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

TEMPORAL QUERY PROCESSING AND OPTIMIZATION IN MULTIPROCESSOR DATABASE MACHINES

T. Y. Cliff Leung
R. R. Muntz

November 1991
CSD-910077

Temporal Query Processing and Optimization in Multiprocessor Database Machines*

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

November 7, 1991

Abstract

In this paper, we address issues involving temporal data fragmentation, temporal query processing, and query optimization in multiprocessor database machines. The impact of conventional data fragmentation schemes on various temporal queries is discussed. We propose parallel processing strategies for complex temporal pattern queries (such as multi-way joins) which are based on partitioning temporal relations on timestamp values, and optimizations for processing snapshot operators (i.e., comparison predicates involving timestamps). We analyze the proposed schemes quantitatively and discuss the advantages and limitations. Implementation aspects of the proposed schemes and other parallel query processing techniques are also presented.

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

1 Introduction

There has been a growing interest in multiprocessor database machines which appear to have better price-performance than traditional DBMSs residing in mainframe computers. A crucial design issue in these database machines is the so-called *fragmentation strategy* which specifies how tables are fragmented and stored in the database system. The fragmentation strategy has a great impact on the efficiency of query processing.

With the availability of cheaper and larger secondary storage devices such as magnetic disks, more historical data tends to be stored online 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 functionalities so that historical data can be accessed and queried more efficiently [Sno87, Sta88].

There are several classes of temporal queries. Among the most difficult to process is the complex temporal pattern queries which are multi-way joins whose join condition often contains a conjunction of several inequality join predicates. In general, these queries are often expensive to process. The difficulty of the problem can be further increased for the large temporal relations. With the availability of relatively cheap parallel database machines, one can exploit parallelism for processing this type of queries.

In [Leu90] we proposed stream processing algorithms for processing temporal inequality join and semijoin operations. In this paper, we develop parallel join strategies based on the stream processing paradigm, and show that they can be attractive alternatives. For an inequality join of two relations, a conventional strategy in multiprocessor database machines is to dynamically and fully replicate the smaller operand relation. The parallel strategies proposed here are based on partitioning temporal relations on timestamp values. An analytical model is developed for estimating the number of tuples that have to be replicated; this model indicates in what situations only a fraction of a relation is replicated among processors as opposed to fully replicating the entire relation.

Another subclass of complex queries is called *snapshot* join queries which are join queries as of a certain time point or over a certain time interval in the past. We also discuss optimizations can be achieved when these queries are processed using our proposed parallel strategies.

The organization of this paper is as follows. In Section 2 we present the fundamental concepts. Section 3 is devoted to the discussion of various existing fragmentation strategies for temporal relations. The parallel query processing strategies and optimization alternatives will be the main focus in Section 4. Implementation issues of the parallel strategies will be addressed in Section 5. Section 6 contains a discussion of other possible parallel query processing strategies. Finally, we discuss the related work in Section 7, and the conclusions and future research work in Section 8.

2 Background Information

In this section, we introduce the fundamental concepts: the temporal data model, a classification of temporal data and queries.

2.1 Data Model & Definitions

In the temporal data model, time points are regarded as integers { 0, 1, $\cdots$, *now* } which are monotonically increasing and where *now* is a special marker that represents the current time. 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 lifespan of a tuple [Cli87, Seg87]; we often say that a tuple $x<s,v,t_s,t_e> \in X$ *starts* at time point t_s and *ends* at time point t_e [1]. Both timestamps are commonly called *effective timestamps* (as opposed to *transaction timestamps*) [Sno87].

Definition 1 *Current tuple* is defined as a tuple whose TS value is a specific time point and whose TE value is "*now*". *History tuple* is defined as a tuple whose TS and TE values are specific time points (i.e. its TE value is not "*now*"). □

Definition 2 "A op c", where A is an attribute, op is a relational operator ($>$, $\geq$, $=$, $<$, $\leq$, $\neq$) and c is a constant, is called a *comparison predicate.* Similarly, "A op B" for attributes A and B is called a *join predicate.* A subclass of join predicates is called *temporal join predicate* if both attributes A and B are timestamps. □

Definition 3 A *query qualification* in conjunctive normal form is a conjunction ($\wedge$) of several terms each of which can be:

- a join predicate,

[1] We concentrate on time-interval temporal relations which may contain any number of time-varying attributes. One can regard a time point t as an "interval" [t,t+1), and the results in this paper remain applicable.

- a disjunction ($\vee$) of several comparison predicates involving the *same* attribute (e.g. "X.V=1 $\vee$ X.V=2").

$\square$

Definition 4 $P(R_1, \cdots, R_m)$, $m \geq 1$, represents a query qualification involving relations R_1, $\cdots$, R_m. $\square$

Definition 5 Substitution: Given a query qualification $P(\cdots, X, \cdots)$ and a tuple $x \in X$, we denote $P(\cdots, x, \cdots)$ as a new query qualification obtained by substituting for all attributes of X in $P(\cdots, X, \cdots)$ the values from the tuple x. Note that $P(\cdots, x, \cdots)$ may become unsatisfiable. $\square$

Definition 6 A general class of queries, called Temporal Select-Join (TSJ), is defined as the set of relational algebraic expressions of the following form:

$$\sigma_{P(R_1,\cdots,R_m)}\ (R_1 \times \cdots \times R_m) \qquad \text{or} \qquad \sigma_{P(R_1,\cdots,R_m)}\ (R_1, \cdots, R_m)$$

where $P(R_1, \cdots, R_m)$ or simply P is a query qualification and $R_1, \cdots, R_m$, $m \geq 1$, are temporal relations. $\square$

The semantics of three temporal operators (called *snapshot* operators), which are commonly found in temporal queries — between, intersect and as of, are also of interest here. All these operators are syntactic sugar — they can be directly specified in terms of comparison predicates involving only timestamps. Their use allow us to express queries more intuitively and to "view" the database content that is active during a particular time interval or time point.

