodicity Constraints].

(3) *Indefinite temporal constraints.* The nature of a temporal constraint can be two types: definite and indefinite. Definite constraints describe events precisely, while indefinite constraints describe events less precisely leaving several possibilities. For example, "Ed had fever between 1 PM and 9 PM but at no other times" is a definite constraint because it ~~tells us for~~ relays each time instance whether Ed had a fever or not. On the other hand, "Ed had fever for some time during 1 PM and 9 PM" is an indefinite constraint because it allows the possibility that Ed

had fever at 5 PM and at no other times, or another possibility that he had fever between 1 PM and 4 PM and at no other times. Hence it does not ~~tell us~~ relay whether Ed had fever at 5 PM.

Conjunctions of temporal constraints can be represented in a number of ways. Consider a scheduling problem where one needs to schedule the events $e_1, \ldots, e_6$. Suppose that there are some scheduling requirements of the form "some (second) event occurs after another (first) event by at least $b$ days." For example, each row of the following table, which is a temporal database relation, represents one such scheduling constraint.

**Scheduling_Requirements**

| Second_Event | First_Event | After_By |
|---|---|---|
| 0 | $e_1$ | 5 |
| 0 | $e_2$ | 2 |
| $e_1$ | $e_3$ | -2 |
| $e_1$ | $e_5$ | -9 |
| $e_2$ | $e_1$ | -6 |
| $e_3$ | $e_2$ | 3 |
| $e_3$ | $e_4$ | -3 |
| $e_4$ | $e_3$ | -5 |
| $e_5$ | $e_4$ | 3 |
| $e_5$ | $e_6$ | 3 |
| $e_6$ | 0 | 1 |

Many queries are easier to evaluate on some alternative representation of the above temporal database relation. Some alternative representations that may be used within a temporal database system are given below.

**Conjunctions of constraints:** Let $x_1, \ldots, x_6$ represent,respectively, the times when events $e_1, \ldots, e_6$ occur. Then the **Scheduling_Requirements** relation can be represented also by the following conjunction of difference constraints:

$$0 - x_1 \geq 5, \quad 0 - x_2 \geq 2, \quad x_1 - x_3 \geq -2, \quad x_1 - x_5 \geq -9, \quad x_2 - x_1 \geq -6, \quad x_3 - x_2 \geq 3,$$
$$x_3 - x_4 \geq -3, \quad x_4 - x_3 \geq -5, \quad x_5 - x_4 \geq 3, \quad x_5 - x_6 \geq 3, \quad x_6 - 0 \geq 1$$

**Labeled Directed Graphs:** In general, the graph contains $n + 1$ vertices representing all the $n$ variables and 0. For each difference constraint of the form $x_i - x_j \geq b$, the graph contains also a directed edge from the vertex representing $x_j$ to the vertex representing $x_i$. The directed edge is labeled by $b$.

**Difference Bound Matrices:** Conjunctions of difference constraints can be represented also by *difference bound matrices (DBMs)* of size $(n + 1) \times (n + 1)$, where $n$ is the number of variables. For each difference constraint of the form $x_i - x_j \geq b$, the DBM contains the value $b$ in its $(j, i)$th entry. The default value is $-\infty$. For example, above set of difference constraints can be represented by the following DBM:

| | 0 | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|---|---|---|---|---|---|---|---|
| 0 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | 1 |
| $x_1$ | 5 | $-\infty$ | $-6$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |
| $x_2$ | 2 | $-\infty$ | $-\infty$ | 3 | $-\infty$ | $-\infty$ | $-\infty$ |
| $x_3$ | $-\infty$ | $-2$ | $-\infty$ | $-\infty$ | $-5$ | $-\infty$ | $-\infty$ |
| $x_4$ | $-\infty$ | $-\infty$ | $-\infty$ | $-3$ | $-\infty$ | 3 | $-\infty$ |
| $x_5$ | $-\infty$ | $-9$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |
| $x_6$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | 3 | $-\infty$ |

A major question about temporal constraint formulas is whether they are *consistent* or *satisfiable*. A formula is consistent or satisfiable if and only if it has at least one substitution for the variables that makes the formula true. Otherwise, it is called *inconsistent* or *unsatisfiable*. For example, if the conjunction of difference constraints that describe the **Scheduling_Requirements** table is inconsistent, then there is no schedule of the events $e_1, \ldots, e_6$ such that all the requirements are satisfied.

Table 1 summarizes some computational complexity results, in terms of $v$ the number of vertices and $e$ the number of edges in the graph representation.

**Table 1: Consistency for Conjunctions of Temporal Constraints**

| Case | Temporal Constraints | Integers | Rationals |
|---|---|---|---|
| 1 | After, Before, Equal, Inequal, AoE, BoE | $O(v+e)$ | $O(v+e)$ |
| 2 | Addition | $O(ve)$ | $O(v+e)$ |
| 3 | Inequal, Difference | NP-complete | $O(v^3)$ |

Most complexity results translate deciding the consistency to classical problems on graphs with efficient and well-known solutions. [2] provides a translation to constraint satisfaction problems, which use efficient search heuristics for large sets of constraints. Many operations on DBMs can be defined. These operators include *conjunction* or *merge* of DBMs, *variable elimination* or *projection* of a variable from a DBM, testing *implication* of a DBM by a disjunction of DBMs, and *transitive closure* of a DBM [12].

*Temporal Constraints on Time Intervals*
In general, deciding the consistency of a conjunction of temporal constraints on intervals is NP-complete. Many computational complexity results for temporal constraints on time intervals follow from the complexity results for temporal constraint on time points. In particular, any conjunction of pointisable temporal constraints on time intervals can be translated to a conjunction of temporal constraints on time points. After the translation, the consistency can be tested as before.

## KEY APPLICATIONS

Temporal constraints are used in scheduling, planning, and temporal database querying [Temporal Databases Queries]. Temporal database queries usually take the form of SQL or Datalog combined with temporal constraints. Examples include Datalog with gap-order constraints [11] and Datalog with periodicity constraints [15].

## FUTURE DIRECTIONS

There are still many open problems on the use of temporal constraints in temporal database query languages. An important problem is finding efficient indexing methods for conjunctions of temporal constraints. The combination of temporal constraints with spatial constraints is an interesting area within spatiotemporal databases [Spatiotemporal Databases] and constraint databases [12].

## CROSS REFERENCE

database query languages; indexing; spatiotemporal databases; temporal dependencies; temporal integrity constraints; temporal periodicity

## RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.


[1] J. F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 26, 1983.

[2] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1–3):61–95, 1991.

[3] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[4] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. *Journal of Computer and System Sciences*, 51(1):1–25, 1995.

[5] P. C. Kanellakis, G. M. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, 1995.

[6] M. Koubarakis. The complexity of query evaluation in indefinite temporal constraint databases. *Theoretical Computer Science*, 171(1–2):25–60, 1997.

[7] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *Proc. 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 260–277. Springer-Verlag, 2005.

[8] G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer-Verlag, 2000.

[9] S. K. Lahiri and M. Musuvathi. An efficient decision procedure for utvpi constraints. In *5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2005.

[10] M. Péron and N. Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In *Proc. 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 268–282. Springer-Verlag, 2007.

[11] P. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–49, 1993.

[12] P. Revesz. *Introduction to Constraint Databases*. Springer-Verlag, 2002.

[13] P. Revesz. Quantifier-elimination for the first-order theory of Boolean algebras with linear cardinality constraints. In *Proc. 8th Conference on Advances in Databases and Information Systems*, volume 3255 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2004.

[14] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *Proc. IEEE International Conference on Very Large Databases*, pages 64–72, 1980.

[15] D. Toman and J. Chomicki. Datalog with integer periodicity constraints. *Journal of Logic Programming*, 35(3):263–90, 1998.

# Temporal Database

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

historical database; time-oriented database

## DEFINITION

A temporal database is a collection of time-referenced data. In such a database, the time references capture some temporal aspect of the data; put differently, the data is timestamped. Two temporal aspects are prevalent. The time references may capture either the past and current states of the database, yielding a transaction-time database, or may capture states of the reality being modeled by the data, yielding a valid-time database, or they may capture both aspects of the data, yielding a bitemporal database.

## HISTORICAL BACKGROUND

"Time" is a fundamental concept that pervades all aspects of daily life. As one indicator, a recent study by Oxford University Press found that the word "time" is the most commonly used noun in the English language. The nouns "year" and "day" rank third and fifth in the study.

Moving to a database context, the capture and representation of time-varying information go back literally thousands of years. The Egyptians and Syrians carved records into stone walls and pyramids of inventories of grain over many years. But it has been only in the last few decades that the advent of increasingly inexpensive and voluminous digital storage has enabled the computerized management of increasingly large volumes of time-referenced data.

Temporal databases have been the subject of intense study since the early 1980's. A series of bibliographies on temporal databases enumerates the thousands of refereed papers that have been written on the subject (the series started with Boulour's paper [5]; the most recent in the series is by Wu et al. [17]). There have also been more specialized bibliographies [2, 9, 12].

The first temporal database conference was held in Sofie Antipolis, France in 1987 [1]. Two workshops, the Temporal Database Workshop in Arlington, Texas in 1993 and in Zürich, Switzerland in 1995 [8], were held subsequently. The premier conferences on the topic are the International Symposium on Temporal Representation and Reasoning (*TIME*) (held annually since 1994), the International Symposium on Spatial and Temporal Databases (*SSTD*) (held biannually), and the Spatio-Temporal Database Management (*STDBM*) series (three up through 2006).

A seminal book collecting much of the results to date in this field appeared in 1993 [15]. Several surveys have been written on the topic [4, 3, 7, 6, 10, 11, 13, 14, 16]. Temporal databases were covered in detail in an advanced database textbook [18].

## SCIENTIFIC FUNDAMENTALS

Time impacts all aspects of a database technology, including database design (at the conceptual, logical, and physical levels) and the technologies utilized by a database management system, such as the query and modification languages, the indexing techniques and data structure utilized, the query optimization and query evaluation techniques, and transaction processing.

The entries related to temporal databases go into more detail about these aspects. The following provides an organization on those entries (which are indicated in *italics*).

***General Concepts***

- Philosophers have thought hard about time (*temporal concepts in philosophy*).
- Two general temporal aspects of data attract special attention: *valid time* and *transaction time*.
- The *time domain* can be differentiated along several aspects: its structure, e.g., linear or branching; discrete versus continuous; boundedness or infinite.
- Just as multiple version of data may be stored, independently, the schemas can be versioned (*schema versioning*).
- The concept of "now" is important (*now in temporal databases*).

***Temporal Data Models***

- *Temporal conceptual models* generally extend and existing conceptual model, such as one of the variants of the Entity-Relationship model.
- *Temporal logical models* generally extend the relational model or an object-oriented model (*temporal object-oriented models*) or XML (*temporal XML*).
- Data can be associated with time in several ways: with time points (*point-stamped temporal models*) or time periods (*period-stamped temporal models*); these may capture valid and/or transaction time; and the associations of the data with the time values may carry probabilities (*temporal probabilistic models*.
- The time values associated with the data are characterized by their *temporal granularity*, and they may possess *temporal indeterminacy* and *temporal periodicity*.
- Data models incorporate *temporal constraints*, *temporal integrity constraints*, and *temporal dependencies*.

***Temporal Query Languages***

- Most *temporal query languages* are based on the relational algebra or calculus. Not surprisingly, much attention has been given to the design of user-level temporal query languages, notably *SQL-based temporal query languages*. For such languages, different notions of *temporal compatibility* have been an important design consideration.
- *Qualitative temporal reasoning* and *temporal logic in database query languages* provide expressive query facilities.
- *Temporal vacuuming* provides a way to control the growth of an otherwise append-only transaction-time database.
- *TSQL2* and its successor SQL/Temporal provided a way for many in the temporal database community to coordinate their efforts in temporal query language design and implementation.

***Physical Level***

- *Temporal query processing* involves disparate architectures, from *temporal strata* outside the conventional DBMS to adding native temporal support within the DBMS.
- *Supporting transaction time* in an efficient manner in the context of transactions is challenging and generally requires changes to the kernel of a DBMS.
- *Temporal algebras* extend the conventional relational algebra. Some specific operators (e.g., *temporal aggregation*, *temporal coalescing*, *temporal joins*)) have received special attention.
- Temporal storage structures and indexing techniques have also received a great deal of attention (*temporal indexing*).

***Temporal Applications***

- *Temporal access control* uses temporal concepts in database security.
- *Temporal data mining* has recently received a lot of attention.
- *Time series* has also been an active area of research.
- Other applications include temporal constraint satisfaction, support for planning systems, and natural language disambiguation.

The concepts of temporal databases are also making their way into research on data warehousing, OLAP, and data streams.

## KEY APPLICATIONS*

As storage cost decreases, more databases are retaining historical data. The dual of such a decrease is that the cost of deletion is effectively increasing, as the application then has to explicitly make the decision on what to retain and what to delete, and the DBMS has to revisit the data on disk in order to move it. Some have asserted that it may be simpler to simply disallow deletion (except for purging of unneeded records, termed *temporal vacuuming*) within a DBMS, rendering all databases by default temporal databases.

Commercial products are starting to include temporal support. Most notably, the Oracle database management system has included temporal support from its 9i version (see the *TSQL2* entry for more detail). Lumigent's LogExplorer product provides an analysis tool for Microsoft SQLServer logs, to allow one to view how rows change over time (a nonsequenced transaction-time query) and then to selectively back out and replay changes, on both relational data and the schema (it effectively treats the schema as a transaction-versioned schema). aTempo's Time Navigator is a data replication tool for DB2, Oracle, Microsoft SQL Server, and Sybase that extracts information from a database to build a slice repository, thereby enabling image-based restoration of a past slice; these are transaction time-slice queries. IBM's DataPropagator can use data replication of a DB2 log to create both before and after images of every row modification to create a transaction-time database that can be later queried.

## FUTURE DIRECTIONS

While much progress has been made in all of the above listed areas, temporal database concepts and technologies have yet to achieve the desired levels of simplification and comprehensibility. While many of the subtleties of temporal data and access and storage thereof have been investigated, in many cases quite thoroughly, a synthesis is still needed of these concepts and technologies into forms that are usable by novices.

Given the simplifications of data management afforded by built-in support of time in database management systems, it is hoped that DBMS vendors will continue to increase temporal support in their products.

## CROSS REFERENCE*

Now in Temporal Databases, Period-Stamped Temporal Models, Point-Stamped Temporal Models, Qualitative Temporal Reasoning, Schema Versioning, SQL-Based Temporal Query Languages, Supporting Transaction Time, Temporal Access Control, Temporal Aggregation, Temporal Algebras, Temporal Coalescing, Temporal Compatibility, Temporal Concepts in Philosophy, Temporal Conceptual Models, Temporal Constraints, Temporal Data Mining, Temporal Data Models, Temporal Dependencies, Temporal Granularity, Temporal Indeterminacy, Temporal Indexing, Temporal Integrity Constraints, Temporal Joins, Temporal Logic in Database Query Languages, Temporal Logical Models, Temporal Object-Oriented Databases, Temporal Periodicity, Temporal Probabilistic Models, Temporal Query Languages, Temporal Query Processing, Temporal Strata, Temporal Vacuuming, Temporal XML, Time Domain, Time Series, TSQL2

## RECOMMENDED READING


[1] AFCET, *Proceedings of the Conference on Temporal Aspects in Information Systems*, Sofie Antopolis, France, May 1987.

[2] K. K. Al-Tara, Richard T. Snodgrass, and Michael D. Soo, "A Bibliography on Spatio-temporal Databases," *International Journal of Geographic Information Systems*, Vol. 8, No. 1, January-February 1994, pp. 95–103.

[3] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen, "Temporal Databases," in *Handbook of Database Technology*, edited by J. Hammer and M. Schneider, Computer & Information Science Series, Chapman & Hall/CRC Press, to appear.

[4] Michael H. Böhlen and Christian S. Jensen, "Temporal Data Model and Query Language Concepts," in *Encyclopedia of Information Systems*, Vol. 4, pp. 437–453, Academic Press, 2003.

[5] A. Boulour, T. Logenia Anderson, L. J. Dekeyser, and Harry K. T. Wong, "The role of time in information processing: A survey," *ACM SIGMOD Record* 12(3):27–50, April 1982.

[6] Jan Chomicki and David Toman, "Temporal Databases," in *Handbook of Time in Artificial Intelligence*, M.' Fisher et al., editors, Elsevier, 2005.

[7] Jan Chomicki, "Temporal Query Languages: A Survey," in *Proceedings of Temporal Logic: ICTL'94*, volume 827, pages 506–534, D. M. Gabbay and H. J. Ohlbach, eds, Springer-Verlag, 1994.

[8] James Clifford and Alexander Tuzhilin (eds.), *Proceedings of the VLDB International Workshop on Temporal Databases (VLDB)*, Zürich, Switzerland, September, in **Recent Advances in Temporal Databases**, Springer-Verlag, New York, 1995.

[9] Fabio Grandi, "Introducing an annotated bibliography on temporal and evolution aspects in the World Wide Web," *ACM SIGMOD Record* 33(2):84–86, June 2004.

[10] Christian S. Jensen and Richard T. Snodgrass, "Temporal Data Management," *IEEE Transactions on Knowledge and Data Engineering*, 11(1): 36–44, January/February 1999.

[11] Gultekin Özsoyoğlu and Richard T. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995, pp. 513–532.

[12] John F. Roddick and Myra Spiliopoulou, "A Bibliography of Temporal, Spatial and Spatio-Temporal Data Mining Research," *SIGKDD Explorations* 1(1):34–38, January 1999.

[13] Richard T. Snodgrass, "Temporal databases: Status and research directions," *ACM SIGMOD Record* 19(4):83–39, December 1990.

[14] Richard T. Snodgrass, "Temporal Databases," in *Proceedings of the International Conference on GIS: From Space to Territory*, Volume 629, September 1992, Andrew U. Frank, I. Compari, and U. Formentini, eds.

[15] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, **Temporal Databases: Theory, Design, and Implementation**, Database Systems and Applications Series, Benjamin/Cummings Pub. Co., Redwood City, CA, March 1993, 633+xx pages.

[16] Ines F. Vega López, Richard T. Snodgrass, and Bongki Moon, "Spatiotemporal Aggregate Computation: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, February 2005.

[17] Yu Wu, Sushil Jagodia, and X. Sean Wang, "Temporal Database Bibliography Update," **Temporal Databases—Research and Practice**, pp. 338–367, Springer-Verlag, LNCS 1399, 1998.

[18] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari, **Advanced Database Systems**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997, 574+xvii pages.

# TEMPORAL DATA MINING

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
http://www.cs.hku.hk/∼nikos

## SYNONYMS

Time Series Data Mining; Sequence Data Mining; Temporal Association Mining

## DEFINITION

Temporal data mining refers to the extraction of implicit, non-trivial, and potentially useful abstract information from large collections of temporal data. Temporal data are sequences of a primary data type, most commonly numerical or categorical values and sometimes multivariate or composite information. Examples of temporal data are regular time series (e.g., stock ticks, EEG), event sequences (e.g., sensor readings, packet traces, medical records, weblog data), and temporal databases (e.g., relations with timestamped tuples, databases with versioning). The common factor of all these sequence types is the total ordering of their elements. They differ on the type of primary information, the regularity of the elements in the sequence, and on whether there is explicit temporal information associated to each element (e.g., timestamps). There are several mining tasks that can be applied on temporal data, most of which directly extend from the corresponding mining tasks on general data types. These tasks include classification and regression (i.e., generation of predictive data models), clustering (i.e., generation of descriptive data models), temporal association analysis between events (i.e., causality relationships), and extraction of temporal patterns (local descriptive models for temporal data).

## HISTORICAL BACKGROUND

Analysis of time series data has been an old problem in statistics [4], the main application being forecasting for different applications (stock market, weather, etc.) Classic statistical models for this purpose include autoregression and hidden Markov models. The term temporal data mining came along as early as the birth of data mining in the beginning of the 90's. Soon after association rules mining in large databases [1] has been established as a core research problem, several researchers became interested in association analysis in long sequences and large temporal databases (see [12] for a survey). One big challenge in temporal data mining is the large volume of the data, which make traditional autoregression analysis techniques inapplicable. Another challenge is the nature of the data which is not limited to numerical-valued time series, but includes sequences of discrete, categorical, and composite values (e.g., sets). This introduces new, interesting types of patterns, like causality relationships between events in time, partial periodic patterns, and calendric patterns.

## SCIENTIFIC FUNDAMENTALS

Classic data mining tasks, like classification, clustering, and association analysis can naturally be applied on large collections of temporal data. Special to temporal databases, are the extraction of patterns that are frequent during specific temporal intervals and the identification of temporal relationships between values or events in large sequences. In the following, the above problems are discussed in more detail.

### Classification and Clustering

Classification of time series is often performed by nearest neighbor (NN) classifiers [13]. Given a time series $\vec{s}$

of unknown label and a database $\mathcal{D}$ of labeled samples, such a classifier (i) searches in $\mathcal{D}$ for the $k$ most similar time series to $\vec{s}$ and (ii) gives $\vec{s}$ the most popular label in the set of $k$ returned time series. This process involves two challenges: definition of an appropriate *similarity* function to be used by the NN classifier and scalability of classification. The dissimilarity (distance) between two time series is typically quantified by their Euclidean distance or the dynamic time warping (DTW) distance. The reader is referred to the "time series query" entry of this Encyclopedia, for more details on these distance measures and indexing of time series data for efficient similarity retrieval. Like classification, clustering of time series can be performed by applying an off-the-shelf clustering algorithm [7] (e.g., $k$-means), after defining an appropriate distance (i.e., dissimilarity) function.

For sequences of categorical data, Hidden Markov Models (HMM) can be used to capture the behavior of the data. HMM can be used for classification as follows. For each class label, a probabilistic state transition model that captures the probabilities of seeing one symbol (state) after the current one can be built. Then a sequence is given the label determined by the HMM that describes its behavior best.

**Prediction**

For continuous-valued sequences, like time series, regression is an alternative to classification. Regression does not use a fixed set of class labels to describe each sequence, but models sequences as functions, which are more appropriate for predicting the values in the future. Autoregression is a special type of regression, where future values are predicted as a linear combination of recent previous values, assuming that the series exhibits a periodic behavior. Formally, an autoregressive model of order $p$ for a time series $\vec{s} = \{s_1, s_2, \ldots\}$ can be described as follows:

$$s_i = e_i + \sum_{j=1}^{p} \phi_j s_{i-j}, \tag{1}$$

where $\phi_j (1 \leq j \leq p)$ are the parameters of autoregression, and $e_i$ is an error term. The error terms are assumed to be independent identically-distributed random variables (i.i.d.) that follow a zero-mean normal distribution. The main trend of a time series is commonly described by a *moving average* function, which is a smoothed abstraction of the same length. Formally, the moving average of order $q$ for a time series $\vec{s} = \{s_1, s_2, \ldots\}$ can be described as follows:

$$MA(\vec{s})_i = e_i + \sum_{j=1}^{q} \psi_j e_{i-j}, \tag{2}$$

where $\psi_j (1 \leq j \leq q)$ are the parameters of the model. By combining the above two concepts, a time series $\vec{s}$ can be described by an autoregressive moving average (ARMA) model:

$$s_i = e_i + \sum_{j=1}^{p} \phi_j s_{i-j} + \sum_{j=1}^{q} \psi_j e_{i-j}, \tag{3}$$

Autoregressive integrated moving average (ARIMA) is a more generalized model, obtained by integrating an ARMA model. In long time series, periodic behaviors tend to be local, so a common practice is to segment the series into pieces with constant behavior and generate an autoregression model at each piece.

**Association Analysis and Extraction of Sequence Patterns**

Agrawal and Srikant [3] proposed one of the first methods for association analysis in timestamped transactional databases. A transactional database records timestamped customer transactions (e.g., sets of books bought at a time) in a store (e.g., bookstore) and the objective of the analysis is to discover causality relationships between sets of items bought by customers. An example of such a *sequential* pattern (taken from the paper) is "5% of customers bought 'Foundation', then 'Foundation and Empire', and then 'Second Foundation' ", which can be represented by {(Foundation),(Foundation and Empire),(Second Foundation)}. In general, sequential patterns are total orders of *sets of items* bought in the same transaction. For example, {(Foundation,Life),(Second Foundation)} models the buying of 'Foundation' and 'Life' at a single transaction followed by 'Second Foundation' at another transaction. The patterns can be extracted by dividing the database that records the transaction history of the bookstore into groups, one per customer, and then treat each group as an ordered sequence. For example, the transactional database shown in Figure 1a is transformed to the grouped table of Figure 1b.

The algorithm for extracting sequential patterns from the transformed database is reminiscent to the Apriori algorithm for frequent itemsets in transactional databases [2]. It takes as input a minimum support threshold

| transaction-ID | time | Customer-ID | itemset |
|---|---|---|---|
| 101 | 10 | C1 | $A$, $B$, $C$ |
| 102 | 11 | C2 | $A$, $C$ |
| 103 | 12 | C3 | $B$ |
| 104 | 15 | C2 | $B$, $E$ |
| 105 | 18 | C2 | $F$ |
| 106 | 20 | C1 | $E$ |
| 107 | 25 | C3 | $F$ |

(a) orignal database

| Customer-ID | time | itemset |
|---|---|---|
| C1 | 10 | $A$, $B$, $C$ |
| C1 | 20 | $E$ |
| C2 | 11 | $A$, $C$ |
| C2 | 15 | $B$, $E$ |
| C2 | 18 | $F$ |
| C3 | 12 | $B$ |
| C3 | 25 | $F$ |

(b) input of mining algorithm

Figure 1: Transformation of a timestamped transactional database

*min_sup* and operates in multiple passes. In the first pass, the items that have been bought by at least *min_sup*% of the customers are put to a frequent items set $L_1$. Then, orderings of pairs of items in $L_1$ form a candidate set $C_2$ of level-2 sequential patterns, the supports of which are counted during the second pass of the transformed database and the frequent ones form $L_2$. A sequence adds to the support of a pattern if the pattern is contained in it. For example, the sequence $\{(A, C), (B, E), (F)\}$ of customer C2 in Figure 1 adds to the support of pattern $\{(A), (F)\}$. In general, after $L_k$ has been formed, the algorithm generates and counts candidate patterns of $k + 1$ items. These candidates are generated by joining pairs $(s_1, s_2)$ of frequent $k$-sequences, such that the subsequence obtained by dropping the first item of $s_1$ is identical to the one obtained by dropping the last item of $s_2$. For example, $\{(A, B), (C)\}$ and $\{(B), (C, D)\}$ generate $\{(A, B), (C, D)\}$. Candidates resulting from the join phase are pruned if they have a subsequence that is not frequent.

Agrawal and Srikant also considered adding constraints when counting the supports of sequential patterns. For example, if 'Foundation and Empire' is bought 3 years after 'Foundation', these two books may be considered unrelated. In addition, they considered relaxing the rigid definition of a transaction by unifying transactions of the same customer that took place close in time. For example, if a customer buys a new book minutes after her previous transaction, this book should be included in the previous transaction (i.e., the customer may have forgotten to include it in her basket before). Parallel to Agrawal and Srikant, Mannila et al. [9] studied the extraction of frequent causality patterns (called episodes) in long event sequences. The main differences of this work are (i) the input is a single very long sequence of events (e.g., a stream of sensor indications), (ii) patterns are instantiated by temporal sliding windows along this stream of events, and (iii) patterns can contain sequential modules (e.g., $A$ after $B$) or parallel modules (e.g., $A$ and $B$ in any order). An example of such an episode is "$C$ first, then $A$ and $B$ in any order, then $D$". To compute frequent episodes Mannila et al. [9] proposed adaptations of the classic Apriori technique [2]. A more efficient technique for mining sequential patterns was later proposed by Zaki [14].

Han et al. [6] studied the problem of mining partial periodic patterns in long event sequences. In many applications, the associations between events follow a periodic behavior. For instance, the actions of people follow habitual patterns on a daily basis (i.e., 'wake-up', then 'have breakfast', then 'go to work', etc.). Given a long event sequence (e.g., the actions of a person over a year) and a time period (e.g., 24 hours), the objective is to identify patterns of events that have high support over all the periodic time intervals (e.g., days). For this purpose, all subsequences corresponding to the activities of each periodic interval can be extracted from the long sequence, and a sequential pattern mining algorithm [3] can be applied. Based on this idea, an efficient technique for periodic pattern mining, which is facilitated by the use of a sophisticated prefix tree data structure, was proposed by Han et al. [6]. In some applications, the time period every when the patterns are repeated is unknown and has to be discovered from the data. Towards this direction, Cao et al. [5] present a data structure that automatically identifies the periodicity and discovers the patterns at only a small number of passes over the data sequence.

**Temporal, Cyclic, and Calendric Association Rules**

An association rule in a transactional database may not be strong (according to specific support and confidence thresholds) in the whole database, but only when considering the transactions in a specified time interval (e.g., during the winter of 2005). An association rule bound to a time interval, where it is strong, is termed temporal association rule [12]. Identification of such a rule can be performed by starting from short time intervals and progressively extending them to the maximum possible length where the rule remains strong.

Özden et al. [10] noticed that association rules in transactional databases (e.g., people who buy turkey they also buy pumpkins) may hold only in particular temporal intervals (e.g., during the last week of November every year). These are termed *cyclic* association rules, because they are valid periodically, at a specific subinterval of a cycle (e.g., year). Such rules can be discovered by identifying the periodic intervals of fixed granularity (e.g., week of the year), which support the associations.
Cyclic rules are assumed to be supported at *exact* intervals (e.g., the last day of January), and at *every* cycle (e.g., every year). In practice, a rule may be supported with some mismatch threshold (e.g., the last weekday of January) and only at the majority of cycles (e.g., 80% of the cycles). Accordingly, the "cyclic" rule concept was extended by Ramaswamy et al. [11] to the more flexible *calendric* association rule. A calendar is defined by a set of time intervals (e.g., the last 3 days of January, every year). For a calendric rule to be strong, it should have enough support and confidence in at least $min\_sup\%$ of the time units included in the calendar. An algerbra for defining calendars and a method for discovering calendric association rules referring to them can be found in Ref. [11].
Li et al. [8] proposed a more generalized framework for calendric association rules. Instead of searching based on a predetermined calendar, they automatically identify the rules and their supporting calendars, taken from a hierarchy of calendar concepts. The hierarchy is expressed by a relation of temporal generalizations of varying granularity, e.g., $R(year, month, day)$. A possible calendric pattern is expressed by setting to each attribute, either a specific value of its domain, or a wildcard value '$*$'. For example, pattern $(*, Jan, 30)$ means the 30th of January each year, while $(2005, *, 30)$ means the the 30th day of each month in year 2005. By defining containment relationships between such patterns (e.g., $(*, *, 30)$ contains all the intervals of $(2005, *, 30)$) and observing that itemset supports for them can be computed constructively (e.g., the support of an itemset in $(*, *, 30)$ can be computed using its support in all $(y, *, 30)$ for any year $y$), Li et al. [8] systematically explore the space of all calendric patterns using the Apriori principle to prune space (e.g., an itemset is not frequent in $(*, *, 30)$ if it is infrequent in all $(y, *, 30)$ for every year $y$).

## KEY APPLICATIONS*

**Weather Forecasting** Temporal causality relationships between events can assist the prediction of weather phenomena. In fact, such patterns have been used for this purpose since the ancient years (e.g., "if swallows fly low, it is going to rain soon").
**Market Basket Analysis** Extension of classic association analysis to consider temporal information finds application in market analysis. Examples include, temporal relationships between products that are purchased within the same period by customers ("5% of customers bought 'Foundation', then 'Foundation and Empire' ") and calendric association rules (e.g., turkey is bought together with pumpkin during the last week of November, every year).
**Stock Market Prediction** Time-series classification and regression is often used by financial analysts to predict the future behavior of stocks. The structure of the time series can be compared with external factors (such as pieces of news) to derive more complex associations that result in better accuracy in prediction.
**Web Data Mining** The World Wide Web can be viewed as a huge graph where nodes correspond to web pages (or web sites) and edges correspond to links between them. Users navigate through the web defining sequences of page visits, which are tracked in weblogs. By analyzing these sequences one can identify frequent sequential patterns between web pages or even classify users based on their behavior (sequences of sites they visit and sequences of data they download).

## CROSS REFERENCES

1. temporal periodicity
2. time series query
3. spatio-temporal data mining
4. association rule mining

## RECOMMENDED READING

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[4] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[5] Huiping Cao, David W. Cheung, and Nikos Mamoulis. Discovering partial periodic patterns in discrete data sequences. In *PAKDD*, pages 653–658, 2004.

[6] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE*, pages 106–115, 1999.

[7] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[8] Yingjiu Li, Peng Ning, Xiaoyang Sean Wang, and Sushil Jajodia. Discovering calendar-based temporal association rules. *Data Knowl. Eng.*, 44(2):193–218, 2003.

[9] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[10] Banu Özden, Sridhar Ramaswamy, and Abraham Silberschatz. Cyclic association rules. In *ICDE*, pages 412–421, 1998.

[11] Sridhar Ramaswamy, Sameer Mahajan, and Abraham Silberschatz. On the discovery of interesting patterns in association rules. In *VLDB*, pages 368–379, 1998.

[12] John F. Roddick and Myra Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE Trans. Knowl. Data Eng.*, 14(4):750–767, 2002.

[13] Li Wei and Eamonn J. Keogh. Semi-supervised time series classification. In *KDD*, pages 748–753, 2006.

[14] Mohammed Javeed Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.

# Temporal Data Models

Christian S. Jensen and Richard T. Snodgrass
University of Aalborg, Denmark and University of Arizona, USA

## SYNONYMS

valid-time data model; transaction-time data model; bitemporal data model; historical data model

## DEFINITION

A "data model" consists of two components, namely a set of objects and a language for querying those objects [4]. In a temporal data model the objects vary over time and the operations in some sense "know" about time. Focus has been on the design of data models where the time references capture valid time or transaction time or a combination of both (for bitemporal data).

## HISTORICAL BACKGROUND

Almost all real-world databases contain time-referenced data. Few interesting databases are entirely stagnant, and when the modeled reality changes, the database must be updated. Usually at least the start time of currently valid data is captured, though most databases also retain previous data.

Two decades of research into temporal databases have unequivocally shown that a *time-referencing table*, containing certain kinds of time-valued columns that capture one or more temporal aspects of data recorded in other columns, is completely different from this table, without the time-valued columns. Effectively designing, querying, and modifying time-referencing tables requires a different set of approaches and techniques. It is possible to handle such data within standard data models, generally at the expense of high data redundancy, awkward modeling, and complicated and unfriendly query languages. An alternative is a data model, probably an extension of an extant, non-temporal data model, that explicitly incorporates time, making it easier to express queries and modification and integrity constraints.

As an example, consider a primary key of the following relation, which records the current position of each employee, identified by their social security number: `EMP(EmpID, POSITION)`. The primary key is obviously `EmpID`. Now add `STARTTIME` and `STOPTIME` attributes. While a primary key of `(EmpID,STARTTIME)` seems to work, such a primary key will not prevent overlapping periods, which would allow an employee to have two positions at a point of time, which is problematic. Stating the primary key constraint properly requires a complex assertion containing a dozen lines of code with multiple sub-queries [3]. Referential integrity is even more challenging.

The last two decades have seen the introduction of a great many temporal data models, not just in the context of relational data, as in the above example, but also with object-oriented, logic-based, and semi-structured data.

## SCIENTIFIC FUNDAMENTALS

### Levels of Abstraction

Temporal data models exist at three abstraction levels: the *conceptual level*, in which the data models are generally extensions of the Entity-Relationship Model, the *logical level*, in which the data models are generally extensions of the relational data model or of an object-oriented data model, and, infrequently, the *physical level*, in which the data model details how the data is to be stored. In terms of prevalence, models at the logical level are far more numerous. However, it has been shown that several of the models that were originally proposed as logical data

models are actually equivalent to the BCDM logical model, and so should more properly be viewed as physical data models [2]. This summary is restricted to logical models, focusing on the objects that are subject to querying rather than the query languages. First examined is the association of time with data, as this is at the core of temporal data management.

## Temporal Aspects of Data

A database models and records information about a part of reality, termed either the modeled reality. Aspects of the modeled reality are represented in the database by a variety of structures termed database entities. In general, times are associated with database entities. The term "fact" is used for any (logical) statement that can meaningfully be assigned a truth value, i.e., true or false.

The facts recorded by database entities are of fundamental interest, and a fundamental temporal aspect may be associated with these: the valid time of a fact is the times when the fact is true in the modeled reality. While all facts have a valid time by definition, the valid time of a fact may not necessarily be recorded in the database. For example, the valid time may not be known, or recording it may not be relevant. Valid time may be used for the capture of more application-specific temporal aspects. Briefly, an application-specific aspect of a fact may be captured as the valid time of another, related fact.

Next, the transaction time of a database entity is the time when the entity is current in the database. Like valid time, this is an important temporal aspect. Transaction time is the basis for supporting accountability and "traceability" requirements. Note that transaction time, unlike valid time, may be associated with any database entity, not only with facts. As for valid time, the transaction-time aspect of a database entity may or may not be captured in the database. The transaction-time aspect of a database entity has a duration: from insertion to deletion. As a consequence of the semantics of transaction time, deleting an entity does not physically remove the entity from the database; rather, the entity remains in the database, but ceases to be part of the database's current state.

Observe that the transaction time of a database fact, say $f$ is the valid time of the related fact, "$f$ is current in the database." This would indicate that supporting transaction time as a separate aspect is redundant. However, both valid and transaction time are aspects of the content of all databases, and recording both of these is essential in a wide range of applications. In addition, transaction time, due to its special semantics, is particularly well-behaved and may be supplied automatically by the DBMS. Specifically, the transaction times of facts stored in the database are bounded by the time the database was created at one end of the time line and by the current time at the other end.

The above discussion suggests why temporal data models generally offer built-in support for one or both of valid and transaction time.

## Representation of Time

The valid and transaction time values of database entities are drawn from some appropriate time domain. There is no single answer to how to perceive time in reality and how to represent time in a database, and different time domains may be distinguished with respect to several orthogonal characteristics. First, the time domain may or may not stretch infinitely into the past and future. Second, time may be perceived as discrete, dense, or continuous. Some feel that time is really continuous; others contend that time is discrete and that continuity is just a convenient abstraction that makes it easier to reason mathematically about certain discrete phenomena. In databases, a finite and discrete time domain is typically assumed, e.g., in the SQL standards. Third, a variety of different structures have been imposed on time. Most often, time is assumed to be totally ordered.

Much research has been conducted on the semantics and representation of time, from quite theoretical topics, such as temporal logic and infinite periodic time sequences, to more applied questions such as how to represent time values in minimal space. Substantial research has been conducted that concerns the use of different time granularities and calendars in general, as well as the issues surrounding the support for indeterminate time values. Also, there is a significant body of research on time data types, e.g., time instants, time intervals (or "periods"), and temporal elements.

## Data Model Objects

The management of temporal aspects has been achieved by building time into the data model objects. Here, the relational model is assumed, with a focus on valid time. One approach is to timestamp tuples with time instants, or points. Then a fact is represented by one tuple for each time point during which the fact is valid. An example instance for the `EMP` relation example is shown in Figure 1.

| EmpID | POSITION | T |
|---|---|---|
| 1 | Sales | 3 |
| 1 | Sales | 4 |
| 1 | Engineering | 5 |
| 1 | Engineering | 6 |
| 2 | Sales | 4 |
| 2 | Sales | 5 |
| 2 | Sales | 6 |
| 2 | Sales | 7 |

Figure 1: Point Model

A distinguishing feature of this approach is that (syntactically) different relations have different information content. Next, timestamps are atomic values that can be easily compared. Assuming a totally ordered time domain, the standard set of comparison predicates, $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$, is sufficient to conveniently compare timestamps. The conceptual simplicity of time points comes at a cost, though. The model offers little support for capturing, e.g., that employee 2 was assigned to Sales during two contiguous periods $[4, 5]$ and $[6, 7]$, instead of during a single contiguous period $[4, 7]$.

It is important to note that the point model is not meant for physical representation, as for all but the most trivial time domains, the space needed when using the point model is prohibitive. The combination of conceptual simplicity and computational complexity has made the point model popular for theoretical studies.

Another type of data model uses time periods as timestamps. This type of model associates each fact with a period that captures the valid time of the fact. Multiple tuples are needed if a fact is valid over disjoint periods. Figure 2 illustrates the approach.

| EmpID | POSITION | T |
|---|---|---|
| 1 | Sales | [3,4] |
| 1 | Engineering | [5,6] |
| 2 | Sales | [4,5] |
| 2 | Sales | [6,7] |

Figure 2: Period (or Interval) Model

The notion of snapshot equivalence, which reflects a point-based view of data, establishes a correspondence between the point-based and period-based models. Imagine that the last two tuples in the relation in Figure 2 were replaced with the single tuple (2, Sales, [4,7]) to obtain a new relation. The resulting two relations are different, but snapshot equivalent. Specifically, the new relation is a coalesced version of the original relation.

In some data models, the relations are taken to contain the exact same information. These models adopt a point-based view and are only period-based in the weak sense that they use time periods as convenient representations of (convex) sets of time points. It then also makes sense for such models to require that their relation instances be coalesced. This requirement ensures that relation instances that are syntactically different are also semantically different, and vice versa. In such models, the relation in Figure 2 is not allowed.

In an inherently period-based model, periods carry meaning beyond denoting a set of points. In some situations, it may make a difference whether an employee holds a position for two short, but consecutive time periods versus for one long time period. Period-based models do not enforce coalescing and capture this distinction naturally.

Next, a frequently mentioned shortcoming of periods is that they are not closed under all set operations, e.g., subtraction. This has led to the proposal that temporal elements be used as timestamps instead. These are finite unions of periods.

With temporal elements, the same two semantics as for periods are possible, although models that use temporal elements seem to prefer the point-based semantics. Figures 3(a) and 3(b) uses temporal elements to capture the example assuming the period-based semantics and point-based semantics, respectively. (As $[4,5] \cup [6,7] = [4,7]$, this later period could have been used instead of $[4,5] \cup [6,7]$.)

| EmpIS | POSITION | T |
|---|---|---|
| 1 | Sales | $[3,4]$ |
| 1 | Engineering | $[5,6]$ |
| 2 | Sales | $[4,5]$ |
| 2 | Sales | $[6,7]$ |

(a) Period View

| EmpID | POSITION | T |
|---|---|---|
| 1 | Sales | $[3,4]$ |
| 1 | Engineering | $[5,6]$ |
| 2 | Sales | $[4,5] \cup [6,7]$ |

(b) Point View

Figure 3: Temporal Element Model

Note that the instance in Figure 3(b) exemplifies the instances used by the point-based bitemporal conceptual data model (BCDM) when restricted to valid time. This model has been used for TSQL2. The BCDM timestamps facts with values that are sets of time points. This is equivalent to temporal elements because the BCDM adopts a discrete and bounded time domain.
Because value-equivalent tuples are not allowed (this corresponds to the enforcement of coalesced relations as discussed in the previous section), the full history of a fact is contained in exactly one tuple, and one tuple contains the full history of exactly one fact. In addition, relation instances that are syntactically different have different information content, and vice versa. This design decision reflects the point-based underpinnings of the BCDM.
With temporal elements, the full history of a fact is contained in a single tuple, but the information in a relation that pertains to some real-world object may still be spread across several tuples. To capture all information about a real-world object in a single tuple, attribute value timestamping has been introduced. This is illustrated in Figure 4 that displays the sample instance using a typical attribute-value timestamped data model.

| EmpID | | POSITION | |
|---|---|---|---|
| $[3,6]$ | 1 | $[3,4]$ | Sales |
| | | $[5,6]$ | Engineering |
| $[4,7]$ | 2 | $[4,7]$ | Sales |

Figure 4: Attribute-Value Timestamped Model

The instance records information about employees and thus holds one tuple for each employee, with a tuple containing all information about an employee. An obvious consequence is that the information about a position cannot be contained in a single tuple. Another observation is that a single tuple may record multiple facts. In the example, the first tuple records two facts: the position type for employee 1 for the two positions, Sales and Engineering.
It should also be noted that different groupings into tuples are possible for this attribute-value timestamping model. Figure 5 illustrates groups the relation instance in Figure 4 on the POSITION attribute, indicating that it is now the positions, not the employees, that are the objects in focus.

| EmpID | | POSITION | |
|---|---|---|---|
| $[3,4]$ | 1 | $[3,7]$ | Sales |
| $[4,7]$ | 2 | | |
| $[5,6]$ | 1 | $[5,6]$ | Engineering |

Figure 5: Attribute-Value Timestamped Model, Grouped on POSITION

Data models that timestamp attribute values may be temporally grouped. In a temporally grouped model, all aspects of a real-world object may be captured by a single tuple [1].

At first sight, that attribute-value timestamped model given above is temporally grouped. However, with a temporally grouped model, a real-world object is allowed to change value for its key attribute. In the example, this means that the instance in Figure 6 should be possible. Now observe that when grouping this instance on `EmpID` or `POSITION`, or both, it is not possible to get back to the original instance. Thus, temporally grouped tuples are not well supported. Clifford et al. [1] explore the notion of temporally grouped in considerable depth.

| EmpID | | POSITION | |
|---|---|---|---|
| $[3,5]$ | 1 | $[3,4]$ | Sales |
| $[6,6]$ | 3 | $[5,6]$ | Engineering |
| $[4,7]$ | 2 | $[4,7]$ | Sales |

Figure 6: Attribute-Value Timestamped Model, Temporally Grouped

## Query Languages

Having covered the objects that are subject to querying, the last major aspect of a logical data model is the query language associated with the objects. Such languages come in several variants.

Some are intended for internal use inside a temporally enhanced database management system. These are typically algebraic query languages. However, algebraic languages have also been invented for more theoretical purposes. For example, an algebra may be used for defining the semantics of a temporal SQL extension. A key point is that an algebra is much simpler than is such an extension. Little is needed in terms of language design; only a formal definition of each operator is needed.

Other query languages are targeted at application programmers and are thus typically intended to replace SQL. The vast majority of these are SQL extensions. Finally, languages have been proposed for the purpose of conducting theoretical studies, e.g., of expressive power.

### KEY APPLICATIONS*

Virtually all databases contain temporal information, and so virtually all database applications would benefit from the availability of data models that provide natural support for such time-varying information.

### FUTURE DIRECTIONS

Rather than come up with a new temporal data model, it now seems better to extend, in an upward-consistent manner, existing non-temporal models to accommodate time-varying data.

### CROSS REFERENCE*

Period-Stamped Temporal Models, Point-Stamped Temporal Models, Supporting Transaction Time, Temporal Access Control, Temporal Concepts in Philosophy, Temporal Compatibility, Temporal Conceptual Models, Temporal Constraints, Temporal Database, Temporal Indeterminacy, Temporal Logical Models, Temporal Object-Oriented Databases, Temporal Probabilistic Models, Temporal Query Languages, Temporal XML, Transaction Time, Valid Time

### RECOMMENDED READING


[1] James Clifford, Albert Croker, and Alexander Tuzhilin, "On Completeness of Historical Relational Query Languages," *ACM TODS* 19(1):64–16, March 1994.

[2] Christian S. Jensen, Michael D. Soo and Richard T. Snodgrass, "Unifying Temporal Data Models via a Conceptual Model," *Information Systems*, Vol. 19, No. 7, December 1994, pp. 513–547.

[3] Richard T. Snodgrass,, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, July 1999.
[4] Dennis C. Tsichritzis and Frederic H. Lochovsky. *Data Models*. Software Series. Prentice-Hall, 1982.

# Temporal Dependencies

Jef Wijsen
University of Mons-Hainaut

**SYNONYMS**
None

**DEFINITION**
Static integrity constraints involve only the current database state. Temporal integrity constraints involve current, past, and future database states; they can be expressed by essentially unrestricted sentences in temporal logic. Certain syntactically restricted classes of temporal constraints have been studied in their own right for considerations of feasibility or practicality; they are usually called temporal dependencies. Most temporal dependencies proposed in the literature are dynamic versions of static functional dependencies.

**HISTORICAL BACKGROUND**
Static dependencies (functional, multivalued, join, and other dependencies) have been investigated in depth since the early years of the relational model. Classical problems about dependencies concern logical implication and axiomatization. The study of a particular dependency class is often motivated by its practical importance in databases. This is undeniably the case for the notion of functional dependency (FD), which is fundamental in database design. A dynamic version of functional dependencies was first proposed by Vianu [7]. Since the mid 90s, several other temporal variants of the notion of FD have been introduced.

**SCIENTIFIC FUNDAMENTALS**
This section gives an overview of several temporal dependency classes proposed in the literature. With the exception of the notion of dynamic algebraic dependency (DAD) [2], all temporal dependencies here presented can be seen as special cases of the notion of constraint-generating dependency (CGD) [1]. The formalism of CGD, presented near the end, thus allows to compare and contrast different temporal dependency classes.

## Functional Dependencies Over Temporal Databases

Since the semantics of temporal versions of FDs will be explained in terms of static FDs, the standard notion of FD is recalled next. All the following definitions are relative to a fixed set $U = \{A_1, \ldots, A_n\}$ of *attributes*.

**Definition** A *tuple* over $U$ is a set $t = \{A_1 : c_1, \ldots, A_n : c_n\}$, where each $c_i$ is a constant. If $X \subseteq U$, then $t[X]$ denotes the restriction of $t$ to $X$. A *relation* over $U$ is a finite set of tuples over $U$.
A *functional dependency* (FD) over $U$ is an expression $X \rightarrow Y$ where $X, Y \subseteq U$. A relation $I$ over $U$ *satisfies* the FD $X \rightarrow Y$ if for all tuples $s, t \in I$, if $s[X] = t[X]$, then $s[Y] = t[Y]$. □

When evaluating FDs over temporal relations, one may want to treat timestamp attributes differently from other attributes. To illustrate this, consider the temporal relation *EmpInterval* with its obvious meaning.

*EmpInterval*

| *Name* | *Sex* | *Sal* | *Project* | *From* | *To* |
|---|---|---|---|---|---|
| Ed | M | 10K | Pulse | 1 | 3 |
| Ed | M | 10K | Wizard | 2 | 3 |
| Ed | M | 12K | Wizard | 4 | 4 |

An employee has a unique sex and a unique salary, but can work for several projects. The salary, unlike the sex, may change over time. The relation *EmpInterval* is "legal" with respect to these company rules, because for any time point $i$, the *snapshot* relation $\{t[Name, Sex, Sal, Project] \mid t \in EmpInterval, t(From) \leq i \leq t(To)\}$ satisfies $Name \rightarrow Sex$ and $Name \rightarrow Sal$. Note that the relation *EmpInterval* violates the FD $Name \rightarrow Sal$, because the last two tuples agree on *Name* but disagree on *Sal*. However, since these tuples have disjoint periods of validity, they do not go against the company rules.

Hence, the intended meaning of an FD expressed over a temporal relation may be that the FD must be satisfied at every snapshot. To indicate that $Name \rightarrow Sal$ has to be evaluated on snapshots, Jensen et al. [6] use the notation $Name \xrightarrow{\mathsf{T}} Sal$. The FD $Name \rightarrow Sex$ need no change, because the sex of an employee should be unique not only at each snapshot, but also over time.

Chomicki and Toman [3] note that no special syntax is needed if tuples are timestamped by time points. For example, given the following point-stamped temporal relation, one can simply impose the classical FDs $Name \rightarrow Sex$ and $Name, T \rightarrow Sal$.

*EmpPoint*

| *Name* | *Sex* | *Sal* | *Project* | *T* |
|---|---|---|---|---|
| Ed | M | 10K | Pulse | 1 |
| Ed | M | 10K | Pulse | 2 |
| Ed | M | 10K | Pulse | 3 |
| Ed | M | 10K | Wizard | 2 |
| Ed | M | 10K | Wizard | 3 |
| Ed | M | 12K | Wizard | 4 |

## Vianu's Dynamic Functional Dependency [7]

Consider an employee table with attributes *Name*, *Sex*, *Merit*, and *Sal*, with their obvious meanings. The primary key is *Name*. Assume an annual companywide update of merits and salaries. In the relation shown next, every tuple is followed by its updated version. Old values appear in columns with a caron ($\check{}$), new values in columns with a caret ($\hat{}$). For example, John Smith's salary increased from 10K to 12K. Such a relation that juxtaposes old and new values is called *action relation*.

| ⊢ old | | | ⊣ | ⊢ new | | | ⊣ |
|---|---|---|---|---|---|---|---|
| $\check{Name}$ | $\check{Sex}$ | $\check{Merit}$ | $\check{Sal}$ | $\hat{Name}$ | $\hat{Sex}$ | $\hat{Merit}$ | $\hat{Sal}$ |
| John Smith | M | Poor | 10K | John Smith | M | Good | 12K |
| An Todd | F | Fair | 10K | An Todd | F | Good | 12K |
| Ed Duval | M | Fair | 10K | Ed Duval | M | Fair | 10K |

The company's policy that *"Each new salary is determined solely by new merit and old salary"* can be expressed by the classical FD $\check{Sal}, \hat{Merit} \rightarrow \hat{Sal}$ on the above action relation. In particular, since John Smith and An Todd agree on old salary and new merit, they must have the same new salary. Such FDs on action relations were introduced by Vianu [7] and called dynamic functional dependencies.

**Definition** For each $A_i \in U$, assume that $\check{A}_i$ and $\hat{A}_i$ are new distinct attributes. Define $\check{U} = \{\check{A}_1, \ldots, \check{A}_n\}$ and $\hat{U} = \{\hat{A}_1, \ldots, \hat{A}_n\}$. For $t = \{A_1 : c_1, \ldots, A_n : c_n\}$, define:

$$\begin{aligned} \check{t} &= \{\check{A}_1 : c_1, \ldots, \check{A}_n : c_n\} \text{ , a tuple over } \check{U}; \text{ and} \\ \hat{t} &= \{\hat{A}_1 : c_1, \ldots, \hat{A}_n : c_n\} \text{ , a tuple over } \hat{U}. \end{aligned}$$

A *Vianu dynamic functional dependency* (VDFD) over $U$ is an FD $X \rightarrow Y$ over $\check{U}\hat{U}$ such that for each $A \in Y$, $XA$ contains at least one attribute from $\check{U}$ and one attribute from $\hat{U}$.

An *update* over $U$ is a triple $\langle I, \mu, J \rangle$, where $I$ and $J$ are relations over $U$ and $\mu$ is a bijective mapping from $I$ to $J$. The update $\langle I, \mu, J \rangle$ *satisfies* the VDFD $X \rightarrow Y$ if the *action relation* $\{\check{t} \cup \hat{s} \mid t \in I, s = \mu(t)\}$ satisfies the FD $X \rightarrow Y$. □

The notion of VDFD directly extends to sequences of database updates. The interaction between dynamic VDFDs and static FDs is studied in [7].

## Temporal Extensions of Functional Dependency Proposed by Wijsen [9, 10, 11, 12]

Instead of extending each tuple with its updated version, as is the case for VDFDs, one can take the union of the old relation and the new relation:

| *Name* | *Sex* | *Merit* | *Sal* | | |
|---|---|---|---|---|---|
| John Smith | M | Poor | 10K | $\top$ | |
| An Todd | F | Fair | 10K | old | |
| Ed Duval | M | Fair | 10K | $\bot$ | $\top$ |
| John Smith | M | Good | 12K | | new |
| An Todd | F | Good | 12K | | $\bot$ |

The company's policy that *"Every change in merit (promotion or demotion) gives rise to a salary change,"* is expressed by $Name, Sal \overset{\circ}{\rightarrow} Merit$ and means that the FD $Name, Sal \rightarrow Merit$ must be satisfied by the union of the old and the new relation. In particular, since Ed Duval's salary did not change, his merit cannot have changed either. Likewise, *"The sex of an employee cannot change"* is expressed by $Name \overset{\circ}{\rightarrow} Sex$. The construct $\overset{\circ}{\rightarrow}$ naturally generalizes to database histories that involve more than two database states.

