Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

QUERY PROCESSING FOR TEMPORAL DATABASES

T.Y. Cliff Leung
Richard R. Muntz

April 1989
CSD-890024

Query Processing for Temporal Databases*

T.Y. Cliff Leung
Richard R. Muntz
University of Calfornia, Los Angeles

April 30, 1989

Abstract

With the need for storing time-varying information in databases and the availability of cheap storage units, building temporal databases becomes more attractive and realistic. In this paper, we consider query processing and optimization for temporal databases; an area that is seldom discussed in the literature. As storing temporal information in databases is a new application area, it is not surprising that it imposes new requirements for effective query processing techniques. Conventional relational systems are often inefficient for temporal queries because these requirements are not taken into consideration. As an example, a temporal join often contains a conjunction of several inequalities involving timestamps. This type of query is conventionally processed using the nested-loop join algorithm, which may not be the most efficient method for this type of qualification. A query having a conjunction of a number of inequalities as the join condition can be processed much more efficiently using new processing strategies.

We discuss a stream processing approach for temporal query processing. Given properly sorted data, the implementation of temporal joins and semijoins as stream processors can be very effective. We also discuss the tradeoffs between sort orders, the amount of local workspace and multiple scans over input streams; particularly, we are interested in the effect of sort ordering on the local workspace requirement. We present stream processing algorithms for various temporal joins and semijoins, and their workspace requirements for various data sort orderings. We note that the optimal sort ordering for a query may depend on the statistics of data instances. Finally, we point out that semantic query optimization can play an important and natural role in temporal databases.

*This work was done under the Tangram Project, supported by DARPA contract F29601-87-C-0072.

Query Processing for Temporal Databases

T.Y. Cliff Leung
Richard R. Muntz
University of Calfornia, Los Angeles

1 Introduction

Many real database applications intrinsically involve time-varying information. With the availability of cheap processing and storage units, there is a growing interest in temporal databases which store the evolving history of the 'enterprise' of interest. Most research on temporal databases can be roughly categorized into three areas [McK86]. The first is the formulation of the semantics of time [All83, Cli83] and is closely related to research issues in artificial intelligence. The second area concerns physical implementation issues [Lum84, Ahn86, Rot87]; the focus is mainly on new access methods and data organization strategies.

The third area is the logical modeling of temporal data [Ben82, Cli85, Sno85, Sho86, Seg87, Sno87, Gad88]. Many of these studies emphasize extending the relational data model to capture time semantics and to support relational temporal query languages. These extended models generally augment relations of the snapshot data model with several temporal attributes (such as ValidFrom and ValidTo attributes [Sno87]) which store the relevant timestamps. New temporal operators are also defined in these extended data models, based upon traditional relational algebraic operators [Ull82], to allow users to query temporal attributes but not update them directly. Recently some studies propose using non-first normal form relation technology to model temporal data [Tan88].

In this paper, we consider query processing and optimization for temporal databases, a topic which is seldom discussed in the literature. We observe that there are several interesting characteristics which are peculiar to temporal queries: (1) a temporal query often contains a conjunction of several inequalities and no equality conditions; (2) temporal data is rich in semantics, and semantic query optimization is desirable in the presence of a number of inequalities; and (3) a temporal query may contain several references to the same relation. These characteristics provide new opportunities for optimization. Ignoring these, as in conventional relational systems, can result in poor performance.

Join and semijoin operations are the most common and expensive computations in database systems. In this paper, we discuss processing various temporal join and semijoin operations using a stream processing approach which takes advantage of data ordering. As temporal data often has certain implicit ordering by time, the stream processing approach, as we demonstrate, is a good alternative. We should emphasize, however, that the stream processing algorithms that we present in this paper are merely additional strategies that a query optimizer should consider, and are by no means substitutes for traditional query processing methods.

The idea of stream processing has also appeared in [Bent79, Pre85, Ore88]. [Ore88] focuses on processing spatial image data stored using pixel representation which is not appropriate in many proposed temporal data models. Its implementation relies on a special transformation, called the z-transform, and consequently there is only one interesting ordering, namely the z-order (lexicographical ordering of z-transformed values.) [Bent79, Pre85] discusses a main memory algorithm, called the plane sweep algorithm, which can be used to report all intersecting pairs of planar line segments. Although temporal attributes (ValidFrom and ValidTo) can be thought of as the endpoints of horizontal line segments, the algorithm is primarily designed to find all line segment pairs which intersects at a single point, and therefore it is not directly applicable to temporal query processing. Moreover, only a single sort ordering of data items is considered by this algorithm while in our temporal data model, tuples can be sorted on the attribute ValidFrom or ValidTo. Nonetheless, these researches share the basic principle of the stream processing paradigm which is that input data should be in a certain order before the processing commences. In this paper, we are more concerned with (1) the impact of various data ordering on performance issues, mainly memory workspace requirements, and (2) efficient processing algorithms for join and semijoin operations. As we show, the optimal sort ordering for these temporal operators may depend on the statistics of data instances as well as the operator itself.

Semantic query optimization has been discussed in the literature [Kin81, Cha84, Jar84, She87] but apparently has not been widely used in conventional systems. Undoubtedly semantic constraints in temporal databases occur more naturally and are more plentiful, and consequently a temporal query optimizer should profitably exploit the semantic constraints. In this paper, we discuss a new type of semantic constraint — *chronological ordering* of data items and how it can be used to optimize a temporal query.

The remainder of the paper is organized as follows. Section 2 gives an overview of the temporal data model that we adopt from [Sho86, Seg87] and discusses the basic categories of temporal queries. We illustrate, in Section 3, the conventional approach to processing a complex temporal query. In Section 4, we discuss a stream processing approach for the implementation of temporal operators. We informally discuss the role of semantic query optimization in Section 5, and finally conclude with directions for future work.

2 Temporal Data Model

In our temporal data model, we consider time as a sequence of discrete, consecutive, equally-distanced points, i.e. Time = $\{t_o, t_1, \cdots, now\}$ which are totally ordered. The sequence of time points can simply be treated as isomorphic to the natural numbers, and therefore we do not specify the time unit.

