A Relational Encoding of a Conceptual Model with Multiple Temporal Dimensions

Donatella Gubiani and Angelo Montanari

Department of Mathematics and Computer Science,
University of Udine, Italy

Abstract. The theoretical interest and the practical relevance of a systematic treatment of multiple temporal dimensions is widely recognized in the database and information system communities. Nevertheless, most relational databases have no temporal support at all. A few of them provide a limited support, in terms of temporal data types and predicates, constructors, and functions for the management of time values (borrowed from the SQL standard). One (resp., two) temporal dimensions are supported by historical and transaction-time (resp., bitemporal) databases only. In this paper, we provide a relational encoding of a conceptual model featuring four temporal dimensions, namely, the classical valid and transaction times, plus the event and availability times. We focus our attention on the distinctive technical features of the proposed temporal extension of the relation model. In the last part of the paper, we briefly show how to implement it in a standard DBMS.

1 Introduction

Despite the pervasiveness of temporal information, most databases (and information systems) basically maintain information about the current state of the world only. Temporal databases can be viewed as an attempt to overcome this limitation, making it possible to keep track of the evolution of the domain of interest (valid time dimension) and/or of the database contents (transaction time dimension). The valid time of a fact can be defined as the time when the fact is true in the modeled domain, while its transaction time is the time when it is current in the database and may be retrieved. Historical (resp., transaction-time) relational databases support the valid (resp., transaction) time dimension. Relational databases that manage both dimensions are called bitemporal databases. In [3], two additional temporal dimensions, respectively called event and availability time, have been proposed to remedy to some weaknesses of valid and transaction times. The event time of a fact is defined as the pair of occurrence times of the real-world events that respectively initiate and terminate its validity interval, while its availability time is the time interval during which it is known and believed correct by the information system the database belongs to (in general, such an interval does not coincide with its transaction time interval). No effective support to these dimensions is provided by existing temporal relational databases. A comprehensive and up-to-date survey of temporal databases can be found in [6].

S.S. Bhowmick, J. Küng, and R. Wagner (Eds.): DEXA 2009, LNCS 5690, pp. 792–806, 2009.

The contribution of this paper is part of the work done within ChronoGeoGraph (CGG) Project [2], which aims at developing a software framework for the conceptual and logical design of spatiotemporal databases. The core of the framework is the CGG model, a conceptual model that extends the classical Enhanced Entity-Relationship model (EER) with additional constructs for spatiotemporal information [5].

As for the spatial features, CGG supports a large set of representation primitives for spatial data. CGG distinguishes between spatial and non spatial entities. A spatial entity is characterized by a set of descriptive and spatial attributes plus a geometry of a given spatial data type (CGG supports 8 different spatial data types). Spatial attributes take their value over a spatial data type as well. A spatial dimension can be added to relations as well. CGG supports topological, metric and direction relations, and the relation of spatial aggregation (the part-of relation over spatial entities). Besides the usual relation of specialization, CGG introduces the relation of cartographic specialization, which supports different spatial representations of the same spatial entity. Finally, CGG supports the field-based view of spatial information by the notion of (spatial) field and the notion of schema territory, which defines the spatial domain over which all spatial elements of the schema are located.

As for the temporal features, CGG allows one to temporally qualify the various constructs by properly annotating them. One or more temporal dimensions can be associated with the schema territory, entities, attributes, relations, and fields. Different temporal dimensions are associated with different constructs. Entities can be provided with an existence time (which can be viewed as the valid time of the entity), possibly paired with a state diagram, a transaction time, an event time, and an availability time. The other constructs can be endowed with a valid time, a transaction time, an event time, and an availability time. Furthermore, CGG introduces a distinction between snapshot and lifespan cardinality constraints for attributes and relations. Snapshot cardinality constraints specify the minimum and maximum number of values that an attribute can take (resp., of instances of a given entity that may participate in a relation) at a given time, while lifespan cardinality constraints specify minimum and maximum bounds with respect to the whole existence of the entity instance (resp., the validity interval of the relation instance). As for attributes, CGG also allows one to collect sets of attributes of a given entity that change in a synchronous way (it defines a temporal collection as a set of entity attributes with a common temporal annotation). Finally, it explicitly keeps track of the events that affect a relevant element, e.g., events that change the state and/or the geometry of an entity, the validity of a relation, the value of an attribute.

The paper addresses the problem of providing a relational encoding of temporal information in CGG schemas. A special attention will be deserved to the management of temporal dimensions. The distinctive features of the proposed temporal extension of the relational model are the use of tuple timestampings, the partition of temporal schemas (resp., instances) into a current component and a historical one, and the development a number of constraints that guarantee

the consistency of the values of the different temporal dimensions. In addition, we implemented the extended temporal model in a standard DBMS, taking advantage of SQL asssertions and triggers, and we developed a translation algorithm mapping CGG schemas into temporally-extended relational ones.

