Case study 9

Temporal databases - Part 1

Objective. In this study we examine various ways to organize the data describing the evolution of a population of entities. The basic model consists in storing the successive states of each entity, completed by the time period during which the state was observable. We distinguish between the *transaction time*, that refers to the data modification time in the database and the *valid time*, referring to modification events of entities in the real world. This study particularly develops *entity-based*, *attribute-based*, *event-based* and *document-oriented* temporal database models. In these models, data management is ensured by triggers that automate as far as possible entity creation, modification and deletion operations.
The next study will be devoted to temporal database querying and transformation.

Keywords. temporal data type, temporal database, history, entity type, time point, time period, time interval, evolution, event, state, transaction time, valid time, bitemporal data, julian day, document-oriented model, JSON, trigger

9.1 Introduction

A database, as it generally is considered in text books and lectures, informs on the current state of the application domain, that is, that part of the real world it is intended to describe. This is the case of the ORDERS.db database we have used as the support of most chapters of the SQLfast tutorial and of some case studies.

It turns out that things change, sometimes at a fast pace. For instance, new customers are registered, new products are offered, orders are placed, prices are modified, customers move to other cities, orders are cancelled and obsolete products are withdrawn.

These changes are propagated to the database, as soon as possible, in such a way that the data always reflect the current state of the application domain. To know the price of a product, we just consult the database.

The problem with this way of managing data is that we lose the trace of past facts. To compute the total amount of an order we need to know the price of products *at the time the order was placed*, not their current price, which may be higher. We may also need to know the previous address(es) of customers, when they have been registered (and what was their first order), which of them have been removed from our customer base, and when. In others words, we would like to know, not only the current state of the application domain, but also its past states, that is, its *history.*[1]

Keeping the trace of past states is not as obvious as we could think, particularly if we intend to *reason* on these states. Taking a save copy of the database at regular time point as we do for a simple document is not an option. Just adding a date column (e.g., *createdOn*, *lastUpdatedOn*) does not work either. To cope with these new problems in a clean and coherent way, we must add to databases a new dimension: *time*. This is what we call a **temporal database**.

This is the goal of this series of studies to develop some of the most appropriate techniques to represent historical data, to manage them and to exploit them. The domain of *temporal data* (of which historical data are a large subset) is particularly rich and would deserve hundreds of pages to describe and develop most of its aspects. Considering the modest objectives of these case studies, we will content ourselves with developing in a very practical way some of these aspects.

Just to give readers the taste of them and the desire to know more!

9.2 Representation of time

Let us first address the way time will be represented in our case studies. Time representation systems appear to be particularly complex when we consider all their physical, geographical, cultural and historical idiosyncrasies. To keep things simple,

1. We will consider past and current states, but not future states, that may pose specific problems.

and considering the kind of application domains we intend to address, we will adopt this subset of the ISO 8601 standard:[2]

1. *dates*: format YYYY-MM-DD.
   Example: 2019-07-24
2. *time* in day: format HH:MM:SS or HH:MM:SS.mmm, where mmm represents milliseconds[3].
   Examples: 16:32:09 or 16:32:09.085
3. *datetime*: date + time separated by one space or by letter T. Also often named *timestamp*.
   Example: 2019-07-24 16:32:09.085 or 2019-07-24T16:32:09.085

In these formats, we ignore time zones as well as the numerous variations and shorthands allowed by the standard.

A key concept of our model is the ***granularity*** (or *precision*) of the temporal dimension. It is defined by the smallest unit of time we can use to specify an instant. In the examples illustrating the time format here above, the granularities are respectively the *day*, the *second* and the *millisecond*. The appropriate granularity depends on the kind of phenomenon we intend to describe. In business processes, such as those concerned by database ORDERS.db, the *day*, or, at best, the *hour*, should be sufficient. We can imagine that anthropology, astronomy or subatomic physics would require quite different granularities. A **time point** is any instant to which a value (a **timestamp**[4]) is assigned in the current granularity. To simplify the development of temporal databases, we will associate with our application domain a unique granularity.

In some applications, instead of the real time values, we will use an **abstract time** representation through positive integers: 2, 7, 20, 40, etc., as shown in Figure 9.3. This will be useful either to simplify some demonstration (real values as 2017-07-24 16:32:09.085 are quite cumbersome!) or to resolve granularity conflicts, for instance to distinguish and order two events that occur at the same time point.

An **event** is a perceived change that occurs in the application domain or in the database and that is considered worth being documented. The main characteristic of an event, besides the nature of the change, is the time at which it occurred. Later, we will use the term **valid time** when the change occurs in the application domain and **transaction time** when the change applies to the contents of the database. When a change in the application domain is recorded in the database, we could want to include in this recording both the valid and transaction times to the change. The database then comprises **bitemporal** data.

2. See notably https://en.wikipedia.org/wiki/ISO_8601 and https://www.iso.org/iso-8601-date-and-time-format.html
3. Actually, according to the standard, the fractional part can be specified up to seven decimals. However, many DBMS impose a limit of three decimals.
4. This term may raise ambiguity. In the SQL language, timestamp is a temporal data type. Here, it is a value created to identify an instant in a timeline, according to the chosen granularity.

We must also mention the concept of **timeline**, which is the ordered set of all the time points that can be associated to events. In the kind of application domains we will cope with, it is common:

1. to identify the *starting point* of the timeline with a definite time, the events that occurred before this point being ignored,
2. to leave the *ending point* undefined, that is, set to *infinite future*. Since it is not easy to represent infinite in a computer, we will associate with the ending point a definite time, but very far in the future.[5]

In the basic case we will study (database TDB.db), we will choose date 2014-11-18 as the starting point and 9999-12-31 as the ending point.

Finally, three terms that must be (re)defined before starting:

- *period*: a portion of a timeline comprised between two time points. A period is defined by these time points. Can be *closed* (includes its starting and ending points), *open* (excludes its starting and ending points) or *half-open* (one of its extreme points is excluded).
- *interval*: a pair of time points delimiting a series of consecutive time points. An *interval* + a *time point* setting its starting point define a period. In other words, a period is an anchored interval. SQL:1999 defines an *interval* as a directed duration expressed either in *years* + *months* or in *days* + *time*.
- *duration*: exact measure of the length of a period. Expressed in non-ambiguous units such as days or smaller. Also called *time distance* or *span*. Example: 41,567.146 seconds

In practice, this vocabulary is used in a quite ambiguous and confusing way. Among the authors in the database domain, *interval* and *period* are often used interchangeably, sometimes in the same document. In both parts of this case study, we will use, whenever no ambiguity may arise, the terms *period* and *interval* to denote any portion of a timeline. When an interval is associated to a state, we will sometimes allow ourself to identify this state to its interval: for instance, *both states overlap* is a shorthand for *the intervals of these states overlap*.

9.3 The concept of current state

The project we will develop concerns a collection of *entities* that we intend to observe and describe during a definite time period. Entities appear, evolve then disappear. We will illustrate the study by a small application domain made up of two sorts of entities, namely *projects* and *employees*:

5. Some authors suggest to represent the ending point by a *null* value. This is consistent but this will make many queries more complex.

1. *projects* have three attributes: their *title*, their *theme* and their *budget*; their title is unique
2. *employees* have five attributes: their *code* (which is unique), their *name*, their *salary*, the *city* they live in and the *project* they work on.

Currently (we mean *now*), our project comprises two projects and five employees. They can be described by rows in relational tables PROJECT and EMPLOYEE of database **TDB.db** [6] (Figures 9.1 and 9.2). Each entity type is represented by a table.[7] In a table, each row describes one entity and each entity is described by one row. Only *active entities* are documented. If an entity does not exist (yet), it has no descriptive row in the table. Similarly, when an entity disappears, there is no row any more to tell that it has, one day, existed. These tables describe the **current state** of the application domain. The rows of these tables are denotations of existing entities and their current attributes.

```
+-----------+------------------+--------+
| TITLE     | THEME            | BUDGET |
+-----------+------------------+--------+
| AGRO-2000 | Crop improvement | 82000  |
| BIOTECH   | Biotechnology    | 140000 |
+-----------+------------------+--------+
```

Figure 9.1 - The current state of project entities

```
+------+-----------+--------+----------+-----------+
| CODE | NAME      | SALARY | CITY     | PROJECT   |
+------+-----------+--------+----------+-----------+
| A237 | Antoine   | 3800   | Grenoble | AGRO-2000 |
| C45  | Carlier   | 3100   | Lille    | BIOTECH   |
| D107 | Delecourt | 3300   | Grenoble | BIOTECH   |
| G96  | Godin     | 3900   | Genève   | AGRO-2000 |
| M158 | Mercier   | 3000   | Paris    | BIOTECH   |
+------+-----------+--------+----------+-----------+
```

Figure 9.2 - The current state of employee entities

9.4 History of an entity

An entity has a *life*, which is the time period during which it is *active* (i.e., it is *living*). It starts with its birth time and ends with its death time or the current time if it still lives. Before its *life*, the entity does not exist, and after its *life*, it does not exist

6. This database is available in directory databases of the SQLfast distribution. Actually, EMPLOYEE and PROJECT are not base tables but views on the historical data.
7. This does not preclude situations in which entities can be composed of several sub-entities. This is the case of database ORDERS.db, in which an order comprises a header and one or several details. Headers are described in table CUSTORDER and details in table DETAIL. Distributing the data of complex entities among several tables is typical of normalized relational databases, in which redundancies are, as far as possible, eliminated.

any more, though its memory can be maintained. The attributes of an entity are assigned specific values at birth time and will generally change their values over time.