We adopt a modified version of the Time Sequence concept in [Sho86, Seg87][1] as the basic data construct in our temporal data model. A temporal data value is a 4-tuple <S,V,ValidFrom,ValidTo>[2] where S is the surrogate or the identity of the object, V is a time-varying attribute of concern, and [ValidFrom,ValidTo) represents the lifespan of the tuple. Naturally, within a tuple the ValidFrom value is always smaller than the ValidTo value. Semantically, the object S has attribute value V during the period [ValidFrom,ValidTo)[3]. A temporal relation is a set of temporal data values (i.e. a set of 4-tuples).

An example of a temporal relation is **Faculty**(Name,Rank,ValidFrom,ValidTo)[4]. Together with the following integrity constraints and assumptions, this example is used in subsequent sections for illustration purposes. Name is the identity of a faculty member. For attribute Rank, we consider only three different ranks — 'Assistant', 'Associate' and 'Full'. Suppose we assume in this example that an assistant professor can be promoted only to an associate professor and then to a full professor. In other words, there is a chronological ordering among the data values that the Rank attribute can assume. For the same faculty member, e.g. "Smith" as illustrated in Figure 1, "$\text{ValidTo}_1 \leq \text{ValidFrom}_2$" and "$\text{ValidTo}_2 \leq \text{ValidFrom}_3$" must hold. The period [ValidFrom,ValidTo) in a tuple is the time during which the faculty holds the indicated rank. We also assume that a faculty member is at exactly one rank at any time between becoming an assistant professor and termination as a full professor. As we mentioned above, for any tuple t, "t.ValidFrom<t.ValidTo" always holds.

[1]A Time Sequence is a totally ordered sequence of temporal data values <Surrogate, Attribute-value, Time>. The attribute value of an object between any two time points (i.e. between consecutive temporal data values) can be computed using an interpolation function.

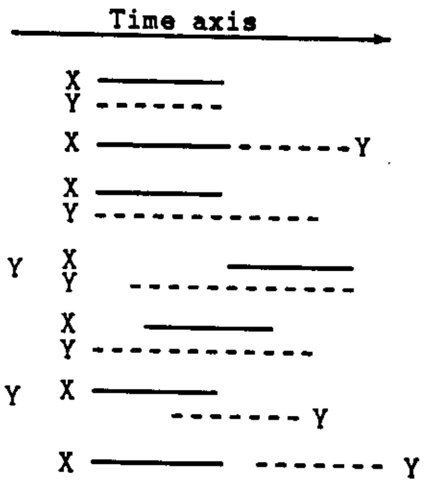
[2]We consider only the valid times in TQuel temporal database taxonomy [Sno85]. Also, for simplicity, we often use TS and TE to abbreviate ValidFrom and ValidTo respectively.

[3]A stepwise-constant interpolation function is applied between the time points ValidFrom and ValidTo.

[4]Borrowed from [Sno87].

	⋮		
Smith	Assistant	$ValidFrom_1$	$ValidTo_1$
Smith	Associate	$ValidFrom_2$	$ValidTo_2$
Smith	Full	$ValidFrom_3$	$ValidTo_3$
	⋮		

Figure 1: A sample Faculty relation



Temporal Operators	Explicit Constraints
(1) X equal Y	X.TS=Y.TS ∧ X.TE=Y.TE
(2) X meets Y	X.TE=Y.TS
(3) X starts Y	X.TS=Y.TS ∧ X.TE<Y.TE
(4) X finishes Y	X.TE=Y.TE ∧ X.TS>Y.TS
(5) X during Y	X.TS>Y.TS ∧ X.TE<Y.TE
(6) X overlaps Y	X.TS<Y.TS ∧ X.TE>Y.TS ∧ X.TE<Y.TE
(7) X before Y	X.TE<Y.TS

TS ≡ ValidFrom
TE ≡ ValidTo
Integrity Constraints: X.TS<X.TE ∧ Y.TS<Y.TE

Figure 2: The 13 possible temporal relationships

3 Conventional Approach

In this section, we describe how conventional relational database systems generally process temporal queries. Using a running example, we illustrate deficiencies in conventional systems, which motivate our investigation of more efficient processing strategies for temporal queries.

Allen [All83] presents thirteen elementary temporal operators of time-intervals which are listed in Figure 2. These temporal operators are actually just syntactic sugar for the "explicit" constraints (i.e. query-specific constraints) which are given in the right hand column of Figure 2, and can be easily incorporated into query languages like SQL and Quel.

Temporal queries using these extended constructs are usually processed in the following way. First, they are translated into equivalent queries in a relational language such as Quel. The translated queries will then be optimized and processed by conventional relational query processors. This 'syntactic sugaring' approach, as we will demonstrate below, is in general not effective for temporal query processing.

Suppose we have a relation **Faculty**(Name,Rank,ValidFrom,ValidTo) as described in the previous section. Consider the following Quel query modified from [Sno87][5]: *Superstar — Who got promoted from assistant to full professor while at least one other faculty remained at the associate rank?*

```
range of f1 is Faculty
range of f2 is Faculty
range of f3 is Faculty
retrieve into Stars(Name=f1.Name,ValidFrom=f1.ValidFrom,ValidTo=f2.ValidTo)
where f3.Rank="Associate"
        and f1.Name=f2.Name and f1.Rank="Assistant" and f2.Rank="Full"
        and (f1 overlap f3) and (f2 overlap f3)[6]
```

These "overlap" operators are translated directly into equivalent clauses involving inequali-

[5]The original TQuel query in [Sno87] is:

```
range of f1 is Faculty
range of f2 is Faculty
range of a is Associate
retrieve into Stars(Name=f1.Name)
        valid from begin of f1 to begin of f2
        where f1.Name=f2.Name and f1.Rank="Assistant" and f2.Rank="Full"
        when (f1 overlap a) and (f2 overlap a)
```

[6]This *overlap* operator defined in [Sno87] is different from "overlaps" in [All83]; it is defined in a general sense and therefore it may also mean the "equal", "start", "finishes" or "during" relationships in Figure 2. For the sake of exposition, we follow [Sno87].

ties. That is,

$$(\text{f1 overlap f3}) \equiv \text{f1.ValidFrom}<\text{f3.ValidTo} \wedge \text{f3.ValidFrom}<\text{f1.ValidTo}$$
$$(\text{f2 overlap f3}) \equiv \text{f2.ValidFrom}<\text{f3.ValidTo} \wedge \text{f3.ValidFrom}<\text{f2.ValidTo}$$

The corresponding relational algebra expression for the Superstar query is:

$$\pi_L\,(\sigma_\theta\,(\text{Faculty}_{f1} \times \text{Faculty}_{f2} \times \text{Faculty}_{f3}))$$

where L	is	f1.Name, f1.ValidFrom, f2.ValidTo
θ	is	f1.Name=f2.Name $\wedge$ f1.Rank="Assistant"
		$\wedge$ f2.Rank="Full" $\wedge$ f3.Rank="Associate" $\wedge$ θ'
θ'	is	f1.ValidFrom<f3.ValidTo $\wedge$ f3.ValidFrom<f1.ValidTo
		$\wedge$ f2.ValidFrom<f3.ValidTo $\wedge$ f3.ValidFrom<f2.ValidTo

This algebraic expression can be represented as a parse tree [Ull82], as depicted in Figure 3(a). The parse tree can then be ameliorated by applying well-known traditional algebraic manipulation methods; e.g. the selections and projection are pushed as far down the parse tree as possible (see Figure 3(b)).