**Definition** A *Wijsen dynamic functional dependency* (WDFD) over $U$ is an expression of the form $X \overset{\circ}{\rightarrow} Y$, where $X, Y \subseteq U$.
A *database history* is a sequence $\langle I_1, I_2, I_3, \ldots \rangle$, where each $I_i$ is a relation over $U$. This history *satisfies* $X \overset{\circ}{\rightarrow} Y$ if for every $i \in \{1, 2, \ldots\}$, $I_i \cup I_{i+1}$ satisfies $X \rightarrow Y$. □

Although $I_{i+1}$ can be thought of as the result of an update performed on $I_i$, there is no need to model a one-one relationship between the tuples of both relations, as was the case for VDFDs. VDFDs and WDFDs capture different types of constraints, even in the presence of some attribute that serves as a time-invariant tuple-identifier. This difference will be illustrated later on in the discussion of constraint-generating dependencies.
In practice, database histories will be be stored in relations with timestamped tuples. Like FDs, WDFDs can result in predictable (i.e. redundant) values. For example, if the following relation has to satisfy $Name, Sal \overset{\circ}{\rightarrow} Merit$, then the value for the placeholder † must be equal to "Poor." Wijsen [9] develops temporal variants of 3NF to avoid data redundancy caused by WDFDs.

| *Name* | *City* | *Merit* | *Sal* | *From* | *To* |
|---|---|---|---|---|---|
| Ed | Paris | Poor | 10K | 1 | 2 |
| Ed | London | † | 10K | 3 | 4 |

WDFDs can be naturally generalized as follows: instead of interpreting FDs over unions of successive database states, the syntax of FD is extended with a binary relation on the time domain, called *time accessibility relation*, that indicates which tuple pairs must satisfy the FD.

**Definition** A *time accessibility relation* (TAR) is a subset of $\{(i, j) \mid 1 \leq i \leq j\}$. A *generalized* WDFD over $U$ is an expression $X \rightarrow_{\alpha} Y$, where $X, Y \subseteq U$ and $\alpha$ is a TAR. This generalized WDFD is satisfied by database history $\langle I_1, I_2, I_3, \ldots \rangle$ if for all $(i, j) \in \alpha$, $s \in I_i$, $t \in I_j$, if $s[X] = t[X]$, then $s[Y] = t[Y]$. □

If the TAR `Next` is defined by $\texttt{Next} = \{(1,1), (1,2), (2,2), (2,3), (3,3), (3,4), \ldots\}$, then $X \rightarrow_{\texttt{Next}} Y$ and $X \overset{\circ}{\rightarrow} Y$ are equivalent. TARs can also capture the notion of time granularity. For example, `MonthTAR` can be defined as the TAR containing $(i, j)$ whenever $i \leq j$ and time points $i, j$ belong to the same month. Then, $Name \rightarrow_{\texttt{MonthTAR}} Sal$ expresses that the salary of an employee cannot change within a month.
The temporal functional dependencies proposed in [11] extend this formalism with a notion of identity, denoted by $\lambda$, similar to object-identity. The identity is time-invariant and allows to relate old and new versions of the same object. For example, $Emp : \lambda \rightarrow_{\texttt{Next}} Name$ means that the name of an employee object cannot change from one time to the next.
Trend dependencies [10, 12] extend generalized WDFDs in still another way by allowing both equalities and inequalities. They can be seen as a temporal extension of the concept of *order dependency* introduced by Ginsburg and Hull [4]. For example, $(Name, =) \rightarrow_{\texttt{Next}} (Sal, \leq)$ expresses that the salary of an employee cannot decrease. Technically, it is satisfied by a database history $\langle I_1, I_2, I_3, \ldots \rangle$ if for all $(i, j) \in \texttt{Next}$, if $s \in I_i$ and $t \in I_j$ and $s(Name) = t(Name)$, then $s(Sal) \leq t(Sal)$.

## Wang et al.'s Temporal Functional Dependency [8]

The dynamic versions of FDs introduced by Vianu and Wijsen can impose constraints on tuples valid at successive time points. Wang et al.'s notion of temporal functional dependency (TFD) concentrates on temporal granularity and compares tuples valid during the same granule of some temporal granularity. The main idea is captured by the following definitions that are simplified versions of the ones found in [8].

**Definition** Assume a linear time domain $(D, <)$. Every nonempty subset of $D$ is called a *granule*. Two distinct granules $G_1$ and $G_2$ are said to be *non-interleaved* if each point of either granule is smaller than all points of the other granule (i.e. either $\forall d_1 \in G_1 \forall d_2 \in G_2(d_1 < d_2)$ or $\forall d_1 \in G_1 \forall d_2 \in G_2(d_2 < d_1)$). A *granularity* is a set $\mathcal{G}$ of pairwise non-interleaved granules. □

Other granularity notions found in the literature (see Temporal Granularity) often assume that granules are indexed by integers. Such index has been omitted here to simplify the formalism.
Common granularities are `Month` and `Year`. If granularity $\mathcal{G}$ is associated with temporal relation $I$, then all tuples of $I$ must be timestamped by granules of $\mathcal{G}$. For example, all tuples in the following relation are timestamped by months. Practical labels, like Nov-2007, are used to denote granules of time points.

*EmpMonth*

| *Name* | *Sal* | *Position* | $T$ : `Month` |
|---|---|---|---|
| Ed | 10K | Lecturer | Nov-2007 |
| Ed | 11K | Lecturer | Dec-2007 |
| Ed | 12K | Professor | Jan-2008 |

**Definition** Assume a set $U = \{A_1, \ldots, A_n\}$ of attributes and a timestamp attribute $T \notin U$. A *timestamped tuple with granularity* $\mathcal{G}$ (or simply $\mathcal{G}$*-tuple*) over $U$ is a set $\{A_1 : c_1, \ldots, A_n : c_n, T : G\}$, where each $c_i$ is a constant and $G \in \mathcal{G}$. If $t = \{A_1 : c_1, \ldots, A_n : c_n, T : G\}$, then define $t[U] = \{A_1 : c_1, \ldots, A_n : c_n\}$ and $t(T) = G$. A *timestamped relation with granularity* $\mathcal{G}$ (or simply $\mathcal{G}$*-relation*) over $U$ is a finite set of $\mathcal{G}$-tuples over $U$. □

The TFD $Name \rightarrow_{\texttt{Month}} Sal$ expresses that the salary of an employee cannot change within a month. Likewise, $Name \rightarrow_{\texttt{Year}} Position$ expresses that the position of an employee cannot change within a year. Both dependencies are satisfied by the relation *EmpMonth* shown above. To check $Name \rightarrow_{\texttt{Year}} Position$, it suffices to verify whether for each year, the FD $Name \rightarrow Position$ is satisfied by the set of tuples whose timestamps fall in that year. For example, for the year 2007, the relation $\{t[Name, Sal, Position] \mid t \in EmpMonth, t(T) \subseteq 2007\}$ must satisfy $Name \rightarrow Position$. This is captured by the following definition.

**Definition** A *temporal functional dependency* (TFD) over $U$ is an expression $X \rightarrow_{\mathcal{H}} Y$, where $X, Y \subseteq U$ and $\mathcal{H}$ is a granularity. A $\mathcal{G}$-relation $I$ over $U$ *satisfies* $X \rightarrow_{\mathcal{H}} Y$ if for each granule $H \in \mathcal{H}$, the relation $\{t[U] \mid t \in I, t(T) \subseteq H\}$ satisfies the FD $X \rightarrow Y$. □

Wang et al. extend classical normalization theory to deal with data redundancy caused by TFDs. For example, since positions cannot change within a year, Ed must necessarily occupy the same position in Nov-2007 and Dec-2007. To avoid this redundancy, the information on positions must be moved into a new relation with time granularity `Year`, as shown next. After this decomposition, Ed's position in 2007 is only stored once.

| *Name* | *Sal* | $T$ : `Month` |
|---|---|---|
| Ed | 10K | Nov-2007 |
| Ed | 11K | Dec-2007 |
| Ed | 12K | Jan-2008 |

| *Name* | *Position* | $T$ : `Year` |
|---|---|---|
| Ed | Lecturer | 2007 |
| Ed | Professor | 2008 |

## Constraint-generating Dependencies [1]

Classical dependency theory assumes that each attribute of a relation takes its values in some uninterpreted domain of constants, which means that data values can only be compared for equality and disequality. The notion of *constraint-generating dependency* (CGD) builds upon the observation that, in practice, certain attributes take their values in specific domains, such as the integers or the reals, on which predicates and functions, such as $\leq$ and $+$, are defined. CGDs can thus constrain data values by formulas in the first-order theory of the underlying domain.

A *constraint-generating k-dependency* takes the following form in tuple relational calculus:

$$\forall t_1 \forall t_2 \ldots \forall t_k ((R_1(t_1) \wedge \cdots \wedge R_k(t_k) \wedge C[t_1, \ldots, t_k]) \implies C'[t_1, \ldots, t_k])$$

where $C$ and $C'$ are arbitrary constraint formulas relating the values of various attributes in the tuples $t_1, \ldots, t_k$. CGDs naturally arise in temporal databases: timestamp attributes take their values from a linearly ordered time domain, possibly equipped with arithmetic. Baudinet et al. [1] specifically mention that the study of CGDs was inspired by the work of Jensen and Snodgrass on temporal specialization and generalization [5]. For example, assume a relation $R$ with two temporal attributes, denoted $VT$ and $TT$. Every tuple $t \in R$ stores information about some event, where $t(TT)$ is the (transaction) time when the event was recorded in the database, and $t(VT)$ is the (valid) time when the event happened in the real world. The following CGD expresses that every event should be recorded within $c$ time units after its occurrence:

$$\forall t (R(t) \implies (t(VT) < t(TT) \wedge t(TT) \leq t(VT) + c)) \ .$$

Given appropriate first-order theories for the underlying domains, CGDs can capture the different temporal dependencies introduced above. Assume a point-stamped temporal relation $Emp(Name, Merit, Sal, VT)$ with its obvious meaning. Assume that *Name* provides a unique and time-invariant identity for each employee. Then, the VDFD $\overset{\vee}{Sal}, \overset{\wedge}{Merit} \rightarrow \overset{\wedge}{Sal}$ can be simulated by the following constraint-generating 4-dependency. In this formula, the tuples $s$ and $s'$ concern the same employee in successive database states (likewise for $t$ and $t'$).

$$\forall s \forall s' \forall t \forall t' \left( \left( \begin{array}{l} Emp(s) \wedge Emp(s') \\ \wedge s(Name) = s'(Name) \\ \wedge s'(VT) = s(VT) + 1 \\ \wedge Emp(t) \wedge Emp(t') \\ \wedge t(Name) = t'(Name) \\ \wedge t'(VT) = t(VT) + 1 \\ \wedge s(Sal) = t(Sal) \\ \wedge s'(Merit) = t'(Merit) \\ \wedge s(VT) = t(VT) \end{array} \right) \implies s'(Sal) = t'(Sal) \right)$$

The WDFD $Name, Sal \overset{\circ}{\rightarrow} Merit$ can be expressed as a constraint-generating 2-dependency:

$$\forall t \forall t' \left( \left( \begin{array}{l} Emp(t) \wedge Emp(t') \\ \wedge t(Name) = t'(Name) \\ \wedge t(Sal) = t'(Sal) \\ \wedge (t'(VT) = t(VT) \vee t'(VT) = t(VT) + 1) \end{array} \right) \implies t(Merit) = t'(Merit) \right)$$

Assume a binary predicate *Month* on the time domain such that $Month(t_1, t_2)$ is true if $t_1$ and $t_2$ are two time instants within the same month. Then, $Name \rightarrow_{\texttt{Month}} Sal$ can be expressed as a constraint-generating 2-dependency:

$$\forall t \forall t' \left( \left( \begin{array}{l} Emp(t) \wedge Emp(t') \\ \wedge t(Name) = t'(Name) \\ \wedge Month(t(VT), t'(VT)) \end{array} \right) \implies t(Sal) = t'(Sal) \right)$$

## Dynamic algebraic dependencies [2]

Consider a database schema containing $Emp(Name, Merit, Sal)$ and $WorksFor(Name, Project)$. A company rule states that *"The Pulse project only recruits employees whose merit has been good at some past time."* The relational algebra expression $E_0$ shown below gets the workers of the Pulse project; the expression $F_0$ gets the good workers. In a consistent database history, if the tuple $t$ is in the answer to $E_0$ on the current database state, then $t$ is in the answer to $F_0$ on some (strict) past database state.

$$\begin{aligned} E_0 &= \pi_{Name}(\sigma_{Project=\text{“Pulse”}} WorksFor) \\ F_0 &= \pi_{Name}(\sigma_{Merit=\text{“Good”}} Emp) \end{aligned}$$

**Definition** A *relational algebra query* $E$ is defined as usual (see the entry Relational Algebra) using the named relational algebra operators $\{\sigma, \pi, \bowtie, \rho, \cup, -\}$. A *dynamic algebraic dependency* (DAD) is an expression of the form $EF$, where $E$ and $F$ are relational algebra queries over the same database schema and with the same output schema. A finite database history $\langle I_0, I_1, \ldots, I_n \rangle$, where $I_0 = \{\}$, *satisfies* the DAD $EF$ if for each $i \in \{1, \ldots, n\}$, for each $t \in E(I_i)$ , there exists $j \in \{1, \ldots, i-1\}$ such that $t \in F(I_j)$. □

The preceding definition uses relational algebra. Nevertheless, every DAD $EF$ can be translated into temporal first-order logic in a straightforward way. Let $Now(\vec{x})$ and $Past(\vec{x})$ be safe relational calculus queries equivalent to $E$ and $F$ respectively, with the same free variables $\vec{x}$. Then the DAD $EF$ is equivalent to the closed formula:

$$\forall \vec{x}(Now(\vec{x}) \implies \blacklozenge Past(\vec{x})) \ .$$

It seems that the expressive power of DADs relative to other dynamic constraints has not been studied in depth. Notice that the simple DAD $\forall x(R(x) \implies \blacklozenge S(x))$ is tuple-generating, in the sense that tuples in $R$ require the existence of past tuples in $S$. The other temporal dependencies presented in this entry are not tuple-generating. Bidoit and De Amo [2] study the existence of an operational specification that can yield all and only the database histories that are consistent. The operational specification assumes that all database updates are performed through a fixed set of update methods, called transactions. These transactions are specified in a transaction language that provides syntax for concatenation and repetition of elementary updates (insert a tuple, delete a tuple, erase all tuples).

## KEY APPLICATIONS

An important motivation for the study of FDs in database courses is schema design. The notion of FD is a prerequisite for understanding the principles of "good" database design (3NF and BCNF). In the same line, the study of temporal functional dependencies has been motivated by potential applications in temporal database design. It seems, however, that one can go a long way in temporal database design by practicing classical, non-temporal normalization theory. As suggested in [8, 3], one can put timestamp attributes on a par with ordinary attributes, write down classical FDs, and apply a standard 3NF decomposition. For example, assume a database schema $\{Name, Sex, Sal, Project, Day, Month, Year\}$.

| *Name* | *Sex* | *Sal* | *Project* | *Day* | *Month* | *Year* |
|---|---|---|---|---|---|---|
| Ed | M | 10K | Pulse | 29-Aug-2007 | Aug-2007 | 2007 |
| Ed | M | 10K | Pulse | 30-Aug-2007 | Aug-2007 | 2007 |
| Ed | M | 10K | Pulse | 31-Aug-2007 | Aug-2007 | 2007 |
| Ed | M | 10K | Wizard | 30-Aug-2007 | Aug-2007 | 2007 |
| Ed | M | 10K | Wizard | 31-Aug-2007 | Aug-2007 | 2007 |
| Ed | M | 12K | Wizard | 1-Sep-2007 | Sep-2007 | 2007 |

The following FDs apply:

$$Name \rightarrow Sex$$
$$Name, Month \rightarrow Sal$$
$$Day \rightarrow Month$$
$$Month \rightarrow Year$$

The latter two FDs capture the relationships that exist between days, months, and years. The standard 3NF synthesis algorithm finds the following decomposition—the last two components may be omitted for obvious reasons:

$$\{Name, Project, Day\}$$
$$\{Name, Sex\}$$
$$\{Name, Sal, Month\}$$
$$\{Day, Month\}$$
$$\{Month, Year\}$$

This decomposition, resulting from standard normalization, is also "good" from a temporal perspective. In general, the "naive" approach seems to prevent data redundancy in all situations where relationships between granularities can be captured by FDs. It is nevertheless true that FDs cannot capture, for example, the relationship between weeks and months. In particular, *Week* $\rightarrow$ *Month* does not hold since certain weeks contain days of two months. In situations where FDs fall short in specifying relationships among time granularities, there may be a need to timestamp by new, artificial time granularities in order to avoid data redundancy [8].

## CROSS REFERENCE

Temporal dependencies are temporal variants of Database Dependencies. See Temporal Integrity Constraints for a general treatment of integrity constraints expressed over temporal databases. In this entry, a simple notion of granularity was used; a more elaborated notion can be found under the entry Temporal Granularity.

## RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.


[1] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-generating dependencies. *J. Comput. Syst. Sci.*, 59(1):94–115, 1999.

[2] N. Bidoit and S. de Amo. A first step towards implementing dynamic algebraic dependences. *Theor. Comput. Sci.*, 190(2):115–149, 1998.

[3] J. Chomicki and D. Toman. Temporal databases. In M. Fisher, D. M. Gabbay, and L. Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier Science, 2005.

[4] S. Ginsburg and R. Hull. Order dependency in the relational model. *Theor. Comput. Sci.*, 26:149–195, 1983.

[5] C. S. Jensen and R. T. Snodgrass. Temporal specialization and generalization. *IEEE Trans. Knowl. Data Eng.*, 6(6):954–974, 1994.

[6] C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Extending existing dependency theory to temporal databases. *IEEE Trans. Knowl. Data Eng.*, 8(4):563–582, 1996.

[7] V. Vianu. Dynamic functional dependencies and database aging. *J. ACM*, 34(1):28–59, 1987.

[8] X. S. Wang, C. Bettini, A. Brodsky, and S. Jajodia. Logical design for temporal databases with multiple granularities. *ACM Trans. Database Syst.*, 22(2):115–170, 1997.

[9] J. Wijsen. Design of temporal relational databases based on dynamic and temporal functional dependencies. In J. Clifford and A. Tuzhilin, editors, *Temporal Databases*, Workshops in Computing, pages 61–76. Springer, 1995.

[10] J. Wijsen. Reasoning about qualitative trends in databases. *Inf. Syst.*, 23(7):463–487, 1998.

[11] J. Wijsen. Temporal FDs on complex objects. *ACM Trans. Database Syst.*, 24(1):127–176, 1999.

[12] J. Wijsen. Trends in databases: Reasoning and mining. *IEEE Trans. Knowl. Data Eng.*, 13(3):426–438, 2001.

# TEMPORAL ELEMENT

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Time period set

## DEFINITION

A *temporal element* is a finite union of $n$-dimensional time intervals. Special cases of temporal elements include *valid-time elements*, *transaction-time elements*, and *bitemporal elements*, which are finite unions of valid-time intervals, transaction-time intervals, and bitemporal intervals, respectively.

## MAIN TEXT

Assuming an $n$-dimensional time domain, an interval is the product of $n$ convex subsets drawn from each of the constituent dimensions.

Given a finite, one-dimensional time domain, a temporal element may be defined equivalently as a subset of the time domain. If the time domain is unbounded and thus infinite, some subsets of the time domain are not temporal elements. These subsets cannot be enumerated in finite space. For non-discrete time domains, the same observation applies.

Temporal elements are often used as timestamps. Unlike time periods, they are closed under the set theoretic operations of union, intersection, and complement, which are very desirable properties when formulating temporal database queries.

The term "temporal element" has been used to denote the concept of a valid-time interval. However, "temporal" is generally used as generic modifier, so more specific modifiers are adopted here for specific kinds of temporal elements. The term "time period set" is an early term for a temporal element. The adopted term has been used much more frequently.

## CROSS REFERENCE*

Bitemporal Interval, Temporal Database, Time Interval, Time Period, Transaction Time, Valid Time

## REFERENCES*

S. K. Gadia, "Temporal Element as a Primitive for Time in Temporal Databases and its Application in Query Optimization," in *Proceedings of the ACM Annual Conference on Computer Science*, Cincinnati, OH, 1986, page 413.

S. K. Gadia, A Homogeneous Relational Model and Query Languages for Temporal Databases, *ACM Transactions on Database Systems* 13(4):418–448, December 1988.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TEMPORAL EXPRESSION

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

none

**DEFINITION**

A *temporal expression* is a syntactic construct used, e.g., in a query that evaluates to a temporal value, i.e., an instant, a time period, a time interval, or a temporal element.

**MAIN TEXT**

Advanced by Gadia [1], a temporal expression is a convenient temporal query language construct.

First, any temporal element is considered a temporal expression. As Gadia uses a discrete and bounded time domain, any subset of the time domain is then a temporal expression. Next, an attribute value of a tuple in Gadia's data model is a function from the time domain to some value domain. Likewise, the attribute values of a tuple are valid during some temporal element. To illustrate, consider an (ungrouped) relation with attributes `Name` and `Position`. An example tuple in this relation is:

$$(\langle\ [4,\ 17]\ \text{Bill}\ \rangle,\ \langle\ [4,\ 8]\ \text{Assistant},\ [9,\ 13]\ \text{Associate},\ [14,\ 17]\ \text{Full}\ \rangle)$$

Now let $X$ be an expression that returns a function from the time domain to some value domain, such as an attribute value, a tuple, or a relation. Then the temporal expression $[\![X]\!]$ returns the domain of $X$. Using the example from above, the temporal expression $[\![\texttt{Position}]\!]$ evaluates to [4, 13] and the temporal expression $[\![\texttt{Position} \neq \text{Associate}]\!]$ evaluates to $[4, 8] \cup [14, 17]$.

The terms "Boolean expression" and "relational expression" may be used for clearly identifying expressions that evaluate to Boolean values and relations.

**CROSS REFERENCE***

Temporal Database, Temporal Element, SQL-Based Temporal Query Languages, Time Instant, Time Interval, Time Period

**REFERENCES***

[1] S. K. Gadia, A Homogeneous Relational Model and Query Languages for Temporal Databases, *ACM Transactions on Database Systems* 13(4):418–448, December 1988.

[2] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TEMPORAL GENERALIZATION

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

None

**DEFINITION**

Temporal generalization comes in three guises. Consider a temporal database in which data items are timestamped with valid and transaction time. Temporal generalization occurs when weakening constraints hitherto applied to the timestamps. Used in this sense, temporal generalization is the opposite of temporal specialization.

Next, a temporal relation is generalized when new timestamps are being associated with its tuples. In larger information systems where data items flow between multiple temporal relations, items may accumulate timestamps by keeping their previous timestamps and gaining new timestamps as they are entered into new temporal relations. Thus, a tuple in a particular relation has multiple timestamps: a valid timestamp, a *primary* transaction timestamp, which records when the tuple was stored in this relation, one or more *inherited* transaction timestamps that record when the tuple was stored in previous relations, and one or more additional timestamps that record when the tuple was manipulated elsewhere in the system.

Finally, a more involved notion of temporal generalization occurs when a derived relation inherit the transaction timestamps from the relations it is derived from.

By describing the temporal generalization that occurs in an information system, important semantics are captures that may be utilized for a variety of purposes. For example, a temporal relation may be queried, with specific restrictions, from a temporal relation that receives tuples with some delay from that relation. Another use is for efficient query processing.

**MAIN TEXT**

The first notion of temporal relation is simply the opposite of temporal specialization.

As an example of the second notion of temporal generalization, consider the following complex yet realistic scenario of a collection of temporal relations maintained by the transportation department of a state government. An *employee relation* is maintained on the workstation of each manager in this department, recording schedules, budgets, and salary levels for the employees under that manager. For the entire department, a single *personnel relation* is maintained on the administrative computer under the data processing group, which also maintains a *financial relation*. The bank, responsible for salary payments, maintains an *accounts relation*. Data items in the form of timestamped tuples move from the employee relation to the personnel relation and then to the financial relation and ultimately to the accounts relation, accumulating transaction timestamps each time they enter a new database. Each timestamp has a relationship with the other transaction timestamps and with the valid timestamp. These can be stated in the schema and utilized during querying to ensure accurate results.

As an example of the third notion of temporal generalization, consider process control in a manufacturing plant. Values from sensors that capture process characteristics such as pressure and temperature may be stored in temporal relations. This data may subsequently be processed further to derive new data that capture relevant aspects of the process being monitored at a higher level of abstraction. The original data based on which the new data was derived may be stored together with the new data, to capture the lineage, or provenance, of that data. As a result, the new data inherits timestamps from the original data.

**CROSS REFERENCE***

Temporal Database, Temporal Specialization, Transaction Time, Valid Time

**REFERENCES***

C. S. Jensen and R. T. Snodgrass, "Temporal Specialization and Generalization," *IEEE Transactions on Knowledge and Data Engineering* 5(6):954–974, December 1994.

# Temporal Granularity

Claudio Bettini
University of Milano, http://dakwe.dico.unimi.it/
X. Sean Wang
University of Vermont, http://www.cs.uvm.edu/~xywang/
Sushil Jajodia
George Mason University, http://csis.gmu.edu/

## SYNONYMS

Time Granularity; Temporal Type

## DEFINITION

In the context of databases, a temporal granularity can be used to specify the temporal qualification of a set of data, similar to its use in the temporal qualification of statements in natural languages. For example, in a relational database, the timestamp associated with an attribute value or a tuple may be interpreted as associating that data with one or more granules of a given temporal granularity (e.g., one or more *days*). As opposed to using instants from a system-specific time domain, the use of user-defined granularities enables both more compact representations and temporal qualifications at different levels of abstraction. Temporal granularities include very common ones like *hours*, *days*, *weeks*, *months*, and *years*, as well as the evolution and specialization of these granularities for specific contexts or applications: *trading days*, *banking days*, *academic semesters*, etc.. Intuitively, a temporal granularity is defined by grouping sets of instants from a time domain into so-called *granules* in a rather flexible way with some mild conditions. For example, the granularity *business days* is defined as the infinite set of granules, each including the time instants composing one working day. A label, for example a date for a day granule, is often used to refer to a particular granule of a granularity. Answering queries in terms of a granularity different from the one used to store data in a database is not simply a matter of syntactic granularity conversion, but it involves subtle semantics issues.

## HISTORICAL BACKGROUND

Temporal granularities have always had a relevant role in the qualification of statements in natural languages, and they still play a major role according to a 2006 study by Oxford University. The study includes words "day", "week", "month", and "year" among the 25 most common nouns in the English language. Temporal granularities have also been used for a long time in computer applications, including personal information management, project management, scheduling, and more. Interestingly, in many situations, their use is limited to a very few common ones, their semantics is often simplified and sometimes confusing, and their management is hard-coded in applications with ad-hoc solutions. The database community seems to be a major driver in formalizing temporal granularities. One of the earliest formalizations was proposed in [5]. At the same time the AI community was investigating formalisms to represent calendar unit systems [10, 9]. In the early nineties, the relevant role played by time granularity and calendars in temporal databases, as well as the need to devise algorithms to manage granular data, became widely recognized by the research community, and some significant progress has been made [14, 4, 11, 13]. Some support for granularities was also included in the design of the temporal query language TSQL2. A comprehensive formal framework for time granularities to be applied in several areas of database research emerged in the mid-nineties, and has been progressively refined in the following years through the investigation of its applications in data mining, temporal database design, query processing, and temporal

constraint reasoning [3]. This framework is based on a set-theoretic approach (partly inspired by [5]) and on an algebraic representation, and it includes techniques to compute basic as well as more complex operations on granules and granularities. The basic notions found a large consensus in the database community [1]. The use of logic to specify formal properties and to reason about granularities as defined in the above framework was investigated in [6]. Programming oriented support for integrating multiple calendars was provided in [12]. In the logic community, an independent line of research on representation and reasoning with multiple granularities investigated classical and non-classical logic extensions based on multi-layered time domains, with applications to the specification of real-time reactive systems. This approach is extensively described in [8]. More recently, the use of automata to represent granularities and to perform basic operations on them has been proposed [7]. This work, partly inspired by previously proposed string-based representation of granularities [15], has the benefit of providing compact representation of granularities. Moreover, decision procedures for some basic problems, such as granularity equivalence and minimization, can be applied directly on that representation.

**SCIENTIFIC FUNDAMENTALS**

What follows is an illustration of the main formal definitions of temporal granularities and their relationships according to the set-theoretic, algebraic approach.

## Definitions

A temporal granularity can be intuitively described as a sequence of time granules, each one consisting of a set of time instants. A granule can be composed of a single instant, a set of contiguous instants (time-interval), or even a set of non-contiguous instants. For example, the `September 2008 business-month`, defined as the collection of all the business days in September 2008, can be used as a granule. When used to describe a phenomena or, in general, when used to timestamp a set of data, a granule is perceived as a non-decomposable temporal entity. A formal definition of temporal granularity is the following.

> Assume a time domain $T$ as a set of totally ordered time instants. A *granularity* is a mapping $G$ from the integers (the *index set*) to the subsets of the time domain such that:
> (1) if $i < j$ and $G(i)$ and $G(j)$ are non-empty, then each element in $G(i)$ is less than all the elements in $G(j)$, and
> (2) if $i < k < j$ and $G(i)$ and $G(j)$ are non-empty, then $G(k)$ is non-empty.

Each non-empty set $G(i)$ in the above definition is called *granule*.

The first condition in the granularity definition states that granules in a granularity do not overlap and that their index order is the same as their time domain order. The second condition states that the subset of the index set for the granules is contiguous. Based on the above definition, while the time domain can be discrete, dense, or continuous, a granularity defines a countable set of granules; each granule is identified by an integer. The index set can thereby provide an "encoding" of the granularity in a computer. Two granules $G(i)$ and $G(j)$ are *contiguous* if there does not exist $t \in T$ such that $\forall s \in G(i)(s < t)$ and $\forall s \in G(j)(s > t)$. Independently, there may be a "textual representation" of each non-empty granule, termed its *label*, that is used for input and output. This representation is generally a string that is more descriptive than the granule's index. An associated mapping, the *label mapping*, defines for each label a unique corresponding index. This mapping can be quite complex, dealing with different languages and character sets, or can be omitted if integers are used directly to refer to granules. For example, "August 2008" and "September 2008" are two labels each referring to the set of time instants (a granule) corresponding to that month.

A granularity is *bounded* if there exist lower and upper bounds $k_1$ and $k_2$ in the index set such that $G(i) = \emptyset$ for all $i$ with $i < k_1$ or $k_2 < i$.

The usual collections `days`, `months`, `weeks` and `years` are granularities. The granularity describing all years starting from 2000 can be defined as a mapping that takes an arbitrary index $i$ to the subset of the time domain corresponding to the year 2000, $i + 1$ to the one corresponding to the year 2001, and so on, with all indexes less than $i$ mapped to the empty set. The years from 2006 to 2010 can also be represented as a granularity $G$, with $G(2006)$ identifying the subset of the time domain corresponding to the year 2006, $G(2007)$ to 2007, and so on, with $G(i) = \emptyset$ for each $i < 2006$ and $i > 2010$.

The union of all the granules in a granularity $G$ is called the *image* of $G$. For example, the image of `business-days-since-2000` is the set of time instants included in each granule representing a business-day, starting from the first one in 2000. The single interval of the time domain starting with the greatest lower bound of the image of a granularity $G$ and ending with the least upper bound ($-\infty$ and $+\infty$ are considered valid lower/upper bounds) is called the *extent* of $G$. Note that many commonly used granularities (e.g., `days`, `months`, `years`) have their image equal to their extent, since each granule is formed by a set of contiguous elements of the time domain and each pair of contiguous indexes is mapped to contiguous granules.

## Granularity Relationships

In the following, some commonly used relationships between granularities are given.

> A granularity $G$ *groups into* a granularity $H$, denoted $G \trianglelefteq H$, if for each index $j$ there exists a (possibly infinite) subset $S$ of the integers such that $H(j) = \bigcup_{i \in S} G(i)$.

For example, `days` groups into `weeks`, but `weeks` does not group into `months`.

> A granularity $G$ is *finer than* a granularity $H$, denoted $G \preceq H$, if for each index $i$, there exists an index $j$ such that $G(i) \subseteq H(j)$. If $G \preceq H$, then $H$ is *coarser than* $G$ ($H \succeq G$).

For example, `business-days` is finer than `weeks`, while `business-days` does not group into `weeks`; `business-days` is finer than `years`, while `weeks` is not.

> A granularity $G$ *groups periodically into* a granularity $H$ if:
> 1) $G \trianglelefteq H$, and
> 2) there exist $n, m \in \mathbf{Z}^+$, where $n$ is less than the number of non-empty granules of $H$, such that for all $i \in \mathbf{Z}$, if $H(i) = \bigcup_{r=0}^{k} G(j_r)$ and $H(i+n) \neq \emptyset$, then $H(i+n) = \bigcup_{r=0}^{k} G(j_r + m)$.

The *groups periodically into* relationship is a special case of *groups into* characterized by a periodic repetition of the "grouping pattern" of granules of $G$ into granules of $H$. Its definition may appear complicated, but it is actually quite simple. Since $G$ groups into $H$, any non-empty granule $H(i)$ is the union of some granules of $G$; for instance, assume it is the union of the granules $G(a_1), G(a_2), \ldots, G(a_k)$. The periodicity property (condition 2 in the definition) ensures that the $n^{\text{th}}$ granule *after* $H(i)$, i.e., $H(i+n)$, if non-empty, is the union of $G(a_1+m), G(a_2+m), \ldots, G(a_k+m)$. This results in a periodic "pattern" of the composition of $n$ granules of $H$ in terms of granules of $G$. The pattern repeats along the time domain by "shifting" each granule of $H$ by $m$ granules of $G$. Many common granularities are in this kind of relationship. For example, `days` groups periodically into `business-days`, with $m = 7$ and $n = 5$, and also groups periodically into `weeks`, with $m = 7$ and $n = 1$; `months` groups periodically into `years` with $m = 12$ and $n = 1$, and `days` groups periodically into `years` with $m = 14,697$ and $n = 400$. Alternatively, the relationship can also be described, by saying, for example, `years` is periodic (or 1-periodic) with respect to `months`, and `years` is periodic (or 400-periodic) with respect to `days`. In general, this relationship guarantees that granularity $H$ can be finitely described in terms of granules of $G$. More details can be found in [3].

> Given a granularity order relationship *g-rel* and a set of granularities, a granularity $G$ in the set is a *bottom granularity* with respect to *g-rel*, if $G$ *g-rel* $H$ for each granularity $H$ in the set.

For example, given the set of all granularities defined over the time domain $(\mathbf{R}; \leq)$, and the granularity relationship $\preceq$ (finer than), the granularity corresponding to the *empty* mapping is the bottom granularity with respect to $\preceq$. Given the set of all granularities defined over the time domain $(\mathbf{Z}; \leq)$, and the granularity relationship $\trianglelefteq$ (groups into), the granularity mapping each index into the corresponding instant (same integer number as the index) is a bottom granularity with respect to $\trianglelefteq$. An example of a set of granularities without a bottom (with respect to $\preceq$ or $\trianglelefteq$) is {`weeks`, `months`}.
Calendars are typically used to describe events or time-related properties over the same span of time using different granularities. For example, the Gregorian calendar comprises the granularities `days`, `months`, and `years`. Considering the notion of bottom granularity, a formal definition of calendar follows.

> A *calendar* is a set of granularities that includes a bottom granularity with respect to $\trianglelefteq$ (groups into).

## Defining new granularities through algebraic operators

In principle, every granularity in a calendar can be defined in terms of the bottom granularity, possibly specifying the composition of granules through the relationships defined above. Several proposals have appeared in the literature for a set of algebraic operators with the goal of facilitating the definition of new granularities in terms of existing ones. These algebras are evaluated with respect to expressiveness, user friendliness, and ability to compute operations on granularities directly on the algebraic representation. Some of the operations that are useful in applications are inter-granule conversions; for example, to compute which day of the week was the k-th day of a particular year, or which interval of days was r-th week of that year, as well as conversions involving different calendars. In the following, the main operators of one of the most expressive calendar algebras [3] are briefly described. Two operators form the backbone of the algebra:

a) The *grouping* operator systematically combines a few granules of the source granularity into one granule in the target granularity. For example, given granularity `days`, granularity `weeks` can be generated by combining 7 granules (corresponding to Monday – Sunday) `week` = $Group_7$(`day`) if we assume that `day`(1) corresponds to Monday, i.e., the first day of a week.

b)The *altering-tick* operator deletes or adds granules from a given granularity (via the help of a second granularity) to form a new one. For example, assume each 30 days are grouped into a granule, forming a granularity `30-day-groups`. Then, an extra day can be added for each January, March, and so on, while two days are dropped from February. The February in leap years can be similarly changed to have an extra day, hence properly representing `month`.

Other auxiliary operators include:

- *shift* shifts the index forward or backward a few granules (e.g., the granule used to be labeled 1 may be re-labeled 10) in order to provide proper alignment for further operations.

- *combine* combines all the granules of one granularity that fall into (using finer-than relationship) a second granularity. For example, by combining all the business days in a week, `business-week` is obtained.

- *subset* generates a new granularity by selecting an interval of granules from a given granularity. For example, choosing the days between year 2000-2010, leads to a granularity that only contains days in these years.

- *select* generates new granularities by selecting granules from the first operand in terms of their relationship with the granules of the second operand. For example, selecting the first day of each week gives the `Mondays` granularity.

More details, including other operators, conditions of applicability, as well as comparison with other algebra proposals can be found in [3]. The algebra directly supports the inter-granule and inter-calendar conversions mentioned above. Some more complex operations on granularities (e.g., temporal constraint propagation in terms of multiple granularities) and the verification of formal properties (e.g., equivalence of algebraic granularity representations) require a conversion in terms of a given bottom granularity. An efficient automatic procedure for this conversion has been devised and implemented [2]. The automaton-based approach may be a valid alternative for the verification of formal properties.

## KEY APPLICATIONS*

Temporal granularities are currently used in several applications, but in most cases their use is limited to very few standard granularities, supported by system calendar libraries and ad-hoc solutions are used to manipulate data associated with them. This approach often leads to unclear semantics, hidden mistakes, low interoperability, and does not take advantage of user-defined granularities and complex operations. The impact of a formal framework for temporal granularities has been deeply investigated for a number of application areas among which logical design of temporal databases, querying databases in terms of arbitrary granularities, data mining, integrity constraint satisfaction, workflows [3]. For example, when temporal dependencies in terms of granularities can be identified in the data (e.g., "salaries of employees do not change within a fiscal year"), specific techniques have been devised for the logical design of temporal databases that can lead to significant benefits. In query processing, it has been shown how to support the retrieval of data in terms of temporal granularities different from the ones

associated to the data stored in the database, provided that assumptions on the data semantics are formalized, possibly as part of the database schema. Time distance constraints in terms of granularities (e.g., *Event2 should occur within two business days after the occurrence of Event1*) have been extensively studied, and algorithms proposed to check for consistency and solutions. Applications include the specification of classes of frequent patterns to be identified in time series, the specification of integrity contraints in databases, and the specification of constraints on activities durations and on temporal distance between specific events in workflows.
Temporal granularities also have several applications in other areas like natural language processing, temporal reasoning in AI, including scheduling and planning, and in computer logic, where they have been mainly considered for program specification and verification.
Future applications may include advanced personal information management (PIM). Time and location-aware devices coupled with advanced calendar applications, supporting user-defined granularities, may offer innovative personalized scheduling and alerting systems.

## EXPERIMENTAL RESULTS*

Several systems and software packages dealing with temporal granularities have been developed, among which MultiCal, Calendar algebra implementation, GSTP Project, and TauZaman. Information about these systems can be easily found online.

## CROSS REFERENCE*

Time domain, Temporal periodicity, Temporal dependencies, Temporal constraints, Temporal data mining, TSQL2.

## RECOMMENDED READING

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] C. Bettini, C.E. Dyreson, W.S. Evans, R.T. Snodgrass, and X. Wang. A glossary of time granularity concepts. In *Temporal Databases, Dagstuhl*, pages 406–413. Springer, 1997.

[2] C. Bettini, S. Mascetti, and X. Wang. Supporting temporal reasoning by mapping calendar expressions to minimal periodic sets. *Journal of Artificial Intelligence Research*, 28:299–348, 2007.

[3] Claudio Bettini, X. Sean Wang, and Sushil Jajodia. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer, 2000.

[4] R. Chandra, A. Segev, and M. Stonebraker. Implementing calendars and temporal rules in next generation databases. In *Proc. of the Tenth International Conference on Data Engineering*, pages 264–273, 1994.

[5] J. Clifford and A. Rao. A simple, general structure for temporal domains. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pages 23–30, France, May 1987. AFCET.

[6] C. Combi, M. Franceschet, and A. Peron. Representing and reasoning about temporal granularities. *Journal of Logic and Computation*, 14(1):51–77, 2004.

[7] U. Dal Lago, A. Montanari, and G. Puppis. Compact and tractable automaton-based representations of time granularities. *Theoretical Computer Science*, 373(1-2):115–141, 2007.

[8] J. Euzenat and A. Montanari. Time granularity. In M. Fisher, D.Gabbay, and L. Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[9] P. Ladkin. The completness of a natural system for reasoning with time intervals. In *Proc. of the 10 $^{th}$ International Joint Conference on Artificial Intelligence*, pages 462–467, 1987.

[10] B. Leban, D. McDonald, and D. Forster. A representation for collections of temporal intervals. In *Proc. of the 5 $^{th}$ National Conference on Artificial Intelligence*, pages 367–371, 1986.

[11] N.A. Lorentzos. DBMS support for nonmetric measurement systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):945–953, 1994.

[12] B. Urgun, C.E. Dyreson, R.T. Snodgrass, J.K. Miller, N. Kline, M.D. Soo, and C.S. Jensen. Integrating multiple calendars using tau-ZAMAN. *Software - Practice and Experience*, 37(3):267–308, 2007.

[13] X. Wang, S. Jajodia, and V.S. Subrahmanian. Temporal modules: An approach toward federated temporal databases. *Information Sciences*, 82:103–128, 1995.

[14] G. Weiderhold, S. Jajodia, and W. Litwin. Integrating temporal data in a heterogeneous environment. In *Temporal Databases: Theory, Design, and Implementation*, pages 563–579. Benjamin/Cummings, 1993.

[15] J. Wijsen. A string-based model for infinite granularities. In *Spatial and Temporal Granularity: Papers from the AAAI Workshop*, AAAI Technical Report WS-00-08, pages 9–16. AAAI Press, 2000.

# TEMPORAL HOMOGENEITY

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

None

**DEFINITION**

Assume a temporal relation where the attribute values of tuples are (partial) functions from some time domain to value domains. A tuple in such a relation is *temporally homogeneous* if the domains of all its attribute values are identical. A temporal relation is temporally homogeneous if all its tuples are temporally homogeneous. Likewise, a temporal database is temporally homogeneous if all its relations are temporally homogeneous.

In addition to being specific to a type of object (tuple, relation, database), homogeneity is also specific to a time dimension when the time domain is multidimensional, as in "temporally homogeneous in the valid-time dimension" or "temporally homogeneous in the transaction-time dimension."

**MAIN TEXT**

The motivation for homogeneity arises from the fact that no timeslices of a homogeneous relation produce null values. Therefore, a homogeneous relational model is the temporal counterpart of the snapshot relational model without nulls. Certain data models assume temporal homogeneity, while other models do not.

A tuple-timestamped temporal relation may be viewed as a specific attribute-value timestamped relation. An attribute value $a$ of a tuple with timestamp $t$ is represented by a function that maps each value in $t$ to $a$. Thus, models that employ tuple timestamping are necessarily temporally homogeneous.

**CROSS REFERENCE***

Lifespan, Temporal Database, SQL-Based Temporal Query Languages, Transaction Time, Valid Time

**REFERENCES***

S. K. Gadia, A Homogeneous Relational Model and Query Languages for Temporal Databases, *ACM Transactions on Database Systems* 13(4):418–448, December 1988.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TITLE

temporal indeterminacy

# BYLINE

Curtis Dyreson
Utah State University
Curtis.Dyreson@usu.edu
http://www.cs.usu.edu/~cdyreson

# SYNONYMS

fuzzy time, imprecise time, temporal incompleteness

# DEFINITION

Temporal indeterminacy refers to “don't know when” information, or more precisely, “don't know *exactly* when.” The modifier ‘temporally indeterminate’ indicates that the modified object has an associated time, but that the time is not known precisely. The time when an event happens, when a time interval begins or ends, or even the duration of a period may be indeterminate. For example, the event of a car accident might be “sometime last week,” the interval an airplane flight takes may be from “Friday to Saturday,” or the duration a graduate student takes to write a dissertation may be “four to fifteen years.”

The adjective ‘temporal’ allows parallel kinds of indeterminacy to be defined, such as spatial indeterminacy. There is a subtle difference between indeterminate and imprecise. In this context, indeterminate is a more general term than imprecise since precision is commonly associated with making measurements. Typically, a precise measurement is preferred to an imprecise one. Imprecise time measurements, however, are just one source of temporally indeterminate information.

# HISTORICAL BACKGROUND

Despite the wealth of research on adding incomplete information to databases, there are few efforts that address incomplete temporal information [6]. Much of the previous research in incomplete information databases has concentrated on issues related to null values, the applicability of fuzzy set theory, and the integration of various combinations of probabilistic reasoning, temporal reasoning, and planning.

In the earliest work on temporal indeterminacy, an indeterminate instant was modeled with a set of possible chronons [13]. Dutta next introduced a fuzzy set approach to handle events that can be interpreted to have multiple occurrences [5]. For example the event “Margaret's salary is high” may occur at various times as Margaret's salary fluctuates to reflect promotions and demotions. The meaning of “high” is incomplete, it is not a crisp predicate. In Dutta's model all the possibilities for high are represented in a *generalized* event and the user selects some subset according to his or her interpretation of “high.” Generalized bitemporal elements were defined somewhat differently in a later paper by Kouramajian and Elmasri [12]. Bitemporal elements

combine transaction time and valid time in the same temporal element, and can include a non-contiguous (i.e., indeterminate) set of noncontiguous possible times. Gadia et al. proposed an interesting model that intertwines support for value and temporal incompleteness [8]. By combining the different kinds of incomplete information, a wide spectrum of attribute values are simultaneously modeled, including values that are completely known, values that are unknown but are known to have occurred, values that are known if they occurred, and values that are unknown even if they occurred. Dyreson and Snodgrass proposed using probabilistic events to model temporal indeterminacy. In their model a time is represented as a probability distribution [7]. Probabilistic times were also comprehensively addressed by Dekhtyar et al. [4].

Reasoning with incomplete information can be computationally expensive. Koubarakis was the first to focus attention on the issue of cost [10][11]. He showed that by restricting the kinds of constraints allowed in representing the indeterminacy, polynomial time algorithms can be obtained. Koubarakis proposed a temporal data model with global and local inequality constraints on the occurrence time of an event. Another constraint-based temporal reasoner is LaTeR, which has been successfully implemented [2]. LaTeR similarly restricts the kinds of constraints allowed (to conjunctions of linear inequalities), but this class of constraints includes many of the important temporal predicates [3].

Temporal indeterminacy has also been addressed in non-relational contexts, for instance in object-oriented databases [1].

# SCIENTIFIC FUNDAMENTALS

There are (at least) three possible sources of indeterminacy in a statement with respect to time: (i) a discrepancy between the *granularity* of the temporal qualification and the occurrence time; (ii) an under-specification of the occurrence time when the granularities of the temporal qualification and the occurrence time coincide; and (iii) relative times.

As a first approximation, a statement is *temporally indeterminate* if the granularity of its reference to time (in the examples, the granularity of days) is coarser than the granularity of the time at which the denoted event(s) occur. Temporal indeterminacy as well as relativity of reference to time is mainly a qualification of a statement rather than of the event it denotes (that is, temporal indeterminacy characterizes the relationship between the granularity of the time reference of a statement and the granularity of an event's occurrence time). It does not depend on the time at which the statement is evaluated. The crucial and critical point is the determination of the time granularity of the event occurrence time.

Generally, a statement whose reference to time has a granularity (e.g., days) which is temporally determinate with respect to every coarser granularity (e.g., months) and temporally indeterminate with respect to every finer granularity (e.g., seconds). But this general rule has exceptions since it does not take into account information about the denoted occurrence time. In particular, for a *macro-event* there exists a (finest) granularity at which its occurrence time can be specified, but with respect to finer granularities, the event as a whole does not make sense, and must, if possible, be decomposed into a set of components.

But not all cases of temporally indeterminate information involve a discrepancy between the granularity of the reference to time and the granularity of the occurrence time. Consider the sentence: "The shop remained open on a Sunday in April 1990 all the day long." 'Days' is the granularity of both the time reference and the occurrence time. Nevertheless, this statement is

temporally indeterminate because the precise day in which the shop remained open is unknown (it is known only that it is one of the Sundays in April 1990).

Statements that contain a relative reference to time are also temporally indeterminate, but the reverse does not hold: temporally-indeterminate statements can contain relative as well as absolute references to time. The statements "Jack was killed sometime in 1990" and "Michelle was born yesterday" contain absolute and relative references to time, respectively, but they are both temporally indeterminate.

The following example illustrates how temporal indeterminacy can be represented in a relational database. Consider the employment relation shown in Figure 1 which is cobbled together from the (somewhat hazy) memories of several employees. Each tuple shows a worker's name, salary, department, and time employed (i.e., the valid time). The first tuple represents Joe's employment in Shoes. It is temporally indeterminate since the exact day when Joe stopped working in Shoes is not known precisely; the ending valid time is recorded as sometime in January 2005 and represented as the indeterminate event "`1 Jan 2005~31 Jan 2005`". Joe then went to work in Admin, which is also temporally indeterminate. Joe started working in Admin "`After leaving Shoes`" which is a temporal constraint on an indeterminate time. The third tuple represents Sue's employment history. She began working in Shoes sometime in the first half of January 2005 ("`1 Jan 2005~15 Jan 2005`") with a uniform probability for each day in that range (which is more information than is known about Joe's termination in Shoes, the probability is missing from that indeterminate event). Finally, Eve began working in Admin on some Monday in January 2005, but it is not known which Monday.

| Name | Dept. | Sal. | Valid Time |
|---|---|---|---|
| Joe | Shoes | 40K | [1 Jan 2003 – 1 Jan 2005~31 Jan 2005] |
| Joe | Admin | 100K | [After leaving Shoes - now] |
| Sue | Shoes | 50K | [1 Jan 2005~15 Jan 2005 (uniform) – now] |
| Eve | Admin | 90K | [A Monday in January 2005 – now] |

**Figure 1 A Relation with Temporal Indeterminacy**

Querying temporal indeterminacy is more challenging than representing it. The two chief challenges are efficiency and expressiveness. Consider a query to find out who was employed on January 10, 2005 as expressed in a temporal version of SQL (e.g., TSQL2) below.

```
SELECT Name
FROM Employee E
WHERE VALID(E) overlaps "10 Jan 2005"
```

There are two well-defined limits on querying incomplete information: the *definite* and the *possible*. The definite answer includes only information that is known. On a tuple-by-tuple basis, determining which employment tuple definitely overlaps January 10, 2005 is straightforward: *none* definitely do. In contrast, every tuple *possibly* overlaps that day. It is efficient to compute both bounds on a tuple-by-tuple basis, but not very expressive. It would be more expressive to be able to find other answers that lie between the bounds. Probabilistic approaches seek to refine the set of potential answers by reasoning with probabilities. For instance, can it be computed who

was *probably* employed (exceeding a probability of 0.5)? The probability that Sue began working before January 10, 2005 is .67 (the probability mass is uniformly distributed among all the possibilities). Since the probability mass function for Joe's termination in Shoes is missing, he can not be included in the employees who were probably employed.

Most of the existing approaches have focused on improving the efficiency of computing probabilistic answers; the *usability* of probabilistic approaches has not yet been determined. It might be very difficult for users to interpret and use probabilities (should a user interpret "probably" as exceeding .75 or .5?) so other approaches (e.g., fuzzy set approaches) may be needed to improve usability. A second research issue concerns reasoning with *intra-tuple* constraints. The definite answer given above is inaccurate. Even though it is not known exactly when Joe stopped working in Shoes or started working in Admin, it is known that he was employed in one or the other on January 10, 2005. The intra-tuple constraint in the second tuple represents this knowledge. Though intra-tuple constraints provide greater reasoning power, reasoning with them often has high computational complexity. Research continues in defining classes of constraints that are meaningful and computationally feasible. Finally, temporal indeterminacy has yet to be considered in new kinds of queries (e.g., roll-up in data warehouses and top-k queries) and new temporal query languages (e.g., τXQuery and TOWL).

# KEY APPLICATIONS

The most common kinds of temporal indeterminacy are valid-time indeterminacy and user-defined time indeterminacy. Transaction-time indeterminacy is rarer because transaction times are always known exactly. Temporal indeterminacy can occur in logistics, especially in planning scenarios where project completion dates are typically inexact. In some scientific fields it is quite rare to know an exact time, for instance, archeology is replete with probabilistic times generated by radio-carbon dating, tree-ring analyses, and by the layering of artifacts and sediments. Indeterminacy can arise even when precise clocks are employed. In a network of road sensors, an "icy road" has an indeterminate lifetime as the change from non-icy to icy is gradual rather than instantaneous; "icy road" has a fuzzy starting and ending time.

# CROSS REFERENCES

temporal constraints, qualitative temporal reasoning, probabilistic temporal databases, temporal granularity

# RECOMMENDED READING


[1] Veronica Biazzo, Rosalba Giugno, Thomas Lukasiewicz, V. S. Subrahmanian. Temporal Probabilistic Object Bases. *IEEE Transactions on Knowledge and Data Engineering*. **15**(4). 2003, pp. 921-939.

[2] Vittorio Brusoni, Luca Console, Paolo Terenziani, and Barbara Pernici, "Extending Temporal Relational Databases to Deal with Imprecise and Qualitative Temporal Information," in *Recent Advances in Temporal Databases*, pp. 3-22, Zurich, Switzerland, September 1995.

[3] Vittorio Brusoni, Luca Console, Paolo Terenziani, Barbara Pernici. Qualitative and Quantitative Temporal Constraints and Relational Databases: Theory, Architecture, and Applications. *IEEE Transactions on Knowledge and Data Engineering*, **11**(6): 948-968 (1999).

[4] Alex Dekhtyar, Robert Ross, and V. S. Subrahmanian. Probabilistic temporal databases, I: algebra. *ACM Transactions on Database Systems*, **26**(1), 2001, pp. 41-95.

[5] S. Dutta, "Generalized Events in Temporal Databases," in *Proceedings of the Fifth International Conference on Data Engineering*, pp. 118-126, Los Angeles, CA, February 1989.

[6] Curtis Dyreson. A Bibliography on Uncertainty Management in Information Systems. In *Uncertainty Management in Information Systems: From Needs to Solutions*, Kluwer Academic Publishers, 1997. pp. 415-458.

[7] Curtis Dyreson and Richard T. Snodgrass. Supporting Valid-time Indeterminacy. *ACM Transactions on Database Systems*, **23**(1), March 1998, pp. 1-57.

[8] Shashi K. Gadia, Sunil S. Nair, and Yiu-Cheong Poon. "Incomplete Information in Relational Temporal Databases," in *Proceedings of the International Conference on Very Large Databases*, Vancouver, Canada, August 1992, pp. 395-406.

[9] Fabio Grandi, Federica Mandreoli: Effective Representation and Efficient Management of Indeterminate Dates. TIME 2001: 164-169.

