Tạp chí Tin học và Điều khiển học, T.21, S.4 (2005), 293–300

A QUERY SUBLANGUAGE FOR TEMPORAL CLINICAL DATABASE SYSTEMS AND ITS IMPLEMENTATION

PHAM VAN CHUNG[1], DUONG TUAN ANH[2]

[1]*Department of Information Technology, Ho Chi Minh City University of Industry*
[2]*Faculty of Information Technology, Ho Chi Minh City University of Technology*

Abstract. We are developing a database implementation to support temporal data management in a treatment monitoring system for patients who have cancer diseases. We extend the standard relational model to support temporal data represented as time intervals, develop a set of operations on time intervals to manipulate time-stamped data and propose a temporal query sublanguage that can work with temporal clinical databases. We implemented this query sublanguage as a layer on top of Oracle DBMS using layered implementation technology.

Tóm tắt. Một cơ sở dữ liệu được xây dựng để hỗ trợ việc quản lý dữ liệu thay đổi theo thời gian trong một hệ thống theo dõi điều trị bệnh nhân ung thư. Chúng tôi mở rộng mô hình quan hệ truyền thống để hỗ trợ những dữ liệu thời gian được diễn tả như những thời khoảng, đề xuất một tập các phép toán trên các khoảng thời gian để chế hóa những dữ liệu được dán tem thời gian và đề xuất một tiểu ngôn ngữ truy vấn có tính thời gian mà có thể làm việc với các cơ sở dữ liệu bệnh viện. Tiểu ngôn ngữ truy vấn này được hiện thực hóa như là một tầng xây lên trên một hệ quản trị cơ sở dữ liệu Oracle bằng cách ứng dụng kỹ thuật "thi công theo tầng".

1. INTRODUCTION

Clinical databases typically contain a significant amount of temporal information. Questions relating to time are pervasive in medical decision-making. Common queries include simple questions: "Did this patient undergo radiation-therapy during the period 1/3/2003 - 29/3/2003?" or "What problems did this patient have during since 1/3/2003 up to now?" Queries relating to temporal order include: "List the patients that have undergone chemotherapy after a surgery".

Over the last decade, the temporal database community has made a significant amount of progress in temporal database systems. The most vibrant field of temporal database research is the area of temporal database models and temporal query languages ([3, 6]). Much of these research results can be applied to temporal clinical database systems. There are very few works on applying temporal databases in clinical data management. The Chronus II system [4, 5], built by O'Connor et al. at Stanford University School of Medicine, is one of the typical temporal query systems. Chronus II extends the standard relational model and the SQL query language to support temporal medical data. The system can store and process valid time event tables as well as state tables. However, in [4, 5], the implementation technology for the temporal query language on top of an existing relational DBMS was not given.

In this paper, we represent our approach to managing the temporal semantics of medical data in clinical database systems. First, we extend the standard relational model to support temporal data represented as time intervals. Second, we propose a temporal query sublanguage SubTSQL that can support the specified operations on temporal data. Finally, we also elaborate on how to implement the temporal query sublanguage on a relational DBMS such as Oracle. This query system is built as a layer that can convert temporal queries to Oracle SQL queries.

In what follows, we assume that the readers have a basic understanding of temporal databases - if not, we refer them to Date et al. [3] or Tansel et al. [6] for an introduction.

2. VALID-TIME TEMPORAL DATA MODEL

Modern clinical database systems typically use relational databases, which provide well-defined data models and query languages. However, the relational model has two significant shortcomings regarding temporal data:

- The relational model provides poor support for storing complex temporal information.
- The SQL query language provides very limited support for expressing temporal queries.

Therefore, applications that work with complex temporal data, such as clinical data should define their own temporal models and query systems.

2.1. Intervals, state tables and event tables

Several extensions to the relational model have been proposed to deal with these above shortcomings. Most researchers focused on *valid-time* databases, in which time factor is attached to all tuples in a temporal table. The valid time denotes *when* facts are true with respect to the real world. In valid time databases, two-dimensional relational tables are extended to incorporate time as a third dimension. In these tables, every tuple holds temporal information denoting the information's valid time.