The rest of the paper is organized as follows. In Section 2 we give a short account of existing temporal relational models. In Section 3 we describe the basic features of the proposed temporal relational model supporting the temporal dimensions of valid, transaction, event, and availability times. We first consider the single temporal dimensions in isolation and then we analyze their interactions. In Section 4 we focus our attention on the specification of temporal keys. Section 5 provides some details about the implementation of the model in a specific DBMS. Finally, in Section 6 we briefly illustrate the translation of CGG schemas into the proposed model.

2 An Account of Existing Temporal Relational Models

The basic relational model only supports temporal data types, e.g., `Date` and `Timestamp`, and predicates, constructors, and functions for the management of time values. It provides no primitives to explicitly deal with temporal dimensions. Various extensions to the relational model have been proposed in the literature to support the valid and/or transaction time dimensions.

Temporal databases can be classified according to the granularity of timestamping, the nature of timestamps, and the temporal interpretation of the primary key. All temporal databases associate one or more timestamp attributes (timestamps for short) with facts, for every supported temporal dimension. The most common options are associating a single timestamp with the whole tuple (tuple timestamping) and a distinct timestamp with any temporal attribute (attribute timestamping). The former preserves First Normal Form (1NF) and its implementation is straightforward; in addition, it allows one to benefit from the standard relational database technology. However, the resulting tables suffer from two weaknesses: data redundancy and vertical anomaly (information about a domain object is not recorded in a single tuple, but it is spread over various tuples). The latter is not affected by the vertical anomaly, because it records the entire history of every domain object in a single tuple. However, in doing that it violates 1NF: for every tuple and every temporal dimension, the value of each temporal attribute is a set of pairs (value, timestamp). As for the nature of timestamps, three different choices of increasing complexity have been considered: time instants, time intervals, and temporal elements. In most cases (as an example, this is not the case with aggregations over time), time intervals are not interpreted as primitive temporal entities, but just as (convex) sets of time instants. In its turn, temporal elements are usually defined as a finite set of pairwise disjoint time intervals. Time intervals and temporal elements allow one to obtain a succinct representation of valid/transaction time periods, but their manipulation is more complicate: either it requires to transform them into time instants, to apply the necessary operations on such instants, and to provide an encoding of the result at the time interval/temporal element level or it imposes the introduction of additional non-trivial operations, such as coalescing.

Replacing a set of contiguous time instants with a single time interval makes it possible to overcome the problem of tuple-timestamping (vertical anomaly and redundancy) and attribute-timestamping (redundancy). However, these problems show up again as soon as a single time intervals must be replaced with two or more ones. The replacement of time intervals with temporal elements solves them, but it involves the violation of 1NF. Finally, there exist different ways of reinterpreting the notion of primary key in the temporal setting. Every temporal relation is obtained by extending an atemporal relation with one or more timestamps. Its temporal key can be defined as a set of (non-temporal) attributes which is a primary key for every temporal snapshot, as in [8,7]. As an alternative, one can introduce an explicit tuple identifier, which plays the same role of the object identifier in the object-oriented model. As a third possibility, one can define the temporal key as a combination of the primary key of the original atemporal relation and a suitable subset of timestamps, e.g., [1].

In the following, we will describe an original temporally-extended relational model supporting the four temporal dimensions described above. It opts for tuple-timestamping, to preserve 1NF, it assumes temporal homogeneity for all relations (a tuple holds over a given interval if and only if it holds at all time instants belonging to it), it makes use of time intervals (resp., time instants) to model valid, transaction, and availability times (resp., event time), and it defines temporal keys as suitable temporal extensions of the primary keys of the original atemporal relations. The closest relatives of such a model are the Time Relational Model (TiRM), the Temporal Relational Model (TRM), and the Historical DataBase Management System (HDBMS). Ben-Zvi's TiRM model [1] supports three temporal dimensions: (i) the *effective time* of a fact, which corresponds to valid time, (ii) the *registration time* of a fact, which is the pair of time instants at which the beginning and ending of its effective time interval are inserted into the database, and (iii) the *deletion time* of a fact, which is the time instant at which it is logically deleted (the combination of registration and deletion times can be viewed as a counterpart of transaction time). TiRM associates five timestamps with every tuple, namely, T_{es} and T_{ee} (for the beginning and ending of effective time), T_{rs} and T_{re} (for the beginning and ending of registration time), and T_d (for the deletion time). T_{es} and T_{ee} are specified by the user, while T_{rs}, T_{re}, and T_d are generated by system. The temporally-extended tuple is called *tuple version*. Tuples with the same value for the atemporal key are called *tuple version set*. A temporal relation is defined as a set of tuple version sets, rather than a set of tuples. Navathe and Ahmed's TRM model [10] supports one temporal dimension only, which corresponds to valid time. It distinguishes between the set R_s of static (atemporal) relations and the set R_t of time-varying (valid-time) relations. Every time-varying relation includes two timestamps t_s and t_e that record the left and right endpoints of valid-time intervals, respectively. The key of a time-varying relation consists of the primary key of its atemporal part (*time-invariant key*, TIK for short) and the timestamp t_s (since the value of t_e can be unknown, the pair (TIK,t_e) is not an alternative key). Sarda's HDBMS model [12] supports one temporal dimension only as well, called real valid time, which corresponds to valid time. The aim of Sarda was to develop