There are several interesting observations about the "conventionally optimized" parse tree in Figure 3(b):

1. The first join in the parse tree can be efficiently implemented as an equi-join using a conventional approach such as nested-loop join, merge join or hash join. The second join operation, a so-called *less-than join*, is a Cartesian product followed by a selection with the condition being a conjunction of inequality predicates — θ'. Traditionally, the best strategy for processing less-than joins appears to be the conventional nested-loop join method. With only a single inequality as the join condition, we have no choice but the nested-loop join method. Since time is assumed as a sequence of toally ordered points, one would wonder if there are any more efficient processing alternatives for a conjunction of several inequalities involving temporal attributes.

   In the past, little attention has been given to this form of qualification because:

   - in traditional database applications, queries seldom contain less-than joins, and
   - when they do, in most situations the join condition has only a single inequality predicate; for example, in an Employee/Department database, we might want to retrieve employees whose salary is higher than his/her manager.

   When we consider temporal databases, the situation is quite different:

- less-than joins appear more frequently and naturally, and therefore need to be optimized,
- the less-than join condition usually contains a conjunction of several inequality predicates (e.g. in the Superstar example) which indicates that optimization might be possible.

Optimization of this form of query in snapshot databases was essentially ignored but for temporal databases this can result in severe performance penalties. In the following sections, we explore a number of efficient processing alternatives.

2. Recall that there is an integrity constraint in the Faculty relation: a chronological ordering of data values — 'Assistant', 'Associate' and 'Full'. This ordering implies that being an assistant professor must occur before being promoted to a full professor, i.e. "f1.ValidTo<f2.ValidFrom" always holds in the presence of (f1.Name=f2.Name). These constraints, together with the "intra-tuple" integrity constraints,

   fi.ValidFrom<fi.ValidTo   for i=1,2,3,

   imply "f1.ValidFrom<f3.ValidTo" and "f3.ValidFrom<f2.ValidTo". Therefore these inequalities in θ' are redundant — i.e. they are subsumed by other inequalities. The important point is not so much this particular case; rather it is the process of semantic query optimization. We argue in more detail later that semantic query optimization is much more applicable in temporal databases than in conventional shapshot databases.

3. There are three references to the Faculty relation in the parse tree implying that it is joined with itself twice — conventional systems would scan the relation several times. If we view the query as a "Superstar" *pattern matching* in the Faculty relation, one might wonder if we are able to answer this query with only a single scan of the relation. Roughly speaking, the pattern that we are looking for is "an assistant professor followed by a full professor and then an associate professor" (which also satisfies other conditions such as the assistant and full professors are the same person.) That is, instead of performing multiple joins, a single scan might be possible by recognizing this query qualification as describing a pattern in the data.

The above observations suggest new requirements for temporal query processing algorithms. These new requirements in turn suggest that, in addition to traditional set-oriented relational operators, we may need other alternatives to process temporal queries. In subsequent sections we will present and discuss a number of such alternatives.

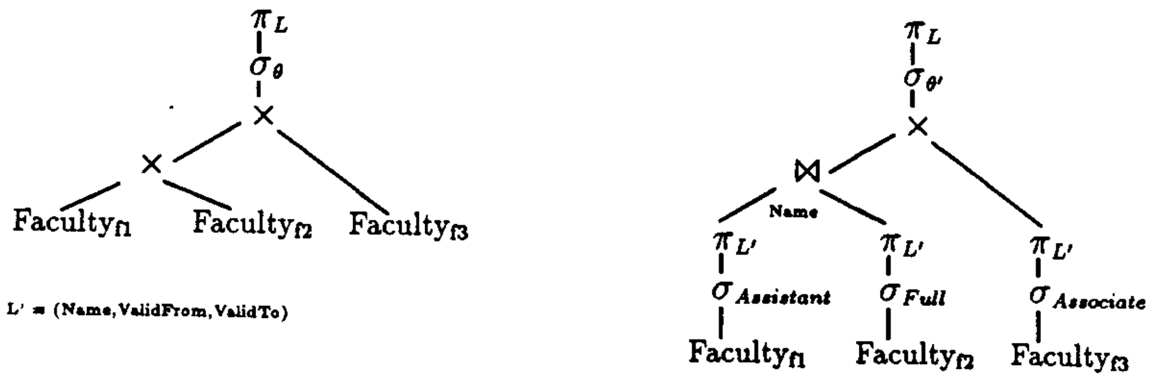
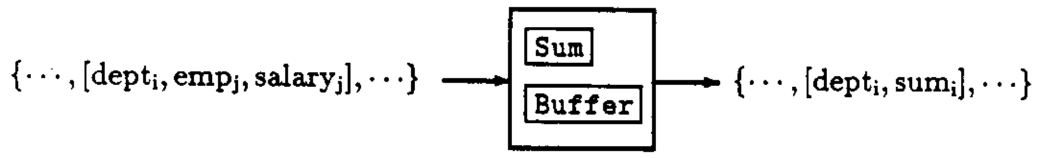


Figure 3: (a) Parse tree for the Superstar expression and (b) its optimized version



$\{\cdots, [dept_i, emp_j, salary_j], \cdots\}$
Sum
Buffer
$\{\cdots, [dept_i, sum_i], \cdots\}$

Figure 4: A stream processor to sum all employees' salaries in each department: emp_j is the employee, $dept_i$ is the department that the employee works with, $salary_j$ is the employee's salary.