Definition 7 Given a time point T and a time interval $[t_s, t_e)$, "T between $[t_s, t_e)$" holds if and only if "$t_s \leq T \wedge T < t_e$" holds. $\square$

Definition 8 Given two time intervals [TS,TE) and $[t_s, t_e)$. "[TS,TE) intersect $[t_s, t_e)$" holds if and only if "TS<t_e $\wedge$ t_s<TE". "$\sigma_P(R_1, \cdots, R_m)$ intersect $[t_s, t_e)$" is defined as

$$\sigma_{P\ \wedge\ [R_1.TS, R_1.TE)\ \text{intersect}\ [t_s, t_e)\ \wedge\ \cdots\ \wedge\ [R_m.TS, R_m.TE)\ \text{intersect}\ [t_s, t_e)}\ (R_1, \cdots, R_m).$$

That is, we deal with tuples that are active during the interval $[t_s, t_e)$. $\square$

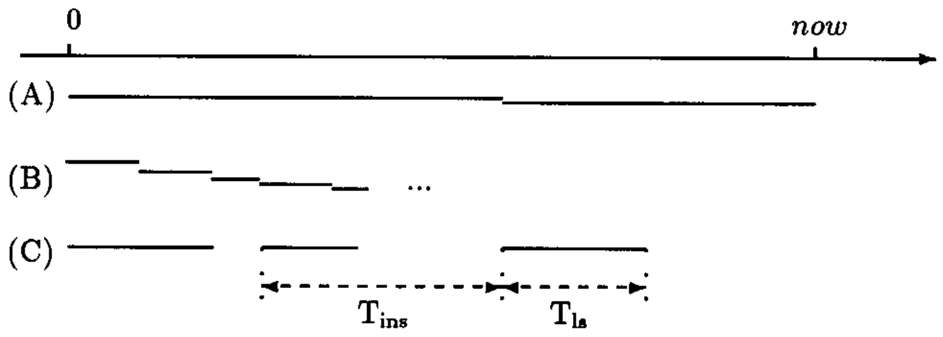


Figure 1: Characterization of temporal data

Definition 9 Given a time interval [TS,TE) and a time point t, "[TS,TE) as of t" holds if and only if "t between [TS,TE)" holds. However, "[TS,TE) as of *now*" is equivalent to "TE=*now*". "$\sigma_P(R_1, \cdots, R_m)$ as of t" is defined as

$$\sigma_{P \wedge [R_1.TS, R_1.TE) \text{ as of } t \wedge \cdots \wedge [R_m.TS, R_m.TE) \text{ as of } t}(R_1, \cdots, R_m).$$

Note that this operator is a special case of the intersect operator. That is, we deal with tuples that are active at the time point t. □

2.2 Temporal Data

Temporal data has several characteristics and statistical properties that are of interest here. These characteristics and statistics represent valuable information that provides some guidelines on evaluating a proposed scheme.

Time-varying attributes (or similarly relationships) can be classified as *continuous* or *non-continuous* [Seg87]. A continuous time-varying attribute is an attribute which must have a valid non-null value during the object's lifespan, whereas a non-continuous attribute may not have a valid value (or equivalently it has a null value) during some period of time. For example, the stock price can be considered as a continuous time-varying attribute (types A and B in Figure 1) while a person checking out a book from a library can be a non-continuous time-varying relationship (type C in Figure 1).

In addition to the above characteristics, we will make use of several statistics. First, the lifespan of relation (denoted as TR_{ls}) is assumed to be [0,*now*), and is continuously expanding. The effective lifespan of an individual tuple, which we mentioned earlier, is specified by the pair of timestamps — TS and TE. The average tuple lifespan is represented by $\overline{T_{ls}}$ and the average time between two consecutive insertions of tuples to a relation is represented by $\overline{T_{ins}}$, as depicted in Figure 1. The mean rate of insertion of tuples into a relation (denoted as λ) is $1/\overline{T_{ins}}$. For continuous attributes, these two figures are directly related: $\overline{T_{ins}}$ is $\overline{T_{ls}}/\overline{Ns}$, where $\overline{Ns}$ is the average number of surrogates in the temporal relation. These statistics will be used to characterize the performance of the proposed schemes to be discussed shortly.

2.3 Classification of Queries

Common queries can be broadly classified using different criteria. Firstly, queries can be partitioned into select and join queries.

Select Query This is often viewed as n-dimensional range search, e.g. selecting employee records whose age is between 40 and 50: $\sigma_{40 \leq age \leq 50}$ (Employee). Various indexing techniques can be used to speed up the search process.

Join Query We consider two types of joins — *inequality join* and *equi-join*. Inequality joins are often expensive to process while equi-joins can be processed efficiently by a number of algorithms such as sort-merge joins and hash joins.

Secondly, we can partition attributes of the query qualification into 1) non-time attributes only and 2) timestamps only. Together with the select and join query classification, we obtain a 2 by 3 matrix:

	non-time attributes only	timestamps only
select	$\sigma_{V \geq 50}$ (R_1)	$\sigma_{TS<50}$ (R_1)
equi-join	$\sigma_{R_1.U=R_2.V}$ (R_1, R_2)	$\sigma_{R_1.TS=R_2.TS}$ (R_1, R_2)
inequality join	$\sigma_{R_1.U<R_2.V}$ (R_1, R_2)	$\sigma_{R_1.TE<R_2.TS}$ (R_1, R_2)

Note that a query can contain a mixture of these qualifications.

Thirdly, any query in these six categories can further be augmented with a snapshot operator, forming a snapshot query. We distinguish two types of snapshot queries: *snapshot select* and *snapshot join* queries. The first type involves selection based on attribute values as of a certain time: e.g. $\sigma_{S=s1}$ (R) as of t_i while the second type involves joining several relations as of a certain time: e.g. $\sigma_P(R_1, \cdots, R_m)$ as of t_i. Note that since snapshot operators also involve timestamps, queries with both a snapshot (especially as of) operator and other qualification on timestamps may produce null responses if the "combined" query qualifications evaluates to false. For example, the following queries return null responses:

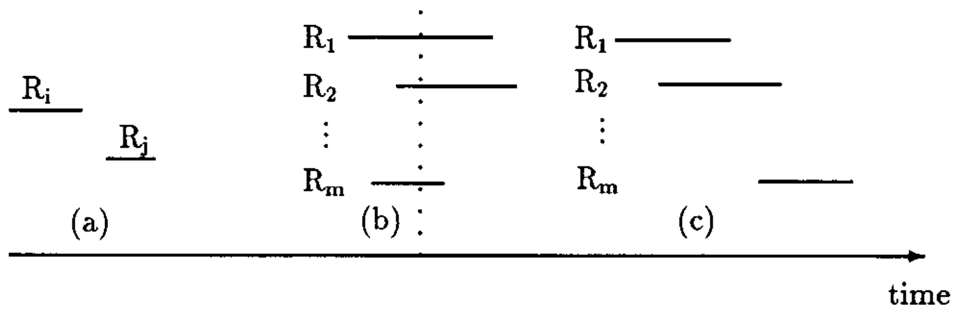


Figure 2: Classes of temporal joins

- $\sigma_{R.TE<t}(R)$ as of t $\equiv \sigma_{R.TE<t \wedge R.TS \leq t \wedge t<R.TE}(R)$
- $\sigma_{R_1.TE<R_2.TS}(R_1, R_2)$ as of t
  $\equiv \sigma_{R_1.TE<R_2.TS \wedge R_1.TS \leq t \wedge t<R_1.TE \wedge R_2.TS \leq t \wedge t<R_2.TE}(R_1, R_2)$

