Temporal Query Processing and Optimization in Multiprocessor Database Machines*

T.Y. Cliff Leung† Richard R. Muntz
Department of Computer Science
University of California, Los Angeles

Abstract

In this paper, we discuss issues involving temporal data fragmentation, temporal query processing, and query optimization in multiprocessor database machines. We propose parallel processing strategies, which are based on partitioning of temporal relations on timestamp values, for multi-way joins (e.g., complex temporal pattern queries) and optimization alternatives. We analyze the proposed schemes quantitatively, and show their advantages in computing complex temporal joins.

1 Introduction

With the availability of cheaper and larger secondary storage devices such as magnetic/optical disks, more historical data can be stored on line instead of being archived onto magnetic tapes or being purged from the database. Recently, there have been active research efforts that attempt to provide basic temporal functionality so that historical data can be accessed and queried more efficiently [Soo91]. There are several classes of temporal queries. Among the most difficult to process are the inequality joins and multi-way joins such as complex temporal pattern queries. Even in centralized database systems, these queries are often expensive to process.

Recently there has been growing interest in multiprocessor database machines which appear to have better price-performance than traditional centralized DBMS residing in mainframe computers [DeW90, Ter85]. A crucial design issue in these database machines is the *fragmentation strategy*, which specifies how tables are fragmented and stored, and which has a significant impact on the efficiency of query processing. Unfortunately, providing temporal functionality in parallel database machines as well as addressing the issue of fragmentation strategies for temporal data have largely been ignored.

In [Leu90] we proposed stream processing algorithms for processing temporal inequality join and semijoin operations. In this paper, we develop parallel join strategies for multiprocessor database machines based on the stream processing paradigm. For an inequality join of two relations, a conventional strategy in multiprocessor database machines, which is not always desirable, is to dynamically and fully replicate the smaller relation. We propose parallel processing strategies that are based on partitioning of temporal relations on timestamp values, and show that they can be attractive alternatives to conventional strategies. An analytic model is developed for estimating the number of tuples that have to be replicated; the model indicates in which situations only a fraction of tuples needs to be replicated among processors as opposed to replicating the entire relation.

Another subclass of complex queries is called snapshot or interval join queries. These queries refer to tuples that are active as of a certain time point or over a certain time interval in the past. Basically, the query qualification contains join predicates and comparison predicates on timestamps. We discuss optimization alternatives when these queries are processed using our approach.

The organization of this paper is as follows. In Section 2 we present the fundamental concepts. The parallel query processing strategies and optimization al-

*This work was partially supported by a MICRO grant from the University of California and the Hughes Aircraft Company.

†The author is currently with IBM Santa Teresa Laboratory, San Jose, USA.

**Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992**

	⋮		
Smith	\$12K	1	10
Smith	\$13K	10	15
Smith	\$20K	15	*now*
	⋮		

Table 1: A Sample Emp(Name,Sal,TS,TE) Relation

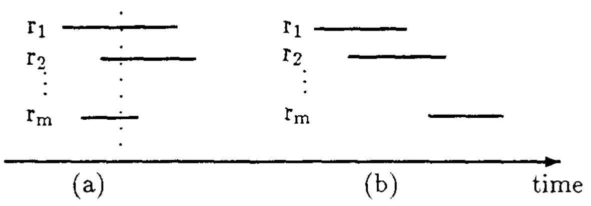


Figure 1: Classes of Temporal Joins

ternatives will be the main focus in Section 3 and Section 4 respectively. Finally, we discuss related work in Section 5, and conclusions and future research are included in Section 6.

2 Data Model

In the temporal data model, time points are regarded as natural numbers { 0, 1, $\cdots$, *now* } and are monotonically increasing. The special marker *now* represents the current time point. A time-interval temporal relation is denoted as X(S,V,TS,TE), where S is the surrogate, V is a time-varying attribute, and the interval [TS,TE) denotes the *effective* lifespan of a tuple [Sno87, Seg87]. The TS and TE attributes are referred to as timestamps. In Table 1 we show a sample employee relation which stores the salary history of employees. All temporal relations are assumed to have a homogeneous lifespan — [0,*now*). Furthermore, we require that for each tuple, the TS value is always smaller than the TE value.

We first propose a classification of Temporal Select-Join (denoted as TSJ) queries. This classification provides a meaningful partitioning of query types with respect to the difficulty and complexity of query processing and optimization. Each class has a restricted form of *query qualification* which is defined as a conjunction of several comparison predicates and join predicates, and thus each class is amenable to a particular query processing algorithm. We consider two special kinds of joins whose formal definitions will be presented shortly; both are referred to as "overlap joins" in the sense that the lifespans of tuples satisfying the join condition must overlap. Informally, their characterizations are:

TSJ_1 — All participating tuples that satisfy the join condition share a common time point, as illustrated in Figure 1(a). For example, finding a complex "event" pattern in which all (interval) events occur during the same period of time (or as of a time point) can be regarded as a TSJ_1 join query.

TSJ_2 — The tuples that satisfy the join condition overlap in a "chain" fashion, as illustrated in Figure 1(b). However, not all participating tuples that satisfy the join condition have to have a common time point. For example, finding a pattern in which (interval) events occur in some overlapping sequence can be regarded as a TSJ_2 join query. Note that all TSJ_1 queries are also TSJ_2 queries.

In this paper, we focus on only TSJ_1 queries.

We now precisely define the classes of queries that are of interest here. Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1,\cdots,R_m)$[1], we construct a join graph, denoted as G, from the query qualification $P(R_1,\cdots,R_m)$ using Definition 1 below. Based on the join graph, we are able to formally define TSJ_1 and TSJ_2 join queries.