4 Stream Processing Approach

We discuss a *stream processing* approach for temporal query evaluation in this section. Algorithms that implement temporal operators are presented. The tradeoffs among sort orders, the amount of local workspace and multiple passes over input streams are discussed. For properly sorted streams of tuples, we show that temporal operations can often be carried out with a single pass of input streams and the amount of workspace required can be small.

4.1 What is stream processing?

Abstractly, a *stream* can be defined as an ordered sequence of data objects. Stream processing is a paradigm which has been widely studied [Abe85, Par89] and used in languages such as Lisp; it is very similar to list processing in which elements of a list are sequentially processed. Stream processing also appears naturally in database systems; it closely resembles the notion of dataflow processing. In the functional data models [Shi81, Bat88, Ore88] a function, which is implemented by a stream processor, is a mapping from input stream(s) into output stream(s). Furthermore, function composition can be viewed as "connecting" a network of stream processors through which data objects flow.

A classical example of stream processing operations is the merge-join. When we merge-join two relations sorted on their key attribute, at any point we need only one tuple from each table as the "state". The join is efficiently implemented as both tables are read only once. Moreover, the output from this join operation is also sorted on the key attribute so that subsequent operations on this output can then take advantage of this ordering [Smi75, Sel79].

There are several intrinsic characteristics of stream processing in database systems. First, a computation on a stream has access only to one element at a time and only in the specified ordering of the stream. Second, the implementation of a function as a stream processor may require keeping some local state information in order to avoid multiple readings of the same stream. The state represents a summary of the history of a computation on the portion of a stream that has been read so far; the state may be composed of copies of some objects or some summary information of the objects previously read (e.g. sum, average, etc.) Using the local state information, the implementation of a stream processor can be expressed in terms of functions on the individual objects at the head of each input stream and the current state. That is, a stream processor takes an object from each input stream and, depending on the current state, it can change the current state to a new state and at the same time output some objects on its output stream(s).

Let us consider a simple stream processor which lists all the departments and computes the sum of all employees' salaries in each department, as shown in Figure 4. If the stream of tuples are grouped by the department name, the local workspace simply contains the partial sum and a buffer for the tuple just read. The point here is that the state contains summary information and the function (i.e. sum) is expressed in terms of the current state and an input object.

The third characteristic of stream processing is that there are often tradeoffs among the following factors:

1. the size of the local workspace which depends on the function itself, the statistics of specific instance of data streams, and the garbage-collection criteria,
2. sort order of input streams, and
3. multiple passes over input streams (i.e. the number of disk accesses).

Very often stream processing requires input streams to be properly sorted in order to perform the computation while only read the input streams once. In addition, the sort ordering of input streams greatly affect the size of local workspace required. Conversely, suppose there is enough local workspace to keep all data objects. Then only a single pass over the input streams is required and (theoretically) the sort ordering would not be important.

For many practical situations in query processing, it is important to make use of the ordering of tuples so that we can minimize the amount of local workspace and the number of passes over input streams. As we mentioned earlier, temporal data implies ordering by time; treating temporal relations as ordered sequences of tuples (i.e. streams of tuples) therefore suggests that stream oriented strategies for temporal query processing could be especially effective. In the next section, we discuss the application of stream processing algorithms for implementing temporal queries. In these discussions, the sort ordering of streams plays a major role.

4.2 Sort Orderings

Suppose we have temporal relations **X**(S,V,TS,TE)[7] and **Y**(S,V,TS,TE). We are interested in the effect of various sort orderings on the efficiency with which it is possible to implement the temporal operators (listed in Figure 2) in the stream processing paradigm. We concentrate only on "inequality-temporal" operators such as the "during" operator; that is, the operators that have only inequalities in their explicit constraints[8]. We focus on how various sort

[7]Recall that TS and TE stand for ValidFrom and ValidTo respectively.

[8]For non-inequality constraints, an obvious stream processing method appears to be sorting both relations on attributes that are involved in the equalities followed by a conventional merge-join (and perhaps combined with filtering using inequality constraints.)

orderings would affect the size of local workspace required for the operations.

Before we proceed, we should note that the temporal operators listed in Figure 2 are in fact join and semijoin operations. Because of this, the only form of state information we need consider is subsets of the tuples previously read and not any summary information such as sum, max, avg, etc.

4.2.1 Contain-join

Contain-join(X,Y) outputs the concatenation of tuples X and Y if the lifespan of X contains that of Y; that is, "X.ValidFrom<Y.ValidFrom $\wedge$ Y.ValidTo<X.ValidTo" — the "during" relationship in Figure 2. Note that Contain-join(X,Y) and Contain-join(Y,X) are not logically equivalent unless X and Y are the same.

The join algorithm assumes that (1) there is an input buffer for reading tuples from each stream (denoted as <Buffer-x, Buffer-y>, and the tuples as x_b and y_b), (2) on the average, the ValidFrom (and ValidTo) values of two consecutive X tuples differ by $1/\lambda_x$ units of time (similarly, $1/\lambda_y$ for Y tuples.) The algorithm for the case when both relations X and Y are sorted on the attribute ValidFrom in ascending order as shown in Figure 5(a) is:

1. Initially there is no state tuple and the first tuple from each stream is read and stored in the buffer.

2. Read phase: copy x_b and y_b into the state space. Reading tuples from both streams depends on the ValidFrom values of x_b and y_b. The first case is when "y_b.ValidFrom <x_b.ValidFrom" as shown in Figure 5(b). As all Y tuples read so far do not join with x_b, more Y tuples should be read such that "y_b.ValidFrom$\geq x_b$.ValidFrom".

   The second case is when "y_b.ValidFrom$\geq x_b$.ValidFrom" as shown in Figure 5(c). The state of the current computation is:
   {X tuples whose lifespan span y_b.ValidFrom}
   $\cup$ {Y tuples whose ValidFrom value lies in l}.
   A tuple from an input stream which allows more state tuples to be discarded will be read. To estimate the number of disposable state tuples, $1/\lambda_x$ and $1/\lambda_y$ are used. If the next X tuple is read, disposable Y tuples are those which satisfy "x_b.ValidFrom $\leq$Y.ValidFrom$\leq \overline{t'}$ ", where the expected value of t' (denoted as $\overline{t'}$) is (x_b.ValidFrom + $1/\lambda_x$). Likewise, disposable X tuples are those which satisfy "y_b.ValidFrom $\leq$ X.ValidFrom$\leq \overline{t''}$ " when the next Y tuple is read, where $\overline{t''}$ is (y_b.ValidFrom+$1/\lambda_y$).