Two types of temporal tables are *event tables*, which hold instant timestamps, and *state tables*, which hold *interval timestamps*. For instance, laboratory-test values are always stored in event tables, while information about drug treatments can be held in state tables, because drug treatments generally elapse over time.

Temporal data in state table can be represented as *intervals* which are bounded by *start* and *stop timepoints*. For example, [d04:d10] is the interval with start timepoint d04 denoting the 4^{th} day and stop timepoint d10 denoting the 10^{th} day.

We can also represent an event with a *pair* of timepoints for lower and upper bounds of the closed interval during which the user has specified that the event occurred. The lower and upper bounds of the interval are equal if the user is certain of the timepoint associated with an event.

Since temporal data in both state tables and event tables can be represented as intervals, we define an *interval-stamping* method for modeling a temporal database for clinical data management. A relation in such a database is called a *history*. Each row, or tuple, in a history will store the temporal dimension of a patient parameter over a closed interval; a pair of columns will be required in each history to represent the endpoints of the interval. For example, a temporal table *Patient* that holds information about patients and their diseases

could look as follows. The temporal information is stored in two columns labeled *V_begin* and *V_end* which give the starting time and ending time of a certain disease that a patient develops.

P_ ID	Problem	Dept	V_begin	V_end
J001	P1	D9	14/02/2000	01/03/2000
J001	P2	C2	10/03/2000	31/12/9999
P005	P3	D8	01/04/2000	12/05/2000
R006	P3	D8	13/02/2000	01/06/2000

In our temporal data model, the timepoints will have only a single *granularity*, which is at the smallest level of interest in the database applications. For example, if the granularity is one day, then we can say that the timepoints are all values of type DATE, and type DATE is the *point type* of intervals. When we consider an interval value, say [d04:d10], we know that the interval includes its begin and end points d04 and d10, by definition. We also know that the interval consists of *a set of points* arranged in according to some agreed ordering ([3]).

2.2. Interval operations

Since intervals are represented as pairs of timepoints, comparisons between intervals are based on timepoint comparisons of the upper and lower bounds. The interval comparison operators are BEFORE, AFTER, DURING, CONTAINS, OVERLAPS, MEETS, STARTS, FINISHES, and EQUALS. This set of comparisons was originally defined by Allen ([1]). Let $I1, I2$ be two intervals, and *begin(I), end(I)* be respectively the lower bound and upper bound of the interval I. The definitions of 9 interval comparisons are given in Table 1.

Table 1. Definitions of interval comparisons

	Comparison Operator	Meaning
1	$I1$ BEFORE $I2$	end($I1$) < begin($I2$)
2	$I1$ AFTER $I2$	end($I2$) <begin($I1$)
3	$I1$ DURING $I2$	(begin($I1$) > begin($I2$)∧ end($I1$) ⩽ end($I2$)∨ (begin($I1$) ⩾ begin($I2$)∧ end($I1$)<end($I2$)
4	$I1$ CONTAINS $I2$	(begin($I2$) > begin($I1$) end($I2$) end($I1$)∨ (begin($I2$) ⩾ begin($I1$)∧ end($I2$) <end($I1$))
5	$I1$ OVERLAPS $I2$	begin($I1$) <begin($I2$)∧ end($I1$) >begin($I2$) ∧ end($I1$) <end($I2$)
6	$I1$ MEETS $I2$	end($I1$) = begin($I2$)
7	$I1$ STARTS $I2$	begin($I1$) =begin($I2$)∧ end($I1$) < end($I2$)
8	$I1$ FINISHES $I2$	begin($I1$) > begin($I2$)∧ end($I1$) =end($I2$)
9	$I1$ EQUALS $I2$	begin($I1$) = begin($I2$)∧ end($I1$) = end($I2$)

Fold operation

Operators such as Union, Difference, Projection, and Cartesian product of the standard relational model remain the same in the valid-time temporal data model. Besides, there one important operator that works on temporal relations: *fold*. Tuples in a temporal relation that

agree on the explicit attribute values and that have adjacent or overlapping time intervals are candidate for folding.