2.4 Temporal Queries

In this section, we discuss several subclasses of TSJ join queries. Each subclass has a restricted form of query qualification and their characterizations can be intuitively stated as follows:

Disjoint Join The join condition between two tuples does not require that their intervals overlap, as illustrated in Figure 2(a). For example, queries with join conditions "$R_i.TE<R_j.TS$" or "$R_i.TE<R_j.TE$" belong to this category.

Overlap Join The join condition between two tuples requires that their intervals share a common time point, i.e. they overlap. We consider two special kinds of overlap joins whose formal definitions will be presented shortly:

TSJ_1 — All participating tuples that satisfy the join condition share a common time point, as illustrated in Figure 2(b). For example, finding a complex "event pattern" in which all events occur during the same period of time can be viewed as a TSJ_1 join query.

TSJ_2 — The tuples that satisfy the join condition overlap in a "chain" fashion, as illustrated in in Figure 2(c). However, the participating tuples that satisfy the join condition do not have to have a common time point. For example, finding an event pattern in which events occur in some overlapping sequence can be viewed as a TSJ_2 join query.

Note that TSJ_1 queries also belong to TSJ_2.

Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}\ (R_1,\cdots,R_m)$, we construct a join graph (denoted as G) from the query qualification $P(R_1,\cdots,R_m)$ using Algorithm 1. Base on the join graph, we will then be able to formally define TSJ_1 and TSJ_2 join queries.

Algorithm 1 Join Graph: There are m nodes in the join graph G; each node represents an operand relation R_i, $1 \leq i \leq m$, and is labeled with the name of that relation. We add an undirected edge between nodes R_i and R_j ($i \neq j$) into G if the following condition is satisfied:

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

That is, for tuples $r_i \in R_i$ and $r_j \in R_j$, r_i and r_j span a common time point [3]. □

Definition 10 A query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}\ (R_1,\cdots,R_m)$ belongs to TSJ_2 if the following conditions hold:

1. The number of operand relations in Q is greater than 1, i.e. $m>1$.
2. The join graph G constructed using the Algorithm 1 is a connected graph, i.e., all nodes in G are connected.

□

Definition 11 A TSJ_2 query is also a TSJ_1 query if the join graph G constructed using the Algorithm 1 is a *fully connected* graph. In other words, for all i and j such that $1 \leq i \leq m$, $1 \leq j \leq m$ and $1 \neq j$,

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

□

Theorem 1 Common Time Point: Given a query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}\ (R_1,\cdots,R_m) \in TSJ_2$. 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) span a common time point. □

[2] Testing implications can be readily achieved via algorithms presented in [Ros80, Ull82, Sun89]. Moreover, semantic optimizations can be used to add edges in the graph.

[3] This condition is defined such that we can also handle the join predicate "X.TE=Y.TS" for a join of two relations.

Proof:

The proof basically follows from the definition of TSJ_1. Consider a m-tuple $<r_1, \cdots, r_m>$, where $r_k \in R_k$, $1 \leq k \leq m$, that satisfies the join condition $P(R_1, \cdots, R_m)$. Without loss of generality, let us assume that tuples r_i and r_j do not share a common time point, where $1 \leq i \leq m$ and $1 \leq j \leq m$. Then, exactly one of the following conditions holds:

1. r_i.TE < r_j.TS, i.e. r_i ends before r_j starts.
2. r_j.TE < r_i.TS, i.e. r_j ends before r_i starts.

In either case, the m-tuple $<r_1, \cdots, r_m>$ would not have been generated by the join because "r_i.TS$\leq r_j$.TE $\wedge$ r_j.TS$\leq r_i$.TE" is false and thus the m-tuple does not satisfy the join condition $P(R_1, \cdots, R_m)$. Q.E.D.

The class of TSJ_1 and TSJ_2 are temporal pattern queries (e.g. multi-way temporal joins) in which the lifespans of tuples intersect. The main characteristic of these join conditions is that a tuple $r_i \in R_i$ which starts at time t does not join with (theoretically) infinitely many "future" tuples $r_j \in R_j$ which start after r_i ends [4]. For example, Cartesian products across multiple relations (i.e. no join predicates) and the query with the join condition "R_i.TE<R_j.TS" do not belong to either TSJ_1 or TSJ_2. This characteristic is crucial in developing the parallel query processing schemes to be described later.

Examples of TSJ_1 query include the commonly called "natural time join" [Cli85, Cli87], "intersection joins" [Gun91] which join relations in first temporal normal form [Seg88]. Also, the following temporal join operators [All83] belong to TSJ_1 [5]:

meet-join(X,Y) — "X.TE=Y.TS"
contain-join(X,Y) — "X.TS<Y.TS $\wedge$ Y.TE<X.TE"
start-join(X,Y) — "X.TS=Y.TS $\wedge$ X.TE<Y.TE"
finish-join(X,Y) — "X.TS>Y.TS $\wedge$ X.TE=Y.TE"
equal-join(X,Y) — "X.TS=Y.TS $\wedge$ X.TE=Y.TE"
overlap-join(X,Y) — "X.TS<Y.TS $\wedge$ Y.TS<X.TE $\wedge$ X.TE<Y.TE"

Note that the overlap-join(X,Y) is asymmetric with respect to the operands. Here we define a symmetric version of the overlap-join(X,Y):

[4] Or conversely, a tuple $r_j \in R_j$ which starts at time t does not join with (theoretically) infinitely many "past" tuples $r_i \in R_i$ which ends before r_j starts.