3. Garbage-collection phase: discard X tuples in the state if "X.ValidTo<y_b.ValidFrom". Also discard Y tuples if "Y.ValidFrom<x_b.ValidFrom". The garbage-collection condi-

tions must guarantee that the Y (and X respectively) tuples being discarded do not satisfy the join condition with any subsequent X (and Y respectively) tuples.

4. Join phase: output X and Y tuples if they satisfy the join condition.

5. The algorithm terminates if either stream has been exhausted and there is no corresponding state tuple. Otherwise, go to Step 2.

Note that the separation of this join algorithm into several phases is primarily for the sake of explanation; it is possible that Steps 2, 3 and 4 can be merged together to gain better performance. Also, the state can be characterized as follows: (1) When there is no Y tuple in the state, the maximal set of X tuples that are required to be kept in the state consists of all overlapping X tuples at time point y_b.ValidFrom. (2) Conversely, when there is no X state tuple, the maximal set of Y state tuples that is required consists of those whose ValidFrom value lie in the lifespan of x_b.

For the case when the relation X is sorted on the attribute ValidFrom and the relation Y is sorted on ValidTo in ascending order, the algorithm is similar to the above with the following exceptions:

1. Read phase: the Y tuples that can be discarded if an X tuple is read would be the same as above, but the disposable X tuples if the next Y tuple is read are those which satisfy "y_b.ValidTo$\leq$X.ValidTo$\leq y_b$.ValidTo$+1/\lambda_y$".

2. Garbage-collection phase: dispose of X tuples if "X.ValidTo$>y_b$.ValidTo", and dispose of Y tuples if "Y.ValidFrom$<x_b$.ValidFrom".

3. The state is {X tuples whose lifespan span y_b.ValidTo} $\cup$ {Y tuples whose lifespans are contained within l}.

We summarize the state information requirements of processing the Contain-join for other sort orderings in Table 1 along with the following remarks. Firstly, it is generally inappropriate to have one relation sorted in ascending order and the other in descending order. Secondly, sorting both relations X and Y on attribute ValidTo in descending order would have the same effect as sorting them on attribute ValidFrom in ascending order because of symmetry (although the ValidFrom and ValidTo attributes exchange their roles); the lower half of Table 1 is therefore the mirror image of the upper half. Because of this, we only show the upper half portion in subsequent tables.

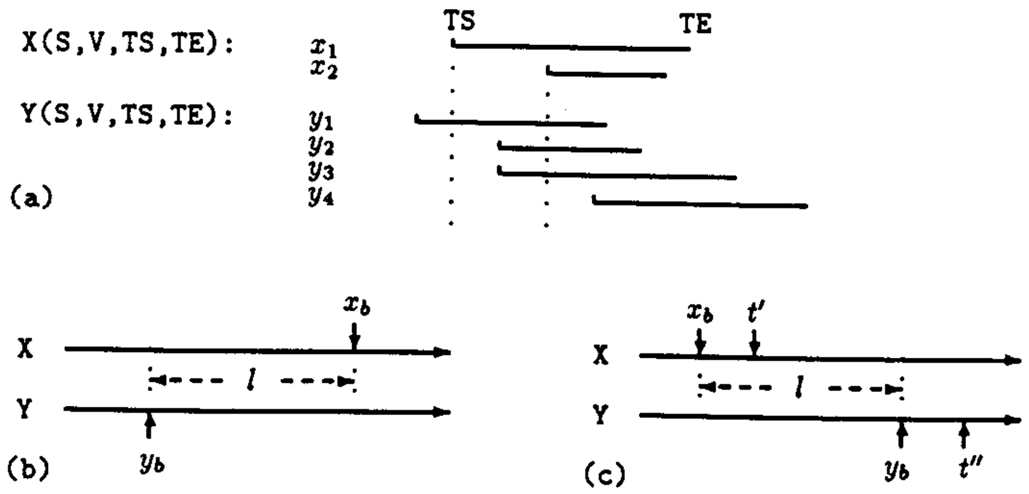


Figure 5: Contain-join: Both X and Y are sorted on TS in ascending order (only temporal attributes are shown)

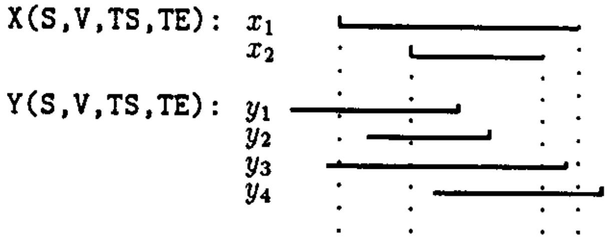


Figure 6: Contain-semijoins: X is sorted on TS and Y is on TE in ascending order

Sort Orders				Contain -join(X,Y)	Contain -semijoin(X,Y)	Contained -semijoin(X,Y)
Relation X		Relation Y				
ValidFrom	↑	ValidFrom	↑	(a)	(c)	(c)
ValidFrom	↓	ValidFrom	↓	-	-	-
ValidFrom	↑	ValidTo	↑	(b)	(d)	-
ValidFrom	↓	ValidTo	↓	-	-	(d)
ValidTo	↑	ValidFrom	↑	-	-	(d)
ValidTo	↓	ValidFrom	↓	(b)	(d)	-
ValidTo	↑	ValidTo	↑	-	-	-
ValidTo	↓	ValidTo	↓	(a)	(c)	(c)

↑ Sorting the corresponding attribute in ascending order.

↓ Sorting the corresponding attribute in descending order.

- The sort ordering is not appropriate for stream processing – no garbage-collection criteria.

(a) state = {X tuples whose lifespan span y_b.ValidFrom}
$\cup$ {Y tuples whose ValidFrom value lie in l}

(b) state = {X tuples whose lifespan span y_b.ValidTo}
$\cup$ {Y tuples whose lifespans are contained within l}

(c) state $\subseteq$ {X tuples whose lifespan span x_b.ValidFrom}
$\cup$ {Y tuples whose ValidFrom values lie in l}

(d) local workspace = <Buffer-x, Buffer-y>.

