

Applying Temporal Joins to Clinical Databases

Martin J O'Connor MSc, Samson W Tu MS, Mark A Musen MD PhD
Stanford Medical Informatics, Stanford University School of Medicine,
Stanford, CA 94305-5479

Clinical databases typically contain a significant amount of temporal information, information that is often crucial in medical decision-support systems. Most recent clinical information systems use the relational model when working with this information. Although these systems have reasonably well-defined semantics for temporal queries on a single relational table, many do not fully address the complex semantics of operations involving multiple temporal tables. Such operations can arise frequently in queries on clinical databases. This paper describes the issues encountered when joining a set of temporal tables, and outlines how such joins are far more complex than non-temporal ones. We describe the semantics of temporal joins in a query management system called Chronus II, a system we have developed to assist in evaluating patients for clinical trials.

Introduction

Most clinical data have a temporal dimension that is frequently recorded in databases by including an *instant time stamp* with particular records. Nearly all laboratory information, for example, contains some sort of time stamp, and times are frequently recorded for patient visits, drug therapies, and many types of treatments. As interactions between clinical decision support systems and databases grow more complex, the data's temporal dimension becomes increasingly important [1].

An instant time stamp permits a range of simple temporal questions about associated data: "Was the patient HIV-positive prior to the visit date?", or "Did the patient receive drug Y in the past week?"

However, associating a *period time stamp* with data enables more complex queries. Period time stamps are composed of a *start time stamp* and a *stop time stamp*, and delimit the data's valid time range. Most recent temporal database systems focus on this type of temporal information [2]. These databases are termed *valid-time* or *historical* databases. Table 1 is a temporal table called PROBLEMLIST, and contains period time stamps. The table contains codes for problems experienced by three patients, together with each problem's valid-time period. The temporal column is usually called the *valid-time component*. A period time stamp permits queries containing duration and concurrency predicates: "Did Smith experience problem P1 for longer than one month?", or "Did Jones experience problem P3 during January or February, 1998?"

Patient	Problem	ValidTime
J. Smith	P1	{[14/Feb/1998-1/Mar/1998]}
J. Smith	P2	{[10/Mar/1998-+']}
P. Jones	P3	{[1/Apr/1998-12/May/1998]}
R. Franks	P3	{[13/Feb/1998-1/Jun/1998]}

Table 1: PROBLEMLIST. A temporal table containing problem codes for a list of patients. The valid-time column shows the time periods during which the associated tuple is valid. The '+' value denotes that the tuple is still valid.

These queries operate on single temporal tables, and a temporal query engine can deal with them in a similar way to standard non-temporal queries. For example, the temporal component of the queries can be implemented as extra predicates to the SQL WHERE clause. Although efficient evaluation of these predicates requires considerable design work, maintaining the data's temporal dimension in a single table query is not significantly difficult. If a row is selected from the source table, its temporal column can simply be copied to the product table. If two rows have the same non-temporal values (*i.e.*, they are *value-equivalent*), some merging of their time periods may also be necessary.

However, combining multiple temporal tables in a single query quickly generates complications. For example, assume a temporal table called DRUGS (Table 2) contains drug treatment information for the patients listed in the PROBLEMLIST table.

Patient	Drug	ValidTime
J. Smith	D1	{[20/Mar/1998-12/May/1998]}
P. Jones	D1	{[1/Apr/1998-6/Jun/1998]}
R. Franks	D2	{[4/Feb/1998-14/May/1998]}

Table 2: DRUGS. A temporal table containing drug regimen information for the patients in the PROBLEMLIST table.

We then ask: "Show all problem and drug combinations for each patient." This query can be expressed in SQL as follows:

```
SELECT T1.Patient, T1.Problem, T2.Drug
FROM PROBLEMLIST AS T1, DRUGS AS T2
WHERE T1.Patient = T2.Patient
```

This query corresponds to a *join* in the relational model because it takes columns from multiple tables and combines them into a single table (Table 3). Unfortunately, all meaningful temporal information is lost, and the join produces combinations that are not temporally valid. For example, Smith received drug D1, and did experience problem P2, but these problems did not occur at *the same time*. Hence, combining them into a single row is not temporally valid.