If we intend to document the life of these entities, we will have to record all the events that affect their existence: birth, modification of their attributes and death. These events form the *history* of the entity. Figure 9.3 depicts the history of project BIOTECH. The event times are real dates, but they are denoted by abstract number for reasons that will be discussed later.

We observe that the project started at time 2, then, at time 7, got a new theme and a lower budget, at time 9, still a lower budget, and so on until now, at which point the project still is active.

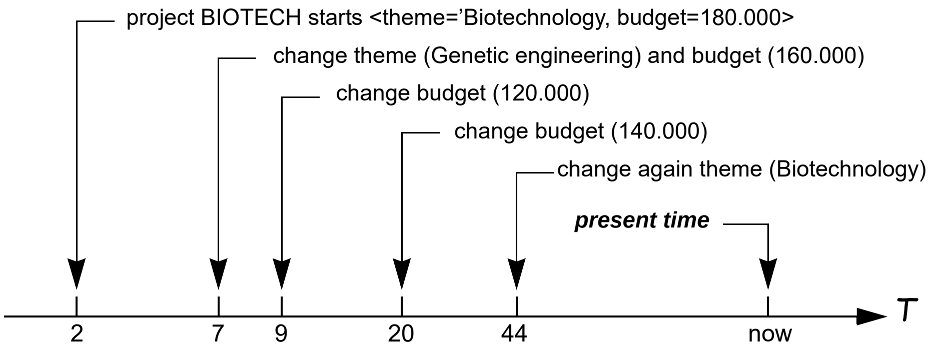

project BIOTECH starts <theme='Biotechnology, budget=180.000>
change theme (Genetic engineering) and budget (160.000)
change budget (120.000)
change budget (140.000)
change again theme (Biotechnology)
present time
2
7
9
20
44
now
T

Figure 9.3 - History of project BIOTECH

now is a time point a bit special: it is constantly moving rightward! It is therefore difficult to assign it an actual temporal value. There are several ways to denote it. For instance through a special symbol such as an empty string or *null* (the SQL marker), or 'until changed' or 'infinite future' or ∞.

For convenience reason that will appear later, we will choose a time point very far in the future, so far that it will never be confused with an actual time point of the application domain.

Times 9999-12-31 or 9999-12-31 23:59:59 or 9999-12-31 23:59:59.999 (in ISO format) depending on the chosen granularity, could be fair choices, at least to model human activities. Abstract time point 999999 could be less far (less than one million events) but still appropriate, at least in our sample database.

Note. The abstract time points must be interpreted with some caution. For example, considering that the time is measured in days, time point 7 does not necessarily denote the date of time point 2 plus 5 days. It just denotes an outstanding date later than the date of time point 2, and that there *may* be four other outstanding date between them, denoted 3, 4, 5 and 6, be they used or not in building the history of other entities. This will become clear with the concept of table CLOCK.

From events to states

Instead of recording these events, we can record the sequence of *states* which each entity goes through. A state is defined by the *time period* during which the values of its properties remain unchanged. When the value of an attribute changes, the current state ends (and from then belongs to the past of the entity) and a new current state starts.

Converted into states, the history of project entities is stored in history table H_PROJECT of database TDB.db. A part of it, shown in Figure 9.4, provides the complete history of project BIOTECH. We have represented each state by a row, in which we show the values of the attributes of the project during this state. The state itself is specified by the starting point (column **start**) and the ending point (column **end**) of its time period.[8] Similarly, the history of employee entities will be stored in history table H_EMPLOYEE.

We observe that the last time point of a period (**end**) is also the first point of the next period. This does not mean that this time point is common to both states. Instead, the **end** point of a state is excluded, so that the time period of a state is a semi-open period, noted **[start,end)**. As we will see later, this convention will simplify many algorithms. Moreover, it remains valid, whatever the granularity of the time measure.

```
+-----------+---------------------+--------+-------+--------+
| TITLE     | THEME               | BUDGET | start | end    |
+-----------+---------------------+--------+-------+--------+
| ...       | ...                 | ...    | ...   | ...    |
| BIOTECH   | Biotechnology       | 180000 | 2     | 7      |
| BIOTECH   | Genetic engineering | 160000 | 7     | 9      |
| BIOTECH   | Genetic engineering | 120000 | 9     | 20     |
| BIOTECH   | Genetic engineering | 140000 | 20    | 44     |
| BIOTECH   | Biotechnology       | 140000 | 44    | 999999 |
| ...       | ...                 | ...    | ...   | ...    |
+-----------+---------------------+--------+-------+--------+
```

Figure 9.4 - Excerpt of table H_PROJECT showing the history of project BIOTECH

For instance, if point 7 denotes date 2016-02-17 and point 9, date 2016-07-24, then the second state of Figure 9.4 is covered by period **[2016-02-17,2016-07-24)**, or, if the time granularity is *one day*, by **[2016-02-17,2016-07-23]**.

8. Names start and end are reserved words in the SQL standard and in some DBMS. If needed, they will be renamed as, for example, Start_time and End_time, or, shorter, Stime and Etime. SQLite is fairly tolerant on this point!

9.4.1 Temporal primary, unique and foreign keys

Keys are standard features of relational databases. *Primary* and *unique keys* support uniqueness constraints while *foreign keys* ensure referential constraints. The *primary key* of a table is a unique key that has been given a special status. In particular, all its components must be declared `not null`. In the mind of the designer, it represents the *essence* of entities[9].

What do they become when we add a temporal dimension to data?

Let us start with the non temporal tables of Figures 9.1 and 9.2, that describe the current states of entities. Let us call them *entity tables* PROJECT and EMPLOYEE. The primary key of PROJECT is column TITLE (that translates project id's) while that of EMPLOYEE is column CODE. Column PROJECT in EMPLOYEE is a foreign key referencing table PROJECT. Its target is the primary key of PROJECT.

Temporal primary keys

Considering that a state is at least one time unit long, the values of **start** of two successive states of an entity are distinct. This suggests that,

- the *temporal primary key* of H_PROJECT is (TITLE,start)
- the *temporal primary key* of H_EMPLOYEE is (CODE,start).

Temporal unique keys

Similarly, each unique key **UK** in entity table **T** gives *temporal unique key* **(UK,start)** in historical table **H_T**.

We could also declare (TITLE,end) and (CODE,end) *unique keys*, but the way historical data will be managed will automatically ensure their uniqueness.

Temporal foreign keys

There is no big risk in declaring (PROJECT,start) a *temporal foreign key* of H_EMPLOYEE referencing H_PROJECT. However, the concept of *referential integrity* is less straightforward than in non temporal tables.

In non temporal database, we must check that:

for each row **e** in EMPLOYEE, there exists a row **p** in PROJECT such that:

• **p**.TITLE = **e**.PROJET

Translated for a temporal database, this property becomes:

for reach row **he** in H_EMPLOYEE, for each time point **t** of **he**, there exists a row **hp** in H_PROJECT such that **hp**.TITLE = **he**.PROJECT and **hp**.start $\leq$ **t** < **hp**.end

9. The *non-nullability* of primary keys is often expressed as *entity integrity.*

More practically,

for each row **he** in H_EMPLOYEE, there exists two rows **hp1**, **hp2** (not necessarily distinct) in H_PROJECT such that (see Figure 9.5):

- **hp1**.TITLE = **hp2**.TITLE = **he**.PROJET
- **hp1**.start ≤ **he**.start et **he**.end ≤ **hp2**.end

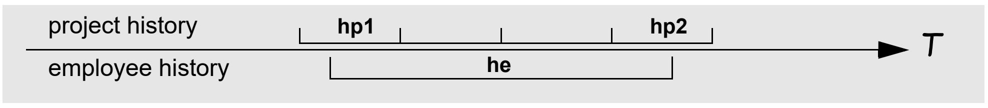


Figure 9.5 - Temporal referential integrity

9.4.2 Normalized history

So, to develop a consistent temporal database, we must enforce some constraints on the *non temporal* schema:

1. Each entity table has a primary key. Its components are **stable**: they cannot be modified. Its components are **not reusable**: the values of the primary key of an entity that has been deleted cannot be assigned to another entity. These properties ensure that all the states of a definite entity can be gathered without any ambiguity.
2. The target of each foreign key is the primary key of the referenced table.
3. The schema of the database does not evolve. Studying schema evolution of historical data is (far) beyond the scope of this studies.

In addition, the history of an entity must satisfy the following properties:

4. There is no gap in its history. This means that each state **s1**, except the last one, is followed by another state **s2** such that **s2.end** = **s2.start**. States **s2** and **s2** are said **consecutive**. A history that includes no gap is called **complete**.
5. Two consecutive states are distinct. They must have at least one different attribute value.
6. The duration of a state is at least one time unit. This means that **start** < **end**.
7. At any time, an entity is in one state only. Two different states may not overlap.

