QUERY PROCESSING & OPTIMIZATION TECHNIQUE FOR TEMPORAL DATABASES

[1]Mir Abid Ali, [2]Dr. Syed Abdul Sattar
[1]Research Scholar, Department of Computer Science Engineering, PP.COMP.SCI.0578, Rayalaseema University, A.P, India.
[2] Director, Research & Development, Nawab Shah Alam Khan College of Engineering & Technology, Hyderabad.

Abstract: Conventional relational systems are frequently incident for temporal query processing for new characteristics are not taken into concern. As an example, a temporal join often contains a conjunction of several inequalities involving only timestamps. With a conventional relational query processor, this type of query is processed using the nested-loop join algorithm, which may not be the most efficient method. This paper explains a stream processing approach for temporal query processing and improvement. Given appropriately arranged information, the usage of temporal joins and semi-joins as stream processors can be exceptionally powerful. We also talk over the tradeoffs between sort orders, the amount of local workspace and multiple scans over input streams; in particular, we are interested in the effect of sort ordering on the local workspace requirement. We present stream processing algorithms for various temporal joins and semi-joins, and their workspace requirements for various data sort orderings. Finally, we communicate how semantic query optimization can show a vital and natural role in optimization of temporal queries.

IndexTerms - Temporal query processing, temporal queries, temporal joins.

I. INTRODUCTION

Many real database applications intrinsically involve time-varying information. With the availability of cheap processing and storage units, there is a developing interest in temporal databases which store the evolving history of the "enterprise" of interest. Maximum studies on temporal databases can be roughly categorized into three areas [3]. The first is the formulation of the semantics of time [3, 9] and is closely related to research issues in knowledge representation. The second area concerns physical implementation issues [2, 3]; the focus is mainly on new access methods and data organization strategies. The third area is the logical modeling of temporal data [5]. Many of these studies emphasize distribute 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 time attributes (such as ValidF'rom and ValidTo attributes [5]) which store the relevant timestamps. New temporal operators are also defined in these extended data models, based upon traditional relational algebraic operators [6], to allow users to query time attributes but not update them directly.

'This work was done under the Tangram Project, supported by DARPA contract F29601-87-C0072.

This paper explains the query processing and optimization for temporal databases, a topic which is seldom discussed in the literature (a notable exception is [8]). We observe that there are numerous interesting characteristics which are peculiar to temporal queries: (1) a temporal query often involves patterns of events; (2) a temporal query often contains a conjunction of several inequalities over the time domain and no equality conditions; and (3) temporal data is rich in semantics, and semantic query optimization is particularly desirable in the existence of a number of inequalities. These characteristics provide new opportunities for optimization. Ignoring these, as in conventional relational systems, can result in poor performance.

We discuss join and semijoin operations which are the most common and expensive computations in database systems. We introduce 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 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 [6]. These studies share the basic principle of the stream processing paradigm which is that input data should be in a certain order before the processing commences. This paper explanations are more concerned with (1) the impact of various data orderings 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. [10] presents a more detailed comparison with related work which is omitted herein consideration of space limitations.

One might argue that stream processing algorithms are *not* useful when data are not sorted properly. With an abundant amount of main memory and processing cycles available, one can sort the input streams "on the fly" with marginal additional cost as assumed in [10]. The algorithms we describe would still be applicable although the streams are actually memory resident.
We informally discuss the role of semantic query optimization in Section 5, and finally, conclude with instructions for upcoming work.

2 IMPLEMENTATION AND CHALLENGES OF TEMPORAL DATABASE

Few out of every odd database requires a temporal database implementation, however, some do. We can enable you to begin. As examined in our past article, the SQL-2011 standard included clauses for the meaning of temporal tables as a component of the SQL/Foundation. In any case, this standard is new and not yet broadly adopted, For the time being, the majority of you should stretch out your current database tables to fuse temporal concepts. In this article, we'll concentrate on temporal tables. These tables are the building blocks of temporal databases.

Temporal Tables – The Important Theories

Theory #1: Valid-Time State Tables

From Wikipedia: "Valid time is the time period during which a reality is valid with respect to the real world." A valid time state table gives a chance to manage data whose values change over time. A valid time state table lets you manage data whose values change over time. For a sample, the interest rate on a loan can be 5% for the first year, and 6% for the second year. During the second year, you still want to know that the rate was 5% in the previous year.