Table 1: Effect of various sort orders on Contain-join, Contain-semijoin & Contained-semijoin

4.2.2 Contained-semijoin & Contain-semijoin[9]

Contained-semijoin(X,Y) selects X tuples if there *exists* a Y tuple such that the lifespan of Y contains that of X. *Contain-semijoin*(X,Y) selects those X tuples whose lifespan contains that of any Y tuple. In a later section, we show that these semijoins can be used to reduce the local workspace required for join operations, and in particular, Contained-semijoin may be used to efficiently process the Superstar query.

For semijoins, a stream processor can output a tuple as soon as it finds the first matching tuple. Because of this, we devise an optimized algorithm which requires just one buffer for each input stream. Suppose the relation X is sorted on attribute ValidFrom and the relation Y is sorted on ValidTo in ascending order as shown in Figure 6. The algorithm for Contain-semijoin(X,Y) (and Contained-semijoin(Y,X) respectively) is as follows:

1. Read an X tuple and store it as x_b.
2. Read the next Y tuple and store it as y_b (the previous y_b is discarded) until one of the following holds:
   - "x_b.ValidFrom<y_b.ValidFrom $\wedge$ y_b.ValidTo<x_b.ValidTo" — i.e. x_b and y_b satisfy the semijoin condition, or
   - "y_b.ValidTo$\geq x_b$.ValidTo", or
   - all Y tuples have been read.

   If "y_b.ValidFrom$\leq x_b$.ValidFrom", immediately go to Step 2. On the other hand, if the semijoin condition is satisfied between x_b and y_b, output x_b. (For Contained-semijoin(Y,X), y_b is output if the condition is met and go to Step 2).

   It can be easily verified that only one Y tuple needs to be kept in the workspace. In Figure 6, when x_1 is fetched, the local workspace contains $<x_1, y_2>$ and for x_2 it is $<x_2, y_4>$.
3. Go to Step 1 unless the termination condition is met.

The local workspace requirements for other sort orderings are listed in Table 5.

[9]Similar to "restriction" operator in [Seg87].

Sort Orders				Overlap-join(X,Y)	Overlap-semijoin(X,Y)
Relation X		Relation Y			
ValidFrom	↑	ValidFrom	↑	(a)	(b)

(*) Other sort orderings are not appropriate and therefore they are not listed here.

(a) state = {X tuples whose lifespan span y_b.ValidFrom} ∪
{Y tuples whose lifespan span x_b.ValidFrom} ∪
{Y tuples whose ValidFrom value lie in l} if y_b.ValidFrom > x_b.ValidFrom
{X tuples whose ValidFrom value lie in l} if x_b.ValidFrom > y_b.ValidFrom.

(b) local workspace = <Buffer-x, Buffer-y>.

Table 2: Effect of various sort orders on the Overlap-join and Overlap-semijoin

Sort Orders on X		Contained-semijoin(X,X)	Contain-semijoin(X,X)
ValidFrom	↑	(a)	(b)
ValidFrom	↓	-	(a)

(a) the state is $\{x_s\}$ and Buffer-x for x_b.

(b) state(x_i) ⊆ $\{x_j \mid j > i \text{ and } x_j \text{ overlaps with } x_i\}$.

Table 3: Effect of various sort orders on the Contained-semijoin(X,X) and Contain-semijoin(X,X)

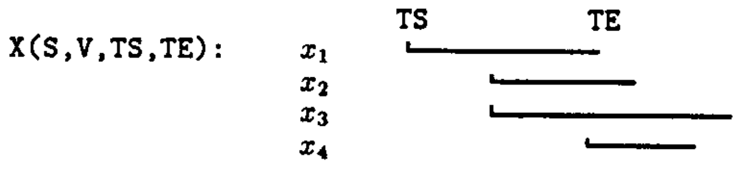


Figure 7: Relation X is sorted on attribute TS in ascending order

4.2.3 Contained-semijoin(X,X) & Contain-semijoin(X,X)

We discuss these semijoins again for the special case in which the two operands are the same stream of tuples. If we directly apply the semijoin algorithms presented previously, the operand stream may be scanned twice even when it is properly sorted. We are therefore interested in devising an efficient algorithm to avoid this inefficiency. It turns out that there is a stream processing algorithm which implements these operations by scanning the operand relation once and only two buffers are required (see Table 3). In the next section, we discuss the circumstances under which this algorithm could be used for the Superstar query.

We now present the algorithm for implementing Contained-semijoin(X,X) which can be modified slightly for Contain-semijoin(X,X). Recall that Contained-semijoin(X,X) selects each X tuple whose lifespan is contained within any other X tuple. Suppose the relation X has primary sort ordering on the attribute ValidFrom and secondary sort ordering on ValidTo in ascending order as shown in Figure 7. The algorithm is:

1. Read the first tuple from the stream and store it as the state tuple (denoted by x_s).
2. Read the next X tuple (x_b) and do :

```
if "$x_s$.ValidFrom=$x_b$.ValidFrom", replace $x_s$ with $x_b$ as the state tuple
else (i.e. "$x_s$.ValidFrom<$x_b$.ValidFrom")
    if "$x_s$.ValidTo$\leq x_b$.ValidTo", replace $x_s$ with $x_b$ as the state tuple
    else (i.e. $x_b$'s lifespan is contained within that of $x_s$) $x_b$ is output
        and $x_s$ remains as the state tuple.
```

3. Repeat Step 2 until all tuples have been read.

Referring to Figure 7, the semijoin algorithm starts by reading tuple x_1 which is kept as the state tuple. When x_2 is read, it becomes the state tuple, and similarly for x_3. Then x_4 is read which satisfies the join condition and therefore is output; x_3 remains in the state. This process stops when there are no more tuples in the stream. The maximum number of state tuples remains at most one. That is, the local workspace contains the state tuple (x_s) and a buffer of the tuple just read (x_b).

It is interesting to consider using a semijoin algorithm as a preprocessor for a join operation. Intuitively, the advantages are: (1) the output stream from a semijoin operation has the same sort ordering as the input stream — *order-preserving*; (2) with proper sort orderings, the semijoin algorithms scan input streams only once, and a number of "dangling" tuples may be eliminated, which may reduce the size of workspace for join operations.

4.2.4 Overlap-join, Overlap-semijoin, Before-join & Before-semijoin

In this section, we briefly consider Overlap and Before operators to complete our discussion on "inequality-temporal" operators. Again, we consider the overlap operator used in the Superstar example. The effect of various sort orders on the overlap operator are listed in Table 2, which shows that the only cases in which stream processing is efficient are when (1) both operands are sorted on the attribute ValidFrom in ascending order, or (2) both operands are sorted on the attribute ValidTo in descending order.