Definition 1 Join Graph. There are m nodes in the join graph G; each node represents an operand relation R_k, $1 \le k \le m$, and is labeled with the name of that relation. We add an undirected edge between nodes R_i and R_j, where $i, j \in \{1,\cdots,m\}$ and $i \ne j$, to G if the following condition is satisfied:

$$P(R_1,\cdots,R_m) \Rightarrow R_i.TS \le R_j.TE \wedge R_j.TS \le R_i.TE^{2}.$$

□

Definition 2 TSJ_2. $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1,\cdots,R_m)$ is a TSJ_2 query if:

1. $m>1$, and
2. the join graph G constructed using Definition 1 is a *connected* graph. □

Definition 3 TSJ_1. A TSJ_2 query is also a TSJ_1 query if the join graph G constructed using Definition 1 is a *fully connected* graph. In other words, for all i and j such that $i, j \in \{1,\cdots,m\}$ and $i \ne j$, the following condition holds:

[1] R_i's need not be distinct.

[2] This condition is defined such that we can also handle the join predicate "X.TE=Y.TS" for a join of two relations. Testing the implications can be readily achieved via algorithms presented in [Ros80, Ull82, Sun89].

$P(R_1,\cdots,R_m) \Rightarrow R_i.TS \leq R_j.TE \wedge R_j.TS \leq R_i.TE$.

That is, for each m-tuple $<r_1,\cdots,r_m>$, where $r_k \in R_k$, $1 \leq k \leq m$, that satisfies the join condition $P(R_1,\cdots,R_m)$, all participating tuples (r_k's) must span a common time point. □

TSJ_1 and TSJ_2 queries are multi-way temporal joins (e.g., temporal pattern queries) in which the lifespans of tuples intersect. Non-TSJ_2 queries include Cartesian products across multiple relations (i.e., no join predicates) and a query with join condition "$R_i.TE<R_j.TS$". This characterization is crucial in developing the parallel processing algorithms to be described later. Examples of TSJ_1 queries include the "natural time-join" [Cli85, Cli87], the "intersection join" [Gun91], and the following temporal join operators [All83, Leu90]:

contain-join(X,Y) ≡ "X.TS<Y.TS ∧ Y.TE<X.TE"
overlap-join(X,Y) ≡ "X.TS<Y.TS ∧ Y.TS<X.TE ∧ X.TE<Y.TE"
intersect-join(X,Y) ≡ "X.TS<Y.TE ∧ Y.TS<X.TE"

We now show how common temporal queries can be formulated in TSJ_1. Consider the following temporal relations which store information on studios, directors and stars in the film industry[3]:

Studio(Sname,Head,TS,TE) – the head of a studio
Dir(Dname,Sname,TS,TE) – directors who worked for a studio
Stars(Star,Dname,TS,TE) – film stars who acted in films directed by a director

where Sname and Dname stand for the name of studios and directors respectively.

Example 1 Find the heads of studios and the directors who worked for the studios at the same time:

$\sigma_{P \wedge \text{Studio.Sname=Dir.Sname}}$ (Studio,Dir)[4]

where P is "intersect-join(Studio,Dir)". □

Example 2 Find all combinations of studio heads, film stars and directors such that they worked during the same period of time and when the star acted in a film that the director directed at the studio:

$\sigma_{P_1 \wedge P_2}$ (Studio,Dir,Stars)

where P_1 is "intersect-join(Studio,Dir) ∧ intersect-join(Dir,Stars) ∧ intersect-join(Studio,Stars)", and P_2 is "Studio.Sname=Dir.Sname ∧ Dir.Dname=Stars.Dname". □

[3] Adopted from examples in [Cli87].

[4] A more appropriate response might include two "computed" fields which represent the lifespan of a joined tuple. In this paper, we focus only on the query qualification which is a major optimization issue.

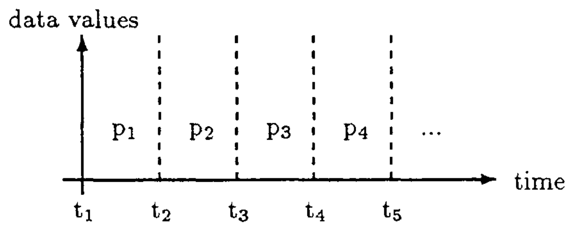


Figure 2: Range-partitioning along a Timestamp

3 Parallel Temporal Query Processing

In this section, we discuss data fragmentation schemes that facilitate the processing of temporal joins. We then introduce the notion of checkpointing the execution state of a query, and present the parallel query processing strategies.

3.1 Data Fragmentation

A number of well-known fragmentation strategies have been proposed and implemented in multiprocessor database machines [Ter85, DeW90, Gha90]. They include: range-partitioning, round-robin, and hashing. Below we discuss the strategy based on range-partitioning the timestamp values; a discussion of other strategies can be found in [Leu92].

The strategy is to partition temporal relations based on a timestamp (e.g., TS) as illustrated in Figure 2. For example, tuples that started during the interval $[t_1,t_2)$ are stored in processor p_1. Suppose there are n processors in the database machine, and let the processors be denoted as p_i, for $1 \leq i \leq n$. Let n_{pi} be the number of intervals in the partitioning function:

$$[t_1,t_2),\ \cdots,\ [t_{n_{pi}-1},t_{n_{pi}}),\ [t_{n_{pi}},t_{n_{pi}+1}).$$

We refer to t_i and $[t_i,t_{i+1})$ as the partitioning boundary and the partitioning interval (or simply partition) respectively. As relation lifespans are assumed to be $[0,now)$, by convention t_1 is 0 and $t_{n_{pi}+1}$ is *now*. In general, we require that the number of partitions be at least as large as the number of processors, i.e., $n_{pi} \geq n$. For simplicity, we adopt the hybrid range-partitioning scheme in [Gha90]: an interval $[t_j,t_{j+1})$ is assigned to p_i if i equals j modulo n. Partitioning relations on the TS (respectively TE) timestamp is called TS (respectively TE) range-partitioning. With TS range-partitioning, processor p_i stores a fragment of relation X, denoted as X_i, which contains tuples of X that started during the interval $[t_i,t_{i+1})$, i.e., "$t_i \leq X.TS < t_{i+1}$" holds. Similarly for the TE range-partitioning, X_i contains tuples that ended during the interval.