a temporal DBMS that receives as input a set of atemporal relation schemas and provides a subset of them (specified by the designer) with a temporal extension. The model allows one to distinguish between properties (historical relations) and instantaneous events. The system automatically associates two timestamps, `from` and `to`, with historical relation schemas, to keep track of their historical evolution, and a single timestamp `at` with events, to record their occurrence time. It allows the timestamps of different temporal relations to refer to different time granularities. The tuples of each historical relation are partitioned in two classes: the *current segment*, which contains only tuples belonging to the current state (tuple whose timestamp `from` has value `null`), and the *history segment*, which contains tuples representing historical data (tuples such that `from` $<$ `now`). New tuples are first inserted in the current segment and later, when their real valid interval ends, moved to the history one. The primary key of a relation in the current segment is defined as in the basic relational case. The key of a relation in the history fragment is defined as follows: a set of (atemporal) attributes $\mathbf{K}$ is a key for a relation $R(\mathbf{X})$, with $\mathbf{K} \subseteq \mathbf{X}$, if for any value $\mathbf{k}$ of $\mathbf{K}$ and any time instant t there is at most one tuple in $R(\mathbf{X})$ with value $\mathbf{k}$ for $\mathbf{K}$ whose real valid interval includes t (keys are time-unique, rather than tuple-unique as in the relational model).

3 A Relational Model with Four Temporal Dimensions

In the following, we describe a temporal extension to the relational model that supports the temporal dimensions of valid, transaction, availability, and event time. As a matter of fact, the resulting model can be viewed as the relation counterpart of the spatio-temporal conceptual model ChronoGeoGraph (CGG), a spatio-temporal model that pairs the classical features of the EER model with a large set of spatial and temporal constructs [5]. First, we take into consideration each temporal dimension in isolation; then, we will deal with their combination.

Valid time. The valid time of a tuple is the time when the fact it represents is true in the modeled domain. We encode valid time intervals by means of two distinct timestamps *VT_start* and *VT_end*. The extension of an atemporal relation $R(\mathbf{X})$ with valid time has the form:

$$R(\mathbf{X}, VT\_start, VT\_end) \tag{1}$$

Let $\mathbf{T}^g$ be the (discrete) temporal domain at granularity g over which timestamps VT_start, VT_end take their value. Any pair of values t_s for VT_start and t_e for VT_end identifies a time interval $[t_s, t_e) \subset \mathbf{T}^g$ (we assume valid time intervals, as well as transaction and availability time intervals, to be closed to the left and open to the right). Valid time intervals consisting of a single chronon are represented as degenerate intervals with coincident endpoints (notationally, $[t_s, t_s+1)$). While the left endpoint VT_start of a valid time interval must always exist, its right endpoint VT_end might be missing[1]. The intended semantics of valid time intervals is captured by the following constraints:

[1] To represent valid time intervals open to the right, most models assign to VT_end either the "value" `null` or the "value" `until change` (`uc` for short). Since `null` is used in a variety of contexts with different meanings, we opt for the second alternative.

$$\begin{array}{l}(i)\ \exists t_s \in \mathbf{T}^g\ t_s = VT\_start \\ (ii)\ \exists t_e \in \mathbf{T}^g\ t_e = VT\_end \vee VT\_end = \texttt{uc} \\ (iii)\ VT\_start < VT\_end\end{array} \quad (2)$$

Transaction time. The *transaction time* of a tuple is the time when the tuple is current in the database. We represent transaction time intervals by means of two distinct timestamps TT_start and TT_end. A transaction time interval is generated whenever a database update is executed. For every interval associated with a tuple in the database, we have that the value of TT_start is less than the current instant and that of TT_end is either `until change`, if the tuple is current, or less than or equal to the current instant, if the tuple is not current. Since deletion of a tuple can never precede its insertion/modification, TT_start must obviously be less than TT_end. The intended semantics of transaction time intervals is captured by the following constraints:

$$\begin{array}{l}(i)\ \exists t \in \mathbf{T}^g\ t = TT\_start \\ (ii)\ TT\_end \leq \texttt{now} \vee TT\_end = \texttt{uc} \\ (iv)\ TT\_start < TT\_end\end{array} \quad (3)$$

As in the HDBMS model, the schema (resp., instance) of every temporal relation is partitioned into two distinct schemas (resp., instances). The first instance, called *current instance*, consists of all and only the tuples which are current in the database. It only features the timestamp TT_start, whose value records the time instant at which the tuple was added to the database (the value TT_end for all current tuples is equal to `uc`, and thus omitted). The second one, called *historical instance*, records the tuples which have been logically deleted from the database. It features the two timestamps TT_start and TT_end that respectively record the times at which insertion and deletion take place.

$$R(\mathbf{X}, TT\_start) \text{ and } R\_history(\mathbf{X}, TT\_start, TT\_end) \quad (4)$$

Tuples are always inserted in the current instance. The time instant at which insertion is executed is automatically assigned to the timestamp TT_start. The logical deletion of a tuple simply moves the tuple from the current instance to the historical one, without changing the value of its attributes. The time instant at which deletion is executed is automatically assigned to the new timestamp TT_end. The update of a tuple in the current instance can be described as a logical deletion of the current tuple followed by the insertion of the updated one (deletion/insertion times are equal to the time instant at which the update is executed). Tuples in the historical instance cannot be deleted or updated.