Patient	Problem	Drug
J. Smith	P1	D1
J. Smith	P2	D1
P. Jones	P2	D1
R. Franks	P3	D2

Table 3. Join result of the PROBLEMLIST and DRUGS tables excluding temporal columns.

An alternative is to select the temporal rows explicitly:

```
SELECT T1.Patient, T1.Problem,
       T2.Drug, T1.ValidTime, T2.ValidTime
FROM PROBLEMLIST AS T1, DRUGS AS T2
WHERE T1.Patient = T2.Patient
```

However, this query also produces a temporally invalid table because it generates two time columns (Table 4).

Thus, successful multiple-table queries require a temporal join that can combine time stamped data from two or more temporal tables. This join must produce a new table that maintains the temporal semantics in the original tables. Although, *in principle*, standard SQL

Patient	Problem	Drug	ValidTime	ValidTime
J. Smith	P1	D1	{{[14/Feb/1998-1/Mar/1998]}}	{{[20/Mar/1998-12/May/1998]}}
J. Smith	P2	D1	{{[10/Mar/1998-+']}}	{{[20/Mar/1998-12/May/1998]}}
P. Jones	P2	D1	{{[1/Apr/1998-12/May/1998]}}	{{[1/Apr/1998-6/Jun/1998]}}
R. Franks	P3	D2	{{[13/Feb/1998-1/Jun/1998]}}	{{[4/Feb/1998-14/May/1998]}}

Table 4. Join result of the PROBLEMLIST and DRUGS table including temporal columns.

queries can be written to produce such a join, the resulting queries are very long, complex and difficult to write [3].

This paper outlines an approach for performing temporal joins. We first discuss temporal databases and briefly outline our temporal query system Chronus II. Throughout this paper, example queries from Chronus II illustrate the discussion.

Background

Most research on temporal query systems focuses on relational databases [3]. Queries to relational databases are typically expressed in SQL. SQL, however, provides poor support for temporal queries, and the relational model lacks a standard means for representing complex temporal information. Several extensions to the relational model have been proposed to address these shortcomings.

Most research has focused on valid-time databases, in which temporal information is attached to all n-tuples in a temporal table. This structure extends the two-dimensional relational table to incorporate a third dimension of time. In these tables, every tuple holds temporal information denoting the information's valid-time. Two types of temporal tables are common: *state tables*, which hold period time stamps, and *event tables*, which hold instant time stamps. For example, laboratory data are typically stored in event tables, while information about patient drug regimens can be

held in state tables, because drug treatments generally extend over time.

Clinical databases typically contain a mixture of state tables, event tables, and non-temporal tables and queries that select information from some combination of all these tables are common. Hence, supporting a variety of temporal joins is crucial.

Chronus II

Chronus II is a temporal query system developed for the EON project [4]. Its design was influenced by the original Chronus query system [5] and by TSQL2, a comprehensive temporal query language specification [3]. Both systems extend the standard relational model to support valid-time temporal queries, an approach that is consistent with most temporal database research. The original Chronus provided only state table joins because it did not support event tables or standard non-temporal relational tables. No implementations of the TSQL2 specification exist.

Chronus II supports state, event, and non-temporal tables, and provides an expressive temporal query language. This query language is independent of, but largely tailored to, the querying needs of clinical decision-support systems.

It was implemented in the Java programming language and was designed to operate as a layer *above* existing relational databases. Chronus II takes a temporal command, generates standard SQL statements for the non-temporal part of the command, and completes the processing of the temporal part in memory. Results are passed to the user or written to a database. It interacts with these databases through a JDBC interface and is not tied to any particular database implementation.