[5] In [Leu90], we note that these temporal join operators are really shorthand for expressions involving only temporal join predicates.

intersect-join(X,Y) — "X.TS<Y.TE ∧ Y.TS<X.TE"

Example 1: Given temporal relations X, Y and Z. The following is a TSJ_1 query:

$$\sigma_{\text{contain-join(X,Y)} \wedge \text{contain-join(Y,Z)}} (X,Y,Z)$$

Although there is no explicit join predicate between relations X and Z, the TSJ_1 definition is still satisfied. However, the following is a TSJ_2 query but not TSJ_1:

$$\sigma_{\text{intersect-join(X,Y)} \wedge \text{intersect-join(Y,Z)}} (X,Y,Z)$$

Note that the join between X and Y, and that between Y and Z are TSJ_1 queries. In Section 6, we will discuss this type of join query in more detail. Until then, we will focus on TSJ_1 queries. □

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 [6]:

Studio(Sname,Head,TS,TE)
Dir(Dname,Sname,TS,TE)
Stars(Star,Dname,TS,TE)

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

Example 2: To find the head of a studio and the director who worked for the studio at the same time:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname}} (\text{Studio,Dir})$$ [7]

□

Example 3: To find all combination of studio heads, film stars and directors such that the star acted in a film that the director directed at the studio during the same period of time:

[6] These relations are adopted from the examples in [Cli87].
[7] 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 the major optimization issue.

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

where P is "Studio.Sname=Dir.Sname $\wedge$ Dir.Dname=Stars.Dname". □

Example 4: To find all combination of studio heads, film stars and directors such that the studio head was a star in a film that the director directed at the studio during the same period of time:

$$\sigma_{\text{P1} \wedge \text{P2}} \text{(Studio,Dir,Stars)}$$

where P1 is "Studio.Sname=Dir.Sname $\wedge$ Head=Star $\wedge$ Dir.Dname=Stars.Dname" and P2 is "intersect-join(Studio,Dir) $\wedge$ intersect-join(Studio,Stars) $\wedge$ intersect-join(Dir,Stars)". □

Example 5: Consider an example in a different application domain: EMP(Eno,Dept,TS,TE) and DEPT(Dno,Floor,TS,TE) which store the employee and department information. To find the employees who ever worked on the first or the second floor:

$$\sigma_{\text{intersect-join(EMP,DEPT)} \wedge \text{Dept=Dno} \wedge \text{(Floor=1} \vee \text{Floor=2)}} \text{(EMP,DEPT)}$$

□

3 Temporal Data Distribution

We consider a generic "shared-nothing" multiprocessor database machine [Sto86, DeW90] as shown in Figure 3, which has n nodes connected via an interconnection network. Each node has a processor, some main memory and secondary storage devices (such as magnetic disks) [8]. In this section, we discuss how a given temporal relation can be partitioned and distributed in such a database machine, and the tradeoffs involved with respect to the efficiency of processing the various classes of query listed earlier. Readers should keep in mind that whether or not a query qualification involves the partitioning attribute generally has a predominant impact on query processing efficiency, and therefore a specific fragmentation strategy can facilitate processing some kinds of queries while it may cause other queries to be more expensive to process.

A number of well-known fragmentation strategies have been proposed and implemented in

[8] We use the terms nodes and processors interchangeably — a processor also refers to its processing capability and associated disks.

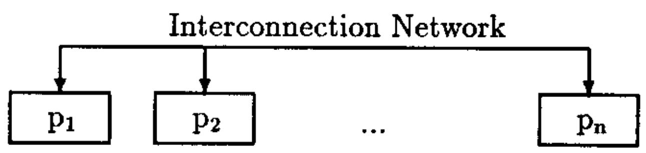


Figure 3: A generic "shared-nothing" multiprocessor database machine

multiprocessor database machines [Ter85, DeW90]. They include:

- round-robin
- hashing
- range-partitioning

Hashing and Round-robin

We first discuss the hashing and round-robin strategies as both strategies attempt to spread the workload more evenly among processors by distributing tuples *randomly.* For our discussion, we consider relations X(S,U,TS,TE) and Y(S,V,TS,TE) which can be fragmented on any attributes such as S, V or TS [9]. We first consider that both relations are fragmented on their surrogates, i.e., S is the partitioning attribute.

Select queries with an equality predicate involving the partitioning attribute (e.g. X.S=1) are processed by a unique processor. On the other hand, selection on non-partitioning attributes (e.g. X.TS=50) and range-searching on the partitioning attribute (e.g. $50 \leq X.S \leq 100$) generally involves all processors. Select queries can be processed in parallel if there is an index on the search attribute (i.e. the TS timestamp). For example, consider a range search for "$45 \leq X.TS \leq 50$". However, the parallel search is efficient only when the number of matching tuples on each processor is relatively large. The reason is that the overhead of starting a subtransaction on a processor becomes relatively small when the processor has to work on a large amount of tuples. On the other hand, if the select query returns only a few matching tuples, the parallel search is relatively inefficient as most processors would have wasted their resources on starting a subtransaction which retrieves only a very few or even no matching tuples.

Let us now consider join queries. A naive approach to achieve parallelism is to fully and dynamically replicate the smaller relation in all processors, which is very expensive unless the

[9] The partitioning attribute can be a single attribute such as S or a "composite" attribute composed of multiple attributes such as S and TS.

relation is small. This expensive data movement does not occur when the join condition implies an equality predicate on the partitioning attribute (i.e. X.S=Y.S) and both operand relations are fragmented using the same partitioning function (e.g. the same hash function). In general, data movement is required for inequality joins such as temporal join operations. For example, consider contain-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TE<X.TE$". We note that tuples with close (but unequal) TS values are likely to be stored at different processors due to the randomness of hashing or round-robin strategies regardless of the partitioning attribute. Unless there is an additional qualification that limits the data that has to be examined, an expensive data movement is required prior to the execution of local joins, in parallel, on each processor.

A snapshot query generally requires all processors to participate since qualifying tuples are likely to be stored at all processors. As in the case of a select query, indexing techniques such as [Elm90, Kol90, Leu92] can be used to speed up the processing locally.

Choosing the partitioning attribute involves several issues. First, the decision will depend on the frequency of various queries. For example, if most (both select and join) queries involve the surrogates, the surrogates may be good candidates for the partitioning attribute. Second, the decision may also depend on the attribute domain itself. As an example, suppose the domain of a time-varying attribute consists of only a small number of entities, say 50. For large relations, hashing on this attribute may produce a skewed data distribution and thus have an adverse effect on query processing efficiency. Note that although surrogates are guaranteed to be unique in the database system (i.e. the values will not be re-used), a surrogate value may appear in a temporal relation more than once. To use the uniqueness of data values as the criterion of selecting the partitioning attribute, one may therefore choose a composite attribute <S,TS> [10].