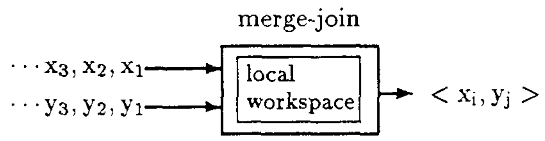


Figure 3: A Merge-join Stream Processor

In this paper, we assume that all operand relations of a given query are *homogeneously* range-partitioned, i.e., relations are partitioned using the same partitioning functions. We note that range-partitioning along the time dimension can facilitate the processing of complex temporal joins. The intuition is that temporal tuples with close timestamp values are likely to be clustered within the same processor and therefore a pair of tuples that satisfy the temporal join condition are likely to be stored at the same processor. Hence, fewer tuples would be copied between processors. In the following sections, we further develop the query processing strategies based on this approach.

There will certainly be many natural situations in which not all relations are homogeneously range-partitioned or relations are fragmented using other schemes. In Section 3.3 we will briefly describe the case when this assumption is relaxed, and we note that our processing strategies still apply.

3.2 Checkpointing Execution State

In order to introduce checkpointing in this context, we first discuss 1) the stream processing paradigm for temporal query processing, and 2) the notion of checkpointing a stream processing execution.

3.2.1 Stream Processing

In [Leu90], we introduced stream processing algorithms for processing temporal joins and semijoins. A *stream* is abstractly defined as an ordered sequence of data objects. A stream processing algorithm can be abstractly described as a *stream processor* which takes one or more input data streams and produces one or more output data streams. A classical example of stream processing operations is the merge-join where both operand relations are sorted on their join attribute as depicted in Figure 3; the output from the join is also a data stream sorted on the join attribute.

Stream processing in database systems has several interesting characteristics. First, a computation on a stream has access only to one element at a time (referenced via a *data stream pointer*) and only in the specified ordering of the stream. Second, stream processors may keep some local state information in order to avoid multiple readings of the same data stream. The state information represents a summary of the history of a computation on the portion of data streams that have been previously read. Depending on the operation itself, the state may be composed of copies of some elements (e.g., tuples for join operations) or some summary information of the objects previously read (e.g., partial sum for aggregate functions). In [Leu90] we noted that the storage requirements for the local state information can depend heavily on the sort ordering of input streams, data statistics and the operation itself. As a simple example from conventional query processing, when we merge-join two relations sorted on their key attribute, at any point in time we need only one tuple from each table as the state information. On the other hand, if an operand relation is not sorted, keeping all its tuples in the local workspace is required prior to reading the other relation. Alternatively, one can reduce the storage size by allowing the unsorted relation to be read multiple times — the merge-join then becomes a nested-loop join.

3.2.2 Checkpointing Stream Processors

We now discuss the notion of *checkpointing* the execution state of a query in the context of stream processing. To illustrate the idea more clearly, we consider a stream processor that implements a query Q with data streams X and Y as shown in Figure 4, and we assume that both X and Y are sorted on a timestamp (either TS or TE) in increasing order. A stream processor that implements the processing of a query Q starts by reading elements at the beginning of data streams. At any time t, the execution state of the stream processor includes:

- state information, denoted as $s_q(t)$, stored in the local workspace of the stream processor.
- $dsp_x(t)$ and $dsp_y(t)$: the data stream pointer for X and Y respectively which represent the positions in the data streams up to which the stream processor has read so far. Recall that data stream elements are accessed one at a time using the data stream pointer.

A checkpoint of the execution state of a query at time t, denoted as $ck_q(t)$, has the following characteristic:

> The execution state at any time t'>t is a function of $ck_q(t)$ and all tuples in the data stream X (respectively Y) between $dsp_x(t)$ (respectively $dsp_y(t)$) and the first tuple in

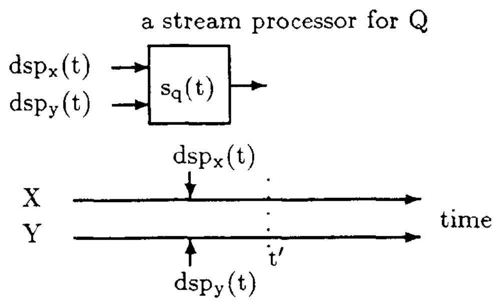


Figure 4: Checkpointing the Execution of a Stream Processor for Query Q

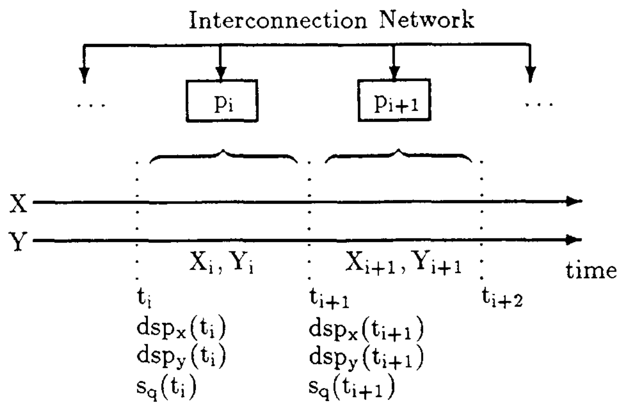


Figure 5: The Parallel Processing Strategy

the data stream after t'. That is, the execution state at t' contains sufficient information so that re-reading the portions of data streams prior to $dsp_x(t)$ and $dsp_y(t)$ can be avoided.

The type of state information required depends on the query itself. We will define what state information is required for TSJ_1 queries shortly, but intuitively, at any time the state information of a join query consists of a subset of tuples of operand relations that were previously read.