There is another constraint on the events that will create the history:

8. For each entity, the modification events occur *chronologically*. This means that, at any time, the value of **start** of the last state of an entity is higher than those of all its other states. Making it possible to change past states (for in-

stance to correct errors) would require a *bitemporal history*, which would be much more complicated to manage and process.

A history that satisfies these constraints is called **normalized**. Weaker forms of history may be considered and coped with. In a non normalized history,

– gaps may exist,

– consecutive states may be *value-equivalent* (their columns have exactly the same values),

– two states may overlap provided they are value-equivalent.

Such forms usually appear in the result set of some temporal queries. They can be normalized through the reduce and normalize operator. These issues will be studied in the second part of the case study.

The 7th constraint is particularly critical: a history in which some entities have more than one distinct state at some time point is considered **corrupt**. It must be fixed before being processed.

9.5 Transaction and valid times

What kind of event *exactly* do columns **start** and **end** measure? If they specify the time points at which changes occur in the application domain, that is, in the real world, they refer to the **valid time**. If they specify the time at which information on the changes is recorded in the database, they refer to the **transaction time**.

When an employee moves from a city to another one on *September 1st, 2017* (2017-09-01), and if this fact is recorded in the database twenty days later, on 2017-09-21, then 2017-09-01 is the valid time of this change while 2017-09-21 is its transaction time.

Both types of time are important. Let us consider a fact **F** that appears at time **vT** and that has been recorded in the database at time **tT**. If we consider the database as the repository of all the relevant knowledge on the application domain, then we can say that fact **F** is *officially* known from time **tT** and *officially* unknown before. This can have deep implication from the legal point of view. Ideally, we should associate both valid and transaction times to the data, that is, record **bitemporal** data. Unfortunately, the management and processing of bitemporal data is particularly complex, so that we will consider one temporal dimension at a time only.

In the next sections, we will develop simple mechanisms that automate as far as possible the management of temporal data according to each of these time dimensions.

9.6 Managing transaction time historical data

Recording transaction time events enjoys a quite interesting property. Indeed, the transaction time associated with an event typically is provided by the system clock. This means that the user has no control on the values of the timestamps, so that the time management can be fully automated. The user just has to tell: *create this new entity*, *modify these attributes of this existing entity* and *delete this existing entity*. In technical words, the data modification events will be created by issuing `insert`, `update` and `delete` queries on the views describing the current states of the entities. This way, the users can ignore all the time management mechanics.

From the SQL engine perspective, these transaction time values are obtained through register `current_timestamp`, that returns the current system time down to the millisecond. Whether this precision is appropriate depends on the pace of the events. In some cases, it will be far too detailed, for instance in administrative tasks, where a delay of one day generally is the rule. In other cases, this precision will be too coarse, as for a web server that processes several hundreds of requests per second.

Whether the application domain is subject to slow events or to fast events, this choice may entail three important consequences.

1. A high time precision requires, in the ISO format (YYYY-MM-DD HH:MM:SS:.mmm), a string of 23 bytes, that is, 46 additional bytes per state. In a fast evolving system, where an entity may be subject to thousands of modification events, timestamping states will consume a lot of space.[10]

2. Even if we choose the most fine-grained precision, there is no guarantee that two events affecting the same entity will always be assigned different timestamps. In such cases, some modifications of an entity will be recorded with the same timestamp, leading the database to lose some of these updates.

3. To simulate the evolution of systems with slow events, such as administrations, collecting real-time data to build test datasets, as we will do in these studies, would take months and even years, which would be unrealistic.

To solve these problems, we add to historical base tables H_PROJECT and H_EMPLOYEE a new table, called CLOCK (Script 9.1), in which we store the timestamps of events (in column TTime) whenever they occur. A unique integer number is assigned to these time points, acting as time id (column TimeID). Columns **start** and **end** in historical tables will contains such time ids instead of actual timestamps, whatever the granularity of the latter. Column TimeID is the primary key of CLOCK but column TTime is not unique.

10. Actually, this space depends on the internal representation of temporal data. For instance, *datetime* values can be coded as *julian day*, which itself is implemented by a floating point number spanning 8 or 16 bytes. See https://en.wikipedia.org/wiki/Julian_day for more detail. Julian day arithmetics is available as UDF functions in SQLfast.

```
create table CLOCK(
   TimeID integer not null primary key autoincrement,
   TTime  varchar(23) not null);
```

Script 9.1 - Table CLOCK stores the time points of the events

Historical table H_PROJECT and H_EMPLOYEE can now be specified (Script 9.2). Considering the nature of the application domain of this case study, representing time points by dates seems quite sufficient.

```
create table H_PROJECT(
   TITLE     char(20)    not null,
   THEME     char(22)    not null,
   BUDGET    decimal(8)  not null,
   start     integer     not null,
   end       integer     not null default 999999,
   primary key (TITLE,start));

create table H_EMPLOYEE(
   CODE      char(5)   not null,
   NAME      char(10)  not null,
   SALARY    integer   not null,
   CITY      char(10)  not null,
   PROJECT   char(20)  not null,
   start     integer   not null,
   end       integer   not null default 999999,
   primary key (CODE,start));
```

Script 9.2 - Historical tables recording the evolution of project and employee entities

Figures 9.6 and 9.7 show the evolution of project and employee entities. Table Clock that translates abstract time points into actual dates is shown in Figure 9.8. The contents of table H_PROJECT shows an interesting fact: a third project, named SURVEYOR, was started on time 12 but was closed on time 36, so that it does not exist any more and does not appear in the current state of the database (Figure 9.1).

```
+-----------+----------------------+--------+-------+--------+
| TITLE     | THEME                | BUDGET | start | end    |
+-----------+----------------------+--------+-------+--------+
| AGRO-2000 | Crop improvement     | 65000  | 21    | 36     |
| AGRO-2000 | Crop improvement     | 75000  | 36    | 41     |
| AGRO-2000 | Crop improvement     | 82000  | 41    | 999999 |
| BIOTECH   | Biotechnology        | 180000 | 2     | 7      |
| BIOTECH   | Genetic engineering  | 160000 | 7     | 9      |
| BIOTECH   | Genetic engineering  | 120000 | 9     | 20     |
| BIOTECH   | Genetic engineering  | 140000 | 20    | 44     |
| BIOTECH   | Biotechnology        | 140000 | 44    | 999999 |
| SURVEYOR  | Satellite monitoring | 310000 | 12    | 18     |
| SURVEYOR   | Satellite monitoring | 375000 | 18     | 31     |
| SURVEYOR   | Satellite monitoring | 345000 | 31     | 40     |
+------------+----------------------+--------+--------+--------+
```

Figure 9.6 - History of PROJECT entities

```
+------+-----------+--------+----------+-----------+-------+--------+
| CODE | NAME      | SALARY | CITY     | PROJECT   | start | end    |
+------+-----------+--------+----------+-----------+-------+--------+
| A237 | Antoine   | 4000   | Grenoble | AGRO-2000 | 24    | 30     |
| A237 | Antoine   | 3800   | Grenoble | BIOTECH   | 30    | 33     |
| A237 | Antoine   | 3800   | Grenoble | AGRO-2000 | 33    | 999999 |
| A68  | Albert    | 3200   | Toulouse | BIOTECH   | 8     | 13     |
| A68  | Albert    | 3700   | Toulouse | SURVEYOR  | 13    | 26     |
| A68  | Albert    | 3900   | Paris    | SURVEYOR  | 26    | 39     |
| C45  | Carlier   | 3100   | Lille    | BIOTECH   | 11    | 22     |
| C45  | Carlier   | 3400   | Lille    | AGRO-2000 | 22    | 28     |
| C45  | Carlier   | 3100   | Lille    | BIOTECH   | 28    | 999999 |
| D107 | Delecourt | 3800   | Grenoble | SURVEYOR  | 15    | 17     |
| D107 | Delecourt | 4100   | Genève   | SURVEYOR  | 17    | 27     |
| D107 | Delecourt | 4100   | Grenoble | SURVEYOR  | 27    | 35     |
| D107 | Delecourt | 3300   | Grenoble | BIOTECH   | 35    | 999999 |
| D122 | Declercq  | 3200   | Paris    | BIOTECH   | 10    | 16     |
| D122 | Declercq  | 3800   | Paris    | SURVEYOR  | 16    | 37     |
| D122 | Declercq  | 4100   | Toulouse | AGRO-2000 | 37    | 42     |
| G96  | Godin     | 3400   | Genève   | BIOTECH   | 3     | 5      |
| G96  | Godin     | 3300   | Genève   | BIOTECH   | 5     | 23     |
| G96  | Godin     | 3900   | Genève   | AGRO-2000 | 23    | 29     |
| G96  | Godin     | 4100   | Genève   | AGRO-2000 | 29    | 38     |
| G96  | Godin     | 3900   | Genève   | AGRO-2000 | 38    | 999999 |
| M158 | Mercier   | 2900   | Paris    | BIOTECH   | 4     | 6      |
| M158 | Mercier   | 3000   | Paris    | BIOTECH   | 6     | 14     |
| M158 | Mercier   | 3600   | Paris    | SURVEYOR  | 14    | 34     |
| M158 | Mercier   | 3000   | Paris    | BIOTECH   | 34    | 999999 |
| N240 | Nguyen    | 3100   | Toulouse | BIOTECH   | 19    | 25     |
| N240 | Nguyen    | 3700   | Grenoble | SURVEYOR  | 25    | 32     |
| N240 | Nguyen    | 3700   | Genève   | AGRO-2000 | 32    | 43     |
+------+-----------+--------+----------+-----------+-------+--------+
```

