# The T4SQL Temporal Query Language*

Carlo Combi
Dipartimento di Informatica
Università degli Studi di Verona
I-37134 Verona, Italy
carlo.combi@univr.it

Angelo Montanari
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
I-33100 Udine, Italy
montana@dimi.uniud.it

Giuseppe Pozzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
I-20133 Milano, Italy
giuseppe.pozzi@polimi.it

## Categories and Subject Descriptors

H.2.3 [**Database management**]: Languages—*Query languages*; H.2.1 [**Database management**]: Logical Design—*Data models*

## General Terms

Languages

## Keywords

Temporal Query Language, SQL

## ABSTRACT

Time characterizes every aspect of our life and its management when storing and querying data is very important. In this paper we propose a new temporal query language, called T4SQL, supporting multiple temporal dimensions of data. Besides the well-known valid and transaction times, it encompasses two additional temporal dimensions, namely, availability and event times. The availability time records when information is known and treated as true by the information system; the event times record the occurrence times of both the event that starts the valid time and the event that ends it. T4SQL is capable to deal with different temporal semantics (atemporal aka non-sequenced, current, sequenced, next) with respect to every temporal dimension. Moreover, T4SQL provides a novel temporal grouping clause and an orthogonal management of temporal properties when defining the selection condition(s) and the schema for the output relation.

*This work was partially supported by contributions from the Italian Ministry of University and Research (MIUR), under the programs COFIN-PRIN "Intelligent analysis of hemodialysis data for the improvement of care processes" and "Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems", the EC funded project COOPER, the University of Udine, the University of Verona, and the Politecnico of Milano.


*CIKM'07,* November 6–8, 2007, Lisboa, Portugal.


## 1. INTRODUCTION

Temporal database management systems (TDBMS) can manage one or more temporal dimensions, beyond temporal data explicitly defined by the user [10]. Several studies have been done in the temporal database community regarding both the proposal of suitable extensions to data models and query languages and the management of temporal issues by standard (atemporal) query languages [2, 10, 11, 12, 15]. As a result, a temporal extension to SQL was officially proposed in 1995 as part of the SQL3 draft standard [4] (SQL3 is the most recent ISO standard version of the SQL language that replaced the SQL92 standard). Since priorities in extending SQL have been assigned to other topics, this part has been recently withdrawn [8]. However, research efforts on temporal databases have definitely highlighted that temporal information needs to be properly managed by SQL, as it is present in a large number of application domains [12]. At the same time, any temporal extension to SQL should preserve the compatibility with existing atemporal queries [12, 13].

Two temporal dimensions of data are commonly recognized as fundamental in temporal database applications [5]: the *valid time*, that records when a given information is valid in the application domain, and the *transaction time*, that records when the information has been entered, modified, or removed from the database. Two additional temporal dimensions have been recently introduced [3]: the *availability time*, that records when the information is known and treated as true by the information system, and the *event time*, that records the occurrence times of both the event that starts the validity time and the event that ends it.

In this paper, we propose a new temporal query language, named T4SQL, that extends, and is fully compatible with, SQL. T4SQL encompasses all the above-mentioned temporal dimensions, namely, *valid* time (VT), *transaction* time (TT), *availability* time, (AT) and *event* time (ET). Moreover, it deals with different temporal semantics (*atemporal* aka nonsequenced, *current*, *sequenced*, *next*) with respect to every temporal dimension.

The paper is organized as follows. Section 2 discusses related work, focusing on TSQL2 and SQL3, two of the most significant temporal query languages proposed in the literature. Section 3 briefly illustrates the temporal dimensions of valid, transaction, availability, and event times, and the adopted temporal data model. Section 4 describes in detail the query language T4SQL, in particular, clauses, tokens, and keywords. Section 5 provides an assessment of the work.

## 2. RELATED WORK

At the beginning of the 90's, the temporal database community made considerable efforts towards the development of a consensus temporal relational model and a corresponding query language. These efforts resulted in the proposal of the TSQL2 model and language [11]. TSQL2 [11] is a query language for temporal relational databases, derived from SQL, that supports both VT and TT. It minimizes the syntactical changes from SQL, it covers most features of existing temporal query languages, and it introduces some original elements.

Temporal aspects have also been considered in the development of the most recent ISO standard version of SQL. A new component, called SQL/Temporal, was formally approved in July 1995 as part of the SQL3 draft standard [4] and then withdrawn [8]. As a matter of fact, most commercial relational database systems provide some extensions to the relational model to represent times and dates through suitable data types and to the SQL query language to allow manipulation of stored time values, but these additions are not yet completely standardized across vendors.

The SQL3 SQL/Temporal component temporally extends the basic relational model, e.g., by adding the data type `PERIOD` and the two temporal dimensions VT and TT, and it supports temporal queries [13]. SQL3 is fully compatible with the previous versions of SQL, so that migration from these atemporal versions is simple and efficient, and existing code can be reused without any additional intervention.

Temporal queries can be specified according to two different semantics, namely, sequenced and non-sequenced semantics. According to the *sequenced semantics*, an SQL statement is executed over a given temporal dimension instant by instant, that is, it takes into consideration one time instant (a state of a relation) at a time. As an example, a primary key constraint with a sequenced semantics checks that, at any time instant, there are no tuples with null values on the key attributes and there are no pairs of tuples with the same values on the key attributes. The *sequenced semantics* allows one to reuse existing SQL code by performing a query at any time instant of the whole temporal axis. The *non-sequenced semantics* is more complex than the sequenced one. According to it, an SQL statement accesses a temporal relation as a whole. While sequenced semantics provides suitable tools that facilitate the access to data, non-sequenced one manipulate all pieces of temporal information in the database in a single step. In the general case, the outcome of a non-sequenced query is an atemporal relation. However, the user can possibly define the temporal dimension(s) of the resulting relation, thus obtaining a temporal relation. The non-sequenced semantics for the VT and TT temporal dimensions is imposed by the token `NONSEQUENCED VALIDTIME` and `NONSEQUENCED TRANSACTIONTIME`, respectively.

As for compatibility requirements, we must distinguish between upward compatibility and temporal upward compatibility. *Upward compatibility* constrains any new SQL standard to be syntactically and semantically compatible with the current one [13]. According to this requirement, SQL3 manages atemporal relations, both syntactically and semantically, as SQL does[1]. *Temporal upward compatibility* constrains any code available for an atemporal relation to produce the same result as if executed over its temporal extension. SQL3 allows one to add one or two temporal dimensions to an atemporal relation. In such a case, any query that originally was running over an atemporal relation, now runs over a temporal one. However, to keep its semantics (and syntax) unchanged, only current information is taken into consideration.