3.2.3 The Approach

We now outline the parallel processing strategy for TSJ_1 queries. Remember that relations are homogeneously range-partitioned on a timestamp (TS or TE), i.e., the partition $[t_i, t_{i+1})$ is assigned to processor p_i as depicted in Figure 5. For the moment, we assume that the data stream pointers at the partitioning boundary t_i, $dsp_x(t_i)$ and $dsp_y(t_i)$, are "pointing" at the relation fragments X_i and Y_i respectively. Given a query Q, the strategy is to *construct* the state information at *every* partitioning boundary so that each processor can *independently* process the query Q on its local relation fragments using the constructed state information. For example, as shown in Figure 5, p_i will process its local fragments X_i and Y_i using the state information $s_q(t_i)$. Similarly, p_{i+1} will process X_{i+1} and Y_{i+1} using the state information $s_q(t_{i+1})$. In general, the strategy has three distinct phases:

Replication Phase Construct the state information for every partition.

Join Phase The query can be executed by each processor using its local relation fragments and the constructed state information.

Merge Phase The query response is produced by merging the results returned from all processors and eliminating duplicates.

Let us emphasize that TSJ_1 queries are multi-way temporal joins in which all participating tuples share a common time point. For the sake of simplicity of exposition, we focus on joins of two relations unless otherwise stated.

3.3 Replication Phase

Intuitively, the replication phase copies tuples whose lifespans intersect with each other such that they co-exist at the same processor for the subsequent join phase. Instead of copying *all* tuples that span a partitioning interval to the corresponding processor, one may limit which tuples need to be copied using a *state predicate* which is defined formally below. For each relation, we derive a state predicate using the query qualification as we will explain shortly. In general, the more restrictive state predicates are, the fewer tuples would have to be replicated.

Definition 4 A *state predicate* for a relation R, denoted as $P|_r$, is a query qualification on R. That is, $P|_r$ is a conjunction of several comparison predicates. □

Consider a query involving relations X and Y whose query qualification is P(X,Y), a state predicate for relation X, denoted as $P|_x$, is obtained from P(X,Y) by substituting join predicates and comparison predicates that involve the relation Y with "true"[5]. That is, $P|_x$ contains only comparison predicates involving only X in P(X,Y). A state predicate for the relation Y, denoted as $P|_y$, is defined analogously.

[5] Recall that we consider only conjunctions of join and comparison predicates as query qualification.

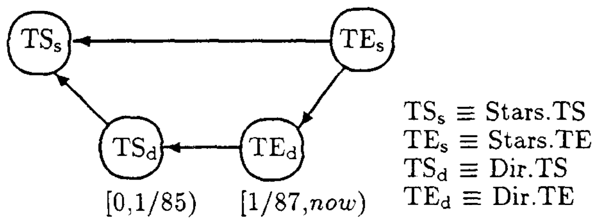


Figure 6: Constraint Graph for Example 4 — Upper and Lower Bounds on Timestamps

Example 3 Consider the film industry examples presented earlier. The query for finding the head of a studio that the director "Fred" worked for at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge P}(\text{Studio,Dir})$$

where P is "Studio.Sname=Dir.Sname ∧ Dir.Dname= Fred". The state predicate for the relation Dir is "Dname=Fred" while the state predicate for the relation Studio is "true". □

By propagating constraints between attributes, one can sometimes find a more restrictive state predicate. For example, consider the query for finding the head of the studio "MGM" and the directors who worked for the studio at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge P}(\text{Studio,Dir})$$

where P is "Studio.Sname=Dir.Sname ∧ Studio.Sname= MGM". Using the first (simplistic) method, the state predicate for the relation Dir is "true" while the state predicate for the relation Studio is "Studio.Sname=MGM". Intuitively, only tuples in relation Dir that satisfy the predicate "Dir.Sname=MGM" would participate in the join and thus only these tuples should be replicated as state information. Likewise, bounds on timestamp values can be propagated between relations [Ull82, Chak84, Jar84, She89]. Because of space limitations, we only illustrate the approach using the following example.

Example 4 Suppose we want the combinations of all stars and directors such that the star acted in films directed by the director during the entire period of time in which the director worked for a studio for the entire interval [1/85,12/86). The query is:

$$\sigma_{\text{contain-join(Stars,Dir)} \wedge P}(\text{Stars,Dir})$$

where P is "Dir.TS<1/85 ∧ 12/86<Dir.TE". The constraints on timestamp values are represented by a constraint graph as shown in the Figure 6. Basically, a node represents a timestamp and a solid arrow represents the "before" (i.e., <) relationship between two timestamps. The values of timestamps "Dir.TS" and "Dir.TE" are bounded by [0,1/85) and [1/87,*now*) respectively[6].

The constraints on timestamp values are then propagated among nodes. For example, in Figure 6, the TS values of relation Stars (i.e., TS_s) are bounded by the interval [0,12/84) while the TE values (i.e., TE_s) are bounded by [2/87,*now*). Thus, the state predicate for the relation Stars becomes "Stars.TS<12/84 ∧ 2/87≤Stars.TE". □

Given a query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, we first derive a state predicate for each operand with the results being denoted by $P|_x$ and $P|_y$. Using these state predicates, one can define the state information required on each processor as follows.

Definition 5 Given that the partition $[t_i, t_{i+1})$ is assigned to processor p_i, the *state information* for a relation R at the partitioning boundary t_i, denoted as $s_r(t_i)$, contains:

$\{ r \mid r \in R \wedge r.TS < t_i \wedge t_i \leq r.TE \wedge P|_r(r) \}$
if R is TS range-partitioned

$\{ r \mid r \in R \wedge r.TS < t_{i+1} \wedge t_{i+1} \leq r.TE \wedge P|_r(r) \}$
if R is TE range-partitioned

where $P|_r$ is the derived state predicate for the relation R, and $P|_r(r)$ holds for the tuple r. □

Essentially, all qualified tuples (based on the state predicate) whose lifespan intersects with the interval $[t_i, t_{i+1})$ and are not stored in the local fragments at processor p_i will be replicated at processor p_i as "state information". In other words, if the lifespans of tuples intersect, they should co-exist at the same processor.

Note that when relations are not homogeneously range-partitioned or relations are fragmented using other schemes, more tuples will be copied between processors. For example, consider a temporal join of relations R and S, where R is TS range-partitioned as before and S is fragmented using a hash function on any attribute(s). The state information for S at partition $[t_i, t_{i+1})$ is:

$\{ s \mid s \in S \wedge s.TS \leq t_{i+1} \wedge t_i \leq s.TE \wedge P|_s(s) \}$.