Figure 9.7 - History of EMPLOYEE entities

```
+--------+------------+
| TimeID | TTime      |
+--------+------------+
| 1      | 2017-07-24 |
| 2      | 2017-07-25 |
| 3      | 2017-07-27 |
| 4      | 2017-07-29 |
| 5      | 2017-07-30 |
| 6      | 2017-08-02 |
| 7      | 2017-08-06 |
| 8      | 2017-08-09 |
| 9      | 2017-08-11 |
| 10     | 2017-08-12 |
| 11     | 2017-08-14 |
| ...    | ...        |
| 43     | 2017-10-26 |
| 44     | 2017-10-27 |
+--------+------------+
```

Figure 9.8 - Table CLOCK records the time points of all the events and assigns them a unique time Id

9.6.1 Current state of entities

Live entities are those that have an *active current state*, with an *infinite future* end time, that is, according to our convention, with TimeID = 999999. So, they are quite easy to select through the SQL views of Script 9.3. The current states of our example database are shown in Figures 9.9 and 9.10.

```
create view PROJECT(TITLE,THEME,BUDGET)
as select TITLE,THEME,BUDGET
   from   H_PROJECT
   where  end = 999999;

create view EMPLOYEE(CODE,NAME,STATUS,CITY,PROJECT)
as select CODE,NAME,STATUS,CITY,PROJECT
   from   H_EMPLOYEE
   where  end = 999999;
```

Script 9.3 - Extracting the current states

```
+-----------+------------------+--------+
| TITLE     | THEME            | BUDGET |
+-----------+------------------+--------+
| AGRO-2000 | Crop improvement | 82000  |
| BIOTECH   | Biotechnology    | 140000 |
+-----------+------------------+--------+
```

Figure 9.9 - The current state of PROJECT entities

```
+------+-----------+--------+----------+-----------+
| CODE | NAME      | SALARY | CITY     | PROJECT   |
+------+-----------+--------+----------+-----------+
| A237 | Antoine   | 3800   | Grenoble | AGRO-2000 |
| C45  | Carlier   | 3100   | Lille    | BIOTECH   |
| D107 | Delecourt | 3300   | Grenoble | BIOTECH   |
| G96  | Godin     | 3900   | Genève   | AGRO-2000 |
| M158 | Mercier   | 3000   | Paris    | BIOTECH   |
+------+-----------+--------+----------+-----------+
```

Figure 9.10 - The current state of EMPLOYEE entities

The main interest of these views is that users will be able to manage data, that is, *create*, *modify* and *destroy* entities, by merely executing `insert`, `update` and `delete` SQL queries **on these views**, and not on the underlying historical tables.

9.6.2 Creating an entity

The nice property of transaction time management is that all data modification actions can be performed on the current state of entities without worrying about complicated temporal manipulations. Therefore, creating a new entity, cannot be simpler:

```
insert into PROJECT(TITLE,THEME,BUDGET)
   values ('BIOTECH','Biotechnology',180000);

insert into EMPLOYEE(CODE,NAME,STATUS,CITY,PROJECT)
   values ('G96','Godin','F','Genève','BIOTECH');
```

The effect of such a query is to insert in table H_PROJECT (same for H_EMPLOYEE) the following row:

```
'BIOTECH','Biotechnology',180000,2,999999)
```

Value start = 2 is the TimeID value returned by the insertion of the current time in table CLOCK and end=999999 tells that this state is active.

Now, we have to consider what must happen *under the hood*. The implementation of `insert` operations will be coded in `instead of` triggers operating on views PROJECT and EMPLOYEE.

Let us begin with the creation of ***project** entities*, that will be recorded by trigger `TRG_PRO_INSERT`. Its body must execute three actions.

1. Checking that there is no known entity with the same entity Id.

   We verify that there is no state, past or current, in H_PROJECT with the same value of TITLE as the one we intend to insert, otherwise, we raise an exception that cancels the insert operation:

   ```
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   ```

2. Creating a time event and storing it in table CLOCK.

   Theoretically, we would just store in table CLOCK the value of register `current_timestamp` or `current_date`:

   ```
   insert into CLOCK(TTime) values(current_date);
   ```

   This would be appropriate in a real temporal database, with real events captured in real time, but certainly not in our modest case study! So, we will generate artificial, but realistic time points. For instance through the following query:

   ```
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5))from CLOCK;
   ```

   Function `random_i(n1,n2)` returns a random integer in interval [**n1**,**n2**] and function `addToDate(d,n)` returns date **d** augmented by **n** days. This query computes the most recent value of TTime in CLOCK, adds to it a random number of days then inserts the result in CLOCK.[11]

3. Inserting data in H_PROJECT with the time ID just created.

   The **end** column is automatically set to 999999 through it `default` clause.

11. Of course, table CLOCK must be initialized with something like:
`insert into CLOCK(TTime) values ('2017-07-24');`.

```
insert into H_PROJECT(TITLE,THEME,BUDGET,start)
values (new.TITLE,new.THEME,new.BUDGET,
        (select max(TimeID) from CLOCK));
```

We are now ready to code this trigger (Script 9.4).

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
      values(new.TITLE,new.THEME,new.BUDGET,
             (select max(TimeID) from CLOCK));
end;
```

Script 9.4 - Trigger controlling the **creation** of a new **project** (transaction time)

Managing the creation of employee entities is similar to that of projects, with an additional concern: controlling foreign key PROJECT. Indeed, the value of column PROJECT of the new row must reference an active PROJECT entity. Hence the statement:

```
if not exists(select * from H_PROJECT
              where  TITLE = new.PROJECT
              and    end = 999999)
   raise_exception('target PROJECT does not exist');
```

The complete trigger is shown in Script 9.5.

Two comments

1. Our case study does not include *unique keys*. Their control can be less strict than that of primary key. Verifying that there is no active entity with the same value of the unique key may be sufficient:

```
if exists(select * from H_PROJECT
          where THEME = new.THEME and end = 999999)
raise_exception('duplicate unique key');
```

2. The way triggers are coded is strongly dependent on the DBMS. For example, the body of an SQLite trigger is a pure sequence of SQL queries. The exception itself is specified through function `raise()` in a `select` query:

```
select raise(ABORT,'duplicate entity PK')
where  exists(select * from H_PROJECT
              where  TITLE = new.TITLE);
```

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_EMPLOYEE where CODE = new.CODE)
      raise_exception('duplicate Entity PK');
   if not exists(select * from H_PROJECT
                 where  TITLE = new.PROJECT
                 and    end = 999999)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
      values(new.CODE,new.NAME,new.SALARY,new.CITY,
             new.PROJECT,(select max(TimeID) from CLOCK));
end;
```

Script 9.5 - Trigger controlling the **creation** of a new **employee** (transaction time)

9.6.3 Evolution of an entity

To modify the state of an entity, a simple, non temporal `update` query will suffice:

```
update PROJECT
set THEME = 'Genetic engineering',
    BUDGET = 160000
where TITLE = 'BIOTECH';

update EMPLOYEE
set PROJECT = 'AGRO-2000',
    SALARY = 3900
where CODE = 'G96';
```

Let us suppose that

1. the *update* of project BIOTECH occurs at a time point with TimeID = 7
2. the current state of this project, just before the modification, is the following:

   `('BIOTECH','Biotechnology',180000,2,999999)`

The translation of the modification requires two actions. First we close the current state by modifying its **`end`** column:

`('BIOTECH','Biotechnology',180000,2,7)`

Then we insert a new current state which starts at time point 7:

`('BIOTECH','Genetic engineering',160000,7,999999)`

Now, the history of project BIOTECH comprises two states. The implementation of `update` operations will be coded in `instead of` triggers acting on views PROJECT and EMPLOYEE.

The trigger controlling the evolution of projects, named `TRG_PRO_UPDATE`, must perform five actions.

1. Verifying that the entity is active.

   No action is required: only active states are visible through view PROJECT. Since no modification is applied to a *non-existent entity*, no exception can be fired in this case.

2. Verifying that at least one attribute is modified.

   Quite straightforward:

   ```
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   ```

3. Creating a time event and storing it in table CLOCK.

   Same as for `insert`.

4. Close the current state

   The Id of the time point just created is extracted and assigned to column **end**:

   ```
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
   ```

5. Insert a new current state.

   Same as for `insert`.