[10] Manolis Koubarakis, "Representation and Querying in Temporal Databases: The Power of Temporal Constraints," in *Proceedings of the International Conference on Data Engineering*, pp. 327-334, Vienna, Austria, April 1993.

[11] Manolis Koubarakis: The Complexity of Query Evaluation in Indefinite Temporal Constraint Databases. *Theoretical Computer Science*, **171**(1-2): 25-60 (1997).

[12] Vijay Kouramajian and Ramez Elmasri, "A Generalized Temporal Model," in the *Uncertainty in Databases and Deductive Systems Workshop*, Ithaca, NY, November, 1994.

[13] Richard T. Snodgrass. Monitoring Distributed Systems: A Relational Approach. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, December 1982.

# Temporal Integrity Constraints

Jef Wijsen
University of Mons-Hainaut

**SYNONYMS**

Dynamic integrity constraints

**DEFINITION**

Temporal integrity constraints are integrity constraints formulated over temporal databases. They can express dynamic properties by referring to data valid at different time points. This is to be contrasted with databases that do not store past or future information: if integrity constraints can only refer to data valid at the current time, they can only express static properties. Languages for expressing temporal integrity constraints extend first-order logic with explicit timestamps or with temporal connectives. An important question is how to check and enforce such temporal integrity constraints efficiently.

**HISTORICAL BACKGROUND**

The use of first-order temporal logic for expressing temporal integrity constraints dates back to the early eighties (see for example [7]). Since the late eighties, progress has been made in the problem of checking temporal integrity [9, 2, 11] without having to store the entire database history. This entry deals with general temporal integrity constraints. The entry Temporal Dependencies deals with temporal variants of specific constraints, in particular with temporal extensions of functional dependencies.

**SCIENTIFIC FUNDAMENTALS**

Integrity constraints, whether they are temporal or not, are an important component of each database schema. They express properties that, ideally, must be satisfied by the stored data at all times. If a database satisfies all the integrity constraints, it is called *consistent*. Integrity constraints are commonly expressed in a declarative way using logic. Declarative integrity constraints generally do not specify how to keep the database consistent when data is inserted, deleted, and modified. An important task is to develop efficient procedures for checking and enforcing such constraints.

Temporal databases store past, current, and future information by associating time to database facts. An integrity constraint can be called "temporal" if it is expressed over a temporal database. By relating facts valid at different points in time, temporal integrity constraints can put restrictions on how the data can change over time. This is to be contrasted with databases that do not store past or future facts: if integrity constraints can only refer to a single database state, they cannot capture time-varying properties of data.

The following section deals with languages for expressing temporal integrity. Section 2 discusses techniques for checking and enforcing temporal integrity constraints.

## 1 Defining Temporal Integrity

While temporal integrity constraints can in principle be expressed as Boolean queries in whichever temporal query language, it turns out that temporal logic on timepoint-stamped data is the prevailing formalism for defining and studying temporal integrity. Section 1.1 illustrates several types of temporal constraints. Section 1.2 deals with several notions of temporal constraint satisfaction. Sections 1.3 and 1.4 contain considerations on expressiveness

and on interval-stamped data.

## 1.1 Temporal, Transition, and Static Constraints

Since first-order logic is the lingua franca for expressing non-temporal integrity constraints, it is natural to express temporal integrity constraints in temporal extensions of first-order logic. Such temporalized logics refer to time either through variables with a type of time points, or by temporal modal operators, such as:

| Operator | Meaning |
|---|---|
| $\bullet p$ | Property $p$ was true at the previous time instant. |
| $\blacklozenge p$ | Property $p$ was true sometime in the past. |
| $\bigcirc p$ | Property $p$ will be true at the next time instant. |
| $\Diamond p$ | Property $p$ will be true sometime in the future. |

Satisfaction of such constraints by a temporal database can be checked if the question "Does (did/will) $R(a_1, \ldots, a_m)$ hold at $t$?" can be answered for any fact $R(a_1, \ldots, a_m)$ and time instant $t$. Alternatively, one can equip facts with time and ask: "Is $R(a_1, \ldots, a_m \mid t)$ true?". Thus, one can abstract from the concrete temporal database representation, which may well contain interval-stamped facts [6]. Every time point $t$ gives rise to a *(database) state* containing all facts true at $t$.

The following examples assume a time granularity of days. The facts $\mathit{WorksFor}(\text{John Smith}, \text{Pulse})$ and $\mathit{Earns}(\text{John Smith}, 20\text{K})$, true on 10 August 2007, express that John worked for the Pulse project and earned a salary of 20K at that date. The alternative encoding is $\mathit{WorksFor}(\text{John Smith}, \text{Pulse} \mid 10\text{ Aug }2007)$ and $\mathit{Earns}(\text{John Smith}, 20\text{K} \mid 10\text{ Aug }2007)$.

Although temporal integrity constraints can generally refer to any number of database states, many natural constraints involve only one or two states. In particular, *transition constraints* only refer to the current and the previous state; *static constraints* refer only to the current state.

The first-order temporal logic (FOTL) formulas (1)–(4) hereafter illustrate different types of integrity constraints. The constraint *"An employee who is dropped from the Pulse project cannot work for that project later on"* can be formulated in FOTL as follows:

$$\neg\exists x(\mathit{WorksFor}(x, \text{Pulse}) \wedge \blacklozenge(\neg\mathit{WorksFor}(x, \text{Pulse}) \wedge \bullet\mathit{WorksFor}(x, \text{Pulse}))) \tag{1}$$

This constraint can be most easily understood by noticing that the subformula $\blacklozenge(\neg\mathit{WorksFor}(x, \text{Pulse}) \wedge \bullet\mathit{WorksFor}(x, \text{Pulse}))$ is true in the current database state for every employee $x$ who was dropped from the Pulse project sometime in the past (this subformula will recur in the discussion of integrity checking later on). This constraint can be equivalently formulated in a two-sorted logic, using two temporal variables $t_1$ and $t_2$:

$$\neg\exists x\exists t_1\exists t_2((t_1 < t_2) \wedge \mathit{WorksFor}(x, \text{Pulse} \mid t_2) \wedge \neg\mathit{WorksFor}(x, \text{Pulse} \mid t_1) \wedge \mathit{WorksFor}(x, \text{Pulse} \mid t_1 - 1))$$

The constraint *"Today's salary cannot be less than yesterday's"* is a transition constraint, and can be formulated as follows:

$$\neg\exists x\exists y\exists z(\mathit{Earns}(x, y) \wedge \bullet(\mathit{Earns}(x, z) \wedge y < z)) \tag{2}$$

Finally, static constraints are illustrated. The most fundamental static constraints in the relational data model are primary keys and foreign keys. The constraint *"No employee has two salaries"* implies that employee names uniquely identify *Earns*-tuples in any database state; it corresponds to a standard primary key. The constraint *"Every employee in the WorksFor relation has a salary"* means that in any database state, the first column of *WorksFor* is a foreign key that references (the primary key of) *Earns*. These constraints can be formulated in FOTL without using temporal connectives:

$$\forall x\forall y\forall z(\mathit{Earns}(x, y) \wedge \mathit{Earns}(x, z) \rightarrow y = z) \tag{3}$$

$$\forall x\forall y(\mathit{WorksFor}(x, y) \rightarrow \exists z\mathit{Earns}(x, z)) \tag{4}$$

Formulas (3) and (4) show a nice thing about using FOTL for expressing temporal integrity: static constraints read as non-temporal constraints expressed in first-order logic. □

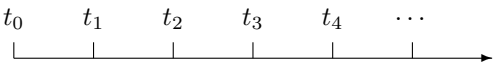
## 1.2 Different Notions of Consistency

The syntax and semantics of the temporal logic used in the previous section are defined next. Different notions of temporal constraint satisfaction are discussed.

Assume a countably infinite set **dom** of constants. In the following syntax, $R$ is any relation name and each $s_i$ is a variable or a constant:

$$C, C' ::= R(s_1, \ldots, s_m) \mid s_1 = s_2 \mid C \wedge C' \mid \neg C \mid \exists x(C) \mid \bigcirc C \mid \Diamond C \mid \bullet C \mid \blacklozenge C$$

The connectives $\bullet$ and $\blacklozenge$ are called *past operators*; $\bigcirc$ and $\Diamond$ are *future operators*. A *past formula* is a formula without future operators; a *future formula* is a formula without past operators. Other modal operators, like the past operator **since** and the future operator **until**, can be added to increase expressiveness; see the entry Temporal Logic in Database Query Languages. The set of time points is assumed to be infinite, discrete and linearly ordered with a smallest element. Thus, the time scale can be depicted as follows:




Discreteness of time is needed in the interpretation of the operators $\bigcirc$ and $\bullet$. Formulas are interpreted relative to an infinite sequence $H = \langle H_0, H_1, H_2, \ldots \rangle$, where each $H_i$ is a finite set of facts formed from relation names and constants of **dom**. Intuitively, $H_i$ contains the facts true at time $t_i$. Such a sequence $H$ is called an *infinite (database) history* and each $H_i$ a *(database) state*. The following rules define inductively what it means for a closed formula $C$ to be *satisfied* by $H$, denoted $H \models_{\mathsf{inf}} C$.

| | | |
|---|---|---|
| $H, i \models_{\mathsf{inf}} R(a_1, \ldots, a_m)$ | iff | $R(a_1, \ldots, a_m) \in H_i$ |
| $H, i \models_{\mathsf{inf}} a_1 = a_2$ | iff | $a_1 = a_2$ |
| $H, i \models_{\mathsf{inf}} C \wedge C'$ | iff | $H, i \models_{\mathsf{inf}} C$ and $H, i \models_{\mathsf{inf}} C'$ |
| $H, i \models_{\mathsf{inf}} \neg C$ | iff | not $H, i \models_{\mathsf{inf}} C$ |
| $H, i \models_{\mathsf{inf}} \exists x(C)$ | iff | $H, i \models_{\mathsf{inf}} C[x \mapsto a]$ for some $a \in$ **dom**, where $C[x \mapsto a]$ is obtained from $C$ by replacing each free occurrence of $x$ with $a$ |
| $H, i \models_{\mathsf{inf}} \bullet C$ | iff | $i > 0$ and $H, i-1 \models_{\mathsf{inf}} C$ |
| $H, i \models_{\mathsf{inf}} \blacklozenge C$ | iff | $H, j \models_{\mathsf{inf}} C$ for some $j$ such that $0 \leq j < i$ |
| $H, i \models_{\mathsf{inf}} \bigcirc C$ | iff | $H, i+1 \models_{\mathsf{inf}} C$ |
| $H, i \models_{\mathsf{inf}} \Diamond C$ | iff | $H, j \models_{\mathsf{inf}} C$ for some $j > i$ |

Note that the truth of each subformula is expressed relative to a *single* "reference" time point $i$ (along with $H$). This characteristic allows efficient techniques for integrity checking [5] and seems crucial to the success of temporal logic. Finally:

$$H \models_{\mathsf{inf}} C \quad \text{iff} \quad H, j \models_{\mathsf{inf}} C \text{ for each } j \geq 0$$

Consistency of infinite database histories is of theoretical interest. In practice, only a finite prefix of $H$ will be known at any one time. Consider, for example, the situation where $H_0$ is the initial database state, and for each $i \geq 0$, the state $H_{i+1}$ results from applying an update to $H_i$. Since every update can be followed by another one, there is no last state in this sequence. However, at any one time, only some *finite history* $\langle H_0, \ldots, H_n \rangle$ up to the most recent update is known, and it is of practical importance to detect constraint violations in such a finite history. It is reasonable to raise a constraint violation when the finite history obtained so far cannot possibly be extended to an infinite consistent history. For example, the constraints

$$\neg\exists x(\mathit{Hire}(x) \wedge \neg\Diamond(\mathit{Promote}(x) \wedge \Diamond \mathit{Retire}(x)))$$
$$\neg\exists x(\mathit{Retire}(x) \wedge \Diamond \mathit{Retire}(x))$$

express that all hired people are promoted before they retire, and that no one can retire twice. Then, the finite history $\langle \{\mathit{Hire}(\mathrm{Ed})\}, \{\mathit{Hire}(\mathrm{An})\}, \{\mathit{Retire}(\mathrm{Ed})\} \rangle$ is inconsistent, because of the absence of $\mathit{Promote}(\mathrm{Ed})$ in the second state. It is assumed here that the database history is append-only and that the past cannot be modified. Two different notions, denoted $\models_{\mathsf{pot}}$ and $\models_{\mathsf{fin}}$, of satisfaction for finite histories are as follows:

$\langle H_0, \ldots, H_n \rangle \models_{\mathsf{pot}} C$ if $\langle H_0, \ldots, H_n \rangle$ can be extended to an infinite history $H = \langle H_0, \ldots, H_n, H_{n+1}, \ldots \rangle$ such that $H \models_{\mathsf{inf}} C$.

2. $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{fin}} C$ if $\langle H_0, \ldots, H_n \rangle$ can be extended to an infinite history $H = \langle H_0, \ldots, H_n, H_{n+1}, \ldots \rangle$ such that $H, i \models_{\mathsf{inf}} C$ for each $i \in \{0, 1, \ldots, n\}$.

Obviously, the first concept, called *potential satisfaction*, is stronger than the second one: $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{pot}} C$ implies $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{fin}} C$. Potential satisfaction is the more natural concept. However, Chomicki [2] shows how to construct a constraint $C$, using only past operators ($\bullet$ and $\blacklozenge$), for which $\models_{\mathsf{pot}}$ is undecidable. On the other hand, $\models_{\mathsf{fin}}$ is decidable for constraints $C$ that use only past operators, because the extra states $H_{n+1}, H_{n+2}, \ldots$ do not matter in that case. It also seems that for most practical past formulas, $\models_{\mathsf{pot}}$ and $\models_{\mathsf{fin}}$ coincide [5]. Chomicki and Niwiński [3] define restricted classes of future formulas for which $\models_{\mathsf{pot}}$ is decidable.

## 1.3 Expressiveness of Temporal Constraints

The only assumption about time used in the constraints shown so far, is that the time domain is discrete and linearly ordered. Many temporal constraints that occur in practice need additional structure on the time domain, such as granularity. The constraint *"The salary of an employee cannot change within a month"* assumes a grouping of time instants into months. It can be expressed in a two-sorted temporal logic extended with a built-in predicate $\texttt{month}(t_1, t_2)$ which is true if $t_1$ and $t_2$ belong to the same month:

$$\neg\exists x \exists y_1 \exists y_2 \exists t_1 \exists t_2 (\texttt{month}(t_1, t_2) \wedge \mathit{Earns}(x, y_1 \mid t_1) \wedge \mathit{Earns}(x, y_2 \mid t_2) \wedge y_1 \neq y_2) \ .$$

Arithmetic on the time domain may be needed to capture constraints involving time distances, durations, and periodicity. For example, the time domain $0, 1, 2, \ldots$ may be partitioned into weeks by the predicate $\texttt{week}(t_1, t_2)$ defined by: $\texttt{week}(t_1, t_2)$ if $t_1 \backslash 7 = t_2 \backslash 7$, where $\backslash$ is the integer division operator. If $0 \leq t_2 - t_1 \leq 7$, then one can say that $t_2$ is *within a week from* $t_1$ (even though $\texttt{week}(t_1, t_2)$ may not hold). The entry Temporal Constraints discusses such arithmetic equalities and inequalities on the time domain.

Temporal databases may provide two temporal dimensions for valid time and transaction time. Valid time, used in the preceding examples, indicates when data is true in the real world. Transaction time records the history of the database itself. If both time dimensions are supported, then constraints can explicitly refer to both the history of the domain of discourse and the system's knowledge about that history [10]. Such types of constraints cannot be expressed in formalisms with only one notion of time.

Temporal logics have been extended in different ways to increase their expressive power. Such extensions include fixpoint operators and second-order quantification over sets of timepoints. On the other hand, several important problems, such as potential constraint satisfaction, are undecidable for FOTL and have motivated the study of syntactic restrictions to achieve decidability [3].

This entry focuses on temporal integrity of databases that use (temporal extensions of) the relational data model. Other data models have also been extended to deal with temporal integrity; time-based cardinality constraints in the Entity-Relationship model are an example.

## 1.4 Constraints on Interval-stamped Temporal Data

All constraints discussed so far make abstraction of the concrete representation of temporal data in a (relational) database. They only assume that the database can tell whether a given fact holds at a given point in time. The notion of finite database history (let alone infinite history) is an abstract concept: in practice, all information can be represented in a single database in which facts are timestamped by time intervals to indicate their period of validity. For example,

| *Emp* | *Sal* | *FromTo* |
|---|---|---|
| John Smith | 10K | [1 May 2007, 31 Dec 2007] |
| John Smith | 11K | [1 Jan 2008, 31 Dec 2008] |

Following [10], constraints over such interval-stamped relations can be expressed in first-order logic extended with a type for time intervals and with Allen's interval relations. The following constraint, stating that *"Salaries of employees cannot decrease,"* uses temporal variables $i_1, i_2$ that range over time intervals:

$$\forall x \forall y \forall z \forall i_1 \forall i_2 (\mathit{Earns}(x, y \mid i_1) \wedge \mathit{Earns}(x, z \mid i_2) \wedge \texttt{before}(i_1, i_2) \implies y \leq z) \ .$$

Nevertheless, it seems that many temporal integrity constraints can be most conveniently expressed under an abstract, point-stamped representation. This is definitely true for static integrity constraints, like primary and

foreign keys. Formalisms based on interval-stamped relations may therefore provide operators like *timeslice* or *unfold* to switch to a point-stamped representation. Such "snapshotting" also underlies the *sequenced* semantics [8], which states that static constraints must hold independently at every point in time.
On the other hand, there are some constraints that concern only the concrete interval-stamped representation itself. For example, the interval-stamped relation *Earns* shown below satisfies constraint (3), but may be illegal if there is a constraint stating that temporal tuples need to be coalesced whenever possible.

| *Emp* | *Sal* | *FromTo* |
|---|---|---|
| John Smith | 10K | [1 May 2007, 30 Sep 2007] |
| John Smith | 10K | [1 Oct 2007, 31 Dec 2007] |
| John Smith | 11K | [1 Jan 2008, 31 Dec 2008] |

## 2 Checking and Enforcing Temporal Integrity

Consider a database history to which a new state is added whenever the database is updated. Consistency is checked whenever a tentative update reaches the database. If the update would result in an inconsistent database history, it is rejected; otherwise the new database state is added to the database history. This scenario functions well if the entire database history is available for checking consistency. However, more efficient methods have been developed that allow to check temporal integrity without having to store the whole database history.
To check integrity after an update, there is generally no need to inspect the entire database history. In particular, static constraints can be checked by inspecting only the new database state; transition constraints can be checked by inspecting the previous and the new database state. Techniques for temporal integrity checking aim at reducing the amount of historical data that needs to be considered after a database update. In "history-less" constraint checking [9, 2, 11], all information that is needed for checking temporal integrity is stored, in a space-efficient way, in the current database state. The past states are then no longer needed for the purpose of integrity checking (they may be needed for answering queries, though).
The idea is similar to Temporal Vacuuming and can be formalized as follows. For a given database schema $\mathbf{S}$, let $\mathsf{FIN\_HISTORIES}(\mathbf{S})$ denote the set of finite database histories over $\mathbf{S}$ and $\mathsf{STATES}(\mathbf{S})$ the set of states over $\mathbf{S}$. Given a database schema $\mathbf{S}$ and a set $\mathbf{C}$ of temporal constraints, the aim is to compute a schema $\mathbf{T}$ and a computable function $E : \mathsf{FIN\_HISTORIES}(\mathbf{S}) \to \mathsf{STATES}(\mathbf{T})$, called *history encoding*, with the following properties:

- for every $H \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, the consistency of $H$ with respect to $\mathbf{C}$ must be decidable from $E(H)$ and $\mathbf{C}$. This can be achieved by computing a new set $\mathbf{C}'$ of (non-temporal) first-order constraints over $\mathbf{T}$ such that for every $H \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, $H \models_{\mathsf{temp}} \mathbf{C}$ if and only if $E(H) \models \mathbf{C}'$, where $\models_{\mathsf{temp}}$ is the desired notion of temporal satisfaction (see Section 1.2). Intuitively, the function $E$ encodes, in a non-temporal database state over the schema $\mathbf{T}$, all information needed for temporal integrity checking.
- $E$ must allow an incremental computation when new database states are added: for every $\langle H_0, \ldots, H_n\rangle \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, the result $E(\langle H_0, \ldots, H_n\rangle)$ must be computable from $E(\langle H_0, \ldots, H_{n-1}\rangle)$ and $H_n$. In turn, $E(\langle H_0, \ldots, H_{n-1}\rangle)$ must be computable from $E(\langle H_0, \ldots, H_{n-2}\rangle)$ and $H_{n-1}$. And so on.
  Formally, there must be a computable function $\Delta : \mathsf{STATES}(\mathbf{T}) \times \mathsf{STATES}(\mathbf{S}) \to \mathsf{STATES}(\mathbf{T})$ and an initial database state $J_{\text{init}} \in \mathsf{STATES}(\mathbf{T})$ such that $E(\langle H_0\rangle) = \Delta(J_{\text{init}}, H_0)$ and for every $n > 0$, $E(\langle H_0, \ldots, H_n\rangle) = \Delta(E(\langle H_0, \ldots, H_{n-1}\rangle), H_n)$. The state $J_{\text{init}}$ is needed to get the computation off the ground.

Note that the history encoding is fully determined by the quartet $(\mathbf{T}, J_{\text{init}}, \Delta, \mathbf{C}')$, which only depends on $\mathbf{S}$ and $\mathbf{C}$ (and not on any database history).
Such history encoding was developed by Chomicki [2] for constraints expressed in Past FOTL (including the **since** modal operator). Importantly, in that encoding, the size of $E(H)$ is polynomially bounded in the number of distinct constants occurring in $H$, irrespective of the length of $H$. Chomicki's *bounded history encoding* can be illustrated by constraint (1), which states that an employee cannot work for the Pulse project if he was dropped from that project in the past. The trick is to maintain an auxiliary relation (call it *DroppedFromPulse*, part of the new schema $\mathbf{T}$) that stores names of employees who were dropped from the Pulse project in the past. Thus, $\mathit{DroppedFromPulse}(x)$ will be true in the current state for every employee name $x$ that satisfies $\blacklozenge(\neg \mathit{WorksFor}(x, \text{Pulse}) \wedge \bullet \mathit{WorksFor}(x, \text{Pulse}))$ in the current state. Then, constraint (1) can be checked by checking $\neg\exists x(\mathit{WorksFor}(x, \text{Pulse}) \wedge \mathit{DroppedFromPulse}(x))$, which, syntactically, is a static constraint. Note incidentally that the label "static" is tricky here, because the constraint refers to past information stored in the current database state. Since history-less constraint checking must not rely on past database states, the

auxiliary relation *DroppedFromPulse* must be maintained incrementally (the function $\Delta$): whenever an employee named $x$ is dropped from the Pulse project (i.e. whenever the tuple $WorksFor(x, \text{Pulse})$ is deleted), the name $x$ must be added to the *DroppedFromPulse* relation. In this way, one remembers who has been dropped from Pulse, but forgets when.

History-less constraint checking thus reduces dynamic to static constraint checking, at the expense of storing in auxiliary relations (over the schema **T**) historical data needed for checking future integrity. Whenever a new database state is created as the result of an update, the auxiliary relations are updated as needed (using the function $\Delta$). This technique is suited for implementation in active database systems [11, 4]: a tentative database update will trigger an abort if it violates consistency; otherwise the update is accepted and will trigger further updates that maintain the auxiliary relations.

The approach described above is characterized by ad hoc updates. A different approach is operational: the database can only be updated through a predefined set of update methods (also called transactions). These transactions are specified in a transaction language that provides syntax for embedding elementary updates (insertions and deletions of tuples) in program control structures. Restrictions may be imposed on the possible execution orders of these transactions. Bidoit and de Amo [1] define dynamic dependencies in a declarative way, and then investigate transaction schemes that can generate all and only the database histories that are consistent.

Although it is convenient to use an abstract temporal representation for specifying temporal constraints, consistency checks must obviously be performed on concrete representations. Techniques for checking static constraints, like primary and foreign keys, need to be revised if one moves from non-temporal to interval-timestamped relations [8]. Primary keys can be enforced on a non-temporal relation by means of a unique-index construct. On the other hand, two distinct tuples in an interval-timestamped relation can agree on the primary key without violating consistency. For example, the consistent temporal relation shown earlier contains two tuples with the same name John Smith.

## KEY APPLICATIONS

Temporal data and integrity constraints naturally occur in many database applications. Transition constraints apply wherever the legality of new values after an update depends on the old values, which happens to be very common. History-less constraint checking seems particularly suited in applications where temporal conditions need to be checked, but where there is no need for issuing general queries against past database states. This may be the case in monitor and control applications [12].

## CROSS REFERENCE

Since temporal integrity constraints are usually expressed in temporal logic, the entry Temporal Logic in Database Query Languages is relevant with respect to expressing temporal integrity constraints. The entry Point-stamped Temporal Models deals with the abstract database representation used to interpret temporal logics, and is to be contrasted with Interval-stamped Temporal Models. The formalization of (bounded) history encoding resembles Temporal Vacuuming. For reasons of feasibility or practicality, certain syntactically limited classes of temporal constraints have been studied in their own right; see Temporal Dependencies. The entry Temporal Constraints deals with equations and inequalities on the time domain that allow to define sets of time points.

## RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.


[1] Nicole Bidoit and Sandra de Amo. A first step towards implementing dynamic algebraic dependences. *Theor. Comput. Sci.*, 190(2):115–149, 1998.

[2] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[3] Jan Chomicki and Damian Niwinski. On the feasibility of checking temporal integrity constraints. *J. Comput. Syst. Sci.*, 51(3):523–535, 1995.

[4] Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowl. Data Eng.*, 7(4):566–582, 1995.

[5] Jan Chomicki and David Toman. Temporal logic in information systems. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, pages 31–70. Kluwer, 1998.

[6] Jan Chomicki and David Toman. Temporal databases. In Michael Fisher, Dov M. Gabbay, and Lluis Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier Science, 2005.

[7] José Mauro Volkmer de Castilho, Marco A. Casanova, and Antonio L. Furtado. A temporal framework for database specifications. In *VLDB*, pages 280–291. Morgan Kaufmann, 1982.

[8] Wei Li, Richard T. Snodgrass, Shiyan Deng, Vineel Kumar Gattu, and Aravindan Kasthurirangan. Efficient sequenced integrity constraint checking. In *ICDE*, pages 131–140. IEEE Computer Society, 2001.

[9] Udo W. Lipeck and Gunter Saake. Monitoring dynamic integrity constraints based on temporal logic. *Inf. Syst.*, 12(3):255–269, 1987.

[10] Dimitris Plexousakis. Integrity constraint and rule maintenance in temporal deductive knowledge bases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *VLDB*, pages 146–157. Morgan Kaufmann, 1993.

[11] A. Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active database systems. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 269–280. ACM Press, 1995.

[12] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *IEEE Trans. Knowl. Data Eng.*, 7(3):471–486, 1995.

# TEMPORAL JOINS

Dengfeng Gao
IBM Silicon Valley Lab
dgao@us.ibm.com

## SYNONYMS

None.

## DEFINITION

A temporal join is a join operation on two temporal relations, in which each tuple has additional attributes indicating a time interval. The temporal join predicates include conventional join predicates as well as a temporal constraint that requires the overlap of the intervals of the two joined tuples. The result of a temporal join is a temporal relation.

Besides binary temporal joins that operate on two temporal relations, there are n-ary temporal joins that operate on more than two temporal relations. Besides temporal overlapping, there are other temporal conditions such as "before" and "after" [1]. This article will concentrate on the binary temporal joins with overlapping temporal condition since most of the previous work has focused on this kind of joins.

## HISTORICAL BACKGROUND

In the past, temporal join operators have been defined in different temporal data models; at times the essentially same operators have even been given different names when defined in different data models. Further, the existing join algorithms have also been constructed within the contexts of different data models. Temporal join operators were first defined by Clifford and Croker [2]. Later many papers studied more temporal join operators and the evaluation algorithms. To enable the comparison of join definitions and implementations across data models, Gao et al. [6] proposed a taxonomy of temporal joins and then use this taxonomy to classify all previously defined temporal joins.

## SCIENTIFIC FUNDAMENTALS

Starting from the core set of conventional relational joins that have long been accepted as "standard" [11]: Cartesian product (whose "join predicate" is the constant expression TRUE), theta-join, equijoin, natural join, left and right outerjoin, and full outerjoin, a temporal counterpart that is a natural, temporal generalization of the set can be defined. The semantics of the temporal join operators are defined as follows.

To be specific, the definitions are based on a single data model that is used most widely in temporal data management implementations, namely the one that timestamps each tuple with an interval. Assume that the time-line is partitioned into minimal-duration intervals, termed *chronons* [5]. The intervals are denoted by inclusive starting and ending chronons.

Two temporal relational schemas, $R$ and $S$, are defined as follows.

$$\begin{aligned} R &= (A_1, \ldots, A_n, \mathrm{T}_s, \mathrm{T}_e) \\ S &= (B_1, \ldots, B_m, \mathrm{T}_s, \mathrm{T}_e) \end{aligned}$$

The $A_i$, $1 \leq i \leq n$, and $B_i$, $1 \leq i \leq m$, are the *explicit attributes* that are found in corresponding snapshot schemas, and $\mathrm{T}_s$ and $\mathrm{T}_e$ are the timestamp start and end attributes, recording when the information recorded by the explicit attributes holds (or held or will hold) true. T will be used as a shorthand for the interval $[\mathrm{T}_s, \mathrm{T}_e]$, and

$A$ and $B$ will be used as a shorthand for $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_n\}$, respectively. Also, $r$ and $s$ are defined to be instances of $R$ and $S$, respectively.
Consider the following two temporal relations. The relations show the canonical example of employees, the departments they work for, and the managers who supervise those departments.

Employee

| EmpName | Dept | T |
|---|---|---|
| Ron | Ship | [1,5] |
| George | Ship | [5,9] |
| Ron | Mail | [6,10] |

Manages

| Dept | MgrName | T |
|---|---|---|
| Load | Ed | [3,8] |
| Ship | Jim | [7,15] |

Tuples in the relations represent facts about the modeled reality. For example, the first tuple in the Employee relation represents the fact that Ron worked for the Shipping department from time 1 to time 5, inclusive. Notice that none of the attributes, including the timestamp attributes T, are set-valued—the relation schemas are in 1NF.

**Cartesian Product**
The temporal Cartesian product is a conventional Cartesian product with a predicate on the timestamp attributes. To define it, two auxiliary definitions are needed.
First, $intersect(U, V)$, where $U$ and $V$ are intervals, returns TRUE if there exists a chronon $t$ such that $t \in U \ \wedge \ t \in V$, and FALSE otherwise. Second, $overlap(U, V)$ returns the maximum interval contained in its two argument intervals. If no non-empty intervals exist, the function returns $\emptyset$. To state this more precisely, let *first* and *last* return the smallest and largest of two argument chronons, respectively. Also let $U_s$ and $U_e$ denote the starting and ending chronons of $U$, and similarly for $V$.

$$overlap(U, V) = \begin{cases} [last(U_s, V_s), first(U_e, V_e)] & \text{if } last(U_s, V_s) \leq first(U_e, V_e) \\ \emptyset & \text{otherwise} \end{cases}$$

The temporal Cartesian product, $r \times^{\mathrm{T}} s$, of two temporal relations $r$ and $s$ is defined as follows.

$$r \times^{\mathrm{T}} s = \{z^{(n+m+2)} \mid \exists x \in r \, \exists y \in s(intersect(x[\mathrm{T}], y[\mathrm{T}]) \wedge z[A] = x[A] \wedge z[B] = y[B] \wedge z[\mathrm{T}] = overlap(x[\mathrm{T}], y[\mathrm{T}]) \wedge z[\mathrm{T}] \neq \emptyset)\}$$

The first line of the definition ensures that matching tuples $x$ and $y$ have overlapping timestamps and sets the explicit attribute values of the result tuple $z$ to the concatenation of the explicit attribute values of $x$ and $y$. The second line computes the timestamp of $z$ and ensures that it is non-empty. The *intersect* predicate is included only for later reference—it may be omitted without changing the meaning of the definition.
Consider the query "Show the names of employees and managers where the employee worked for the company while the manager managed some department in the company." This can be satisfied using the temporal Cartesian product.

Employee $\times^{\mathrm{T}}$ Manages

| EmpName | Dept | Dept | MgrName | T |
|---|---|---|---|---|
| Ron | Ship | Load | Ed | [3,5] |
| George | Ship | Load | Ed | [5,8] |
| George | Ship | Ship | Jim | [7,9] |
| Ron | Mail | Load | Ed | [6,8] |
| Ron | Mail | Ship | Jim | [7,10] |

The *overlap* function is necessary and sufficient to ensure *snapshot reducibility*, as will be discussed in detail later. Basically, the temporal Cartesian product acts as though it is a conventional Cartesian product applied independently at each point in time. When operating on interval-stamped data, this semantics corresponds to an intersection: the result will be valid during those times when contributing tuples from *both* input relations are valid.

**Theta-Join**
Like the conventional theta-join, the temporal theta-join supports an unrestricted predicate $P$ on the explicit attributes of its input arguments. The temporal theta-join, $r \bowtie^{\mathrm{T}}_P s$, of two relations $r$ and $s$ selects those tuples from $r \times^{\mathrm{T}} s$ that satisfy predicate $P(r[A], s[B])$. Let $\sigma$ denote the standard selection operator.

The temporal theta-join, $r \bowtie^{\mathrm{T}}_{P} s$, of two temporal relations $r$ and $s$ is defined as follows.

$$r \bowtie^{\mathrm{T}}_{P} s = \sigma_{P(r[A],s[B])}(r \times^{\mathrm{T}} s)$$

**Equijoin**

Like snapshot equijoin, the temporal equijoin operator enforces equality matching between specified subsets of the explicit attributes of the input relations.

The temporal equijoin on two temporal relations $r$ and $s$ on attributes $A' \subseteq A$ and $B' \subseteq B$ is defined as the theta-join with predicate $P \equiv r[A'] = s[B']$:

$$r \bowtie^{\mathrm{T}}_{r[A']=s[B']} s \ .$$

**Natural Join**

The temporal natural join bears the same relationship to the temporal equijoin as does their snapshot counterparts. Namely, the temporal natural join is simply a temporal equijoin on identically named explicit attributes, followed by a subsequent projection operation.

To define this join, the relation schemas are augmented with explicit join attributes, $C_i$, $1 \leq i \leq k$, which are abbreviated by $C$.

$$\begin{aligned} R &= (A_1, \ldots, A_n, C_1, \ldots, C_k, \mathrm{T}_s, \mathrm{T}_e) \\ S &= (B_1, \ldots, B_m, C_1, \ldots, C_k, \mathrm{T}_s, \mathrm{T}_e) \end{aligned}$$

The temporal natural join of $r$ and $s$, $r \bowtie^{\mathrm{T}} s$, is defined as follows.

$$\begin{aligned} r \bowtie^{\mathrm{T}} s = \{z^{(n+m+k+2)} \mid \exists x \in r\, \exists y \in s (&x[C] = y[C] \wedge \\ &z[A] = x[A] \wedge z[B] = x[B] \wedge z[C] = y[C] \wedge \\ &z[\mathrm{T}] = \mathit{overlap}(x[\mathrm{T}], y[\mathrm{T}]) \wedge z[\mathrm{T}] \neq \emptyset)\} \end{aligned}$$

The first two lines ensure that tuples $x$ and $y$ agree on the values of the join attributes $C$ and set the explicit attribute of the result tuple $z$ to the concatenation of the non-join attributes $A$ and $B$ and a single copy of the join attributes, $C$. The third line computes the timestamp of $z$ as the overlap of the timestamps of $x$ and $y$, and ensures that $x[\mathrm{T}]$ and $y[\mathrm{T}]$ actually overlap.

The temporal natural join plays the same important role in reconstructing normalized temporal relations as does the snapshot natural join for normalized snapshot relations [9]. Most previous work in temporal join evaluation has addressed, either implicitly or explicitly, the implementation of the temporal natural join (or the closely related temporal equijoin).

**Outerjoins and Outer Cartesian Products**

Like the snapshot outerjoin, temporal outerjoins and Cartesian products retain *dangling tuples*, i.e., tuples that do not participate in the join. However, in a temporal database, a tuple may dangle over a portion of its time interval and be covered over others; this situation must be accounted for in a temporal outerjoin or Cartesian product.

The temporal outerjoin may be defined as the union of two subjoins, analogous to the snapshot outerjoin. The two subjoins are the temporal left outerjoin and the temporal right outerjoin. As the left and right outerjoins are symmetric, only the left outerjoin is defined here.

Two auxiliary functions are needed. The *coalesce* function collapses value-equivalent tuples—tuples with mutually equal non-timestamp attribute values [10]—in a temporal relation into a single tuple with the same non-timestamp attribute values and a timestamp that is the finite union of intervals that precisely contains the chronons in the timestamps of the value-equivalent tuples. (Finite unions of time intervals are termed *temporal elements* [7].) The definition of *coalesce* uses the function *chronons* that returns the set of chronons contained in the argument interval.

$$\begin{aligned} \mathit{coalesce}(r) = \{z^{(n+2)} \mid \exists x \in r (&z[A] = x[A] \Rightarrow \mathit{chronons}(x[\mathrm{T}]) \subseteq z[\mathrm{T}] \wedge \\ &\forall x' \in r (x[A] = x'[A] \Rightarrow (\mathit{chronons}(x'[\mathrm{T}]) \subseteq z[\mathrm{T}]))) \wedge \\ &\forall t \in z[\mathrm{T}]\, \exists x'' \in r (z[A] = x''[A] \wedge t \in \mathit{chronons}(x''[\mathrm{T}]))\} \end{aligned}$$

The first two lines of the definition coalesce all value-equivalent tuples in relation $r$. The third line ensures that no spurious chronons are generated.

Now a function *expand* is defined that returns the set of maximal intervals contained in an argument temporal element, $T$. Prior to defining *expand* an auxiliary function *intervals* is defined that returns the set of intervals contained in an argument temporal element.

$$intervals(T) = \{[t_s, t_e] \mid t_s \in T \wedge t_e \in T \wedge \forall t \in chronons([t_s, t_e])(t \in T)\}$$

The first two conditions ensures that the beginning and ending chronons of the interval are elements of $T$. The third condition ensures that the interval is contiguous within $T$.

Using *intervals*, *expand* is defined as follows.

$$expand(T) = \{[t_s, t_e] \mid [t_s, t_e] \in intervals(T) \wedge \neg\exists [t'_s, t'_e] \in intervals(T)(chronons([t_s, t_e]) \subset chronons([t'_s, t'_e]))\}$$

The first line ensures that a member of the result is an interval contained in $T$. The second line ensures that the interval is indeed maximal.

The temporal left outerjoin is now ready to be defined. Let $R$ and $S$ be defined as for the temporal equijoin. $A' \subseteq A$ and $B' \subseteq B$ are used as the explicit join attributes.

The temporal left outerjoin, $r \overset{\mathrm{T}}{\bowtie}_{r[A']=s[B']} s$ of two temporal relations $r$ and $s$ is defined as follows.

$$\begin{aligned} r \overset{\mathrm{T}}{\bowtie}_{r[A']=s[B']} s = \{z^{(n+m+2)} \mid\ & \exists x \in coalesce(r)\ \exists y \in coalesce(s) \\ & (x[A'] = y[B'] \wedge z[A] = x[A] \wedge z[\mathrm{T}] \neq \emptyset \wedge \\ & \quad ((z[B] = y[B] \wedge z[\mathrm{T}] \in \{expand(x[\mathrm{T}] \cap y[\mathrm{T}])\}) \vee \\ & \quad\ (z[B] = \text{null} \wedge z[\mathrm{T}] \in \{expand(x[\mathrm{T}]) - expand(y[\mathrm{T}])\}))) \vee \\ & \exists x \in coalesce(r)\ \forall y \in coalesce(s) \\ & (x[A'] \neq y[B'] \Rightarrow z[A] = x[A] \wedge z[B] = \text{null} \wedge \\ & \quad z[\mathrm{T}] \in expand(x[T]) \wedge z[\mathrm{T}] \neq \emptyset)\} \end{aligned}$$

The first four lines of the definition handle the case where, for a tuple $x$ deriving from the left argument, a tuple $y$ with matching explicit join attribute values is found. For those time intervals of $x$ that are not shared with $y$, tuples with null values in the attributes of $y$ are generated. The final three lines of the definition handle the case where no matching tuple $y$ is found. Tuples with null values in the attributes of $y$ are generated.

The temporal outerjoin may be defined as simply the union of the temporal left and the temporal right outerjoins (the union operator eliminates the duplicate equijoin tuples). Similarly, a temporal outer Cartesian product is a temporal outerjoin without the equijoin condition ($A' = B' = \emptyset$).

Table 1 summarizes how previous work is represented in the taxonomy. For each operator defined in previous work, the table lists the defining publication, researchers, the corresponding taxonomy operator, and any restrictions assumed by the original operators. In early work, Clifford [2] indicated that a an `INTERSECTION-JOIN` should

Table 1: Temporal Join Operators

| Operator | Initial Citation | Taxonomy Operator | Restrictions |
|---|---|---|---|
| $\Theta$-JOIN | [3] | Theta-join | None |
| EQUIJOIN | [3] | Equijoin | None |
| NATURAL-JOIN | [3] | Natural Join | None |
| TIME-JOIN | [3] | Cartesian Product | 1 |
| T-join | [8] | Cartesian Product | None |
| Cartesian product | [4] | Outer Cartesian Product | None |
| TE-JOIN | [13] | Equijoin | 2 |
| TE-OUTERJOIN | [13] | Left Outerjoin | 2 |
| EVENT-JOIN | [13] | Outerjoin | 2 |
| Valid-Time Theta-Join | [14] | Theta-join | None |
| Valid-Time Left Join | [14] | Left Outerjoin | None |
| GTE-Join | [15] | Equijoin | 2, 3 |

*Restrictions:*
1 = restricts also the valid time of the result tuples
2 = matching only on surrogate attributes
3 = includes also intersection predicates with an argument surrogate range and a time range

be defined that represents the categorized non-outer joins and Cartesian products, and he proposed that a `UNION-JOIN` be defined for the outer variants.

**Reducibility**
The following paragraphs show how the temporal operators reduce to snapshot operators. Reducibility guarantees that the semantics of snapshot operator is preserved in its more complex, temporal counterpart.
For example, the semantics of the temporal natural join reduces to the semantics of the snapshot natural join in that the result of first joining two temporal relations and then transforming the result to a snapshot relation yields a result that is the same as that obtained by first transforming the arguments to snapshot relations and then joining the snapshot relations. This commutativity diagram is shown in Figure 1 and stated formally in the first equality of the following theorem.

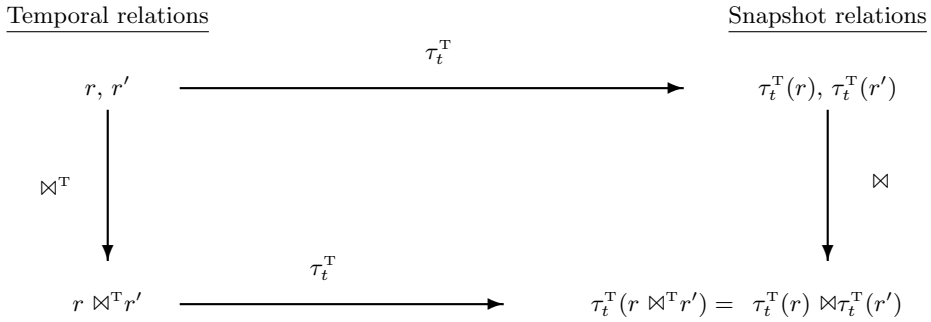



Figure 1: Reducibility of Temporal Natural Join to Snapshot Natural Join

The timeslice operation $\tau^{\mathrm{T}}$ takes a temporal relation $r$ as argument and a chronon $t$ as parameter. It returns the corresponding snapshot relation, i.e., with the schema of $r$, but without the timestamp attributes, that contains (the non-timestamp portion of) all tuples $x$ from $r$ for which $t$ belongs to $x[\mathrm{T}]$. It follows from the next theorem that the temporal joins defined here reduce to their snapshot counterparts.

**Theorem 1** Let $t$ denote a chronon and let $r$ and $s$ be relation instances of the proper types for the operators they are applied to. Then the following hold for all $t$:

$$\begin{aligned}
\tau_t^{\mathrm{T}}(r \bowtie^{\mathrm{T}} s) &= \tau_t^{\mathrm{T}}(r) \bowtie \tau_t^{\mathrm{T}}(s) \\
\tau_t^{\mathrm{T}}(r \times^{\mathrm{T}} s) &= \tau_t^{\mathrm{T}}(r) \times \tau_t^{\mathrm{T}}(s) \\
\tau_t^{\mathrm{T}}(r \bowtie_P^{\mathrm{T}} s) &= \tau_t^{\mathrm{T}}(r) \bowtie_P \tau_t^{\mathrm{T}}(s) \\
\tau_t^{\mathrm{T}}(r \rtimes^{\mathrm{T}} s) &= \tau_t^{\mathrm{T}}(r) \rtimes \tau_t^{\mathrm{T}}(s) \\
\tau_t^{\mathrm{T}}(r \ltimes^{\mathrm{T}} s) &= \tau_t^{\mathrm{T}}(r) \ltimes \tau_t^{\mathrm{T}}(s)
\end{aligned}$$

Due to the space limit, the proof of this theorem is not provided here. The details can be found in the related paper [6].

**Evaluation Algorithms**
Algorithms for temporal join evaluation are necessarily more complex than their snapshot counterparts. Whereas snapshot evaluation algorithms match input tuples on their explicit join attributes, temporal join evaluation algorithms typically must in addition ensure that temporal restrictions are met. Furthermore, this problem is exacerbated in two ways. Timestamps are typically complex data types, e.g., intervals, requiring inequality predicates, which conventional query processors are not optimized to handle. Also, a temporal database is usually larger than a corresponding snapshot database due to the versioning of tuples.
There are two categories of evaluation algorithms. Index-based algorithms use an auxiliary access path, i.e., a data structure that identifies tuples or their locations using a join-attribute value. Non-index-based algorithms do not employ auxiliary access paths. The large number of temporal indexes have been proposed in the literature [12]. Gao et al. [6] provided a taxonomy of non-index-based temporal join algorithms.

## KEY APPLICATIONS

Temporal joins are used to model relationships between temporal relations with respect to the temporal dimensions. Data warehouses usually need to store and analyze historical data. Temporal joins can be used (alone or together with other temporal relational operators) to perform the analysis on historical data.

## CROSS REFERENCE

Temporal Database, Temporal Data Models, Temporal Algebras, Temporal Indexing, Temporal Query Processing.

## RECOMMENDED READING


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

[2] J. Clifford and A. Uz Tansel. On An Algebra For Historical Relational Databases: Two Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1985.

[3] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, Los Angeles, CA, February 1987. IEEE Computer Society, IEEE Computer Society Press.

[4] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) Revisited. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 1, pages 6–27. Benjamin/Cummings Publishing Company, 1993.

[5] C. E. Dyreson and R. T. Snodgrass. Timestamp Semantics and Representation. *Information Systems*, 18(3):143–166, 1993.

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

[7] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM TODS* 13(4): 418-448, 1988.

[8] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the IEEE Conference on Data Engineering*, Kobe, Japan, 1991.

[9] C. S. Jensen, R. T. Snodgrass and M. D. Soo. Extending Existing Dependency Theory to Temporal Databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):563–582, August, 1996.

[10] C. S. Jensen (editor). The Consensus Glossary of Temporal Database Concepts—February 1998 Version. In Temporal Databases: Research and Practice. Springer, LNCS 1399, Berlin, 1998, pages 367-405.

[11] P. Mishra and M. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[12] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys* 31(2): 158-221, June 1999.

[13] A. Segev and H. Gunadhi. Event-join Optimization in Temporal Relational Databases. In *Proceedings of the Conference on Very Large Databases*, pages 205–215, August 1989.

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

[15] D. Zhang, V. J. Tsotras and B. Seeger. Efficient Temporal Join Processing Using Indices. In *Proceedings of the International Conference on Data Engineering*, February 2002.

# Temporal Logic in Database Query Languages

Jan Chomicki, University at Buffalo, USA, http://www.cse.buffalo.edu/~chomicki
David Toman, University of Waterloo, Canada, http://www.cs.uwaterloo.ca/~david

## SYNONYMS

None.

## DEFINITION

The term "temporal logic" is used, in the area of formal logic, to describe systems for representing and reasoning about propositions and predicates whose truth depends on time. These systems are developed around a set of *temporal connectives*, such as *sometime in the future* or *until*, that provide implicit references to time instants. First-order temporal logic is a variant of temporal logic that allows first-order predicate (relational) symbols, variables and quantifiers, in addition to temporal connectives. This logic can be used as a natural *temporal query language* for point-stamped temporal databases. A query (a temporal logic formula) is evaluated with respect to an *evaluation point (time instant)*. Each such point determines a specific database snapshot that can be viewed as a relational database. Thus, the evaluation of temporal logic queries resembles the evaluation of first-order (relational calculus) queries equipped with an additional capability to "move" the evaluation point using *temporal connectives*. In this way, it becomes possible to refer in a single query to multiple snapshots of a given temporal database. The answer to a temporal logic query evaluated with respect to all time instants forms a point-stamped temporal relation.

## HISTORICAL BACKGROUND

Temporal Logic was developed, originally under the name of *tense logic*, by Arthur Prior in the late 1950s for representing and reasoning about natural languages. It was introduced to computer science by Amir Pnueli [8] as a tool for formal verification of software systems. The first proposal for using a temporal logic in the database context was by Sernadas [9]. Subsequently, de Castilho *et al.* [3] initiated the study of temporal-logic integrity constraints in relational databases. Tuzhilin and Clifford [13] considered temporal logic as a query language for temporal databases.

## SCIENTIFIC FUNDAMENTALS

*Temporal Logic* is a variant of *modal* logic, tailored to expressing statements whose truth is relative to an underlying time domain which is commonly a *linearly ordered* set of time instants (see the entry Time Domain). The modalities are expressed using natural-language statements of the form *sometime in the future*, *always in the future*, etc., and are captured in the syntax of the logic using *temporal connectives*.

Temporal logics are usually rooted in *propositional logic*. However, for the purposes of querying (single-dimensional, valid time) point-stamped temporal databases, *Linear-Time First-Order Temporal Logic (FOTL)*, an extension of first-order logic (relational calculus) with *temporal connectives*, is used. More formally, given a relational schema $\rho$ (of a snapshot of a point-stamped temporal database), the syntax of FOTL queries is defined as follows:

$$Q ::= r(x_{i_1}, \ldots, x_{i_k}) \mid x_i = x_j \mid Q \wedge Q \mid \neg Q \mid \exists x.Q \mid Q \textbf{ since } Q \mid Q \textbf{ until } Q$$

for $r \in \rho$. The additional **since** and **until** connectives are the *temporal connectives*. The connectives completely *encapsulate* the structure of time: FOTL only refers to time *implicitly* using these connectives. In contrast, Temporal Relational Calculus (TRC) uses *explicit* temporal variables and attributes to refer to time. Note that there are no restrictions on the nesting of the temporal connectives, the Boolean connectives, and the quantifiers.

The meaning of all temporal logic formulas is defined relative to a particular time instant called the *evaluation point*. Intuitively, the evaluation point can be viewed as representing the *current instant* or *now*. The standard

first-order parts of FOTL formulas are then evaluated in the snapshot of the temporal database determined by the evaluation point. The temporal connectives make it possible to change the evaluation point, i.e., to "move" it to the past or to the future. In this way the combination of first-order constructs and temporal connectives allows to ask queries that refer to multiple snapshots of the temporal database. For example the query $Q_1$ **since** $Q_2$ asks for all answers that make $Q_2$ true sometime in the past and $Q_1$ true between then and now. Similarly, queries about the future are formulated using the **until** connective. More formally, answers to FOTL queries are defined using a *satisfaction relation* that, for a given FOTL query, links a temporal database and an evaluation point (time instant) with the valuations that make the given query true with respect to the snapshot of that database at that particular evaluation point. This definition, given below, extends the standard definition of satisfaction for first-order logic.

**Definition**. [FOTL Semantics] Let $DB$ be a point-stamped temporal database with a data domain D, a point-based time domain $\mathrm{T}_P$, and a (snapshot) schema $\rho$.

The *satisfaction relation* $DB, \theta, t \models Q$, where $Q$ is an FOTL formula, $\theta$ a valuation, and $t \in \mathrm{T}_P$, is defined inductively with respect to the structure of the formula $Q$:

$$
\begin{array}{ll}
DB, \theta, t \models r_j(x_{i_1}, \ldots, x_{i_k}) & \text{if } (\theta(x_{i_1}), \ldots, \theta(x_{i_k})) \in \mathbf{r}_j^{DB(t)} \\
DB, \theta, t \models x_i = x_j & \text{if } \theta(x_i) = \theta(x_j) \\
DB, \theta, t \models Q_1 \wedge Q_2 & \text{if } DB, \theta, t \models Q_1 \text{ and } DB, \theta, t \models Q_2 \\
DB, \theta, t \models \neg Q_1 & \text{if not } DB, \theta, t \models Q_1 \\
DB, \theta, t \models \exists x_i . Q_1 & \text{if there is } a \in D \text{ such that } DB, \theta[x_i \mapsto a], t \models Q_1 \\
DB, \theta, t \models Q_1 \textbf{ since } Q_2 & \text{if } \exists t_2 . t_2 < t \text{ and } DB, \theta, t_2 \models Q_2 \text{ and } (\forall t_1 . t_2 < t_1 < t \text{ implies } DB, \theta, t_1 \models Q_1) \\
DB, \theta, t \models Q_1 \textbf{ until } Q_2 & \text{if } \exists t_2 . t_2 > t \text{ and } DB, \theta, t_2 \models Q_2 \text{ and } (\forall t_1 . t_2 > t_1 > t \text{ implies } DB, \theta, t_1 \models Q_1)
\end{array}
$$

where $\mathbf{r}_j^{DB(t)}$ is the instance of the relation $r_j$ in the snapshot $DB(t)$ of the database $DB$ at the instant $t$.

*The answer to an FOTL query $Q$ over $DB$* is the set of tuples:

$$Q(DB) := \{(t, \theta(x_1), \ldots, \theta(x_k)) : DB, \theta, t \models Q\},$$

where $x_1, \ldots, x_k$ are the free variables of $Q$.

Note that answers to FOTL queries are (valid-time) point-stamped temporal relations.

Other commonly used temporal connectives, such as $\Diamond$ (*sometime in the future*), $\Box$ (*always in the future*), $\blacklozenge$ (*sometime in the past*), and $\blacksquare$ (*always in the past*) can be defined in terms of **since** and **until** as follows:

$$
\begin{array}{ll}
\Diamond X_1 := \text{true } \textbf{until } X_1 & \blacklozenge X_1 := \text{true } \textbf{since } X_1 \\
\Box X_1 := \neg \Diamond \neg X_1 & \blacksquare X_1 := \neg \blacklozenge \neg X_1
\end{array}
$$

For a discrete linear order, the $\bigcirc$ (*next*) and $\bullet$ (*previous*) operators are defined as follows:

$$\bigcirc X_1 := \text{false } \textbf{until } X_1 \qquad \bullet X_1 := \text{false } \textbf{since } X_1$$

The connectives **since**, $\blacklozenge$, $\blacksquare$, and $\bullet$ are called *past temporal connectives* (as they refer to the past) and **until**, $\Diamond$, $\Box$, and $\bigcirc$ *future temporal connectives*.