Theory #2: Transaction-Time State Tables

Semantic query optimization has been discussed in the literature [8] but apparently has not been widely used in conventional systems. Semantic constraints in temporal databases occur more naturally and are more plentiful, and consequently, a query optimizer should profitably exploit the semantics. We will briefly discuss the role of semantic query optimization in temporal databases after discussing more basic query optimization issues.

The rest of the paper is structured as follows. Section 2 provides an outline of the temporal data model that we adopt from [6] and discusses the basic categories of temporal queries. We illustrate, in Section 3, the conventional approach to processing a complex temporal query. In the 4th Section, we discuss a stream processing approach for the implementation of temporal operators.

Theory #2: Transaction-Time State Tables

From Wikipedia: "Transaction time is the time period through which a fact deposited in the database is considered to be true." When you effectively catch the sequence of states for an evolving table, you have made a substantial transaction-time state table. The tables independent from anyone else now contain the expected data to return in time or to "rewind" to a specific minute and see the information that was substantial right then and there.

Theory #3: BiTemporal Tables

From Wikipedia: "Bitemporal data combines both Valid and Transaction Time." Valid-time state tables and transaction-time state tables are orthogonal. You don't have to implement both at once; you can do one or the other, and even if you do both, you don't have to keep the information in a single table. For example, it is common to find designs where the transaction information is stored in different tables. In such cases, only the latest valid information is stored in one table while the other table contains the historical records.

On the off chance that you choose to make a table both a valid-time and a transaction-time table, at that point, you have made a bitemporal table. Utilizing Snodgrass' words, "a bitemporal table permits a magnificent expressiveness while investigating and separating data" from

such tables. Step by step instructions to Implement a Temporal Table:

How to Implement a Temporal Table:

In the accompanying three areas, we quickly depict what you have to do with your tables to actualize a temporal database.

We will allude to the Hudson Foods hamburger review referenced in our first article. This is a precedent given in Richard Snodgrass' book, outlining an unmistakable instance of an association enduring noteworthy budgetary punishments because of not having a time-varying database. This model includes following dairy cattle between various pens, and how that information changes after some time. The data beneath is a super disentanglement of the model incorporated into Chapter 2 of the book.

In a feed yard, cattle are grouped into "lots". Cattle from one lot can be partitioned into numerous pens. We characterize a table LOT_LOC that tracks what number of cattle from each part live in each pen of each feed yard. cattle from each lot are moved from pen to pen, in this way the information is changing after some time.

LOT_LOC(LOT_ID_NUM, PEN_ID, HD_CNT, FROM_DATE, TO_DATE)

How about we center around two segments: FROM_DATE and TO_DATE.

These two columns render the table a "valid time state table": it records information valid at some time in the modeled reality, and it records states; that is, facts that are true over a period of time. The FROM_DATE and TO_DATE sections delimit the "valid time" or "period of validity" of the data in the line. A key idea in this structure is thinking about your "temporal granularity." You should pick the correct granularity for your concern. This could be days, hours, minutes, seconds, or whatever you require as a granularity level. For the past table, the granularity level is multi-day.

Think about the accompanying three lines:

(100, 3001, 32, 2014-06-01, 2014-06-02)

(100, 3001, 30, 2014-06-02, 9999-12-31)

(100, 3002, 02, 2014-06-02, 9999-12-31)

We can tell the accompanying from the information: from June first, 2014, to June second, 2014, there were 32 cows from part 100 in pen 3001; on June second, 2014, the cattle from lot 100 were part in two unique pens. Thirty cows stayed in pen 3001, and two cows were moved to pen 3002.
This is the latest valid data. The primary column is presently invalid for the current date. As should be obvious, we can proceed to include and change this information. It will be conceivable to question the information with the end goal that we see a real perspective of the information as it existed at a specific time. We could compose questions that would enable us to follow back every one of the pens containing cows from a specific lot before or on a specific date.

The Design Works, Be that as it may, it's entangled.

This plan would have understood Hudson Foods information issues and would have brought about a more engaged and lessened sustenance review. This plan idea can be utilized to execute temporal concepts into any standard database table. There are, nonetheless, difficulties to this plan, which we will talk about in the following article.