Range-partitioning

The range-partitioning strategy can be characterized by clustering and storing tuples with close (or equal) partitioning attribute values at the same processor. That is, the partitioning attribute is also a clustering attribute. We discuss three types of range-partitioning strategies for a temporal relation Y(S,V,TS,TE).

The first strategy is to partition a relation Y along a non-time attribute dimension (e.g. V) as illustrated in Figure 4. This strategy partitions the attribute domain into a fixed number of intervals, and tuples in the same range are stored at the same processor. For example, as shown in Figure 4, tuples with V values in the range of $[v_1,v_2)$ are stored in processor p_1. The query processing algorithms for select, join and snapshot queries under this partitioning strategy are

[10] For continuous time-varying attributes (i.e., each object has one attribute value at any point in time), a value of the composite attribute <surrogate,timestamp> can uniquely determine a tuple. For this reason, one can also choose <S,TE> as the partitioning attribute.

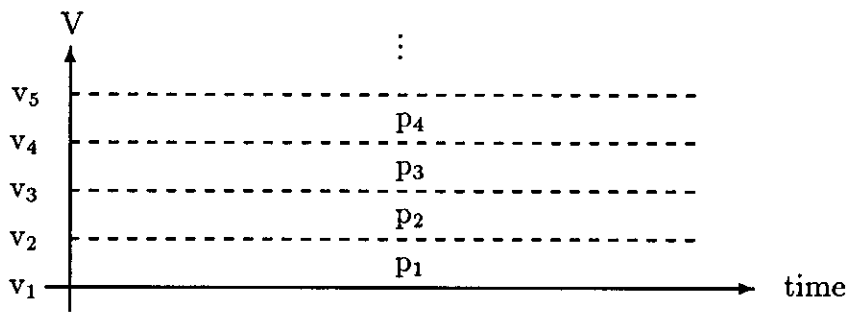


Figure 4: Range-partitioning along non-time attribute

similar to the hashing and round-robin schemes that we discussed earlier. For example, select queries with an equality predicate involving the partitioning attribute (e.g. V=1) are executed in a single processor. However, there are some (and perhaps slight) differences. First, range search queries on the partition attributes may sometimes be executed on only a subset of the processors. For example, a range search "$Y.V<v_3$" is performed only at processors p_1 and p_2. For an equi-join involving the partition attribute (e.g. X.U=Y.V), the join can be executed in parallel without data movement if both join attributes are the partitioning attribute and both relations are fragmented using the same range-partition function.

Usually the number of partition ranges is the same as the number of processors in the system and therefore each processor is assigned with a single range. Recently it has been proposed in [DeW90] that the domain is partitioned into a large number of smaller ranges and thus each processor is responsible for more than one range. This approach enables processing select queries more efficiently while keeping the workload among processors more balanced at the same time. This work, however, does not address join queries (including temporal joins) and snapshot queries.

The second strategy is to partition the relation along a timestamp attribute (e.g. TS) as illustrated in Figure 5. For example, tuples that start during the interval $[t_1,t_2)$ are stored in processor p_1. Using this scheme, equi-joins on non-partitioning attributes (e.g. non-time attributes) become more expensive to process as many tuples have to be moved among processors. However, partitioning relations on timestamp attributes may be a good alternative for temporal query processing — below we discuss this scheme in more detail.

Let the processors be denoted as p_i, for $1 \leq i \leq n$. There are a total of n_{pi} intervals in the partition function:

$$[t_1,t_2), \quad \ldots, \quad [t_{n_{pi}-1},t_{n_{pi}}), \quad [t_{n_{pi}},t_{n_{pi}+1})$$

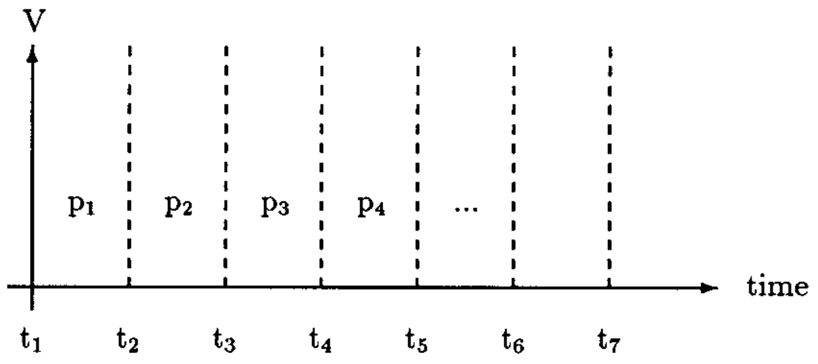


Figure 5: Range-partitioning along timestamp

We refer to t_i and $[t_i,t_{i+1})$ as partition boundaries and partition intervals respectively. Partitioning relations on TS (and TE respectively) timestamp is called TS (and TE respectively) range-partitioning. As the relation lifespan is 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 partition intervals 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 [DeW90]: an interval $[t_j,t_{j+1})$ is assigned to p_i if i equals j modulo n. For TS range-partitioning, processor p_i stores a fragment of X, denoted as X_i, which contains tuples of X that start during the interval $[t_i,t_{i+1})$, i.e., "x.TS between $[t_i,t_{i+1})$" holds. Similarly for TE range-partitioning, X_i contains tuples that end during the partition interval.

Consider that both relations X and Y are TS range-partitioned using the same partition function. That is, tuples that start during the interval $[t_i,t_{i+1})$ are stored at processor p_i. For contain-join(X,Y), tuples with close TS values are likely to be clustered within the same processor and therefore a pair of tuples that satisfy the join condition are likely to be stored at the same processor. However, a partition boundary may lie between the start times of tuples x and y that satisfy the join condition so x and y are actually stored at different processors. In short, processing temporal join in parallel may still require dynamically copying some fraction of relations between the processors.

For range search queries and snapshot select queries involving the partition attributes, we note that the queries may sometimes be executed on only a subset of the processors. For example, suppose we want to find the attribute values as of a certain time t_s which falls in the partition interval $[t_i,t_{i+1})$. As qualified tuples can start at any time earlier than t_s, processors from p_1 to p_i (which may be only a subset of the processors) would generally have to participate in the search, unless there is some additional information which would limit the search to a smaller subset of processors.