Example Chronus II Query

We wish to vary our earlier request to show problem and drug combinations for all patients, while showing only combinations that involve drug regimens lasting longer than two weeks, and excluding combinations where a drug regimen was initiated before the start of a problem. This query can be written in Chronus II as follows:

```
TEMPORAL SELECT T1.Patient, T1.Problem, T2.Drug
FROM PROBLEMLIST AS T1, DRUGS AS T2
WHERE T1.Patient = T2.Patient
WHEN DURATION(T2) > WEEKS(2) AND
    AFTER(START(T1), START(T2))
```

Although not all details of the above query are relevant to this discussion, a few points are noteworthy. The query resembles standard SQL with some extra clauses. Apart from the initial ‘TEMPORAL’ keyword, the most obvious addition is the WHEN clause. This clause is analogous to the WHERE clause and contains temporal predicates. The result of this query will be presented in the next section.

Temporal Joins

Relationally joining multiple tables involves generating a subset of the Cartesian product of all tuples in the tables. In a SQL query, this subset is determined by the columns specified in the SELECT clause and by the predicates in the WHERE clause. In a temporal join, it is further restricted by temporal predicates. However, a temporal join differs from a non-temporal one because the temporal attributes in the operand tables must be treated specially. To perform temporally valid joins, we must combine the temporal information from the operand tables and generate a single meaningful temporal column.

The valid-time component in each temporal table must be well-defined before performing such joins. If all temporal information and queries concerned past events, capturing the semantics of the information would not be difficult. However, temporal databases must also describe events that are *ongoing*, and may also need to describe *future* events. Thus, the range of temporal information modeled by a table’s valid-time component must be carefully defined before it can be used in a join. This definition will also prove crucial when joining temporal with non-temporal tables.

Semantics of Time Stamps

When recording state information, a value for the end time stamp is often unknown, i.e., the event is still ongoing. If a fact is still valid, a suitable value for the end time stamp is not obvious.

There are several possible solutions to this problem [6]. The most straightforward one is to use the *present time* or *transaction time* when recording currently valid information. However, this approach will produce a database table that becomes out-of-date the instant after the fact is recorded: the database will record the fact as being no longer true, while in reality it continues to be valid.

A related approach is to record an explicit ‘end-of-time’ value, such as “12/31/9999”, as the end time stamp field. However, this solution again produces a database that inaccurately reflects reality because it makes potentially invalid assumptions about the future. A query involving the future may return erroneous results if it assumes that some facts will always be valid.

Alternatively, special values may be put in a period’s end time stamp. These values are not explicit time values, and are interpreted specially by the temporal query system. Many systems use a value of ‘forever’, but queries involving the future must be excluded or this approach suffers from the same failings as the ‘end-of-time’ method.

A more robust solution is to use the special value of ‘until changed’, which notes that a fact is true until recorded otherwise. This solution avoids the problem of invalid future assumptions. An ‘until changed’ time stamp will not be used in future state computations because the query system will not assume that a fact will *remain* true because it *is* now true. The fact may indeed prove true at a later time, but we simply could not have known so when making the query. Many systems use the term ‘now’ instead of ‘until changed’, but apply the same semantics. The word ‘now’, however, does not accurately capture the meaning of the time stamp.

In some cases, using the ‘forever’ time stamp as a period’s end time stamp may be appropriate if it is known that a fact will remain true forever. Similarly, if a fact has always been true, a special ‘beginning’ time stamp can be used.

These issues do not arise with event tables because they record events happening at instants in time. By definition, the time of an event must be known before it can be recorded.

We have simplified this discussion by assuming that all time stamps are recorded at the same granularity. In reality, temporal data are recorded at different granularities, particularly in clinical databases. However, relaxing this assumption would not change the nature of this discussion. The related issue of temporal uncertainty has also not been addressed. In many clinical situations, the *exact* time of events is not known. Dealing with this uncertainty in temporal joins is beyond the scope of this paper.

Joining Temporal Tables

This paper will discuss three principal types of pure temporal joins: (1) state table with state table, (2) event table with event table, and (3) state table with event table. To simplify the discussion, only two-way joins are shown. Joins of three or more tables can be implemented using a succession of two-way joins.