**Example**. The sensor information about who enters or exits a room is kept in the relations *Entry* (Figure 1a) and *Exit* (Figure 1b). Consider the query $Q_a$*: "For every time instant, who is in the room Bell 224 at that instant?"* It can be written in temporal logic as:

$$\exists r.\ ((\neg Exit(r, p)) \textbf{ since } Entry(r, p)) \wedge r = \text{''Bell 224''}$$

The answer to $Q_a$ in the given database is presented in Figure 1c, under the assumption that time instants correspond to minutes.

a)

| *Entry* | | |
|---|---|---|
| *Time* | *Room* | *Person* |
| 8 : 30 | Bell 224 | John |
| 9 : 00 | Bell 224 | Mary |
| 11 : 00 | Capen 10 | John |

b)

| *Exit* | | |
|---|---|---|
| *Time* | *Room* | *Person* |
| 10 : 30 | Bell 224 | John |
| 10 : 00 | Bell 224 | Mary |
| 12 : 00 | Capen 10 | John |

c)

| *Time* | *Person* |
|---|---|
| 8 : 31 | John |
| ... | ... |
| 10 : 30 | John |
| 9 : 01 | Mary |
| ... | ... |
| 10 : 00 | Mary |

Figure 1: The *Entry* relation, the *Exit* relation, and the *Who is in Bell 224* relation.

## Extensions

Several natural extensions of FOTL are obtained by modifying various components of the language:

**Multiple Temporal Contexts (Multi-dimensional TLs).** Standard temporal logics use a single *evaluation point* to relativize the truth of formulas with respect to time. However, there is no principled reason why not to use more than one evaluation point simultaneously. The logics taking this path are called *multidimensional temporal logics* or, more precisely, *n-dimensional* temporal logics (for $n \geq 1$). The satisfaction relation is extended, for a particular $n$, in a natural way, as follows:

$$DB, \theta, t_1, \ldots, t_n \models Q$$

where $t_1, \ldots, t_n$ are the evaluation points. In a similar fashion the definitions of *temporal connectives* are extended to this setting. Two-dimensional connectives, for example, seem to be the natural basis for temporal logic-style query language for the bitemporal data model. Unfortunately, there is no consensus on the selection of two-dimensional temporal operators.

An interesting variant of this extension are *interval temporal logics* that associate truth with *intervals*—these, however, can be considered *pairs* of time points [5, 14].

**More Complex Time Domains.** While most temporal logics assume that the time domain is equipped with a linear order only, in many practical settings the time domain has additional structure. For example, there may be a way to refer to *duration* (the distance of two time points). The linear-order temporal connectives are then generalized to *metric temporal connectives*:

$$DB, \theta, t \models Q_1 \ \mathbf{since}_{\sim m} \ Q_2 \text{ if } \exists t_2.t - t_2 \sim m \text{ and } DB, \theta, t_2 \models Q_2 \text{ and } (\forall t_1.t_2 < t_1 < t \text{ implies } DB, \theta, t_1 \models Q_1)$$
$$DB, \theta, t \models Q_1 \ \mathbf{until}_{\sim m} \ Q_2 \text{ if } \exists t_2.t_2 - t \sim m \text{ and } DB, \theta, t_2 \models Q_2 \text{ and } (\forall t_1.t_2 > t_1 > t \text{ implies } DB, \theta, t_1 \models Q_1)$$

for $\sim \in \{<, \leq, =, \geq, >\}$. Intuitively, these connectives provide means of placing constraints on how far in the past/future certain subformulas must be true. The resulting logic is then the *Metric First-order Temporal Logic*, a first-order variant of *Metric Temporal Logic* [7].

**Example**. To demonstrate the expressive power of Metric Temporal Logic, consider the query $Q_b$*: "For every time instant, who has been in the room Bell 224 at that instant for at least 2 hours?"*:

$$\exists r.\ ((\neg Exit(r, p))\ \mathbf{since}_{\geq 2:00}\ Entry(r, p)) \wedge r = \text{"Bell 224"}$$

**More Powerful Languages for Defining Temporal Connectives.** Another extension introduces a more powerful *language* for specifying temporal connectives over the underlying linearly-ordered time domain. Vardi and Wolper show that temporal logics with connectives defined using first-order formulas cannot express various natural conditions such as "every other day". To remedy this shortcoming, they propose temporal connectives defined with the help of *regular expressions* (ETL [16]) or *fixpoints* (temporal $\mu$-calculus [15]). Such extensions carry over straightforwardly to the first-order setting.

**More Complex Underlying Query Languages.** Last, instead of relational calculus, temporal connectives can be added to a more powerful language, such as Datalog. The resulting language is called *Templog* [2]. With suitable restrictions, query evaluation in this language is decidable and the language itself is equivalent to $\text{Datalog}_{1S}$ [2].

## Expressive Power

The **since** and **until** temporal connectives can be equivalently defined using *formulas* in the underlying theory of linear order as follows:

$$X_1 \textbf{ since } X_2 \;:=\; \exists t_2.t_0 > t_2 \wedge X_2 \wedge \forall t_1(t_0 > t_1 > t_2 \rightarrow X_1)$$
$$X_1 \textbf{ until } X_2 \;:=\; \exists t_2.t_0 < t_2 \wedge X_2 \wedge \forall t_1(t_0 < t_1 < t_2 \rightarrow X_1)$$

where $X_1$ and $X_2$ are *placeholders* that will be substituted with other formulas to be evaluated at the time instants $t_1$ and $t_2$, respectively. This observation indicates that every FOTL query can be equivalently expressed in TRC. The explicit translation parallels the inductive definition of FOTL satisfaction, uniformly parameterizing the formulas by $t_0$. In this way, an atomic formula $r(x_1, \ldots, x_k)$ (where $r$ is a non-temporal snapshot relation) becomes $R(t_0, x_1, \ldots, x_k)$ (where $R$ is a point-timestamped relation), and so on. For a particular $t_0$, evaluating $r(x_1, \ldots, x_k)$ in $DB(t_0)$, the snapshot of the database $DB$ at $t_0$, yields exactly the same valuations as evaluating $R(t_0, x_1, \ldots, x_k)$ in $DB$. The embedding of the temporal connectives uses the definitions above. For example, the embedding of the **since** connective looks as follows:

$$\begin{aligned}\text{Embed}(Q_1 \textbf{ since } Q_2) \;=\; & \exists t_2(t_0 > t_2 \wedge \forall t_0(t_2 = t_0 \rightarrow \text{Embed}(Q_2)) \wedge \\ & \forall t_1(t_0 > t_1 > t_2 \rightarrow \forall t_0(t_1 = t_0 \rightarrow \text{Embed}(Q_1))))\end{aligned}$$

Note that $t_0$ is the only free variable in Embed($Q_1$), Embed($Q_2$), and Embed($Q_1$ **since** $Q_2$). Thus, in addition to applying the (first-order) definition of the temporal connective, the embedding performs an additional *renaming* of the temporal variable denoting the evaluation point for the subformulas, because the only free variable outside of the expanded temporal connectives must be called $t_0$.

Additional temporal connectives can be defined using the same approach, as formulas in the underlying theory of the temporal domain. However, for linear orders—the most common choice for such a theory—Kamp [6] has shown that all the connectives that are first-order definable in terms of linear order can be readily formulated using **since** and **until**.

In addition, it is easy to see on the basis of the above embedding that all FOTL queries (and their subqueries) define point-stamped temporal relations. This *closure property* makes FOTL amenable to specifying operators for temporal relational algebra(s) over the point-stamped temporal model. On the other hand, many, if not most, other temporal query languages, in particular various temporal extensions of SQL, are based on TRC and use temporal variables and attributes to explicitly access to timestamps. These languages do not share the above closure property. Surprisingly, and in contrast to the propositional setting, one can prove that query languages based on FOTL are strictly weaker than query languages based on TRC [1, 12]. The query

SNAPSHOT EQUALITY:
"are there two distinct time instants at which a unary relation $R$ contains exactly the same values?"

cannot be expressed in FOTL. On the other hand, SNAPSHOT EQUALITY can be easily expressed in TRC as follows:

$$\exists t_1, t_2.t_1 < t_2 \wedge \forall x.R(t_1, x) \iff R(t_2, x)$$

Intuitively, the subformula "$\forall x.R(t_1, x) \iff R(t_2, x)$" in the above query requires the simultaneous use of *two* distinct evaluation points $t_1$ and $t_2$ in the scope of a universal quantifier, which is not possible in FOTL. The result can be rephrased by saying that FOTL and other temporal query languages closed over the point-stamped temporal relations fail to achieve (the temporal variant of) Codd's completeness. In particular, there cannot be a temporal relational algebra over the (single-dimensional) point-stamped temporal model that can express all TRC queries.

In addition, this weakness is inherent to other languages with *implicit access* to timestamps, provided the underlying time domain is a linear order. In particular:

- adding a finite number of additional temporal connectives defined in the theory of linear order (including constants) is not sufficient for expressing SNAPSHOT EQUALITY [12];
- introducing *multidimensional temporal connectives* [4], while sufficient to express SNAPSHOT EQUALITY, is still not sufficient to express all TRC queries [11]. This also means that in the bitemporal model, the associated query languages cannot simultaneously preserve closure with respect to bitemporal relations and be expressively equivalent to TRC;

- using temporal connectives that are defined by fixpoints and/or regular expressions (see the earlier discussion) is also insufficient to express SNAPSHOT EQUALITY. Due to their non-first-order nature, the resulting query language(s) are incomparable, in terms of their expressive power, to TRC [1].

The only currently known way of achieving first-order completeness is based on using temporal connectives defined over a time domain whose structure allows the definition of pairing and projections (e.g., integer arithmetic). In this way temporal connectives can use pairing to simulate an unbounded number of variables and in turn the full TRC. However, such a solution is not very appealing, as the timestamps in the intermediate temporal relations do not represent time instants, but rather (encoded) tuples of such instants.

## KEY APPLICATIONS

The main application area of FOTL is in the area of temporal integrity constraints. It is based on the observation that a sequence of relational database states resulting from updates may be viewed as a snapshot temporal database and constrained using Boolean FOTL formulas. Being able to refer to the past states of the database, temporal logic constraints generalize dynamic constraints. Temporal logic is also influential in the area of design and analysis of temporal query languages such as temporal relational algebras.

## CROSS REFERENCE

bitemporal data model, Datalog, point-stamped data model, relational calculus, temporal integrity constraints, temporal query languages, temporal relational calculus, temporal algebras, time domain, time instant, TSQL2, valid time.

## RECOMMENDED READING


[1] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal Versus First-Order Logic to Query Temporal Databases. In *ACM Symposium on Principles of Database Systems*, pages 49–57, 1996.

[2] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [10], chapter 13, pages 294–320.

[3] J. M. V. de Castilho, M. A. Casanova, and A. L. Furtado. A Temporal Framework for Database Specifications. In *International Conference on Very Large Data Bases, VLDB'82*, pages 280–291, 1982.

[4] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Oxford University Press, 1994.

[5] V. Goranko, A. Montanari, and G. Sciavicco. A Road Map of Interval Temporal Logics and Duration Calculi. *Journal of Applied Non-Classical Logics*, 14(1-2):9–54, 2004.

[6] J. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[7] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[9] A. Sernadas. Temporal Aspects of Logical Procedure Definition. *Information Systems*, 5:167–187, 1980.

[10] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[11] D. Toman. On Incompleteness of Multi-dimensional First-order Temporal Logics. In *International Symposium on Temporal Representation and Reasoning and International Conference on Temporal Logic*, pages 99–106, 2003.

[12] D. Toman and D. Niwinski. First-Order Queries over Temporal Databases Inexpressible in Temporal Logic. In *International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, 1996.

[13] A. Tuzhilin and J. Clifford. A Temporal Relational Algebra as a Basis for Temporal Relational Completeness. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *International Conference on Very Large Data Bases, VLDB'90*, pages 13–23, 1990.

[14] J. van Benthem. *The Logic of Time*. D.Reidel, 2nd edition, 1991.

[15] M. Y. Vardi. A Temporal Fixpoint Calculus. In *ACM Symposium on Principles of Programming Languages*, 1988.

[16] P. Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56:72–99, 1983.

# TEMPORAL LOGICAL MODELS

Arie Shoshani
Lawrence Berkeley National Laboratory, Berkeley, California
http://sdm.lbl.gov/~arie

## SYNONYMS

Historical data models

## DEFINITION

Temporal logical models refer to the logical structure of data that captures the temporal behavior and operations over such structures. The term "logical" is used to distinguish such temporal structures from the physical storage organization and implementation. For example, the behavior of temporal events and operations over them can be described logically in a way that is independent of the physical structure (e.g. linked lists) or indexing of the events. Temporal logical models include concepts of data values that are collected or are changed over time, such as continuous physical phenomena, a series of discrete events, and interval data over time. The challenge is one of having a single comprehensive model that captures this diversity of behavior.

## HISTORICAL BACKGROUND

In the 1980's several researchers focused on dealing with temporal data, both on the modeling concepts and on physical organization and indexing of temporal data. This led to the temporal database field to be established, and several books were written or edited on the subject (for example [3, 4, 5]). Since then, the subject continues to appear in specific application domains, or in combination with other concepts, such as spatio-temporal databases, and managing streaming data.

## SCIENTIFIC FUNDAMENTALS

### The treatment of time in database systems

Time is a natural part of the physical world and an indispensable part of human activity, yet many database models treat temporal behavior as an afterthought. For example, weather (temperature, clouds, and storms) is a continuous phenomenon in time, yet it is treated as discrete events per day or per hour. In contrast, some human activities are fundamentally discrete events, such as salary which may change annually, but are treated as continuous concepts, where the salary is the same for the duration between the discrete events of salary changes. The main reason for the inconsistent treatment of time is that temporal objects and their semantics are not explicit in the data model. Consider for example, temperature measurements at some weather station as shown in Figure 1. These are represented in conventional database systems (such as relational data models) as a two-part concept of time-of-measurement and value-of-measurement attributes, but the fact that the measurements are taken at evenly spaced intervals (e.g. every half an hour) and that the temperature represents a continuous phenomenon is not captured. Consequently, if one asks what the temperature was at 12:35am, no such value exists. Furthermore, the interpolation function associated with getting this value is unknown. It could be a simple weighted averaging of the two nearest values, or a more sophisticated curve interpolation function.

### Temporal data behavior

Temporal logical models are models designed to capture the behavior of temporal data sequences. First, some examples that illustrate the concepts that need to be captured by the model are presented.

Example 1: wind velocity. Usually, the measurements of wind velocity are taken by devices at regular time periods, for example every hour. These are referred to as "time series". In this example, the measured quantity is not a single value, but has a more complex structure. It measures the direction of the wind and the velocity of the wind, which can be represented as a three-dimensional vector. The measured

phenomenon is continuous, of course, but for this application it is determined by the database designers that a certain time granularity for queries is desired, such as values by minutes. Since the values are collected only hourly, an interpolation function must be provided and associated with this time sequence. The behavior is similar to the temperature behavior shown in Figure 1, except that the measured values are three-dimensional vectors for each time point.

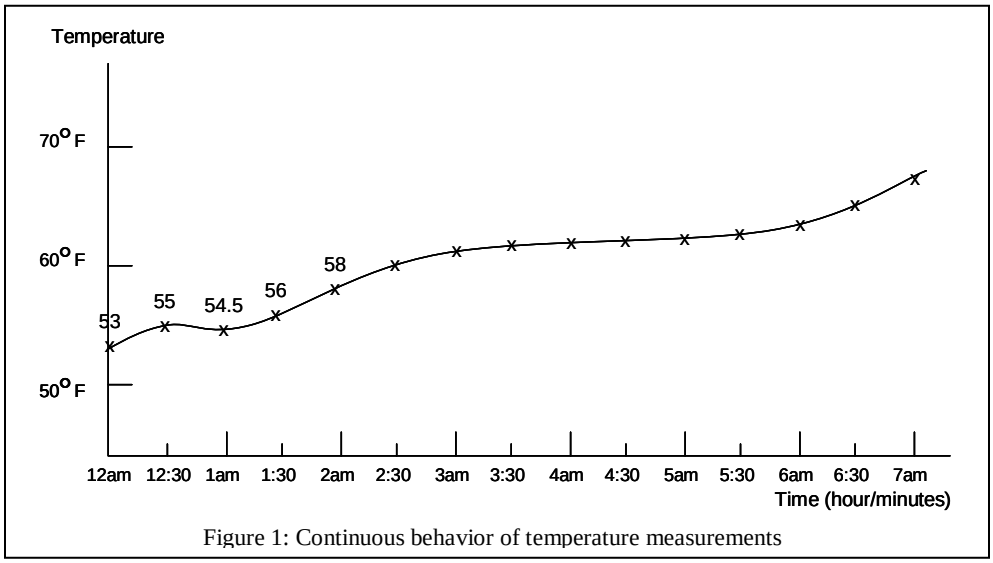



Figure 1: Continuous behavior of temperature measurements

Example 2: bank account. The amount of money in the bank account changes when transactions take place. Money can be added or taken out of the account at irregular times. The value of the account is the same for the duration of time between transactions. This is shown in Figure 2, where the granularity of the time points is in minutes. Note that the days shown should have precise dates in the database. Another aspect in this example is that in the case of a deposit of a check, funds may not be available until the check clears. Thus, there are two times associated with the deposit, the time of the transaction, and the time when funds are made available.

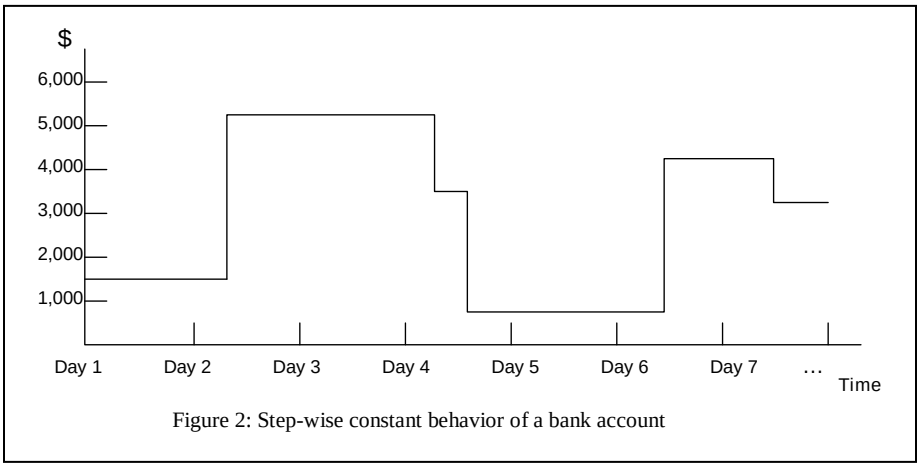



Figure 2: Step-wise constant behavior of a bank account

Example 3: hospitalization visits. Hospital visits of an individual occur typically at irregular times, and each can last a period of time, usually measured in days. The value associated with the hospital visit time sequence is Boolean; that is, only the fact that a visit took place or did not. This is an example where the concept of an interval must be represented in the data model. This is shown in Figure 3, where the granularity is a day, and the interval durations spans days. Here again, the days shown will have to have precise dates in the database.

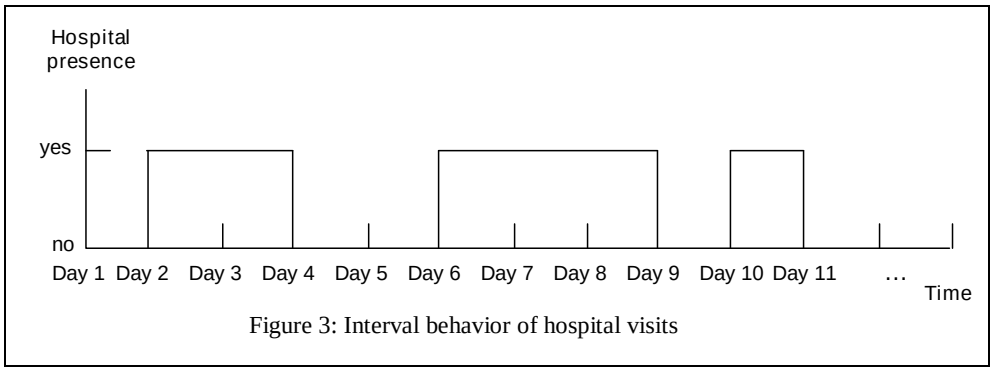



Figure 3: Interval behavior of hospital visits

Example 4: store revenue. Suppose that a store owner wishes to keep in a database the total revenue per day. The time sequence is regular, i.e. a time series (not counting days when the store is closed). The values per day do not represent continuous phenomena, but rather they are discrete in time, collected every day at the end of that day. This is the same as representing discrete events, such as the time of an accident, etc. In general, it does not make sense to apply interpolation to such a time sequence. However, if some data is missing, an interpolation rule could be used to infer the missing values. This is shown in Figure 4. This is a time series, because only the days of business are shown (Monday – Friday).

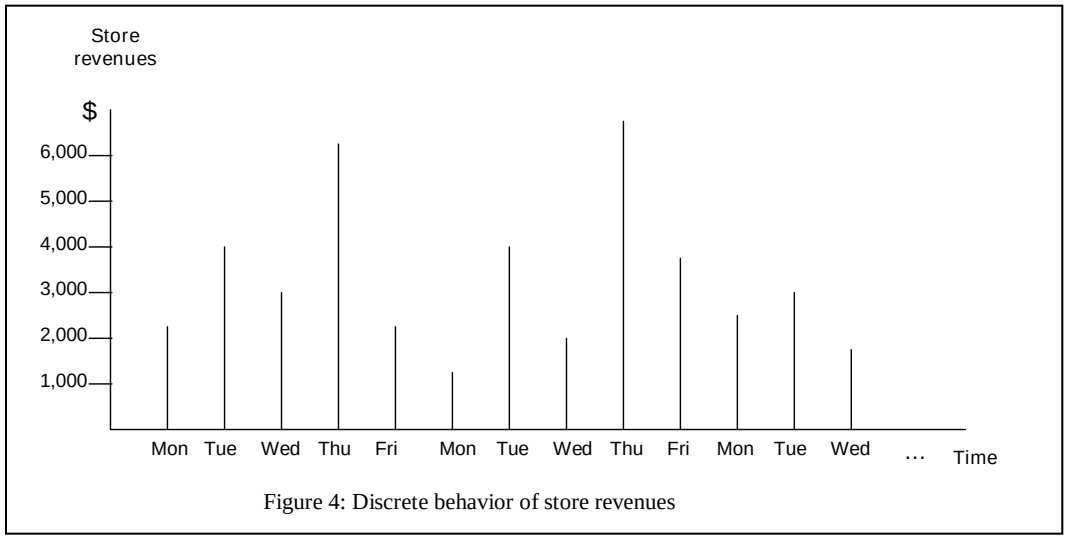



Figure 4: Discrete behavior of store revenues

In the example above, only a single time sequence is shown, but there could be a collection of related time sequences. For example, time sequences of the quantity of each item sold in a store. For each item, there is a time sequence. However, all time sequences in this case have the same behavior, and they are collected in tandem per day. Such groups of related time sequences are referred to as "time sequence collections" [1]. As is discussed later, this concept is important for operations performed over collections of time sequences.

### Behavioral properties of temporal sequences

As is evident from the above examples, there are certain properties that can be specified to capture the logical behavior of temporal sequences. Given that such properties are supported by database systems, it is only necessary in such systems to store temporal data as time-value pairs in the general case, or simply ordered sequences of values for time series. These properties are discussed next, along with the possible category values they can assume.

Time-granularity: value and unit

The time-granularity indicates the time points for which data values can exist in the model. It is the smallest unit of time measure between time points in the time sequence. For example, if deposits and withdrawals to a bank account can be recorded with a minute precision, then the time granularity is said to be a minute. However, in cases where data values can be interpolated, an interpolation-granularity needs to be specified as well. For example, the temperatures shown in Figure 1 are recorded every half an hour, and therefore the time granularity is 30 minutes, but given that values can be interpolated up to a minute precision, it is necessary to specify that the interpolation-granularity is a minute. This point is discussed further below in the section on "interpolation rule". Note that for regular time sequences (time series), it is often necessary to describe the origin of the time sequence, and the time granularity is relative to that origin. A formal treatment of time granularity can be found in [6].

1) Regularity: regular (time series), irregular

As mentioned above time series are regular sequence. They can be described by specifying the "time-step" between the time points. The time-step together with the "life span" (described next) specify fully the time points for which data values are expected to be provided. Because of its regular nature, it is not necessary to store the time points in the databases – these can be calculated. However, this is a physical implementation choice of the system, and the time values can be stored explicitly to provide faster access. Time series are prevalent in many applications, such as statistics gathering, stock market, etc.

Irregular time sequences are useful for event data that occurs in unpredictable patterns, such as bank withdrawals, visits to the library, or making a phone call. A typical phenomena in such sequences, is that most of the time points have no values associated with them. For example, suppose that the time granularity for recording phone calls is a minute. The time sequence of phone calls will typically be mostly empty (or null). For this reason, irregular time sequences are usually represented as time-value pairs in the database.

Life span: begin-time, end-time

The life span indicates for what period of time the time sequence is valid. The begin-time is always necessary, and has to be specified with the same precision of the time granularity. For example, if for the temperature time series in Figure 1, the begin-time was January 1, 1:15am, and the granularity was 30 minutes, then the time points will be 1:15am, 1:45am, 2:15am, etc.

The life span end-time can be specified as "open-ended". That means that this time series is active.

Behavior type: continuous, step-wise-constant, interval, discrete

These types were illustrated in the examples of the previous section. For example 1, on wind velocity, the type is continuous. For example 2, the amount available in a bank account, the type is step-wise-constant. For example 3, of hospital visits, the type is interval. For example 4, the store revenues per day, the type id discrete. Note that the interval type can be considered as a special case of step-wise-constant type having the Boolean values (0 or 1). Another thing worth noting is that discrete time sequences cannot be associated with an interpolation rule. The interpolation rules for the other types are discussed next.

Interpolation rule: interpolation-granularity, interpolation-function

The interpolation-granularity has to be specified in order for the underlying system to enforce the data points for which interpolation can be applied in response to queries. Note that the interpolation-granularity has to be in smaller units than the time-granularity, and the number of interpolation-granularity points in a time-granularity unit must be an integer. For example, while temperature in the example of Figure 1 has time-granularity of 30 minutes, the interpolation-granularity can be 5 minute.

The interpolation-function for the step-wise-constant and interval types are straight-forward, and are implied by the type. But, for a continuous type an interpolation-function must be specified. It should be

possible to provide the system with such a function for each continuous time sequence. If no interpolation-function is provided, the system can use a default function.

Value type: binary, numeric, character, multi-valued, etc.

This property of temporal sequences is no different from specifying attribute types in conventional database systems. The case of a binary type is special to interval events, and is not always supported by conventional system. Also, multi-valued or multi-component attributes are special requirements for more sophisticated time sequences that exist in scientific data, such as the wind velocity in example 2.

Transaction vs. valid time: transaction, valid

Similar to the bank account in example 2 where the deposit time was different from the time when funds are available, there are many examples where temporal data is recorded in the database before the data values are valid. This concept was explored extensively in [2], and referred to as "transaction time" and "valid time". Actually, there are situations where the transaction time can occur after the valid time for retroactive cases. For example, a salary raise can be added to a database in March of some year, but is retroactive to January of that year. This concept has led to an extensive literature on representing it as an extension of query languages, including a temporal extension to the SQL query language, referred to as TSQL [3]. It is worth noting that other concepts of multiple temporal dimensions were also introduced in the literature, in addition to transaction and valid times. For example, [7] introduced the concept of "event time" - times of the events that initiates and terminates the interval validity, and "availability time" – the time interval during which facts are available.

If multiple temporal dimensions are needed in the model, they can be thought of as multiple correlated time sequences. However, in general, each time dimension can have different properties. For example, the transaction time sequence for bank deposits can have a granularity of a minute, while the valid time for the available funds can be daily.

## Operation over temporal data

Because of the inherent time order of temporal data, operations over them, such as "when", "preceding", "following", etc. are based on the time order. Similarly, the concept of a "time window" is natural. Various researchers have developed precise semantics to query languages by adding temporal operators to existing query languages, including relational query languages, such as SQL, relational algebras, such as QUEL, functional query languages, such as DAPLEX, deductive query languages, such a Datalog, and entity-relationship languages. Many such examples can be found in the books on temporal databases [4, 5]. In order to explain the various operators, they are classified into the following four categories.

### Predicate operators over time sequences

Predicate operators refer to either specific times or a time interval. For specific times, the obvious predicates include "before", "after", "when", etc. But, in addition, there are operators that refer to the life span, such as "first, and "last". For time intervals, operators such as "during" or "interval" are used. Also, when specifying an interval, the keyword "now" is used to refer to time sequences that are active, such as "interval (01-01-2007, now). Note that the time values used must be expressed at the granularity of the time sequence (or the interpolation-granularity if interpolation is allowed). In some time sequences, it is useful to use an integer to refer to the $n^{th}$ time instance, such as t-33 to mean the $33^{rd}$ time point in the time sequence. However, this is not included in most proposed query languages.

Another purpose of predicate operators is to get back the time periods where some condition on the data values hold. For example, suppose that a time sequences represents temperature at some location. The query "get periods where temperature > 100" (the units are °F) will return a (Boolean) interval time sequence, where the temperature was greater than 100. Note that "periods" is treated as a keyword.

Aggregation operators over time windows

The usual statistical operators supported by database systems (sum, count, average, min, max) can be applied to a specific time window (t_begin, t_end), to the entire time sequence (first, last), or to the combinations (first, t_end), (t_begin, last). In general, "first" or "last" can be substituted by an instance number, such as "t-33" mentioned above. Here, again, the time has to be specified in the granularity of the time sequence.

Another way to apply operators over windows is to combine that with the "group by" concept. This is quite natural for temporal sequence that involve calendar concepts of month, day, minute, second, etc. For example, suppose that a time sequence represents daily sales. One can have a query "sum sales by month". This is the same as asking for multiple windows, each over the days in each month.

Aggregation operators over time sequence collections

In a typical database, time sequences are defined over multiple object instances. For example, one can have in an organization the salary history of all of its employees. Asking for the "average salary over all employees" over time requires the average operation to be applied over the entire collection of time sequences. This operation is not straight forward if all salary raises do not occur at the same time. This operation will generate a time sequence whose time points are the union of all the time points of the time sequences, where the average values are performed piecewise on the resulting intervals.

Similar to the case of aggregation over time windows, where the "group by" operation can be applied, it is possible to group by object instances in this case. For example, if employees are organized by departments, one can ask for "average salary over all employees per department".

Composition of time sequences

Composition refers to algebraic operations over different time sequences. For example, suppose that in addition to salary history recorded for each employee in an organization, the history of commissions earned is recorded. In order to obtain "total income", the salary and the commission time sequences have to be added for each employee. This amounts to the temporal extension of algebraic operations on multiple attributes in non-temporal query languages.

Combinations of the above operators

It is conceptually reasonable to combine the above operators in query languages. For example, it should be possible to have the aggregation and composition operators applied only to a certain time window, such as getting the "average salary over all employees for the last three years". Furthermore, it should be possible to apply a temporal operator to the result of another temporal operator. This requires that the result of operations over time sequences is either a time sequence or a scalar. If it is a time sequence, temporal operators can be applied. If it is a scalar (a single value) it can be applied as a predicate value. This requirement is consistent with other languages, such as the relational language, where the operation on relations always generates a relation or a scalar.

Additional related concepts

There are many concepts introduced in the literature that capture other logical aspects of temporal operations and semantics. This broad literature cannot be covered here; instead, several concepts are mentioned next. [8] explores temporal specialization and generalization. [9] and [10] develop unified models for supporting point-based and interval-based semantics. [11] argues that temporal data models have to include explicitly the concept of ordered data, and proposes a formal framework for that. [12] develops a single framework for supporting both time series and version-based temporal data. There are also many papers that discuss how to efficiently support temporal operations (such as aggregation), see for example [13, 14]. Finally, temporal aggregation operations on time windows have in explored in the context of streaming data – see entries on "stream data management" and "stream mining".

## KEY APPLICATIONS

Temporal data is ubiquitous. It naturally exists in applications that have time series data, such as stock market historical data, or history of transactions in bank accounts. In addition, it is a basic requirement of scientific databases collecting data from instruments or performing simulation over time steps. In the past, many databases contained only the current (most updated) data, such as current salary of employees, current inventories in a store or a warehouse, etc. The main reason for that was the cost of storage and efficiency of processing queries. One could not afford keeping all the historical data. More recently, as the cost of storage is plummeting, and compute engines are faster and can operate in parallel, historical data is routinely kept. While it is still worth keeping a version for current data for some applications for efficiency of access, many applications now use historical data for pattern and trend analysis over time, especially in data warehouse applications.

## FUTURE DIRECTIONS

While a lot of research was done on temporal data, the concepts and operations over such data are only partially supported, if at all, in commercial and open source database system. Some support only the concept of date_time (it is complex enough, crossing time zones and century boundaries), but the support for properties of time sequences and operations over them are still not generally available. Building such database systems is still a challenge.

## CROSS REFERENCES

Data Models, Query Languages, Database Design, Spatial and Multidimensional Databases, Data Warehouse, Stream Data Management, Stream Mining, Temporal Aggregation, Temporal Join.

## RECOMMENDED READING


[1] Arie Segev, Arie Shoshani: Logical Modeling of Temporal Data. SIGMOD Conf. 1987: 454-466.
[2] Richard T. Snodgrass, Ilsoo Ahn: Temporal Databases. IEEE Computer 19 (9): 35-42 (1986)
[3] Richard T. Snodgrass: The TSQL2 Temporal Query Language Kluwer 1995
[4] Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Jajodia, Arie Segev, Richard T. Snodgrass: Temporal Databases: Theory, Design, and Implementation Benjamin/Cummings 1993
[5] Opher Etzion, Sushil Jajodia, Suryanarayana M. Sripada (Eds.): Temporal Databases: Research and Practice. (The book grow out of a Dagstuhl Seminar, June 23-27, 1997). Lecture Notes in Computer Science 1399 Springer 1998, ISBN 3-540-64519-5
[6] Claudio Bettini, Xiaoyang Sean Wang, Sushil Jajodia: A General Framework for Time Granularity and Its Application to Temporal Reasoning. Ann. Math. Artif. Intell. 22 (1-2): 29-58 (1998).
[7] Carlo Combi, Angelo Montanari: Data Models with Multiple Temporal Dimensions: Completing the Picture. CAiSE 2001: 187-202
[8] Christian S. Jensen, Richard T. Snodgrass: Temporal Specialization and Generalization. IEEE Trans. Knowl. Data Eng. 6(6): 954-974 (1994)
[9] Cindy Xinmin Chen, Carlo Zaniolo: Universal Temporal Extensions for Database Languages. ICDE 1999: 428-437
[10] Paolo Terenziani, Richard T. Snodgrass: 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)
[11] Yan-Nei Law, Haixun Wang, Carlo Zaniolo: Query Languages and Data Models for Database Sequences and Data Streams. VLDB 2004: 492-503
[12] Jae Yong Lee, Ramez Elmasri, Jongho Won: An Integrated Temporal Data Model Incorporating Time Series Concept. Data Knowl. Eng. 24(3): 257-276 (1998)
[13] Sung Tak Kang, Yon Dohn Chung, Myoung-Ho Kim: An efficient method for temporal aggregation with range-condition attributes. Inf. Sci. 168(1-4): 243-265 (2004)
[14] Bongki Moon, Inés Fernando Vega López, Vijaykumar Immanuel: Efficient Algorithms for Large-Scale Temporal Aggregation. IEEE Trans. Knowl. Data Eng. 15(3): 744-759 (2003)

# Temporal Object-Oriented Databases

Carlo Combi
University of Verona, http://www.di.univr.it/∼combi/

16th January 2008

## SYNONYMS

Temporal Object Databases.

## DEFINITION

In a strict sense, a temporal object-oriented database is a database managed by an object-oriented database system able to explicitly deal with (possibly) several temporal dimensions of data. The managed temporal dimensions are usually valid and/or transaction times. In a wider sense, a temporal object-oriented database is a collection of data having some temporal aspect and managed by an object-oriented database system.

## HISTORICAL BACKGROUND

Research studies on time, temporal information, and object-oriented data started at the end of eighties and continued in the nineties. From the seminal work by Clifford and Croker on objects in time [1], several different topics have been discussed. Segev and Rose studied both the modeling issues and the definition of suitable query languages [2]; Wuu and Dayal showed how to use an object-oriented data model to properly represent several temporal aspects of data [3]; Goralwalla et al. studied the adoption and extension of an object-oriented database system, TIGUKAT, for managing temporal data, and finally proposed a framework allowing one to classify and define different temporal object-oriented data models, according to the choices adopted for representing time concepts and for dealing with data having temporal dimensions [4]. Schema evolution in the context of temporal object-oriented databases has been studied by Goralwalla et al. and by Edelweiss et al. [5, 6]. Bertino et al. studied the problem of formally defining temporal object-oriented models [7] and, more recently, they proposed an extension of the ODMG data model, to deal with temporalities [8]. Since the end of nineties, research has focused on access methods and storage structures, and on temporalities in object relational databases, which, in contrast to object-oriented databases, extend the relational model with some object-oriented features still maintaining all the relational structures and the related, well studied, systems. As for the

first research direction, for example, Norvag studied storage structures and indexing techniques for temporal object databases supporting transaction time [9]. Recently, some effort has been done on extending object-relational database systems with some capability to deal with temporal information; often, as in the case of the work by Zimanyi and colleagues, studies on temporalities in object-relational databases are faced in the context of spatio-temporal databases, where temporalities have to be considered together with the presence of spatial information [10]. In these years, there were few survey papers on temporal object-oriented databases, even though they were considered as a part of survey papers on temporal databases [11, 12]. Since the nineties, temporal object-oriented databases have also been studied in some more application-oriented research efforts: among them are temporal object models and languages for multimedia and for medical data, as, for example, in [13, 14].

# SCIENTIFIC FUNDAMENTALS

Object-oriented (OO) methodologies and technologies applied to the database field have some useful features - abstract data type definition, inheritance, complex object management - in modeling, managing, and storing complex information, as that related to time and to temporal information. Objects and times have, in some sense, a twofold connection: from a first point of view, complex, structured types could be suitably defined and used to manage both complex time concepts and temporal dimensions of the represented information; on the other side, object-oriented data models and languages may be suitably extended with some built-in temporal features. In the first case, the object-oriented technology is performed to show that complex concepts, as those related to time and temporal information, can be suitably represented and managed through types, generalization hierarchies, and so on. In the second case, the focus is slightly different and is on extending object-oriented models and languages to consider time-related concepts as first-class citizens: so, for example, each object, besides an object identifier, will have a lifespan, managed by the system.

Even though different approaches and proposals in the literature do not completely agree on the meaning of different terms used by the "object-oriented" community, there is a kind of agreement on the basic concepts of any object-oriented data model. An *object* may represent any entity of the real world, e.g., a patient, a therapy, a time interval. The main feature of an object is its *identity*, which is immutable, persists during the entire existence of the object, and is usually provided by the database system through an identifier, called OID (Object IDentifier). An object is characterized by a state, described by properties (*attributes* and *associations* with other objects) and by a behavior, defined by *methods*, describing modalities, by which it is possible to interact with the object itself. Objects are created as instances of a *type*; a *class* is the collection (also named *extent*) of all the objects of a given type stored into the database at a certain moment. A *type* describes (i) the structure of properties through attributes and associations, and (ii) the behavior of its instances through methods applicable to objects instances of that type.

To introduce the faced topics, a simple clinical database is used, storing data on symptoms and therapies of patients and will graphically represent its schema through

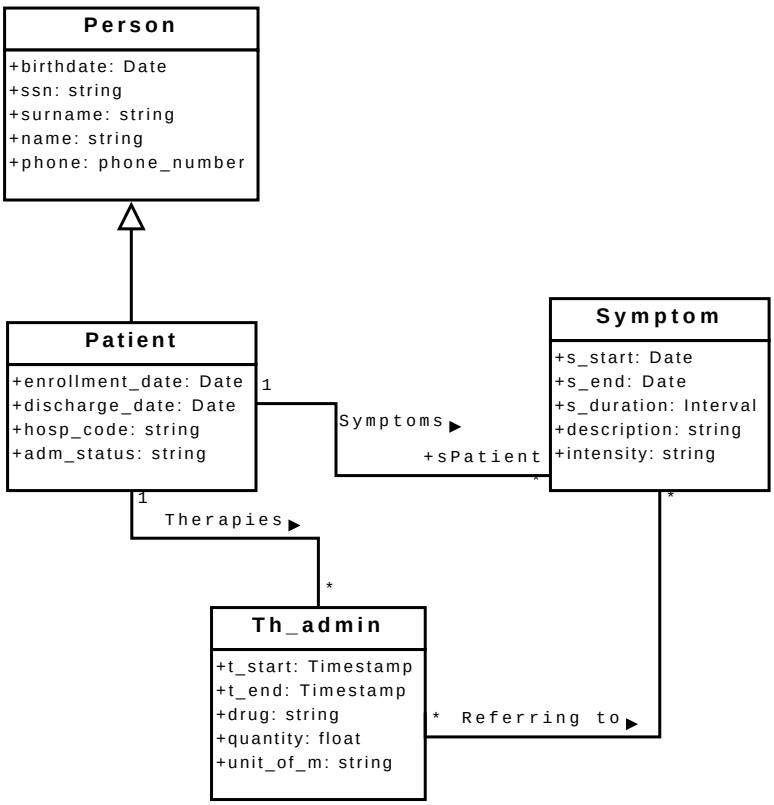



Figure 1: Object-oriented schema of the database about patients' symptoms and therapies, using ODMG standard time types.

the well-known UML notation for class diagrams, adopting the primitive types provided by the ODMG data model [15].

## Time and abstract data types

Through suitable abstract data types (ADTs), it is possible to represent several and different features of time primitives; more specifically ADTs can be suitably used to represent anchored (i.e., time points and time periods) and unanchored (i.e., durations) temporal primitives. Usually, any OO database system allows one to define ADTs, which can be possibly used to represent complex temporal concepts; moreover, any database system usually provides several built-in primitive and complex data types, allowing the user to represent the most usual time concepts.

Considering, for example, the clinical database depicted in Figure 1, it is possible to observe different data types for time; according to the ODMG data model, the types Date and Timestamp represent time points corresponding to days and to seconds, respectively, while the type Interval stands for spans of time, i.e., distances between

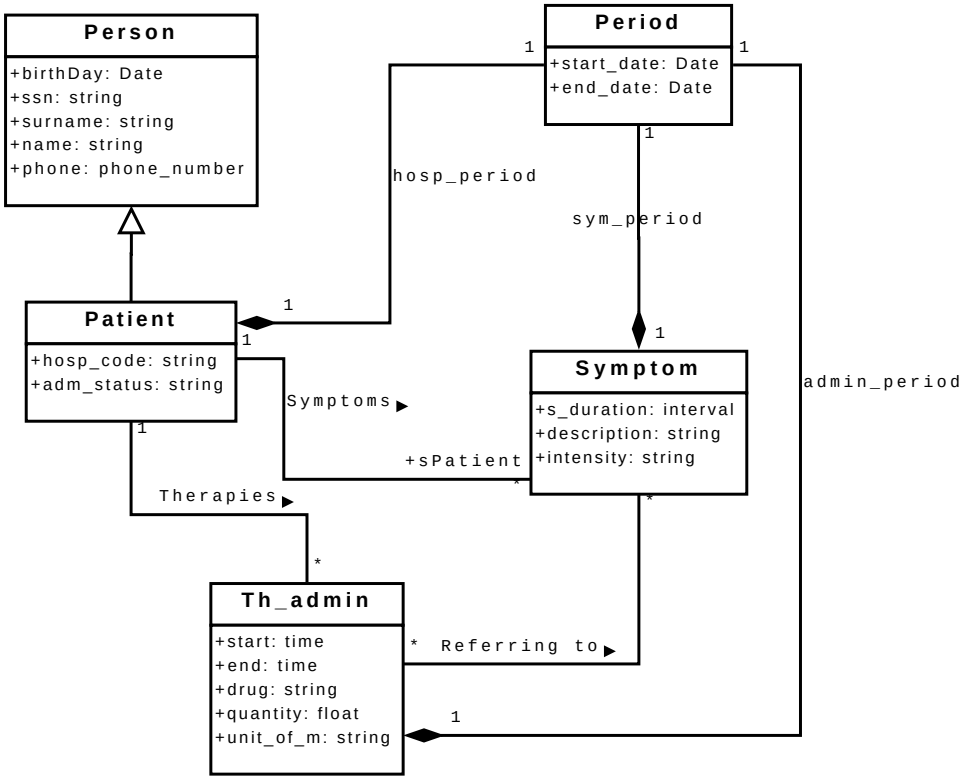



Figure 2: Object-oriented schema of the database about patients' symptoms and therapies, introducing the type Period.

two time points.

According to this approach, as underlined by Wuu and Dayal [3], different types of time could be defined, starting from some general concept of time, through aggregation and generalization. For example, the ODMG type Timestamp could be viewed as a specialization of a supertype Point, having two operations defined, i.e., those for comparisons ($=$ and $>$), as its behavior. Various time types can be defined as subtypes of Point. The two operations defined for Point are inherited, and could be suitably redefined for different subtypes. Indeed, properties of specific ordering relationships determine different time structures, e.g., total vs. partial, or dense vs. discrete. Moreover, through the basic structured types provided by the model, it is possible to build new types for time concepts. As an example, it could be important to specify a type allowing one to represent periods of time; this way, it is possible to represent in a more compact and powerful way the periods over which a therapy has been administered, a symptom was holding, a patient was hospitalized. Thus, in the database schema a new type Period will be used as part of types Patient, Th_admin, and Symptom, as depicted in Figure 2.

ADTs may be used for managing even more complex time concepts [13], allow-

ing one to deal with times given at different granularities or with indeterminacy in a uniform way.

## Temporal object data models

Up to this point, the proposed solutions just use ADTs to represent complex time concepts; the sound association of time to objects for representing temporal data, i.e., time evolving information, according to the well-known temporal dimensions, such as valid and/or transaction times, is left to the user. Focusing, without loss of generality, on valid time, in the given example the semantics of valid time must be properly managed by the application designer, to check that symptoms occurred and therapies were administered when the patient was alive, that therapies related to symptoms were administered after that symptoms appeared, that there are no different objects describing the same symptom with two overlapping periods, and so on.

Managing this kind of temporal aspects in object databases has been the main focus of several research studies [11]: the main approaches adopted in dealing with the temporal dimension of data in object-oriented data models are i) the direct use of an object-oriented data model (sometimes already extended to deal with other features as version management), and ii) the modeling of the temporal dimensions of data through ad-hoc data models. The first approach is based on the idea that the rich (and extensible) type system usually provided by OO database systems allows one to represent temporal dimensions of data as required by different possible application domains. The second approach, instead, tries to provide the user with a data model where temporal dimensions are first-class citizens, avoiding the user the effort of modeling from scratch temporal features of data for each considered application domain.

### General OO models using OO concepts for modeling temporal dimensions

Among the proposed object-oriented systems able to deal with temporal information, OODAPLEX and TIGUKAT adopt the direct use of an object-oriented data model [3, 14]. In these systems suitable data types allow the database designer to model temporal information. For example, TIGUKAT models the valid time at the level of objects and of collections of objects. More particularly, for each single application it is possible to use the rich set of system-supported types, to define the real semantics of valid (or transaction) time.

According to this approach, the example database schema could be modified, to manage also the valid time semantics: as depicted in Figure 3, objects having the valid time dimension are explicitly managed through the type VT_Obj, the supertype of all types representing temporal information, while the type Sym_History, based on the (template) type T_validHistory is used to manage the set of objects representing symptoms.

### OO models having explicit constructs for temporal dimensions of data

Besides the direct use of an OO data model, another approach which has been widely adopted in dealing with the temporal dimension of data by object-oriented data models

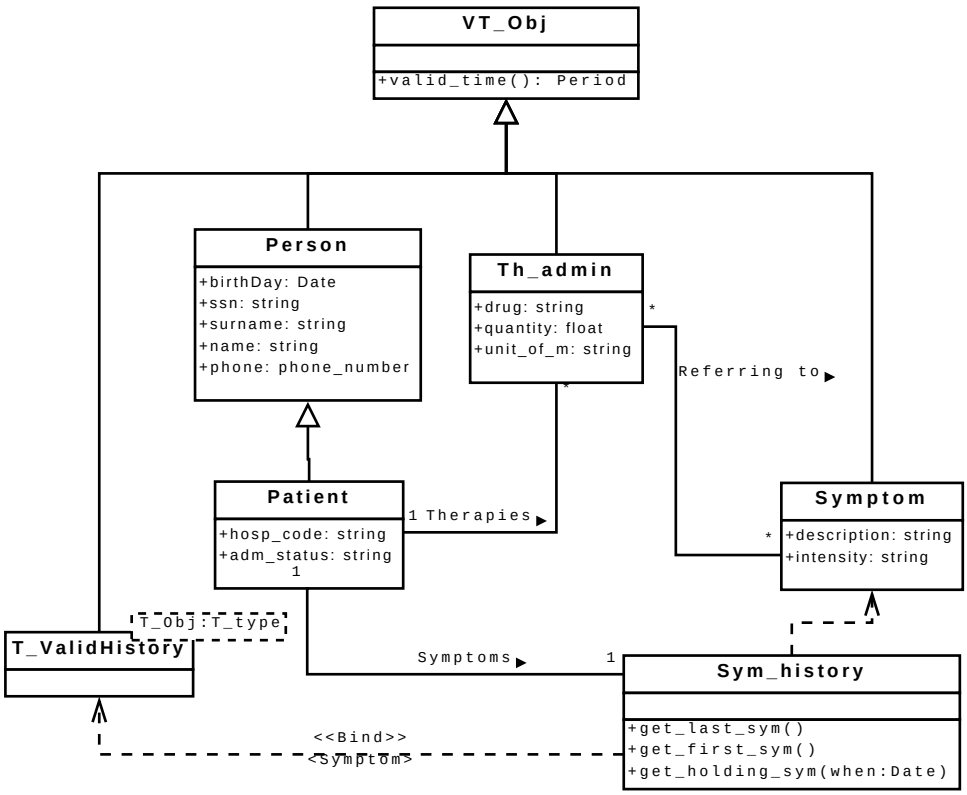



Figure 3: Object-oriented schema of the database about patients' symptoms and therapies, modeling temporal dimensions through suitable types.

consists of temporally-oriented extensions, which allow the user to explicitly model and consider temporal dimensions of data. As an example of this approach, Figure 4 depicts the schema related to patients' symptoms and therapies: the temporal object-oriented data model underlying this schema allows one to manage the valid time of objects, often referred to as *lifespan* in the object-oriented terminology (this aspect is represented through the stereotype temporal and the operation lifespan() for all types); moreover, the temporal model allows the designer to specify that attributes within types can be time-varying according to different granularities: for example, the intensity of a symptom may change over time. Finally, temporalities can be represented even for associations, specifying the granularity to consider for them. Several constraints can be defined, and implicitly verified, when defining a temporal object-oriented data model. For example, in this case, any object attribute could be constrained to hold over some subinterval of the lifespan of the considered object: $\forall o \in \mathsf{Symptom}(o.\mathsf{intensity}.VT() \subseteq o.\mathsf{lifespan}())$, where $VT()$ returns the overall period over which the (varying) values of a temporal attribute have been specified. In a similar way, a temporal relationship is allowed only when both the related objects

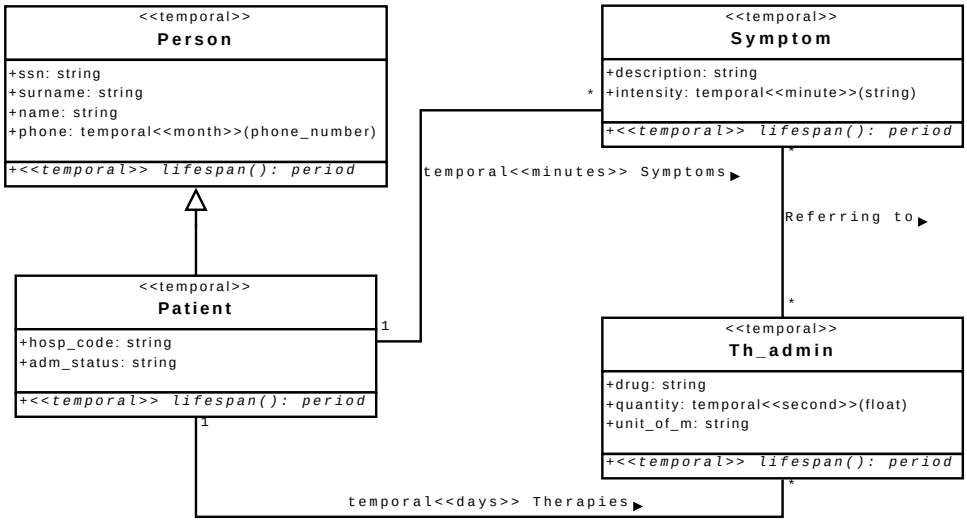



Figure 4: The temporal object-oriented schema of the database about patients' symptoms and therapies.

exist. As for the inheritance, objects could be allowed to move from some supertype to a subtype during their existence; the constraint here is that their lifespan as instances of a subtype must be a subinterval of their lifespan as instances of a supertype: $\forall o \in \mathsf{Patient}(o.\mathsf{lifespan}() \subseteq ((\mathsf{Person})\ o).\mathsf{lifespan}())$

## Temporal object query languages

According to the approaches for object-oriented data models dealing with temporal information, even when querying temporal data, it is possible to either adopt a generic object-oriented query language and use directly its constructs for temporal data or extend an (atemporal) query language with some ad-hoc keywords and clauses to deal with temporal data in a more powerful and expressive way [2, 15, 13]. For both the approaches, the main focus is on querying data: data definition and data manipulation are usually performed through the object-oriented language provided by the database system [13].

## Temporal object-oriented database systems

Usually, temporal object-oriented database systems, offering a temporal data model and a temporal query language, are realized on top of object-oriented database systems, through a software layer able to translate temporalities in data and queries into suitable data structures and statements of the underlying OO system [13]. According to this point of view, only recently some studies have considered technical aspects at the physical data level; among them it is worth mentioning here the indexing of time

objects and storage architectures for transaction time object databases [9].

# KEY APPLICATIONS

Clinical database systems are among the applications of temporal object-oriented databases, i.e., the real world databases where the structural complexity of data needs for the object-oriented technology. Indeed, clinical database systems have to manage temporal data, often with multimedia features, and complex relationships among data, due both to the healthcare organization and to the medical and clinical knowledge. Attention in this field has been paid on the management of clinical data given at different granularities and with indeterminacy [13, 14]. Another interesting application domain is that of video database systems, where temporal aspects of object technology have been studied for the management of temporal aspects of videos. Complex temporal queries have been studied in this context, involving spatio-temporal constrains between moving objects.

# FUTURE DIRECTIONS

New and more intriguing topics have attracted the attention of the temporal database community in these last years; however, the results obtained for temporal object-oriented databases could be properly used in different contexts, such as that of temporal object-relational databases, which seem to attract also the attention of the main companies developing commercial database systems, and that of semistructured temporal databases, where several OO concepts could be studied and extended to deal with temporalities for partially structured information (as that represented through XML data).

# CROSS REFERENCE

Object-Relational Data Model, Object-Oriented Data Model, Object Query Language, Spatio-temporal Data Models, Temporal Granularity, Temporal Indeterminacy, Temporal Query Languages.

# RECOMMENDED READING

## References


[1] Clifford, J. and Croker, A. (1988) Objects in time. *IEEE Data Eng. Bull.*, **11**, 11–18.