Few out of every odd database requires a temporal database implementation, yet some do. We can enable you to begin. As discussed in our past two articles, the SQL-2011 standard included conditions for the meaning of temporal tables as a feature of the SQL/Foundation. Be that as it may, this standard is new and not yet broadly embraced. For the present, the vast majority of you should stretch out your present database tables to consolidate temporal concepts. We have demonstrated to you that expanding your present database tables is moderately simple.

In this article, we'll concentrate on the difficulties in broadening your current tables into temporal tables, and why executing a genuine temporal database is less demanding.
The Challenges of Implementing a Temporal Table

Challenge-1
SQL comes up short on a time-range data type, so you need to utilize two diverse time columns to capture that data. As a result, both time columns must turn out to be a piece of the essential key, so they shouldn't be permitted to be NULL. (You could utilize NULL, yet in specific cases, you may get stuck in an unfortunate situation. For instance, read this blog: The Index You've Added is Useless. Why? To stay away from a NULL value, the TO_DATE was given a non-null value of 9999-12-31 for lines that are as of now legitimate in time.

Why a temporal database is less demanding - > temporal databases bolster a time-range data type. The presence of this time extends deliberation rearranges the structure. Contemplations, for example, the likelihood of a NULL in the from/to essential key column, are not any more pertinent.

Challenge-2
Without the FROM_DATE and TO_DATE columns, the essential keys would be LOT_ID_NUM and pen once you include the FROM_DATE and TO_DATE columns, the primary key must incorporate four columns: FROM_DATE, TO_DATE, LOT_ID_NUM, and PEN_ID. A same lot and pen could exist in various focuses in time; they are not anymore one of a kind for each line, and you have to incorporate the time run with the end goal to look after uniqueness. Characterizing the FROM_DATE and TO_DATE segments as a major aspect of the essential key is a test with non-temporal databases. In a non-temporal database, the semantic significance presently is something like "LOT_ID_NUM, PEN_ID, FROM_DATE, and TO_DATE are the exceptional identifiers".

Why a temporal database is less demanding - > A temporal database would enable you to characterize just LOT_ID_NUM and PEN_ID as the primary keys, which would have the semantic importance of "LOT_ID_NUM and PEN_ID are the remarkable identifiers at any minute in time."

Challenge -3
All inquiries must be time-mindful. Regardless of whether you just need to inquire the last substantial information in time, despite everything you need to add a WHERE clause to ensure you get only the most recent information. As a rule, every one of your inquiries will presently be somewhat more intricate (or much more unpredictable).

Why a temporal database is less demanding - >
On the off chance that you were utilizing a temporal database, your tables would follow time-varying legitimacy naturally. You would inquiry the most recent legitimate information (without a WHERE proviso) as a matter of course.

You additionally would gain admittance to other decent highlights, for example, "time breakpoints." For instance, you could set a breakpoint at 2012-12-01, and starting there on the entirety of your inquiries would be for information that was legitimate until 2012-12-01 just; for instance, "select the pens that contained cattle from part 55 preceding 2012-12-01". Your breakpoint could likewise be a period go like TIME_RANGE (2012-11-01, 2012-12-01); for instance, "select the pens that contained dairy cattle from part 55 between 2012-11-01 and 2012-12-01".

Challenge-4

Every one of your updates is currently more unpredictable. Each refresh now includes evolving (at least one) existing row(s) and making (at least one) new row(s).

Why a temporal database is easier

A temporal database will consequently guarantee that once a row is refreshed its earlier data isn't lost. In a temporal database, you are not by any stretch of the imagination refreshing columns yet rather changing their time legitimacy.

Challenge-5.

In a temporal table, you don't erase information. You change the time range with the end goal to demonstrate that the information isn't substantial any longer. On the off chance that distinctive clients and applications are getting to your tables, it is hard to control and check that everyone regards the presumptions for this table.

A temporal database will guarantee that all clients/applications executing inquiries against the "current valid time" will get precise outcomes. Invalid/erased information won't be returned.

A temporal database will naturally authorize the principles for a DELETE task to guarantee earlier information isn't lost, paying little heed to the customer.