In the last years, research on temporal query languages focused on specific issues as the efficient implementation of a valid-time extension of SQL by an object-relational database system [2], the design and implementation of a prototyping tool supporting SQL temporal language extensions [7], the definition of temporal aggregates and temporal universal quantifiers in standard SQL [15], and the formalization of an SQL extension for the management of spatio-temporal data [14].

## 3. A DATA MODEL WITH MULTIPLE TEMPORAL DIMENSIONS

In the following, we briefly analyze the temporal dimensions of valid, transaction, availability, and event times[2], and we describe the distinctive features of the T4SQL data model supporting them.

### 3.1 Temporal Dimensions of Data

Valid time and transaction time are widely recognized as the two basic temporal dimensions of data. They can be defined as follows [5]:

DEFINITION 3.1. *The* valid time *(VT) of a fact is the time when the fact is true in the modelled reality.*

DEFINITION 3.2. *The* transaction time *(TT) of a fact is the time when the fact is current in the database and may be retrieved.*

VT and TT are orthogonal dimensions, which are independently recorded and satisfy specific properties. VT is usually provided by database users, while TT is system-generated. They can be related to each other in different ways. Jensen and Snodgrass propose sixteen different *specializations* of (bi)temporal relations, based one the relationships between the VT and TT of timestamped facts [9]. As an example, they classify a relation as retroactive if for every tuple of it the starting point $VT_s$ of VT precedes the starting point $TT_s$ of TT, that is, $VT_s \leq TT_s$.

Even though VT and TT suffice for many database applications, they turned out to be inadequate to cope with the temporal requirements of complex organizations such as hospitals and public institutions. In these contexts, one often needs to model both the time at which someone/the information system becomes aware of a fact (availability time) and the time at which the fact is stored into the database. While the latter is captured by TT, the *availability time* AT has been explicitly added in [3].

[1]It must be observed that upward compatibility can be seen as a particular case for the non-sequenced semantics, which takes into consideration only information associated with the current state.

[2]These temporal dimensions have been recently analyzed by Grandi et al. in the context of XML data for the management of evolving normative documents [6]. In addition, the authors discussed the possible coexistence of two different valid time dimensions.

DEFINITION 3.3. *The* availability time *(AT) of a fact is the time when the fact is known and believed correct by the information system.*

In many application domains, e.g., the medical one, decisions are taken on the basis of the available information, no matter whether it is stored in the database or not. AT captures this temporal dimension. Since there can be facts which are erroneously considered true by the information system, AT must be an interval: the starting point of AT is the time at which the fact becomes available to the information system, while its ending point is the time at which the information system realizes that the fact is not correct. As for TT, an ending point equal to *uc* (until changed) means that the fact is currently classified as correct.

In [1], Chakravarthy and Kim introduce a fourth temporal dimension, called *event time*, to distinguish between retroactive and delayed updates. In [3], Combi and Montanari refine it by showing that two event times are needed to suitably model relevant phenomena. The choice of adding ET as a separate temporal dimension has been extensively debated in the literature [10] and is discussed in detail in [3].

DEFINITION 3.4. *The* event time *(ET) of a fact is the occurrence time of a real-world event that either initiates or terminates the validity interval of the fact.*

Given a set of relevant facts, the event times are the occurrence times of the events, e.g., decisions or actions, that respectively initiate and terminate them. For any given fact, the occurrence time of the terminating event ($\mathsf{ET}_t$) does not precede that of the initiating event ($\mathsf{ET}_i$), that is, $\mathsf{ET}_i \leq \mathsf{ET}_t$.

## 3.2 The T4SQL Data Model

The T4SQL data model is a straightforward extension of the relational model, where every relation is equipped with the four temporal dimensions. VT, TT, and AT are represented as intervals, while ET is represented by a pair of attributes, that respectively record the occurrence time of the initiating event and of the terminating one. We assign the special value *now* to the starting/ending instant of VT to denote the current time. Moreover, we assign the special value *uc* to the ending instants of TT and AT to indicate that the corresponding tuple is still current in the database and believed correct by the information system, respectively.

The T4SQL temporal data model encompasses a single type of key constraints, namely, the *snapshot key* constraint (*key* for short). Given a temporal relation $R$, defined on a set of (atemporal) attributes $X$ and the special attributes VT, TT, AT, $\mathsf{ET}_i$, and $\mathsf{ET}_t$, and a set $K \subseteq X$, we say that $K$ is a *snapshot key* for $R$ if the following conditions hold:

- $\forall a \in K(t[a] \neq \texttt{null})$
- $\forall t1, t2\ ((t1[VT, TT, AT] \cap t2[VT, TT, AT] \neq \emptyset \wedge t1 \neq t2) \Rightarrow t1[K] \neq t2[K])$

where $t[VT, TT, AT]$ denotes the temporal region (a cube) associated with the tuple $t$. Observe that ET does not contribute to the definition of the snapshot key. $\mathsf{ET}_i$ and $\mathsf{ET}_t$ indeed denote two time instants (possibly related to different events), which do not identify any meaningful interval.

Other types of temporal key can be added, dealing with more complex temporal constraints in temporal relations; however, their analysis is out of the scope of this paper. On the contrary, the model imposes some basic constraints on the relationships between the values of the various temporal dimensions: we cannot assign a value to ET if there exists no value for the corresponding VT and we cannot assign a value to AT if there exists no value for the corresponding TT.

In the following, we shall use the two relations of Table 1 as a running example. The relation `PatTherapy` stores information about patients (attribute `PatId`) and prescribed therapies (attributes `Therapy` and `Dosage`), while the relation `PatSymptom` stores information about patients and detected symptoms (attributes `Symptom` and `SevLevel`). Both relations feature the four temporal dimensions VT, $\mathsf{ET}_i$, $\mathsf{ET}_t$, AT, TT. (`Patid`, `Therapy`) and (`PatId`, `Symptom`) are snapshot keys for the two relations, respectively.

# 4. THE QUERY LANGUAGE T4SQL

In this section, we describe the main features of T4SQL. First of all, it supports different semantics: *current*, which considers only current tuples; *sequenced*, which corresponds the homonymous SQL3 semantics; *atemporal*, which is equivalent to the SQL3 non-sequenced one; and *next*, which allows one to link consecutive states when evaluating a query. As a matter of fact, *current*, *sequenced*, and *next* can all be viewed as special cases of the *atemporal* semantics. However, they allow one to express meaningful classes of queries in a much more compact way.