   ```
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)

   values(old.TITLE,new.THEME,new.BUDGET,
          (select max(TimeID) from CLOCK));
   ```

   We notice that the value of column TITLE (the primary key) is left unchanged (old.TITLE), whatever the new value possibly provided by the user.

The complete trigger is given in Script 9.6.

The trigger controlling the evolution of employees is similar to that of projects but it must cope with a new constraint: is a new project is specified for the employee, this project must exist and be active:

```
if new.PROJECT <> old.PROJECT
and not exists(select * from H_PROJECT
               where  TITLE = new.PROJECT
               and    end = 999999)
raise_exception('target PROJECT does not exist');
```

The trigger for EMPLOYEE is shown in Script 9.7.

```
create trigger TRG_PRO_UPDATE instead of update on PROJECT
begin
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
      values(old.TITLE,new.THEME,new.BUDGET,
             (select max(TimeID) from CLOCK));
end;
```

Script 9.6 - Trigger controlling the **modification** of a **project** (transaction time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
begin
   if     new.NAME = old.NAME and new.SALARY  = old.SALARY
      and new.CITY = old.CITY and new.PROJECT = old.PROJECT
      raise_exception('no attribute modified');
   if new.PROJECT <> old.PROJECT
      and not exists(select * from H_PROJECT
                     where  TITLE = new.PROJECT
                     and    end = 999999)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_EMPLOYEE
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
      values(old.CODE,new.NAME,new.SALARY,new.CITY,
             new.PROJECT,(select max(TimeID) from CLOCK));
end;
```

Script 9.7 - Trigger controlling the **modification** of an **employee** (transaction time)

9.6.4 Deleting an entity

Deleting an entity follows the standard technique, once again applied to view PROJECT and EMPLOYEE:

```
delete from PROJECT where TITLE = 'SURVEYOR';

delete from EMPLOYEE where CODE = 'D122';
```

Let us suppose that the current state of project SURVEYOR is the following:

```
('SURVEYOR','Satellite monitoring',345000,29,999999)
```

To delete this project (actually to *close* it) at time point 36, we just modify its **end** column to give it value 36:

```
('SURVEYOR','Satellite monitoring',345000,29,36)
```

However, the foreign key of EMPLOYEE dictates specific constraints on delete operations on project entities. The current state of PROJECT can be closed only if the referential constraint will be validated at operation completion. If some employees still are working on the project we intend to delete, we must first take care of them: either by deleting them as well or by preventing the project to be deleted, according to the *delete mode* declared for this foreign key. If we choose the `no action` delete mode, then we must first check that there is no employee working of project SURVEYOR any more:

```
if exists(select * from H_EMPLOYEE
          where  PROJECT = old.TITLE and end = 999999)
   raise_exception('dependent EMPLOYEEs still exist');
```

The code of the trigger that deletes PROJECT entities is shown in Script 9.8. The trigger for EMPLOYEE entities is similar (not shown).

It is interesting to notice that the last state of a deleted entity provides two informations: one about the last update of the entity (or on its creation) and the other one about its deletion.

```
create trigger TRG_PRO_DELETE instead of delete on PROJECT
begin
   if exists(select * from H_EMPLOYEE
             where  PROJECT = old.TITLE
             and    end = 999999)
      raise_exception('dependent EMPLOYEEs still exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
end;
```

Script 9.8 - Trigger controlling the **deletion** of a **project** (transaction time)

9.7 Managing valid time historical data

The term *valid time* refers to the time as it flows in the application domain, that is, in the *real world*.

Let us study the behavior of temporal data that describe what happens in the real world. We can keep the same data structures as in Section 9.6 to store temporal data except for the nature of columns **start** and **end** which are of type **`date`** (or `datetime` or `timestamp`, according to the time granularity) instead of **`integer`**. Now, the values of **start** and **end** are those provided by users. They will need additional checking and control.

To make the process manageable, we will impose a constraint on the temporal sequence of the events affecting each entity:[12]

> the state changes of an entity are recorded in **chronological order** of their occurrence. Indeed, notifying a change that occurred before the last recorded event will require a potentially complex reorganization of the history of the entity in which some closed states could have to be deleted, split or merged.

9.7.1 Current state of entities

We still use constant 9999-12-31 to indicate the *infinite future* that will be assigned to the **end** column to indicate *current states*. The SQL views declared in Script 9.9 extract the description of the current project and employee entities[13]. The latter are shown in Figures 9.11 and 9.12. Columns **start** and **end** now are assigned user data and therefore are visible.

```
create view PROJECT(TITLE,THEME,BUDGET,start,end)
as select TITLE,THEME,BUDGET,start,end
   from   H_PROJECT
   where  end = '9999-12-31';

create view EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start,end)
as select CODE,NAME,SALARY,CITY,PROJECT,start,end
   from   H_EMPLOYEE
   where  end = '9999-12-31';
```

Script 9.9 - Extracting the current states

12. This constraint can be (partially) removed in other forms of historical data, for instance those based on individual attribute history. Anyway, correctly coping with this scenarios requires a bitemporal database.
13. These views **cannot** be declared `with check option`! Why?

```
+-----------+-----------------+--------+------------+------------+
| TITLE     | THEME           | BUDGET | start      | end        |
+-----------+-----------------+--------+------------+------------+
| AGRO-2000 | Crop improvement | 82000  | 2017-07-05 | 9999-12-31 |
| BIOTECH   | Biotechnology   | 140000 | 2017-09-17 | 9999-12-31 |
+-----------+-----------------+--------+------------+------------+
```

Figure 9.11 - The current state of PROJECT entities (valid time)

```
+------+-----------+--------+----------+-----------+------------+------------+
| CODE | NAME      | SALARY | CITY     | PROJECT   | start      | end        |
+------+-----------+--------+----------+-----------+------------+------------+
| A237 | Antoine   | 3800   | Grenoble | AGRO-2000 | 2016-12-29 | 9999-12-31 |
| C45  | Carlier   | 3100   | Lille    | BIOTECH   | 2016-12-11 | 9999-12-31 |
| D107 | Delecourt | 3300   | Grenoble | BIOTECH   | 2017-04-03 | 9999-12-31 |
| G96  | Godin     | 3900   | Genève   | AGRO-2000 | 2017-05-19 | 9999-12-31 |
| M158 | Mercier   | 3000   | Paris    | BIOTECH   | 2017-01-14 | 9999-12-31 |
+------+-----------+--------+----------+-----------+------------+------------+
```

Figure 9.12 - The current state of EMPLOYEE entities (valid time)

9.7.2 Creating of an entity

To create an entity, we just execute an **insert** query on the PROJECT or EMPLOYEE view:

```
insert into PROJECT(TITLE,THEME,BUDGET,start)
   values ('BIOTECH','Biotechnology',180000,'2014-11-18');
insert into EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values ('G96','Godin',3400,'Genève','BIOTECH','2014-11-23');
```

The effect of the first query is to insert in table H_PROJECT the following row, in which value end = 9999-12-31 tells that this state is current:

```
   ('BIOTECH','Biotechnology',180000,'2014-11-18','9999-12-31')
```

The *project* entity we intend to create must have a primary key value that has never been used before. Since the event date is provided by the user, we must check its validity, through function `isDate(d)`, that indicates whether value **d** is a valid date. The trigger that controls the creation of PROJECT entities is shown in Script 9.10.

The creation of an EMPLOYEE entity needs an additional checking: the PROJECT entity referenced by the value of column PROJECT must exist and be active. The trigger associated with the creation of PROJECT entities is shown in Script 9.11.

9.7.3 Evolution of an entity

The state of an entity is modified through an update query on views PROJECT or EMPLOYEE:

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   if not isDate(new.start)
      raise_exception('invalid start date');
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
   values (new.TITLE,new.THEME,new.BUDGET,new.start);
end;
```

Script 9.10 - Trigger controlling the **creation** of a new **project** (valid time)

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_EMPLOYEE where CODE = new.CODE)
      raise_exception('duplicate entity PK');
   if isNotDate(new.start)
      raise_exception('invalid start date');
   if not exists(select * from H_PROJECT
                 where  TITLE = new.PROJECT
                 and    end = '9999-12-31');
      raise_exception('target PROJECT does not exist');
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values (new.CODE,new.NAME,new.SALARY,
           new.CITY,new.PROJECT,new.start);
end;
```

Script 9.11 - Trigger controlling the **creation** of a new **employee** (valid time)

```
update PROJECT
set start = '2015-02-27',
    THEME = 'Genetic engineering', BUDGET = 160000
where TITLE = 'BIOTECH';

update EMPLOYEE
set start = '2017-05-19', SALARY = 3900
where CODE = 'G96';
```

The execution of these queries requires two actions. First closing the current state by modifying its **end** column:

```
('BIOTECH','Biotechnology',180000,
 '2014-11-18','2015-02-27')
```

then inserting a new current state which starts at time point `2015-02-27`:

```
    ('BIOTECH','Genetic engineering',160000,
     '2015-02-27','9999-12-31')
```

Checking the value of column **start** is a bit more complex than that required in `insert` queries. Not only this data must be syntactically correct (function **`isDate`**) but the *chronological order* must be satisfied. This means that the new value of **start** must be greater than that of the current state of the entity. Technically:

```
     if not isDate(new.start)
        or new.start <= (select start from H_PROJECT
                         where  TITLE = new.TITLE
                         and    end = '9999-12-31')
        raise_exception('invalid start date');
```

Controlling *referential integrity* when updating an employee also is a bit more tricky than for transaction time. Of course, the target project must be active but this is not sufficient: this project must also have existed at the **start** time of the updated employee. For example, if the employee is claimed to work on project AGRO-2000 since 2016-08-01, this update must be rejected because this project started on 2016-08-31 only.