We mentioned earlier that the best approach for implementing Before-join appears to be the nested-loop join. It is easy to verify that there is no sort ordering that would significantly limit the amount of state information required when the Before-join is implemented by a stream processor. However, we do not mean to imply that sorting is useless for nested-loop joins; with proper sort orders, nested-loop join can avoid scanning the inner relation in its entirety. For Before-semijoin, one can easily devise a simple algorithm which scans both operand relations only once and is independent of any sort orderings; we omit the detail for brevity.

5 Semantic Query Optimization

Semantic query optimization techniques have been introduced and shown to be potentially useful in many studies [Kin81, Cha84, Jar84, She87]. However the technique has not been widely used in conventional systems. The reason, we speculate, might be that conventional application domains are seldom rich enough in semantics, i.e. they contain only a few useful semantic constraints which the query optimizer can profitably exploit. For temporal databases, time is unarguably rich in semantics and many temporal semantic properties/constraints do occur naturally. It is therefore our belief that, unlike conventional applications, semantic query optimization can play a significant role in temporal databases. In this section, we discuss informally the significance of semantic query optimization in temporal query processing; its formal treatment is now underway.

Earlier we mentioned an interesting integrity constraint in the Faculty relation, namely the chronological ordering of data values which the attribute Rank can assume — 'Assistant', 'Associate' and 'Full'. For every faculty, being an assistant professor must occur before being promoted to an associate professor, which must then occur before becoming a full professor.

There are two consequences if the database system does not capture and use this constraint. First, and most important, the optimizer would not be able to recognize that the less-than join in the Superstar example is in fact a Contained-semijoin. The less-than join operation shown in Figure 3(b) can be described pictorially using Figure 8. The equi-join on "f1.Name=f2.Name" (in Figure 3(b)) concatenates those f1 and f2 tuples corresponding to those assistant professors promoted to full professors. The less-than join then selects those f1 and f2 tuple pairs which satisfy the less-than join condition (θ') as shown in Figure 8(a). With the above semantic constraint, it is not difficult to see that

$$\text{f1.ValidFrom} < \text{f3.ValidTo and f3.ValidFrom} < \text{f2.ValidTo}$$

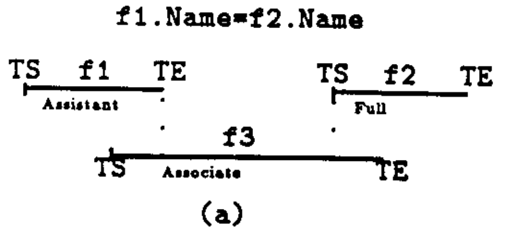
are redundant and the less-than join condition can be reduced to a Contained-semijoin condition as shown in Figure 8(b). Being able to recognize a Contained-semijoin allows the database system to make use of sort orderings and therefore the stream processing technique mentioned in the previous section.

Taking this example one step further, suppose that there is no re-hiring of faculty members, e.g. no assistant professors left the university and then later were re-hired as full professors. That is, in Figure 1 "$\text{ValidTo}_1=\text{ValidFrom}_2$" and "$\text{ValidTo}_2=\text{ValidFrom}_3$" are always true. In addition, suppose that all faculty members are hired as assistant professors. With this continuous employment assumption, the Superstar query can be transformed into: ***List associate professor X if there exists another associate professor Y such that X is promoted from assistant professor level later than Y, but X is promoted to full professor rank earlier than Y.*** The relational algebraic expression for this query can be simplified into:

$$\pi_{\text{i.Name,i.ValidFrom,i.ValidTo}}(\text{Contained-semijoin}(\sigma_{\theta''}(\text{Faculty}_i), \sigma_{\theta''}(\text{Faculty}_j)))$$

where $\theta'' \equiv$ 'Rank=Associate'. As shown in Figure 8(b), the period [f1.TE, f2.TS) is actually the time during which the faculty member is at the associate professor level. When the associate professor tuples are sorted on the ValidFrom attribute in ascending order (or we explicitly sort on this attribute), the algorithm discussed in section 4.2.3 can be used to perform the semijoin which requires only a single scan of tuples (i.e. all associate professor tuples) and the local workspace is composed of only a state tuple and an input buffer. For this particular query, the stream processing algorithm can be extremely efficient.

The second consequence of the constraint on the Rank attribute is that we are able to eliminate two redundant inequalities in θ'; their presence makes it harder to recognize the join as Contained-semijoin and there is also some overhead due to testing redundant qualification. Eliminating redundant qualifications is indeed a by-product of semantic query optimization.



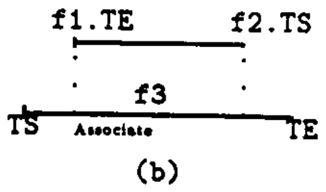


Figure 8: (a) The less-than join in the Superstar query, and (b) its equivalent Contained-semijoin condition after semantic optimization

6 Conclusions & Future Work

We have illustrated deficiencies of conventional systems for temporal query processing using the complex Superstar query. This example leads to several observations which suggest new requirements for temporal query processing strategies. The most interesting and important observation is that less-than joins occur more often and naturally in temporal queries, and usually contain a conjunction of a number of inequalities. For the Superstar example, it may be more efficient to implement the less-than join using Contain-semijoin instead of using nested-loop join algorithm especially when tuples are properly sorted. These observations motivate our investigation of the stream processing strategies, opening up many new avenues of research in temporal query optimization techniques.

We have considered stream processing techniques for processing various temporal join and semijoin operators. Given data integrity constraints and a temporal query, we discussed the effect of various sort orderings of streams of tuples on the efficiency with which the operator is implemented and the local workspace requirement in the stream processing environment. In particular, we note that the optimal sort order may depend on the query itself and the statistics of data instances.

We have also discussed semantic query optimization in temporal databases. In temporal databases like TQuel [Sno87], relations are augmented with temporal attributes such as ValidFrom and ValidTo. Users are not allowed to update these attributes directly although a set of temporal operators are provided for data manipulation. From an algebraic manipulation point of view, these system-defined attributes are the same as any user-defined attributes. The main difference becomes evident when the semantics of ValidFrom and ValidTo attributes are utilized in the semantic query optimization process. As we can see from the Superstar example, the system might not be able to evaluate the query using Contained-semijoin without knowing the "intra-tuple" integrity constraint. We note that time is rich in semantics and temporal data often implies ordering by time, it is not surprising that semantic optimization can play a significant role in temporal databases.

