ISSN 0361-7688, Programming and Computer Software, 2008, Vol. 34, No. 1, pp. 1–6. © Pleiades Publishing, Ltd., 2008.
Original Russian Text © B.A. Novikov, E.A. Gorshkova, 2008, published in Programmirovanie, 2008, Vol. 34, No. 1.

Temporal Databases: From Theory to Applications

B. A. Novikov and E. A. Gorshkova
St. Petersburg State University, Universitetskii pr. 28, St. Petersburg, 198504 Russia
E-mail: borisnov@acm.org

Received September 3, 2007

Abstract—Problems concerning the date and time representation in databases are well studied in scientific literature. Temporal databases are required in many applications including the conventional applications of DBMSs such as financial systems. However, commercial systems and standards for the query language do not support temporal data features. A simple and efficient approach for implementing temporal capabilities over conventional commercial DBMSs is proposed and various aspects of its practical applications are described.

DOI: 10.1134/S0361768808010015

1. INTRODUCTION

The necessity of a special treatment of date and time attributes in databases was first understood in the beginning of the 1970s [1]. The databases that support such a treatment are called temporal databases.

It is well known that fast data retrieval and update (that is, the possibility of the on-line access to the up-to-date information) is one of the most important properties of database management systems. However, it is also very important for many applications to access information pertaining to the past or future.

For example, in a system that stores only the up-to-date information, it is generally impossible to repeatedly obtain reports. Obtaining a report as of the first of a certain month requires taking special measures because the state of the database may change at any time and affect the contents of the report based on the current state of the database.

The most intensive studies of temporal databases had been performed during 20 years beginning from the late 1970s. The complete bibliography of temporal databases studies numbered several thousands of entries in 1995 (see [2]); therefore, it seems impossible to compose a representative compact survey.

As a result of long-term studies, all the most important aspects of the development and application of temporal databases were elaborated including semantic problems, data models, integrity constraints, organization of storage structures, indexing, and evolution of schemata [3–13]. The more recent book [14] treats the same range of problems but uses a somewhat different terminology.

Although the theoretical aspects are well understood, none of the widely used commercial DBMSs supports temporal capabilities. The attempt to include the temporal facilities in the SQL standard also failed.

The absence of temporal facilities in DBMSs results in the fact that the functional capabilities required to support temporal data are developed and implemented individually for each application. As a rule, such solutions are incomplete and have serious drawbacks, which include the following ones.

• The use of integrity constraints is complicated and their efficiency is low.

• The logic of the execution of queries containing joins is obscure for developers; as a result, some database management functionality is implemented in applications.

• As a result of the absence of established design patterns, inconsistent implementations appear even within the same application.

• The same functional capabilities are repeatedly implemented.

Note that almost all studies concerning temporal databases assume that the means supporting temporal data are implemented in the DBMS. Such an approach ensures that all the required capabilities including the query language are supported; however, the full-scale implementation of temporal features at the DBMS level is very costly and no ready-made solutions are available.

In this paper, we describe a partial implementation of temporal capabilities in the framework of widely used commercial DBMSs. As we cannot significantly change the well-established procedures of designing and developing information systems, we propose a tool that can be used in the framework of the available technologies. The main goals of this project can be formulated as follows.

• Provide means for using temporal properties in the framework of the conventional relational or object-relational DBMSs.

• Restrict oneself to minor modifications of the well-established methods of designing applications and databases.

• The implementation should not degrade the performance of the parts of the system that do not use temporal data.

• No impediments to using conventional means for integrity control must be incurred.

• The cost of the implementation must be low.

Let us recall some concepts of the theory of temporal databases that we need in this paper.

The semantics of time in temporal databases may be different. Most often, the *system* (or *transaction*) time is considered, which reflects the time when the database was updated; and the *natural* or *valid* time, which indicates the actual time when the facts registered in the database were valid. As a database can support various time semantics, the concept of time *dimensions* supported by the database is used. The systems that support both the transaction and the valid time are called *bitemporal*.

The support of the valid time when it differs from the transaction time cannot be implemented exclusively at the DBMS level because information about the valid time must be provided by applications when the data are modified.

The accuracy of time representation is bounded; therefore, it is usually assumed in the theory of temporal databases that the time is discrete. Under this assumption, one may assume that a temporal database is a set of database states at each moment in time. Such a representation, which is called *snapshot representation*, cannot be used for actually storing data due to its redundancy.