Definition (Fold Operation). When an n-ary relation R is *folded* on interval attribute A_i, $1 \leqslant i \leqslant n$, all its tuples whose A_j components match $\forall j \neq i$ and whose A_i components can merge, are replaced in the resulting relation by a single tuple with the same A_j components, but in its i^{th} component is formed by a merging of the i^{th} component of these tuples.

Unfolded relations can arise in many ways, e.g. via a projection or union operator, or on update or insertion without enforcing folding.

3. A TEMPORAL QUERY SUBLANGUAGE

Incorporating time into the relational databases requires not only extensions to the relational model but also extensions to the SQL query language. The most comprehensive outline of a temporal query language is the TSQL2 query language ([8]). The TSQL2 specification integrates much of the current thinking in relational temporal database research and serves as a useful unified basis of research in the area. Unfortunately, no implementations of the TSQL2 query language exist. Because of its size and complexity, a complete implementation would require a daunting work. Not all features of the TSQL2, however, are necessary to build a workable temporal query system.

Here from TSQL2, we select a set of core features that we have found essential when building temporal query systems for clinical databases. We name this query sublanguage SubTSQL. This query sublanguage is sufficient to support the valid-time temporal data model given in Section 2 of which temporal clinical database is just a particular case. Later, we can gradually include additional query features into the query sublanguage if we encounter any temporal database application that requires so. That means SubTSQL can be the core of any general temporal query language.

SubTSQL is based on the simple structural framework of SQL, with syntactic extensions to support operations on events and states in histories. We now describe how temporal projection, selection, and joins are a set of algebraic operators that support temporal querying requirements. Assume that the valid time component in temporal table(s) must be well-defined before performing the operation. That means temporal tables do not contain tuples with the same non-temporal attribute values but overlapping or consecutive time intervals. Such tuples are automatically folded in advance by merging their time intervals.

Temporal projection. Temporal projection is similar to standard projection, except that the restriction applies to only the non-temporal attributes. Both timestamp columns cannot be excluded in the resultant history, because these columns maintain the temporal dimension of the data. After temporal projection, *folding* is enforced in order that adjoining intervals should be merged into a single interval in the resultant relation.

Temporal selection. SubTSQL adds the following new construct to standard SQL: selection based on temporal comparisons of timepoints and intervals using terms in a WHEN clause. The WHEN clause is used to express the temporal part of a query. The syntax of the SubTSQL retrieval statement contains the following clauses:

SELECT *select_item_list* FROM *table_name_list*

WHEN *temporal_comparison*

WHERE *search_condition*

The *temporal comparison* in the WHEN clause has the following form:

WHEN *a interval_compare_operator b*

where a, b are intervals or temporal tables; and *interval_compare_operator* can be one of the following keywords: BEFORE, AFTER, DURING, CONTAINS, OVERLAPS, MEETS, STARTS, FINISHES, and EQUALS.

Temporal join. This join has the most special semantics: the valid-time intervals of the resultant table are created from the *intersection* of the overlapping valid-time elements of the tables specified in the join. The valid time component in each temporal table must be well-defined before performing such joins.

Insertion of data. A new tuple can be inserted to a temporal table with the specified attribute values including an interval [*V_end*, *V_begin*] which builds the initial tuple lifespan. The syntax of the INSERT statement contains the following clauses:

INSERT INTO $\langle table - name \rangle$

$(\langle column - name - list \rangle)$

VALUES $\langle field - namevalues \rangle$

When a new tuple is inserted into table, then the *Fold* operator is enforced in order to merge the intervals, if necessary.

Modification of data. When updating a temporal table, a WHEN clause can be used to indicate the valid time associated with the update. The syntax of the UPDATE statement contains the following clauses:

UPDATE $\langle table - name \rangle$ SET $\langle column - name \rangle = \langle newvalue \rangle$

WHEN $\langle valid - time \rangle$

WHERE $\langle condition \rangle$

Only tuples that have a valid-time intersecting with the specified period in the WHEN clause are updated by the above command. Notice that using UPDATE command may result in a relation with more tuples than the original one.