There are several advantages in separating the historical schema/instance from the current one. First, TT_end can be omitted in the current schema. Second, transaction time management is fully automatized. Third, since tuples in the historical instance cannot be modified, constraint checking can be restricted to tuples in the current instance (we will come back to this in Section 4). Finally, an improvement in query performance is often achieved. Whenever a query refers to current information only (we expect it to be the most common case), its execution can ignore all tuples in the historical instance.

Availability time. The availability time of a tuple is the time interval during which the fact it represents is known and believed correct by the information system the database belongs to. Availability time intervals are encoded by a pair of timestamps AT_start, AT_end and must satisfy the same constraints that transaction time intervals must satisfy:

$$\begin{array}{ll} (i) & \exists t \in \mathbf{T}^g t = AT\_start \\ (ii) & AT\_end \leq \mathtt{now} \vee AT\_end = \mathtt{uc} \\ (iii) & AT\_start < AT\_end \end{array} \tag{5}$$

Additional constraints are imposed on the relationships between availability and transaction times. First, a fact can be stored in the database only if it is or was known by the information system. Similarly, a fact can be (logically) deleted from the database only if it is not believed correct/up-to-date by the information system. Moreover, if a fact is known and believed correct by the information system and it has been added to the database ($AT\_end = uc$), then the corresponding tuple must belong to the current instance ($TT\_end = uc$); conversely, it can never happen that $AT\_end \neq uc$ and $TT\_end = uc$. Finally, we must consider the case in which the information systems acquires and discharges some fact before its insertion in the database. Such a situation can be modeled by letting $AT\_end \leq TT\_start$ (when inserted in the database, information was already out-of-date) if and only if $TT\_start = TT\_end$ (information never became current in the database).

$$\begin{array}{ll} (i) & AT\_start \leq TT\_start \\ (ii) & AT\_end \leq TT\_end \\ (iii) & AT\_end = uc \Rightarrow TT\_end = uc \\ (iv) & TT\_end = uc \Rightarrow AT\_end = uc \\ (v) & AT\_end \leq TT\_start \Leftrightarrow TT\_start = TT\_end \end{array} \tag{6}$$

Since in any realistic scenario the choice of including availability time and excluding transaction time looks meaningless, we do not consider temporal relation schemas with availability time and without transaction time. In addition, we must find a way to deal with information that never becomes current in the database ($TT\_start = TT\_end$), preserving the condition that imposes to insert any new fact in the current instance and to move it to the historical one when it is logically deleted. To cope with this problem, we include both the AT_start and the AT_end timestamps in the current schema. As a result, we obtain the following schema:

$$\begin{array}{l} \text{R}(\mathbf{X}, \text{TT\_start}, \text{AT\_start}, \text{AT\_end}) \\ \text{R\_history}(\mathbf{X}, \text{TT\_start}, \text{TT\_end}, \text{AT\_start}, \text{AT\_end}) \end{array} \tag{7}$$

We must distinguish two different modalities of tuple insertion. The first one provides a value for AT_start, but no value for AT_end. In this case, the system assigns the specified value to AT_start, it sets TT_start to the current time, and it adds the tuple to the current instance. The second one deals with the case in which a value less than (or equal to) the current time is given to AT_end.

The system assigns to AT_start and AT_end the specified values, it sets both TT_start and TT_end to the current time (thus $TT\_start \geq AT\_end$), it inserts the tuple in the current instance, and it immediately moves it to the historical one. Two different modalities of (logical) tuple deletion must be considered as well, depending on the value of AT_end. The first case is that of synchronous deletion: both AT_end and TT_end are set to the current time and the tuple is automatically moved from the current instance to the historical one. The second case considers a possible delay in the registration of a deletion from the information system: the system replaces the value *uc* of AT_end with the deletion time (which is less than the current time), it sets TT_end to the current time, and it automatically moves the tuple from the current instance to the historical one.

Event time. The *event time* of a tuple consists of the occurrence times of the real-world events that respectively initiate and terminate the valid time interval of the fact it represents. To model it, we add two timestamps ET_start, ET_end to the relation schema. By definition, event time can be added only to relation schemas provided with valid time. No constraints are imposed on event time.