T4SQL queries receive relations with the four temporal dimensions as input, via the `FROM` clause of a statement, and they return relations with at most the four temporal dimensions. Relations which do not feature all the four temporal dimensions are preliminarily converted to *complete* relations according to the following rules (alternative rules can be defined to cope with specific application domains)[3]:

- If the relation has no (possibly implicitly defined) VT, the associated VT is the current timestamp, that is, the valid time is $[now, now)$.
- If the relation has no (possibly implicitly defined) TT, we assume that information has been entered when the relation has been created and will last until it will be explicitly changed or deleted. Accordingly, TT is $[c, uc)$, where $c$ is the creation timestamp for the relation.
- Let $R$ be a relation provided with VT and devoid of ET. For every tuple of the corresponding complete relation $R'$, the following properties hold:

  $\mathsf{ET}_i = VT_s$ (the initiating event time is equal to the starting instant $VT_s$ of the valid time);

  $\mathsf{ET}_t = VT_e$ (the terminating event time is equal to the ending instant $VT_e$ of the valid time).

  Relations $R$ and $R'$ are called *co-active*.
- Let $R$ be a relation provided with TT and devoid of AT. For every tuple of the corresponding relation $R'$, the following properties hold:

[3]A temporal relation is complete if it has all the four temporal dimensions VT, TT, AT, and ET (i.e., $\mathsf{ET}_i$ and $\mathsf{ET}_t$).

**PatTherapy**

| PatId | Therapy | Dosage | VT | $ET_i$ | $ET_t$ | AT | TT |
|---|---|---|---|---|---|---|---|
| $p2$ | Paracetamol | dose1 | [2006-07-01, 2006-07-12) | 2006-06-28 | 2006-06-28 | [2006-06-28, 2006-09-06) | [2006-06-29, 2006-09-08) |
| $p1$ | Streptomycin | dose3 | [2006-05-08, 2006-11-15) | 2006-05-05 | 2006-11-12 | [2006-11-13, $uc$) | [2006-12-01, $uc$) |
| $p2$ | Paracetamol | dose2 | [2006-07-01, 2006-07-10) | 2006-06-28 | 2006-07-10 | [2006-09-07, $uc$) | [2006-09-08, $uc$) |

**PatSymptom**

| PatId | Symptom | SevLevel | VT | $ET_i$ | $ET_t$ | AT | TT |
|---|---|---|---|---|---|---|---|
| $p2$ | Fever | 1 | [2006-06-25, $now$) | 2006-06-23 | null | [2006-06-27, 2006-07-11) | [2006-06-28, 2006-07-11) |
| $p2$ | Fever | 2 | [2006-06-25, 2006-07-11) | 2006-06-23 | 2006-06-28 | [2006-07-12, $uc$) | [2006-07-15, $uc$) |
| $p1$ | Fever and dry cough | 4 | [2006-05-01, 2006-10-21) | 2006-03-23 | 2006-05-05 | [2006-11-13, $uc$) | [2006-11-15, $uc$) |

**Table 1: database instance for patient symptoms and related therapies.**

$AT_s = TT_s$ (the starting instant of the availability time is equal to the starting instant of the transaction time);

$AT_e = TT_e$ (the ending instant of the availability time is equal to the ending instant of the transaction time).

T4SQL borrows constants and standard temporal data types from SQL92 and the `PERIOD` data type from SQL3. Given a tuple of a relation `R`, the values it takes over the temporal dimensions can be recovered by means of the following functions:

`VALID(R)` returns the VT of the `R` tuple;
`TRANSACTION(R)` returns the TT of the `R` tuple;
`AVAILABLE(R)` returns the AT of the `R` tuple;
`INITIATING_ET(R)` returns the $ET_i$ of the `R` tuple;
`TERMINATING_ET(R)` returns the $ET_t$ of the `R` tuple.

The syntax of T4SQL extends that of SQL. Its (incomplete) BNF is as follows:

```
SEMANTICS <sem> [ON] <dim> [[TIMESLICE] <ts_exp>]
       {, <sem> [ON] <dim> [[TIMESLICE] <ts_exp>]}
SELECT  <sel_element_list>
  [WITH <w_exp> [AS] <dim> {, <w_exp> [AS] <dim>}]
  [TGROUPING(<temp_attribute> [AS] <new_name>
          {, <temp_attribute> [AS] <new_name>})]
 FROM <tables>
 WHERE <cond>
 WHEN  <t_cond>
 GROUP BY <list_group_element>
 HAVING <g_cond>

 <sem>:= ATEMPORAL | CURRENT | SEQUENCED |
         NEXT(<number)
 <dim>:= VALID | TRANSACTION | AVAILABILITY |
         INITIATING_ET | TERMINATING_ET
 <list_group_element> ::= <group_element>
        {, <group_element>}
 <group_element> ::= <attribute> |
        <temp_attribute> USING <part_size>
```

## 4.1 The Clause `SEMANTICS`

T4SQL enables the user to specify different semantics for different temporal dimensions (delegating their management to the underlying DBMS) as follows:

```
SEMANTICS <sem> [ON] <dim> [[TIMESLICE] <ts_exp>]
       {, <sem> [ON] <dim> [[TIMESLICE] <ts_exp>]}
```

where `<sem>` specifies the semantics to apply to the temporal dimension `<dim>`. The item `<ts_exp>` is a constant either of type `PERIOD` (for intervals) or of type `DATE` (for instants). Let $r$ be the accessed relation and let $d$ and $p$ be the values of `<dim>` and `<ts_exp>`, respectively. A tuple $t \in r$ is taken into consideration only if $t[d] \cap p \neq \emptyset$. In addition, the value of $t$ over $d$ is changed to $t[d] \cap p$. The tokens `ON` and `TIMESLICE` are optional, but they can be used for the sake of readability. Observe that, in a given query, a temporal dimension can occur only once in a `SEMANTICS` clause: a unique semantic interpretation can be given to any temporal dimension.