The *interval* representation and the *event-based representation* are more practical. In the interval representation, each state of the data object is assigned a time interval during which this state was valid. In the event-based representation, the data value, the time when this value was changed, and the event that triggered this change (that is, the executed operation) are stored.

All the representations are equivalent; that is, there are algorithms that transform one representation to the other. The choice of the representation depends on the intended use of data (this is because the efficiency of query execution depends on the representation).

An important design decision concerning the structure of data storage is the choice of granularity for the interval or event-based representation. For example, in the framework of the relational model, one can have an interval representation at the level of tables, records, or particular attribute values.

2. BASIC IMPLEMENTATION PRINCIPLES

2.1. Requirements for the Historical Data

The storage of historical data is a system rather than subject-oriented requirement. This requirement can be important for applications in various domains. The modules implementing the main business logic of the system can be designed and developed independently of the temporal aspects of the application. In essence, the support of the history of changes belongs to the basic functionality of the system along with the support of transaction integrity and authorized data access.

In this paper, we use the interval temporal data representation. Consider an arbitrary entity for which the history of changes must be stored. Along with the current state of this entity (the ordinary state of a business object), there is also a representation of this entity in time. This is an abstraction that can be used to learn, in addition to ordinary properties, the interval of time when this representation was valid. On the object level, the relationship between the valid and temporal representations generally depends on the implementation. For example, inheritance, support of certain interfaces, aspect-oriented programming, or an object naming convention can be used for this purpose. Here, it is only important that the past representation is somehow associated with the objects of the system.

The support of temporal features does not affect the analysis and design of the system's business logic and does not require any specific development procedure. The design procedure for the applications that require historical data storage must include the following steps:

• designing the business logic of the system;

• adding the requirements for keeping history of changes for some business entities.

Therefore, the design procedure should enable one to specify the entities and relations for which the history of changes must be stored. For example, UML stereotypes can be used for that purpose.

It is certainly incorrect to assume that the support of historical data does not require changes in the application itself. The application must be able to deal with the historical data, which requires the development of special interfaces for data access and graphical interfaces for presenting the historical data to the end users. The orthogonality of the temporal aspects to the business logic of the application only means that the conceptual data model can be designed independently of the requirement for the support of the history of changes. The data model that contains the valid state of the system's entities will be called the *basic* model, and the data model that adds the properties concerning the history of changes to the basic model will be called the *temporal* model. In this paper, we give a detailed description of the representation of the temporal model in a relational database.

The method proposed for the support of historical data does not require additional coding. The tables that store the history of changes and the triggers that update the data in those tables can be generated automatically from the schema of the basic model.