[2] Rose, E. and Segev, A. (1993) TOOSQL - A Temporal Object-Oriented Query Language. *12th International Conference of the Entity-Relationship Approach*, pp. 122–136, LNCS 823, Springer.

[3] Wuu, G. T. J. and Dayal, U. (1993) A Uniform Model for Temporal and Versioned Object-oriented Databases. *Temporal Databases*, pp. 230–247.

[4] Goralwalla, I. A., Özsu, M. T., and Szafron, D. (1997) An object-oriented framework for temporal data models. *Temporal Databases, Dagstuhl*, pp. 1–35.

[5] Goralwalla, I. A., Szafron, D., Özsu, M. T., and Peters, R. J. (1998) A temporal approach to managing schema evolution in object database systems. *Data Knowl. Eng.*, **28**, 73–105.

[6] de Matos Galante, R., dos Santos, C. S., Edelweiss, N., and Moreira, A. F. (2005) Temporal and versioning model for schema evolution in object-oriented databases. *Data Knowl. Eng.*, **53**, 99–128.

[7] Bertino, E., Ferrari, E., and Guerrini, G. (1996) A formal temporal object-oriented data model. Apers, P. M. G., Bouzeghoub, M., and Gardarin, G. (eds.), *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, vol. 1057 of *Lecture Notes in Computer Science*, pp. 342–356, Springer.

[8] Bertino, E., Ferrari, E., Guerrini, G., and Merlo, I. (2003) T-odmg: an odmg compliant temporal object model supporting multiple granularity management. *Inf. Syst.*, **28**, 885–927.

[9] Nørvåg, K. (2004) The vagabond approach to logging and recovery in transaction-time temporal object database systems. *IEEE Trans. Knowl. Data Eng.*, **16**, 504–518.

[10] Zimányi, E. and Minout, M. (2006) Implementing conceptual spatio-temporal schemas in object-relational dbmss. Meersman, R., Tari, Z., and Herrero, P. (eds.), *OTM Workshops (2)*, vol. 4278 of *Lecture Notes in Computer Science*, pp. 1648–1657, Springer.

[11] Snodgrass, R. *Temporal Object-Oriented Databases: A Critical Comparison*, pp. 386–408.

[12] Ozsoyoglu, G. and Snodgrass, R. T. (1995) Temporal and real-time databases: a Survey. *IEEE Transactions on Knowledge and Data Engineering*, **7**, 513–532.

[13] Combi, C., Cucchi, C., and Pinciroli, F. (1997) Applying Object-Oriented Technologies in Modeling and Querying Temporally-Oriented Clinical Databases Dealing with Temporal Granularity and Indeterminacy. *IEEE Transactions on Information Technology in Biomedicine*, **1**, 100–127.

[14] Goralwalla, I., Ozsu, M., and Szafron, D. (1997) Modeling Medical Trials in Pharmacoeconomics Using a Temporal Object Model. *Computers in Biology and Medicine*, **27**, 369–387.

[15] Cattel, R. and Barry, D. (eds.) (2000) *The Object Data Standard: ODMG 3.0*. Morgan - Kaufmann.

# TEMPORAL PERIODICITY

**Paolo Terenziani**
**Universita' del Piemonte Orientale "Amedeo Avogadro",** http://www.di.unito.it/~terenz

## SYNONYMS
None

## DEFINITION
Informally, *periodic events* are events that repeat regularly in time (e.g., each Tuesday), and *temporal periodicity* is their temporal *periodic pattern* of repetition. A pattern is *periodic* if it can be represented by specifying a finite portion of it, and the duration of each repetition. For instance, supposing that day 1 is a Monday, the pair *<'day 2', '7 days'>* may implicitly represent all Tuesdays.
A useful generalization of periodic patterns are *eventually* periodic ones, i.e., patterns that can be expressed by the union of a periodic pattern and a finite non-periodic one.

The above notion of periodic events can be further extended. For instance, Tuzhilin and Clifford [14] distinguish between *"strongly"* periodic events, that occur at equally distant moments in time (e.g., a class, scheduled to meet once a week, on Wednesday at 11am), *"nearly periodic"* events, occurring at regular periods, but not necessarily at equally distant moments of time (e.g., a meeting, that has to be held once a week, but not necessarily on the same day), and *"intermittent"* events, such that if one of them occurred then the next one will follow some time in the future, but it is not clear when (e.g., a person visiting "periodically" a pub). Most of the approaches discussed in the following cope only with "strongly" periodic events.

Finally, it is worth highlighting that "(periodic) temporal granularity", "calendar", and "calendric system" are notions closely related to temporal periodicity.

## HISTORICAL BACKGROUND
Temporal periodicity is pervasive of the world all around us. Many natural and artificial phenomena take place at periodic time, and temporal periodicity seems to be an intrinsic part of the way humans approach reality. Many real-world applications, including process control, data access control, data broadcasting, planning, scheduling, multimedia, active databases, banking, law and so on need to deal with periodic events. The problem of how to store and query periodic data has been widely studied in the fields of databases, logic, and artificial intelligence.
In all such areas it is widely agreed that, since many different data conventions exist, a pre-defined set of periodicities would not suffice. For instance, Snodgrass and Soo [12] have emphasized that the use of a calendar depends on the cultural, legal, and even business orientation of the users, and listed many examples of different calendric systems. As a consequence, many approaches to *user-defined* temporal periodicity have been proposed. The core issue is the definition of expressive *formal languages* to *represent* and *query* user-defined temporal periodicity. In particular, an *implicit* representation [2] is needed, since it allows one to cope with data holding at periodic times in a compact way instead of explicitly listing all the instances (extensions) of the given periodicity (e.g., all "days" in a possibly infinite frame of time). Additionally, also the *set-theoretic operations* (intersection, union, difference) on definable periodicities can be provided (e.g., in order to make the formalism a suitable candidate to

represent periodic data in temporal databases). Operationally, also *mapping functions* between periodicities are an important issue to be taken into account.
Within the database community, the problem of providing an implicit treatment of temporal periodicity has been intensively investigated since the late 1980's. Roughly speaking, such approaches can be divided into three mainstreams (the terminology is derived from Baudinette et al. [2] and Niezette and Stevenne [9]):

(1) *Deductive rule*-based approaches, using deductive rules. For instance, Chomicki and Imielinsky [4] dealt with periodicity via the introduction of the successor function in Datalog;
(2) *Constraint*-based approaches, using mathematical formulae and constraints (e.g., [6]);
(3) *Symbolic* approaches, providing symbolic formal languages to cope with temporal periodicity in a compositional (and hopefully *natural* and *commonsense*) way (consider, e.g., [8, 9]).

Tuzhilin and Clifford [14] have proposed a comprehensive survey of many works in such mainstreams, considering also several approaches in the areas of logics and of artificial intelligence.

## SCIENTIFIC FUNDAMENTALS

In the following, the main features of the three mainstreams mentioned above are analyzed.

### 1. Deductive rule-based approaches

Probably the first milestone among *deductive rule-based* approaches is the seminal work by Chomicki and Imielinski [4], who used $\text{Datalog}_{1S}$ to represent temporal periodicity. $\text{Datalog}_{1S}$ is the extension to Datalog with the successor function. In their approach, *one* temporal parameter is added to $\text{Datalog}_{1S}$ predicates, to represent time. For instance, in their approach, the schedule of backups in a distributed system can be represented by the following rules:

$\text{backup}(T+24,X) \leftarrow \text{backup}(T,X)$
$\text{backup}(T,Y) \leftarrow \text{dependent}(X,Y), \text{backup}(T,X)$

The first rule states that a backup on a machine should be taken every 24 hours. The second rules requires that all backups should be taken simultaneously on all dependent machines (e.g., sharing files). Of course, $\text{Datalog}_{1S}$ programs may have infinite least Herbrand models, so that infinite query answers may be generated. However, in their later works Chomicki and Imielinski have provided a finite representation of them.
Another influential rule-based approach is based on the adoption of Templog, an extension of logic programming based on temporal logic. In this language, predicates can vary with time, but the time point they refer to is defined implicitly by temporal operators rather than by an explicit temporal argument [2]. Specifically, three temporal operators are used in Templog: *next*, which refers to the next time instant, *always*, which refers to the present and all the future time instants, and *eventually*, which refers to the present or to some future time instant.

### 2. Constraint-based approaches

While deductive rule-based approaches rely on deductive database theory, the approaches of the other mainstreams apply to relational (or object-oriented) databases.
Kabanza et al. [6] have defined a *constraint-based* formalism based on the concept of *linear repeating points* (henceforth lpr's). A lrp is a set of points $\{x(n)\}$ defined by an expression of the form $x(n)=c+kn$ where *k* and *c* are integer constants and *n* ranges over the integers.
A *generalized tuple* of temporal arity *k* is a tuple with *k* temporal attributes, each one represented by a lrp, possibly including *constraints* expressed by linear equalities or disequalities between temporal attributes. Semantically, a generalized tuple denotes a (possibly

infinite) set of (ordinary) tuples, one tuple for each value of the temporal attributes satisfying the lrp's definitions and the constraints. For instance, the generalized tuple

$$(a_1, \ldots, a_n \mid [5+4n_1, 7+4n_2] \wedge X_1=X_2-2)$$

(with data part $a_1, \ldots, a_n$) represents the infinite set of tuples with temporal attributes $X_1$ and $X_2$, such that $X_1=5+4n_1$, $X_2=7+4n_2$, $X_1=X_2-2$, for some integers $n_1$ and $n_2$, i.e.,

$$\{ \ldots, (a_1, \ldots, a_n \mid [1,3]), (a_1, \ldots, a_n \mid [5,7]), (a_1, \ldots, a_n \mid [9,11]), \ldots\}$$

A *generalized relation* is a finite set of generalized tuples of the same schema.
In Kabanza et al. [6], the algebraic operations (e.g., intersection) have been defined over generalized relations as mathematical manipulations of the formulae coding lrp's.

A comparative analysis of deductive rule-based and constraint-based approaches has been provided, e.g., by Baudinet et al. [2], showing that they have the same *data expressiveness* (i.e., the set of temporal databases that can be expressed in such languages is the same). Specifically, as concerns the temporal component, they express *eventually periodic* sets of points (which are, indeed, points that can be defined by *Presburger Arithmetics* – see below). In such an approach, the *query* expressiveness and the computational complexity of such formalisms have also been studied.

**3. Symbolic approaches**

In constraint-based approaches, temporal periodicity is represented through mathematical formulae. Several authors have suggested that, although expressive, such approaches do not cope with temporal periodicity in a "commonsense" (in the sense of "human-oriented") way, arguing in favor of a *symbolic* representation, in which complex periodicities can be compositionally built in terms of simpler ones (see, e.g., the discussions in [9, 13]). A milestone in the area of symbolic approaches to temporal periodicity is the early approach by Leban et al., [8]. In Leban's approach, *collections* are the core notion. A *collection* is a *structured* set of *intervals* (elsewhere called *periods*). A base collection, partitioning the whole timeline (e.g., the collection of seconds), is used as the basis of the construction. A new partition P' of the timeline (called *calendar* in Leban's terminology) can be built from another partition P as follow:

$$P'=<P; s_0, \ldots, s_{n-1}>$$

where $s_0, \ldots, s_{n-1}$ are natural numbers greater than 0. Intuitively, $s_0, \ldots, s_{n-1}$ define the number of periods of P whose unions yield periods of P', intending that the union operation has to be repeated cyclically. For example, Weeks={Days;7} defines weeks as aggregations of seven days. Analogously, months (not considering leap years for the sake of brevity) can be defined by the expression Months={Days; 31,28,31,30,31,30,31,31,30,31,30,31}.
Two classes of operators, *dicing* and *slicing*, are defined in order to operate on collections. The dicing operators provide means to further divide each period in a collection within another collection. For example, given the definitions of *Days* and *Weeks*, Day:during:Weeks breaks up weeks into the collection of days they contain. Other dicing operators are allowed (taken from a subset of Allen's relations [1]). Slicing operators provide a way of selecting periods from collections. For instance, in the expression 2/ Day:during:Weeks the slicing operator "2/" is used in order to select the second day in each week. Different types of slicing operators are provided (e.g., -n/ selects the *n-th* last interval from each collection).
Collection expressions can be arbitrarily built by using a combination of these operators.
While Leban's approach was mainly aimed to represent periodicity within knowledge bases (i.e., in artificial intelligence contexts), later on it has played an increasingly important role also in the area of temporal databases. Recently, Terenziani [13] has defined a temporal extension to relational databases, in which the valid time of periodic tuples can be modeled through (an extension of) Leban's language, and symbolic manipulation is used in order to perform algebraic operations on the temporal relations.

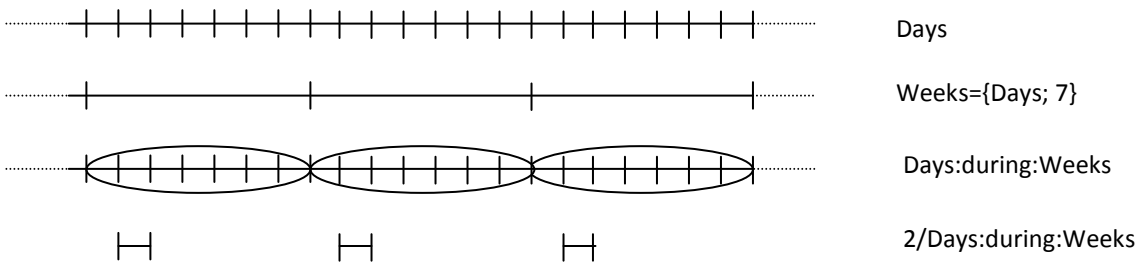



Figure 1. Operators in Leban's language

Another early symbolic approach which has been widely used within the database and artificial intelligence areas is the one by Niezette and Stevenne [9], who provided a formalism as well as the algebraic operations on it, mostly as an upper layer built upon *linear repeating points* [6]. A comparison between such a formalism and Leban's one, as well as an analysis of the relationships between the periodicities defined by such languages and *periodic granularities* has been provided by Bettini and De Sibi [3]. A recent influential symbolic approach, based on the notion of periodic granularities, has been proposed by Ning et al. [10].

An important issue concerns the formal analysis of the expressiveness (and of the semantics) of the implicit representation formalisms proposed to cope with temporal periodicity. *Presburger Arithmetics*, i.e., the first-order theory of addition and ordering over integers, is a natural reference to evaluate the expressiveness (and semantics) of such languages, because of its simplicity, decidability, and expressiveness, since it turns out that all sets definable in Presburger Arithmetics are *finite*, *periodic*, or *eventually periodic*. A recent comparison of the expressiveness of several constraint-based and symbolic approaches, based on Presburger Arithmetics, has been provided by Egidi and Terenziani [5].

While the above-mentioned approaches in [6, 9, 13] mainly focused on extending the relational model to cope with temporal periodicity, Kurt and Ozsoyoglu [7] devoted more attention to the definition of a *periodic temporal object oriented SQL*. They defined the temporal type of *periodic elements* to model strictly periodic and also eventually periodic events. Periodic elements consist of both an aperiodic part and a periodic part, represented by the repetition pattern and the period of repetition. They also defined the set-theoretic operations of union, intersection, complement and difference, which are closed with respect to periodic elements.

Moreover, in the last years, periodicity has also started to be studied in the context of moving objects. In particular, Revesz and Cai [11] have taken into account also periodic (called *cyclic* periodic) and eventually periodic (called *acyclic* periodic) *movements* of objects. They have proposed an extended relational data model in which objects are approximated by unions of *parametric rectangles*. Movement is coped with by modeling the $x$ and $y$ dimensions of rectangles through functions of the form $f(t \bmod p)$, where $t$ denotes time, $p$ the repetition period, and $\bmod$ the module function. Revesz and Cai have also defined the algebraic operations on the data model.

## KEY APPLICATIONS

Temporal periodicity plays an important role in many application areas, including process control, data access control, office automation, data broadcasting, planning, scheduling, multimedia, active databases, banking, and so on. Languages to specify user-defined periodicities in the *queries* have been already supported by many approaches, including commercial ones. For instance, Oracle provides a language to specify periodicity in the queries to *time series* (e.g., to ask for the values of IBM at the stock exchange *each Monday*). Even more interestingly, such languages can be used in the *data*, in order to express (finite and infinite) *periodic valid times* in an implicit and compact way. However, such a move involves an in-depth revision and extension of "standard" database theory and technology, which have been partly addressed within the temporal database research community, but have not been fully implemented in commercial products yet.

## CROSS REFERENCES

(other topics in the Encyclopedia which may be of interest to the reader of this entry)
Temporal granularity
Calendar
Calendric System
Anchor
Valid Time
Allen's Relations
Time Series

## RECOMMENDED READING


[1] Allen, J.F., Maintaining knowledge about temporal intervals, C*ommunications of the ACM 26(11)*, 832-843, 1983.

[2] Baudinet, M, Chomicki,J., and Wolper, P., Temporal Deductive Databases, in A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.) Temporal Databases, Benjamin/Cummings, 294-320, 1993.

[3] Bettini, C., and De Sibi, R., Symbolic Representation of user-defined time granularities, Annals of Mathematics and Artificial Intelligence 30(1-4), 53-92, 2000.

[4] Chomicki, J., and Imielinsky, T., "Temporal Deductive Databases and Infinite Objects", *Proc. seventh ACM Symp. Principles of Database Systems*, pp. 61-73, Austin, Texas, March 1988.

[5] Egidi, L., and Terenziani, P., A mathematical framework for the semantics of symbolic languages representing periodic time, Annals of Mathematics and Artificial Intelligence 46, 317-347, 2006.

[6] Kabanza, F., Stevenne, J.-M., and Wolper, P., Handling Infinite Temporal Data, *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 392-403, Nashville, Tennessee, April 1990.

[7] Kurt, A., and Ozsoyoglu, M., Modelling and Querying Periodic Temporal Databases, in *Workshop of the 6$^{th}$ International Conference on Database and Expert Systems Applications (DEXA)*, 124-133, 1995.

[8] Leban, B., McDonald, D.D., and Forster, D.R., A representation for collections of temporal intervals, *Proc. fifth National Conf. on Artificial Intelligence (AAAI),* pp. 367-371, Philadelphia, PA, August 1986.

[9] Niezette, M., and Stevenne, J.-M., An Efficient Symbolic Representation of Periodic Time, *Proc. first Int'l Conf. Information and Knowledge Management*, Baltimore, Maryland, Nov. 1992.

[10] Ning, P., Wang, X.S., and Jajodia, S., An algebraic representation of calendars, Annals of Mathematics and Artificial Intelligence 36(1-2) 5-38, 2002.
[11] P. Revesz P., and Cai M., Efficient Querying of Periodic Spatio-Temporal Databases, Annals of Mathematics and Artificial Intelligence 36(4), 437-457, 2002.
[12] Snodgrass, R.T., and Soo, M.D., Supporting multiple calendars, in Snodgrass, R.T. (ed.) The TSQL2 Temporal Query Language, Kluwer Academic Publisher, 103-121, 1995.
[13] Terenziani, P., Symbolic User-defined Periodicity in Temporal Relational Databases, IEEE Transactions on Knowledge and Data Engineering, 15(2), 489-509, 2003.
[14] Tuzhilin, A., and Clifford, J., On Periodicity in Temporal Databases, *Information Systems,* vol. 20, no.8, pp. 619-639, December 1995.

# TEMPORAL PROJECTION

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Temporal assignment

## DEFINITION

In a query or update statement, *temporal projection* pairs the computed facts with their associated times, usually derived from the associated times of the underlying facts.

The generic notion of temporal projection may be applied to various specific time dimensions. For example, *valid-time projection* associates with derived facts the times at which they are valid, usually based on the valid times of the underlying facts.

## MAIN TEXT

While almost all temporal query languages support temporal projection, the flexibility of that support varies greatly.

In some languages, temporal projection is implicit and is based the intersection of the times of the underlying facts. Other languages have special constructs to specify temporal projection.

The term "temporal projection" has been used extensively in the literature. It derives from the `retrieve` clause in Quel as well as the `SELECT` clause in SQL, which both serve the purpose of the relational algebra operator projection, in addition to allowing the specification of derived attribute values.

The related concept called "temporal assignment" is roughly speaking a function that maps a set of time values to a set of values of an attribute. One purpose of a temporal assignment would be to indicate when different values of the attribute are valid.

## CROSS REFERENCE*

SQL-Based Temporal Query Languages, Temporal Database, Temporal Query Language, TSQL2, Valid Time

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# Temporal Query Languages

Christian S. Jensen and Richard T. Snodgrass
University of Aalborg, Denmark and University of Arizona, USA

## SYNONYMS

historical query language

## DEFINITION

A temporal query language is a database query language that offers some form of built-in support for the querying and modification of time-referenced data, as well enabling the specification of assertions and constraints on such data. A temporal query language is usually quite closely associated with a temporal data model that defines the underlying data structures that the language applies to.

## HISTORICAL BACKGROUND

When the relational data model was proposed by Codd, he also proposed two query languages: relational calculus and relational algebra. Similarly, temporal data models are closely coupled with temporal query languages.

Most databases store time-referenced, or temporal, data. The ISO standard Structured Query Language SQL [4] is often the language of choice when developing applications that utilize the information captured in such databases. In spite of this, users realize that temporal data management is at times difficult with SQL. To understand some of the difficulties, it is instructive to attempt to formulate the following straightforward, realistic queries, assertions, and modifications in SQL. An intermediate SQL programmer can express all of them in SQL for a database without time-referenced data in perhaps five minutes. However, even SQL experts find these same queries challenging to do in several *hours* when the data is time-referenced [6].

An `Employee` table has three attributes: `Name`, `Manager`, and `Dept`. Temporal information is retained by adding a fourth attribute, `When`, of data type `PERIOD`. Attribute `Manager` is a foreign key for `Employee.Name`. This means that at each point in time, the `Manager` value of a tuple also occurs as a `Name` value (probably in a different tuple). This cannot be expressed via SQL's foreign-key constraint, which fails to take time into account. Formulating this constraint as an assertion is challenging.

- Consider the query "List those employees who are not managers." This can easily be expressed in SQL, using `EXCEPT` or `NOT EXISTS`, on the original, non-temporal relation with three attributes. Things are just a little harder with the `When` attribute; a `WHERE` predicate is required to extract the current employees. To formulate the query "List those employees who were not managers, and indicate when," `EXCEPT` and `NOT EXISTS` do not work because they do not consider time. This simple temporal query is hard to formulate, even for SQL experts.
- Consider the query "Give the number of employees in each department." Again, this is a simple SQL query using the `COUNT` aggregate when formulated on non-temporal data. To formulate the query on temporal data, i.e., "Give *the history of* the number of employees in each department," is extremely difficult without built-in temporal support in the language.
- The modification "Change the manager of the Tools department for 2004 to Bob" is difficult in SQL because only a portion of many validity periods are to be changed, with the information outside of 2004 being retained.

Most users know only too well that while SQL is an extremely powerful language for writing queries on the current state, the language provides much less help when writing temporal queries, modifications, and assertions

and constraints.
Hence there is a need for query languages that explicitly "understand" time and offer built-in support for the management of temporal data. Fortunately, the outcomes of several decades of research in temporal query languages demonstrate that it is indeed possible to build support for temporal data management into query languages so that statements such as the above are easily formulated.

## SCIENTIFIC FUNDAMENTALS

Structure may be applied to the plethora of temporal query languages by categorizing these according to different concerns.

### Language Extension Approaches

One attempt at bringing structure to the diverse collection of temporal query languages associates these languages with four approaches that emphasize how temporal support is being added to a non-temporal query language [1].

**Abstract Data Types for Time** From a design and implementation perspective, the simplest approach to improving the temporal data management capabilities of an existing query language is to introduce time data types with associated predicates and functions. This approach is common for query languages associated with temporal object-oriented databases.
Data types for time points, timer intervals (periods), and for durations of time may be envisioned, as may data types for temporal elements, i.e., finite unions of time intervals. The predicates associated with time-interval data types are often inspired by Allen's thirteen interval relationships. With reference to these, different sets of practical proposals for predicates have been proposed.
However, while being relatively easy to achieve, the introduction of appropriate time data types results only in modest improvements of temporal data management capabilities of the query language.

**Use of Point Timestamps** An interval timestamp associated with a tuple in a temporal relational data model is often intended to capture the fact that the information recorded by the tuple is valid at each time point contained in the interval. This way, the interval is simply a compact representation of a set of time points. Thus, the same information can be captured by a single tuple timestamped with an interval and a set of identical tuples, each timestamped with a different time point from the interval (with no time points missing).
One attraction of intervals is that they are of fixed size. Another is that they appear to be very intuitive to most users—the notion of an interval is conceptually very natural, and we use it frequently in our daily thinking and interactions. In some respects, the most straightforward and simplest means of capturing temporal aspects is to use interval-valued timestamps.
However, the observation has also been advanced that the difficulty in formulating temporal queries on relations with interval-timestamped tuples is exactly the intervals—Allen has shown that there are thirteen possible relations between a pair of intervals. It has been argued that a language such as SQL is unprepared to support something (an interval) that represent something (a set of consecutive time points) that it is not.
Based on this view, it is reasonable to equip an SQL extended with interval-valued timestamps with the ability to *unfold* and *fold* timestamps. The unfold function maps an interval-stamped tuple (or set of tuples) to the corresponding set of point-stamped tuples (set of tuples), and the fold function collapses a set of point-stamped tuples in the corresponding interval-stamped tuple(s). This way, it is possible to manipulate both point- and interval-stamped relations in the same language. If deemed advantageous when formulating a statement, one can effectively avoid the intervals by first unfolding all argument relations. Then the statement is formulated on the point-stamped relations. At the end, the result can be folded back into an interval-stamped format that lends itself to presentation to humans.
A more radical approach to designing a temporal query language is to completely abandon interval timestamps and use only point timestamps. This yields a very clean and simple design, although it appears that database modification and the presentation of query results to humans must still rely on intervals and thus are "outside" the approach.
The strength of this approach lies in its generalization of queries on non-temporal relations to corresponding queries

on corresponding temporal relations. The idea is to extend the non-temporal query with equality constraints on the timestamp attribute of the temporal relation, to separate different temporal database states during query evaluation.

**Syntactical Defaults** This approach introduces what may be termed syntactic defaults along with the introduction of temporal abstract data types, the purpose being to make the formulation of common temporal queries more convenient. Common defaults concern the access to the current state of a temporal relation and the handling of temporal generalizations of common non-temporal queries, e.g., joins. The temporal generalization of a non-temporal join is one where two tuples join if their timestamps intersect and where the timestamp of the result tuple is the intersection of the timestamps. Essentially, the non-temporal query is computed for each point in time, and the results of these queries are consolidated into a single result. The nature of the consolidation depends on the data type of the timestamps; if intervals are used, the consolidation involves coalescing.
The most comprehensive approach based on syntactic defaults is the TSQL2 language [5] and many of the earlier query languages that the creators of this language were attempting to consolidate. As an example, TSQL2 includes a default valid clause in statements that computes the intersection of the valid times of the tuples in the argument relations mentioned in the statement's from clause, which is then returned in the result. So as explained above, the timestamp of a tuple that results from joining two relations is the intersection of the timestamps of the two argument tuples that produce the tuple. When there is only one argument relation, the valid clause produces the original timestamps.
The introduction of such defaults are very effective in enabling the concise formulation of common queries, but they also tend to complicate the semantics of the resulting temporal query language.

**Semantic Defaults** The use of semantic defaults is motivated in part by the difficulties in systematically extending a large and unorthogonal language such as SQL with syntactic defaults that are easy to understand and that do not interact in unintended ways. With this approach, so-called *statement modifiers* are added to a non-temporal query language, e.g., SQL, in order to obtain built-in temporal support.
It was argued earlier that statements that are easily formulated in SQL on non-temporal relations are very difficult to formulate on temporal relations. The basic idea is then to make it easy to systematically formulate temporal queries from non-temporal queries. With statement modifiers, a temporal query is then formulated by first formulating the "corresponding" non-temporal query (i.e., assuming that there are no timestamp attributes on the argument relations) and then applying a statement modifier to this query.
For example, to formulate a temporal join the first step is to formulate the corresponding non-temporal join. Next, a modifier is placed in front of this query to indicate that the non-temporal query is to be rendered temporal by computing it at each time point. The modifier ensures that the argument timestamps overlap and that the resulting timestamp is the intersection of the argument intervals. The attraction of using statement modifiers is that these may be placed in front of any non-temporal query to render that query temporal.
Statement modifiers are capable of specifying the semantics of the temporal queries unambiguously, independently of the syntactic complexity of the queries that the modifiers are applied to. This renders semantic defaults scalable across the constructs of the language being extended. With modifiers, the users need thus not worry about which predicates are needed on timestamps and how to express timestamps to be associated with result tuples. Further, the use of statement modifiers makes it possible to give more meaning to interval timestamps; they need no longer be simply compact representations of convex sets of time points.

## Additional Characterizations of Temporal Query Languages

Several additional dimensions exist on which temporal query languages can be characterized. One is *abstractness*: is the query language at an abstract level or at a concrete level. Examples of the former are Temporal Relational Calculus and First-order Temporal Logic; an example of the latter is TSQL2. See the entries on Abstract Versus Concrete Temporal Query Languages and Temporal Logic in Database Query Languages.
Another dimension is *level*: is the query language at a logical or a physical level? A physical query language assumes a specific representation whereas a logical query language admits several representations. Examples of the former are examined in McKenzie's survey of relational algebras [3]; an example of the latter is the collection of algebraic operators defined on the Bitemporal Conceptual Data Model [2], which can be mapped to at least

five representational models.
A third dimension is whether the query language supports a *period-stamped temporal model* or a *point-stamped temporal model*.
Other entries (indicated in *italics*) examine the long and deep research into temporal query languages in a more detailed fashion. *Qualitative temporal reasoning* and *temporal logic in database query languages* provide expressive query facilities. *Temporal vacuuming* provides a way to control the growth of a database. *TSQL2* and its successor SQL/Temporal provided a way for many in the temporal database community to coordinate their efforts in temporal query language design and implementation. *Temporal query processing* involves disparate architectures, from *temporal strata* outside the conventional DBMS to adding native temporal support within the DBMS. *Supporting transaction time* generally requires changes within the kernel of a DBMS. *Temporal algebras* extend the conventional relational algebra. Some specific operators (e.g., *temporal aggregation*, *temporal coalescing*, *temporal joins*)) have received special attention. Finally, the Oracle database management system includes support for valid and transaction time, both individually and in concert.

## FUTURE DIRECTIONS

Given the substantial decrease in code size (a factor of three [6]) and dramatic decrease in conceptual complexity of temporal applications that temporal query languages offer, it is hoped that DBMS vendors will continue to incorporate temporal language constructs into their products.

## CROSS REFERENCE*

Abstract Versus Concrete Temporal Query Languages, Allen's Relations, Now in Temporal Databases, Period-Stamped Temporal Models, Point-Stamped Temporal Models, Qualitative Temporal Reasoning, Schema Versioning, Supporting Transaction Time, Temporal Aggregation, Temporal Algebras, Temporal Coalescing, Temporal Data Models, Temporal Joins, Temporal Logic in Database Query Languages, Temporal Object-Oriented Databases, Temporal Query Processing, Temporal Strata, Temporal Upward Compatibility, Temporal Vacuuming, Temporal XML, TSQL2

## RECOMMENDED READING


[1] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen, "How Would You Like to Aggregate Your Temporal Data?," *Proceedings of the 13th International Symposium on Temporal Representation and Reasoning*, Budapest, Hungary, pp. 121-136, June 15-17, 2006.

[2] Christian S. Jensen, Michael D. Soo and Richard T. Snodgrass, "Unifying Temporal Data Models via a Conceptual Model," *Information Systems*, Vol. 19, No. 7, December 1994, pp. 513–547.

[3] Edwin McKenzie and Richard T. Snodgrass, "An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases," *ACM Computing Surveys*, Vol. 23, No. 4, December 1991, pp. 501–543.

[4] Jim Melton and Alan R. Simon, **Understanding the New SQL: A Complete Guide**. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[5] R. T Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. **The TSQL2 Temporal Query Language**. Kluwer Academic Publishers 1995.

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

# Temporal Query Processing

Michael Böhlen, Free University of Bozen-Bolzano, Italy

## SYNONYMS

none

## DEFINITION

Temporal query processing refers to the techniques used by database management system to process temporal statements. This ranges from the implementation of query execution plans to the design of system architectures. This entry surveys different system architectures. It is possible to identify three general system architectures that have been used to systematically offer temporal query processing functionality to applications [7]: The *layered* approach uses an off-the-shelf database system and extends it by implementing the missing functionality in a layer between the database system and the applications. The *monolithic* approach integrates the necessary application-specific extensions directly into the database system. The *extensible* approach relies on a database system that allows to plug user-defined extensions into the database system.

## HISTORICAL BACKGROUND

In order to deploy systems that offer support for temporal query processing new systems must be designed and implemented. Temporal extensions of database systems are quite different from other database system extensions. On the one hand, the complexity of the newly added types, often an interval, is usually quite low. Therefore, existing access structures and query processing techniques are often deemed sufficient to manage temporal information [8, 9, 12]. On the other hand the temporal aspect is ubiquitous and affects all parts of a database system. This is quite different if extensions for multimedia, geographical or spatio-temporal data are considered. Such extensions deal with complex objects, for example, objects with complex boundaries, moving points, or movies, but their impact on the various components of a database system is limited.

A first overview of temporal database system implementations appeared in 1996 [2]. While the system architecture was not the focus, the overview describes the approached followed by the systems. Most systems follow the layered approach, including ChronoLog, ARCADIA, TimeDB, VT-SQL, and Tiger. A comprehensive study of a layered query processing architecture was done by Slivinskas et al [11]. The authors use the Volcano extensible query optimizer to optimize and process queries. A monolithic approach is pursued by HDBMS, TDBMS, T-REQUIEM, and T-squared DBMS. A successful implementation of an index structure for temporal data has been done with the help of Informix's datablade technology [1],

## SCIENTIFIC FUNDAMENTALS

Figure 1 provides an overview of the different architectures that have been used to systematically provide temporal database functionality: the layered, monolithic, and extensible architectures. The gray parts denote the temporal extensions. The architecture balances initial investments, functionality, performance and lock-ins.

**Functionalities** The different operators in a temporal database system have different characteristics with respect to query processing.

Temporal selection, temporal projection and temporal union resemble their non-temporal counterparts and they do not require dedicated new database functionality. Note though that this only holds if the operators may return uncoalesced relation instances [3] and if no special values, such as *now*, are required. If the operators must returned coalesced relations or if *now* must be supported the support offered by conventional database systems is lacking.

Temporal Cartesian product (and temporal joins) are also well supported by standard database systems. In many cases existing index structures can be used to index interval timestamps [8].

Temporal difference is more subtle and not well-supported by traditional database systems. It is possible to formulate temporal difference in, e.g., SQL, but it is cumbersome to do so. Algebraic formulations and efficient

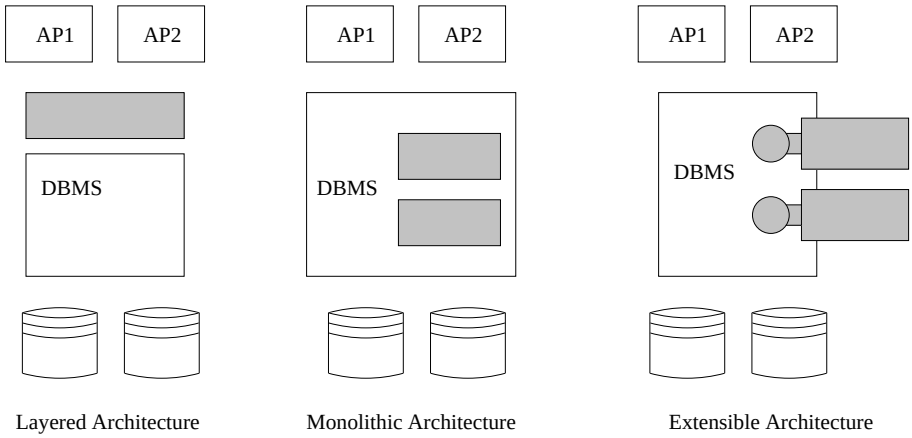



Figure 1: Illustration of Architectural Choices for Temporal Database Systems

implementations have been studied by Dunn et al. [4].
Temporal aggregation is even worse than temporal difference. Formulating a temporal aggregation in SQL is a challenge even for advanced SQL programmers and yields a statement that current database systems cannot evaluate efficiently [12].
Temporal coalescing and temporal integrity constraints [12] are other examples of operations that current database system do not handle efficiently. Temporal coalescing is an algebraic operator and a declarative SQL implementation is inefficient.

**The Layered Architecture** A common approach to design an extended database systems with new data types and operations for time-referenced data is to use an off-the-shelf database system and implement a layer on top providing data types and services for temporal applications. The database system with such a component is then used by different applications having similar data type and operation requirements. Database systems enhanced in this way exploit the standard data types and data model, often the relational model, as a basis. They define new data types and possibly a new layer that provides application specific support for data definition and query language, query processing and optimization, indexing, and transaction management. Applications are written against the extended interface.
The layered approach has the advantage of using standard components. There is a clear separation of responsibilities: application-specific development can be performed and supported independent of the database system development. Improvements in the database system component are directly available in the whole system with almost no additional effort. On the other hand, the flexibility is limited. Development not foreseen in the database system component has to be implemented bypassing the database system. The more effort is put into such an application-specific data management extension, the more difficult it gets to change the system and take advantage of database system improvements. Also (legacy) applications might access the database system over different interfaces. For such applications accessing the extended database system through a special purpose layer is not always an option.
The reuse of a standard database system is an advantage but at the same time also a constraint. The layer translates and delegates temporal requests to sequences of nontemporal request. If some of the functionality of the database systems should be extended or changed, e.g., a refined transaction processing for transaction time, this cannot be done easily with a layered approach. For the advanced temporal functionality the layered architecture might not offer satisfactory performance.

**The Monolithic Architecture** Many systems that use a monolithic architecture have originally been designed as stand-alone applications without database functionality. The designers of a monolithic architecture then extend their system with database system functionality. They add query functionality, transaction management, and multi-user capabilities, thereby gradually creating a specialized database system. The data management aspects

traditionally associated with database system and the application-specific functionality are integrated into one component.
Instead of adding general database system functionality to an application it is also possible to incorporate the desired application domain semantics into the database system. Typically, this is done by database companies who have complete control over and knowledge of their source code or by open source communities.
Because of the tight integration of the general data management aspects and the application specific functionality, monolithic systems can be optimized for the specific application domain. This results in good performance. Standard and specialized index structures can be combined for good results. Transaction management can be provided in a uniform way for standard as well as new data types. However, implementing a monolithic system is difficult and a big (initial) effort, since all aspects of a database system have to be taken into account. Another drawback is that enterprises tend to be reluctant to replace their database system, which increases the threshold for the adoption of the temporal functionality. With monolithic systems there is is a high risk of vendor lock-ins.

**The Extensible Architecture** Extensible database systems can be extended with application-specific modules. Traditional database functionality like indexing, query optimization, and transaction management is supported for new data types and functions in a seamless fashion.
The first extensible system prototypes have been developed to support non-standard database system applications like geographical, multimedia or engineering information systems. Research on extensible systems has been carried out in several projects, e.g., Ingres [13], Postgres[5], and Volcano [6]. These projects addressed, among other, data model extensions, storage and indexing of complex objects as well as transaction management and query optimization in the presence of complex objects. Today a number of commercial approaches are available, e.g., datablades from Informix, cartridges from Oracle, and extenders from DB2. A limitation is that the extensions only permit extension that were foreseen initially. A comprehensive temporal support might require support that goes beyond data types and access structures.
The SQL99 standard [10] specifies new data types and type constructors in order to better support advanced applications.
The extensible architecture balances the advantages and disadvantages of the layered and monolithic architectures, respectively. There is a better potential to implement advanced temporal functionality with a satisfactory performance than in the layered architecture. However, functionality not foreseen by the extensible database system might still be difficult ti implement.

## KEY APPLICATIONS

All applications that want to provide systematic support for time-varying information must choose one of the basic system architectures. The chosen architecture balances (initial) investments, future lock-ins, and performance.

## FUTURE DIRECTIONS

The architecture determines the initially required effort to provide support for time-varying information and may limit functionality and performance. Changing from one architecture to another is not supported. Such transitions would be important to support the graceful evolution of temporal database applications.

## CROSS REFERENCE

Temporal Data Model, Temporal Database, Temporal Strata

## RECOMMENDED READING

~~Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.~~


[1] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Developing a datablade for a new index. In *ICDE*, pages 314–323, 1999.
[2] M. H. Böhlen. Temporal Database System Implementations. *SIGMOD Record*, 24(4):16, December 1995.
[3] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Mumbai (Bombay), India, September 1996.
[4] J. Dunn, S. Davey, A. Descour, and R. T. Snodgrass. Sequenced subset operators: Definition and implementation. In *ICDE*, pages 81–92, 2002.
[5] The Postgresql Global. Postgresql developer's guide.
[6] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[7] M. Koubarakis, T. K. Sellis, A. U. Frank, S. Grumbach, R. H. Güting, C. S. Jensen, N. A. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona, editors. *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, volume 2520 of *Lecture Notes in Computer Science*. Springer, 2003.

[8] H-P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, pages 407–418, 2000.

[9] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 14, pages 329–355. Benjamin/Cummings Publishing Company, 1993.

[10] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, 1993.

[11] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable query optimization and evaluation in temporal middleware. In *SIGMOD Conference*, pages 127–138, 2001.

[12] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[13] M. Stonebraker, editor. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley, 1986.

# Temporal Relational Calculus

Jan Chomicki, University at Buffalo, USA, http://www.cse.buffalo.edu/~chomicki
David Toman, University of Waterloo, Canada, http://www.cs.uwaterloo.ca/~david

## SYNONYMS

two-sorted first-order logic

## DEFINITION

*Temporal Relational Calculus (TRC)* is a temporal query language extending the relational calculus. In addition to data variables and quantifiers ranging over a data domain (a universe of *uninterpreted constants*), temporal relational calculus allows *temporal variables* and quantifiers ranging over an appropriate time domain [1].

## MAIN TEXT

A natural temporal extension of the relational calculus allows explicit variables and quantification over a given time domain, in addition to the variables and quantifiers over a data domain of uninterpreted constants. The language is simply the two-sorted version (variables and constants are temporal or non-temporal) of first-order logic over a data domain $D$ and a time domain $T$.

The syntax of the two-sorted first-order language over a database schema $\rho = \{R_1, \ldots, R_k\}$ is defined by the grammar rule:

$$Q ::= R(t_i, x_{i_1}, \ldots, x_{i_k}) \mid t_i < t_j \mid x_i = x_j \mid Q \wedge Q \mid \neg Q \mid \exists x_i.Q \mid \exists t_i.Q$$

In the grammar, $t_i$'s are used to denote temporal variables and $x_i$'s to denote data (non-temporal) variables. The atomic formulae $t_i < t_j$ provide means to refer to the underlying ordering of the time domain. Note that the schema $\rho$ contains schemas of *timestamped temporal relations* (see the entry Point-stamped Temporal Models).

Given a point-timestamped database $DB$ and a two-sorted valuation $\theta$, the semantics of a TRC query $Q$ is defined in the standard way (similarly to the semantics of relational calculus) using the *satisfaction relation* $DB, \theta \models Q$:

$$\begin{array}{ll}
DB, \theta \models R_j(t_i, x_{i_1}, \ldots, x_{i_k}) & \text{if } R_j \in \rho \text{ and } (\theta(t_i), \theta(x_{i_1}), \ldots, \theta(x_{i_k})) \in R_j^{DB} \\
DB, \theta \models t_i < t_j & \text{if } \theta(t_i) < \theta(t_j) \\
DB, \theta \models x_i = x_j & \text{if } \theta(x_i) = \theta(x_j) \\
DB, \theta \models Q_1 \wedge Q_2 & \text{if } DB, \theta \models Q_1 \text{ and } DB, \theta \models Q_2 \\
DB, \theta \models \neg Q_1 & \text{if not } DB, \theta \models Q_1 \\
DB, \theta \models \exists t_i.Q_1 & \text{if there is } s \in T \text{ such that } DB, \theta[t_i \mapsto s] \models Q_1 \\
DB, \theta \models \exists x_i.Q_1 & \text{if there is } a \in D \text{ such that } DB, \theta[x_i \mapsto a] \models Q_1
\end{array}$$

where $R_j^{DB}$ is the interpretation of the predicate symbol $R_j$ in the database $DB$.

The *answer to a query $Q$ over $DB$* is the set $Q(DB)$ of valuations that make $Q$ true in $DB$. Namely, $Q(DB) := \{\theta_{|FV(Q)} : DB, \theta \models Q\}$ where $\theta_{|FV(Q)}$ is the restriction of the valuation $\theta$ to the free variables of $Q$.

In many cases, the definition of TRC imposes additional restrictions on valid TRC queries:

**Restrictions on free variables:** often the number of free temporal variables in TRC queries can be restricted to guarantee closure over the underlying data model (e.g., a single-dimensional timestamp data model or the bitemporal model). Note that this restriction applies only to queries, not to subformulas of queries.

**Range restrictions:** another common restriction is to require queries to be range restricted to guarantee domain independence. In the case of TRC (and many other abstract query languages), these restrictions depend crucially on the chosen *concrete encoding* of temporal databases (see the entry Abstract and Concrete Temporal Query Languages). For example, no range restrictions are needed for temporal variables when

queries are evaluated over interval-based database encodings, because the complement of an interval can be finitely represented by intervals.

The schemas of atomic relations, $R_j(t_i, x_{i_1}, \ldots, x_{i_k})$, typically contain a single temporal attribute/variable, often in fixed (e.g., first) position: this arrangement simply reflects the choice of the underlying temporal data model to be the single-dimensional valid time model. However, TRC can be similarly defined for multidimensional temporal data models (such as the bitemporal model) or for models without a predefined number of temporal attributes by appropriately modifying or relaxing the requirements on the structure of relation schemas (see the entry Point-stamped Temporal Models).

An interesting observation is that a variant of TRC, in which temporal variables range over *intervals* and that utilizes *Allen's interval relations* as basic comparisons between interval values, is equivalent to TRC over two-dimensional temporal relations, with the two temporal attributes standing for interval endpoints.

## CROSS REFERENCE

abstract and concrete temporal query languages, point-stamped temporal data models, relational calculus, relational model, temporal logic in query languages, temporal query languages, temporal relation, time domain, time instant, TSQL2, valid time.

## REFERENCES

[1] J. Chomicki and D. Toman. Temporal Databases. In M. Fischer, D. Gabbay, and L. Villa, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier *Foundations of Artificial Intelligence*, 2005.

# TEMPORAL SPECIALIZATION

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Temporal restriction

## DEFINITION

Temporal specialization denotes the restriction of the interrelationships between otherwise independent (implicit or explicit) timestamps in temporal relations. An example is a relation where tuples are always inserted after the facts they record were valid in reality. In such a relation, the transaction time of a tuple would always be after the valid time. Temporal specialization may be applied to relation schemas, relation instances, and individual tuples.

## MAIN TEXT

Data models exist where relations are required to be specialized, and temporal specializations often constitute important semantics about temporal relations that may be utilized for, e.g., improving the efficiency of query processing.

Temporal specialization encompasses kinds of specialization: one is based on the relationships between isolated events, and one based on inter-event relationships; two additional kinds consider intervals instead of events; and one is based on the so-called completeness of the capture of the past database states.

The taxonomy based on isolated events, illustrated in Figure 1, considers the relationship between a single valid time and a single transaction time. For example, in a *retroactive* relation, an item is valid before it is operated on (inserted, deleted, or modified) in the database. In a *degenerate* relation, there is no time delay between sampling a value and storing it in the database: the valid and transaction timestamps for the value are identical.

The interevent-based taxonomy is based on the interrelationships of multiple event timestamped items, and includes non-decreasing, non-increasing, and sequential. Regularity is captured through the categories of transaction time event regular, valid time event regular, and temporal event regular, and strict versions of these. The interinterval-based taxonomy uses Allen's relations.

To understand the last kind of specialization, which concerns the completeness of the capture of past states, recall that a standard transaction-time database captures all previously current states—each time a database modification occurs, a new previously current state is created. In contrast a valid-time database captures only the current database state. In-between these extremes, one may envision a spectrum of databases with incomplete support for transaction time. For example, this would occur if a web archive was to take a snapshot of a collection of web sites at regular intervals, e.g., every week. If a site was updated several times during the same week, states would be missing from the database. Such incomplete databases are considered specializations of more complete ones.

Concerning the synonym, the chosen term is more widely used than the alternative term. The chosen term indicates that specialization is done with respect to the temporal aspects of the data items being timestamped. It is natural to apply the term temporal generalization to the opposite of temporal specialization. "Temporal restriction" has no obvious opposite term.

## CROSS REFERENCE*

Allen's Relations, Bitemporal Relation, Temporal Database, Temporal Generalization, Transaction Time, Valid Time

## REFERENCES*

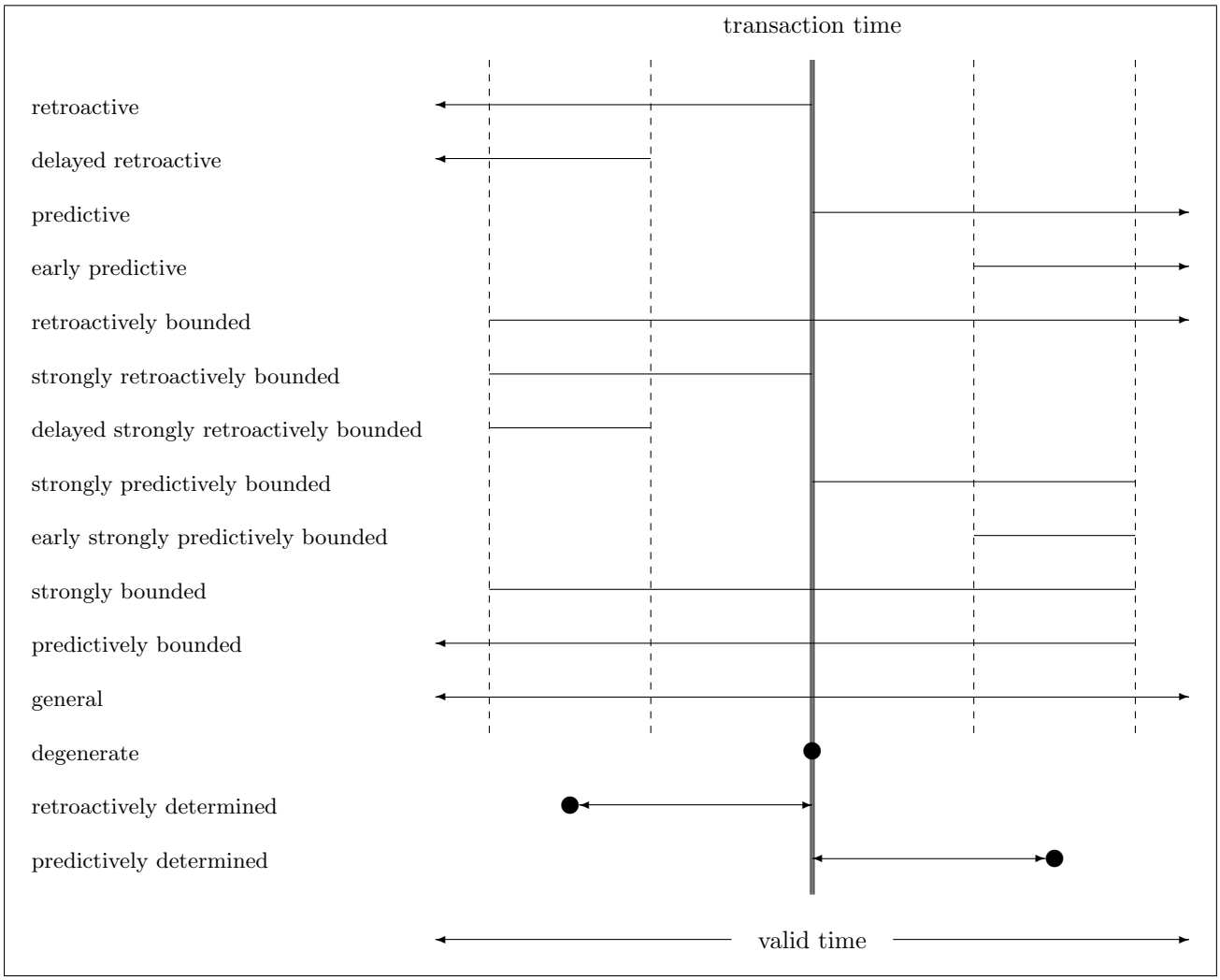



Figure 1: Temporal Specialization based on isolated events—restrictions on the valid timestamp relative to the transaction timestamp. Adapted from Jensen and Snodgrass (1994).

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

C. S. Jensen and R. T. Snodgrass, "Specialized Temporal Relations," in *Proceedings of the IEEE International Conference on Data Engineering*, Phoenix, AZ, February 1992, pp. 594–603.

C. S. Jensen and R. T. Snodgrass, "Temporal Specialization and Generalization," *IEEE Transactions on Knowledge and Data Engineering* 5(6):954–974, December 1994.

# Temporal Strata

Kristian Torp
Department of Computer Science
Aalborg University
Aalborg, Denmark
torp@cs.aau.dk

## SYNONYMS

temporal layer; layered architecture; temporal middleware; wrapper

## DEFINITION

A temporal stratum is an architecture for implementing a temporal DBMS. The stratum is a software layer that sits on top of an existing DBMS. The layer translates a query written in a temporal query language into one or more queries in a conventional query language (typically SQL). The translated queries can then be executed by the underlying DBMS. The DBMS returns the result of the query/queries to the user directly or via the stratum. The core idea of the stratum is to provide new temporal query functionality to the users without changing the underlying DBMS. A temporal stratum can be implemented as a simple translator (temporal SQL to standard SQL) or as an advanced software component that also does part of the query processing and optimization. In the latter case, the temporal stratum can implement query processing algorithms that take the special nature of temporal data into consideration. Examples are algorithms for temporal join, temporal coalescing, and temporal aggregation.

## HISTORICAL BACKGROUND

Applications that store and query multiple versions of data have existed for a very long. These applications have been implemented using for example triggers and log-like tables. Supporting multiple versions of data can be time consuming to build and computationally intensive to execute, since the major DBMS vendors only have limited temporal support. See [7] for details.

Parallel to the work in the software industry the research community has proposed a large number of temporal data models and temporal query languages. For an overview see [6]. However, most of this research is not supported by implementations. A notable exception to this is the Postgres DBMS that has built-in support for transaction time [8]. Postgres has later evolved into the popular open-source DBMS PostgreSQL that does not have transaction-time support. Most recently, the Immortal DB research prototype [2] has looked at how to built-in temporal support in the Microsoft SQL Server DBMS.

The DBMS vendors have not been interested in implementing the major changes to the core of their DBMSs that are needed to add temporal support to the existing DBMSs. In addition, the proposal to add temporal support to SQL3, called "SQL/Temporal" part 7 of the ISO SQL3 standard, has had very limited support. The temporal database research community has therefore been faced with a challenge of how to experimentally validate their proposals for new temporal data models and temporal query languages since it is a daunting task to build a temporal DBMS from scratch.

To meet this challenge it has been proposed to implement a temporal DBMS as a software layer on top of an existing DBMS. This has been termed a temporal stratum approach. Some of the first proposals for a temporal stratum mainly consider translating a query in a temporal query language to one or more queries in SQL [9].