As an example, suppose that we want to extract from the relation `PatSymptom` the patients that suffered from "`Fever`", together with the time periods over which it happened. In a relational DBMS where temporal dimensions must be explicitly dealt with by the user, the query would appear as follows (we assume that the DBMS supports the `DATE` and `PERIOD` data types):

```
SELECT    PatId, ps.valid
FROM      PatSymptom AS ps
WHERE     Symptom = 'Fever' AND
          ps.transaction CONTAINS DATE 'uc'
```

In T4SQL the user only needs to specify the semantics of the involved temporal dimensions, whose management is delegated to the T4SQL interpreter. The resulting code turns out to be simpler, more readable, and less error-prone. The above query can be encoded in T4SQL as follows:

```
SEMANTICS SEQUENCED ON VALID
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

Such a query forces VT to be interpreted according to the `SEQUENCED` semantics and automatically deals with TT so that only current tuples are taken into consideration. If we want to restrict our attention to the patients that suffered from fever during 2005, we can take advantage of the `TIMESLICE` qualifier as follows:

```
SEMANTICS SEQUENCED ON VALID TIMESLICE
          PERIOD '[2005-01-01 - 2005-12-31]'
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

where the lower and upper bounds of a constant of type `PERIOD` are depicted by the symbols '`[`' and '`]`'.

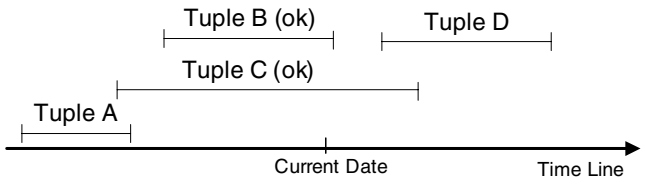


**Figure 1: an example of the CURRENT semantics.**

When processing a T4SQL statement, the SEMANTICS clause is interpreted before the FROM clause, since all the statements need to be interpreted according to the specified semantics. In the following, we shall analyze in detail the four available types of semantics, namely, ATEMPORAL, CURRENT, SEQUENCED, and NEXT.

### 4.1.1 ATEMPORAL *Semantics*

If the ATEMPORAL semantics is adopted, the corresponding attribute(s) is considered as atemporal (timeless), i.e., the ATEMPORAL semantics is exactly the same as the *non-sequenced* semantics of Section 2. The ATEMPORAL semantics provides the user with the highest level of freedom in managing temporal dimensions, where any support from the system is disabled.

As an example, to retrieve all the patients for which the symptom "Fever" has never been observed, we need to span over the entire valid time temporal axis. The set of patients can be computed by executing the following T4SQL query:

```
SEMANTICS ATEMPORAL ON VALID
SELECT    PatId
FROM      PatSymptom ps
WHERE     ps.PatId NOT IN (
  SEMANTICS ATEMPORAL ON VALID
  SELECT    PatId
  FROM      PatSymptom
  WHERE     Symptom = 'Fever')
```

By default, duplicates are removed, as it happens when the SQL DISTINCT qualifier is used.

The execution of a T4SQL query produces a relation with at most four temporal dimensions. As a general rule, the output relation does not include the temporal dimension(s) to which the ATEMPORAL semantics has been applied. As a consequence, there can be situations where the output relation is completely devoid of temporal dimensions. In the above example, the output relation is devoid of VT. To maintain the involved temporal dimensions in the output relation, the user must explicitly provide suitable functions to compute them. Taking advantage of that, one can execute an ATEMPORAL query to obtain the same result that could be obtained by applying any other semantics (as a matter of fact, the code of an ATEMPORAL query is a much more complex than the code of the corresponding specialized query).

### 4.1.2 CURRENT *Semantics*

When applied to a temporal dimension $d$, the CURRENT semantics selects all and only those tuples $t$ such that the current date belongs to $t[d]$ (see Figure 1).

The meaning of the CURRENT semantics depends on the temporal data type associated with the involved temporal dimension:

- if the data type is PERIOD (this is the case with VT, TT, and AT), the query considers all and only the tuples satisfying the condition $t[d]$ CONTAINS DATE 'now', if $d$ is VT, and $t[d]$ CONTAINS DATE 'uc', if $d$ is TT or AT, respectively;
- if the data type is DATE (this is the case with ET), the query considers all and only the tuples satisfying the condition $t[d] =$ DATE 'now'.

As an example, to retrieve all the patients from the relation PatSymptom as currently stored, we can execute the following T4SQL query:

```
SEMANTICS CURRENT ON TRANSACTION,
                     ATEMPORAL ON VALID
SELECT    PatId
FROM      PatSymptom
```

Such a query takes into consideration the tuples whose TT contains the current date (timestamp) only. The output relation contains neither TT nor VT. As it happens with the ATEMPORAL semantics, it does not include the temporal dimension(s) to which the CURRENT semantics has been applied. In such a way, we guarantee the compatibility of T4SQL with SQL92 (we shall further discuss this later). Moreover, as the CURRENT semantics only considers the current state of the database with respect to a given temporal dimension, that is, the current date, there is no reason to include by default a temporal dimension in the output relation (the output is implicitly associated with the time at which the query is evaluated).

In the above query, the CURRENT semantics can be replaced with the ATEMPORAL semantics as follows:

```
SEMANTICS ATEMPORAL ON TRANSACTION,
                       ATEMPORAL ON VALID
SELECT    PatId
FROM      PatSymptom AS ps
WHERE     TRANSACTION(ps) CONTAINS DATE 'uc'
```

On the one hand, the ATEMPORAL semantics allows the user to include transaction time in the output relation (as in Section 4.2)[4]. On the other hand, the complexity of the resulting query is higher because the user must explicitly manage transaction time.

The CURRENT semantics cannot be coupled with TIMESLICE as CURRENT can be seen as a TIMESLICE selecting the current date only. As an example, a query including the statement CURRENT ON VALID TIMESLICE DATE '2006-01-01' is invalid. The above query can be rewritten as follows:

```
SEMANTICS ATEMPORAL ON TRANSACTION
          TIMESLICE DATE 'uc', ATEMPORAL ON VALID
SELECT    PatId
FROM      PatSymptom
```

### 4.1.3 SEQUENCED *Semantics*

As in existing temporal query languages (see Section 2), the SEQUENCED semantics forces a *step by step* evaluation of the statement. When applied to a temporal dimension $d$, it takes into consideration all the combinations of the tuples

[4]This is the only advantage of using the ATEMPORAL semantics instead of the CURRENT semantics.

of the relations that occur in the FROM clause whose time periods over $d$ have a nonempty intersection.

Let us consider, for instance, the SEQUENCED semantics over VT. For every date over the VT axis, we select all and only those tuples where the VT contains the considered date.

The SEQUENCED semantics allows one to perform historical analyses that, for every date, only consider information that holds at that date. Unlike the ATEMPORAL and CURRENT semantics, the temporal dimension on which the SEQUENCED semantics is applied is included in the output relation. By default, the VT value of every tuple in the output relation is determined as follows. Let $T_1, \cdots, T_n$ be the relations that contribute to the result, $d$ be the temporal dimension to which the SEQUENCED semantics is applied, and $T_r$ be the output relation. For every tuple $t_r \in T_r$ and every $t_i \in T_i$, with $1 \leq i \leq n$, that contribute to the generation of $t_r$, we have that $t_r[d] = \bigcap_{i=1}^{n} t_i[d]$.