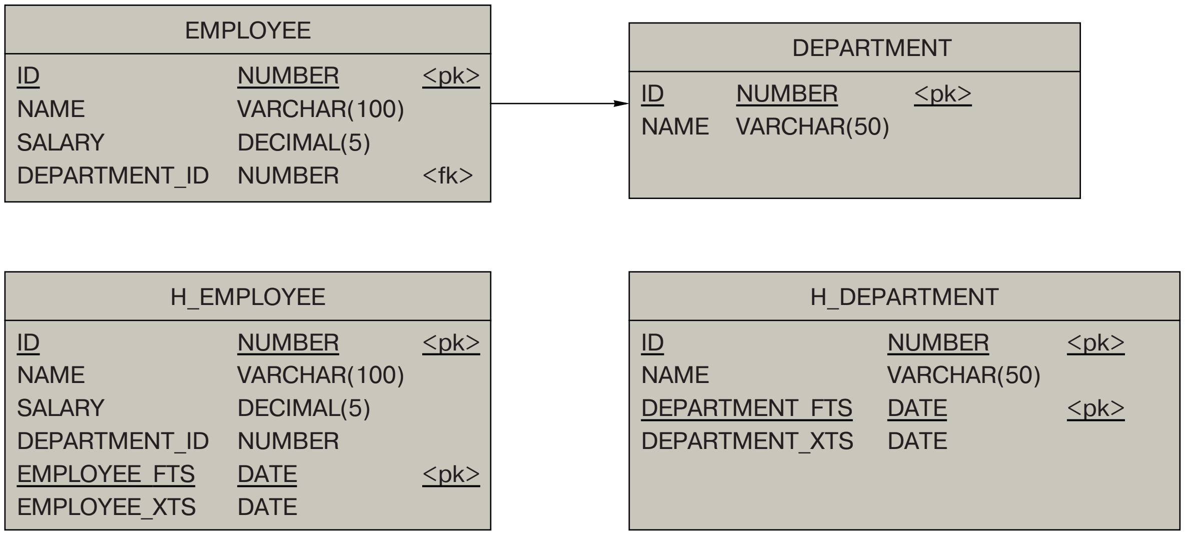

Fig. 1. An example of representation

2.2. Representation of Historical Data

To create the temporal schema, we define the entities and relations for which the history of changes must be stored. In the basic schema, these entities and relations are associated with relational tables. For each such table, we create an additional table with the same name as in the basic schema with the prefix *H_*. For example, for the table *EMPLOYEE*, we create the table *H_EMPLOYEE*. The new table contains the same columns as the basic table and two additional columns—the beginning and the end of the time interval within which the values in the table row were valid. The names of the additional columns are formed by the name of the basic table postfixed by *_FTS* (eFfective Time Stamp) and *_XTS* (eXpire Time Stamp), respectively.

Each H-table has a primary key and a foreign key. The primary key consists of the primary key of the basic table and the *FTS* column. For the table *H_EMPLOYEE*, the primary key is

```
primary key (ID, EMPLOYEE_FTS)
```

The primary key of the basic table serves as the foreign key in the H-table:

```
foreign key(ID) references EMPLOYEE(ID)
```

The data in the H-tables are updated automatically; therefore, no integrity constraints are included in the temporal schema.

2.3. Updating the History of Changes

Consider the insertion and update of records in the H-tables. The main rule is that the application should not directly update the data in those tables; the H-tables are updated automatically. The automatic update can be implemented using triggers or as the functionality of the application's framework. Here are the rules of data updating.

• When a record is inserted in a table of the basic schema, the same data are inserted in the corresponding H-table of the temporal schema. The value of the *FTS* field is set to the current date and time, and the value of the *XTS* field is set to a distant future time, for example, December 31, 4999. It is important that this date (it will be called *END_DATE*) is always greater than the current date for the lifespan of the application. Thus, the valid data are always available in the H-Table, and the *XTS* field of valid records is *END_DATE*.

• When a record is updated in a table of the basic schema, the corresponding record in the H-table (with the same value of the primary key and the *XTS* field equal to *END_DATE*) is also updated. This record stops to be valid, and the *XTS* field is set to the current time. A new record is inserted into the H-table with current values of the fields, the *FTS* field is set to the current time, and the XTS field is set to *END_DATE*.

• When a record is deleted from a table of the basic schema, the corresponding record in the H-table (that has *END_DATE* in the *XTS* field) is updated by setting the *XTS* field equal to the current time.

The H-tables keep the entire history of data change. Each record in an H-table was valid during the time interval [*FTS*, *XTS*]. The history of changes is continuous, the value in the *FTS* field coincides with the value of the *XTS* field of the preceding record with the same primary key. An example of updating data in the table *EMPLOYEE* is shown in Fig. 2.

The primary key of an H-table is composed of the primary key of the basic table and the field *FTS*. As the time is discrete, the scheme described above cannot guarantee that the values of the primary key are unique. This problem is solved by an additional check. If another record with the current value of *FTS* already exists, the least significant bit of the *FTS* in the new record is incremented.

EMPLOYEE

ID	NAME	SALARY	DEPARTMENT_ID
100	Ann	2500	1
101	Joe	2200	2

H_EMPLOYEE

ID	NAME	SALARY	DEPARTMENT_ID	EMPLOYEE_FTS	EMPLOYEE_XTS
100	Ann	2300	2	10/10/2006	01/12/2006
100	Ann	2300	1	01/12/2006	01/03/2007
100	Ann	2500	1	01/03/2007	END_DATE
101	Joe	2200	2	10/10/2006	END_DATE

Fig. 2. An example of history updating.

Although all the data in the basic schema are also available from the temporal schema, the basic schema is useful for the following reasons.

• Integrity constraints are not defined in the temporal schema, and they cannot be verified by the DBMS.

• The tables in the temporal schema can be much larger than the tables in the basic schema.

• For some queries, joins involving data from the temporal schema are less efficient than joins of the tables in the basic schema.

The implementation proposed above also assumes other types of redundancy; however, the degree of redundancy is chosen so as to facilitate efficient query execution.