Deletion of Data. When deleting data from a temporal table, a WHEN clause can be used to indicate the valid time associated with the deletion. The syntax of the DELETE statement contains the following clauses:

DELETE FROM $\langle table - name \rangle$

WHEN $\langle valid - time \rangle$

WHERE $\langle condition \rangle$

Only tuples that have a valid-time intersecting with the specified period are affected by the above command Notice that similar to UPDATE, using DELETE command may result in a relation with more tuples than the original one.

Dozens of query examples in the research have been tested on our implemented temporal query system. The test data files come from the clinical database developed for Ho Chi Minh City Cancer Hospital.

4. LAYERED IMPLEMENTATION OF SubTSQL

While developing a full-fledge DBMS that supports a temporal query language is a daunting task that only the major vendors can expect to accomplish, the implementation approach called *layered implementation* proposed by Torp et al. in 1997 [7] provides a faster development. According to this approach, a temporal query language is implemented via a software layer on top of an existing relational DBMS. The most important advantage of building on top of an existing DBMS is the possibility of reusing the services of the underlying DBMS. Another advantage is that we can achieve the compatibility of temporal query language with standard SQL with a minimum coding effort. The compatibility requires that the operation of the bulk of legacy code is not affected when temporal support is adopted.

Torp et al. [7] proposed *layered temporal database architecture* to implement a temporal query language with a minimum of temporal support. This approach is consistent with the desire for gradual availability of increasingly more temporal support. The layer uses the DBMS as a "black box" and there is minimal interaction between the layer and the DBMS. The architecture is given in Figure 1. It consists of the following modules.

• *Scanner.* The user inputs a SubTSQL statement Q to Layer, then the Scanner in Layer analyses it into tokens. Any errors found during scanning are reported. If no errors are found, Q is sent to Parser.

•*Parser.* Parser analyses the syntax of Q based on formal syntax of SubTSQL. Any errors found during parsing are reported. If no errors are found, Q is sent to Code Generator.

• *Code Generator.* Q is a temporal query; the Code Generator converts it to the equivalent Oracle-SQL command that is then sent to Oracle-DBMS.

•*Output Processor.* The result of Oracle-SQL is returned to Layer, and the Output Processor presents it to user.

The query system is a layer on top of the Oracle DBMS and its query language. This layer can convert temporal query commands into standard SQL commands. It is written in Visual C++ with more than 2000 lines. Experimental results on real data files show that the performance of the layer is quite promising.

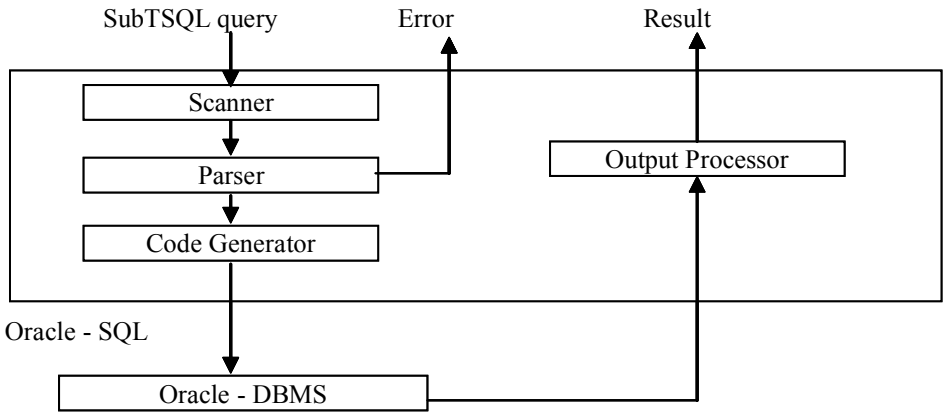

Figure 1. The layered implementation architecture

Translation of SubTSQL Query to Oracle-SQL

The code generator in Layer can convert a SubTSQL command to a semantically-equivalent Oracle-SQL command. Generally, the code generator in layer has to translate any interval comparison using one of the nine comparison operators given in Section 2.2 to equivalent expressions in oracle SQL command. The translation respects the semantics of the set of comparison operators described in Table 1. For example, given the table *Patient* as in Section 2.1 and the following SUBTSQL query