Consider again the case of VT. According to the given rule, every tuple $t_r$ in the output relation is valid at all the time instants where all the tuples that contribute to it are valid and thus the VT of every tuple in the output relation is the intersection of the VT of the contributing tuples.

The following query returns, for every patient, the symptoms whose VT overlaps the VT of the prescribed therapy according to the current state of the database:

```
SEMANTICS SEQUENCED ON VALID,
          CURRENT ON TRANSACTION
SELECT    Symptom, PatId
FROM      PatSymptom AS ps, PatTherapy AS pt
WHERE     ps.PatId = pt.PatId
```

When a query involves one relation only, the SEQUENCED semantics takes into consideration all its tuples and associates with the tuples in the output relation the value they assume on the considered temporal dimension.

The following query returns the patients who suffered from fever on the basis of what the database in its current state believed correct on December $1^{st}$, 2006:

```
SEMANTICS SEQUENCED ON VALID, CURRENT ON
  TRANSACTION, SEQUENCED ON AVAILABLE
  TIMESLICE PERIOD '[2006-12-01 - 2006-12-01]'
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

The result is a temporal relation with one explicit attribute (Patid) and one implicit attribute (VT) managed by the system. Both the SEQUENCED and the CURRENT semantics can be replaced with the ATEMPORAL semantics as follows (the particular form that the SELECT clause assumes will be described in Section 4.2):

```
SEMANTICS ATEMPORAL ON VALID, ATEMPORAL ON
  TRANSACTION, ATEMPORAL ON AVAILABLE
SELECT   PatId WITH VALID(ps) AS VALID
FROM     PatSymptom AS ps
WHERE    Symptom = 'Fever' AND
         TRANSACTION(ps) CONTAINS DATE 'uc' AND
         AVAILABLE(ps) CONTAINS DATE '2006-12-01'
```

As a general rule, any query involving the SEQUENCED semantics can be replaced by a query involving the ATEMPORAL semantics.

### 4.1.4 NEXT Semantics

Given a temporal dimension, the NEXT semantics allows the user to retrieve information about the same object as observed in two consecutive dates over it (any temporal dimension takes value over an ordered temporal domain), that is, the NEXT semantics takes into consideration only pairs of tuples related to the same entity (i.e., pairs of tuples of a given relation with the same *snapshot* key, as shown in Section 4) and selects those pairs which are consecutive with respect to the temporal ordering. As an example, we may use the NEXT semantics to establish, for every patient, the time elapsed between two consecutive administrations of the same therapy.

When the NEXT semantics is applied, the query first executes a join between two instances of the same relation in the FROM clause and then it removes the pairs of tuples which do not feature the same snapshot key; next, it selects the pairs of tuples with consecutive time values. Thanks to the snapshot key constraint, the tuples with the same value of the snapshot key are totally ordered with respect to their time values (in the cases of VT, TT, and AT) and thus the successor of any tuple (if any) is unique, provided that we do not adopt the ATEMPORAL semantics for some of the other temporal dimensions. Moreover, in the case of $\mathsf{ET}_i$ and $\mathsf{ET}_t$, we deal with (instantaneous) time values of DATE type, while, in the cases of VT, TT, and AT, we deal with time values of type PERIOD.

Figures 2.a and 2.b show the temporal positions of a set of tuples, related to the same entity, whose temporal dimension takes value over the DATE data type (for $\mathsf{ET}_i$ and $\mathsf{ET}_t$) and over the PERIOD data type (for VT, TT, and AT), respectively. The successor relation over the set of tuples induced by the NEXT semantics is represented by means of arrows. Figure 2.a considers a temporal dimension of type DATE and shows the case of a tuple with multiple successors, while Figure 2.b considers a temporal dimension of type PERIOD, where every tuple has (at most) one successor.

By default, the NEXT semantics refers to the whole temporal domain: the query looks for the successor of a given tuple starting from the right endpoint of its time interval till the very last instant representable by the system. However, a suitable integer parameter of the NEXT semantics makes it possible to constrain the width of the interval over which the successor of the tuple can be searched. The complete syntax for the NEXT semantics is:

```
NEXT[(<duration>)] [ON]
      <dimension> [[TIMESLICE] <ts_exp>]
```

where <duration> is an integer number that defines the width of the time interval where the successor (if any) must be found and <dimension> is the temporal dimension the NEXT semantics must be applied to. The <duration> parameter refers to the smallest granularity of the considered temporal data type. For instance, in the case of VT, the type is PERIOD and thus <duration> refers the day granularity.

Let us consider again the example depicted in Figure 2.b. To restrict the search to pairs of tuples with adjacent time intervals, we apply the semantics NEXT(0). The effects of such a restriction are shown in Figure 3: the pair of intervals B and C is considered, while the pair of intervals A and B is disregarded.

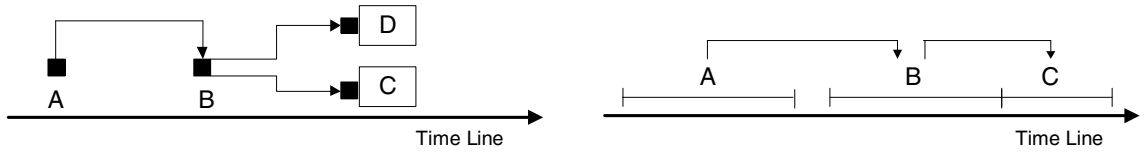


**Figure 2: a) NEXT semantics over values of type DATE; b) NEXT semantics over values of type PERIOD.**

EXAMPLE 4.1. *Given the PatTherapy relation, to retrieve for every patient the time period elapsed between two consecutive administrations of the same drug, we can execute the following T4SQL query:*

```
SEMANTICS NEXT ON VALID
SELECT  PatId, Therapy,
        (BEGIN(VALID(NEXT(pt)))-END(VALID(pt))) DAY
FROM    PatTherapy AS pt
```

The function NEXT used in the SELECT clause is applied to a tuple variable pt and returns the successor tuple NEXT(pt), according to the NEXT semantics. BEGIN(VALID(NEXT(pt))) thus returns to the left endpoint of the successor tuple VT.

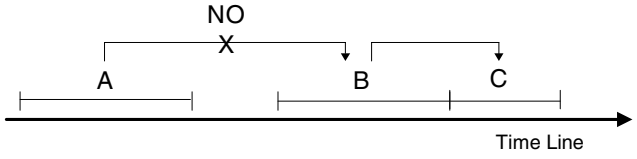


**Figure 3: NEXT(0) semantics over values of type PERIOD.**

EXAMPLE 4.2. *To retrieve all the patients who received a therapy 'Paracetamol' moving from a dosage 'd1' to a dosage 'd2', we can execute the following T4SQL query:*