There are many directions for future research. We are currently pursuing the following areas: a complete temporal data model, statistical information gathering and formalizing semantic query optimization in temporal databases. In this paper, we rely on the concept of a Time Sequence as our basic data construct. In the TQuel data model [Sno87], two other temporal attributes (TransactionStart and TransactionStop) can be augmented to relational tables to capture the 'rollback' capability. Moreover, a temporal relation may naturally have multiple time-varying attributes such as Rank and Salary. We are extending our data model to incorporate these features so that queries can be evaluated effectively in the stream processing paradigm.

Statistical information about the database is known to be important in query optimization. For temporal databases, it appears to be more critical. In addition to conventional statistical information such as relation size and image size of indices, estimating the amount of local workspace becomes necessary. There is also a question of how this information can be obtained efficiently and summarized in a suitable form for the optimizer.

As we mentioned earlier, we are now working on formulating the semantic optimization, particularly formalizing the role of chronological ordering of data items in query optimization.

Acknowledgement

The authors are grateful to Tom Page, Hanan Samet and C.D. Shum for improvements they suggested on the manuscript.

References

[Abe85] Abelson H. and Sussman G.J. 'Structure and Interpretation of Computer Programs,' The MIT Press, 1985.

[Ahn86] Ahn I. 'Towards an Implementation of Database Management Systems with Temporal Support,' *Proceedings of the Int. Conf. on Data Engineering*, February 1986, pp.374–381.

[All83] Allen James F. 'Maintaining Knowledge about Temporal Intervals,' *Communications of the ACM*, Vol.26, No.11, November 1983, pp.832–843.

[Bat88] Batory D.S., Leung T.Y. and Wise T.E. 'Implementation Concepts for an Extensible Data Model and Data Language,', *ACM Trans. on Database Systems*, Vol.13, No.3, September 1988, pp.231–262.

[Ben82] Ben-Zvi J. 'The Time Relational Model,' (Unpublished) PhD Thesis, Dept. of Computer Science, University of California, Los Angeles, 1982.

[Bent79] Bentley J. 'Algorithms for Reporting and Counting Geometric Intersections,' *IEEE Trans. on Computers*, Vol.C-28, No.9,September 1979, pp.643–647.

[Cha84] Chakravarthy U.S., Fishman D.H. and Minker J. 'Semantic Query Optimization in Expert System and Database Systems,' *Expert Database Systems*, 1984, pp.326–341.

[Cli83] Clifford J., and Warren D.S. 'A Model for Historical Databases,' *ACM Trans. on Database Systems*, Vol.8, No.2, June 1983, pp.214–254.

[Cli85] Clifford J., and Tansel A. 'On an Algebra for Historical Relational Databases: Two Views,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1985, pp.247–265.

[Cli87] Clifford J., and Croker A. 'The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans,' *Proceedings of the Int. Conf. on Data Engineering*, February 1986, pp.474–481.

[Gad88] Gadia S.K. 'A Homogeneous Relational Model and Query Languages for Temporal Databases,' *ACM Trans. on Database Systems*, Vol.13, No.4, December 1988, pp.418–448.

[Jar84] Jarke M. 'External Semantic Query Simplification: A Graph-theoretic Approach and its Implementation in Prolog,' *Expert Database Systems*, 1984, pp.467–482.

[Kin81] King J.J. 'QUIST: A System for Semantic Query Optimization in Relational Databases,' *Proceedings of the 12th Int. Conf. on Very Large Data Bases*, August 1981, pp.510–517.

[Lum84] Lum V., et. al. 'Designing DBMS support for the Temporal Dimension,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, June 1984, pp.150–130.

[McK86] McKenzie E. 'Bibliography: Temporal databases,' *ACM SIGMOD Record*, Vol.15, No.4, December 1986, pp.40–52.

[Ore88] Orenstein J. 'PROBE Spatial Data Modeling and Query Processing in an Image Database Application,' *IEEE Trans. on Software Engineering*, Vol.14, No.5, May 1988, pp.611–629.

[Par89] Parker D.S. 'Stream Data Analysis in Prolog,' Dept. of Computer Science, Technical Report CSD-890004, January 1989.

[Pre85] Preparata F.P. and Shamos M.I. 'Computational Geometry: An Introduction,' Springer-Verlag, 1985.

[Rot87] Rotem D. and Segev A. 'Physical Design of Temporal Databases,' *Proceedings of the Int. Conf. on Data Engineering*, February 1987, pp.547–553.

[Seg87] Segev A. and Shoshani A. 'Logical Modeling of Temporal Data,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1987, pp.454–466.

[Sel79] Selinger P. et. al. 'Access Path Selection in a Relational Database Management System,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1979, pp.23–34.

[She87] Shenoy S.T. and Ozsoyoglu Z.M. 'A System for Semantic Query Optimization,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, June 1987, pp.181–195.

[Shi81] Shipman D.W. 'The Functional Data Model and the Data Language DAPLEX,' *ACM Trans. on Database Systems*, Vol. 6, No. 1, March 1981, pp.140–173.

[Sho86] Shoshani A. and Kawagoe K. 'Temporal Data Management,' *Proceedings of the 12th Int. Conf. on Very Large Data Bases*, August 1986, pp.79–88.

[Smi75] Smith J. and Chang P. 'Optimizing the Performance of a Relational Algebra Database Interface,' *Communications of the ACM*, Vol.18, No.10, October 1975, pp.568–579.

[Sno85] Snodgrass R., and Ahn I. 'A Taxomony of Time in Databases,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1985, pp.236–246.

[Sno87] Snodgrass R. 'The Temporal Query Language TQuel,' *ACM Trans. on Database Systems*, Vol. 12, No. 2, June 1987, pp.247–298.

[Tan88] Tansel A. 'Non First Normal Form Temporal Relational Model,' *IEEE Data Engineering*, Vol. 11, No. 4, December 1988, pp.46–52.

[Ull82] Ullman J.D. 'Principles of Database Systems,' 2nd edition, Computer Science Press, Rockville, Md., 1982.