We emphasize that only a fraction of S tuples (instead of the entire relation) is replicated at a partition.

As soon as the state information for all operand relations at all partitions have been constructed, the join

[6] We assume that the time granularity in this example is "month", i.e., consecutive months are mapped into natural numbers.

phase, which is the focus of the following subsection, can proceed. That is, the query can be processed in parallel without additional data transfers.

3.4 Join Phase & Merge Phase

For a query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, each processor p_i can execute Q using its local relation fragments and the state information constructed at p_i. The response to Q is the union of the results (eliminating duplicates) from all processors:

$$\bigcup_{1 \le i \le n_{pi}} \{ \sigma_P((X_i \cup s_x(t_i)), (Y_i \cup s_y(t_i))) \}$$

where n_{pi} is the total number of partitions. For a TSJ_1 join query involving m relations, we can use the following strategy whose correctness can be found in [Leu92]:

$$\bigcup_{1 \le i \le n_{pi}} \{ \sigma_P((R_{1,i} \cup s_{r_1}(t_i)), \cdots, (R_{m,i} \cup s_{r_m}(t_i)))\}$$

where $R_{j,i}$ is the ith fragment (i.e., for partition $[t_i, t_{i+1})$) of the relation R_j, $j \in \{1,\cdots,m\}$, which is stored at processor p_i.

4 Optimization Alternatives

In this section, we discuss several optimization alternatives for processing TSJ_1 queries in parallel database machines, and present a quantitative analysis on the overhead of the replication phase.

4.1 Reducing State Information

Depending on the properties of the user query and data statistics, there are opportunities for reducing the number of tuples to be replicated as state information. First, we define the *asymmetry* property of operands in a TSJ_1 join query with respect to the TS and TE timestamps.

Definition 6 Given $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m) \in TSJ_1$. The relation R_k, $k \in \{1,\cdots,m\}$, has the asymmetry property with respect to the *TS* timestamp if the following condition is satisfied:

$$P(R_1, \cdots, R_m) \Rightarrow R_k.TS \ge R_i.TS, \qquad \forall\ 1 \le i \le m.$$ □

Definition 7 Given $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m) \in TSJ_1$. The relation R_k, $k \in \{1,\cdots,m\}$, has the asymmetry property with respect to the *TE* timestamp if the following condition is satisfied:

$$P(R_1, \cdots, R_m) \Rightarrow R_k.TE \le R_i.TE, \qquad \forall\ 1 \le i \le m.$$ □

For each m-tuple $<r_1, \cdots, r_m>$ that satisfies the query qualification P, where $r_i \in R_i$ for $1 \le i \le m$, the asymmetry property with respect to the TS timestamp means that the tuple r_k must have the maximal TS value among all participating tuples. For example, consider contain-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TE<X.TE$" and overlap-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TS<X.TE \wedge X.TE<Y.TE$". The relation Y in both contain-join(X,Y) and overlap-join(X,Y) has the asymmetry property with respect to the TS timestamp. Similarly, the asymmetry property with respect to the TE timestamp means that the tuple r_k must have the minimal TE value among all participating tuples. For example, the relation Y in contain-join(X,Y) and the relation X in overlap-join(X,Y) have this asymmetry property.

Depending on whether a relation is TS or TE range-partitioned, the asymmetry properties can be used to show that constructing the state information for some relation is redundant in the sense the result produced by the local joins using those tuples (as state information) would have been produced by some other processors, and therefore the replication phase for that particular relation can be eliminated.

Property 1 Property of Redundant State Information. Given:

- a query $Q \equiv \sigma_P(R_1, \cdots, R_m) \in TSJ_1$, and
- there are m′, where $1 \le m' \le m$, relations which have the asymmetry property with respect to their partitioning timestamp (we use a subscript j to denote these relations as R_j where $j = \{1,\cdots,m'\}$).

Conditions under which the state information for a relation is redundant are:

- If all R_j's, $j \in \{1,\cdots,m'\}$, are TS range-partitioned, then all R_j's hold the property.
- If all R_j's, $j \in \{1,\cdots,m'\}$, are TE range-partitioned, then all R_j's hold the property.
- If some R_j's are TS range-partitioned while others are TE range-partitioned, then R_j's can be partitioned into two disjoint sets:

  $R_j|_{TS}$ and $R_j|_{TE}$.

  The first set corresponds to TS range-partitioning while the second set corresponds to TE range-partitioning. Then relations in either set (not both sets) hold the property[7]. □

[7] Then we have a choice of selecting which relations to have the redundant state information property.

Theorem 1 The replication phase for a relation can be eliminated if the relation holds the property of redundant state information[8]. □

Example 5 Consider Example 4 whose join condition is "contain-join(Stars,Dir)", i.e., "Stars.TS $<$ Dir.TS $\wedge$ Dir.TE$<$Stars.TE". If the relation Dir is TS range-partitioned, its replication phase can be eliminated. That is, one has to replicate only tuples of relation Stars as state information. □

There are several interesting observations that can be made. First, when all temporal join predicates are inequalities, only *one* operand relation has the redundant state information property. Second, for contain-join(X,Y) the relation Y has the asymmetry property with respect to both the TS and TE timestamps. For this reason, state information for the relation Y need not be constructed regardless of whether the relation is TS or TE range-partitioned. Thirdly, when there is an *equality* temporal join predicate (e.g., "X.TS=Y.TS" or "X.TE=Y.TE") between two relations, and both relations have the asymmetry property with respect to their join attribute (i.e., timestamp), Y has the redundant state information property if the state information of X is redundant (or vice versa). As another example, consider meet-join(X,Y), whose join condition is "X.TE=Y.TS", and X is TE range-partitioned while Y is TS range-partitioned. Both relations have the asymmetry property with respect to their respective partitioning timestamp, and thus the state information for both X and Y are redundant.