SELECT P_ID, Problem, V_begin, V_end FROM Patient

WHEN Patient DURING (25/3/2000 , 25/5/2000)

WHERE Dept = 'D8';

then the corresponding converted Oracle-SQL query will be as follows.

SELECT P_ID , Problem , V_begin, V_end FROM Patient

WHERE Dept ='D8' AND

(Patient.V_begin) > *to*_date('25/3/2000','dd/mm/yyyy')

AND(Patient.V_begin)⩽ (*to_date*('25/5/2000','dd/mm/yyyy'))

OR((Patient.V_end) ⩾ (*to*_date('25/5/2000','dd/mm/yyyy');

AND(Patient.V_end) < (*to*_date('25/5/2000','dd/mm/yyyy'))

where *to_date* is an Oracle built-in function that converts character data of the proper format to date data type. The condition part added in the WHERE clause after Dept = 'D8' describes the semantics of the DURING interval comparison operator. Generally, the code generator in Layer has to translate any interval comparison using one of the nine comparison operators given in Section 2.2 to equivalent expressions in Oracle SQL.

5. CONCLUSION

In this paper, we outlined our approach to modeling temporal clinical databases. The tables in temporal databases include valid time through two columns indicating the lower bounds and upper bounds of intervals. We define a query sublanguage SubTSQL in which query commands may include a WHEN clause to describe comparison predicates between time intervals. We implemented the query language as a layer on top of Oracle DBMS using layered implementation approach. All query examples have been tested with real data from a clinical database developed for Ho Chi Minh City Cancer Hospital and the experimental results are quite promising.

As the next phase in this ongoing research, we plan to develop knowledge-based temporal abstraction mechanisms which help physisians to detect patterns and trends in time-oriented patient data. Temporal query system and temporal abstraction module are two main modules in a decision-support system which is being built for supporting the treatment of cancer patients.

Acknowledgements. The authors are grateful to Medical Informatics Section of Ho Chi Minh City Cancer Hospital for permitting them to access to real clinical data relevant to this research.

REFERENCES

[1] J. F. Allen, Maintaining knowledge about temporal intervals, *Communication of the ACM* **26** (1993) 832–843.

[2] A. K. Das, S. W. Tu, G. P. Purcell, and M. A. Musen, An extended SQL for temporal data management in clinical decision-support systems, *Proc. of 16th Annual Symposium on Computer Applications in Medical Care*, Baltimore, 1992 (128–132).

[3] C. J. Date, Hugh Darwen, N. A. Lorentzos, *Temporal Data and the Relation Model*, Morgan Kaufmann Publishers, 2003.

[4] M. J. O'Connor, S. W. Tu, and M. A. Mussen, Applying temporal joins to clinical databases, *Proc. of the AMIA Annual Symposium*, Washington, D.C., 1999 (335–339).

[5] M. J. O'Connor, S. W. Tu, and M. A. Mussen, The Chronus II Temporal Database Mediator, *Technical Report, Stanford Medical Informatics*, Standard University School of Medicine, Stanford, CA, 2001.

[6] A. U. Tansel, et al. (eds.), *Temporal Databases - Theory, Design and Implementation*, Benjamin/Cummings Publishing, 1993.

[7] K. Torp, C. S. Jensen, and M. Bohlen, Layered Implementation of Temporal DBMSs-Concepts and Techniques, *Technical Report, TR-2, TimeCenter*, 1997.

[8] R. T. Snodgrass (ed.), *An Evaluation of TSQL2 Commentary*, TSQL2 Design Committee, September, 1994.

Received on March 24, 2005
Revised on December 26, 2005