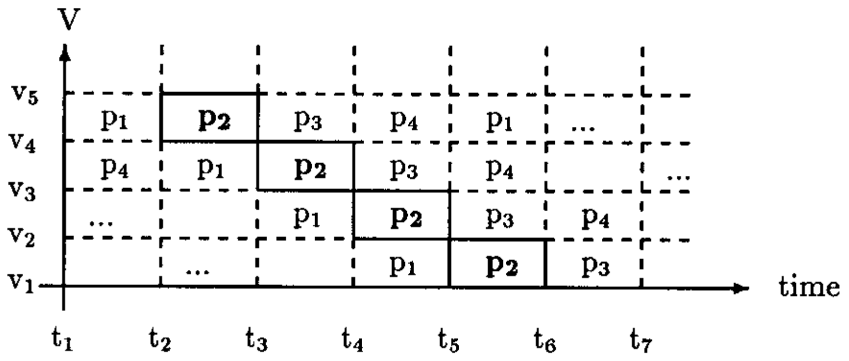

V
v_5
v_4
v_3
v_2
v_1
p_1 p_2 p_3 p_4 p_1 ...
p_4 p_1 p_2 p_3 p_4 ...
... p_1 p_2 p_3 p_4
... p_1 p_2 p_3
time
t_1 t_2 t_3 t_4 t_5 t_6 t_7

Figure 6: Two dimensional range-partitioning a temporal relation

In [Leu90] we noted that sorting relations on different timestamps may improve the efficiency of processing some queries. For example, processing contain-semijoin(X,Y) using stream processing algorithms requires only minimal buffer space (enought for one tuple from each relation) when X is sorted on TS and Y is sorted on TE. Consider another example query: meet-join(Y,X) whose join condition is "Y.TE=X.TS". These queries may be more efficiently processed if the relation Y is range-partitioned on the TE timestamp and X is range-partitioned on the TS timestamp. In other words, selecting a particular timestamp as the partitioning attribute may also depend on the types of temporal join operations.

The above two types of range-partitioning schemes can be considered as one dimensional as there is only one partitioning attribute. A third strategy is to partition the relation in a two dimensional fashion; one can imagine we superimpose a Grid File structure [Nie84] on the search space. Each grid (i.e. rectangle) can be of different size and is assigned to a specific processor. One possible approach is illustrated in Figure 6: we partition the search space horizontally and vertically into regular bands, resulting in a number of rectangular grids of the same size. Grids within a vertical band are assigned to processors in a "staggering" fashion. For example, processor p_2 stores grids with solid boundaries as shown in Figure 6.

We note several interesting points with respect to query processing compared with the one dimensional schemes. For simplicity of explanation, let us compare the TS range-partitioning scheme and the above two dimensional scheme (i.e. the TS timestamp is the partitioning attribute in both cases). For the two dimensional scheme, a range search may be executed on a subset of processors (as in the TS range-partitioning scheme). For example, tuples with condition "$X.V>v_2 \wedge X.V<v_3 \wedge X.TS=t_4$" can be retrieved by processor p_2. Range search on only one partitioning attribute, however, involves some optimization issues. For example, in the TS range-partitioning scheme (i.e. one dimensional), a search on "$X.TS=t_4$" is performed only at a single processor (i.e. p_4) whereas in the 2-dimensional scheme, the search is performed

at all processors. That is, the two dimensional scheme attempts to spread the workload across all processors and thus reduce the query response time. Ideally, the reduced query response time would be approximately about 1/n (where n is the number of processors) of the response time in the one dimensional scheme plus the overhead associated with coordinating the parallel search. This strategy may be beneficial if the number of matching tuples on each processor is relatively large. When the number of matching tuples is small (e.g. only a few tuples were inserted at time point t_4), the gain due to parallelism may not be offset by the associated overhead. For an equi-join to be executed in parallel without data movement, the join condition has to imply "X.V=Y.V $\wedge$ X.TS=Y.TS" since there are two partitioning attributes. Note that the class of join queries to be processed in this fashion is smaller as the join condition is more restrictive. For this reason, join queries and snapshot join queries are generally more costly to process.

In the remainder of this paper, we will further develop parallel query processing schemes based on range-partitioning on timestamp attributes and show that these schemes may be good alternatives for complex temporal pattern queries and snapshot queries.

4 Parallel Temporal Query Processing

In this section, we discuss some parallel processing strategies for complex temporal queries based on the following approach:

> Relations are range-partitioned along the time dimension. Each processor will work *independently* on the partition intervals that are assigned to it. The query response is the union of results from all processors.

We first introduce the notion of checkpointing the execution state of a query, and show that once sufficient state information has been constructed at every processor (i.e., replicating some tuples between processors), queries can be processed in parallel without additional data transfers. We discuss several optimization strategies and present a preliminary quantitative analysis of our approach.

Before we proceed, we differentiate two classes of range-partition function: *homogeneous* and *heterogeneous*. In this section, we focus on TSJ_1 join queries whose operand relations are homogeneously range-partitioned.

Definition 12 Relations are referred to as homogeneously range-partitioned if their partition functions are the same, i.e., all the partition boundaries are the same. If different range-partition functions are used, it is referred to as heterogeneous. □

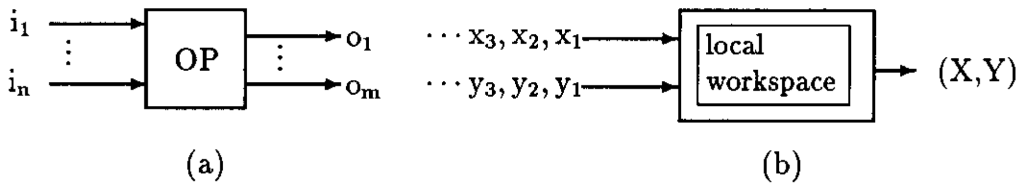


Figure 7: (a) A generic stream processor and (b) a merge-join stream processor

4.1 Notion of Checkpointing Execution State

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

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 stream(s) and produces one or more output data streams. A generic stream processor is shown in Figure 7(a). 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 7(b); the output from the join operation 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 heavily depend on the sort ordering of input streams, data statistics and the operation itself. For example, 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 one of 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. In such a case, the merge-join is actually a nested-loop join.

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 8. For the sake of simplifying our discussion, we assume that both X and Y are sorted on a timestamp (i.e. 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 point t, the execution state of the stream processor includes:

- state information, denoted as $s_t(Q)$, stored in the local workspace of the stream processor.
- $dsp_t(X)$ and $dsp_t(Y)$: the data stream pointer for X and Y respectively which represents the position of the data stream at which the stream processor has read so far. As we mentioned earlier, data stream elements are accessed one at a time using the data stream pointer and in a specified ordering of the data stream.