In addition to the above properties, the data statistics of each local fragment can be used to further reduce data replication. Let us consider a join of two relations — X and Y. Suppose that the database system keeps the maximum and minimum of the TS and TE values for every relation fragment. For example, the TS and TE values of a relation fragment Y_i (i.e., the interval $[t_i,t_{i+1})$) of the relation Y are bounded by the intervals: $[Y_i.TS_{min}, Y_i.TS_{max}]$ and $[Y_i.TE_{min}, Y_i.TE_{max}]$ respectively. We further suppose that the fragment Y_i is stored at processor p_i. Together with the query qualification, the statistics can be used to further reduce data replication of the relation X. To illustrate this point, we consider the following examples, assuming that both X and Y are TS range-partitioned:

- Consider the overlap-join(X,Y) whose join condition is "X.TS$<$Y.TS $\wedge$ Y.TS$<$X.TE $\wedge$ X.TE $<$Y.TE". Intuitively, tuples in the relation X that span the partitioning boundary t_i and whose TE values are smaller than or equal to $Y_i.TS_{min}$ need not be sent to processor p_i because these X tuples do not join with any tuples in Y_i. This is also true for X tuples that span t_i and whose TE values are larger than or equal to $Y_i.TE_{max}$.
- Consider the contain-join(X,Y) whose join condition is "X.TS$<$Y.TS $\wedge$ Y.TE$<$X.TE". Tuples in the relation X that span t_i and whose TE values are smaller than or equal to $Y_i.TE_{min}$ need not be sent to p_i as state information.
- Consider the meet-join(X,Y) whose join condition is "X.TE=Y.TS". Tuples in the relation X that span t_i and whose TE values are smaller than $Y_i.TS_{min}$ or larger than $Y_i.TS_{max}$ need not be sent to p_i as state information.

[8] A proof of this theorem can be found in [Leu92].

4.2 Participant Processors

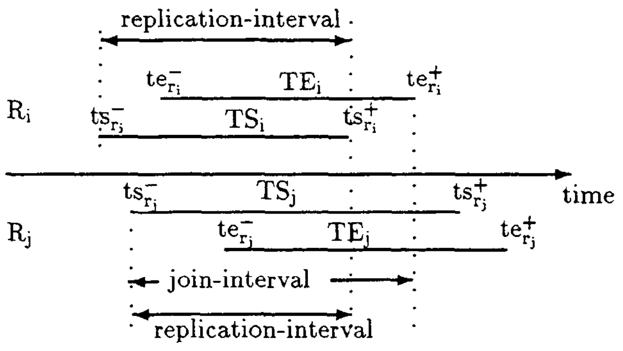
For the parallel processing strategies that we discussed earlier, *all* processors participate in the replication and join phases. However, for some TSJ_1 queries, it can be determined a priori that some processors necessarily return a null response when they perform the local join. Similarly, it can also be determined a priori that some processors need not replicate some fragments of a relation in the replication phase since the relation fragments will not contribute to the query response. These situations may occur when the query qualification contains some comparison predicates involving timestamps (such as in snapshot or interval queries). To illustrate the idea, we first define the notion of *replication-interval* and *join-interval.*

Definition 8 The *replication-interval* for an operand relation in a query is defined as the minimal interval with the property that only tuples whose partitioning timestamp value falls within the interval must participate as state information[9]. □

Definition 9 The *join-interval* for a query is defined as the minimal interval with the property that only tuples whose partitioning timestamp value falls within the interval can possibly contribute to the query response. □

Definition 10 A *join processor* is referred to as a processor that has to participate in the join phase (of our parallel processing strategy), i.e., a processor which has a partitioning interval that intersects with the join-interval. Otherwise, it is referred to as a non-join processor which necessarily returns a null response. □

[9] Note that if the relation has the property of redundant state information discussed in the previous section, the corresponding replication-interval is necessarily null (i.e., no tuples will be replicated as state information).



Constraints:
$ts^-_{r_i} \le TS_i < ts^+_{r_i}$ $ts^-_{r_j} \le TS_j < ts^+_{r_j}$
$te^-_{r_i} \le TE_i < te^+_{r_i}$ $te^-_{r_j} \le TE_j < te^+_{r_j}$
$TS_i < TE_i$ and $TS_j < TE_j$ for all tuples

R_i, R_j: TS range-partitioned

Figure 7: Determining the Join-interval and the Replication-intervals

Definition 11 A *replication processor* is referred to as a processor that has to participate in the replication phase (of our parallel processing strategy), i.e., the processor which has a partitioning interval that intersects with the replication-intervals. Otherwise, it is referred to as a non-replication processor. □

If the join-interval is null, the join response is necessarily null. Similarly, if the replication-interval for a relation is null, tuples of that relation need not be replicated as state information. Otherwise, tuples in the replication-intervals are replicated on join processors as state information for the join phase.

The join-interval and replication-intervals for a given query depend on the following:

1. the TS and TE range-partitioning functions, and
2. the query qualification:
   - the relationship between the comparison predicates involving timestamps and the temporal join predicates.
   - the property of redundant state information discussed earlier.

Earlier we briefly mentioned the issue of determining upper and lower bounds on the TS and TE values of each individual relation by propagating constraints between relations using a constraint graph. Below we address other issues.

Consider a TSJ_1 join query with relations R_i and R_j. Suppose that the upper and lower bounds of the timestamp values have been determined. As illustrated in Figure 7, we denote ts^-_r and ts^+_r as the lower and upper bounds on the TS timestamp of relation R respectively, and similarly te^-_r and te^+_r for the TE timestamp. Here we consider only the case when both R_i and R_j are TS range-partitioned; a discussion for other situations can be found in [Leu92]. There are three situations to be considered:

1. When neither R_i nor R_j has the property of redundant state information, the join-interval for the query is:

   $[\max(ts^-_{r_i}, ts^-_{r_j}), \min(\max(ts^+_{r_i}, ts^+_{r_j}), te^+_{r_i}, te^+_{r_j}))$[10].