3 TEMPORAL DATA MODEL

In our temporal data model, we consider time as a sequence of discrete, consecutive, equally-distanced points, i.e. Time = *{to, t l,. . .,* 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 [11]' as the basic data construct in our temporal data model. A temporal data value is a 4-tuple <S, V, ValidFrom, ValidTo>

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)2. 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) Together with the following integrity constraints

and assumptions, this example is used in subsequent sections for illustration purposes. The name is the identity of a faculty member. For attribute Rank, we consider only three different ranks - 'Assistant', 'Associate' and 'Full'.

We will assume in this example that an assistant professor can be promoted only to an associate professor and then to 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, UValidTol<ValidFrom2" and "ValidTozSValidRoms" must hold. The period [ValidFrom, ValidTo) in a tuple is the time during which the faculty member 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.

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

Fig-1. A Sample Faculty Relation

'A Time Sequence is a totally ordered sequence of temporal data values <Surrogate, Attribute-value, Time>. For continuous Time Sequence, the attribute due of an object between any two-time points (i.e. between consecutive temporal data values) can be computed using an interpolation function.'We consider only the valid times in TQuel temporal database taxonomy [7]. Also, for simplicity, we often use TS (which stands for Time Start) to abbreviate ValidFrom, and TE (which stands for Time End) to abbreviate ValidTo. A stepwise-constant interpolation function is applied between the lime points ValidFrom and ValidTo. V can be generated as a list of attributes.'Borrowed from 1321. Also, the relation is in TNF

4 CONVENTIONAL APPROACHES

In this section, we describe the deficiencies of conventional relational database systems in processing temporal queries. Allen [3] presents thirteen elementary temporal operators of time-intervals which are listed in Figure 2. These temporal operators are actually just syntactic sugar for the 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 are then optimized and processed by conventional relational query processors. This approach is generally not effective for temporal query processing as we demonstrate below.

Suppose we have a relation Faculty (Name, Rank, ValidFrom, ValidTo) as described in the previous section. Consider the following
Quel query modified from [3214: *Superstar - Who got* promoted from assistant to full professor while at least one other faculty remained at the associate rank?

```
        Range of fl is Faculty
        Range of f2 is Faculty
        Range of f3 is Faculty
 Retrieve into Stars (Name=fl.Name,ValidRom=fl.ValidFrom,
                ValidTo=B.ValidTo)
        Where fl.Name=B.Name and f3.Rank="Associate"
        And fl.Rank="Assistant" and f2.Rank="Full"
        And (fl overlap f3) and (f2 overlap f3)'
```

These "overlap" operators can be translated directly into equivalent
Clauses (e.g. in Quel) involving inequalities. That is,

```
(fl overlap f3) = fl.ValidFkom<f3.ValidTo
                  ^fJ.ValidFrom<fl.ValidTo
(0 overlap f3) = f2.ValidFrom<f3.ValidTo
                  ^f3.ValidFrom<f?.ValidTo
```

'The original TQuel query in is:
The range of fl is Faculty
The range of f2 is Faculty
Range of a is Associate
Retrieve into Stars(Name=fl.Name) Valid from beginning of fl to begin of E!
Where fl.Named2.Name When (fl overlap a) and (f2 overlap a)
and fl.Rank="Assistant" and f2.Rank="Full" 'This overlap operator defined in [11] is different from "overlaps" in [3]; 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 [11].

Temporal Operators	Time axis →	Explicit Constraints
(1) X equal Y	xxxx yyyy	X.TS=Y.TS∧X.TE=Y.TE
(2) X meets Y	xxxyyy	X.TE=Y.TS
(3) X starts Y	xxxx yyyyyyy	X.TS=Y.TS∧X.TE<Y.TE
(4) X finishes Y	xxxx yyyyyyy	X.TE=Y.TE∧X.TS>Y.TS
(5) X during Y	xxxx yyyyyyy	X.TS>Y.TS∧X.TE<Y.TE
(6) X overlaps Y	xxxx yyyy	X.TS<Y.TS∧X.TE>Y.TS ∧X.TE<Y.TE
(7) X before Y	xxx yyy	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

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 ∧ f1.Rank="Assistant"
∧ f2.Rank="Full" ∧ f3.Rank="Associate" ∧ θ'
θ' is f1.ValidFrom<f3.ValidTo ∧ f3.ValidFrom<f1.ValidTo
∧ f2.ValidFrom<f3.ValidTo ∧ f3.ValidFrom<f2.ValidTo

This algebraic expression can be represented as a parse tree[9], as depicted in Figure 3(a). The parse tree can then 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. 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, we are looking for a pattern composed of three-tuples - an assistant professor, a full professor and an associate professor. 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.