$$R(\mathbf{X}, VT\_start, VT\_end, ET\_start, ET\_end) \tag{8}$$

Relations with multiple temporal dimensions. We conclude the section with an analysis of temporal relations provided with two or more temporal dimensions. As a general rule, we start from an atemporal relational schema and we add the appropriate timestamps for every supported temporal dimension. However, we cannot add available (resp., event) time without adding transaction (resp., valid) time as well. The addition of transaction time forces the partition of the relation schema in a current schema and a historical one. As an example, a temporal schema with the four temporal dimensions can be obtained by an atemporal schema $R(\mathbf{X})$ as follows. First, we add valid time:

$$R(\mathbf{X}, VT\_start, VT\_end) \tag{9}$$

Then, we add event time:

$$R(\mathbf{X}, VT\_start, VT\_end, ET\_start, ET\_end) \tag{10}$$

The addition of transaction time forces the splitting of the table:

$$\begin{array}{l} \text{R (}\mathbf{X}\text{, VT\_start, VT\_end, ET\_start, ET\_end, TT\_start)} \\ \text{R\_history (}\mathbf{X}\text{, VT\_start, VT\_end, ET\_start, ET\_end,TT\_start, TT\_end)} \end{array} \tag{11}$$

Finally, the addition of available time affects both schemas (in a different way):

$$\begin{array}{l} \text{R(}\mathbf{X}\text{, VT\_start, VT\_end, ET\_start, ET\_end, TT\_start, AT\_start, AT\_end)} \\ \text{R\_history (}\mathbf{X}\text{, VT\_start, VT\_end, ET\_start, ET\_end, TT\_start, TT\_end,} \\ \qquad\qquad \text{AT\_start, AT\_end)} \end{array} \tag{12}$$

A well-known problem in temporal databases is to assign a consistent value to missing temporal dimensions, thus providing every relation with a temporal interpretation with respect to all temporal dimensions. In such a way, no relations

are ignored during (temporal) query evaluation. The assignment of a value to missing temporal dimensions is done according to the following rules.

- Transaction time is missing. Tuples are current in the database when they can be retrieved from it, that is, we assume the transaction time interval of tuples to be $[now, now]$).
- Valid time is missing. Valid time is assimilated to transaction time: if transaction time is present, then valid time intervals are equal to transaction time intervals; otherwise, tuples are valid at the time instant in which they are retrieved from the database, that is, we assume the valid time interval of tuples to be $[now, now]$).
- Event time is missing. We assume $ET\_start = VT\_start$ and $ET\_end = VT\_end$ (on-time events).
- Available time is missing. We assume $AT\_start = TT\_start$ and $AT\_end = TT\_end$ (no delay in registration).

Such rules can be turned into suitable projection functions (one for each temporal dimension) that, given a relation instance, return the temporal values it explicitly or implicitly takes on temporal dimensions. We consider transaction and valid times; the cases of event and availability times are similar. Given a (temporal) relation R, let r_c (resp., r_h) be the instance of its current (resp., historical) schema (if transaction time is missing, R has a current schema only).

Definition 1. *Let R be a (temporal) relation. If $TT\_start, TT\_end \in R$, then $\pi_{TT}(r_c) = [\pi_{TT\_start}(r_c), \mathbf{now}]$ and $\pi_{TT}(r_h) = [\pi_{TT\_start}(r_h), \pi_{TT\_end}(r_h)]$. If $TT\_start, TT\_end \notin R$, then $\pi_{TT}(r_c) = [\mathbf{now}, \mathbf{now}]$.*

Definition 2. *Let R be a (temporal) relation. If both $VT\_start, VT\_end \in R$ and $TT\_start, TT\_end \in R$, then $\pi_{VT}(r_i) = [\pi_{VT\_start}(r_i), \pi_{VT\_end}(r_i)]$, for $i \in \{c, h\}$. If $VT\_start, VT\_end \in R$ and $TT\_start, TT\_end \notin R$, then $\pi_{VT}(r_c) = [\pi_{VT\_start}(r_c), \pi_{VT\_end}(r_c)]$. If $VT\_start, VT\_end \notin R$ and $TT\_start, TT\_end \in R$, then $\pi_{VT}(r_c) = \pi_{TT}(r_c)$ and $\pi_{VT}(r_h) = \pi_{TT}(r_h)$. If both VT_start, $VT\_end \notin R$ and $TT\_start, TT\_end \notin R$, then $\pi_{VT}(r_c) = \pi_{TT}(r_c) = [\mathbf{now}, \mathbf{now}]$.*

4 Temporal Primary Keys and Functional Dependencies

In this section we deal with the problem of specifying primary key and functional dependencies of a temporal relation. As it happens in the relational setting, there is a close connection between them; however, the addition of multiple temporal dimensions introduces various technical intricacies.

Both problems have been already addressed in the temporal databases literature, but there are no consensus solutions to them. As for temporal keys, different alternatives have been proposed, which range from the addition of one or more temporal attributes to the primary key of the atemporal schema [8] to the introduction of explicit object identifiers that uniquely identify each tuple in the temporal relation [14]. As for temporal functional dependencies (TFDs), a short account of existing proposals can be found in [4]. The simplest ones

define TFDs as classical functional dependencies on the temporal snapshots of the relation [8], the most complex ones allow TFDs to constrain the values of (atemporal) attributes at different time points [13,14].

Our goal is to guarantee an appropriate trade-off between expressiveness and effectiveness. In particular, we would like to maintain the notion of temporal key and temporal dependency as simple and easy to manage as possible. The solution we propose deals with multiple temporal dimensions retaining much of the simplicity of the relational model. In addition, the separation between current and historical schemas/instances makes it possible to simplify the process of constraint checking.