It is clear that, in order to execute queries efficiently, the database should be fine-tuned; in particular, a set of indexes on the temporal tables should be chosen. However, those issues are beyond the scope of this paper.

We stress that various integrity constraints can be imposed on the basic schema that stores the current values of the data, which ensures the creation of highly reliable databases.

3. QUERY EXECUTION

3.1. Queries Involving Temporal Conditions

Queries on the data in the basic schema provide current values of the data. The support of temporal data makes it possible to obtain data that were valid during a certain time interval in the past, the history of changes of entities, execute queries involving time predicates, and generate various reports. For example, we can find out how much the salary changed when a certain employee was the manager or what was the average salary during a certain time interval.

In this paper, we consider the following types of queries.

• *Snapshot queries* make it possible to obtain the state of entities and their relations at a certain time in the past. For example, one can find out what was the salary of an employee on December 31, 2000.

• *Tracking log queries* retrieve the history of entity changes. For example, such a query can retrieve the history of employees' salary changes during the last five years.

Let us discuss how the representation of historical data proposed above can be used to formulate queries in SQL.

3.2. Snapshot Queries

Snapshot queries are, in essence, ordinary queries with an additional parameter—the time for which the retrieved data were valid. To transform an ordinary query into a snapshot one, the following actions should be performed.

(1) Replace the references to the basic tables by the references to the corresponding H-tables.

(2) For each H-table involved in the query, add the additional condition *snapshot_date >= FTS and snapshot_date < XTS* to the *WHERE* clause.

Then, the records that were valid at the specified time will be selected from each H-table. For example, the query that selects the current salary of an employee is

```
SELECT E.SALARY
FROM EMPLOYEE E
WHERE E.ID = 100
```

This query can be transformed into a query that selects the salary of that employee at a specified time:

```
SELECT E.SALARY
FROM H_EMPLOYEE E
WHERE E.ID = 100
   AND EMPLOYEE_FTS >= :snapshot_date
   AND EMPLOYEE_XTS < :snapshot_date
```

Since the time intervals for the same employee do not overlap, both queries return exactly one record. Note that the valid data are duplicated in the H-tables; therefore, the last query can also return the valid data (for example, when *snapshot_date* = *current_date*).

3.3. Tracking Log Queries

Consider the queries that retrieve the history of entity changes. The complexity of such queries depends on the number of tables involved because the intervals in which the selected records were valid must be matched for different tables.

To create a tracking log query, we need two functions with a variable number of parameters. One of them returns the maximal argument and the other one returns the minimal argument. In the queries, we will use the following expressions:

• *maximal*($table_1\_fts$, …, $table_n\_fts$) selects the maximal argument from the list of arguments. We will pass to this function the *_FTS* values of all the H-tables involved in the query. For brevity, the value returned by this function will be denoted by *MAX_FTS*.

• *minimal*($table_1\_xts$, …, $table_N\_xts$) selects the minimal argument from the list of arguments. We will pass to this function the *_XTS* values of all the H-tables involved in the query. For brevity, the value returned by this function will be denoted by *MIN_XTS*.

To transform a query into a tracking log query, the following actions should be performed.

(1) Replace the references to the basic tables by the references to the corresponding H-tables.

(2) Add the expressions *MAX_FTS* and *MIN_XTS* to the list of selected fields.

(3) Add the additional condition *MAX_FTS* < *MIN_XTS* to the *WHERE* clause.

(4) As may be required, bound the interval for which the history of changes is retrieved. To this end, add the condition *MIN_XTS* > *S AND MAX_FTS* < *E*, where *S* and *E* are, respectively, the beginning and the end of the time interval, to the *WHERE* clause.

Consider the query that selects information about an employee and the department he is with:

```
SELECT E.*, D.*
FROM EMPLOYEE E,
     DEPARTMENT D
WHERE E.DEPARTMENT_ID = D.ID
  AND E.ID = 100
```

With time, the employee's salary, department, and even the name can change. The tracking log query that selects the same data for a certain time interval is as follows.

```
SELECT E.*, D.*,
       MAXIMAL(EMPLOYEE_FTS,
               DEPARTMENT_FTS) AS MAX_FTS,
       MINIMAL(EMPLOYEE_XTS,
               DEPARTMENT_XTS) AS MIN_FTS,
FROM H_EMPLOYEE E,
     H_DEPARTMENT D
WHERE E.DEPARTMENT_ID = D.ID
  AND E.ID = 100
  AND MAX_FTS < MIN_XTS
  AND MAX_FTS < :E
  AND MIN_XTS > :S
```

4. CONCLUSIONS

The implementation of the interval representation of temporal data described in this paper is not novel. Similar implementations are used in many commercial program packages and by many development teams. However, neither the issues concerning the efficiency, nor the support of integrity constraints, nor the application development procedures, nor the design of queries are considered.