```
SEMANTICS NEXT(0) ON VALID
SELECT    PatId
FROM      PatTherapy AS pt
WHERE     pt.Therapy = 'Paracetamol' AND
          pt.Dosage = 'd1' AND
          NEXT(pt).Dosage = 'd2'
```

By default the temporal dimension to which the NEXT semantics is applied does not belong to the output relation. As a general rule, indeed, T4SQL does not associate any time value to a tuple obtained from two tuples with disjunct time intervals over the considered temporal dimension. However, the user can always force the inclusion of such a dimension in the output relation and define a way to compute its value by using the WITH clause described below.

### 4.1.5 Default Values

When temporal dimensions or their related temporal qualifiers are missing in the SEMANTICS clause, default values apply. Compatibility of T4SQL with respect to SQL92 drives the choices of default values.

Let us define *atemporal*, or *snapshot*, a query devoid of any construct for the management of temporal dimensions. In T4SQL any atemporal query over a temporal relation produces the same result that would be produced by the execution of the query over the corresponding atemporal relation by a non-temporal DBMS. This definition conforms the condition of *upward compatibility* reported in Section 2.

Let us consider a query compatible with SQL92, which obviously cannot include the SEMANTICS clause. In T4SQL any such query, when evaluated over a temporal relation, only takes into consideration information known by the DBMS and valid at query execution time and it returns an atemporal relation which is computed according to the semantics of SQL92. This requires the default semantics for VT, TT, and AT to be the CURRENT semantics: the query only considers information associated with the current state of the database (with the same semantics as in SQL92) and it does not include any temporal dimension in the output relation.

A more accurate analysis is needed for ET. Assuming the CURRENT semantics as the default semantics for ET would force both $ET_i$ and $ET_t$ to coincide with the current date, which is clearly meaningless. As an example, consider a tuple with VT equal to [2006-01-01, 2006-12-31], belonging to a co-active relation, and assume the current date to be 2006-06-01. The tuple turns out to be valid at the current date, but it would not be considered because both its $ET_i$ and its $ET_t$ differ from the current date. For these reasons, the default semantics for ET in T4SQL is the ATEMPORAL semantics, which does not influence the tuple selection process and does not force the inclusion of ET in the output relation.

As an example, given the PatSymptom relation, to retrieve the patients who are suffering from 'Fever', we can execute the following SQL92 query devoid of temporal conditions:

```
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

The output relation is an atemporal one that returns information which is valid at the current time. The equivalent T4SQL encoding of the above query is:

```
SEMANTICS CURRENT ON VALID,
          CURRENT ON TRANSACTION,
          CURRENT ON AVAILABLE,
          ATEMPORAL ON INITIATING_ET,
          ATEMPORAL ON TERMINATING_ET
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

Default values apply to the TIMESLICE qualifier as well. The default value for the TIMESLICE period is the temporal domain of the involved temporal dimension. As an example, the query:

```
SEMANTICS SEQUENCED ON VALID
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

is equivalent to the query:

```
SEMANTICS SEQUENCED ON VALID
     TIMESLICE PERIOD '[0000-01-01 - 9999-12-31]'
SELECT    PatId
FROM      PatSymptom
WHERE     Symptom = 'Fever'
```

We conclude this part about default values by pointing out that the *temporal upward compatibility* constraint of Section 2 is respected as, for any temporal relation, T4SQL only considers current information for VT, TT, and AT.

## 4.2 The Clause SELECT

The SELECT clause of SQL92 executes a projection on the attributes to be included in the output relation. T4SQL deals with both temporal and atemporal relations, and thus it must couple the management of explicit attributes (those included in the SELECT clause) with that of temporal dimensions. To separate explicit attributes from implicit ones, T4SQL uses the token WITH in the SELECT clause. Any previous statement is evaluated according to the SQL92 semantics for SELECT, while any subsequent statement is used to compute the temporal dimension(s) to be included in the output relation. The syntax of the T4SQL SELECT clause is:

```
SELECT  <sel_element_list>
 [WITH <w_exp> [AS] <dim> {, <w_exp> [AS] <dim>}]
```

where <w_exp> is an expression that returns a period (if the temporal dimension takes value over a PERIOD data type) or a date (if it takes value over a DATE data type), and <dim> is a temporal dimension. The token AS can possibly be used to increase readability.

As we already explained, if no temporal dimension is specified for the output relation, the following rules are applied: temporal dimensions on which ATEMPORAL, CURRENT, and NEXT semantics are applied are not included in the output relation; temporal dimensions on which the SEQUENCED semantics is applied are included in the result relation and the time interval of every tuple in the output relation is the intersection of the time intervals of the tuples in the input relations contributing to it.

EXAMPLE 4.3. *To retrieve the patients who suffered from the symptom 'Fever', received the drug 'Paracetamol', and solved their disease within 5 days from the beginning of the therapy, we can execute the following T4SQL query, where the VT value of every tuple in the output relation is a subinterval of the interval that has the beginning of the symptom as its left endpoint and the end of the therapy as its right endpoint:*

```
SEMANTICS SEQUENCED ON VALID
SELECT    PatId WITH PERIOD (BEGIN(VALID(ps)),
            END(VALID(pt))) AS VALID
FROM      PatSymptom AS ps, PatTherapy AS pt
WHERE     ps.Symptom = 'Fever' AND
          pt.Therapy = 'Paracetamol' AND
          ps.PatId = pt.PatId AND
          (END(VALID(ps)) - BEGIN(VALID(pt))) DAY
                              < INTERVAL '5' DAY
```

The output relation has the VT attribute. However, the VT interval of the tuples in the output relation is not equal to the intersection of the VT intervals of the contributing tuples, but it is computed according to the expression following the token WITH.