```
     if new.PROJECT <> old.PROJECT
        and not (exists(select * from H_PROJECT
                        where  TITLE = new.PROJECT
                        and    end = '9999-12-31')
                 and new.start >= (select min(start)
                                   from   H_PROJECT
                                   where  TITLE = new.PROJECT))
        raise_exception('target PROJECT does not exist');
```

As in the transaction time triggers, the value of the entity id (PROJECT.TITLE and EMPLOYEE.CODE) must be preserved, even if the user tries to modify it.

Suggested triggers controlling the evolution of projects and employees are shown in Scripts 9.12 and 9.13. The role of the `when` clause will be explained in the next section.

```
create trigger TRG_PRO_UPDATE instead of update on PROJECT
when new.end = '9999-12-31'
begin
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   if isNotDate(new.start)
      or new.start <= (select start from H_PROJECT
                       where  TITLE = new.TITLE
                       and    end = '9999-12-31')
      raise_exception('invalid start date');            ./..
   update H_PROJECT
   set    end = new.start
   where  TITLE = old.TITLE and end = '9999-12-31';
   insert into H_PROJECT(TITLE,start,THEME,BUDGET)
   values (old.TITLE,new.start,new.THEME,new.BUDGET);
end;
```

Script 9.12 - Trigger controlling the **modification** of a **project** (valid time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
when new.end = '9999-12-31'
begin
   if new.NAME = old.NAME and new.SALARY = old.SALARY
      and new.CITY = old.CITY and new.PROJECT = old.PROJECT
      raise_exception('no attribute modified');
   if isNotDate(new.start)
      or new.start <= (select start from H_EMPLOYEE
                        where  CODE = new.CODE
                        and    end = '9999-12-31')
      raise_exception('invalid start date');
   if new.PROJECT <> old.PROJECT
      and not (exists(select * from H_PROJECT
                       where  TITLE = new.PROJECT
                       and    end = '9999-12-31')
               and new.start >= (select min(start)
                                  from   H_PROJECT
                                  where  TITLE = new.PROJECT))
      raise_exception('target PROJECT does not exist');
   update H_EMPLOYEE
   set    end = new.start
   where  CODE = old.CODE and end = '9999-12-31';
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values (old.CODE,new.NAME,new.SALARY,new.CITY,new.PROJECT,
           new.start);
end;
```

Script 9.13 - Trigger controlling the **modification** of an **employee** (valid time)

9.7.4 Deleting an entity

Deleting an entity does not follow the technique developed for transaction time. Indeed, the query must convey a user data, namely the deletion date, with which the current state will be closed. Therefore a `delete` query cannot work:

```
delete PROJECT where TITLE = 'SURVEYOR';
```

Instead, we will use a special version of the `update` query, that only changes the value of the **end** attribute of the entity:

```
update PROJECT set end = '2017-05-19' where TITLE = 'SURVEYOR';
```

Let us suppose that the last current state of project SURVEYOR is the following:

```
('SURVEYOR','Satellite monitoring',345000,
 '2016-12-20','9999-12-31')
```

If this project was closed on May 19, 2017, then its new last state will be:

```
('SURVEYOR','Satellite monitoring',345000,
 '2016-12-20','2017-05-19')
```

Like in project updating, we must check the validity of the `end` date. In addition, we verify than no active employee is still working on this project any more. All this is implemented in trigger `TRG_PRO_DELETE` (Script 9.14).

```
create trigger TRG_PRO_DELETE instead of update on PROJECT
when new.end < '9999-12-31'
begin
   if isNotDate(new.end)
      or new.end <= (select start from H_PROJECT
                     where  TITLE = new.TITLE
                     and    end = '9999-12-31')
      raise_exception('invalid end date');
   if exists(select * from H_EMPLOYEE
             where  PROJECT = old.TITLE
             and    end = '9999-12-31')
      raise_exception('dependent EMPLOYEEs still exist');
   update H_PROJECT
   set    end = new.end
   where  TITLE = new.TITLE and end = '9999-12-31';
end;
```

Script 9.14 - Trigger controlling the **deletion** of a **project** (valid time)

9.8 From entity-based to tuple-based history

The discussion of entity-based temporal data can be generalized to more abstract patterns in which the data describe the history of a series of time-varying columns, whatever the meaning one assigns to this assembly.

For example, a row may represent a relationship between two (or more) entities that held during a specified period. This row would comprise the primary keys of the entities, possibly some additional data associated with this relationship, and an interval. Let us suppose that an employee can at any time work on more than one project. The schema of our example database must then be modified: column PROJECT of H_EMPLOYEE is removed and replaced by a new table, called H_WORK_ON, with columns (CODE,TITLE,start,end) and primary key (CODE, TITLE,start).

As another example, we could extract from table H_EMPLOYEE columns (PROJECT,CITY,start,end) that tells from which cities the employees who worked on which project came and during which periods.[14]

To cope with such various interpretation of temporal data, we qualify such temporal table *tuple-based*.[15]

9.9 Alternative temporal database models

The data organization of temporal databases that we have developed in this study (*entity-based history*) is both intuitive and easy to manage. As we will show in the next part of this study, it is also (fairly) easy to process. However, there are quite a lot of alternative ways to organize historical data in a database. We will mention and briefly discuss some of them.

9.9.1 Table horizontal splitting

If processing the current states of the entities is the highest priority (anyway, it is the most frequent use of a database), then it can be more efficient to split each historical table into a main table (called, say, EMPLOYEE) that contains all the current states and another, secondary table (called H_EMPLOYEE) in which all past states are stored.[16]

This way, the main table will be managed and processed in a much more efficient way: smaller table, shorter scans, smaller and faster indexes.

14. This extraction will be discussed in the second part of this study under the name *temporal projection*.
15. A *tuple* is a data structure formed by a series of **t** values (kind of generalization of cou***ple***, tri***ple***, etc.) *Tuple* is the theoretical name of *row* in a table.
16. This is the way several DBMS implement transaction time historical data.

This distribution of current and past states in two different tables may make temporal processing more complex. A straightforward solution consists in recording current states both in the historical table and in the current state table.

9.9.2 Table vertical splitting

In many historical tables, data management maintains the history of some columns, called *time-varying* columns, while for the others, called *constant columns*, only the last state is recorded, because they are never updated or because there is no need to keep their history.

In an entity-based history table, the values of constant columns of an entity are duplicated in all the state rows of this entity. To reduce this waste of space, we can collect all the constant columns, included the components of the primary key, in a table while the other columns (together with the components of the primary key) are gathered into a standard temporal table.

9.9.3 Attribute-based history

The *Entity-based* organization has a major drawback: when one attribute changes, all the values of the other attributes are copied in the new current state, therefore creating important redundancies. In the *attribute-based* organization, the history of each attribute is stored in an independent historical table. When an attribute changes, only its historical table is affected. This model can be studied as the *Column-oriented data model*, discussed in study *Schema-less databases - Part 1*, to which a temporal dimension is added.

The history of the employee population is implemented into four tables (as many as there are attributes, primary key excluded), two of which are represented in Figure 9.13.

```
+------+--------+-------+--------+ +------+----------+-------+--------+
| CODE | SALARY | start | end    | | CODE | CITY     | start | end    |
+------+--------+-------+--------+ +------+----------+-------+--------+
| A237 | 4000   | 24    | 30     | | A237 | Grenoble | 24    | 999999 |
| A237 | 3800   | 30    | 999999 | | A68  | Toulouse | 8     | 26     |
| A68  | 3200   | 8     | 13     | | A68  | Paris    | 26    | 39     |
| A68  | 3700   | 13    | 26     | | C45  | Lille    | 11    | 999999 |
| A68  | 3900   | 26    | 39     | | D107 | Grenoble | 15    | 17     |
| C45  | 3100   | 11    | 22     | | D107 | Genève   | 17    | 27     |
| C45  | 3400   | 22    | 28     | | D107 | Grenoble | 27    | 999999 |
| C45  | 3100   | 28    | 999999 | | D122 | Paris    | 10    | 37     |
| D107 | 3800   | 15    | 17     | | D122 | Toulouse | 37    | 42     |
| D107 | 4100   | 17    | 35     | | G96  | Genève   | 3     | 999999 |
| D107 | 3300   | 35    | 999999 | | M158 | Paris    | 4     | 999999 |
| D122 | 3200   | 10    | 16     | | N240 | Toulouse | 19    | 25     |
| D122 | 3800   | 16    | 37     | | N240 | Grenoble | 25    | 32     |
| D122 | 4100   | 37    | 42     | | N240 | Genève   | 32    | 43     |
| G96  | 3400   | 3     | 5      | +------+----------+-------+--------+
| G96  | 3300   | 5     | 23     |
| G96  | 3900   | 23    | 29     |
| G96  | 4100   | 29    | 38     |
| G96  | 3900   | 38    | 999999 |
| M158 | 2900   | 4     | 6      |
| M158 | 3000   | 6     | 14     |
| M158 | 3600   | 14    | 34     |
| M158 | 3000   | 34    | 999999 |
| N240 | 3100   | 19    | 25     |
| N240 | 3700   | 25    | 43     |
+------+--------+-------+--------+
```

Figure 9.13 - Independent historical tables H_SALARY and H_CITY store the evolution of two attributes of employees.