The technology proposed in this paper takes into account all those factors and, therefore, ensures the development of highly efficient and reliable software.

In this paper, we described only the basic aspects of the support of temporal features in nontemporal DBMSs. Many important topics remained beyond the scope of the paper. Some of them are as follows:

• additional functionality of applications (for example, the *Undo* function);

• implementation and use of event-based representation;

• interaction with the transaction support mechanism;

• representation and use of the valid time (in contrast to the transaction time).

The proposed technology has been used for years in many commercial projects that use large and complex databases in vital applications.

ACKNOWLEDGMENTS

This study was supported in part by the Russian Foundation for Basic Research, project no. 07-07-00268.

REFERENCES

1. Findler, N. and Chen, D., On the Problems of Time Retrieval, Temporal Relations, Causality, and Coexistence, *Proc. Second Int. Joint Conf. on Artificial Intelligence*, London, 1971.
2. Soo, M.D., Temporal Database Bibliography Update, *SIGMOD Rec.*, 1995, vol. 25, no. 1, pp. 14–23.
3. Bubenko, J.A., Jr., The Temporal Dimension in Information Modelling, *Working Conference on Architecture and Models in Data Base Management Systems*, Nijssen, G. M., Ed., Nice: North-Holland, 1977, pp. 93–118.
4. Abbod, T., Brown, K., and Noble, H., Providing Time-Related Constraints for Conventional Database Systems, *Proc. of the 13th International Conference on Very Large Data Bases*, Hammersley, P., Ed., San Francisco: Morgan Kaufmann, 1987, pp. 167–175.
5. Ahn, I., Towards an Implementation of Database Management Systems with Temporal Support, in *Proc. of the Int. Conf. on Data Engineering*, Los Angeles: IEEE Computer Society, 1986, pp. 374–381.
6. Ahn, I. and Snodgrass, R., Partitioned Storage for Temporal Databases, *Inf. Syst.*, 1988, vol. 13, no. 4, pp. 369–391.
7. Ariav, G. and Clifford, J., A System Architecture for Temporally Oriented Data Management, *Proc. of the Fifth Int. Conf. on Information Systems*, Tuscon, Arizona, 1984, pp. 177–186.
8. Dadam, P. and Teuhola, J., Managing Schema Versions in a Time-Versioned Non-First-Normal-Form Relational Database, *Technical Report TR 87.01.001*, Heidelberg Scientific Center IBM, 1987.
9. Dyreson, C.E., Snodgrass, R.T., and Jensen, C.S., On the Semantics of "Now" in Temporal Databases, *TempIS Technical Report no. 42*, Computer Science Department, University of Arizona, Tuscon, 1993.
10. Grandi, F., Scalas, M. R., and Tiberio, P.A., A History-Oriented Data View and Operation Semantics for Temporal Relational Databases, *Proc. of the Int. Workshop on an Infrastructure for Temporal Databases*, R. T. Snodgrass, Ed., Arlington, Texas, 1993.
11. Hsu, S. H., Page and Tuple Level Storage Structures for Historical Databases, *TempIS Technical Report no. 34*, Computer Science Department, University of Arizona, Tuscon, 1992.
12. Androutsopolous, I., Ritchie, G.D., and Thanisch, P., Experience Using TSQL2 in a Natural Language Interface, in *Recent Advances in Temporal Databases*, Clifford, S. and Tuzhilin, A., Eds., Zurich: Springer, 1995, pp. 113–132.
13. Bohlen, M.H., Jensen, C.S., and Snodgrass, R.T., Evaluating the Completeness of TSQL2, in *Recent Advances in Temporal Databases*, Clifford, S. and Tuzhilin, A., Eds., Zurich: Springer, 1995, pp. 153–174.
14. Date, C.J., Darwen, H., and Lorentzos, N., *Temporal Data & the Relational Model*, San-Francisco: Morgan Kaufmann, 2002.