The temporal database research community has shown that some temporal queries are very inefficient to execute in plain SQL, either formulated directly in SQL or translated via a simple temporal stratum from a temporal

| name | dept | VTS | VTE | TTS | TTE |
|---|---|---|---|---|---|
| Jim | NY | 3 | *now* | 3 | *uc* |
| Joe | LA | 4 | *now* | 4 | 11 |
| Joe | LA | 4 | 11 | 11 | *uc* |
| Joe | UK | 11 | *now* | 11 | *uc* |
| Sam | NY | 12 | *now* | 12 | 14 |
| Sam | NY | 12 | 14 | 14 | *uc* |

A

```
create table emp(
   name varchar2(30) not null,
   dept varchar2(30) not null,
   vts  date         not null,
   vte  date         not null,
   tts  date         not null,
   tte  date         not null)
```

B

Figure 1: (A) The Bitemporal Table `emp` (B) SQL Implementation

```
-- at time 4
insert into emp value('Joe', 'LA')
```

A

```
insert into emp values
  ('Joe', 'LA',
   '2007-09-04', '9999-12-31',
   '2007-09-04', '9999-12-31');
```

B

Figure 2: (A) Temporal Insert (B) Mapping to SQL

query language to SQL. For this reason it has be researched how to make the temporal stratum approach more advanced such that it uses the underlying DBMS when this is efficient and does the query execution in the layer when the underlying DBMS is found to be inefficient [4, 5].

## SCIENTIFIC FUNDAMENTALS

A bitemporal database supports both valid time and transaction time. In the following it is assume that all tables have bitemporal support. Figure 1A shows the bitemporal table `emp`. The table has two explicit columns, `name` and `dept`, and four implicit timestamp columns: `VTS`, `VTE`, `TTS`, and `TTE`. The first row in the table says that Jim was in the New York department from the 3rd (assuming the month of September 2007) and is still there, indicated by the variable *now*. This information was also entered on the 3rd and is still considered to be the best valid information, indicated by the variable *uc* that means until changed.

It is straightforward to implement a bitemporal table in a conventional DBMS. The implicit attributes are simply made explicit. For the `emp` table an SQL table with six columns are created. This is shown in Figure 1B. Existing DBMSs do not support variables and therefore the temporal stratum has to convert the variables *now* and *uc* to values in the domain of columns `VTE` and `TTE`. It has been shown that it is the most convenient to use the maximum value (9999-12-31) in the `date` domain for both *now* and *uc* [10].

Primary key and unique key constraints are quite complicated to implement in a temporal stratum. A temporal query language such as ATSQL [1] has to be temporal upwards compatible, i.e., all non-temporal SQL statements have to work as before. For primary keys this means that a primary key on a bitemporal table cannot be directly mapped to a primary key in the underlying DBMS. As an example, if the `emp` only has to store the current version of where employees are, the `name` column can be used as a primary key. However, since `emp` has bitemporal support there are now three rows in the example table where the name is 'Joe'. Due to space constraints, an example is do not list here, please see [7] for a concrete example.

To look at how modifications are handled in a temporal stratum first consider the temporal insert statement shown in Figure 2A. This is a temporal upward compatible insert statement (it looks like a standard SQL statement). The mapping to SQL is shown in Figure 2B. The columns `vts` and `tts` are set to the current date (3rd of September 2007). The *now* and *uc* variables are set to the maximum `date`. The result is the second row in Figure 1A.

At time 11 Joe is updated from being in the LA department to the UK department. The temporal upwards compatible update statement for this is shown in Figure 3A. This update statement is mapped to an SQL update

```
-- at time 11
update set dept = 'UK'
where name = 'Joe'
```

A

```
-- stop the old row
update emp set
  vte = '2007-09-11'
where vts <= '2007-09-11'
and   vte > '2007-09-11'
and   tte = '9999-12-31'
and   name = 'Joe';
-- insert knowledge about the past
insert into emp values
  ('Joe', 'LA', '2007-09-04', '2007-09-11',
                '2007-09-11', '9999-12-31');
-- insert new knowledge
insert into emp values
  ('Joe', 'UK', '2007-09-11', '9999-12-31',
                '2007-09-11', '9999-12-31');
```

B

Figure 3: (A) Temporal Update (B) Mapping to SQL

```
sequenced select *
from emp
as of '2007-09-13';
```

A

```
select name, dept, vts, vte
from   emp
where  tts <= '2007-09-13'
and    tte >  '2007-09-13'
```

B

| Name | Dept | VTS | VTE |
|---|---|---|---|
| Jim | NY | 3 | *now* |
| Joe | LA | 4 | 11 |
| Joe | UK | 11 | *now* |
| Sam | NY | 12 | *now* |

C

Figure 4: Transaction-Time Slicing

of the existing row and two SQL insert statements. The update ends the current belief by updating the `TTE` column to the current date. The first SQL insert statement stores for how long it was believed that Joe was in the LA department. The second SQL insert statement stores the new belief that Joe is in the UK department. The temporal update corresponds to the updated second row plus the third and fourth row in Figure 1A. A delete statement is mapped like the SQL update statement and the first SQL insert statement in Figure 3B. In Figure 1A a temporal insert of Sam at time 12 and a temporal delete of Sam at time 14 are shown as rows number five and six. Note that the SQL delete statement is never used for mapping temporal modification statements. For a complete coverage of implementing temporal modification statements in a temporal stratum, please see [10].

It is possible to see past states of the database. In particular the following will look at the `emp` table as of the 13th. This is called a time slicing, i.e., the database is rewound to 13th to see the content of the database as of this date.

The ATSQL query in Figure 4A is a sequenced query that selects the explicit attribute and the valid-time attributes (`vts` and `vte`). The equivalent SQL query translated by a temporal stratum is shown in Figure 4B. The result of the query is shown in Figure 4C (using *now* instead of the maximum date). The temporal SQL query is only slightly simpler than the equivalent standard SQL query.

To see the benefits of a temporal stratum it is necessary to look at more complicated queries. In the following the focus is on temporal aggregation queries. Alternative complicated queries are temporal join or temporal coalescing.

Assume that a boss wants to see how many employees a company has had over (valid) time looking at the database as of the 13th. The result shown in Figure 4C is used as an input for this query. For convenience it is assumed that this timeslice query is converted to a view call `empAt13`.

The content of the database is illustrated in Figure 5 where the vertical dotted lines indicates the time where the result has to be split. The temporal query that expresses this is shown in Figure 6A and the result of the query is shown in Figure 6B. Note that the result is not coalesced. For a discussion of the *sequenced validtime* clause

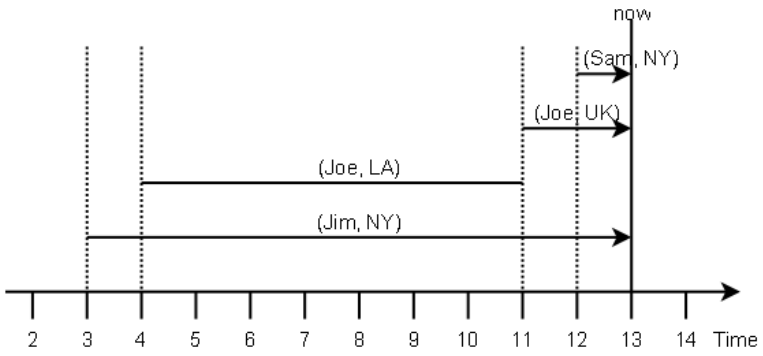



Figure 5: Visual Representation of the Content in Figure 4C

```
sequenced validtime
select count(Name)
from   emps
```

A

| Count | VTS | VTE |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 4 | 11 |
| 2 | 11 | 12 |
| 3 | 12 | *now* |

B

Figure 6: Temporal Aggregation

please see the entry Temporal Query Languages in this encyclopedia.
To execute this query the temporal stratum has to find the constant periods, i.e., the periods where the count is the same. Here the temporal stratum can do either direct conversion to standard SQL or do part of the query processing in the stratum. The direct converted query is listed in Figure 7. As can be seen the standard SQL is very complicated compared to the equivalent temporal SQL query in Figure 6A. The benefit for the user should be obvious. In line 1 of Figure 7 the count and the valid-time start and end associated with the count are selected. In line 2 the view `empat13` from Figure 4B is used. In addition, the `const_period` is introduced in lines 2 to 39. In line 40 only those periods that overlap the group currently being considered are included. In line 41 the groups are formed based on the valid-time start and valid-time end. Finally, in line 42 the output is listed in the valid-time start order.
The next question is then the efficiency of executing the query in the underlying DBMS or doing part of the query processing in the temporal stratum. It has been experimentally shown that for temporal aggregation it can be up to ten times more efficient to do part of the query processing in a temporal stratum [4].
A different approach to adding temporal support to an existing DBMS is to use an extensible DBMS such as IBM Informix, Oracle, or DB2. This is the approach taken in [11]. Here temporal support is added to IBM Informix. Compared to a stratum approach it is not possible in the extension approach to use a new temporal SQL. The temporal extension are accessed by the user via new operators or function calls.

## KEY APPLICATIONS

There is no support for valid-time or transaction-time in existing DBMSs. However, in Oracle 10g there is a flashback option [3] that allows a user to see the state of the entire database or a single table as of a previous instance in time. As an example, the flashback query in Figure 8 will get the state of the `emp` table as of the 1st of September 2007 at 08.00 in the morning. The result can be used for further querying.

## FUTURE DIRECTIONS

The Sarbanes-Oxley Act is a US federal law that requires companies to retain all of there data for a period of five or more years. This law may spur additional research with temporal databases and in particular temporal

```
select count(name), const_period.vts as vts, const_period.vte as vte
from empat13, (select t1.vts as vts, t1.vte as vte
               from   empat13 t1
               where  not exists ( select *
                      from   empat13 t2
                      where (t1.vts < t2.vts and t2.vts < t1.vte) or
                            (t1.vts < t2.vte and t2.vte < t1.vte))
               union
               select t1.vts as vts, t2.vts as vte
               from   empat13 t1, empat13 t2
               where  t1.vts < t2.vts and t2.vts < t1.vte
               and    not exists (select *
                      from   empat13 t3
                      where  (t1.vts < t3.vts and t3.vts < t2.vts) or
                             (t1.vts < t3.vte and t3.vte < t2.vts))
               union
               select t1.vts as vts, t2.vte as vte
               from   empat13 t1, empat13 t2
               where  t1.vts < t2.vte and t2.vte < t1.vte
               and    not exists (select *
                      from   empat13 t3
                      where  (t1.vts < t3.vts and t3.vts < t2.vte) or
                             (t1.vts < t3.vte and t3.vte < t2.vte))
             union
             select t1.vte as vts, t2.vts as vte
             from   empat13 t1, empat13 t2
             where  t1.vte < t2.vts
             and    not exists (select *
                    from   empat13 t3
                    where  (t1.vte < t3.vts and t3.vts < t2.vts) or
                           (t1.vte < t3.vte and t3.vte < t2.vts))
             union
             select t1.vte as vts, t2.vte as vte
             from   empat13 t1, empat13 t2
             where  t2.vts < t1.vte and t1.vte < t2.vte
             and    not exists (select *
                    from   empat13 t3
                    where  (t1.vte < t3.vts and t3.vts < t2.vte) or
                           (t1.vte < t3.vte and t3.vte < t2.vte))) const_period
where empat13.vts <= const_period.vts and const_period.vte <= empat13.vte
group by const_period.vts, const_period.vte
order by const_period.vts
```

Figure 7: Temporal Aggregation in Standard SQL

```
flashback table emp to timestamp ('2007-09-01 08:00:00');
```

Figure 8: A Flashback in the Oracle DBMS

database architecture such as a temporal stratum.

**CROSS REFERENCE**

TEMPORAL DATA MODEL
TEMPORAL QUERY LANGUAGES
TEMPORAL QUERY PROCESSING
TEMPORAL QUERY OPTIMIZATION
SUPPORTING TRANSACTION TIME

**RECOMMENDED READING**

~~Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.~~


[1] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal Statement Modifiers TODS 25(4), pp 407–456, 2000.
[2] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction Time Support Inside a Database Engine. ICDE, 2006.
[3] Oracle Corp. Oracle Flashback Technology. http://www.oracle.com/technology/deploy/availability/htdocs-/Flashback_Overview.htm, as of 4.9.2007.
[4] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable Query Optimization and Evaluation in Temporal Middleware. SIGMOD, 2001.
[5] G. Slivinskas and C. S. Jensen. Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types. ADBIS, 2001.
[6] R. T. Snodgrass (editor). The TSQL2 Temporal Query Language. Kluwer Academic Publishers, 1995.
[7] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann, 1999.
[8] M. Stonebraker and L. A Rowe: The design of POSTGRES. SIGMOD, 1986.
[9] K. Torp, C.S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. IDEAS, 1997.
[10] K. Torp, C.S. Jensen, and R. T. Snodgrass. Effective Timestamping in Databases. VLDB Journal 8(3-4), pp. 267–288, 2000.
[11] J. Yang, H. C. Ying, and J. Widom. TIP: a temporal extension to Informix. SIGMOD, 2000.

# Temporal Vacuuming

John F. Roddick, Flinders University, Australia
David Toman, University of Waterloo, Canada

## SYNONYMS

data expiration

## DEFINITION

Transaction-time temporal databases are inherently append-only resulting, over time, in a large historical sequence of database states. Data vacuuming allows for a strategic, and irrevocable, deletion of obsolete data.

## HISTORICAL BACKGROUND

The term *vacuuming* was first used in relation to databases in the Postgres database system as a mechanism for moving old data to archival storage [11]. It was later refined by Jensen and Mark in the context of temporal databases to refer to the removal of *obsolete* information [4] and subsequently developed into a comprehensive and usable adjunct to temporal databases [6, 8, 12]. Data expiry has also been investigated in the context of data warehouses by Garcia-Molina et al [2] and others [9].

## SCIENTIFIC FUNDAMENTALS

In many applications, data about the past needs be retained for further use. This idea can be formalized, at least on the conceptual level, in terms of an append-only transaction-time temporal database or a history. However, a naive and unrestricted storage of all past data inevitably leads to unreasonable demands on storage and subsequently impacts negatively on efficiency of queries over such histories. Hence techniques that allow selective removal of *no longer needed* data have been developed and, at least in prototype systems, deployed.

The parts of the historical data that are to be retained/deleted are specified in terms of *vacuuming specifications*. These specifications state, for example, that data beyond certain absolute or relative time point is *obsolete* (as opposed to merely *superceded*) and thus can be removed. For example, the regulation

*"Taxation data must be retained for the last five years"*

can be considered a specification of what data individuals must retain concerning their taxation returns and what can be discarded. However, one must be careful when designing such specifications as once a part of the history is deleted, it can no longer be reconstructed from the remaining data. Consider the alternative regulation

*"Taxation data must be retained for past years, except for the last year".*

While this specification seems to suggest that the data of the last year can be discarded, doing so would lead to a problem in the following year (as this year's return won't be the last one any more). Hence, this specification is intuitively not well formed and must be avoided. Vacuuming specifications can be alternatively phrased in terms of what may be deleted, rather than what should be retained. For example, rather than

*"Taxation data must be retained for the last five years"*

it would be better to rephrase it as

*"Taxation data over five years old may be deleted".*

The reason for this is that vacuuming specifications indicate specific deletion actions and there is thus less chance of misinterpretation.

Another issue with vacuuming specifications relates to the granularity of data that is removed from the history. For example, if data items (such as tax receipts) are temporally correlated with other items that may appear in different, perhaps much older, parts of the history (such as investments), those parts of the history may have to

be retained as well. Such considerations must be taken into account when deciding whether the specifications are allowed to refer to the complete history or whether selective vacuuming of individual data items is permitted. In both cases, issues of data integrity needs to be considered.

## Formal Vacuuming Specifications

A transparent way to understand the above issues is to consider the result of applying a vacuuming specification to a history (i.e., the *retained* data) to be a view defined on the original history.

**Definition.** Let $H = \langle S_0, S_1, \ldots, S_k \rangle$ be a history. The instances $S_i$ represent the state of the data at time $i$ and all states share a common fixed schema. $\mathbf{T}_H$ and $\mathbf{D}_H$ are used to denote the *active* temporal and data domains of $H$, respectively. A *vacuuming specification* is a function (a view) $E : H \to H'$, where $H'$ is called the *residual history* (with some, potentially different but fixed schema).

The idea behind this approach is that the instance of the view represents the *result* of applying the vacuuming specification to the original history and *it is this instance that has to be maintained* in the system. While such view(s) usually map histories to other histories (sometimes called the *residual histories*), in principle, there is no restriction on the schema of these views nor on the language that defines the view. This approach allows one to address the two main questions concerning a vacuuming specification:

**Is a given specification well formed?** This question relates to anomalies such as the one outlined in the introductory example. As only the instance of the view and not the original history itself is *stored*, a new instance of the view must be definable in terms of the current instance of the view $E(H)$ whenever a new state of the history $S$ is created by progression of time. This condition can be formalized by requiring:

$$E(H; S) = \Delta(E(H), S)$$

for some function (query) $\Delta$ where $H; S$ is the extension of $H$ with a new state $S$. To start this process, a constant, $\emptyset$, is technically needed to represent the instance of the view in the beginning (i.e., for an empty initial history). The condition above essentially states that the view $E$ must be *self-maintainable* in terms of the pair $(\emptyset, \Delta)$. The pair $(\emptyset, \Delta)$ is called a *realization* of $E$.

**What queries does a specification support?** The second question concerns which queries can be correctly answered over the residual histories. Again, for a query $Q$ to be answerable, it must be the case that

$$Q(H) = Q'(E(H))$$

for some function (query) $Q'$ and all histories $H$. $Q'$ is a *reformulation* of $Q$ with respect to $E$. This requirement states that queries preserved by the vacuuming process are exactly those that can be answered only using the view $E$.

In addition, for the approach to be *practical*, the construction of $\Delta$ and $\emptyset$ from $E$ and of $Q'$ from $Q$ and $E$, respectively, must be effective.

**Definition.** A vacuuming specification represented by a self-maintainable view $E$ is a *faithful history encoding for a query* $Q$ if $Q$ is answerable using the view $E$.

Given a vacuuming specification $E$ over a history $H$ that is self-maintainable using $(\emptyset, \Delta)$ and a query $Q'$ that answers $Q$ using $E$; the triple $(\emptyset, \Delta, Q')$ is then called the *expiration operator* of $H$ for $Q$.

## Space/storage Requirements

Understanding vacuuming specifications in terms of self-maintainable materialized views also provides a natural tool for comparing different specifications with respect to *how well they remove unnecessary data*. This can be measured by studying the size of the instances of $E$ with respect to several parameters of $H$:

- the size of the history itself, $|H|$,
- the size of the active data domain, $|\mathbf{D}_H|$, and
- the length of the history, $|\mathbf{T}_H|$.

In particular, the dependency of $|E(H)|$ on $|\mathbf{T}_H|$ is important as the progression of time is commonly the major factor in the size of $H$. It is easy to see that vacuuming specification with a *linear* bound in terms of $\mathbf{T}_T$ always exists: it is, e.g., the identity used to define both $E$ and $Q'$. However, such a specification is not very useful and better results can be probably achieved using standard *compression algorithms*. Therefore the main interest is in two main cases defined in terms of $|\mathbf{T}_H|$:

1.specifications bounded by $\Omega(1)$, and

2.specifications bounded by $\Omega(\log(|\mathbf{T}_H|))$.

In the first case the vacuuming specification provides a *bounded encoding of a history*. Note that in both cases, the size of $E(H)$ will still depend on the other parameters, e.g., $|\mathbf{D}_H|$. This, however, must be expected, as intuitively, the more different constants (individuals) $H$ refers to, the larger $E(H)$ is likely to be (for example, to store the *names* of the individuals).

**Vacuuming in Valid-Time Databases.** In contrast to transaction-time temporal databases (or histories), valid-time temporal databases allow *arbitrary* updates of the temporal data: hence information about future can be recorded and data about the past can be modifies and/or deleted. This way, vacuuming specifications reduce to appropriate updates of the valid time temporal database.

Moreover, when allowing arbitrary updates of the database, it is easy to show that the only *faithful history encoding*s are those that are lossless (in the sense that $H$ can be reconstructed from the instance of $E$).

**Example.** Consider a valid time temporal database $H$ with a schema $\{R\}$ and a query $Q$ asking *"return the contents of the last stare of R recorded in H"*. Then, for a vacuuming specification $E$ to be a *faithful encoding* of $H$ (w.r.t. $Q$), it must be possible to answer $Q$ using only the instance of $E$ after updating of $H$. Now consider a sequence of updates of the form *"delete the last state of R in H"*. These updates, combined with $Q$, can *reconstruct* the contents of $R$ for an arbitrary state of $H$. This can only be possible if $E$ is lossless.

This, however, means that any such encoding must occupy roughly the same storage as the original database, making vacuuming useless. Similar result can be shown even for valid time databases in which updates are restricted to insertions.

## Approaches to Vacuuming

The ability to vacuum data from a history depends on the expressive power of the query language in which queries over the history are formulated and on the number of the actual queries. For example, allowing an arbitrary number of *ad-hoc* queries precludes any possibility effective vacuuming of data, as finite relational structures can be completely characterized by first-order queries. Thus, for common temporal query languages, this observation leaves us with two essential options:

1.an *administrative* solution is adopted and a given history is vacuumed using a set of *policies* independent of queries. Ad-hoc querying of the history can be allowed in this case. However, queries that try to access already expired values (i.e., for which the view is not faithful history encoding) have to fail in a predefined manner, perhaps by informing the application that the returned answer may be only approximate, or

2.a query driven data expiration technique is used. Such a technique, however, can only work for a fixed set of queries *known in advance*.

## Administrative Approaches to Vacuuming

One approach to vacuuming data histories, and, in turn, to defining *expiration operators*, can be based on *vacuuming specifications* that define query/application-independent policies. However, when data is removed from a history in such a way, the system should be able to characterize queries whose answers are not affected. A particular way to provide vacuuming specifications (such as through the ideas of Skyt et al [6, 8]) is using *deletion* ($\rho$) and *keep* ($\kappa$) expressions. These would be invoked from time to time, perhaps by a vacuuming daemon. The complete specification may contain both deletion and keep specifications. For example:

$$\rho(EmpDep) : \sigma_{TT_{end} \leq NOW - 1yr}(EmpDep)$$
$$\kappa(EmpDep) : \sigma_{EmpStatus = 'Retain'}(EmpDep)$$
$$\rho(EmpDep) : \sigma_{VT_{end} \leq NOW - 7yrs}(EmpDep)$$

This specification states that unless the Employee has a status of 'Retain' all corrected data should be vacuumed after 1 year and all superceded data vacuumed after 7 years. For safety, keep specifications always override delete specifications (note that the ordering of the individual deletion and keep expressions is significant).

**Vacuuming in Practice.** Vacuuming specifications are generally given as either part of the relation definition or as a stand-alone vacuuming specification. In TSQL2, for example, a `CREATE TABLE` command such as:

```
CREATE TABLE EmpDep (
      Name     CHAR(30) NOT NULL,
      Dept     CHAR(30) NOT NULL)
AS TRANSACTION YEAR(2) TO DAY
VACUUM NOBIND (DATE 'now - 7 days');
```

specifies *inter alia* that only queries referencing data valid within the last seven days are permissible [3] while

```
CREATE TABLE EmpDep ( ...  )  VACUUM DATE '12 Sep 2007';
```

specifies that only query referencing any data entered on or after 12 September 2007 are permissible. The `VACUUM` clause provides a specification of what temporal range constitutes a valid query with the `NOBIND` keyword allowing `DATE` to be the date that the query was executed (as opposed to the date that the table was created).

An alternative is to allow tuple-level expiration of data. In this case, the expiration date of data is specified on insert. For example, in the work of Schmidt et al [5] users might enter:

```
INSERT INTO EmpDep VALUES ('Plato', 'Literature', ...)
                   EXPIRES TIMESTAMP '2007-09-12 23:59:59';
```

to indicate that the tuple may be vacuumed after the date specified.

## Application/Query-driven Approaches

There are many applications that collect data over time but for which there are no *natural* or *a priori* given vacuuming specifications. However, it is still important to control the size of the past data needed. Hence, it is a natural question whether appropriate specifications can be derived *from the (queries in the) applications* themselves (this requires an *a priori* fixed *finite* set of queries – in the case of ad-hoc querying such a specification cannot exist. Formally, given a query language $\mathcal{L}$, a computable mapping of queries $Q \in \mathcal{L}$ to triples $(\emptyset, \Delta, Q')$, such that $(\emptyset, \Delta, Q')$ is an expiration operator for $Q$ over $H$, has to be constructed. Figure 1 summarizes the results known for various temporal query languages and provides references to the actual techniques and proofs.

| Temporal Query Language | Lower bound | Upper bound | Reference |
|---|---|---|---|
| Past FO Temporal Logic | bounded | POLY($\mathbf{D}_H$) | [1] |
| Past Temporal $\mu$-Calculus | bounded | POLY($\mathbf{D}_H$) | [13] |
| Temporal Relational Calculus | bounded | ELEM($\mathbf{D}_H$) | [12] |
| Future Temporal $\mu$-Calculus | $\Omega(\lvert\mathbf{T}_H\rvert)$ | $O(\lvert H\rvert)$ | [15] |
| Propositional Past TL w/duplicates | $\Omega(\lvert\mathbf{T}_H\rvert)$ | $O(\lvert H\rvert)$ | [15] |
| Past Temporal $\mu$-Calculus w/bounded duplicates | $\Omega(\lvert\log(\mathbf{T}_H\rvert))$ | $\Omega(\lvert\log(\mathbf{T}_H\rvert))$ | [15] |
| Conjunctive Past TL Queries w/duplicates | $\Omega(\lvert\log(\mathbf{T}_H\rvert))$ | $\Omega(\lvert\log(\mathbf{T}_H\rvert))$ | [15] |
| Past TL Queries w/counting | $\Omega(\lvert\log(\mathbf{T}_H\rvert))$ | $O(\lvert H\rvert)$ | [14] |

Figure 1: Space bounds for Residual Histories.

## KEY APPLICATIONS

The major application domains for vacuuming are historical databases (that, being append only, need a mechanism to limit their size), logs (particularly those collected for more than one purpose with different statutes and business processes), monitoring applications (with rollback requirements) and garbage collection (in programming

languages). Also, as *data streams* are essentially *histories*, the techniques and results developed for vacuuming and data expiration can be applied to query processing over data streams. In particular, expiration operators for a given query yield immediately a *synopsis* for the same query in a streaming setting. This observation also allows the transfer of the space complexity bounds.

## FUTURE DIRECTIONS

Most approaches have concentrated on specifying what data to retain for given queries to continue to be answered perfectly. There are two other possibilities:

- given a particular vacuuming specification and a query that is not supported fully by this specification, can the degree can this query be answered by the residual history be determined? Some suggestions are given by Skyt and Jensen [7] who propose that queries that may return results affected by vacuuming should also provide suggestions for an alternative, similar query.
- given a requirement that certain queries should not be answered (e.g., for legal reasons), what would be the vacuuming specifications that would guarantee this, in particular in the conjunction with the issue of approximate answers above?

Both of these are areas for further research. Finally, most vacuuming research assumes a static schema definition (or at least, an overarching applicable schema definition). Having the versioning of schema while also handling the vacuuming of data is also an open problem.

## CROSS REFERENCE

Point-stamped data models, Temporal Query Languages, Schema Versioning, self-maintainable materialized views, Query rewriting over views, Synopses for Data Streams

## RECOMMENDED READING


[1] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.

[2] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data in a warehouse. In A. Gupta, O. Shmueli, and J. Widom, editors, *International Conference on Very Large Data Bases, VLDB'98*, pages 500–511, New York, NY, USA, 1998. Morgan Kaufmann.

[3] C. Jensen. Vacuuming. In R. Snodgrass, editor, *[10]*, pages Chapter 23, pp. 451–462. Kluwer Academic Publishing, 1995.

[4] C. S. Jensen and L. Mark. A framework for vacuuming temporal databases. Technical Report CS-TR-2516, University of Maryland at College Park, 1990.

[5] A. Schmidt, C. Jensen, and S. Saltenis. Expiration times for data management. *22nd International Conference on Data Engineering, ICDE'06*, pages 36–36, 2006.

[6] J. Skyt. *Specification-Based Techniques for the Reduction of Temporal and Multidimensional Data*. PhD thesis, Aalborg University, 2001.

[7] J. Skyt and C. S. Jensen. Vacuuming temporal databases. TimeCenter Technical Report TR-32, Aalborg University, 1998.

[8] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Data and Knowledge Engineering*, 44(1):1–29, 2003.

[9] J. Skyt, C. S. Jensen, and T. B. Pedersen. Specification-based data reduction in dimensional data warehouses. In *18th International Conference on Data Engineering*, page 278, San Jose, CA, 2002. IEEE Computer Society.

[10] R. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishing, New York, 1995.

[11] M. Stonebraker and L. Rowe. The design of postgres. In C. Zaniolo, editor, *ACM SIGMOD International Conference on the Management of Data*, pages 340–355, Washington, 1986. ACM.

[12] D. Toman. Expiration of historical databases. In *International Symposium on Temporal Representation and Reasoning, TIME'01*, pages 128–135, 2001.

[13] D. Toman. Logical Data Expiration for Fixpoint Extensions of Temporal Logics. In *International Symposium on Advances in Spatial and Temporal Databases SSTD 2003*, pages 380–393, 2003.

[14] D. Toman. On Incompleteness of Multi-dimensional First-order Temporal Logics. In *International Symposium on Temporal Representation and Reasoning and International Conference on Temporal Logic*, pages 99–106, 2003.

[15] D. Toman. On Construction of Holistic Synopses under the Duplicate Semantics of Streaming Queries. In *International Symposium on Temporal Representation and Reasoning, TIME'07*, pages 150–162, 2007.

# TITLE

temporal XML

# BYLINE

Curtis Dyreson
Utah State University
Curtis.Dyreson@usu.edu
http://www.cs.usu.edu/~cdyreson

Fabio Grandi
Alma Mater Studiorum Università di Bologna
fgrandi@deis.unibo.it
http://www-db.deis.unibo.it/~fgrandi

# SYNONYMS

temporal semi-structured data

# DEFINITION

Temporal XML is a *timestamped instance* of an XML data model or, more literally, an XML document in which *specially-interpreted* timestamps are present. In general an XML data model instance is a tree or graph in which each node corresponds to an element, attribute, or value, and each edge represents the lexical nesting of the child in the parent's content. In temporal XML, a timestamp is added to some nodes or edges in the instance. The timestamp represents the lifetime of the node or edge in one or more temporal dimensions, usually valid time or transaction time. As an example, Figure 1 shows a fragment of a temporal XML data model. The bibliographic data in the figure contains information about publishers, books, and authors. The figure also has timestamps that represent *when* each piece of data was entered into the data collection (i.e., the timestamps represent the *transaction-time* lifetime of each element). The bibliography began on 12/21/01, and remains current (until *now*). Information about the Butterfly Books publisher was entered on 1/1/04, and it started publishing a book by Jane Austen on 2/2/04. The title of that book was originally misspelled, but was corrected on 5/29/05. Alternatively, temporal XML is literally an XML document or data collection in which specially-interpreted timestamps, formatted in XML, are included. Such a document yields a temporal XML data model instance when parsed.

# HISTORICAL BACKGROUND

XML is becoming an important language for data representation and exchange, especially in web applications. XML is used to "mark-up" a data collection or document adding meaning and structure. The mark-up consists of elements inserted into the data. Usually an XML document is modeled as a tree in which each interior node corresponds to an element in the document and each leaf to a text value, attribute, or empty element. Temporal XML adds timestamps to the nodes and/or edges in the data model instance. The timestamps represent the lifetime of the nodes (edges).

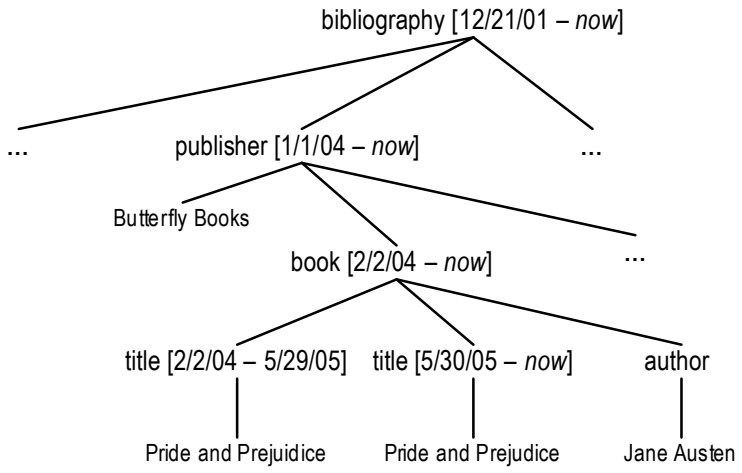



**Figure 1** A temporal XML fragment

Grandi has created a good bibliography of research in this area [7]. Chawathe et al. were the first to study time in an XML-like setting [2]. They encoded times in edge labels in a semistructured database and extended the Lorel query language with temporal constructs. Dyreson et al. extended their research with collapsing and coalescing operators [4]. Grandi and Mandreoli presented techniques for adding explicit valid-time timestamps in an XML document [8]. Amagasa et al. next developed a temporal extension of the XML data model [1]. Following that a range of temporal XML topics was investigated, from storage issues and indexing [10][11][12][13][14] to querying [5][6][14]. The timestamping of XML documents (or parts thereof) has also been considered in the more general context of versioning of XML documents [11][15]. Finally, schemes for validating and representing times in XML documents have also been considered [3][9].

# SCIENTIFIC FUNDAMENTALS

It is important to distinguish between "the representation in XML of a time" and "temporal XML." Times are common in many XML documents, especially documents that record the history of an enterprise. There is nothing special about the representation or modeling of these times; they would be modeled just the same as any other snippet of XML, e.g., represented within a `<time>` element. Temporal XML on the other hand is different. It models both the components within a document or data collection and their lifetimes. An instructive way to think about the difference is that temporal XML weds metadata in the form of timestamps to data contained in a document, i.e., to the elements or parts of the document that are annotated by the timestamps. Research in temporal XML builds on earlier research in temporal (relational) databases. Though many of the concepts and ideas carry over to temporal XML research, the ideas have to be adapted to the tree-like model of XML.

Many temporal XML data models impose a transaction-time constraint on the times along every path in a model instance: the timestamp of a child must be during (inclusive) the timestamp of its parent [1][3]. Said differently, no child may outlive its parent in transaction time. The reason for this constraint is that every snapshot of a temporal data model instance must be a single, complete, valid non-temporal XML data model instance. A non-temporal instance has a single

root; but if in a temporal instance a child outlives its parent then, in some snapshot(s), the child represents a second root since it has no parent, thus violating a model property. In valid time it is more common to relax this constraint and model a temporal data collection as a sequence of forests where a child that outlives its parent is interpreted to mean that the child is the root in some snapshot(s) of some tree in the forest [9]. For instance, the valid time of Jane Austen's book Pride and Prejudice would extend from its time of publication (1813) to *now*, far exceeding the lifetime of its publication by Butterfly Books.

Another interesting situation is when a child moves among parents over time (for instance, in the data collection shown in Figure 1 if the book Pride and Prejudice were published by two, different publishers). A directed graph data model is better suited to modeling such movement as a node (e.g., the book) can have multiple incoming edges (e.g., an edge from each publisher) [4]. Various constraints have been proposed for relationships among the timestamps on nodes and edges in the graph.

Timestamps on nodes/edges in a data model instance changes query evaluation. At the core of all XML query languages (and different from SQL or relational query languages) are path expressions that navigate to nodes in a data model instance. In a temporal data model instance a query has to account for the timestamps along each path that it explores. In general, a node is only available during the intersection of times on every node and edge in the path to it (though a node in a graph data model can be reached along multiple paths). Temporal XML queries can be evaluated using a sequenced semantics [6], that is, simultaneously evaluated in every snapshot or non-sequenced [5][13] where differences between versions can be extracted and paths between versions are directly supported by the data model.

# KEY APPLICATIONS

Temporal XML can be used to model an evolving document or data collection. In many situations "old" documents or document versions are still of use. For instance, in an industrial domain an airplane parts manufacturer has to retain part plan histories to produce parts for older planes, while in the legal domain a tax firm has to keep a complete history of tax laws for audits. Currently, the de facto method for storing old documents is an *archive*. An archive is a warehouse for deleted or modified documents. Archives can be site-specific or built for a number of sites, e.g., the Internet Archive. But the method to retrieve documents from an archive varies widely from site to site which is problematic because then queries also have to vary. Moreover, archives typically only support retrievals of entire document versions, not a full range of temporal queries or version histories of individual elements. In contrast, temporal XML provides a basis for supporting a full range of temporal queries. Temporal XML can also be explicitly used to represent, store or view historical data, including structured data, or to encode multi-version documents. Multi-version documents are compact representations of XML documents which maintain their identity through modifications and amendments. A temporally consistent individual version or range of consecutive versions (timeslice) can be extracted by means of a temporal query. Temporal XML has also been proposed as a medium of communication with temporal relational databases in the context of traditional enterprise applications.

# FUTURE DIRECTIONS

The future of temporal XML is tied to the continued growth of XML as an important medium for data storage and exchange. Currently, many sites promote XML by publishing data formatted in

XML (e.g., genomic and proteomic data can be obtained in three, different XML formats from the National Center for Biotechnology Information (NCBI)). Building a temporal XML data collection by accumulating snapshots gathered from these sites is vital to answering queries such as "What new data has emerged over the past six months?". As search engines become more XML-aware, they could also benefit enormously from making time a relevant component in ranking resources, e.g., a search for "mp3 players" should lower the ranking of discontinued products. The growth of the Semantic Web may lead to XML being supplanted by new languages for knowledge representation such as the Ontology Web Language (OWL). Temporal extensions of these languages will not be far behind; OWL already has one such extension: the Time-determined Ontology Web Language (TOWL).

# CROSS REFERENCES

XML, temporal databases, temporal queries

# RECOMMENDED READING


[1] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura,, "A Data Model for Temporal XML documents," in *DEXA* 2000: 334-344.

[2] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom "Representing and Querying Changes in Semistructured Data," in *ICDE* 1998: 4-13.

[3] Faiz Currim, Sabah Currim, Curtis Dyreson, and Richard T. Snodgrass, "A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with τXSchema," in *EDBT* 2004: 348-365.

[4] Curtis Dyreson, Michael H. Böhlen, and Christian S. Jensen, "Capturing and Querying Multiple Aspects of Semistructured Data," in *VLDB* 1999: 290-301.

[5] Curtis E. Dyreson, "Observing Transaction-Time Semantics with *TT*XPath," in *WISE*, 2001: 193-202.

[6] Dengfeng Gao and Richard T. Snodgrass, "Temporal Slicing in the Evaluation of XML Queries," in *VLDB* 2003: 632-643.

[7] Fabio Grandi. *Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web*. SIGMOD Record, **33**(2), 2004.

[8] Fabio Grandi, Federica Mandreoli, "The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents," in *ADVIS* 2000: 294-303.

[9] Fabio Grandi, Federica Mandreoli, and Paolo Tiberio. *Temporal Modelling and Management of Normative Documents in XML Format*. Data & Knowledge Engineering, 2005, **54**(3): 327-254.

[10] Shu-Yao Chien. Vassilis J. Tsotras. Carlo Zaniolo. *Efficient Schemes for Managing Multiversion XML Documents*. VLDB Journal, 2002. **11**(4): 332–353.

[11] Theodoros Mitakos, Manolis Gergatsoulis, Yannis Stavrakas, and Efstathios V. Ioannidis, *Representing Time-Dependent Information in Multidimensional XML*. Journal of Computing and Information Technology, 2001, **9**(3): 233-238.

[12] Fushen Wang and Carlo Zaniolo, "X-BiT: An XML-based Bitemporal Data Model," in *ER* 2004: 810-824.

[13] Fushen Wang and Carlo Zaniolo. *An XML-based Approach to Publishing and Querying the History of Databases*. World Wide Web, 2005, **8**(3): 233-259.

[14] Flavio Rizzolo and Alejandro A. Vaisman. *Temporal XML: Modeling, Indexing and Query Processing*. VLDB Journal, 2007 (*in press*), DOI 10.1007/s00778-007-0058-x.

[15] Raymond K. Wong, Franky Lam, and Mehmet A. Orgun. *Modelling and Manipulating Multidimensional Data in Semistructured Databases*. World Wide Web, 2001, **4**(1-2): 79-99.

# Time Domain

Angelo Montanari* and Jan Chomicki+

*Dipartimento di Matematica e Informatica,
Università degli Studi di Udine, Udine, Italy
http://users.dimi.uniud.it/∼angelo.montanari/

+ Department of Computer Science and Engineering
University at Buffalo, SUNY, USA
http://www.cse.buffalo.edu/∼chomicki/

## Synonyms

Temporal domain, temporal structure

## Definition

In its full generality, a time domain can be defined as a set of *temporal individuals* connected by a set of *temporal relations*. Different choices for the temporal individuals and/or the temporal relations give rise to different temporal ontologies.

In the database context, the most common temporal ontology takes *time instants* (equivalently, points or moments) as the temporal individuals and a *linear order* over them as the (unique) temporal relation [CT05]. In addition, one may distinguish between discrete and dense, possibly continuous, time domains and between bounded and unbounded time domains. In the discrete case, one may further consider whether the time domain is finite or infinite and, in the case of unbounded domains, one can differentiate between left-bounded, right-bounded, and totally unbounded domains. Moreover, besides linear time, one may consider *branching time*, where the linear order is replaced with a partial one (a tree or even a directed acyclic graph), or circular time, which can be used to represent temporal periodicity.

As for temporal individuals, time instants can be replaced with *time intervals* (equivalently, periods or stretches of time) connected by (a subset of) Allen's relations `before`, `meets`, `overlaps`, `starts`, `during`, `equal`, and `finishes`, and their inverses or suitable combinations [GMS04]. As in the case of instant-based domains, we may distinguish between discrete and dense

domains, bounded and unbounded domains, linear, branching, and circular domains, and so on.

Finally, as most temporal database applications deal with both qualitative and quantitative temporal aspects, instant-based time domains are usually assumed to be isomorphic to specific numerical structures, such as those of natural, integer, rational, and real numbers, or to fragments of them, while interval-based ones are obtained as suitable intervallic constructions over them. In such a way, time domains are endowed with *metrical features*.

## Historical background

The nature of time and the choice between time instants and time intervals as the primary objects of a temporal ontology have been a subject of active philosophical debate since the times of Zeno and Aristotle. In the twentieth century, major contributions to the investigation of time came from a number of disciplines. A prominent role was played by Prior who extensively studied various aspects of time, including axiomatic systems of tense logic based on different time domains.

Nowadays, besides physics, philosophy, and linguistics, there is a considerable interest in temporal structures in mathematics (theories of linear and branching orders), artificial intelligence (theories of action and change, representation of and reasoning with temporal constraints, planning), and theoretical computer science (specification and verification of concurrent and distributed systems, formal analysis of hybrid temporal systems that feature both discrete and continuous components). A comprehensive study and logical analysis of instant-based and interval-based temporal ontologies, languages, and logical systems can be found in [vB91].

As for *temporal databases*, the choice of the time domain over which temporal components take their value is at the core of any application. In most cases, a discrete, finite, and linearly ordered (instant-based) time domain is assumed. This is the case, for instance, with SQL standards [S00]. However, there is no single way to represent time in a database, as witnessed by the literature in the field. To model when something happened, time instants are commonly used; validity of a fact over time is naturally represented by the (convex) set of time instants at which the fact holds, the time *period of validity* in the temporal database terminology; finally, to capture processes as well as some kinds of temporal aggregation, time intervals are needed.

## Scientific fundamentals

**Basics.** The choice between time instants and time intervals as the basic time constituents is a fundamental decision step that all temporal systems have in common. In mathematics, the choice of *time instants*, that is, points in time without duration, is prevalent. Even though quite abstract, such a solution has

turned out to be extremely fruitful and relatively easy to deal with in practice. In computer science, additional motivations for this choice come from the natural description of computations as possibly infinite sequences of instantaneous steps.

The alternative option of taking *time intervals*, that is, stretches of time with duration, as temporal individuals seems to better adhere to the concrete experience of people. Physical phenomena as well as natural language expressions involving time can be more easily described in terms of time intervals instead of time instants. Nevertheless, the complexity of any systematic treatment of time intervals prevents many systems from the adoption of an interval-based ontology.

The instant and the interval ontologies are systematically investigated and compared in [vB91]. The author identifies the conditions an instant-based (resp., interval-based) structure must satisfy to be considered an adequate model of time. Then, through an axiomatic encoding of such conditions in an appropriate language, he provides a number of (first-order and higher order) logical theories of both instant-based and interval-based discrete, dense, and continuous structures. Finally, he illustrates the strong connections that link the two time ontologies. In particular, he shows how interval-based temporal structures can be obtained from instant-based ones through the standard process of interval formation and how instant-based temporal structures can be derived from interval-based ones by a (non-trivial) limiting construction.

A metric of time is often introduced to allow one to deal with time distance and/or duration. In particular, a time metric is needed to define calendar times, such as those based on the commonly used Gregorian calendar.

**Temporal models and query languages.** The choice of the time domain has an impact on various components of temporal databases. In particular, it influences temporal data models and temporal query languages.

As for *temporal data models*, almost all of them adopt an instant-based time ontology. Moreover, most of them assume the domain to be linear, discrete and finite. However, many variants of this basic structure have been taken into consideration [MP93]. Right-unbounded domains have been used to record information about the future. Dense and continuous domains have been considered in the context of temporal constraint databases, that allow one to represent large, or even infinite, sets of values, including time values, in a compact way. Branching time has been exploited in applications where several alternatives have to be considered in the future and/or past evolution of temporal data.

Many data models distinguish between absolute (anchored) and relative (unanchored) time values. Absolute time values denote specific temporal individuals. In general, they are associated with a time metric, such as that of calendar times. As an example, the 14th of September 2007 is an absolute time value that denotes a specific element of the domain of days in the Gregorian calendar. Relative time values specify the distances between pairs of time instants or the durations of time intervals. Absolute and relative time values can

also be used in combination. As an example, the expression 7 days after the 14th of September 2007 denotes the 21st of September 2007.

As for *temporal query languages*, they typically assume that time is isomorphic to natural numbers. This is in agreement with the most common, *linear-time* dialect of temporal logic. In temporal constraint databases, however, the use of classical query languages like relational calculus or algebra accommodates a variety of time domains, including dense and continuous ones.

**Time domain and granularity.** Despite its apparent simplicity, the addition of the notion of time domain to temporal databases presents various subtleties. The main ones concern the nature of the elements of the domain. As soon as calendar times come into play, indeed, the abstract notion of instant-based time domain must be contextualized with respect to a specific *granularity*. Any given temporal granularity can be viewed as a suitable abstraction of the real time line that partitions it into a denumerable sequence of homogeneous stretches of time. The elements of the partition, granules in the temporal database terminology, become the individuals (non-decomposable time units) of a discrete time domain. With respect to the considered granularity, these temporal individuals can be assimilated to time instants. Obviously, if a shift to a finer granularity takes place, e.g., if we move from the domain of months to the domain of days, a single granule must be replaced with a set of granules. In such a way, being instantaneous is not more an intrinsic property of a temporal individual, but it depends on the time granularity we refer to. A detailed analysis of the limitations of the temporal database management of instant-based time domains can be found in [S00].

**The association of time with data.** The association of the elements of the time domain with data is done by *timestamping*. A timestamp is a time value associated with a data object. In the relational setting, we distinguish between attribute-timestamped data models, where timestamps are associated with attribute values, and tuple-timestamped data models, where timestamps are associated with tuples of values. As a third possibility, a timestamp can be associated with an entire relation/database.

Timestamps can be single elements as well as sets of elements of the time domain. Time instants are usually associated with relevant events, e.g., they can be used to record the day of the hiring or of the dismissal of an employee. (Convex) sets of time instants are associated with facts that hold over time. As an example, if a person E works for a company C from the 1st of February 2007 to the 31st of May 2007, we keep track of the fact that every day in between the 1st of February 2007 and the 31st of May 2007, endpoints included, E is an employee of C.

Time intervals are needed to deal with situations where validity over an interval cannot be reduced to validity over its subintervals (including point subintervals). This is the case with processes that relate to an interval as a whole, meaning that if a process consumes a certain interval it cannot possibly

transpire during any proper subinterval thereof. Examples are the processes of baking a cake or of flying from Venice to Montreal. This is also the case when validity of a fact at/over consecutive instants/intervals does not imply its validity over the whole interval. As an example, two consecutive phone calls with the same values are different from a single phone call over the whole period. This is also the case for some kinds of temporal aggregation [BGJ06]. Finally, the use of time intervals is common in several areas of AI, including knowledge representation and qualitative reasoning, e.g., [AF94].

It is important to avoid any confusion between this latter use of intervals as timestamps and their use as compact representations of sets of time points (time periods in the temporal database literature). Time intervals are indeed often used to obtain succinct representations of (convex) sets of time instants. In such a case, validity over a time period is interpreted as validity at every time instant belonging to it. As an example, the fact that a person E worked for a company C from the 1st of February 2007 to the 31st of May 2007 can be represented by the tuple (E, C, [2007/02/01, 2007/05/31]) meaning that E worked for C every day in the closed interval [2007/02/01, 2007/05/31] .

## Key applications

As already pointed out, the time domain is an essential component of any temporal data model, and thus its addition to SQL standards does not come as a surprise.

In SQL, time domains are encoded via *temporal data types*. In SQL-92, five (anchored) time instant data types, three basic forms and two variations, are supported (DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE). In addition, SQL-92 features two (unanchored) data types that allow one to model positive (a shift from an instant to a future one) and negative (a shift from an instant to a past one) distances between instants. One can be used to specify distances in terms of years and months (the YEAR-MONTH INTERVAL type), the other to specify distances in terms of days, hours, minutes, seconds, and fractions of a second (the DAY-TIME INTERVAL type). As a matter of fact, the choice of using the word interval to designate a time distance instead of a temporal individual – in contrast with the standard use of this word in computer science – is unfortunate, because it confuses a derived element of the time domain (the interval) with a property of it (its duration). An additional (unanchored) temporal data type, called PERIOD, was included in the SQL/Temporal proposal for the SQL3 standard, which was eventually withdrawn. A period is a convex sets of time instants that can be succinctly represented as a pair of time instants, namely, the first and the last instants with respect to the given order.

SQL also provides *predicates*, *constructors*, and *functions* for the management of time values. General predicates, such as the equal-to and less-than predicates, can be used to compare pairs of comparable values of any given temporal type; moreover, the specific overlap predicate can be used to check whether

two time periods overlap. Temporal constructors are expressions that return a temporal value of a suitable type. It is possible to distinguish datetime constructors, that return a time instant of one of the given data types, and interval constructors, that return a value of YEAR-MONTH INTERVAL or DAY-TIME INTERVAL types. As for functions, they include the datetime value functions, such as the CURRENT_DATE function, that return an instant of the appropriate type, the CAST functions, that convert a value belonging to a given (temporal or non temporal) source data type into a value of the target temporal data type, and the extraction functions, that can be used to access specific fields of instant or interval time values.

## Future directions

Despite the strong prevalence of instant-based data models in current temporal databases, a number of interesting problems, such as, for instance, that of temporal aggregation, motivate a systematic study and development of *interval-based data models*. Moreover, in both instant-based and interval-based data models intervals are defined as suitable sets of elements of an instant-based time domain. The possibility of assuming time intervals as the primitive temporal constituents of the temporal domain is still largely unexplored. Such an alternative deserves a serious investigation.

## Cross references

Temporal Data Model, Temporal Query Languages, Temporal Granularity, Temporal Indeterminacy, Temporal Periodicity, Point-Stamped Temporal Models, Period-Stamped Temporal Models, Temporal Constraints, Temporal Algebras, Now in Temporal Databases

## Recommended reading

[AF94] Actions and Events in Interval Temporal Logic, James Allen and G. Ferguson, Journal of Logic and Computation, 4(5):531-579, 1994.

[vB91] The Logic of Time. A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse (second edition), Johan van Benthem, Kluwer Academic Publisher, 1991.

[BGJ06] How Would You Like to Aggregate Your Temporal Data?, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen, in Proceedings of the 13th International Symposium on Temporal Representation and Reasoning (TIME), IEEE Comp. Society, 2006, pp. 121-136.

[CT05] Temporal Databases, Jan Chomicki and David Toman, Chapter 14 of the Handbook of Temporal Reasoning in Artificial Intelligence, Michael Fisher, Dov Gabbay, and Lluis Vila (Eds.), Elsevier B. V., 2005, pp. 429-467.

[GMS04] A Road Map of Interval Temporal Logics and Duration Calculi, Valentin Goranko, Angelo Montanari, and Guido Sciavicco, Journal of Applied Non-Classical Logics, 14(1-2):9-54, Edition Hermés-Lavoisier, 2004.

[MP93] Temporal Reasoning, Angelo Montanari and Barbara Pernici, Chapter 21 of Temporal Databases: Theory, Design and Implementation, A. Tansell, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (Eds.), Database Systems and Applications Series, Benjamin/Cummings Pub. Co., Redwood City, CA, 1993, pp.534-562.

[S00] Developing Time-Oriented Database Applications in SQL. Chapter 3: Instants and Intervals, Richard T. Snodgrass, Morgan Kauffman Publishers, 2000, pp. 24-87.

# Time in Philosophical Logic

Peter Øhrstrøm, Department of Communication and Psychology,
Aalborg University, Denmark, http://www.hum.aau.dk/~poe/

Per F. V. Hasle, Department of Communication and Psychology,
Aalborg University, Denmark, http://www.kommunikation.aau.dk/sprog/per_hasle_cv.htm

## SYNONYMS

Temporal logic; The Logic of Time

## DEFINITION

The aim of the study of time in philosophical logic is to provide a conceptual framework for an interdisciplinary study of the nature of time and to formalize and study various conceptions and systems of time. In addition, the introduction of time into logic has led to the development of formal systems, which are particularly well suited to represent and study temporal phenomena such as program execution, temporal databases, and argumentation in natural language.

## HISTORICAL BACKGROUND

The philosophy of time is based on a long tradition going back to ancient thought. It is an accepted wisdom within the field that no attempt to clarify the concept of time can be more than an accentuation of some aspects of time at the expense of others. Plato's statement that time is the "moving image of eternity" and Aristotle's suggestion that "time is the number of motion with respect to earlier and later" are no exceptions (see [17]). According to St. Augustine (354–430) time cannot be satisfactorily described using just one single definition or explanation: "What, then, is time? If no one asks me, I know: if I wish to explain it to one that asketh, I know not." [5, p. 40] Time is not definable in terms of other concepts. On the other hand, according to the Augustinian insight, all human beings have a tacit knowledge of what time is. In a sense, the endeavour of the logic of time is to study important manifestations and structures of this tacit knowledge.

There were many interesting contributions to the study of time in Scholastic philosophy, e.g. the analysis of the notions of beginning and ending, the duration of the present, temporal ampliation, the logic of 'while', future contingency, and the logic of tenses. Anselm of Canterbury (ca. 1033–1109), William of Sherwood (ca. 1200–1270), William of Ockham (ca. 1285–1349), John Buridan (ca. 1295–1358), and Paul of Venice (ca. 1369–1429) all contributed significantly to the development of the philosophical and logical analysis of time. - With the Renaissance, however, the logical approach to the study of time fell into disrepute, although it never disappeared completely from philosophy.

However, the $20^{\text{th}}$ century has seen a very important revival of the philosophical study of time. The most important contribution to the modern philosophy of time was made in the 1950s and 1960s by A.N. Prior (1914–69). In his endeavours, A.N. Prior took great inspiration from ancient and medieval thinkers and especially their work on time and logic.