2. 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, a scaled less-than join, is a Cartesian product charted by a selection with the condition being a conjunction of inequality predicates - 6' '. Traditionally, the best strategy for processing less-than joins appears to be the
'Note that range search (e.g. salary > 10K and salary < 20K) is different
from this form of query qualification.

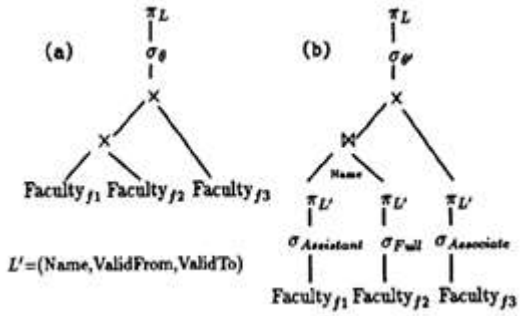

Figure-3 (a) Parse tree for the superstar expression (b) its optimized version

Conventional nested-loop join method. With only a single inequality as the join condition, we have no optimal but the nested-loop join method. Since time points are totally ordered and the join condition is a concurrence of several inequalities over time domain, one might wonder if there are any more efficient processing alternatives. In the past, little courtesy has been given to this form of qualification because: in traditional database applications, queries seldom contain less-than joins, and when inequalities do occur, in most situations the join condition has d y a single inequality predicate; for example, in an Employee/Department database, we might want to retrieve employees who earn more than their manager. The situation is quite different when we consider temporal databases: less-than joins appear more frequently and naturally because temporal queries often involve patterns of events, and therefore less-than joins need to be *ex*plicitly considered in query optimization, the join condition often contains a conjunction of several Inequality predicates which further indicates that optimization might be possible. Recall that there is an integrity constraint in the Faculty relation: a chronological ordering of data dues – 'Assistant', 'Associate' and 'Full'. This ordering implies that being an assistant professor must occur before being promoted to a full professor i.e. "fl.ValidTo<f2.ValidFrom" always holds in the presence of (fl.Name=D.Name).

These constraints, together with the "intra-tuple" integrity constraints,

$$fi.ValidFrom<fi.ValidTo \text{ for } i=1,2,3$$

imply "fl.ValidTo<f2.ValidFrom" and "f3.ValidFrom< f2. ValidTo". Therefore these inequalities in 6' 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 each department: emp is the employee, $dept_h$ is the department that the employee works with, salary is the employee's salary

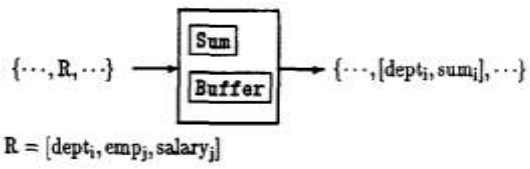

Figure 4: A stream processor to sum all employees' salaries in

The above Observations suggest that, in addition to traditional set-oriented relational operators, we may need other alto process temporal queries. In subsequent sections, we will present and discuss a number of such alternatives.

4 STREAM PROCESSING APPROACH

We discuss a stream processing approach for temporal query processing in this section. Algorithms that implement temporal operators are presented. The tradeoffs between sort orders, the amount of indigenous workspace and multiple passes over input streams are discussed. For properly sorted streams of tuples, we show that temporal operators 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 demarcated as an ordered arrangement of data objects [1,10]. Stream processing resembles the notion of data flow processing in database systems [4,7]. A classic example of stream processing is the merge-join; it is efficiently executed and the sort ordering of its output can be utilized by subsequent operations [10].
There are several intrinsic characteristic s 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 keep some local state information in order to avoid multiple readings of streams. The state information 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, min, count 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. For example, a simple stream processor, which is depicted in Figure 4, lists all the departments and computes the total salaries of all employees in each department. The point here is that the state contains summary information (i.e. the partial sum), and the function (i.e. sum) is expressed in terms of the current state and an input object.
Third, there are often tradeoffs among the following factors:
(1) The minimal size of the local workspace which depends on the function itself and the statistics of a specific instance of the data stream,
(2) Sort order of input streams, and
(3) Multiple passes over input streams.