As a general rule, we define TFDs as temporal generalizations of (atemporal) functional dependencies (FDs), which are obtained by making the latter time dependent. From a notational point of view, we replace every FD $\mathbf{Z} \rightarrow \mathbf{Y}$ by the corresponding TFD $\mathbf{Z} \rightarrow_{\mathbf{T}} \mathbf{Y}$. The role of the four temporal dimensions in TFDs is quite different. By means of TFDs, we constrain FDs to be satisfied by pairs of tuples at common valid time instants with respect to common transaction or availability time instants. As availability (resp., transaction) time intervals may start before (resp., end after) than the corresponding transaction (resp., availability) time intervals, this amounts to require functional dependency to be satisfied with respect to common availability/transaction time instants belonging to a time interval that starts when the availability interval starts and ends when the transaction time interval ends. Event time plays no role in the definition of TFDs.

Definition 3. *Given a temporal relation R with atemporal schema $R(\boldsymbol{X})$ and a TDF $\boldsymbol{Z} \rightarrow_{\boldsymbol{T}} \boldsymbol{Y}$, with $\boldsymbol{Z}, \boldsymbol{Y} \subseteq \boldsymbol{X}$, we say that an instance $r \in R$ satisfies the TFD if and only if, for each pair of tuples $a, b \in r$, if $a[\boldsymbol{Z}] = b[\boldsymbol{Z}]$, $\pi_{VT}(a) \cap \pi_{VT}(b) \neq \emptyset$ (their valid time intervals overlap), and $\pi_{TT}(a) \cap \pi_{TT}(b) \neq \emptyset \vee \pi_{AT}(a) \cap \pi_{AT}(b) \neq \emptyset$ (their transaction or availability time intervals overlap), then $a[\boldsymbol{Y}] = b[\boldsymbol{Y}]$.*

Missing temporal dimensions are implicitly added according to the assignment rules given in Section 3. The notion of violation of a TFD is defined in the obvious way. We say that two tuples are *temporally inconsistent* if they violate a TFD.

Let us consider now the problem of specifying the key of a temporal relation schema (temporal key for short). As anticipated in Section 2, we basically define the (primary) temporal key as a temporal extension of the primary key of the original atemporal relation. We distinguish between the current schema and the historical schema of a temporal relation: the temporal key of the current schema add valid time to the atemporal key, while the temporal key of the historical schema add both valid and transaction times to the atemporal key. If valid time is missing, the temporal key of the current schema coincides with the atemporal one, while that of the historical schema consists of the atemporal key extended with transaction time. The fact that we compactly represent both valid and transaction times by means of interval timestamps, instead of instant ones, introduces some complications. A simple example of these complications is given in Table 1.

Table 1. The current instance of a table *Employee* devoid of transaction time

SSN	Salary	VT_start	VT_end
XXXNNN88HH	1000	15/10/2000	31/07/2006
XXXNNN88HH	1200	01/10/2003	31/07/2007

The two tuples belonging to the relation in Table 1 are temporally inconsistent, because they assign both the value 1000 and the value 1200 to the salary of employee *XXXNNN88HH* over the valid time interval [01/10/2003,31/07/2006). Such an inconsistency can be obviously detected by replacing the interval timestamp (VT_start, VT_end) by the instant one VT, by choosing (SSN, VT) as the temporal key, and by replacing every tuple by a set of tuples, one for each time instant in the valid time interval. However, the resulting instance turns out to be extremely redundant: a single tuple is replaced by a number of tuples that only differ in their temporal value. To avoid to introduce such a redundancy, we decided to maintain the interval timestamp. Unfortunately, in such a case, all possible choices for the attributes of the temporal key, namely, (SSN, VT_start), (SSN, VT_end), and (SSN, VT_start, VT_end), do not detect the inconsistency in Table 1. As a consequence, the satisfaction of the key constraint (for any possible choice of the temporal key) does not suffice to conclude that there are not temporal inconsistencies and thus temporal consistency must be explicitly checked.

Table 2. Temporal keys for temporal relations: a summary

Cases	Temporal keys	Temporal dimensions
atemporal	$R(\underline{K}, ...)$	-
valid time	$R(\underline{K}, \underline{VT\ start}, ...)$	V VE
transaction time	R($\underline{K}$,...) R_history($\underline{K}$,$\underline{TT\ start}$,...)	T TA
valid and transaction times	R($\underline{K}$,$\underline{VT\_start}$,...) R_history($\underline{K}$,$\underline{VT\ start}$,$\underline{TT\ start}$,...)	VT VTE VTA $VTAE$

Among the three possible choices for the temporal key of the current schema, we opt for the addition of VT_start to the atemporal key. It detects more temporal inconsistencies than the temporal key that includes both VT_start and VT_end and, unlike VT_end, VT_start does not assume the "value" `uc`. Analogously, for the historical schema we choose to add to the atemporal key VT_start and TT_start. A summary of the resulting cases is given in Table 2.