   The replication-interval for relation R_k, where $k \in \{i,j\}$, is:

   $[\max(\min(ts^-_{r_i}, ts^-_{r_j}), ts^-_{r_k}), \min(\min(ts^+_{r_i}, ts^+_{r_j}), ts^+_{r_k}))$.

   In Figure 7, the join-interval is $[ts^-_{r_j}, te^+_{r_i})$, and the replication-interval for R_i is $[ts^-_{r_i}, ts^+_{r_i})$ while that of R_j is $[ts^-_{r_j}, ts^+_{r_i})$.

2. When the relation R_i (but not R_j) has the property of redundant state information, the join-interval becomes:

   $[\max(ts^-_{r_i}, ts^-_{r_j}), \min(ts^+_{r_i}, te^+_{r_j}))$,

   and the replication-interval for R_i is a null interval while that of R_j is:

   $[ts^-_{r_j}, \min(ts^+_{r_i}, ts^+_{r_j}))$.

3. When both relations R_i and R_j have the property of redundant state information, the join-interval becomes:

   $[\max(ts^-_{r_i}, ts^-_{r_j}), \min(ts^+_{r_i}, ts^+_{r_j}))$

   and the replication-intervals of both relations are null intervals.

Example 6 Find the directors who joined a studio sometime during [1/85,1/86) and also became the head of the studio sometime during [1/86,1/87):

σ_P (Studio,Dir)

where P is "intersect-join(Studio,Dir) $\wedge$ Studio.Sname = Dir.Sname $\wedge$ 1/85≤Dir.TS $\wedge$ Dir.TS<1/86 $\wedge$ 1/86≤Studio.TS $\wedge$ Studio.TS<1/87". Suppose that both relations Studio and Dir are TS range-partitioned. The constraints on the TS timestamps

[10] The operator max(A,B) (respectively min(A,B)) returns the larger (respectively smaller) value of A and B.

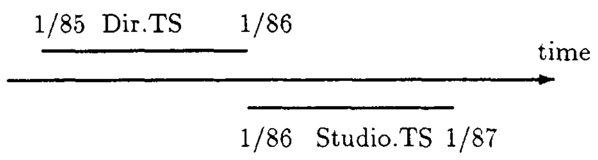


Figure 8: Constraints on TS Timestamps of Relations Dir and Studio

of both relations are illustrated in Figure 8. The join-interval for the query is [1/86,1/87). The replication-interval for relation Dir is [1/85,1/86) and that for relation Studio is a null interval. That is, the state predicates for relations Dir and Studio are "$1/85 \leq$ Dir.TS $\wedge$ Dir.TS $< 1/86 \wedge$ Dir.TE $\geq 1/86$" and "false" respectively. The join phase involves only joining the state information for the relation Dir and the local fragments of relation Studio that started during [1/86,1/87)[11]. □

To recap, using the query qualification one can a priori determine which processors have to send data as state information and which processors have to receive data as state information. In the following subsection, we discuss the overhead of constructing the state information.

4.3 Quantitative Analysis

The overhead associated with constructing the state information for a relation R can be measured in terms of the number of tuples to be replicated since the communication and/or storage costs will be directly related to this number (and the tuple size). We let λ be the rate of insertion of tuples into the relation R (in terms of number of tuples per time unit), $\overline{T_{ls}}$ be the average tuple lifespan, and TR_{ls} be the relation lifespan. Using Little's result [Lit61], the average number of tuples that are active as of a particular time, denoted by $\overline{n}$, is given by:

$$\overline{n} = \lambda \cdot \overline{T_{ls}}.$$

A natural assumption is that the average number of active tuples at partitioning boundaries is also $\overline{n}$. Similarly, the total number of tuples in the relation R is:

$$\lambda \cdot TR_{ls}.$$

Suppose that the selectivity of the state predicate q that is used to construct the state information for the relation R is σ_q and is defined as the fraction of tuples in R that satisfy q. The number of tuples that are copied as state information is then given by:

$$\sigma_q \cdot n_p \cdot \overline{n} = \sigma_q \cdot n_p \cdot \lambda \cdot \overline{T_{ls}}$$

where n_p is the number of partitioning boundaries at which state information has to be constructed (i.e., the partitioning intervals that overlap with the join-interval that we discussed in the previous section). Note that n_p must be smaller than the total number of partitioning intervals (n_{pi}).

Definition 12 The *overhead* is defined as the ratio of the number of tuples to be copied over the total number of tuples in the relation:

$$\sigma_q \cdot n_p \cdot \lambda \cdot \overline{T_{ls}}/(\lambda \cdot TR_{ls}) = \sigma_q \cdot n_p \cdot \overline{T_{ls}}/TR_{ls}. \quad \square$$

The quantity is consistent with our intuition that:

- n_p: the overhead increases as the number of partitions with state information increases.
- σ_q: the more selective the state predicate (which constructs the state information) is, the less overhead is incurred.
- $\overline{T_{ls}}/TR_{ls}$: the overhead is smaller for relations with relatively short tuple lifespans (compared with the relation lifespan).

5 Previous Work

The parallel processing schemes that we present in this paper is a substantial extension of the work on generalized data stream indexing [Leu92a] — the notion of checkpointing the execution state of a query appears in both [Leu92a] and this paper. In [Leu92a], we proposed an indexing technique based on periodically checkpointing on data streams which are sorted on the timestamp values. Checkpoints are stored in a file structure which is in turn indexed on checkpoint times. In this paper, we apply the idea of checkpointing in parallel database machines.

[Kar90] is apparently the first publication that appears to support temporal features in multiprocessor database machines. The paper, however, only discussed a front-end syntactic translator for a relational database system regardless of whether or not the database system is residing on a multiprocessor database machine. Moreover, there is no discussion on query processing optimization or fragmentation strategies.

In [DeW91] a "partitioned band" join algorithm was proposed to evaluate the so-called "band join"[12]:

[11] One can obtain tighter bounds by examining (and thus accessing) local fragment of relation Studio that started during [1/86,1/87), as well as further analyzing the temporal join predicates involved.

[12] One can think of it as a "fuzzy" equi-join.