Very often stream processing requires input streams to be properly sorted in order to perform the computation while only reading the input streams once. In addition, the sort orderings 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. In the next section, we discuss the application of stream processing algorithms to temporal databases. 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. Because of space limitation, we concentrate on the "during" relationship which has only inequalities in its explicit constraints? The implementation of Overlap and before operators in the stream processing paradigm are discussed in [16]. Before we proceed, we should note that the temporal operators listed in Figure 2 are in fact join and semi join operations. Because of this, the only form of state information we need consider in subsets of the tuples previously read and not any summary information such as sum, min, etc.

4.2.1 Contain-join(X ,*Y*)

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 *h* Y.ValidTo<X.ValidTo" - i.e. the "during" relationship in Figure 2. The generic algorithm for Contain-join(X,Y) is shown in Figure *5*. The specific instance of this generic algorithm depends on the sort orderings of streams. In this paper, we present the Contain-join algorithm in more detail for the case when both relations X and Y are sorted on the attribute ValidFrom in ascending order (see Figure 6(a)). The following conventions and notations are used in the algorithm:

(1) there is
an input buffer for reading tuples from each stream (denoted
as <Buffer-x, Buffer-y>), and the tuples in these buffers are denoted as Xb and Yb respectively;
(2) the expected difference
between ValidFrom values of two consecutive X tuples is *r*, (similarly, *RV* for Y tuples), and
(3) *2* denotes the absolute value of
the difference between yb.validFrom and xb.ValidFrom.

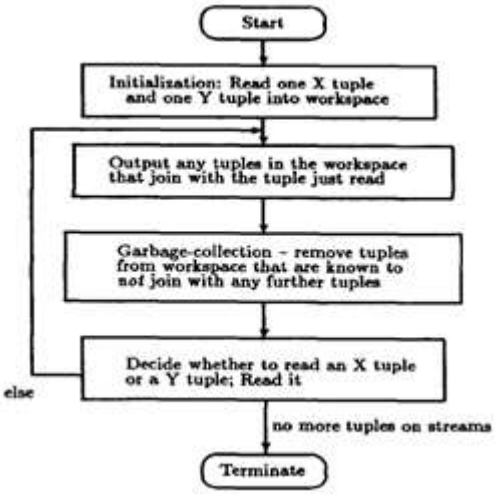

Figure-5 Generic Join Algorithm

'Recall that TS and TE stand for ValidFrom and ValidTo respectively. 'For a temporal operator with equality predicate(s), 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 other predicates of the operator.)

Contain-join(X,Y): [9]

1 Initially the first tuple from each stream is read and stored in the buffer.

2 Join phase: Output any tuples in the workspace (i.e. the state) that join with the tuple just read. Note that additional housekeeping must be done for maintaining the output sort ordering.

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 conditions 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 Read phase: Copy the previously read tuple(s) into the workspace as state tuple(s). There are two different situations in deciding which stream of tuples is to be read. The first case is when "y_b.ValidFrom<x_b.ValidFrom" as shown in Figure 6(b). As all Y tuples read so far do not join with x_b, clearly the optimal step is to read the next Y tuple.

The second case is when "y_b.ValidFrom≥x_b.ValidFrom" as shown in Figure 6(c). The workspace contains
(1) X tuples whose lifespan span y_b.ValidFrom and
(2) Y tuples whose ValidFrom value is in region I.
A possible heuristic to decide whether to read the next X tuple or Y tuple is presented below.