In principle, constraint checking can be executed whenever a tuple is inserted in the current database or moved from the current to the historical database (tuples in the historical database cannot change their values). However, when a

tuple is transferred from the current to the historical database, it only changes the value of TT_end and, possibly, the value of AT_end. Such changes cannot cause any inconsistency in the historical database[2]. Hence, constraint checking can be confined to insertions in the current database. When a tuple is inserted in the current database, an inconsistency may arise with respect to both current and historical tuples. In the former case, according to the proposed model, the intersection of both transaction and availability time intervals associated with current tuples is always not empty and thus the only constraint one needs to check on the current database is:

$$\forall a, b \in R(\mathbf{X}) \forall \mathbf{Y} \subseteq \mathbf{X}(a[\mathbf{K}] = b[\mathbf{K}] \wedge \pi_{VT}(a) \cap \pi_{VT}(b) \neq \emptyset \rightarrow a[\mathbf{Y}] = b[\mathbf{Y}]) \quad (13)$$

where $\mathbf{X}$ is the set of atemporal attributes of R (and $R\_history$) and $\mathbf{K}$ is the atemporal key of R (and $R\_history$). In the latter case, the intersection of transaction time intervals associated with the inserted tuple and a historical one is always empty and thus the only constraint one needs to check is:

$$\begin{aligned} &\forall a \in R(\mathbf{X})\ \forall b \in R\_history(\mathbf{X})\ \forall \mathbf{Y} \subseteq \mathbf{X}(a[\mathbf{K}] = b[\mathbf{K}] \wedge \\ &\wedge \pi_{VT}(a) \cap \pi_{VT}(b) \neq \emptyset \wedge \pi_{AT}(a) \cap \pi_{AT}(b) \neq \emptyset \rightarrow a[\mathbf{Y}] = b[\mathbf{Y}]) \end{aligned} \quad (14)$$

This constraint can be violated only if the relation R (and $R\_history$) includes the three dimensions VTA. An inconsistency may occur if and only if the value of AT_start for the inserted tuple a is less than the value of TT_start (if $\pi_{AT}(a) = \pi_{TT}(a)$, then the intersection of the availability time intervals for a and any historical tuple is empty).

5 Implementation

In this section, we briefly describe an implementation of the proposed model in the Oracle DBMS [11]. As for the definition of the relational schemas and of data types, we use the standard SQL facilities featured by Oracle SQL [9]. To deal with timestamps, we take advantage of the `timestamp` data type (the conventional value `uc` is represented by the `null` value). Temporal constraints are encoded either as generic SQL assertions, using the SQL construct `check` constraint (the simplest ones), or as triggers (the most complex ones). As a concrete example of constraint management, we describe the triggers that rule the transition of relation tuples from the current instance to the historical one (for the sake of simplicity, we assume the relations to be devoid of availability time).

When a tuple is inserted in the current instance, the trigger sets TT_start to the value `systimestamp(0)` (the current time of the system):

[2] As a matter of fact, this implies that temporal keys for historical schemas are not really necessary. We decided to keep them to comply with the relation model, but they could be removed without causing any problem.

```
CREATE OR REPLACE TRIGGER nameTable_insertTT
  BEFORE INSERT ON nameTable
  FOR EACH ROW
BEGIN
   SELECT systimestamp(0) INTO :new.TT_start
     FROM dual;
END;
```

The deletion of a tuple from the current instance consists of the assignment of the value `systimestamp(0)` to its *TT_end* and of its insertion in the historical instance:

```
CREATE OR REPLACE TRIGGER nameTable_delete
  BEFORE DELETE ON nameTable
  FOR EACH ROW
DECLARE
   now timestamp;
BEGIN
   SELECT systimestamp(0) INTO now
     FROM dual;
   INSERT INTO nameTable_history (A, TT_end)
     VALUES (:old.A,now);
END;
```

Tuple updates are implemented as a deletion followed by an insertion as usual:

```
CREATE OR REPLACE TRIGGER nameTable_update
  BEFORE UPDATE ON nameTable
  FOR EACH ROW
DECLARE
   now timestamp;
BEGIN
   SELECT systimestamp(0) INTO now
     FROM dual;
   INSERT INTO nameTable_history (A, TT_end)
     VALUES (:old.A, now);
   :new.TT_start := now;
END;
```

Finally, the following trigger disallows the execution of updates or deletions on the historical database (similar triggers have been added to prevent the user to execute other improper actions, e.g., to operate on transaction timestamps):

```
CREATE OR REPLACE TRIGGER nameTable_history_upde
  BEFORE UPDATE OR DELETE ON nameTable_history
  FOR EACH ROW
BEGIN
   raise_application_error(-20001,'Historical tables cannot
      be updated or deleted');
END;
```

As an alternative, one can create one or more user views that specify the privileges of (different classes of) database users, e.g., information in the historical database can be queried, but not updated.

6 Mapping CGG Schemas into the Temporal Model

In [2] we define and implement a translation of CGG schemas into the above-described temporal model. On the one hand, the translation algorithm revises and extends the standard relational encoding of basic ER primitives (entities, relations, specializations,..); on the other hand, it introduces specific rules for the management of spatial and temporal information. Here, we briefly summarize the treatment of CGG temporal features.