Joining Two State Tables

The earlier Chronus II example query was a temporal join of two state tables. In order to perform this join, we must first assemble the non-temporal columns as we would for a non-temporal join. The columns are assembled by generating the cross product of the non-temporal columns from the operand tables, and then

excluding rows that do not satisfy the predicates in the WHERE and WHEN clauses.

The *temporal* columns must be treated separately. We must consider the semantics of the valid-time component in each operand tuple before attempting to combine these tuples into the final joined table. If we wish to combine elements from multiple tuples in a temporal join, we can do so only if the facts they capture are true *at the same time*. That is, the temporal periods associated with each tuple must overlap.¹

Thus, we must examine the source tuples for each candidate tuple in the reduced cross product to see if their valid-time periods intersect. If they do overlap, the candidate tuple is included in the final join product and the result of this intersection is used as the valid-time period of the new tuple. If they do not overlap, this tuple is excluded from the result of the join. Thus, the join preserves the original temporal semantics (Table 5).

Patient	Problem	Drug	ValidTime
J. Smith	P2	D1	{[20/Mar/1998-12/May/1998]}
P. Jones	P3	D1	{[1/Apr/1998-12/May/1998]}

Table 5. Temporal join of the PROBLEMLIST and DRUGS tables restricted to problems lasting longer than two weeks and whose onset occurs before each drug regimen.

The semantics of the ‘+’ or ‘until changed’ time stamp are worth noting. When the query is made, this time stamp is internally replaced with the value of the current time. If the above query is made after 20/Mar/1998, the intersection of the time period [10/Mar/1998-+’] with [20/Mar/1998-12/May/1998] will result in a time period of [20/Mar/1998-12/May/1998]. If the query were asked after 20/Mar/1998, and before 12/May/1998, the end time stamp would record the date of the query. Also, if both time periods in this example had an ‘until changed’ end time stamp, the end time stamp of the final intersected period would retain this value.

Joining two Event Tables

A similar approach works when joining event tables. Because we are now dealing with instant time stamps, we can determine if the tuples can be combined temporally by using the temporal equality operator. That is, tuples from two event tables can be joined into one table only if their instant time stamps are equal. If they are, the events clearly occurred at the same time, so their combination is temporally valid. The instant time stamp for each tuple in the product table can be taken directly from any one of its operand tuples.

Joining a State and an Event Table

In order to combine facts in a temporal join, a fact specified in an event table must occur *during* the lifetime of a fact specified in a state table. Therefore, to combine an event tuple with a state tuple, the event tuple’s time stamp must occur during the period specified by the state tuple’s time stamp. The product tuple’s valid-time will be the *instant* that they overlapped. As a consequence, joining a state table and an event table will always produce an event table.

Joining Temporal and Non-Temporal Tables

There are two main approaches for combining a temporal with a non-temporal table: (1) turn the temporal table into a non-temporal table, or (2) turn the non-temporal table into temporal table, and then join the two tables.

A temporal table can be transformed into a non-temporal table by removing its valid-time component. However, stripping the time component will not produce a valid table because doing so effectively promotes all historical information into the present time. Excluding tuples that are not valid when the stripping takes place avoids this problem. In either case, all historical information is lost.

If we wish to ask temporal questions, we must produce a temporal table from a non-temporal one. This process requires generating a suitable valid-time component for the non-temporal table. The non-temporal table can be converted into an event table with the present time as its valid-time component. However, this approach is unlikely to retain the semantics of the original table. For example, a non-temporal-turned-event table could inaccurately classify a patient’s gender as true only at the present time.

A state table provides a more flexible solution. An obvious approach is to construct a valid-time component with ‘beginning’, and ‘forever’ as its start and end time stamp, respectively. Other possibilities include using ‘now’ and ‘until-changed’ as the end time stamp. However, without knowing the meaning of the original non-temporal data, we can never hope to be fully accurate in our choice. In any case, using ‘beginning’ and ‘forever’ as start and end time stamps will generally produce satisfactory results. Once a suitable valid-time component for the non-temporal table is chosen, the join can proceed as normal.