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

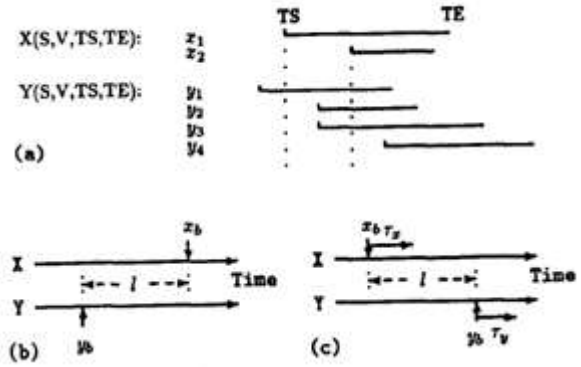

Figure-6 Contain-join: Both X and Y are sorted on TS in ascending order (only time attributes are shown)

The heuristic algorithm which can be used to decide whether to read an X tuple or a Y tuple is as follows. If the next X tuple is read, the expected ValidFrom due is Zb.ValidFrom+r_x. The number of Y tuples that would be garbage-collected *can* be estimated as the number of Y tuples in the workspace with ValidFrom value in the interval [za.ValidRom, Zb.ValidFrom+r_x. If the next Y tuple is read, the expected ValidFrom value is yb.ValidFrom+t_y. The number of disposable X tuples can be estimated as the number of X tuples in the workspace with ValidTo value in the interval [ya.ValidRom, by.ValidFrom+t_y]. Based on these two estimations, a decision is made on which would yield a greater reduction in the number of tuples in the workspace. Using this heuristic, the state (i.e. workspace contents) can be characterized as follows: (1) if we keep reading X tuples such
that all Y tuples in the state have been garbage-collected, the maximal set of X tuples that are required consists of all X tuples that span the time instant ya.ValidFrom. (2) conversely, if we keep reading Y tuples such that there is no X state tuple, the maximal set of Y state tuples that is required consists of those whose ValidFrom value lies in the lifespan of Zb.

We now summarize the state information requirements of processing the Contain-join for other sort orderings in Table 1. The algorithm for the case (b), i.e. when the relation X is sorted on ValidFrom and the relation Y is sorted on ValidTo, is similar to the one presented above (readers may refer to 1161 for details). Note that (1) it is generally inappropriate to have one relation sorted in ascending order and the other in descending order. (2) 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.

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

Sort Orders X		Sort Orders Y		Contain-join(X,Y)	Contain-semijoin(X,Y)	Contained-semijoin(X,Y)
TS	↑	TS	↑	(a)	(c)	(c)
TS	↓	TS	↓	-	-	-
TS	↑	TE	↑	(b)	(d)	-
TS	↓	TE	↓	-	-	(d)
TE	↑	TS	↑	-	-	(d)
TE	↓	TS	↓	(b)	(d)	-
TE	↑	TE	↑	-	-	-
TE	↓	TE	↓	(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.TS} ∪ {Y tuples whose TS value lie in region l}

(b) state = {X tuples whose lifespan span y_b.TE} ∪ {Y tuples whose lifespans are contained within region l}

(c) state ⊆ {X tuples whose lifespan span x_b.TS} ∪ {Y tuples whose TS values lie in region l}

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

4.2.2 Contained- & Contain-semijoin(X,Y)"

Contain-semijoin(X, Y) is defined as {x I x belongs to X and 3 y belong to Y s.t. 2's lifespan contains y's lifespan}. Contained-semijoin(X, Y) is
defined as {z I z EX and 3 y E Y s.t. 2's lifespan is contained in y's lifespan}. In a later section, we show that 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. Based on this observation, 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 7. The Contain-semijoin(X, Y) algorithm for this sort order is as follows.
'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 combined
together to gain better performance.

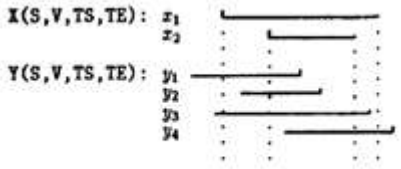

Figure-7 Contain-semi joins: X is sorted on TS and Y is on TE in ascending order

Contain-semijoin(X,Y):

Contain-semijoin(X,Y):

Repeat the following steps until one stream of tuples is exhausted. Suppose we have x_b and y_b tuples in the workspace. Then, one of the following conditions must hold:

- "x_b.ValidFrom<y_b.ValidFrom ∧ y_b.ValidTo<x_b.ValidTo" — i.e. x_b and y_b satisfy the semijoin condition, or
- "x_b.ValidFrom≥y_b.ValidFrom"[11], or
- "y_b.ValidTo≥x_b.ValidTo".