The notion of checkpointing the execution state of a query has the following characteristics in terms of the state information and data stream pointers:

- The execution state at time t is stored in a checkpoint, denoted as $ck_t(Q)$.
- Consider a time point t' which is greater than t. The execution state at t' is a function of $ck_t(Q)$ and all tuples in the data stream X (and Y respectively) between $dsp_t(X)$ (and $dsp_t(Y)$ respectively) and the first tuple whose TS value is greater than t'. That is, the execution state at t' contains sufficient information so that re-reading the portions of data streams prior to $dsp_t(X)$ and $dsp_t(Y)$ 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 to put it simply, at any time the state information of a join query consists of a subset of tuples of operand relations. One can think of the state information of the query at each checkpoint as the union of the state information of individual operand relation.

State Information at Partition Boundaries

We now outline the parallel processing strategy for TSJ_1 queries. Given that relations are homogeneously range-partitioned on timestamp attributes (TS or TE), i.e., processor p_i

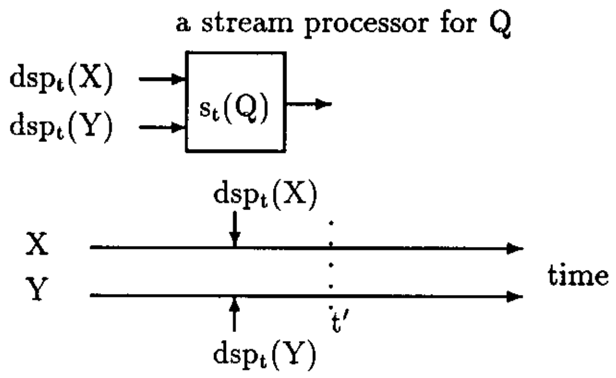


Figure 8: Checkpointing the execution of a stream processor for query Q

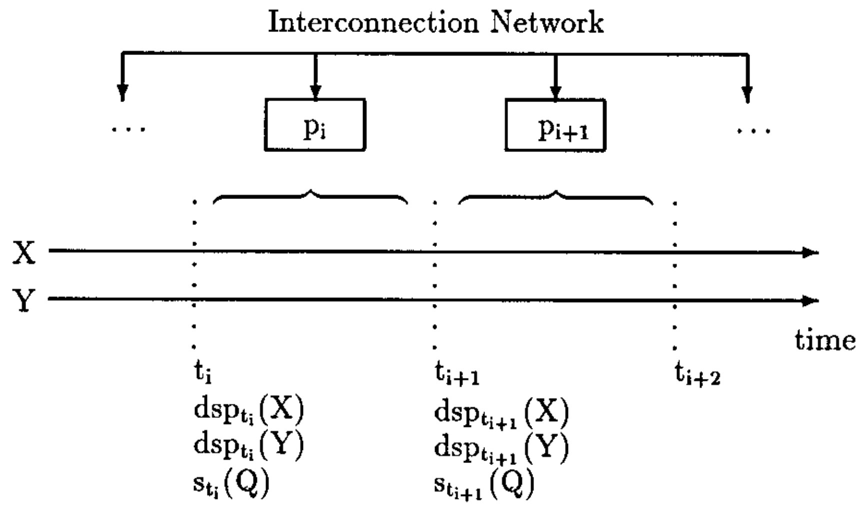


Figure 9: Constructing state information at partition boundaries

is assigned an interval $[t_i, t_{i+1})$ as depicted in Figure 9. For the moment, we assume that the data stream pointers at the partition boundary t_i, $dsp_{t_i}(X)$ and $dsp_{t_i}(Y)$, are "pointing" at the relation fragments X_i and Y_i respectively. Given a query Q, the strategy is to construct *sufficient* state information at *every* partition 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 9, p_i will process its local fragments X_i and Y_i using the state information $s_{t_i}(Q)$. Similarly, p_{i+1} will process X_{i+1} and Y_{i+1} using the state information $s_{t_{i+1}}(Q)$. In general, the strategy has three distinct phases:

Replication Phase Construct sufficient state information at 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 (and eliminating duplicates) the results returned from all processors.

Let us emphasize that the TSJ_1 queries are multi-way temporal joins, i.e., there are m operand relations. For the sake of simplicity of exposition, we concentrate on joins of two relations unless we state otherwise.

4.2 Replication Phase

In this subsection, we discuss the construction of state information at each partition boundary for a query to be processed in parallel by each processor.

Definition 13 The *state predicate* of a relation R is a query qualification on R [11]. □

The state predicate of a relation is used to construct the state information of the relation required on each processor. The issue is how to derive the state predicate of individual relations using the user query qualification. For a query involving relations X and Y whose query qualification is P(X,Y), the Algorithm 2 shows the simpliest approach to derive state predicates from P(X,Y).

Algorithm 2 The state predicate for the relation X, denoted as $P|_x$, is obtained by replacing the following predicates in P(X,Y) with "true":

[11] Recall that a query qualification on R is a conjunction of several terms and a term is a disjunction of some comparison predicates involving the same attribute.

- join predicates and
- comparison predicates that involve the relation Y.

That is, $P|_x$ contains only comparison predicates involving only X in P(X,Y). Similarly, we can obtain the state predicate for the relation Y denoted as $P|_y$. □

Example 6: Consider the film industry examples presented earlier. The query to find the head of a studio that the director "Fred" worked for at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname} \wedge \text{Dir.Dname=Fred}}\text{(Studio,Dir)}$$

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

Deriving state predicates using the Algorithm 2 can be "improved" by propagating constraints between attributes, that is, the derived state predicates can become more "restrictive". For example, consider the query to find the head of the studio "MGM" and the directors who worked for the studio at the same time is:

$$\sigma_{\text{intersect-join(Studio,Dir)} \wedge \text{Studio.Sname=Dir.Sname} \wedge \text{Studio.Sname=MGM}}\text{(Studio,Dir)}$$

Using the above 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. For the rest of this subsection, we describe a mechanism in which bounds on timestamp values can be propagated between relations; propagating other types constraints (e.g. equality comparison predicates in this example) can be achieved in a similar manner [Ull82, Chak84, Jar84, She89].