A "band join" between relations R and S on attributes R.A and S.B is a join in which the join condition is "$R.A-c_1 \leq S.B \leq R.A+c_2$", where c_1 and c_2 are non-negative constants.

In the band join algorithm, ranges of the operand relations R_i and S_i, where $i \in \{1,\cdots,n\}$, are found such that (1) $R = \bigcup_i R_i$ and $S = \bigcup_i S_i$, and (2) for every tuple r in R_i, it is required that all tuples of S that join with r appear in S_i. The complete join is formed by joining R_i and S_i for each range ($i=1,\cdots,n$) and merging the result. With the assumption that the width of a "band" (i.e., c_1+c_2) is small, the major concern in [DeW91] is to choose the ranges such that each of the R_i fits entirely into the buffer pool. For the parallel version of the band join algorithm, each join between ranges R_i and S_i can be performed by a separate processor.

One can process the above band join using our strategies as follows. Suppose both relations R and S are range-partitioned based on the join attribute (R.A and S.B) using the same partitioning function. This assumption is easily relaxed and would just result in greater data movement. We further suppose that a partitioning interval $[v_i,v_{i+1})$ is assigned to a processor p_i, i.e., a tuple $r \in R$ (similarly for tuples in S) is stored at p_i if its join attribute value falls into this interval. The replication phase then involves copying tuples $s \in S$ to p_i if the value of $s.B$ falls into the interval $[v_i-c_1,v_i)$ or $[v_{i+1},v_{i+1}+c_2)$. When both c_1 and c_2 are small, tuples from only processors p_{i-1} and p_{i+1} are replicated at processor p_i. After the replication process, the join can be processed as the merging of the results of the parallel local joins. If say R is range-partitioned on the join attribute but S is not, then the same strategy works except that the tuples in S replicated on processor p_i (i.e., those tuples whose attribute B values fall into $[v_i-c_1,v_{i+1}+c_2)$) may come from all other processors.

In this paper, we consider temporal join queries with several time-interval relations and investigate the query processing issues by studying the qualification clause as opposed to individual join operators. Moreover, we address the issues related to both TS and TE range-partitioning schemes, as well as several query optimization strategies.

6 Conclusions & Future Work

We discuss parallel query processing strategies for the complex temporal join queries and a number of optimization alternatives. Based on the query qualification, some work can be shown to be redundant and therefore can be eliminated from the replication or join phase. We note that data statistics and the characteristics of the query qualification can also be used to reduce the state information to be constructed and thus the overhead of the replication phase can further be reduced. Finally, we provide an analytic method which allows us to estimate the overhead associated with the replication phase, i.e., the number of tuples that should be copied between processors. The overhead is small when the average tuple lifespan is small compared with the relation lifespan or the query qualification is restrictive.

In [Leu92], we reported a preliminary study on various fragmentation schemes for temporal data. We also proposed a fragmentation scheme in which current and history tuples are fragmented and stored using different strategies — current tuples are partitioned based on another function such as hashing on surrogate attribute while history tuples are range-partitioned on the timestamp values. The approach allows more efficient accesses to current tuples via surrogates and yet facilitates the processing of temporal joins. There are many research directions which require further investigation, and some preliminary results can be found in [Leu92]. The most challenging ones include the parallel query processing strategies for TSJ_1 queries whose operand relations are heterogeneously range-partitioned, i.e., the relations are (timestamp) range-partitioned using different partitioning functions. Also it appears that our parallel join strategies presented here can be easily adopted to process TSJ_2 queries: the join sequence can be obtained by a graph reduction algorithm on the join graph constructed using Definition 1. We plan to investigate this further.

Acknowledgements

The authors are grateful to Prof. Richard Snodgrass and the anonymous reviewers for improvements they suggested on the paper.

References

[All83] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Chak84] U.S. Chakravarthy, D.H. Fishman, and J. Minker. Semantic Query Optimization in Expert System and Database Systems. In *Expert Database Systems*, pages 326–341, 1984.

[Cli85] J. Clifford and A. Tansel. On an Algebra for Historical Relational Databases: Two

Views. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 247–265, May 1985.

[Cli87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 528–537, February 1987.

[DeW90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[DeW91] D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 443–452, 1991.

[Gha90] S. Ghandeharizadeh and D. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 481–492, 1990.

[Gun91] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 336–344, 1991.

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

[Kar90] S. Karimi, M. Bassiouni, and A. Orooji. Supporting Temporal Capabilities in a Multi-computer Database System. In *Proc. of the Int. Conf. on Databases, Parallel Architectures, and their Applications*, pages 20–26, March 1990.

[Leu90] T.Y. Leung and R.R. Muntz. Query Processing for Temporal Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 200–207, 1990.

[Leu92] T.Y. Leung. *Query Processing and Optimization in Temporal Database Systems.* PhD thesis, University of California at Los Angeles, 1992. Department of Computer Science.

[Leu92a] T.Y. Leung and R.R. Muntz. Generalized Data Stream Indexing and Temporal Query Processing. In *2nd Int. Workshop on Research Issues on Data Engineering — Transaction and Query Processing (RIDE-TQP)*, February 1992.

[Lit61] J. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operational Research*, 9, 1961.

[Ros80] D. Rosenkrantz and H. Hunt. Processing Conjunctive Predicates and Queries. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 64–72, 1980.

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

[She89] S.T. Shenoy and Z.M. Ozsoyoglu. Design and Implementation of a Semantic Query Optimizer. *IEEE Trans. on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[Sno87] R. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. on Database Systems*, 12(2):247–298, June 1987.

[Soo91] M.D. Soo. Bibliography on Temporal Databases. *The ACM SIGMOD Record*, 20(1):14-23, March 1991.

[Sun89] X. Sun, N. Kamel, and L. Ni. Solving Implication Problems in Database Applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 185–192, June 1989.

[Ter85] Teradata Corporation. *DBC/1012 Database Computer System Manual Release 2.0*, November 1985. Document No. C10-0001-02.

[Ull82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, second edition, 1982.