Discussion

Disease processes and many clinical interventions have temporal durations. A model that captures these durations as explicit temporal intervals would allow queries for temporal patterns. Performing these queries

¹ To combine non-overlapping elements in a join, we can use a standard non-temporal join.

would be difficult in an instant time stamped database [7]. However, few current clinical information systems support intervals, and the medical informatics field has no standard means for making temporal queries using intervals. The Arden Syntax, for example, does not provide direct support for intervals, and has limited temporal querying abilities [8].

The Arden Syntax does, however, support a type of query that allows the combination of multiple temporally concurrent data elements into a single list. This process provides a simple way of doing a temporal join, and illustrates that temporal joins are necessary even in instant-based databases. The semantics of this join are not well specified, however, and the process cannot be extended to interval-based data. Also, because instant time stamps encode only explicit time values, they cannot directly represent ongoing clinical situations.

We have outlined a data model that allows the temporal combination of instant and interval-based data, while rigorously preserving the data's temporal semantics. Defining these semantics requires great care. We have implemented a query system called Chronus II that supports temporal joins. This system is being used by guideline-based applications in the EON [4] and RÉSUMÉ projects [9]. It is deployed in a local clinic as part of the ATHENA guideline advisory system [10].

An added impetus for interval support in clinical databases is that the database community has accumulated considerable experience with valid-time databases. This experience can be applied to the clinical domain. Future SQL standards are also likely to use ideas from this body of research. For example, some components from the TSQL2 temporal query specification have already been incorporated into the SQL3 standard [11].

Although few current clinical databases support interval-based data, these intervals can often be constructed from existing instant time stamps. This process usually requires detailed knowledge of the underlying data, and may entail a customized solution for each clinical domain. However, it does provide a path to enable the exploitation of the full power of valid-time temporal databases.

Acknowledgement

This work has been supported, in part, by grant LM05708 from the National Library of Medicine.

We thank Valerie Natale for her valuable editorial comments.

References

1. Shahar Y, Combi C. *Timing is everything: time-oriented clinical information systems*. Western Journal of Medicine, 1997; **168**:105-113.
2. McKenzie LE, Snodgrass RT. *Evaluation of relational algebra incorporating the time dimension in databases*. ACM Computing Survey, 1991; **23**:501-543.
3. Snodgrass RT (Ed). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.
4. Musen MA, Tu SW, Das AK, Shahar Y. *EON: A component-based approach to automation of protocol-directed therapy*. Journal of the American Medical Informatics Association, 1996; **3**(6): 367-388.
5. Das AK, Musen MA. *A temporal query system for protocol-directed decision-support*. Methods of Information in Medicine, 1994; **33**:358-370.
6. Snodgrass RT. *On the semantics of 'now' in databases*. ACM Transactions on Database Systems, 1997; **22**(2): 171-214.
7. Clifford J, Dyreson CE, Isakowitz T, Jensen CS, Combi C, Shahar Y. *Temporal reasoning and temporal data maintenance in medicine: issues and challenges*. Computers in Biology and Medicine, 1997; **27**(5):353-368.
8. Hripcsak G, Ludemann P, Allan Pryor T, Wigertz, OB, Clayton P. *Rationale for the Arden Syntax*. Computers and Biomedical Research, 1994; **27**: 291-324.
9. Shahar Y, Musen MA. *RÉSUMÉ: a temporal-abstraction system for patient monitoring*. Computers and Biomedical Research, 1993; **26**:255-273.
10. Advani A, Tu SW, O'Connor MJ, Coleman R, Goldstein MK, Musen MA. *Integrating a Modern Knowledge-Based System Architecture with a Legacy VA Database: The ATHENA and EON Projects at Stanford*. AMIA, Washington, D.C., 1999.
11. Snodgrass RT, Jensen CS, Steiner A. *Transitioning temporal support in TSQL2 to SQL3*. Temporal Databases: Research and Practice, 1998; 150-194.