The last ingredient of the T4SQL SELECT clause is the coalescing operator. In the T4SQL data model temporal relations must satisfy the *snapshot key* constraint. However, the SELECT clause may produce a temporal relation including pairs of tuples with the same value on all the atemporal attributes and with values that have a nonempty intersection on temporal one(s). This can be avoided by using the *coalescing* operator that merges tuples with overlapping or adjacent time values. As an example, Table 2 and Table 3 show how the output of an hypothetical query looks like before and after the application of the coalescing operator, respectively.

| PatId | VT |
|---|---|
| 1 | [2006-01-01 - 2006-06-01] |
| 1 | [2006-05-01 - 2006-08-01] |
| 1 | [2006-09-01 - 2006-09-30] |
| 1 | [2006-10-01 - 2006-10-20] |
| 1 | [2006-10-21 - 2006-11-01] |

**Table 2: query result before *coalescing*.**

| PatId | VT |
|---|---|
| 1 | [2006-01-01 - 2006-08-01] |
| 1 | [2006-09-01 - 2006-11-01] |

**Table 3: query result after *coalescing*.**

## 4.3 The Clause FROM

The FROM clause of SQL92 specifies the relations that contribute to the execution of the selection and join operations and/or to the definition of the output relation. As for join operations, SQL92 features various forms of join, including the INNER join (which is the default), the NATURAL JOIN, and the LEFT OUTER, RIGHT OUTER and FULL OUTER joins. The token ON specifies the join condition (if any). The FROM clause of T4SQL only differs from that of SQL92 in the replacement of the token JOIN with the token TJOIN (*temporal join*).

Temporal conditions depend on the adopted semantics, that is, explicit user-defined conditions are coupled with implicit temporal conditions induced by the semantics. More precisely:

- when the CURRENT semantics is adopted, T4SQL only considers tuples whose time intervals include the current date;
- when the SEQUENCED semantics is adopted, T4SQL only considers combinations of tuples from the relations in the FROM clause with overlapping time intervals;
- when the NEXT semantics is adopted, T4SQL only considers pairs of consecutive tuples.

EXAMPLE 4.4. *To retrieve, according to the current state of the database, all the patients which, during the assumption of a therapy, showed any symptom that lasted for a period greater than 10 days, we can execute the following T4SQL query:*

```
SEMANTICS SEQUENCED ON VALID
SELECT    PatId, Therapy, Symptom
FROM      PatTherapy AS pt TJOIN
          PatSymptom AS ps ON
          (pt.PatId = ps.PatId)
WHERE     (END(VALID(ps)) - BEGIN(VALID(ps))) DAY
                              > INTERVAL '10' DAY
```

## 4.4 The Clauses WHERE and WHEN

In SQL92, the WHERE clause evaluates selection predicates over tuples from the relations in the FROM clause. The WHERE clause of T4SQL extends the SQL92 clause possibly including temporal conditions. As temporal conditions may turn out to be very complex, the user can optionally separate temporal conditions from atemporal ones by using the WHEN clause. The semantics of WHEN is very similar to that WHERE, but it only includes temporal conditions. Consider again the query of Example 4.4. By using the WHEN clause, it can be rewritten as follows:

```
SEMANTICS SEQUENCED ON VALID
SELECT    PatId, Therapy, Symptom
FROM      PatTherapy AS pt, PatSymptom AS ps
WHERE     pt.PatId = ps.PatId
WHEN      (END(VALID(ps)) - BEGIN(VALID(ps))) DAY
                              > INTERVAL '10' DAY
```

## 4.5 The Clauses GROUP BY and HAVING

In a temporal query language, the GROUP BY clause can be used to implement a *temporal grouping*, rather than an instantaneous grouping over atemporal attributes. As a general rule, the selection of the tuples to be grouped together (we have one group for every partitioning element) is performed according to the value of the considered temporal attribute(s), the periods associated with the different elements of the partition, and the comparison operator used in the temporal grouping.

Let $P$ be the time interval associated with a given element of the partition, $P_t$ be the value taken by a tuple $t$ on the considered temporal attribute, and $Op$ be the temporal comparison operator. The tuple $t$ will belong to the group associated with the considered element of the partition if and only if $P\ Op\ P_t$ holds. To simplify the specification of temporal grouping, OVERLAPS is taken as the default value of the comparison operator. Given a tuple $t$ and an element of the partition with time interval $P$, if $P_t$ OVERLAPS $P$, then $t$ belongs to the group associated with such an element; moreover, the value of its temporal attribute, with respect to such a group, is equal to $P \cap P_t$.

The result of the temporal grouping can be explained as follows. The time interval $P_t$ associated with the temporal attribute of the tuple can be viewed as a set of dates. Whenever $P_t$ and the time interval $P$ of the element of the partition satisfy the requested condition, e.g., whenever they overlap, the tuple belongs to the group induced by the element of the partition and the time interval associated with the temporal attribute is restricted to the set of dates included in $P$.

Consider the case of a temporal grouping, based on VT, that partitions the temporal domain in months. Figure 4 shows the partition of the VT domain and a set of tuples with their periods of validity. With every element of the partition we associate the group of tuples whose VT overlaps

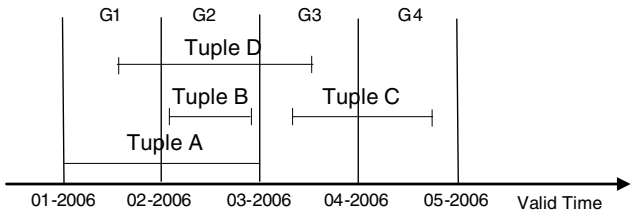


**Figure 4: a monthly-base VT partitioning.**

the time interval of the element. The result is summarized in Table 4.

| Group | Tuple | Valid Time |
|---|---|---|
| G1 | A | [2006-01-01 - 2006-01-31] |
| | D | [2006-01-10 - 2006-01-31] |
| G2 | A | [2006-02-01 - 2006-02-28] |
| | B | [2006-02-05 - 2006-02-23] |
| | D | [2006-02-01 - 2006-02-28] |
| G3 | C | [2006-03-10 - 2006-03-31] |
| | D | [2006-03-01 - 2006-03-15] |
| G4 | C | [2006-04-01 - 2006-04-20] |

**Table 4: the result of the temporal grouping.**

The token USING distinguishes between classic (atemporal) grouping and temporal grouping. Implemented temporal partitions are SECOND, MINUTE, HOUR, DAY, MONTH, and YEAR. An an example, the temporal grouping: "GROUP BY <temp_attribute> USING MONTH" groups the tuples on the basis of a monthly partition of the domain of the temporal dimension <temp_attribute>.

The grouping attribute(s) can possibly be included in the SELECT clause by means of the token TGROUPING(...). Such a token forces T4SQL to include in the SELECT clause as many (time interval) attributes as the parameters of the temporal grouping are, possibly renamed as specified by the user. As a result, for every grouped temporal dimension, T4SQL couples any temporal group in the resulting table with a constant value of type PERIOD that specifies the value of the time slice corresponding to the group.