The triggers that control the creation and deletion of entities are fairly straightforward. Scripts 9.15, 9.16, 9.17 manage the historical attribute tables according to the transaction time dimension. When the condition of execution are met, the operation is distributed among all the historical tables. Adaptation to valid time is as easy.

We notice that, in the trigger controlling the `update` operation (Script 9.16), only attribute tables corresponding to a change are affected.

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_NAME where CODE = new.CODE)
      raise_exception('duplicate entity PK');
   if not exists(select * from H_THEME
                 where  TITLE = new.PROJECT
                 and    end = $future$)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   insert into H_STATUS(CODE,SALARY,start) values
      (new.CODE,new.SALARY,(select max(TimeID) from CLOCK));
   insert into H_CITY(CODE,CITY,start) values
      (new.CODE,new.CITY,(select max(TimeID) from CLOCK));
   ...
end;
```

Script 9.15 - Inserting the data of a new **employee** in an attribute-based historical database (transaction time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
when new.NAME <> old.NAME or new.SALARY  <> old.SALARY
  or new.CITY <> old.CITY or new.PROJECT <> old.PROJECT
                                                      ./...
```

```
begin
   if new.PROJECT <> old.PROJECT
      and not exists(select * from H_THEME
                     where  TITLE = new.PROJECT
                     and    end = $future$)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   if new.SALARY <> old.SALARY begin
      update H_SALARY
      set    end = (select max(TimeID) from CLOCK)
      where  CODE = old.CODE and end = 999999;
      insert into H_SALARY(CODE,SALARY,start) value
         (old.CODE,new.SALARY,(select max(TimeID) from CLOCK));
   end;
   if new.CITY <> old.CITY begin
      update H_CITY
      set    end = (select max(TimeID) from CLOCK)
      where  CODE = old.CODE and end = 999999;
      insert into H_CITY(CODE,CITY,start) values
         (old.CODE,new.CITY,(select max(TimeID) from CLOCK));
   end;
   ...
end;
```

Script 9.16 - Updating the data of an **employee** in an attribute-based historical database (transaction time)

```
create trigger TRG_EMP_DELETE
instead of delete on EMPLOYEE
begin
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   update H_SALARY
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   update H_CITY
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   ...
end;
```

Script 9.17 - Deleting the data of an **employee** in an attribute-based historical database (transaction time)

Alternative trigger organization

In each of these triggers, all the operations on historical tables are gathered into a single piece of code. A quite different organization could be thought of, in which a specific *update trigger* is associates with each historical table (Script 9.18). The change events are filtered with a `when` clause so that only the triggers of the tables in which a change is required will fire. Of course, we must first get a new timestamp from table CLOCK. This can be done with an additional trigger that just includes statement `insert into CLOCK(TTime)` and that fires when at least one change is observed.

Unfortunately, though this distributed organization may appear quite elegant, its execution protocol may be less obvious. Indeed, we have created N + 1 triggers, where N is the number of attributes that may change, and these triggers are of the same type: same table (view EMPLOYEE), same event (`update`), same position (`instead of`).

First, not all DBMS allow multiple triggers of the same kind. Secondly, the trigger updating table CLOCK must fire **before** all the others. This means that the DBMS must provide some way to specify in which order triggers of the same type will fire. Most DBMS provide it, though often in non standard ways,[17] except SQLite, in which this order is arbitrary or undetermined.

Converting an *entity-based* history table to the *attribute-based* format is fairly easy but requires a special operator that will be studied in the second part on the case study: the *temporal projection*. The inverse conversion will be performed through a *temporal join*, an operator that also will be described in the next part.

```
create trigger TRG_SALARY_UPDATE instead of insert on EMPLOYEE
when new.SALARY <> old.SALARY
begin
   update H_STATUS
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   insert into H_SALARY(CODE,SALARY,start) values
      (old.CODE,new.STATUS,(select max(TimeID) from CLOCK));
end;
```

Script 9.18 - The trigger controlling changes of attribute **Status** of **employee** entities, in a distributed trigger organization

17. In some DBMS, the order is explicitly specified in the DDL code of the trigger (Oracle [from 11g], InterBase and, to some extent, SQL Server) while in others, the order is that of trigger creation (SQL standard, DB2, MySQL) or the alphabetical order of trigger names (PostgreSQL).

9.9.4 Event-based history

If we look at Figure 9.14 (copy of Figure 9.3), in which all the modification events affecting project BIOTECH are positioned on the timeline, we could wonder why we have not merely recorded these events instead of the successive states of the project. This is what we develop in this section.

Figure 9.15 shows how events are described in what we could call the *event-based history* of project BIOTECH. For each event, we record the entity primary key (TITLE), the values of other attributes as they result from the operation, the time point of the event and the nature of the operation. This fragment of history shows that project BIOTECH has been created in time 2 then updated three times. We know that the project still is active because no delete operation has been recorded yet.

The event-based history of all the projects of our study is shown in Figure 9.16.

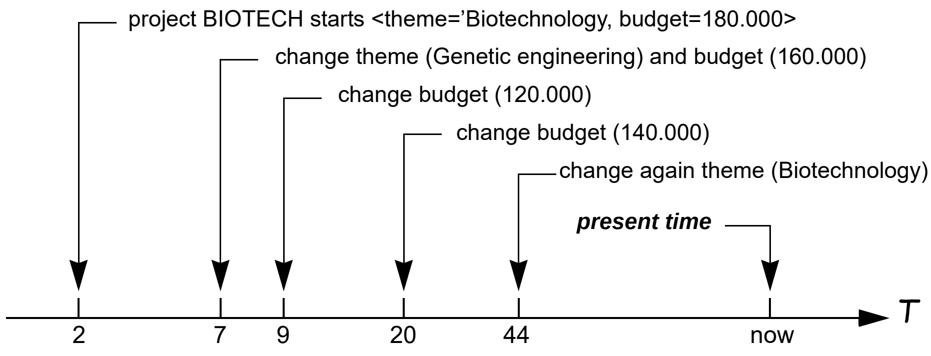

project BIOTECH starts <theme='Biotechnology, budget=180.000>
change theme (Genetic engineering) and budget (160.000)
change budget (120.000)
change budget (140.000)
change again theme (Biotechnology)
present time
2
7
9
20
44
now
T

Figure 9.14 - History of project BIOTECH (copy of Figure 9.3)

```
| BIOTECH   | Biotechnology        | 180000 | 2    | create |
| BIOTECH   | Genetic engineering  | 160000 | 7    | update |
| BIOTECH   | Genetic engineering  | 120000 | 9    | update |
| BIOTECH   | Genetic engineering  | 140000 | 20   | update |
| BIOTECH   | Biotechnology        | 140000 | 44   | update |
```

Figure 9.15 - The event-based history of project BIOTECH

```
+-----------+----------------------+--------+------+--------+
| TITLE     | THEME                | BUDGET | time | oper   |
+-----------+----------------------+--------+------+--------+
| AGRO-2000 | Crop improvement     | 65000  | 21   | create |
| AGRO-2000 | Crop improvement     | 75000  | 36   | update |
| AGRO-2000 | Crop improvement     | 82000  | 41   | update |
| BIOTECH   | Biotechnology        | 180000 | 2    | create |
| BIOTECH   | Genetic engineering  | 160000 | 7    | update |
| BIOTECH   | Genetic engineering  | 120000 | 9    | update |
| BIOTECH   | Genetic engineering  | 140000 | 20   | update |
| BIOTECH   | Biotechnology        | 140000 | 44   | update |
| SURVEYOR  | Satellite monitoring | 310000 | 12   | create |
| SURVEYOR  | Satellite monitoring | 375000 | 18   | update |
| SURVEYOR   | Satellite monitoring | 345000 | 31   | update |
| SURVEYOR   | Satellite monitoring | 345000 | 40   | delete |
+-----------+---------------------+--------+------+--------+
```

Figure 9.16 - The event-based history of all the projects

The event-based history of an entity type is controlled by three triggers as usual. The trigger of Script 9.19 controls the creation of project entities with transaction time dimension.