The translation introduces a set of relation schemas for every temporal entity and relation in the CGG schema. Such a set consists of a root schema, called kernel, that plays the role of reference schema for all relation schemas generated by a given entity or relation. Each single relation schema is linked to the kernel by means of a suitable foreign key as shown in Figure 1.

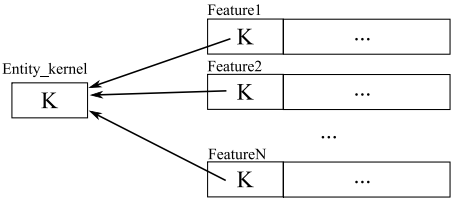

Fig. 1. The relational translation of a temporal entity

Let us consider the case of a temporal entity E with attributes $\mathbf{X} = \{k_1, \ldots, k_n, a_1, \ldots, a_m\}$, whose conceptual key is $\mathbf{K} = \{k_1, \ldots, k_n\}$, with $n \geq 1$ (the case of temporal relations is similar).

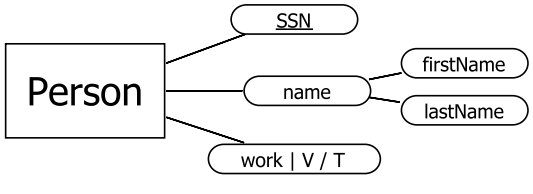

Fig. 2. A CGG entity with different types of attribute

The schema of the kernel consists of the key attributes $k_1, \ldots, k_n$,, that is, $E\_kernel(\underline{k_1, \ldots, k_n})$. It allows one to identify all entity instances. The temporal features of the entity (temporal qualification of the entity, sets of synchronized

temporal attributes, ..) are distributed over different component relations. In addition, each component relation includes key attributes to allow one to merge information about an entity instance. All atemporal (single-valued) attributes $\{a_{i_1}, \ldots, a_{i_l}\}$ are collected in a single component relation $E_{a_{i_1},\ldots,a_{i_l}}(\underline{k_1, \ldots, k_n}, a_{i_1}, \ldots, a_{i_l})$. A distinct component relation is then added for each group of temporal attributes that change their values in a synchronous way. As an example, consider the entity in Figure 2. Its relational translation is as follows:

$Person\_kernel(\underline{SSN})$
$Person\_atemporalNochange(\underline{SSN}, firstName, lastName)$
$Person\_work(\underline{SSN}, \underline{VT\_start}, VT\_end, TT\_start, work)$
$Person\_work\_history(\underline{SSN}, \underline{VT\_start}, VT\_end, \underline{TT\_start}, TT\_end, work)$.

References

1. Ben-Zvi, J.: The time relational model. PhD thesis, University of California, Los Angeles (1982)
2. ChronoGeoGraph (2009), `http://dbms.dimi.uniud.it/cgg/`
3. Combi, C., Montanari, A.: Data models with multiple temporal dimensions: Completing the picture. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 187–202. Springer, Heidelberg (2001)
4. Combi, C., Montanari, A., Rossato, R.: A uniform algebraic characterization of temporal functional dependencies. In: Proc. of the 12th International Symposium on Temporal Representation and Reasoning (TIME), pp. 91–99. IEEE Computer Society Press, Los Alamitos (2005)
5. Gubiani, D., Montanari, A.: ChronoGeoGraph: an expressive spatio-temporal conceptual model. In: Ceci, M., Malerba, D., Tanca, L. (eds.) Proc. of the 15th Italian Symposium on Advanced Database Systems (SEBD), pp. 160–171 (2007)
6. Jensen, C.S., Snodgrass, R.T. (eds.): Temporal Database Entries for the Springer Encyclopedia of Database Systems. Technical Report TIMECENTER TR-90 (2008)
7. Jensen, C.S., Snodgrass, R.T.: Semantics of time-varying attributes and their use for temporal database design. In: Papazoglou, M.P. (ed.) ER 1995 and OOER 1995. LNCS, vol. 1021, pp. 366–377. Springer, Heidelberg (1995)
8. Jensen, C.S., et al.: The consensus glossary of temporal database concepts - February 1998 version. In: Temporal Databases, Dagstuhl (TDB), pp. 367–405 (1997)
9. Lorentz, D., et al.: Oracle Database SQL Reference (2005)
10. Navathe, S.B., Ahmed, R.: Temporal extensions to the relational model and SQL. In: Uz Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R. (eds.) Temporal Databases: theory, design, and implementation, pp. 92–109. The Benjamin/Cummings Publishing Company (1993)
11. Oracle. Oracle 10g (2007), `http://www.oracle.com`
12. Sarda, N.L.: HSQL: A historical query language. In: Uz Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R. (eds.) Temporal Databases: theory, design, and implementation, pp. 110–140. The Benjamin/Cummings Publishing Company (1993)
13. Vianu, V.: Dynamic functional dependency and database aging. Journal of the ACM 34(1), 28–59 (1987)
14. Wijsen, J.: Temporal FDs on complex objects. ACM Transactions on Database Systems 24(1), 127–176 (1999)