The Aristotelian idea of time as the number of motion with respect to earlier and later actually unites two different pictures of time, the dynamic and the static view. On the one hand, time is linked to motion, i.e. changes in the world (the flow of time), and on the other hand time can be conceived as a stationary order of events represented by numbers. In his works, A.N. Prior logically analysed the tension between the dynamic and the static approach to time, and developed four possible positions in regard to this tension. In particular, A.N. Prior used the idea of branching time to demonstrate that there is a model of time which is logically consistent with his ideas of free

choice and indeterminism. (See [8, p. 189 ff.].)
After A.N. Prior's development of formalised temporal logic, a number of important concepts have been studied within this framework. In relation to temporal databases the studies of the topology of time and discussions regarding time in narratives are particularly interesting.

**SCIENTIFIC FUNDAMENTALS**
In the present context, the following four questions regarding time in philosophical logic seem to be especially important:

What is the relation between dynamic and static time?

2. What does it mean to treat time as "branching"?
3. What is the relation between punctual and durational time (i.e. instants and durations)?
4. What is the role of time in storytelling (narratives)?

In the following, a brief introduction to each of these issues will be given.

## 1 Dynamical and static time: A-theory vs. B-theory

The basic set of concepts for the *dynamic* understanding of time are past, present, and future. In his very influential analysis of time the philosopher John Ellis McTaggart (1866–1925) suggested to call these concepts (i.e., the tenses) the A-concepts. The tenses are well suited for describing the flow of time, since the future will become present, and the present will become past, i.e. flow into past. The basic set of concepts for the *static* understanding of time are before/after and "simultaneous with". Following McTaggart, these are called the B-concepts, and they seem especially apt for describing the permanent and temporal order of events. The two kinds of temporal notions can give rise to two different approaches to time. First, there is the dynamic approach (the A-theory) according to which the essential notions are past, present and future. In this view, time is seen "from the inside". Secondly, there is the static view of time (the B-theory) according to which time is understood as a set of instants (or durations) ordered by the before-after relation. Here time is seen "from the outside". It may be said to be a God's eye-perspective on time.
There is also an ontological difference between the two theories. According to the A-theory the tenses are real whereas the B-theorists consider them to be secondary and unreal. According to the A-theory the Now is real and objective, whereas the B-theories consider the Now to be purely subjective.
The debate between proponents of the two theories received a fresh impetus with A.N. Prior's formal analysis of the problem. (See [9, p. 216 ff.]). According to the B-theory, time is considered to be a partially ordered set of instants, and propositions are said to be true or false at the instants belonging to the set. According to the A-theory, time is conceived in terms of the operators $P$ (Past) and $F$ (Future), which are understood as being relative to a "Now". A.N. Prior suggested a distinction between four possible grades of tenselogical involvement corresponding to four different views of how to relate the A-notions (past, present and future) to the B-notions ('earlier than'/'later than', 'simultaneous with'):

The B-notions are more fundamental than the A-notions. Therefore, in principle, the A-notions have to be defined in terms of the B-notions.

2. The B-notions are just as fundamental as the A-notions. The A-notions cannot be defined in terms of the B-notions or vice versa. The two sets of notions have to be treated on a par.
3. The A-notions are more fundamental than the B-notions. All B-notions can be defined in terms of the A-notions and a primitive notion of temporal possibility.
4. The A-notions are more fundamental than the B-notions. Therefore, in principle the B-notions have to be defined in terms of the A-notions. Even the notion of temporal possibility can be defined on terms of the A-notions.

A.N. Prior's four grades of tense-logical involvement represent four different views of time and also four different foundations of temporal logic. In fact, theory 1 is the proper B-theory and theory 3 and 4 are versions of the proper A-theory. Theory 2 is a kind of intermediate theory.
In theory 1, the tense operators, $P$ (past) and $F$ (future), can be introduced in the following way:

$$T(t, Fq) \equiv_{def} \exists t_1 : t < t_1 \Rightarrow T(t_1, q)$$

$$T(t, Pq) \equiv_{def} \exists t_1 : t_1 < t \Rightarrow T(t_1, q)$$

where $T(t, q)$ is read "$q$ is true at $t$", and $t < t_1$ is read "$t$ is before $t_1$".
In theory 3 and 4, A.N. Prior has shown how instants can be introduced as maximally consistent sets of tense-logical propositions and how the before-after relation can be consistently defined in terms of tense-logical concepts (i.e., A-notions).
From a B-theoretical viewpoint, at any instant, an infinite number of propositions, including tensed ones, will be true about that instant. But from the A-theoretical point of view, precisely the infinite conjunction of the propositions in this set is a construction which, when called an "instant", makes the B-theoretical notion of "instant" secondary and derivable.
It should be noted, that whereas the A-theorist (theory 3 or 4) can translate any B-statement into his language, many A-statements cannot be translated into the B-language. For instance, there is no way to translate the A-statement "it is raining now in Aalborg" into the B-language. The "now" cannot be explained in terms of the B-language consisting of an ordered set of instants and the notion of a proposition being true at an instant. This asymmetry seems to be a rather strong argument in favour of the A-theory (i.e., A.N. Prior's theory 3 or 4).

## 2 Linear vs. Branching Time

The idea of formalised branching time was first brought forward by Saul Kripke in a letter to A.N. Prior in 1958 [8, p. 189-90]. Kripke's later development of the semantics for modal logics is well-known within computer science. But it has in fact been shown by Jack Copeland [3] that the kernel of the ideas published by Kripke were in fact present already in the work of Meredith and A.N. Prior in 1956.
The difference between A.N. Prior's theory 3 and 4 is important if time is considered to be branching. In theory 3, the notion of possibility is primitive. In theory 4, this notion can be derived from the tenses. But then it turns out to be very difficult to distinguish between the possible future, the necessary future and the "plain" future — e.g. between "possibly tomorrow", and "necessarily tomorrow" and just "tomorrow". In all obvious models constructed in accordance with A.N. Prior's theory 4, "tomorrow" is conflated either with "possibly tomorrow" or with "necessarily tomorrow". On the basis of theory 3, there is no difficulty in maintaining a difference between the three kinds of notions discussed. In a theory 3 model, one can refer not only to what happens in some possible future, $\Diamond Fq$, and to what happens in all possible futures, $\Box Fq$, but one can also refer to what is going to happen in the future, $Fq$, as something different from the possible as well as the necessary future. A branching time model with this property is said to be Ockhamistic, whereas a branching time model in which $Fq$ is identified with $\Box Fq$ is said to be Peircean. Graphically, the two kinds of branching time models can be presented as in Figures 1 and 2 respectively.

## 3 Punctual vs. Durational Time

The notion of a 'duration' is important within the study of time. Several logicians have tried to formulate a logic of durations. The medieval logician John Buridan (ca. 1295-1358) regarded the present as a duration and not as a point in time. One example which he considered was the sentence: 'If a thing is moving, then it was moving'. In his analysis Buridan suggested that the logic of tenses can be established in two different ways based on the durational structure of time. Either the tenses can be taken absolutely, in the sense that no part of the present time is said to be past or future. Or the tenses can be taken in the relative sense, according to which "the earlier part of the present time is called past with respect to the later, and the later part is called future with respect to the earlier." Buridan pointed out that if a thing is moving now, then there is a part of the present during which it is moving, and hence, it is moving in some part of the present, which is earlier than some other part of the present. Therefore, if the thing is moving, then it was moving (if the past is taken in the relative sense), i.e., $\text{moving}(x) \Rightarrow P(\text{moving}(x))$. For this reason, the above sentence must be accepted if the past is understood relatively, whereas it has to be rejected if the past tense is understood absolutely. The reason is that one could in principle imagine a beginning of a process of motion. (Details can be found in [8, p. 43 ff.].)
The first modern logician to formulate a kind of durational calculus was A.G. Walker [15]. Walker [1947] suggested a model according to which time is considered as a structure $(S, <)$, where $S$ is a non-empty set of periods (also

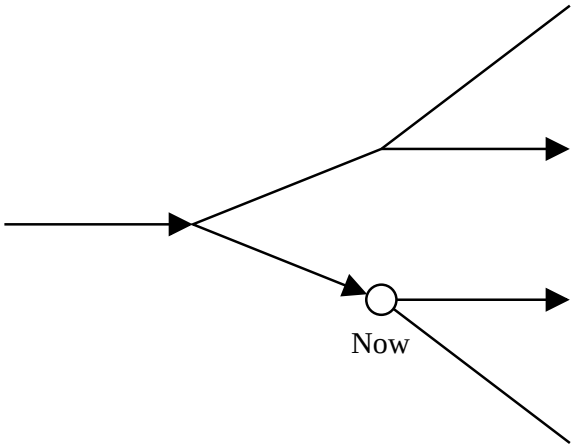



Figure 1: An Ockhamistic model of branching time. At every branching point there will be one possible future which is the true future.

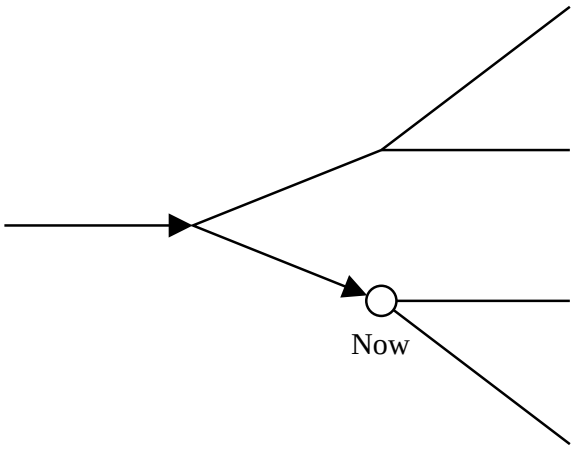



Figure 2: A Peircean model of branching time. There is no difference between the status of the possible futures at any branching point.

called 'durations' or 'intervals'). The '$a < b$'-relation is to be considered as 'strict' in the sense that no overlap between $a$ and $b$ is permitted, and the ordering is assumed to be irreflexive, asymmetrical, and transitive. In addition, he considered the notion of overlap, which can be defined as:

$$a|b \equiv_{def} \neg(a < b \vee b < a)$$

Walker formulated an axiomatic system using the following two axioms:
**DEFINITION**

(W1)$a|a$
(W2)$(a < b \wedge b|c \wedge c < d) \Rightarrow a < d$

Using a set-theoretic method, Walker demonstrated that it is possible to define instants in terms of durations, thus making it possible to view a temporal instant as a 'secondary' construct from the logic of durations.
In 1972 Charles Hamblin [6] independently also put forth a theory of the logic of durations. He achieved his results using a different technique involving the relation:

$$a\, meets\, b \equiv_{def} a < b \wedge \neg(\exists c : a < c \wedge c < b)$$

A decade later, James Allen [1], in part together with Patrick Hayes [2], showed that two arbitrary durations (in

linear time) can be related in exactly 13 ways. It has been shown that all these durational theories are equivalent when seen from an ontological point of view. They all show that just as durations (temporal intervals) can be set-theoretically constructed from an instant-logic, it is also possible to construct instants mathematically from durations. In fact, all the durational theories put forth so far appear to give rise to the same ontological model. The theories formulated by Walker, Hamblin, and Allen can all be said to be B-theoretical. But already Buridan's studies suggested that it is possible to take an A-theoretical approach to durational logic. In modern durational logic an idea similar to Buridan's absolute/relative distinction was introduced in 1980 by Peter Röper [13] and others (see [8, p. 312 ff]).

## 4 Time and Narratives

A narrative is a text which presupposes a kind of event structure, i.e., a story. The temporal order of the story is often called 'told time'. In many cases the story can be represented as a linear sequence of events. However, even if the event structure of the system is linear, the discourse structure can be rather complicated, since the reader (user) can in principle be given access to the events in any order. The order in which the event are presented is often referred to as 'telling time'. Keisuke Ohtsura and William F. Brewer [10] have studied some interesting aspects regarding the relation between the event structure ('told time') and the discourse structure ('telling time') of a narrative text.

**KEY APPLICATIONS***

The philosophy of time has typically been carried out for its own sake. In many cases philosophers and logicians have seen the study of time as intimately related to essential aspects of human existence as such. For this reason, the study of time within philosophical logic has been motivated by a fundamental interest in the concepts dealing with time themselves and not by the search for a possible application. Nevertheless, such fundamental studies of time have turned out to give rise to theories and models which are useful in many ways. For instance, A.N. Prior's analysis of the systematic relation between the dynamic and the static approach to time led him to the invention of what is now called hybrid logic (http://hylo.loria.fr). In general, temporal logic has turned out to be very useful in artificial intelligence and in other parts of computer science.

A.N. Prior's tense logic seems especially relevant for a proper description of the use of interactive systems. A description of such systems from a B-logical point of view alone cannot be satisfactory, since that would ignore the user's 'nowness' which is essential in relation to the user's choices and thus to the very concept of interactivity. If we, on the other hand, make a conceptual start from A.N. Prior's tense logic (i.e. the A-logical point of view), all B-logical notions can be defined in terms of the A-language.

The need for an A-logical description becomes even clearer when one turns to a temporal analysis of systems which are non-linear even from a B-logical perspective, for instance a game-like multimedia system. In her studies of narratives and possible-world semantics, Marie-Laure Ryan [14] has made it clear that such a system is not to be viewed as a static representation of a specific state of affairs. Rather, it contains many different narrative lines which thread together many different states of affairs. Thus it is the choices of the user which will send the history in case on its specific trajectory.

**CROSS REFERENCE***

Allen's Relations
Now in Temporal Databases
Qualitative Temporsl Reasoning
Temporal Database
Temporal Granularity
Temporal Logic in Database Query Languages
Temporal Logical Models
Temporal Object-Oriented Databases
Time Domain

**RECOMMENDED READING**

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.
[2] J. F. Allen and J. P. Hayes. A common-sense theory of time. In *Proc. of the Ninth Int. Joint Conf. on Artificial Intelligence*, pages 528–531, 1985.
[3] Jack Copeland. Meredith, Prior, and the history of possible world semantics. *Synthese*, 150(3):373–97, June 2006.
[4] J.T. Fraser, F.C. Haber, and G.H. Müller, editors. *The Study of Time*, volume I. Springer-Verlag, Berlin, 1972.
[5] R. Gale, editor. *The Philosophy of Time*. Prometheus Books, New Jersey, 1968.
[6] C.L. Hamblin. Instants and intervals. In Fraser et al. [4], pages 324–331.
[7] P. Hasle and P. Øhstrøm. Foundations of Temporal Logic – the WWW-site for Prior-studies. http://www.huminf.aau.dk/prior/.
[8] Peter Øhrstrøm and Per Hasle. *Temporal Logic. From Ancient Ideas to Artificial Intelligence*. Kluwer Academic Publishers, 1995.
[9] Peter Øhrstrøm and Per Hasle. The flow of time into logic and computer science. *Bulletin of the European Association for Theoretical Computer Science*, (82):191–226, 2004.
[10] Keisuke Ohtsuka and William F. Brewer. Discourse organization in the comprehension of temporal order in narrative texts. *Discourse Processes*, 15:317–336, 1992.
[11] A.N. Prior. *Past, Present and Future*. Oxford University Press, Oxford, 1967.
[12] A.N. Prior. *Papers on Time and Tense*. Oxford University Press, 2nd edition, 2002.
[13] Peter Röper. Intervals and tenses. *Journal of Philosophical Logic*, 9:451–469, 1980.
[14] Marie-Laure Ryan. *Possible Worlds, Artificial Intelligence, and Narrative Theory*. Indiana University Press, 1991.
[15] A.G. Walker. Durées et instants. *La Revue Scientifique*, (3266):131 ff., 1947.
[16] G.J. Whitrow. Reflections on the concept of time. In Fraser et al. [4], pages 1–11.
[17] G.J. Whitrow. *The Natural Philosophy of Time*. Oxford University Press, Oxford, 2nd edition, 1980.

# TIME INSTANT

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona

**SYNONYMS**

Event; Moment; Time Point

**DEFINITION**

A time *instant* is a single, atomic time point in the time domain.

**MAIN TEXT**

Various models of time have been proposed in the philosophical and logical literature of time. These view time, among other things, as discrete, dense, or continuous.

Instants in the dense model of time are isomorphic to the rational numbers: between any two instants there is always another. Continuous models of time are isomorphic to the real numbers, i.e., they are dense and also, unlike the rational numbers, without "gaps."

A discrete time domain is isomorphic to (a possibly bounded subset of) the natural numbers, and a specific instant of such a domain then corresponds to some natural number.

The elements of a discrete time domain are often associated with some fixed duration. For example, a time domain can be used where the time elements are specific seconds. Such time elements are often called *chronons*. In this way, a discrete time domain can approximate a dense or continuous time domain.

A time domain may be constructed from another time domain by mapping its elements to granules. In this case, multiple instants belong to the same granule, and the same granule may therefore represent different instants. For example, given a time domain of seconds, a time domain of day-long granules can be constructed.

Concerning the synonyms, the term "event" is already used widely within temporal databases, but is often given a different meaning, while the term "moment" may be confused with the distinct terms "chronon" or "granule."

**CROSS REFERENCE***

Chronon, Event, Temporal Database, Temporal Domain, Time Domain, Time Granularity, Time in Philosophical Logic

**REFERENCES***

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TIME INTERVAL

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Duration; Span; Time distance; Time period

## DEFINITION

**Definition 1:**

A *time interval* is a convex subset of the time domain. A time interval may be open or closed (at either end) and can be defined unambiguously by its two delimiting time instants. In a system that models the time domain using granules, an interval may be represented by a set of contiguous granules. See the definition of *time period*.

**Definition 2:**

An *interval* is a directed duration of time. A duration is an amount of time with known length, but no specific starting or ending instants. For example, the duration "one week" is known to have a length of seven days, but can refer to any block of seven consecutive days. An interval is either positive, denoting forward motion of time, or negative, denoting backwards motion in time.

## MAIN TEXT

Unfortunately, the term "time interval" is being used in the literature with two distinct meanings: as the time between two instants, in the general database research literature and beyond, and as a directed duration of time, in the SQL database language. The term "time period" is associated with the first definition above.

Definition 1 is recommended for non-SQL-related scientific work. Definition 2 is recommended for SQL-related work.

Concerning the synonyms, the unambiguous term "span" has been used previously in the research literature, but its use seems to be less widespread than "interval." While precise, the term "time distance" is also less commonly used. A "duration" is generally considered to be non-directional, i.e., always positive.

## CROSS REFERENCE*

Temporal Database, Temporal Granularity, Time Instant, Time Period, Time Span

## REFERENCES*

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

N. A. Lorentzos and Y. G. Mitsopoulos, SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering* 9(3):480–499, 1997.

**TITLE: Time period**

Nikos A. Lorentzos
Informatics Laboratory
Science Department
Agricultural University of Athens
Iera Odos 75, 11855 Athens, Greece
http://www.aua.gr/~lorentzos
lorentzos@aua.gr

**SYNONYMS:** Time interval

**DEFINITION**
If the time domain is a totally ordered set T = {$t_1$, $t_2$, $t_3$ …} then *a time period* over T is defined as a convex subset of elements from T.

*Example*: If T = {$d_1$, $d_2$, $d_3$, … }, where $d_i$ are consecutive dates, then [$d_{10}$, $d_{20}$] and [$d_{30}$, $d_{80}$] represent two time periods over T.

**MAIN TEXT**
In the area of temporal databases, a time period over T is usually defined as a distinct data type. Some researchers define a time period data type of the form [$t_p$, $t_q$]. Some others define such a data type of the form [$t_p$, $t_q$), i.e. its right end is closed.

Note that, initially, the term *time interval* was used instead of *time period*. This was later abandoned in order to avoid confusion, given that *interval* is a reserved word in SQL. Instead, time interval in today used with a different meaning (see time interval).

**CROSS REFERENCES**
Time Domain, Temporal Granularity, Absolute time, Crhonon, Period-stamped data model.

# Time Series Query

Like Gao
Teradata Corporation, Like.Gao@teradata.com
X. Sean Wang
Department of Computer Science, University of Vermont, Sean.Wang@uvm.edu
http://www.cs.uvm.edu/∼xywang

## SYNONYMS

Time sequence query; Time series search; Time sequence search

## DEFINITION

A time series query refers to one that finds, from a set of time series, the time series or subseries that satisfy the given search criteria. *Time series* are sequences of data points spaced at strictly increasing times. The search criteria are domain specific rules defined with time series statistics or models, temporal dependencies, similarity between time series or patterns, etc. In particular, similarity queries are of great importance for many real world applications like stock analysis, weather forecasting, network traffic monitoring, etc., which often involve high volume of time series data and may use different similarity measures or pattern descriptions. In many cases, query processing consists of evaluating these queries in real-time or quasi-real time by using time series approximation techniques, indexing methods, incremental computation, and specialized searching strategies.

## HISTORICAL BACKGROUND

Time series queries play a key role in temporal data mining applications such as time series analysis and forecasting. It was in the recent years that these applications with massive time series data became possible due to the rapidly emerging query processing techniques, especially those for similarity queries. In 1993, Rakesh Agrawal et al. [1] proposed an indexing method for processing similarity queries in sequence databases. The key was to map time series to a lower dimensionality space by only using the first few Fourier coefficients of the time series, and build R*-trees to index the time series. This work laid out a general approaches of using indexes to answer similarity queries with time series. In 1994, Christos Faloutsos et al. [5] extended this work to subsequence matching and proposed the GEMINI framework for indexing time series. In 1997, Davood Rafiei and Alberto Mendelzon [12] proposed a set of linear transformations on the Fourier series representation of a time series that can be used as the basis for similarity queries. Subsequent work have focused on new dimensionality reduction techniques [9, 10, 8], new similarity measures [4, 14, 2], and queries over streaming time series [15, 6, 3].

## SCIENTIFIC FUNDAMENTALS

### Basic Concepts

A *time series* $x$ is a sequence of data points spaced at strictly increasing time. The number of elements in the sequence is called the *length* of the time series. The data points are often real numbers, and therefore $x$ can often be represented as a vector of real numbers, $x(n) = \langle x_{t_1}, \ldots, x_{t_n} \rangle$, where $n$ is the length of $x$, and each $t_i$ is the timestamp associated with the data point with $t_i < t_{i+1}$ for $i = 1, \ldots, n-1$. The time intervals between successive data points are usually, but not always, assumed uniform, and hence $t_{i+1} - t_i$ is often assumed a constant for $i = 1, \ldots, n-1$. When the specific $t_i$ values are irrelevant but only signify the temporal order, $x$ is also represented as $x(n) = \langle x_1, \ldots, x_n \rangle$. A subsequence of $x$ that consists of consecutive elements is called a *subseries* or *segment* of $x$. Time series normally refer to those finite sequences of *static* elements. If the sequence has new elements continuously appended over time, they are specially called *streaming* time series [6].

Raw time series often need pre-processing to fit the application needs. It is possible to perform the following pre-processing to remove irregularities of time series. *Time regulation* is to convert a non-uniform time series to a uniform one with interpolation functions to obtain the elements at uniform time intervals. *Normalization* is to make the distance between two time series invariant to offset shifting and/or amplitude scaling. For

example, given time series $x$, its mean $\bar{x}$ and standard deviation $\sigma_x$, the normalization function can be either $\tilde{x} = (x - \bar{x})$ or $\tilde{x} = (x - \bar{x})/\sigma_x$. *Linear trend reduction* is to remove the seasonal trend impact on time series, e.g., $\tilde{x}_{t_i} = x_{t_i} - (a * t_i + b)$ for all $i$. To reduce data noise, *smoothing techniques* such as moving average can also be applied.

Similarity is the degree of resemblance between two time series, and the choice of similarity measure is highly domain dependent. For applications without value scaling and time shifting concerns, a simple $L_p$-norm Distance, of which $L_1$ and $L_2$ are the well known Manhattan and Euclidean Distances, respectively, is sufficient. For other applications, more robust similarity measures may be needed, such as scale invariant distances (such as correlation), warping distances that allow an elastic time shifting (such as DTW or Dynamic Time Warping, Longest Common Subsequence and Spatial Assembling Distances [3]), Edit Distance With Real Penalty(ERP) [2], and model based and histogram based distances. Since most similarity measures are defined non-negative, and the smaller the values, the closer the time series, the notions of similarity and distance are often used interchangeably.

**Example 1** (`$L_p$-norm Distances, a.k.a. Minkowski Distance`): Given time series $x(n)$ and $y(n)$, and positive integer $p$, let

$$L_p(x, y) = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p} .$$

A special case is given as $L_\infty(x, y) = \max\{|x_i - y_i|, i = 1, \ldots, n\}$. Note when $p \to \infty$, $L_p(x, y) = L_\infty(x, y)$. □

**Example 2** (`DTW Distance`): Given time series $x(m)$ and $y(n)$, then recursively, for all $1 \le i \le m$ and $1 \le j \le n$

$$DTW(x(i), y(j)) = d(x_i, y_j) + \min\{DTW(x(i-1), y(j-1)), DTW(x(i-1), y(j)), DTW(x(i), y(j-1))\},$$

where $d()$ is a distance function defined on two elements $x_i$ and $y_j$, and $x(i)$ and $y(j)$ denote the prefixes of the time series $x(m)$ and $y(n)$ of lengths $i$ and $j$, respectively. The base case of the above is when $i = j = 1$ in which case $DTW(x(1), y(1)) = d(x_1, y_1)$. When both $i$ and $j$ are 0, $DTW(x(0), y(0))$ is defined as 0. Otherwise, when either $i$ or $j$ is out of range, $DTW(x(i), y(j))$ is defined as $+\infty$. DTW is usually accompanied with one of the following global constraints, in regard to the two prefix lengths $i$ and $j$.

`Sakoe-Chiba Band`: The allowed range of $i$ and $j$ in the definition above satisfy $|j - i| \le r$ for some $r \ge 0$.

`Itakura Parallelogram`: Use the constraint $g(i) \le j - i \le f(i)$ for $i$ and $j$, where $f$ and $g$ are functions of $i$ such that the allowed region given by the constraint shows a parallelogram shape with two opposing corners at $(0, 0)$ and $(m, n)$, respectively. □

**Example 3** (`Edit Distance With Real Penalty [2]`): Given time series $x(m)$ and $y(n)$, recursively for all $i \le m$ and $j \le n$, let

$$ERP(x(i), y(j)) = \min \begin{cases} ERP(x(i-1), y(j-1)) + d(x_i, y_j) \\ ERP(x(i-1), y(j)) + d(x_i, g) \\ ERP(x(i), y(j-1)) + d(g, y_j) \end{cases}$$

In the above, $g$ is a constant value (and can be 0), $x(i) = y(j) = \langle g \rangle$ (i.e., a time series of length 1) is assumed for all $i \le 0$ and $j \le 0$, and $d(a, b) = |a - b|$. The base case of the above is when both argument time series are of length 1 in which case $ERP(x, y) = d(x_1, y_1)$. Intuitively, the constant $g$ is used to fill a *gap* referring to an added element. □

**Time Series Query**

Many forms of time series queries have been proposed over the years. Among them, one of the mostly used is *similarity search*, defined as follows: Given a set of candidate time series $X$, a similarity measure $D$, and a query series $y$, (1) find all series in $X$ whose distance to $y$ is within a given threshold (near neighbor query), and (2) find $k$ series in $X$ that are closest to $y$ ($k$-nearest neighbor query). Other queries are also possible, e.g., all pairs query that finds, in $X$, all pairs of series with distance below a given threshold. Besides similarity search, other types of queries include detecting elastic burst over streaming time series, retrieving values at any arbitrary time [13], etc. In the following, time series query refers to similarity search.

The time series query can be either *whole series matching* or *subseries matching*. The former refers to the query that concerns the whole time series, both for the query series $y$ and each candidate series $x$ in $X$. The latter concerns the subseries, which have the same length as that of $y$, of all $x$ in $X$. For example, given time series $y(l)$, for each $x(n) \in X$, the latter query needs to consider $x(i + 1, i + l) = \langle x_{i+1}, \ldots, x_{i+l} \rangle$ for all $0 \le i \le n - l$.

In the above definition, if the query object is a set of *patterns*, the query is called *pattern matching*. A pattern is an abstract data model of time series, often seen as a short sequence, representing a class of time series that have the same properties. For pattern matching, all the involved time series may be mapped to the space in which the patterns and the similarity measure are defined.

Like the notion of time series, time series queries by default refer to those with static time series. In case of streaming time series, the queries are often monitoring the subseries within a sliding widows, and need to be evaluated periodically or continually to identify the similar series or those with the given patterns [6, 15].

**Query Processing: Index-based Methods for Similarity Search**

Due to large volumes of data and the complexity of similarity measures, directly evaluating time series queries is both I/O and CPU intensive. There are many approaches to process these queries efficiently. Among them, one is to convert time series to other data types (e.g., strings and DNA sequences), so that the corresponding search techniques (e.g., string matching and DNA sequence matching) can be applied. The reader is referred to these entries in this Encyclopedia. Another approach is to index time series based on their approximations (or *features*), which is detailed in the following.

Time series $x$ of length $n$ can be viewed as a point in an $n$-dimensional space. However, spatial access methods such as *kd*-tree and *R*-tree cannot be used to index the time series directly. The problem is due to the *dimensionality curse* – the performance of spatial access methods degrades rapidly once the dimensionality is above 16, while $n$ is normally much larger than this number.

The general solution is to map time series to points in a lower $N$-dimensional space ($N << n$) and then construct the indexes in this space. The mapped points are called the time series *approximation*. Each dimension of the $N$-dimensional space represents one characteristic of the time series, such as mean, variance, slope, peak values, or the Fourier coefficient, at some coarse levels of time granularity and possibly within a shifted time window. Further, the domain of the $N$-dimensional space can be nominal so the time series approximation can be represented as a symbolic sequence [11].

**Example 4** `(Piecewise Aggregate/Constant Approximation [7, 14])`: Time series $x$ of length $mN$ can be mapped to a point in the $N$-dimensional space: $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_N)$ where the value in the $i^{th}$ dimension is the mean over the $i^{th}$ segment of $x$, $\bar{x}_i = \frac{1}{m}\sum_{j=m(i-1)+1}^{mi} x_j$. □

**Example 5** `(Line Fitting Approximation [11])`: Given an alphabet $\mathcal{A}\{$"*up*", "*down*", "$flat$"$\}$, define a mapping function $s(z) \in \mathcal{A}$ where $z$ is a time series of length $m$. Time series $x$ of length $mN$ can be mapped to a length-$N$ symbolic sequence, $\langle s(x_1, \ldots, x_m), s(x_{m+1}, \ldots, x_{2m}), \ldots, s(x_{(N-1)m+1}, \ldots, x_{mN})\rangle$, e.g., $\langle$"*up*", "*up*", ..., "*down*"$\rangle$. Function $s$ can be line fitting and, based on the slop of the fitting line, decides if the value is "*up*" or "*down*" etc. □

A multi-step algorithm can be used to process a query. Take the $k$-nearest neighbor search as an example. First step: find $k$-nearest neighbors in the lower-dimensional index structure. Find the actual distance between the query series and the $k^{th}$ nearest neighbor. Second step: Use this actual distance as a range query to find (on the lower-dimensional index) all the database points. Third step: calculate the actual distances found in step 2 and obtain the actual $k$-nearest neighbors. An improvement to this algorithm is to incrementally obtain nearest neighbors in the lower-dimensional approximation, and each time an actual distance is obtained, it is used to remove some database points that are returned by the range query (third step above).

The index-based methods need to guarantee soundness (no false alarms) and completeness (no false dismissals) in the query result. Soundness can be guaranteed by checking the original data as in step 3. The completeness can be guaranteed only if the chosen approximation method has the *lower-bounding property* [5]. That is, given a similarity measure $D$, for any candidate time series $x$ and query time series $y$, let $\bar{x}$ and $\bar{y}$ be their lower dimensional approximations and $D'$ be the distance defined on $\bar{x}$ and $\bar{y}$, then $D'(\bar{y}, \bar{x}) \leq D(y, x)$ must hold for any $x$ and $y$.

**Example 6** `(Lower Bounding Approximation for Euclidian Distance)`: Method (1): apply an orthonormal transform (Fourier transform, Wavelet transform, and SVD) to both query and candidate time series and ignore many "insignificant" axes after the transform. The distance defined on the remaining axes gives the lower bounding approximation for Euclidian Distance [1]. Method (2): apply segmented mean approximation to both query and candidate time series. It's easy to see $\sqrt[p]{m}L_p(\bar{x}, \bar{y}) \leq L_p(x, y)$ for all $p$, while $m$ is the factor of dimensionality reduction, i.e., $x$'s length divided by $\bar{x}$'s. Since this low bounding works for all $p$, one index tree can be used for all $L_p$-norm distances [7, 14]. □

**Example 7** `(Lower Bounding Approximation for DTW Distance [8])`: To derive a lower bounding approxi-

mation for DTW, approximate the candidate time series $x$ using segmented mean approximation, and approximate the query time series $y$ as follows. Let $y = \langle y_1, \ldots, y_{mN} \rangle$, define $U = \langle U_1, \ldots, U_{mN} \rangle$ and $L = \langle L_1, \ldots, L_{mN} \rangle$, where $U_i = \max(y_{i-r}, \ldots, y_{i+r})$ and $L_i = \min(y_{i-r}, \ldots, y_{i+r})$ ($r$ is the allowed range for $m - n$ in Sakoe-Chiba Band or Itakura Parallelogram). Sequences $U$ and $L$ form a bounding envelope that encloses $y$ from above and below. Then reduce $U$ and $L$ to a lower dimension $N$, define $\hat{U} = \langle \hat{U}_1, \ldots, \hat{U}_N \rangle$ and $\hat{L} = \langle \hat{L}_1, \ldots, \hat{L}_N \rangle$, where $\hat{U}_i = \max(U_{(i-1)m+1}, \ldots, U_{im})$ and $\hat{L}_i = \min(L_{(i-1)m}, \ldots, L_{im})$, that is, $\hat{U}$ and $\hat{L}$ are piecewise constant functions that bound $U$ and $L$, respectively. Let

$$LB\_PAA(y, \bar{x}) = \sqrt{m \sum_{i=1}^{N} \begin{cases} (\bar{x}_i - \hat{U}_i)^2 & \text{if } \bar{x}_i > \hat{U}_i; \\ (\bar{x}_i - \hat{L}_i)^2 & \text{if } \bar{x}_i < \hat{L}_i; \\ 0 & \text{otherwise.} \end{cases}},$$

then $LB\_PAA(y, \bar{x}) \leq DTW(y, x)$. In this case, $\bar{y} = y$. □

**Query Processing: Similarity Search over Streaming Time Series**

These queries are different from those with static time series, in that 1) having a sliding window or windows of multiple lengths at the same time; 2) continuous monitoring; and 3) incremental evaluation. In the following, consider the two problems: Euclidean distance or correlation monitoring among pairs of streams, and Euclidean distance or correlation monitoring between a stream time series and a time series database.

The first query problem is, given many streaming time series, how to find pairs of time series that have strong (positive or negative) correlations in the last sliding window [15].

The idea is to use the notion of "basic windows", similar to segmented mean application. Instead of mean, the coefficients of the Fourier transform of each segment is used to approximate the time series. Given $x = \langle x_1, \ldots, x_b \rangle$ and $y = \langle y_1, \ldots, y_b \rangle$, where $b$ is the size of the basic window. If $x_i = \sum_{m=0}^{N-1} C_m^x f_m(i)$ and $y_i = \sum_{m=0}^{N-1} C_m^y f_m(i)$, assuming $f_m$ is a family of orthogonal functions, then the inner product of $x$ and $y$,

$$x * y = \sum_{i=1}^{b} x_i y_i = \sum_{i=1}^{b} (\sum_{m=0}^{N-1} C_m^x f_m(i) \sum_{p=0}^{N-1} C_p^y f_p(i)) = \sum_{m=0}^{N-1} \sum_{p=0}^{N-1} C_m^x C_p^y (\sum_{i=1}^{b} f_m(i) f_p(i)).$$

Note $\sum_{i=1}^{b} f_m(i) f_p(i)$ does not depend on $x$ and $y$ and can be pre-computed. From this, the inner product of two time series can be computed for each sliding window aligned with the basic windows (i.e., a sliding window must be the union of some basic windows). Fourier bases can be used as the $f$ functions, and discrete Fourier transform (DFT) can compute the coefficients efficiently in an incremental way.

By only taking a few Fourier coefficients (small $N$), the approximate inner products and hence the Euclidean distance can be evaluated efficiently. To compute correlations, the normalized series of $\hat{x}_i = (x_i - \bar{x})/\sigma_x$ need only be considered, where $\bar{x}$ and $\sigma_x$ are the mean and standard deviation of $x$ over the sliding window.

A step further: since two series are highly correlated if their DFT coefficients are similar, an indexing structure can help to store the coefficients and look for series with high correlation only in series with similar coefficients.

The second query problem is, given a database of pattern time series, a streaming time series, and window size $N$, how to find the nearest neighbor of the streaming time series (using the last $N$ values) in the database, at each time position [6].

The idea is a batch processing that uses fast Fourier transform (FFT) and its inverse to calculate the cross correlation of streaming time series and patterns at many time positions. Given $x = \langle x_1, \ldots, x_N \rangle$ and $y = \langle y_1, \ldots, y_N \rangle$, the circular cross correlation sequence is defined as

$$CirCCorr_d^{x,y} = \sum_{i=1}^{N} x_{(d+i-1) \bmod N} y_i, \quad d = 1, 2, \ldots, N,$$

where $d$ is the time lag. Let $\dot{x}$ and $\dot{y}$ be the DFT transforms of $x$ and $y$ respectively, then sequence $\langle \dot{x}_1 \dot{y}_1^*, \ldots, \dot{x}_N \dot{y}_N^* \rangle$ is the result of DFT transform of $CirCCorr^{x,y}$. Here $\dot{y}_i^*$ is so-called conjugate of $\dot{y}_i$.

With the CirCCorr, calculation of the Euclidean distances of a number of time positions can be done in a batch mode. This is faster than calculating the individual distances, as the batch process has time complexity $O(N \lg N)$, as compared to the direct computing of $O(Nl)$, where $l$ ($l < N$) is the number of time positions covered by one batch processing. So it is profitable to wait for a few time steps and then find the nearest neighbors for these time

steps all together. The longer the wait is, the more computation time saved. However, this causes a lengthening of the response time, i.e., a loss of the chance of finding the answer as early as possible. To overcome this, one may use a certain model to roughly predict the future values of the time series and apply the batch processing to compute all the Euclidean distance (or correlations) of many future time positions. When the actual values come, triangular inequality can filter out a lot of time series in the database that are not the nearest neighbor [6].

**KEY APPLICATIONS***

Market data analysis and trend predication, network traffic control, intrusion detection, temporal data mining

**DATA SETS***

1) UCR Time Series Classification/Clustering Page: http://www.cs.ucr.edu/∼eamonn/time_series_data/

2) PhysioBank: Physiologic Signal Archives for Biomedical Research (including ECG and synthetic time series with known characteristics): http://www.physionet.org/physiobank/

**URL TO CODE***

1) Above URL 1.

2) ANN: A Library for Approximate Nearest Neighbor Searching: http://www.cs.umd.edu/∼mount/ANN/

**CROSS REFERENCE***

Temporal data mining, Dimensionality Curse, Dimensionality Reduction Techniques, R-tree and its variants, kd-tree, Sequential patterns, Singular Value Decomposition, Discrete Wavelet Transformation, Indexing, Indexing and Similarity Search, Multidimensional Indexing, Nearest Neighbor Query, Range query, Stream Similarity Mining, Top-K Selection Queries on Multimedia Data Sets, Multi-Step Query Processing

**RECOMMENDED READING**

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient Similarity Search In Sequence Databases. In *Proc. 4th International Conference of Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, 1993.

[2] Lei Chen and Raymond Ng. On the marriage of lp-norms and edit distance. In *VLDB Conference*, pages 792–803, 2004.

[3] Yueguo Chen, Mario A. Nascimento, Beng Chin Ooi, and Anthony K. H. Tung. Spade: On shape-based pattern detection in streaming time series. In *ICDE*, pages 786–795, 2007.

[4] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Finding similar time series. In *Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.

[5] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD Conference*, pages 419–429, 1994.

[6] Like Gao and X. Sean Wang. Continuous similarity-based queries on streaming time series. *IEEE Trans. Knowl. Data Eng.*, 17(10):1320–1332, 2005.

[7] Eamonn J. Keogh and Michael J. Pazzani. Scaling up dynamic time warping for datamining applications. In *Knowledge Discovery and Data Mining*, pages 285–289, 2000.

[8] Eamonn J. Keogh and Chotirat (Ann) Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[9] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD Conference*, pages 289–300, 1997.

[10] Kin pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[11] Yunyao Qu, Changzhou Wang, Like Gao, and X. Sean Wang. Supporting movement pattern queries in user-specified scales. *IEEE Trans. Knowl. Data Eng.*, 15(1):26–42, 2003.

[12] Davood Rafiei and Alberto Mendelzon. Similarity-based queries for time series data. In *SIGMOD Conference*, pages 13–25, New York, NY, USA, 1997. ACM.

[13] P. Revesz, R. Chen, and M. Ouyang. Approximate query evaluation using linear constraint databases. In *Proc. Eight International Symposium on Temporal Representation and Reasoning*, pages 170–175, 2001.

[14] Byoung-Kee Yi and Christos Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *The VLDB Journal*, pages 385–394, 2000.

[15] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB Conference*, pages 358–369, 2002.

# TIME SPAN

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Time interval; Time distance

**DEFINITION**

A span is a directed duration of time. A duration is an amount of time with known length, but no specific starting or ending instants. For example, the duration "one week" is known to have a length of seven days, but can refer to any block of seven consecutive days. A span is either positive, denoting forward motion of time, or negative, denoting backwards motion in time.

**MAIN TEXT**

Concerning the synonyms, the terms "time interval" is generally understood to denote an anchored span in the general community of computer science. Only in the SQL language does "time interval" denote a span. The term "span," which has only one definition, is thus recommended over "time interval" for works not related to the SQL language. This use is unambiguous.

A "duration" is generally considered to be non-directional, i.e., always positive. The term "time distance" is precise, but is longer.

**CROSS REFERENCE***

Fixed Span, Temporal Database, Time Instant, Time Interval, Variable Span

**REFERENCES***

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TITLE

time-line clock

# BYLINE

Curtis Dyreson
Utah State University
Curtis.Dyreson@usu.edu
http://www.cs.usu.edu/~cdyreson

# SYNONYMS

Clock, base-line clock, time-segment clock

# DEFINITION

In the discrete model of time, a *time-line clock* is defined as a set of *physical clocks* coupled with some specification of when each physical clock is authoritative. Each *chronon* in a time-line clock is a chronon (or a regular division of a chronon) in an identified, underlying physical clock. The time-line clock switches from one physical clock to the next at a *synchronization point*. A synchronization point correlates two, distinct physical clock measurements.

# MAIN TEXT

A time-line clock is the clock for (concrete) times stored in a temporal database. A time-line clock glues together a sequence of physical clocks to provide a consistent, clear semantics for a time-line. Since the range of most physical clocks is limited, a time-line clock is usually composed of many physical clocks. For instance, a tree-ring clock can only be used to date past events, and the atomic clock can only be used to date events since the 1950s. Though several physical clocks might be needed to build a time-line, in some cases a single physical clock suffices. For instance SQL2 uses the mean solar day clock—the basis of the *Gregorian calendar*—as its time-line clock.

# CROSS REFERENCES

Chronon, Physical Clock, Time Instant

# REFERENCES

J. T. Fraser. Time: The Familiar Stranger. The University of Massachusetts Press, 1987. 408 pages.

Curtis E. Dyreson and Richard T. Snodgrass. The Baseline Clock. *The TSQL2 Temporal Query Language*, Kluwer, 1995: 73-92.

Curtis E. Dyreson, Richard T. Snodgrass: Timestamp semantics and representation. Information Systems **18**(3): 143-166 (1993).

# TIMESLICE OPERATOR

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Rollback operator; State query

## DEFINITION

The *valid-timeslice operator* may be applied to any temporal relation that captures valid time. Given also a valid-time element as a parameter, it returns the argument relation reduced in the valid-time dimension to just those times specified by the valid-time element. The *transaction timeslice operator* is defined similarly, with the exception that the argument relation must capture transaction time.

## MAIN TEXT

Several types of timeslice operators are possible. Some may restrict the time parameter to intervals or instants. Some operators may, given an instant parameter, return a conventional relation or a transaction-time relation when applied to a valid-time or a bitemporal relation, respectively; other operators may always return a result relation of the same type as the argument relation.

Oracle supports timeslicing through its flashback queries. Such queries can retrieve all the versions of a row between two transaction times (a key-transaction-time-range query) and allows tables and databases to be rolled back to a previous transaction time, discarding all changes after that time.

Concerning the synonyms, "rollback operator" is an early term that has now been abandoned. This term indicates that the result of a timeslice is a relation obtained by moving backwards in time, presumably from the current transaction time. This kind of result is less general than those that may be obtained using a timeslice operator. Specifically, this kind of result assumes a time parameter that extends from the beginning of the time domain to some past time (with respect to the current time). Similarly, "state query" suggests a less general functionality than what is actually offered by timeslice operators.

## CROSS REFERENCE*

Bitemporal Relation, Temporal Database, Temporal Element, Temporal Query Languages, Time Instant, Time Interval, Transaction Time, TSQL2, Valid Time

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# TRANSACTION TIME

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Registration time; Extrinsic time; Physical time; Transaction commit time; Belief time

**DEFINITION**

A database fact is stored in a database at some point in time, and after it is stored, it remains current, or part of the current database state, until it is logically deleted. The *transaction time* of a database fact is the time when the fact is current in the database. As a consequence, the transaction time of a fact is generally not a time instant, but rather has duration.

The transaction time of a fact cannot extend into the future. Also, as it is impossible to change the past, meaning that (past) transaction times cannot be changed.

In the context of a database management system that supports user transactions, the transaction times of facts are consistent with the serialization order of the transactions that inserted or logically deleted them. Transaction times may be implemented using transaction commit times, and are system-generated and -supplied.

**MAIN TEXT**

A database is normally understood to contain statements that can be assigned a truth value, also called facts, that are about the reality modeled by the database and that hold true during some non-empty part of the time domain. Transaction times, like valid times, may be associated with such facts. It may also be noted that it is possible for a database to contain the following to different, albeit related, facts: a non-timestamped fact and that fact timestamped with a valid time. The first would belong to a snapshot relation, and the second would belong to a valid-time relation. Both of these facts may be assigned a transaction timestamp. The resulting facts would then be stored in relations that also support transaction time.

A transaction time database is append-only and thus ever-growing. To remove data from such a database, *temporal vacuuming* may be applied.

The term "transaction time" has the advantage of being almost universally accepted and it has no conflicts with to other important temporal aspect of data, valid time.

Oracle explicitly supports transaction time. Applications can access prior transaction-time states of their database, by means of transaction timeslice queries. Database modifications and conventional queries are temporally upward compatible.

Concerning the alternatives, the term "registration time" seems to be straightforward. However, this term may leave the impression that the transaction time is only the time instant when a fact is inserted into the database. "Extrinsic time" is rarely used. "Physical time" is also used infrequently and seems vague. "Transaction commit time" is lengthy, but more importantly, the term appears to indicate that the transaction time associated with a fact must be identical to the time when that fact is committed to the database, which is an unnecessary restriction. The term is also misleading because the transaction time of a fact is not a single time instant as implied. The term "belief time" stems from the view that the current database state represents the current belief about the aspects of reality being captured by the database. This term is used infrequently.

**CROSS REFERENCE***

Supporting Transaction Time Databases, Temporal Compatibility, Temporal Database, Temporal Generalization, Temporal Specialization, Temporal Vacuuming, Timeslice Query, Transaction-Time Indexing, User-Defined Time, Valid Time

**REFERENCES***

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

R. T. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Austin, TX, May 1985, pp. 236–246.

R. T. Snodgrass and I. Ahn, "Temporal Databases," *IEEE Computer*, Vol. 19, No. 9, September, 1986, pp. 35–42.

# TRANSACTION-TIME INDEXING

Mirella M. Moro
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
http://www.inf.ufrgs.br/~mirella/

Vassilis J. Tsotras
University of California, Riverside
Riverside, CA 92521, USA
http://www.cs.ucr.edu/~tsotras

## SYNONYMS

Transaction-Time Access Methods

## DEFINITION

A transaction-time index is a temporal index that enables fast access to transaction-time datasets. In a traditional database, an index is used for selection queries. When accessing transaction-time databases, selection queries also involve the transaction-time dimension. The characteristics of the transaction-time axis imply various properties that such temporal index should have to be efficient. As with traditional indices, the performance is described by three costs: (i) storage cost (i.e., the number of pages the index occupies on the disk), (ii) update cost (the number of pages accessed to perform an update on the index; for example when adding, deleting or updating a record) and (iii) query cost (the number of pages accessed for the index to answer a query).

## HISTORICAL BACKGROUND

Most of the early work on temporal indexing has concentrated on providing solutions for transaction-time databases. A basic property of transaction-time is that it always *increases*. This is consistent with the serialization order of transactions in a database system. Each newly recorded piece of data is time-stamped with a new, larger, transaction time. The immediate implication of this property is that previous transaction times *cannot* be changed (since every new change must be stamped with a new, larger, transaction time). This is useful for applications where every action must be registered and maintained unchanged after registration, as in auditing, billing, etc. Note that a transaction-time database records the history of a database activity rather than "real" world history. As such it can “rollback” to, or answer queries for, any of its previous states.

Consider, for example, a query on a temporal relation as it was at a given transaction time. There are two obvious, but inefficient approaches to support this query, namely the “copy” and “log” approaches. In the “copy” approach, the whole relation is “flushed” (copied) to disk for every transaction for which a new record is added or modified. Answering the above rollback query is simple: the system has to then query the copy that has the largest transaction-time less or equal to the requested time. Nevertheless, this approach is inefficient for its storage and update costs (the storage can easily become quadratic to the number of records in the temporal database and the update is linear, since the whole temporal relation needs to be flushed to disk for a single record update). In contrast, the “log” solution simply maintains a log of the updates to the temporal database. Clearly, this approach uses minimal space (linear to the number of updates) and minimal update cost (simply add an update record at the end of the log), but the query time is prohibitively large since the whole log may need to be traversed for reconstructing a past state of the temporal database. Various early works on transaction-time indexing behave asymptotically like the “log” or the “copy” approaches. For a worst-case comparison of these methods see [7]. Later on, two methodologies were proposed to construct more efficient transaction-time indices, namely the (i) *overlapping* [3,10] and (ii) (partially) *persistent* approaches [6,9,1]. These methodologies attempt to combine the benefits of the fast query time from the “copy” approach with the low space and update costs of the “log” approach.

## SCIENTIFIC FUNDAMENTALS

The distinct properties of the transaction-time dimension and their implications to the index design are discussed through an example; this discussion has been influenced by [8]. Consider an initially empty collection of objects. This collection evolves over time as changes are applied. Time is assumed discrete and always increasing. A change is the addition or deletion of an object, or the value change of an object's attribute. A real life example would be the evolution of the employees in a company. Each employee has a surrogate (*ssn*) and a salary attribute. The changes include additions of new employees

(as they are hired or re-hired), salary changes or employee deletions (as they retire or leave the company). Each change is time-stamped with the time it occurs (if more than one change happen at a given time, all of them get the same timestamp). Note that an object attribute value change can be simply "seen" as the artificial deletion of the object followed by the simultaneous rebirth (at the same time instant) of this object having the modified attribute value. Hence, the following discussion concentrates only on object additions or deletions.

In this example, an object is called "alive" from the time that it is added in the collection until (if ever) it is deleted from it. The set *s(t)*, consisting of all alive objects at time *t*, forms the state of the evolving collection at *t*. Figure 1 illustrates a representative evolution shown as of time *t* = 53. Lines ending to ">" correspond to objects that have not yet been deleted at *t* = 53. For simplicity, at most one change per time instant is assumed. For example, at time *t* = 10 the state is *s*(10) = *{u, f, c}.* The interval created by the consecutive time instants an object is alive is the "lifetime" interval for this object. Note that the term "interval" is used here to mean a "convex subset of the time domain" (and not a "directed duration"). This concept has also been named a "period"; in this discussion however, only the term "interval" is used. In Figure 1, the lifetime interval for object *b* is [2,10). An object can have many non-overlapping lifetime intervals.

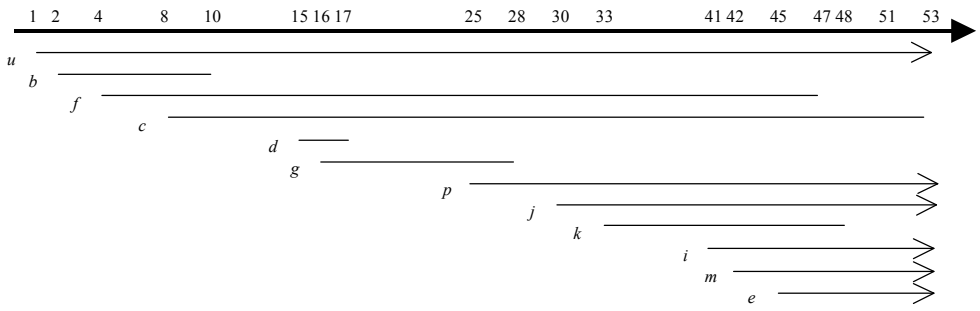



Figure 1. An example of a transaction-time evolution.

Note that in the above evolving set example, changes are always applied to the current state *s(t),* i.e., past states *cannot* be changed. That is, at time *t* = 17, the deletion of object *d* is applied to *s*(16) = *{u, f, c, d, g}* to create *s*(17) = *{u, f, c, g}.* This implies that, at time *t* = 54, no object can be retroactively added to state *s*(5), neither the interval of object *d* can be changed to become [15, 25). All such changes are not allowed as they would affect previous states and not the most current state *s*(53).

Assume that all the states *s(t)* of the above evolution need to be stored in a database. Since time is always increasing and the past is unchanged, a transaction time database can be utilized with the implicit updating assumption that when an object is added or deleted from the evolving set at time *t*, a transaction updates the database system about this change at the same time, i.e., this transaction has commit timestamp *t*. When a new object is added in the collection at time *t*, a record representing this object is stored in the database accompanied by a transaction-time interval of the form *[t, UC).* In this setting, *UC* (Until Changed) is a variable representing the fact that at the time the object is added in the collection, it is not yet known when (if ever) it will be deleted from it. If this object is later deleted at time *t'*, the transaction-time interval of the corresponding record is updated to *[t, t').* A real-world object deletion is thus represented in the database as a "logical" deletion: the record of the deleted object is still retained in the database, accompanied by an appropriate transaction-time interval. Since the past is kept, a transaction-time database conceptually stores, and can thus answer queries about, any past state *s(t)*.

Based on the above discussion, an index for a transaction-time database should have the following properties: (a) store past logical states, (b) support addition/deletion/modification changes on the objects of the current logical state, and (c) efficiently access and query any database state.

Since a fact can be entered in the database at a different time than when it happened in reality, the transaction-time interval associated with a record is actually related to the process of updating the database (the database activity) and may not accurately represent the times the corresponding object

was valid in reality. Note that a valid-time database has a different abstraction, which can be visualized as a dynamic collection of "interval-objects". The term interval-object is used to emphasize that the object carries a valid-time interval to represent the temporal validity of some object property. Reality is more accurately represented if both time dimensions are supported. A bi-temporal database has the characteristics of both approaches. Its abstraction maintains the evolution (through the support of transaction-time) of a dynamic collection of (valid-time) interval-objects. The reader is referred to the Valid-time Indexing and Bi-temporal Indexing entries of this encyclopedia for further reading.