If the semijoin condition is satisfied, x_b is output and the next X tuple is read. If "x_b.ValidFrom≥y_b.ValidFrom" holds, the next Y tuple should be read because y_b cannot be contained in further X tuples. On the other hand, if "y_b.ValidTo≥x_b.ValidTo" holds the next X tuple should be read because x_b cannot be joined with further Y tuples.

It can be easily verified that only one tuple from each stream needs to be in the workspace. For example, in Figure 7, when x_1 is fetched, the local workspace contains <x_1, y_2> and for x_2 it is <x_2, y_4>.

It should be mentioned that the above algorithm can be applied to Contained-semi join(Y, X) for the same sort ordering with a slight modification - when the semi join condition is satisfied, yb is output and the next Y tuple is read. For other sort orderings (e.g. both streams are sorted on ValidFrom), we list the local workspace requirements in Table 1. We note that for Contain-semijoin(X, X) and Contained-semijoin(X, X), the stream of tuples may be scanned twice if we apply the semijoin algorithm presented above. To avoid this kind of inefficiency, we, therefore, devise a more efficient algorithm which scans the stream only once provided that it is sorted properly. As an example, suppose the relation X has primary sort ordering on the attribute ValidFrom and secondary sort ordering on ValidTo in ascending order. The algorithm for Contained-semijoin(X, X) is as follows.

Contained-semijoin(X,X):

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≤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.

"It does not matter what the relationship between ZB.ValidTo and yb.ValidTo is.

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 - *&-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 to join operations.

5 SEMANTIC QUERY OPTIMIZATION

Semantic query optimization techniques have been introduced and shown to be potentially useful in many studies [8]. However, the technique has not been widely used in conventional systems. The reason, we speculate, might be that conventional application domains ax 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 'FUI'.

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 "fl.Name=fL.The name" (in Figure 3(b)) concatenates those fl and f2 tuples corresponding

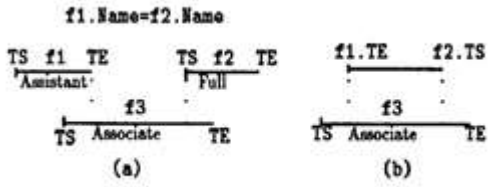

Figure-8 (a) The less-than join in the superstar query, and (b) its equivalent contained-semijoin condition after sematic optimisation

to those assistant professors promoted to full professors. The less than join then selects those fl and f2 tuple pairs which satisfy the less-than join condition *(0')* as shown in Figure 8(a). With the above semantic constraint, it is not " It to see that fl.ValidFrom<B.ValidTo and B.ValidFiom<f2.ValidTo is 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 techniques mentioned in the previous section. The second consequence of the constraint on the Rank attribute is that we are able to eliminate two redundant inequalities in *0';* 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.

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 and suggesting 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 such as in (321, relations are augmented with time attributes such as Validation 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 Validfiom 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. 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.

REFERENCES

[l] Abelson H. and Sussman G.J. ‘Structure and Interpretation of Computer Programs,’ The MIT Press, 1985.

[2] 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.

[3] Allen James F. ‘Maintaining Knowledge about Temporal Intervals,’ Communications of the ACM, Vo1.26, No.11, November 1983, pp.832-843.

[4] Batory D.S., Leung T.Y. and Wise T.E. ‘Implementation Concepts for an Extensible Data Model and Data Language,’, ACM %ns. on Database Systems, vo1.13, No.3,n September 1988, pp.231-262.

[5] Ben-Zvi J. ‘The Time Relational Model,’ (Unpublished) Ph.D. Thesis, Dept. of Computer Science, University of California,

Los Angeles, 1982.

[6] Bentley J. 'Algorithms for Reporting and Counting Geometric Intersections,' IEEE 2"s. on Computers, Vo1.C-28,No.9, September 1979, pp.643-647.

[7] Boral H., Dewitt D. 'Applying Data Flow Techniques to Data Bae Machines,' IEEE Computer, Vo1.15, No.8, August 1982, pp.57-63.

[8] Chakravarthy U.S., Fishman D.H., and Minker J. 'Semantic Query Optimization in Expert System and Database Systems,' Expert Database Systems, 1984, pp.32G-341.

[9] Clifford J., and Warren D.S. 'A Model for Historical Databases,' ACM Tmns. on Database Systems, Vo1.8, No.2, June 1983, pp.214-254.

[l0] 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.

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