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_PROJECT(TITLE,THEME,BUDGET,time,oper) values
      (new.TITLE,new.THEME,new.BUDGET,
       (select max(TimeID) from CLOCK),'create');
end;
```

Script 9.19 - Controlling the creation of an entity in the **event-based** history model (transaction time)

9.9.5 Document-oriented history

This organization is derived from the *object model* (also called *document-oriented model*) described in study *Schema-less databases - Part 3*. It can be applied to the *entity-based* and *attribute-based* models. The idea is to associate with each attribute (except those forming the primary key) the list of values it has taken, each of them completed by the list of the intervals during which it took this value.

Let us consider the example of the employee entity with CODE = `C45`. From its creation (on **11**) until time **22** (excluded), it was assigned salary **`3100`**. On time **22**, this salary was changed to **`3400`**. This value then changed to **`3100`** on **`28`** and is still valid now (**999999**). So, the value of SALARY was **`3100`** in intervals `[11,22]` and `[28,999999]` and **`3400`** in interval `[22,28]`. If we encode these facts in JSON, we create a description that looks like this:

```
[{"value":3100,"intervals":[[11,22],[28,999999]]},
 {"value":3400,"intervals":[[22,28]]}]
```

This description is an array of JSON objects, each of them comprising two attributes, namely "**value**" and "**intervals**". Attribute "**value**" specifies the salary value while attribute "**intervals**" is an array of intervals. Each interval is an array of two time points. Figure 9.17 shows the state of all the temporal attributes of employee entity **`D107`** from its creation until now.

```
CODE:    D107
NAME:    [{"value":"Delecourt", "intervals":[[15,999999]]}]
SALARY: [{"value":"3800",       "intervals":[[15,17]]},
         {"value":"4100",       "intervals":[[17,35]]}]
         {"value":"3300",       "intervals":[[35,999999]]}]
CITY:    [{"value":"Grenoble",  "intervals":[[15,17],[27,999999]]},
          {"value":"Genève",    "intervals":[[17,27]]}]
PROJECT:[{"value":"SURVEYOR",   "intervals":[[15,35]]},
          {"value":"BIOTECH",   "intervals":[[35,999999]]}]
```

Figure 9.17 - Temporal attributes of employee entity D107 represented by JSON objects

Figure 9.18 is a representation of the history of employee entities according to the document-oriented model. Each row of this table contains the complete history of an employee entity. To make data more readable, we have replaced the fairly cumbersome JSON syntax by a lighter, though still intuitive, format. In addition, infinite future `'999999'` has been replaced by `'--'`.

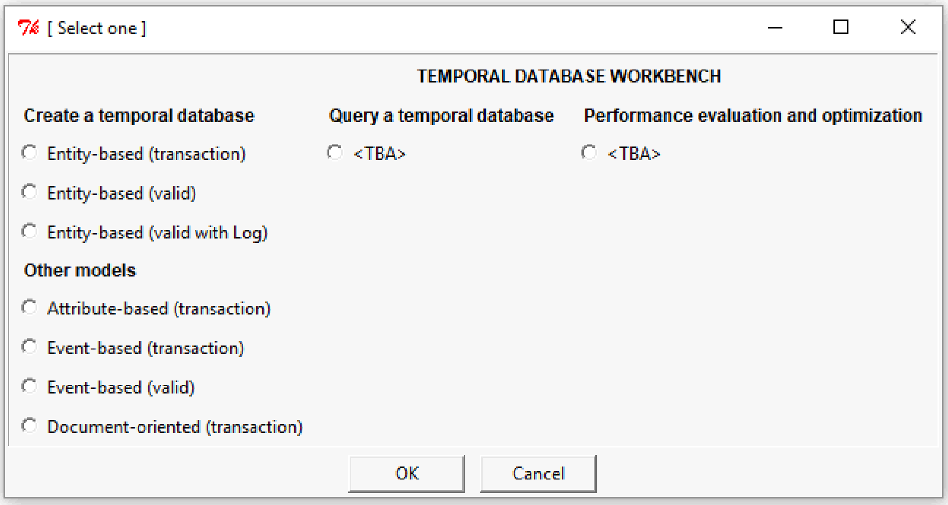
```
+------+-----------+-------------+-----------------+------------------+
| CODE | NAME      | Salaries    | Cities          | Projects         |
+------+-----------+-------------+-----------------+------------------+
| A237 | Antoine   | 4000 [24,30] | Grenoble [24,--] | AGRO-2000 [24,30] |
|      |           | 3800 [30,--] |                 |           [33,--] |
|      |           |             |                 | BIOTECH   [30,33] |
| A68  | Albert    | 3200 [8,13]  | Toulouse [8,26]  | BIOTECH   [8,13]  |
|      |           | 3700 [13,26] | Paris    [26,39] | SURVEYOR  [13,39] |
|      |           | 3900 [26,39] |                 |                  |
| C45  | Carlier   | 3100 [11,22] | Lille    [11,--] | BIOTECH   [11,22] |
|      |           |      [28,--] |                 |           [28,--] |
|      |           | 3400 [22,28] |                 | AGRO-2000 [22,28] |
| D107 | Delecourt | 3800 [15,17] | Grenoble [15,17] | SURVEYOR  [15,35] |
|      |           | 4100 [17,35] |          [27,--] | BIOTECH   [35,--] |
|      |           | 3300 [35,--] | Genève   [17,27] |                  |
| D122 | Declercq  | 3200 [10,16] | Paris    [10,37] | BIOTECH   [10,16] |
|      |           | 3800 [16,37] | Toulouse [37,42] | SURVEYOR  [16,37] |
|      |           | 4100 [37,42] |                 | AGRO-2000 [37,42] |
| G96  | Godin     | 3400 [3,5]   | Genève   [5,--]  | BIOTECH   [3,23]  |
|      |           | 3300 [5,22]  |                 | AGRO-2000 [23,--] |
|      |           | 3900 [22,29] |                 |                  |
|      |           |      [38,--] |                 |                  |
|      |           | 4100 [29,38] |                 |                  |
| M158 | Mercier   | 2900 [4,6]   | Paris    [4,--]  | BIOTECH   [4,14]  |
|      |           | 3000 [6,14]  |                 |           [34,--] |
|      |           |      [34,--] |                 | SURVEYOR  [14,34] |
|      |           | 3600 [14,34] |                 |                  |
| N240 | Nguyen    | 3100 [19,25] | Toulouse [19,25] | BIOTECH   [19,25] |
|      |           | 3700 [25,43] | Grenoble [25,32] | SURVEYOR  [25,32] |
|      |           |             | Genève   [32,43] | AGRO-2000 [32,43] |
+------+-----------+-------------+-----------------+------------------+
```

Figure 9.18 - Document-oriented history of employee entities (transaction time)

The document-oriented format of historical data can be derived from the entity-based and attribute-based data, and conversely. The conversion algorithms will be developed in the second part of this case study.

9.10 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case_Temporal_DB. Actually, they can be run from main script **TDB-MAIN.sql**, that displays the selection box of Figure 9.19 (the missing items will be added in the second part of this study).


[Select one]

TEMPORAL DATABASE WORKBENCH

Create a temporal database	Query a temporal database	Performance evaluation and optimization
○ Entity-based (transaction)	○ <TBA>	○ <TBA>
○ Entity-based (valid)		
○ Entity-based (valid with Log)		
Other models		
○ Attribute-based (transaction)		
○ Event-based (transaction)		
○ Event-based (valid)		
○ Document-oriented (transaction)		

Figure 9.19 - Selecting a temporal data model

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

9.11 Some references

The scientific community has been very active on temporal databases since the eighties and has produced thousands of contributions related to concepts, models and query languages. Richard Snodgrass and Christopher Jensen have been among the most prolific contributors, notably through the TSQL2 language, an SQL extension addressing the temporal dimension of data. Some of the concepts of this language have been progressively (and very slowly!) incorporated into the SQL standards. Nowadays, most major DBMS have extended their SQL offering with temporal features inspired by TSQL2.

A collection of *must have* references are available on the site of Richard Snodgrass at the University of Arizona (*https://www2.cs.arizona.edu/~rts/publications.html*).

- Jensen C. S. and Snodgrass R. T. *Temporal Data Management*, IEEE Trans. on Knowledge and Data Engineering, Vol. 11, No 1, Jan. 1999, 9 pages.

  A short survey of temporal database concepts and design. Free pdf version available.
- Snodgrass, R., T. *Developing Time-Oriented Database Applications in SQL*, 2000, Morgan-Kaufmann, 528 pages.

  An in-depth analysis of the most essential concepts and algorithms in temporal database. Free pdf version available. More on this reference in the second part.
- Jensen C. S. and Snodgrass R. T. (editors). *Temporal Database Entries for the Springer Encyclopedia of Database Systems*. 345 pages.

  The collection of the entries of the *Springer Encyclopedia of Database Systems.* related to temporal databases. Free pdf version available.
- See also webpage http://www.cs.arizona.edu/people/rts/sql3.html, entitled *TSQL2 and SQL3 Interactions*, examine how the proposals of temporal TSQL2 language are being integrated in SQL standards and in some popular RDBMS.

A different model of temporal databases has been proposed by H. Darwen:

- Date C. J., Darwen H. and Lorentzos, N. L. *Time and Relational Theory: Temporal Databases in the Relational Model and SQL*. Morgan-Kaufmann, 2014. 558 pages.