Traditional indices like the $B^+$-tree or the R-tree are not efficient for transaction-time databases because they do not take advantage of the special characteristics of transaction time (i.e., that transaction time is always increasing and that changes are always applied on the latest database state). There are various index proposals that are based on the (partially) *persistent* data-structure approach; examples are the Time-Split B-tree (TSB) [6], the Multiversion B-tree (MVBT) [1], the Multiversion Access Structure [11], the Snapshot Index [9], etc. It should be noted that all the above approaches facilitate "time-splits": when a page gets full, current records from this page are copied to a new page (this operation is explained in detail below). Time-splits were first introduced in the Write-Once B-tree (WOBT), a B-tree index proposed for write-once disks [5]. Later, the Time-Split B-tree used time-splits for read-write media and also introduced other splitting policies (e.g., splitting by other than the current time, key splits etc.) Both the WOBT and TSB use deletion markers when records are deleted and do not consolidate pages with few current records. The MVBT uses the time splitting approach of the WOBT, drops the deletion markers and consolidated pages with few current records. It thus achieves the best asymptotic behavior and it is discussed in detail below. Among the index solutions based on the *overlapping* data-structure approach [2], the Overlapping B-tree [10] is used as a representative and discussed further.

For the purposes of this discussion, the so-called *range-timeslice* query is considered, which provides a key range and a specific time instant selection. For example: "find all objects with keys in range [$K_1$, $K_2$] whose lifetimes contain time instant *t*". This corresponds to a query "find the employees with ids in range [100, .., 500] whose entries were in the database on July 1st, 2007". Let *n* be the total number of updates in a transaction-time database; note that *n* corresponds to the minimal information needed for storing the whole evolution. [7] presents a lower bound for answering a range-timeslice query. In particular, any method that uses linear space (i.e, $O(n/B)$ pages, where *B* is the number of object records that fit in a page) would need $O(\log_B n + s/B)$ I/O's to answer such a query (where an I/O transfers one page, and *s* corresponds to the size of the answer, i.e., the number of objects that satisfy the query).

**The Multiversion B-tree (MVBT):** The MVBT approach transforms a timeslice query to a partial persistence problem. In particular, a data structure is called *persistent* [4] if an update creates a new version of the data structure while the previous version is still retained and can be accessed. Otherwise, if old versions of the structure are discarded, the structure is termed *ephemeral*. *Partial* persistence implies that only the newest version of the structure can be modified to create a new version.

The key observation is that partial persistence "suits" nicely transaction-time evolution since these changes are always applied on the latest state $s(t)$ of the evolving set (Figure 1). To support key range queries on a given $s(t)$, one could use an ordinary $B^+$-tree to index the objects in $s(t)$ (that is, the keys of the objects in $s(t)$ appear in the data pages of the $B^+$-tree). As $s(t)$ evolves over time through object changes, so does its corresponding $B^+$-tree. Storing copies of all the states that the $B^+$-tree took during the evolution of $s(t)$ is clearly inefficient. Instead, one should "see" the evolution of the $B^+$-tree as a partial persistence problem, i.e., as a set of updates that create subsequent versions of the $B^+$-tree.

Conceptually, the MVBT stores all the states assumed by the $B^+$-tree through its transaction-time evolution. Its structure is a directed acyclic graph of pages. This graph embeds many $B^+$-trees and has a number of root pages. Each root is responsible for providing access to a subsequent part of the $B^+$-tree's evolution. Data records in the MVBT leaf pages maintain the transaction-time evolution of the corresponding $B^+$-tree data records (that is, of the objects in $s(t)$). Each record is thus extended to include an interval [*insertion-time*, *deletion-time*), representing the transaction-times that the corresponding object was inserted/deleted from $s(t)$. During this interval the data-record is termed *alive*. Hence, the MVBT directly represents object deletions. Index records in the non-leaf pages of the MVBT maintain the

evolution of the corresponding index records of the $B^+$-tree and are also augmented with insertion-time and deletion-time fields.

Assume that each page in the MVBT has a capacity of holding $B$ records. A page is called *alive* if it has not been *time-split* (see below). With the exception of root pages, for all transaction-times $t$ that a page is alive, it must have at least $q$ records that are alive at $t$ ($q < B$). This requirement enables clustering of the alive objects at a given time in a small number of pages, which in turn will minimize the query I/O. Conceptually, a data page forms a rectangle in the time-key space; for any time in this rectangle the page should contain at least $q$ alive records. As a result, if the search algorithm accesses this page for any time during its rectangle, it is guaranteed to find at least $q$ alive records. That is, when a page is accessed, it contributes enough records for the query answer.

The first step of an update (insertion or deletion) at the transaction time $t$ locates the target leaf page in a way similar to the corresponding operations in an ordinary $B^+$-tree. Note that, only the latest state of the $B^+$-tree is traversed in this step. An update leads to a *structural change* if at least one new page is created. *Non-structural* are those updates which are handled within an existing page.

After locating the target leaf page, an insert operation at the current transaction time $t$ adds a data record with a transaction interval of [$t$, *UC*) to the target leaf page. This may trigger a structural change in the MVBT, if the target leaf page already has $B$ records. Similarly, a delete operation at transaction time $t$ finds the target data record and changes the record's interval to [insertion-time, $t$). This may trigger a structural change if the resulting page ends up having less than $q$ alive records at the current transaction time. The former structural change is called a *page overflow*, and the latter is a *weak version underflow* [1]. Page overflow and weak version underflow need special handling: a *time-split* is performed on the target leaf-page. The time-split on a page $x$ at time $t$ is performed by copying to a new page $y$ the records alive in page $x$ at $t$. Page $x$ is considered *dead* after time $t$. Then the resulting new page has to be incorporated in the structure [1].

Since updates can propagate to ancestors, a root page may become full and time-split. This creates a new root page, which in turn may be split at a later transaction time to create another root and so on. By construction, each root of the MVBT is alive for a subsequent, non-intersecting transaction-time interval. Efficient access to the root that was alive at time $t$ is possible by keeping an index on the roots, indexed by their time-split times. Since time-split times are in order, this root index is easily kept (this index is called the root* in [1]). In general, not many splits propagate to the top, so the number of root splits is small and the root* structure can be kept in main memory. If this is not the case, a small index can be created on top of the root* structure.

Answering a range-timeslice query on transaction time $t$ has two parts. First, using the root index, the root alive at $t$ is found. This part is conceptually equivalent to accessing $s(t)$ or, more explicitly, accessing the $B^+$-tree indexing the objects of $s(t)$. Second, the answer is found by searching this tree in a top-down fashion as in a $B^+$-tree. This search considers the record transaction interval. The transaction interval of every record returned or traversed should include the transaction time $t$, while its key attribute should satisfy the key query predicate. A range-timeslice query takes $O(\log_B n + s/B)$ I/O's while the space is linear to $n$ (i.e., $O(n/B)$). Hence, the MVBT optimally solves the range-timeslice query. The update processing is $O(\log_B m)$ per change, where $m$ is the size of $s(t)$ when the change took place. This is because a change that occurred at time $t$ traverses what is logically a $B^+$-tree on the $m$ elements of $s(t)$.

**The Overlapping B-tree:** Similar to the MVBT approach, the evolving set $s(t)$ is indexed by a $B^+$-tree. The intuition behind the Overlapping B-tree [10, 2] is that the $B^+$-trees of subsequent versions (states) of $s(t)$ will not differ much. A similar approach was taken in the EXODUS DBMS [3]. The Overlapping B-tree is thus a graph structure that superimposes many ordinary $B^+$-trees (Figure 2). An update at some time $t$ creates a new version of $s(t)$ and a new root in the structure. If a subtree does not change between subsequent versions, it will be shared by the new root. Sharing common subtrees among subsequent $B^+$-trees is done through index nodes of the new $B^+$-tree that point to nodes of previous $B^+$-tree(s). An update will create a new copy of the data page it refers to. This implies that a new path is also created leading to the new page; this path is indexed under the new root.

To address a range-timeslice query for a given transaction time *t*, the latest root that was created before or at *t* must be found. Hence, roots are time-stamped with the transaction time of their creation. These timestamps can be easily indexed on a separate $B^+$-tree. This is similar to the MVBT root* structure; however, the Overlapping B-tree creates a new root per version. After the appropriate root is found, the search continues traversing the tree under this root, as if an ordinary $B^+$-tree was present for state $s(t)$.

Updating the Overlapping B-tree involves traversing the structure from the current root version and locating the data page that needs to be updated. Then a copy of the page is created as well as a new path to this page. This implies $O(\log_B m)$ I/O's per update, where *m* is the current size of the evolving set. An advantage of the Overlapping structure is in the simplicity of its implementation. Note that except the roots, the other nodes do not involve any time-stamping. Such time-stamping is not needed because pages do not have to share records from various versions. Even if a single record changes in a page, the page cannot be shared by different versions; rather a new copy of the page is created. This, however, comes at the expense of the space performance. The Overlapping B-tree occupies $O(n \log_B n)$ pages since, in the worst case, every version creates an extra tree path. Further performance results on this access method can be found in [10].

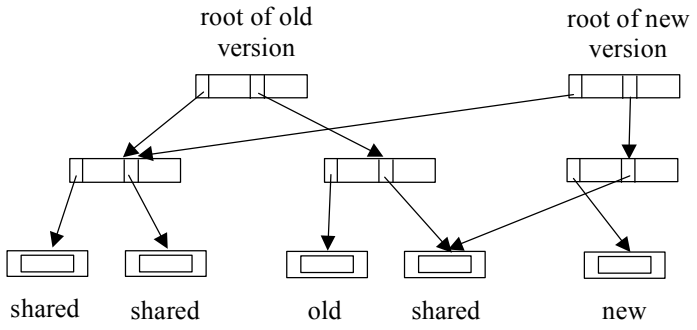



Figure 2. The Overlapping B-tree.

**KEY APPLICATIONS**

The characteristics of transaction-time make such databases ideal for applications that need to maintain their past; examples are: billing, accounting, tax-related etc.

**CROSS REFERENCES**

Temporal Databases, Transaction-time, Valid-time, Valid-Time Indexing, Bi-Temporal Indexing, B+-tree

**RECOMMENDED READING**

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer (1996). An Asymptotically Optimal Multiversion B-Tree. VLDB J. 5(4): 264-275.

[2] F. W. Burton, M.M. Huntbach, and J.G. Kollias (1985). Multiple Generation Text Files Using Overlapping Tree Structures. Comput. J. 28(4): 414-416.

[3] M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita: Object and File Management in the EXODUS Extensible Database System. VLDB 1986: 91-100.

[4] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan (1989). Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124.

[5] M.C. Easton: Key-Sequence Data Sets on Inedible Storage. IBM Journal of Research and Development 30(3): 230-241 (1986)

[6] D. Lomet and B. Salzberg (1989). Access Methods for Multiversion Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 315-324.

[7] B. Salzberg and V.J. Tsotras (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158-221.

[8] R.T. Snodgrass and I. Ahn (1986). Temporal Databases. IEEE Computer 19(9): 35-42.

[9] V.J. Tsotras and N. Kangelaris (1995). The Snapshot Index: An I/O-optimal access method for timeslice queries. Inf. Syst. 20(3): 237-260.

[10] T. Tzouramanis, Y. Manolopoulos, and N.A. Lorentzos (1999). Overlapping B+-Trees: An Implementation of a Transaction Time Access Method. Data Knowledge Engineering. 29(3): 381-404.

[11] P.J. Varman and R.M. Verma (1997). An Efficient Multiversion Access Structure. IEEE Transactions on Knowledge Data Engineering. 9(3): 391-409.

# TSQL2

Richard T. Snodgrass
University of Arizona

## SYNONYMS

none

## DEFINITION

TSQL2 (*Temporal Structured Query Language*) is a temporal extension of SQL-92 designed in 1993–4 by a committee comprised of Richard T. Snodgrass, Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada. The goal of this language design committee was to consolidate past research on temporal query languages, by the committee members as well as many others, by developing a specification for a consensus language that could form a common core for research.

## HISTORICAL BACKGROUND

Temporal databases have been an active research topic since 1980. By the early 1990's, several dozen temporal query languages had been proposed, and many temporal database researchers felt that the time had come to consolidate approaches to temporal data models and calculus-based query languages to achieve a consensus query language and associated data model upon which future research could be based.

In April 1992 Richard Snodgrass circulated a white paper that proposed that a temporal extension to SQL be produced by the research community. Shortly thereafter, the temporal database community organized the ARPA/NSF International Workshop on an Infrastructure for Temporal Databases, held in Arlington Texas in June 1993 [3]. Discussions at that workshop indicated that there was substantial interest in a temporal extension to the conventional relational query language SQL-92 [2]. A general invitation was sent to the community, and about a dozen people volunteered to develop a language specification. Several people later joined the committee. The goal of this language design committee was to develop a specification for a consensus extension to SQL-92, termed the *Temporal Structured Query Language*, or TSQL2.

The group corresponded via electronic mail from early July 1993, submitting, debating, and refining proposals for the various aspects and elements of the language. In September 1993, the first draft specification, accompanied by thirteen commentaries, was distributed to the committee. In December 1993, a much enlarged draft, accompanied by some twenty-four commentaries, was distributed to the committee. A preliminary language specification was made public in March 1994 [5], and a tutorial of the language appeared in September 1994 [6]. The final language specification and 28 commentaries were also made available via anonymous FTP in early October 2004. The final specification and commentaries appeared in a book [7] that was distributed at a temporal database workshop in summer of 1995, less than two years after the committee had been founded. TSQL2 is remarkable, and perhaps unique, in that it was designed entirely via electronic mail, by a committee that never met physically (in fact, no one on the committee has met every other committee member).

Work then commenced to incorporate elements and underlying insights of TSQL2 into SQL3. The first step was to propose a new part to SQL3, termed SQL/Temporal. This new part was accepted at the Ottawa meeting in January, 1995 as Part 7 of the SQL3 specification [8]. A modification of TSQL2's `PERIOD` data type is included in that part.

The focus at that point changed to adding valid-time and transaction-time support to SQL/Temporal. Two change proposals, one on valid-time support and one on transaction-time support, were unanimously accepted

by ANSI and forwarded to ISO [9, 10]; a summary appeared shortly thereafter [11]. A comprehensive set of case studies [4] showed that while SQL-92 required 1848 lines, only 520 lines of SQL/Temporal were required to achieve exactly the same functionality. These case studies showed that, over a wide range of data definition, query, and modification fragments, the SQL-92 version is *three* times longer in numbers of lines than the SQL/Temporal version, and many times more complex. In fact, very few SQL/Temporal statements were more than ten lines long; some statements in SQL-92 comprised literally dozens of lines of highly complex code. Due to disagreements within the ISO committee as to where temporal support in SQL should go, the project responsible for temporal support was canceled near the end of 2001. Nevertheless, concepts and constructs from SQL/Temporal have been implemented in the Oracle database management system, and other products have also included temporal support.
Oracle 9i includes support for transaction time. Its flashback queries allow an application to access prior transaction-time states of its database; they are transaction timeslice queries. Database modifications and conventional queries are temporally upward compatible. Oracle 10g extends flashback queries to retrieve all the versions of a row between two transaction times (a key-transaction-time-range query) and allows tables and databases to be rolled back to a previous transaction time, discarding all changes after that time. The Oracle 10g Workspace Manager includes the period data type, valid-time support, transaction-time support, support for bitemporal tables, and support for sequenced primary keys, sequenced uniqueness, sequenced referential integrity, and sequenced selection and projection, in a manner quite similar to that proposed in SQL/Temporal.

**SCIENTIFIC FUNDAMENTALS**
The goals that underpinned the process that led to the TSQL2 language design are first considered, then the language concepts underlying TSQL2 are reviewed.

## Design Goal for TSQL2

TSQL2 is a temporal query language, designed to query and manipulate time-varying data stored in a relational database. It is an upward-compatible extension of the international standard relational query language SQL-92. The TSQL2 language design committee started their work by establishing a number of ground rules with the objective of achieving a coherent design.

TSQL2 will be a *language* design.

- TSQL2 is to be a *relational* query language, not an object-oriented query language.
- TSQL2 should be consistent with existing standards, not another standard.
- TSQL2 should be comprehensive and should reflect areas of convergence.
- The language will have an associated algebra.

The committee enumerated the desired features of TSQL2; these guided the design of the language. The first batch concerned the data model itself.

TSQL2 should not distinguish between snapshot equivalent instances, i.e., snapshot equivalence and identity should be synonymous.

- TSQL2 should support only one valid-time dimension.
- For simplicity, tuple timestamping should be employed.
- TSQL2 should be based on homogeneous tuples.
- Valid-time support should include support for both the past and the future.
- Timestamp values should not be limited in range or precision.

The next concerned the language proper.

TSQL2 should be a consistent, fully upwardly compatible extension of SQL-92.

- TSQL2 should allow the restructuring of tables on any set of attributes.
- TSQL2 should allow for flexible temporal projection, but TSQL2 syntax should reveal clearly when non-standard temporal projections are being done.

- Operations in TSQL2 should not accord any explicit attributes special semantics. For example, operations should not rely on the notion of a key.
- Temporal support should be optional, on a per-table basis.

  Tables not specified as temporal should be considered as snapshot tables. It is important to be an extension of SQL-92's data model when possible, not a replacement. Hence, the schema definition language should allow the definition of snapshot tables. Similarly, it should be possible to derive a snapshot table from a temporal table.
- User-defined time support should include instants, periods, and intervals.
- Existing aggregates should have temporal analogues in TSQL2.
- Multiple calendar and multiple language support should be present in timestamp input and output, and timestamp operations. SQL-92 supports only one calendar, a particular variant of the Gregorian calendar, and one time format. The many uses of temporal databases demand much more flexibility.
- It should be possible to derive temporal and non-temporal tables from underlying temporal and non-temporal tables.

Finally, the committee agreed upon three features aimed at ease of implementation.

TSQL2 tables should be implementable in terms of conventional first normal form tables. In particular, the language should be implementable via a data model that employs period-timestamped tuples. This is the most straightforward representation, in terms of extending current relational technology.

- TSQL2 must have an efficiently implementable algebra that allows for optimization and that is an extension of the conventional relational algebra, on which current DBMS implementations are based. The temporal algebra used with the TSQL2 temporal data model should contain temporal operators that are extensions of the operations in the relational algebra. Snapshot reducibility is also highly desired, so that, for example, optimization strategies will continue to work in the new data model.
- The language data model should *accept* implementation using other models, such as models that timestamp attribute values. The language data model should allow multiple representational data models. In particular, it would be best if the data model accommodated the major temporal data models proposed to date, including attribute timestamped models.

## Language Concepts in TSQL2

The following is a brief outline of the major concepts behind TSQL2.

**Time Ontology**

The TSQL2 model of time is bounded on both ends. The model refrains from deciding whether time is ultimately continuous, dense, or discrete. Specifically, TSQL2 does not allow the user to ask a question that will differentiate the alternatives. Instead, the model accommodates all three alternatives by assuming that an instant on a time-line is much smaller than a chronon, which is the smallest entity that a timestamp can represent exactly (the size of a chronon is implementation-dependent). Thus, an instant can only be approximately represented. A discrete image of the represented times emerges at run-time as timestamps are scaled to user-specified (or default) granularities and as operations on those timestamps are performed to the given scale.

An instant is modeled by a timestamp coupled with an associated scale (e.g., day, year, month). A period is modeled by a pair of two instants in the same scale, with the constraint that the instant timestamp that starts the period equals or precedes (in the given scale) the instant timestamp that terminates the period.

**Base Line Clock**

A semantics must be given to each time that is stored in the database. SQL-92 specifies that times are given in UTC seconds, which are, however, not defined before 1958, and in any case cannot be used to date prehistoric time, as UTC is based in part on solar time. TSQL2 includes the concept of a *baseline clock*, which provides the semantics of timestamps. The baseline clock relates each second to physical phenomena and partitions the time line into a set of contiguous *periods*. Each period runs on a different clock. *Synchronization points* delimit period boundaries. The baseline clock and its representation are independent of any calendar.

**Data Types**

SQL-92's datetime and interval data types are augmented with a *period* datetime, of specifiable range and precision. The range and precision can be expressed as an integer (e.g., a precision of 3 fractional digits) or as an interval

(e.g., a precision of a week). Operators are available to compare timestamps and to compute new timestamps, with a user-specified precision. Temporal values can be input and output in user-specifiable formats, in a variety of natural languages. *Calendars* and *calendric systems* permit the application-dependent semantics of time to be incorporated.

A surrogate data is introduced in TSQL2. Surrogates are unique identifiers that can be compared for equality, but the values of which cannot be seen by the users. In this sense, a surrogate is "pure" identity and does not describe a property (i.e., it has no observable value). Surrogates are useful in tying together representations of multiple temporal states of the same object; they are not a replacement for keys.

**Time-lines**

Three time-lines are supported in TSQL2: user-defined time, valid time, and transaction time. Hence values from disparate time-lines can be compared, at an appropriate precision. Transaction-time is bounded by `initiation`, the time when the database was created, and `until changed`. In addition, user-defined and valid time have two special values, `beginning` and `forever`, which are the least and greatest values in the ordering. Transaction time has the special value `until changed`.

Valid and user-defined times can be indeterminate. In *temporal indeterminacy*, it is known that an event stored in a temporal database did in fact occur, but it is not known exactly *when* that event occurred. An instant (interval, period) can be specified as determinate or indeterminate; if the latter then the possible mass functions, as well as the generality of the indeterminacy to be represented, can be specified. The quality of the underlying data (termed its *credibility*) and the *plausibility* of the ordering predicates expressed in the query can be controlled on a per-query or global basis.

Finally, instant timestamps can be *now-relative*. A now-relative time of "*now* - 1 day", interpreted when the query was executed on June 12, 1993, would have the *bound* value of "June 11, 1993." The users can specify whether values to be stored in the database are to be bound (i.e., not now-relative) or unbound.

**Aggregates**

The conventional SQL-92 aggregates are extended to take into account change across time. They are extended to return time-varying values and to permit grouping via a partitioning of the underlying time line, termed *temporal grouping*. Values can be *weighted* by their duration during the computation of an aggregate. Finally, a new temporal aggregate, `RISING`, is added. A taxonomy of temporal aggregates [7, Chapter 21] identifies fourteen possible kinds of aggregates; there are instances of all of these kinds in TSQL2.

**Valid-time Tables**

The snapshot tables supported by SQL-92 continue to be available in TSQL2, which, in addition, supports *state* tables, where each tuple is timestamped with a *temporal element* that is a union of periods. As an example, the Employee table with attributes Name, Salary, and Manager could contain the tuple (Tony, 10000, LeeAnn). The temporal element timestamp would record the maximal (non-contiguous) periods in which Tony made $10000 and had LeeAnn as his manager. Information about other values of Tony's salary or other managers would be stored in other tuples. The timestamp is implicitly associated with each tuple; it is not another column in the table. The range, precision, and indeterminacy of a temporal element can be specified.

Temporal elements are closed under union, difference, and intersection. Timestamping tuples with temporal elements is conceptually appealing and can support multiple representational data models. Dependency theory can be extended to apply in full to this temporal data model.

TSQL2 also supports *event* tables, in which each tuple is timestamped with an *instant set*. As an example, a Hired table with attributes Name and Position could contain the tuple (LeeAnn, Manager). The instant set timestamp would record the instant(s) when LeeAnn was hired as a Manager. (Other information about her positions would be stored in separate tables.) As for state tables, the timestamps are associated implicitly with tuples.

**Transaction-time and Bitemporal Tables**

Orthogonally to valid time, transaction time can be associated with tables. The transaction time of a tuple, which is a temporal element, specifies when that tuple was considered to be part of the current database state. If the tuple (Tony, 10000, LeeAnn) was stored in the database on March 15, 1992 (say, with an `APPEND` statement) and removed from the database on June 1, 1992 (say, with a `DELETE` statement), then the transaction time of that tuple would be the period from March 15, 1992 to June 1, 1992.

Transaction timestamps have an implementation-dependent range and precision, and they are determinate.

In summary, there are six kinds of tables: snapshot (no temporal support beyond user-defined time), valid-time state tables (timestamped with valid-time elements), valid-time event tables (timestamped with valid-time

instant sets), transaction-time tables (timestamped with transaction-time elements), bitemporal state tables (timestamped with bitemporal elements), and bitemporal event tables (timestamped with bitemporal instant sets).

**Schema Specification**

The `CREATE TABLE` and `ALTER` statements allow specification of the valid- and transaction-time aspects of temporal tables. The scale and precision of the valid timestamps can also be specified and later altered.

**Restructuring**

The `FROM` clause in TSQL2 allows tables to be *restructured* so that the temporal elements associated with tuples with identical values on a subset of the columns are coalesced. For example, to determine when Tony made a Salary of $10000, independent of who his manager was, the Employee table could be restructured on the Name and Salary columns. The timestamp of this restructured tuple would specify the periods when Tony made $10000, information which might be gathered from several underlying tuples specifying different managers.

Similarly, to determine when Tony had LeeAnn as his manager, independent of his salary, the table would be restructured on the Name and Manager columns. To determine when Tony was an employee, independent of how much he made or who his manager was, the table could be restructured on only the Name column.

Restructuring can also involve *partitioning* of the temporal element or instant set into its constituent maximal periods or instants, respectively. Many queries refer to a continuous property, in which maximal periods are relevant.

**Temporal Selection**

The valid-time timestamp of a table may participate in predicates in the `WHERE` clause by via `VALID()` applied to the table (or correlation variable) name. The transaction-time of a table can be accessed via `TRANSACTION()`. The operators have been extended to take temporal elements and instant sets as arguments.

**Temporal Projection**

Conventional snapshot tables, as well as valid-time tables, can be derived from underlying snapshot or valid-time tables. An optional `VALID` or `VALID INTERSECT` clause is used to specify the timestamp of the derived tuple. The transaction time of an appended or modified tuple is supplied by the DBMS.

**Update**

The update statements have been extended in a manner similar to the `SELECT` statement, to specify the temporal extent of the update.

**Cursors**

Cursors have been extended to optionally return the valid time of the retrieved tuple.

**Schema Versioning**

Schema *evolution*, where the schema may change, is already supported in SQL-92. However, old schemas are discarded; the data is always consistent with the current schema. Transaction time support dictates that previous schemas be accessible, which calls for *schema versioning*. TSQL2 supports a minimal level of schema versioning.

**Vacuuming**

Updates, including (*logical*) deletions, to transaction time tables result in insertions at the physical level. Despite the continuing decrease in cost of data storage, it is still, for various reasons, not always acceptable that all data be retained forever. TSQL2 supports a simple form of *vacuuming*, i.e., physical deletion, from such tables.

**System Tables**

The `TABLE` base table has been extended to include information on the valid and transaction time components (if present) of a table. Two other base tables have been added to the definition schema.

**SQL-92 Compatibility**

All aspects of TSQL2 are pure extensions of SQL-92. The user-defined time in TSQL2 is a consistent replacement for that of SQL-92. This was done to permit support of multiple calendars and literal representations. Legacy applications can be supported through a default `SQL92_calendric_system`.

The defaults for the new clauses used to support temporal tables were designed to satisfy snapshot reducibility, thereby ensuring that these extensions constitute a strict superset of SQL-92.

**Implementation**

During the design of the language, considerable effort was expended to ensure that the language could be implemented with only moderate modification to a conventional, SQL-92-compliant DBMS. In particular, an algebra has been demonstrated that can be implemented in terms of a period-stamped (or instant-stamped, for event tables) tuple representational model; few extensions to the conventional algebra were required to fully

support the TSQL2 constructs. This algebra is snapshot reducible to the conventional relational algebra. Support for multiple calendars, multiple languages, mixed precision, and indeterminacy have been included in prototypes that demonstrated that these extensions have little deleterious effect on execution performance. Mappings from the data model underlying TSQL2, the bitemporal conceptual data model [1], to various representational data models have been defined [7].

## KEY APPLICATIONS*

TSQL2 continues to offer a common core for temporal database research, as well as a springboard for change proposals for extensions to the SQL standard.

## FUTURE DIRECTIONS

Given the dramatic decrease in code size and complexity for temporal applications that TSQL2 and SQL/Temporal offers, it is hoped that other DBMS vendors will take Oracle's lead and incorporate these proposed language constructs into their products.

## URL TO CODE*

http://www.cs.arizona.edu/people/rts/tsql2.html This web page includes links to the ISO documents.
http://www.sigmod.org/dblp/db/books/collections/snodgrass95.html

## CROSS REFERENCE*

Applicability Period, Fixed Span, Now in Temporal Databases, Period-Stamped Temporal Models, Schema Versioning, Temporal Aggregation, Temporal Algebras, Temporal Compatibility, Temporal Integrity Constraints, Temporal Joins, Temporal Logical Models, Temporal Query Languages, Temporal Vacuuming, Time-Line Clock, Transaction Time, TUC, Until Changed, Valid Time, Value Equivalence, Variable Span

## RECOMMENDED READING

**Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.**


[1] Christian S. Jensen, Michael D. Soo and Richard T. Snodgrass, "Unifying Temporal Data Models via a Conceptual Model," *Information Systems*, Vol. 19, No. 7, December 1994, pp. 513–547.

[2] Jim Melton and Alan R. Simon, **Understanding the New SQL: A Complete Guide**. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[3] Richard T. Snodgrass (editor), *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, June 14–16, 1993.

[4] Richard T. Snodgrass,, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, July 1999.

[5] Richard T. Snodgrass, Ilsoo Ahn, Gad Ariav, Don S. Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suri M. Sripada, "TSQL2 Language Specification," *SIGMOD Record*, Vol. 23, No. 1, March 1994, pp. 65–86.

[6] Richard T. Snodgrass,Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada, "A TSQL2 Tutorial," in *ACM SIGMOD Record*, Vol. 23, No. 3, September 1994, pp. 27–33.

[7] Richard T. Snodgrass, editor, **The TSQL2 Temporal Query Language**, Kluwer Academic Publishers, 1995, 674+xxiv pages. The TSQL2 Language Design Committee consisted of Richard Snodgrass (chair), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang

Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada.

[8] Richard T. Snodgrass, Krishna Kulkarni, Henry Kucera, and Nelson Mattos, "Proposal for a new SQL Part—Temporal," ISO/IEC JTC1/SC21 WG3 DBL RIO-75, X3H2-94-481, November 2, 1994, 26 pages.

[9] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen and Andreas Steiner, "Adding Valid Time to SQL/Temporal," change proposal, ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/ WG3 DBL MAD-146r2, November 1996, 77 pages.

[10] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen and Andreas Steiner, "Adding Transaction Time to SQL/Temporal," change proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/ WG3 DBL MAD-147r2, November 1996, 47 pages.

[11] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner, "Transitioning Temporal Support in TSQL2 to SQL3," in **Temporal Databases: Research and Practice**, Ophir Etzion, Sushil Jajodia, and Suryanarayana M. Sripada (eds.), Springer, pp. 150–194, 1998.

# USER-DEFINED TIME

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

none

**DEFINITION**

The concept of *user-defined time* denotes time-valued attributes of database items with which the data model and query language associate no special semantics. Such attributes may have as their domains all domains that reference time, e.g., date and time. The domains may be instant-, period-, and interval-valued. Example user-defined time attributes include "birth day," "hiring date," and "contract duration." Thus, user-defined time attributes are parallel to attributes that record, e.g., salary, using domain "money," and publication count, using domain "integer." User-defined time attributes contrast transaction-time and valid-time attributes, which carry special semantics.

**MAIN TEXT**

The valid time and transaction time attributes of a database item "are about" the other attributes of the database item. The valid time records when the information recorded by the attributes is true in the modeled reality, and the transaction time captures when the data item was part of the current database state. In contrast, user-defined time attributes are simply "other" attributes that may be used for the capture of information with which valid time can be associated.

Conventional database management systems generally support a time and/or date attribute domain. The SQL2 standard has explicit support for user-defined time in its `datetime` and `interval` types.

**CROSS REFERENCE***

Temporal Database, Transaction Time, Valid Time

**REFERENCES***

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

R. T. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Austin, TX, May 1985, pp. 236–246.

R. T. Snodgrass and I. Ahn, "Temporal Databases," *IEEE Computer*, Vol. 19, No. 9, September, 1986, pp. 35–42.

# VALID TIME

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Real-world time; Intrinsic time; Logical time; Data time

## DEFINITION

The *valid time* of a fact is the time when the fact is true in the modeled reality. Any subset of the time domain may be associated with a fact. Thus, valid timestamps may be sets of time instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.

## MAIN TEXT

While other temporal aspects have been proposed, such as "decision time" and "event time," closer analysis indicates that the decision and event time of a fact can be captured as either a valid time or a transaction time of some related fact.

For example, consider a valid-time `Faculty` table with attributes `Name` and `Position` (that is, either Assistant Professor, Associate Professor, or Professor) that captures the positions of faculty members. The valid time then captures when a faculty member held a specific position.

The "decision time" of a faculty member holding a specific position would be the time the decision was made to hire or promote the faculty member into that position. In fact, there are generally several decision times for a promotion: recommendation by departmental promotion and tenure committee, recommendation by department chair, recommendation by college promotion and tenure committee, recommendation by Dean, and finally decision by Provost. Each of these decisions may be modeled as a separate fact, with an (event) valid time that captures when that decision was made.

Oracle explicitly supports valid time in its Workspace Manager. The support includes sequenced primary keys, sequenced uniqueness, sequenced referential integrity, and sequenced selection and projection.

The term "valid time" is widely accepted; it is short and easily spelled and pronounced. Most importantly, it is intuitive.

Concerning the alternatives, the term "real-world time" derives from the common identification of the modeled reality (opposed to the reality of the model) as the real world. This term is less frequently used "Intrinsic time" is the opposite of extrinsic time. Choosing intrinsic time for valid time would require one to choose extrinsic time for transaction time. The terms are appropriate: The time when a fact is true is intrinsic to the fact; when it happened to be stored in a database is clearly an extrinsic property. However, "intrinsic" is rarely used. "Logical time" has been used for valid time in conjunction with "physical time" for transaction time. As the discussion of intrinsic time had to include extrinsic time, discussing logical time also requires the consideration of physical time. Both terms are more rarely used than valid and transaction time, and they do not posses clear advantages over these. The term "data time" is probably the most rarely used alternative. While it is clearly brief and easily spelled and pronounced, it is not intuitively clear that the data time of a fact refers to the valid time as defined above.

## CROSS REFERENCE*

Nonsequenced Semantics, Sequenced Semantics, Temporal Database, Time Instant, Time Interval, Time Period, Transaction Time, User-Defined Time, Valid-Time Indexing

## REFERENCES*

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

R. T. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Austin, TX, May 1985, pp. 236–246.

R. T. Snodgrass and I. Ahn, "Temporal Databases," *IEEE Computer*, Vol. 19, No. 9, September, 1986, pp. 35–42.

# VALID-TIME INDEXING

Mirella M. Moro
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
http://www.inf.ufrgs.br/~mirella/

Vassilis J. Tsotras
University of California, Riverside
Riverside, CA 92521, USA
http://www.cs.ucr.edu/~tsotras

## SYNONYMS
Valid-Time Access Methods

## DEFINITION
A valid-time index is a temporal index that enables fast access to valid-time datasets. In a traditional database, an index is used for selection queries. When accessing valid-time databases such selections also involve the valid-time dimension (the time when a fact becomes valid in reality). The characteristics of the valid-time dimension imply various properties that the temporal index should have in order to be efficient. As traditional indices, the performance of a temporal index is described by three costs: (i) storage cost (i.e., the number of pages the index occupies on the disk), (ii) update cost (the number of pages accessed to perform an update on the index; for example when adding, deleting or updating a record), and (iii) query cost (the number of pages accessed for the index to answer a query).

## HISTORICAL BACKGROUND
A valid-time database maintains the entire temporal behavior of an enterprise as best known now [12]. It stores the current knowledge about the enterprise's past, current or even future behavior. If errors are discovered about this temporal behavior, they are corrected by modifying the database. If the knowledge about the enterprise is updated, the new knowledge modifies the existing one. When a correction or an update is applied, previous values are not retained. It is thus not possible to view the database as it was before the correction/update. This is in contrast to a transaction-time database, which maintains the database activity (rather than the real world history) and can thus rollback to a past state. Hence in a valid-time database the past can change, while in a transaction-time database it cannot.

The problem of indexing valid-time databases can be reduced to indexing dynamic collections of intervals, where an interval represents the temporal validity of a record. Note that the term “interval” is used here to mean a “convex subset of the time domain" (and not a "directed duration"). This concept has also been named a “period”; in this discussion however, only the term “interval” is used.

To index a dynamic collection of intervals, one could use R-trees or related dynamic access methods. Relatively fewer approaches have been proposed for indexing valid-time databases. There have been various approaches proposed for the related problem of managing intervals in secondary storage. Given the problem difficulty, the majority of these approaches have focused on achieving good worst case behavior; as a result, they are mainly of theoretical importance. For a more complete discussion the reader is referred to a comprehensive survey [11].

## SCIENTIFIC FUNDAMENTALS
The following scenario exemplifies the distinct properties of the valid time dimension. Consider a dynamic collection of “interval-objects”. The term interval-object is used to emphasize that the object carries a valid-time interval to represent the temporal validity of some object property. (In contrast, and to emphasize that transaction-time represents the database activity rather than reality, note that intervals stored in a transaction time database correspond to when a record was inserted/updated in the database.)

The allowable changes in this environment are the addition/deletion/modification of an interval-object. A difference with the transaction-time abstraction (the reader is referred to the Transaction-Time Indexing entry for more details) is that the collection's evolution (past states) is *not* kept. An example of a dynamic collection of interval-objects appears in Figure 1. Assume that collection $C_a$ has been recorded in some erasable medium and a change happens, namely object $I_z$ is deleted. This change is applied on the recorded data and physically deletes object $I_z$. The medium now stores collection $C_b$, i.e., collection $C_a$ is

*not* retained. Note that when considering the valid time dimension, changes do not necessarily come in increasing time order (as is the case in transaction-time databases); rather they can affect *any* object in the collection. This implies that a valid-time database can correct errors in previously recorded data. However, only a single data state is kept, the one resulting after the correction is applied.

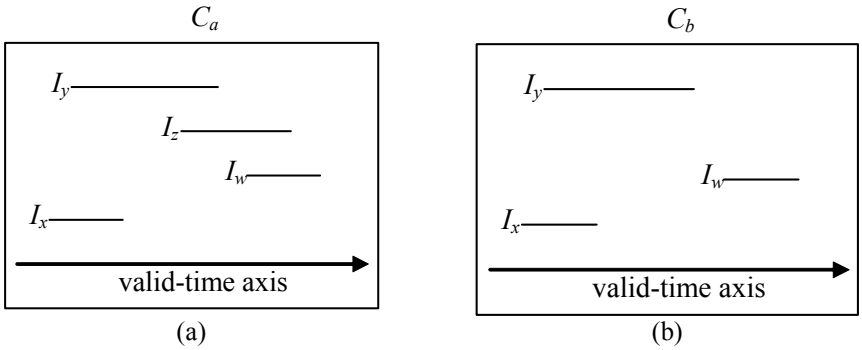



Figure 1. Two valid-time databases.

As a real-life example, consider the collection of contracts in a company. Each contract has an identity (*contract_no*) and an interval representing the contract's duration or validity. In collection $C_a$, there were four contracts in the company. But assume an error was discovered: contract $I_z$ was never a company contract (maybe it was mistakenly entered). Then, this information is permanently deleted from the collection, which now is collection $C_b$.

The notion of time is now related to the valid-time axis. Given a valid-time instant, interval-objects can be classified as past, future or current as related to this instant, if their valid-time interval is before, after or contains the given instant. Valid-time databases can be used to correct errors anywhere in the valid-time domain (past, current or future) because the record of any interval-object in the collection can be changed, independently of its position on the valid-time axis. Note that a valid-time database may store records with the same surrogate but with non-intersecting valid-time intervals. For example, another object with identity $I_x$ could be added in the collection at $C_b$ as long as its valid-time interval does not intersect with the valid-time interval of the existing $I_x$ object; the new collection will contain both $I_x$ objects, each representing object $I_x$ at different times in the valid-time dimension.

From the above discussion, an index used for a valid-time database should: (a) store the latest collection of interval-objects, (b) support addition/deletion/modification changes to this collection, and (c) efficiently query the interval-objects contained in the collection when the query is asked. Hence, a valid-time index should manage a *dynamic* collection of intervals.

Related is thus research on the interval management problem. Early work in main memory data-structures has proposed three separate approaches into managing intervals, namely, the Interval Tree, the Segment Tree and the Priority Search Tree [8]. Such approaches are focused on the “stabbing query” [6], i.e., given a collection of intervals, find all intervals that contain a given query point $q$. The stabbing query is one of the simpler queries for a valid-time database environment, since it does not involve any object key attribute (rather, only the object intervals).

Various methods have been proposed for making these main-memory structures disk-resident. Among them are: the External Memory Interval Tree [1], the Binary-Blocked Interval Tree [4], the External Segment Tree [2], the Segment Tree with Path Caching [10] and the External Memory Segment Tree [1], the Metablock Tree [6], the External Priority Search Tree [5], and the Priority Search Tree with Path Caching [10]. Many of these approaches are mainly of theoretical interest as they concentrate on achieving a worst-case optimal external solution to the 1-dimensional stabbing query. For good average behavior, methods based on multi-dimensional, indices should be preferred.

Since intervals are 2-dimensional objects, a dynamic multi-dimensional index like an R-tree may be used. Moreover, the R-tree can also index other attributes of the valid-time objects, thus enabling queries involving non-temporal attributes as well. For example, “find contracts that were active on time *t* and had contract-id in the range (*r1*,*r2*)”. While simple, the traditional R-tree approach may not always be very efficient. The R-tree will attempt to cluster intervals according to their length, thus creating pages with possibly large overlapping. It was observed in [7] that for data with non-uniform interval lengths (i.e., a large proportion of “short” intervals and a small proportion of “long” intervals), this overlapping is clearly increased, affecting the query and update performance of the index. This in turn decreases query and update efficiency.

Another straightforward approach is to transform intervals into 2-dimensional points and then use a Point Access Method (quad-tree, grid file, hB-tree, etc.). In Figure 2, interval $I = (x_1,y_1)$ corresponds to a single point in the 2-dimensional space. Since an interval’s end-time is always greater or equal than its start-time, all intervals are represented by points above the diagonal $x = y$. Note that an interval $(x,y)$ contains a query time $v$ if and only if its start-time $x$ is less than or equal to $v$ and its end-time $y$ is greater than or equal to $v$. Then an interval contains query $v$ if and only if its corresponding 2-dimensional point lies inside the box generated by lines $x = 0$, $x = v$, $y = v$, and $y = \infty$ (the shaded area in Figure 2). This approach avoids the overlapping mentioned above. Long intervals will tend to be stored together in pages with other long intervals (similarly, for the short intervals). However, no worst case guarantees for good clustering are possible.

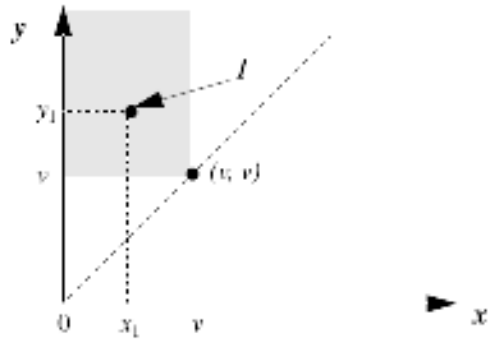



Figure 2. An interval $I = (x_1,y_1)$ corresponds to a point in a 2-dimensional space.

A related approach (MAP21) is proposed in [9]. An interval (*I, r*) is mapped to a point $z = (I \times 10^s) + r$, where *s* is the maximum number of digits needed to represent any point in the interval range. This is enough to map each interval to a separate point. A regular $B^+$-tree is then used to index these points. An advantage of this approach is that interval insertions/deletions are easy using the $B^+$-tree. To answer a stabbing query about *q*, the point closer but less than *q* is found among the points indexed in the $B^+$-tree, and then a sequential search for all intervals before *q* is performed. At worse, many intervals that do not intersect *q* can be found (this approach assumes that in practice the maximal interval length is known, which limits how far back the sequential search continues from *q*).

Another approach is to use a combination of an R-tree with Segment Tree properties. The Segment R-tree (SR-tree) was proposed in [7]. The SR-tree is an R-tree where intervals can be stored in both leaf and non-leaf nodes. An interval *I* is placed to the highest level node *X* of the tree such that *I* spans at least one of the intervals represented by *X*’s child nodes. If *I* does not span *X*, it spans at least one of its children but is not fully contained in *X*, then *I* is fragmented. Figure 3 shows an example of the SR-tree approach. The top rectangle depicts the R-tree nodes (root, A, B, C, D and E) as well as the stored intervals. Interval *L* spans the rectangle of node C, but is not contained in node A. It is thus fragmented between nodes A and E.

Using this idea, long intervals will be placed in higher levels of the R-tree. Hence, the SR-tree tends to decrease the overlapping in leaf nodes (in the regular R-tree, a long interval stored in a leaf node will

“elongate” the area of this node thus exacerbating the overlap problem). However, having large numbers of spanning records or fragments of spanning records stored high up in the tree decreases the fan-out of the index as there is less room for pointers to children. It is suggested to vary the size of the nodes in the tree, making higher-up nodes larger. “Varying the size” of a node means that several pages are used for one node. This adds some page accesses to the search cost.

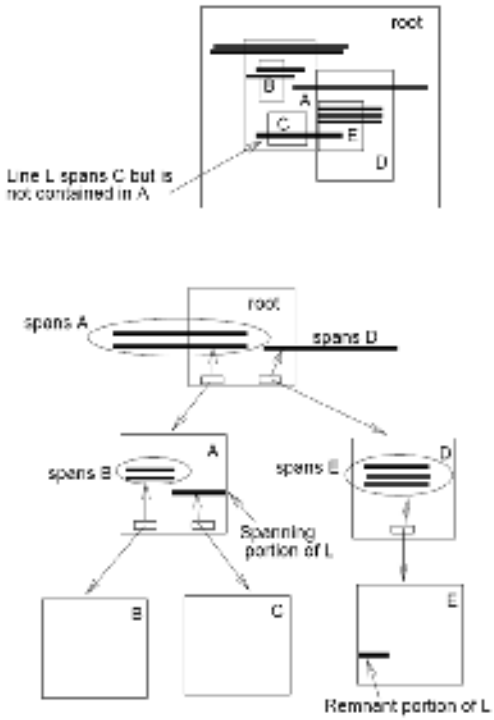


Figure 3. An example of the SR-tree approach.

As with the R-tree, if the interval is inserted at a leaf (because it did not span anything) the boundaries of the space covered by the leaf node in which it is placed may be expanded. Expansions may be needed on all nodes on the path to the leaf, which contains the new record. This may change the spanning relationships since existing intervals may no longer span children, which have been expanded. In this case, such intervals are reinserted in the tree, possibly being demoted to occupants of nodes they previously spanned. Splitting nodes may also cause changes in spanning relationships as they make children smaller - former occupants of a node may be promoted to spanning records in the parent.

In contrast to the traditional R-tree, the space used by the SR-tree is no longer linear. An interval may be stored in more than one non-leaf nodes (in the *spanning* and *remnant* portions of this interval). Due to the use of the segment-tree property, the space can be as much as $O(n \log_B n)$. Inserting an interval still takes logarithmic time. However, due to possible promotions, demotions and fragmentation, insertion is slower than in the R-tree. Even though the segment property tends to reduce the overlapping problem, the (pathological) worst case performance for the deletion and query time remains the same as for the R-tree organization (that is, at worst, the whole R-tree may have to searched for an interval). The average case behavior is however logarithmic. Deletion is a bit more complex, as all the remnants of the deleted interval have to be deleted too. The original SR-tree proposal thus assumed that deletions of intervals are not that frequent.

The SR-tree search algorithm is similar to that of the original R-tree. It descends the index depth-first, descending only those branches that contain the given query point *q*. In addition, at each node encountered during the search, all spanning intervals stored at the node are added to the answer. To improve the performance of the structure, the *Skeleton SR-tree* has also been proposed [7], which is an SR-tree that pre-partitions the entire domain into some number of regions. This pre-partition is based on

some initial assumption on the distribution of data and the number of intervals to be inserted. Then the Skeleton SR-tree is populated with data; if the data distribution is changed, the structure of the Skeleton SR-tree can be changed too.

When indexing valid-time intervals, overlapping may also incur if the valid-time intervals extend to the ever-increasing *now*. One approach could be to use the largest possible valid-time to represent the variable *now*. In [3] the problem of addressing both the *now* and transaction-time Until-Changed (*UC)* variables is addressed by using bounding rectangles/regions that increase as the time proceeds. A variation of the R-tree, the GR-tree is presented. More details appear in [3].

## KEY APPLICATIONS

The importance of temporal indexing emanates from the many applications that maintain temporal data. The ever increasing nature of time imposes the need for many applications to store large amounts of temporal data. Specialized indexing techniques are needed to access such data. Temporal indexing has offered many such solutions that enable fast access.

## CROSS REFERENCES

Temporal Database, Transaction-time, Valid-time, Transaction-time Indexing, Bi-temporal Indexing, B+-tree, R-tree

## RECOMMENDED READING


[1] L. Arge and J. S. Vitter (1996). Optimal Dynamic Interval Management in External Memory. In Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, pages 560-569.

[2] G. Blankenagel and R.H. Gueting (1994). External Segment Trees. Algorithmica, 12(6):498-532.

[3] R. Bliujute, C.S. Jensen, S. Saltenis and G. Slivinskas (1998). R-Tree Based Indexing of Now-Relative Bitemporal Data. VLDB Conference, pp: 345-356.

[4] Y.-J. Chiang and C.T. Silva. (1999). External Memory Techniques for Isosurface Extraction in Scientific Visualization, External Memory Algorithms and Visualization, J. Abello and J.S. Vitter (Eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, vol. 50, pages 247-277.

[5] C. Icking, R. Klein, and T. Ottmann (1988). Priority Search Trees in Secondary Memory. In *Graph Theoretic Concepts in Computer Science*, pages 84-93. Springer Verlag LNCS 314.

[6] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. S. Vitter (1993). Indexing for Data Models with Constraint and Classes. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems,* pages 233-243.

[7] C. Kolovson and M. Stonebraker (1991). Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 138-147.

[8] K. Mehlhorn (1984). *Data Structures and Efficient Algorithms*, Vol.3: Multi-dimensional Searching and Computational Geometry. Springer Verlag, EATCS Monographs.

[9] M. A. Nascimento and M. H. Dunham (1999). Indexing Valid Time Databases via $B^+$-Trees. IEEE Transaction on Knowledge Data Engineering, 11(6): 929-947.

[10] S. Ramaswamy and S. Subramanian (1994). Path Caching: a Technique for Optimal External Searching. In *Proceedings of the 13rd ACM Symposium on Principles of Database Systems,* pages 25-35.

[11] B. Salzberg and V. J. Tsotras (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158-221.

[12] R.T. Snodgrass and I. Ahn (1986). Temporal Databases. IEEE Computer 19(9):35-42.

**TITLE: Value Equivalence**

Nikos A. Lorentzos
Informatics Laboratory
Science Department
Agricultural University of Athens
Iera Odos 75, 11855 Athens, Greece
http://www.aua.gr/~lorentzos
lorentzos@aua.gr

**SYNONYMS**
none

**DEFINITION**
In temporal databases the scheme of a temporal relation S has the form S(**E, I**) where **E** and **I** represent two disjoint sets of attributes, termed by some authors *explicit* and *implicit*, respectively. *Explicit* are the attributes in which ordinary data is recorded, such as Employee_Id, Name, Salary etc. *Implicit* are the attributes in which either valid time (one or more) or transaction time is recorded. Given two tuples ($\mathbf{e}_1$, $\mathbf{i}_1$) and ($\mathbf{e}_2$, $\mathbf{i}_2$) with the scheme of S, it is said that there is a *value equivalence* between them if and only if $\mathbf{e}_1 = \mathbf{e}_2$. Equivalently, it is said that ($\mathbf{e}_1$, $\mathbf{i}_1$) and ($\mathbf{e}_2$, $\mathbf{i}_2$) are *value equivalent*.

*Example*: Consider one tuple, (Alex, 100, d400, d799), with scheme SALARY(Name, Amount, ValidTime), indicating that Alex's salary was 100 on each of the dates d400, d401, …, d799. Let also another tuple of the same scheme be (Alex, 100, d500, d899). For the given scheme, **E** = {Name, Amount} and **I** = {ValidTime}. Given that both tuples have the same value for Name and Amount, they are value equivalent.

**MAIN TEXT**
The term has been introduced in the context of temporal databases, mainly in order to ease discussion on issues such as temporal coalescing.

**CROSS REFERENCES**
Temporal Coalescing, Period stamped Temporal Models

# VARIABLE TIME SPAN

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

**SYNONYMS**

Moving Span

**DEFINITION**

A span is *variable* if its duration is dependent on the assumed context.

**MAIN TEXT**

Given a specific setting, any span is either a fixed span or a variable span. The possibly most obvious example of a variable span is "one month," in the Gregorian calendar. Its duration may be any of 28, 29, 30, and 31 days, depending on which particular month is intended. The span "one hour" is fixed because it always has a duration of 60 minutes.

**CROSS REFERENCE***

Fixed Span, Temporal Database, Time Span

**REFERENCES***

C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass and X. S. Wang, "A Glossary of Time Granularity Concepts," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer, pp. 406–413, 1998.

C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.

# WEAK EQUIVALENCE

Christian S. Jensen and Richard T. Snodgrass
Aalborg University, Denmark and University of Arizona, USA

## SYNONYMS

Snapshot equivalence; Temporally Weak

## DEFINITION

Informally, two tuples are *snapshot equivalent* or *weakly equivalent* if all pairs of timeslices with the same time instant parameter of the tuples are identical.

Let temporal relation schema $R$ have $n$ time dimensions, $D_i$, $i = 1, \ldots, n$, and let $\tau^i$, $i = 1, \ldots, n$ be corresponding timeslice operators, e.g., the valid timeslice and transaction timeslice operators. Then, formally, tuples $x$ and $y$ are weakly equivalent if

$$\forall t_1 \in D_1 \ldots \forall t_n \in D_n(\tau^n_{t_n}(\ldots(\tau^1_{t_1}(x))\ldots) = \tau^n_{t_n}(\ldots(\tau^1_{t_1}(y))\ldots))$$

Similarly, two relations are snapshot equivalent or weakly equivalent if at every instant their snapshots are equal. Snapshot equivalence, or weak equivalence, is a binary relation that can be applied to tuples and to relations.

## MAIN TEXT

The notion of weak equivalence captures the information content of a temporal relation in a point-based sense, where the actual timestamps used are not important as long as the same timeslices result. For example, consider the two relations with just a single attribute: {(a, [3, 9]} and {(a, [3, 5]), (a, [6, 9])}. These relations are different, but weakly equivalent.

Both "snapshot equivalent" and "weakly equivalent" are being used in the temporal database community. "Weak equivalence" was originally introduced by Aho et al. in 1979 to relate two algebraic expressions [1,2]. This concept has subsequently been covered in several textbooks. One must rely on the context to disambiguate this usage from the usage specific to temporal databases. The synonym "temporally weak" does not seem intuitive—in what sense are tuples or relations weak?

## CROSS REFERENCE*

Snapshot Equivalence, Temporal Database, Time Instant, Timeslice Operator, Transaction Time, Valid Time

## REFERENCES*

[1] Alfred V. Aho, Yehoshua Sagiv, Jeffrey D. Ullman: Efficient Optimization of a Class of Relational Expressions. ACM Trans. Database Syst. 4(4): 435-454 (1979)

[2] Alfred V. Aho, Yehoshua Sagiv, Jeffrey D. Ullman: Equivalences Among Relational Expressions. SIAM J. Comput. 8(2): 218-246 (1979)

[3] S. K. Gadia, "Weak Temporal Relations," in *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, MA, 1985, pp. 70-77.

[4] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), LNCS 1399, Springer-Verlag, pp. 367–405, 1998.