T4SQL assigns to grouping the same semantics as SQL92 does. In addition, it introduces the token WEIGHTED to be used in combination with the functions MAX, MIN, AVG, and SUM. Such a token must precede the aggregation function it is applied to and computes a weighted version of the function that takes into account the size of tuple time intervals.

Let $t_1, \ldots, t_n$ be the tuples belonging to a given temporal group (interval) $G$, $t_1[d], \ldots, t_n[d]$ be the values of the tuples on the temporal attribute $d$ over which the temporal grouping has been done, and $dur(x)$ be the duration of the time interval $x$. Moreover, let $t_1[a], \ldots, t_n[a]$ be the values of the tuples on the attribute $a$ which is the parameter of the weighted aggregate function. The weighted versions of the functions are computed as follows:

- WEIGHTED MAX(a) $= max_{i=1}^{n}\{t_i[a] \cdot \frac{dur(t_i[d])}{dur(G)}\}$;
- WEIGHTED MIN(a) $= min_{i=1}^{n}\{t_i(a) \cdot \frac{dur(t_i[d])}{dur(G)}\}$;
- WEIGHTED SUM(a) $= \sum_{i=1}^{n}(t_i[a] \cdot \frac{dur(t_i[d])}{dur(G)})$;

- WEIGHTED AVG(a) $= \frac{\sum_{i=1}^{n}(t_i[a] \cdot \frac{dur(t_i[d]))}{dur(G)}}{n}$.

The syntax of the GROUP BY clause is the following:

```
GROUP BY <list_group_element>
<list_group_element> ::= <group_element>
                       {, <group_element>}
<group_element> ::= <attribute> |
      <temp_attribute> USING <part_size>
```

where <temp_attribute> denotes a temporal dimension and <part_size> specifies the size of the partition which can assume the values SECOND, MINUTE, HOUR, DAY, MONTH, or YEAR.

The syntax of the aggregation functions is [WEIGHTED] <function> (<attribute>). The novelty is the above-described optional token WEIGHTED. The HAVING clause of T4SQL basically coincides with that of SQL92 (the only difference is that weighted functions can be used in it).

EXAMPLE 4.5. *To retrieve, for every year, the average duration of prescribed therapies, we can execute the following T4SQL query:*

```
SELECT    TGROUPING(VALID(pt) AS TherYear),
          AVG(CAST(INTERVAL(VALID(pt) DAY))
          AS INTEGER)
FROM      PatTherapy AS pt
GROUP BY  VALID(pt) USING YEAR
```

EXAMPLE 4.6. *Given the table PatSymptom, to compute the average level of weighted severity (SevLevel is in a range 1÷10) for patients who suffered from fever and dry cough every month, returning only the patients where this average value is greater than 3, we can execute the following T4SQL query:*

```
SELECT    PatId, TGROUPING(VALID(t) AS SymMonth),
          WEIGHTED AVG(SevLevel)
FROM      PatSymptom AS ps
WHERE     Symptom = 'Fever and dry cough'
GROUP BY  ps.PatId, VALID(ps) USING MONTH
HAVING    WEIGHTED AVG(SevLevel) > 3
```

# 5. CONCLUSIONS

In this paper, we proposed a new query language, called T4SQL, that operates on multidimensional temporal relations. T4SQL allows one to query temporal relations provided with (a subset of) the temporal dimensions of *valid*, *transaction*, *availability*, and *event* time, according to different semantics. Among the distinctive features of T4SQL we mention (i) the management of the NEXT semantics, that allows one to select pairs of temporally consecutive tuples; (ii) the ability of explicitly assigning a value to every temporal dimension of the tuples in the output relation, and (iii) the support for temporal grouping.

Any T4SQL query can be translated into an equivalent (atemporal) SQL query. However, as already shown in [12] for VT and TT queries, the corresponding SQL queries are more complex, their size is bigger, and their execution is often quite inefficient. This is the case, for instance, with queries involving temporal grouping. The corresponding SQL queries must indeed include suitable statements to properly manage the temporal bounds of the elements of the partition.

As for the ongoing work, we are exploring some techniques for translating T4SQL queries into efficient SQL queries. Moreover, we plan to extend the temporal data model by data definition and data management capabilities.

**Acknowledgments.** We would like to thank our former students Fabio Valeri and Federico Moretto for their contribution to the development of T4SQL.

# 6. REFERENCES

[1] Sharma Chakravarthy and Seung-Kyum Kim. Resolution of time concepts in temporal databases. *Inf. Sci.*, 80(1-2):91–125, 1994.

[2] Cindy Xinmin Chen, Jiejun Kong, and Carlo Zaniolo. Design and implementation of a temporal extension of SQL. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 689–691. IEEE Computer Society, 2003.

[3] Carlo Combi and Angelo Montanari. Data models with multiple temporal dimensions: Completing the picture. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *CAiSE*, volume 2068 of *LNCS*, pages 187–202. Springer, 2001.

[4] Andrew Eisenberg and Jim Melton. SQL standardization: The next steps. *SIGMOD Record*, 29(1):63–67, 2000.

[5] Christian S. Jensen et al. The consensus glossary of temporal database concepts - February 1998 version. In *Temporal Databases, Dagstuhl*, pages 367–405, 1998.

[6] Fabio Grandi, Federica Mandreoli, and Paolo Tiberio. Temporal specialization and generalization. *Data & Knowl. Eng.*, 54:327–354, 2005.

[7] James Green and Roger Johnson. ProSQL: A prototyping tool for SQL temporal language extensions. In Anne E. James, Brian Lings, and Muhammad Younas, editors, *BNCOD*, volume 2712 of *LNCS*, pages 190–197. Springer, 2003.

[8] Keith W. Hare. JCC's SQL standards page. (accessed: September 1, 2006). http://www.jcc.com/sql.htm, February, 2006.

[9] Christian S. Jensen and Richard T. Snodgrass. Temporal specialization and generalization. *IEEE Trans. Knowl. Data Eng.*, 6(6):954–974, 1994.

[10] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

[11] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[12] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 2000.

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

[14] Jose Ramon Rios Viqueira and Nikos A. Lorentzos. SQL extension for spatio-temporal data. *VLDB Journal*, 16(2):179–200, 2007.

[15] Esteban Zimányi. Temporal aggregates and temporal universal quantification in standard SQL. *SIGMOD Record*, 35(2):16–21, 2006.