We first consider constraints (i.e. upper and lower bounds) on timestamps of individual relation R:

$$\text{ts}^- \le \text{TS} < \text{ts}^+ \text{ and } \text{te}^- \le \text{TE} < \text{te}^+$$

where ts^-, ts^+, te^- and te^+ are constants. That is, the values of TS timestamp are bounded by the interval $[\text{ts}^-, \text{ts}^+)$ and those of TE are bounded by $[\text{te}^-, \text{te}^+)$. If a particular timestamp is not explicitly constrained, its default constraint interval is $[0, \mathit{now})$. Since in our data model

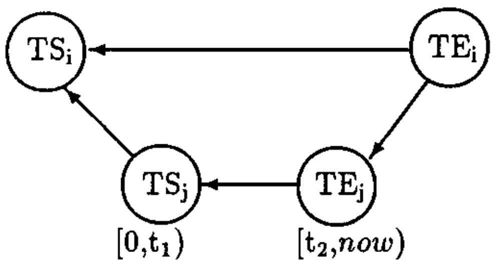


Figure 10: Constraints on timestamps: upper and lower bounds

the TS value must be smaller than the TE value in each tuple, the two constraint intervals are therefore related:

- If $te^- \leq ts^-$, the constraint on TE becomes: $ts^- + 1 \leq TE < te^+$.
- If $te^+ \leq ts^+$, the constraint on TS becomes: $ts^- \leq TS < te^+ - 1$.

That is, the data values of TS or TE can further be constrained. From now on, without loss of generality, we assume that "$ts^- < te^-$" and "$ts^+ < te^+$" hold for each individual relation in our discussion. Note that if the lower bound is larger than or equal to the upper bound, the query response must be necessarily null.

Definition 14 Given numbers $a_1, \cdots, a_m$, for some $m>1$, $\mathsf{max}(a_1, \cdots, a_m)$ returns a_j such that $a_j \geq a_i$ where $1 \leq j \leq m$ and $1 \leq i \leq m$ while $\mathsf{min}(a_1, \cdots, a_m)$ returns a_j such that $a_j \leq a_i$ where $1 \leq j \leq m$ and $1 \leq i \leq m$. □

The relationship among constraints on timestamps can be easily explained and determined using a constraint graph structure: Algorithm 3 can be used to construct a constraint graph G using the query qualification P [12].

Algorithm 3 Constraint Graph Construction [13]:

1. For each timestamp T of an operand relation, there is a node (labeled with T) in the graph G. Each node is tagged with a pair of values representing the upper and lower

[12] Similar constraint graph construction algorithms appear in [Ull82, Chak84, Jar84].

[13] Note that whether or not P has a comparison predicate that involves the operator "$\neq$" and a timestamp (e.g. "$R_i.TS \neq t_1$") does not have any impact on the graph construction algorithm.

bound of the constraint on the timestamp data values. For example, in Figure 10(a), the TS and TE timestamp values are bounded by $[ts^-, ts^+)$ and $[te^-, te^+)$ respectively.

2. For every relation R_i, a solid directed arc from the node R_i.TE to node R_i.TS is added to G.

3. If "$T_1<T_2$" is a predicate in P for any timestamps T_1 and T_2, a solid directed arc from the node T_2 to the node T_1 is added to G. Similarly, a dotted directed arc from T_2 to T_1 is added to G if "$T_1 \leq T_2$" is in P. [14]

4. If "$T_1=T_2$" is a predicate in P for any timestamps T_1 and T_2 of different relations, we merge these nodes together resulting a single node (labeled with $\{T_1,T_2\}$) that represents both timestamps T_1 and T_2. The largest lower bound of nodes T_1 and T_2 becomes the lower bound of the new node $\{T_1,T_2\}$. Similarly, the smallest upper bound of T_1 and T_2 becomes its upper bound.

5. Given the graph G that is constructed so far, we detect if there is a cycle in G. A path from a node T_2 to a node T_1 exists if (1) there is a (solid or dotted) directed arc from T_2 to T_1, or (2) there is a directed arc from T_2 to another node T_3 and there is a path from T_3 to T_1. A cycle exists if there is a path from any node to itself. There are two cases when a cycle exists:

   - The cycle has at least one solid directed arc (which represents "<" relationship). In this case, the user qualification is identically false and thus the query produces a null response.
   - All the directed arcs in the cycle are dotted arcs (which represent "≤" relationship). In this case, all the nodes in the cycle are merged together as in the case of "=" relationship. The lower and upper bounds of the new node are determined accordingly.

Note that for any TSJ_1 query, all nodes in G are in partial ordering (i.e. G has one or more roots). □

Example 7: In Figure 10, we show the constraint graph for the following query:

$$\sigma_{R_j.TS<t_1 \wedge \text{contain-join}(R_i,R_j) \wedge R_j.TE \geq t_2} (R_i,R_j).$$

□

[14] In [Leu90], we noted that semantic query optimization can play a significant role in query optimization. Using additional semantic information regarding temporal relationships between timestamps, more arcs may be added in the graph.

When the constraint graph is constructed using the Algorithm 3, constraints can be propagated between nodes (using Algorithm 4 below) [15].

Algorithm 4 Constraint Propagation: The upper bounds are propagated from roots to leaves whereas the lower bounds are propagated in the opposition direction. Suppose that the constraint on a node T_1 is $[t_1^-, t_1^+)$ and that on a node T_2 is $[t_2^-, t_2^+)$.

1. Dotted arc: for a dotted directed arc from a node T_2 to a node T_1, the propagation of the upper bound of T_2 to T_1 results the new upper bound of T_1 being $\mathsf{min}(t_1^+, t_2^+)$. The propagation of the lower bound of node T_1 to T_2 results the new lower bound of T_2 being $\mathsf{max}(t_1^-, t_2^-)$.
2. Solid arc: If the arc from T_2 to T_1 is solid, the new upper bound of node T_1 is $\mathsf{min}(t_1^+, t_2^+ - 1)$. The propagation of the lower bound of T_1 to T_2 results the new lower bound of T_2 being $\mathsf{max}(t_1^-, t_2^- + 1)$.

□

Example 8: Consider Figure 10(b), the TS values of relation R_i (i.e. TS_i) are bounded by the interval $[0, t_1-1)$ while the TE values (i.e. TE_j) are bounded by $[t_2+1, now)$. The state predicate for the relation R_i becomes "$R_i.TS < t_1 - 1 \wedge R_i.TE \geq t_2+1$". □

Definition 15 The state information of a relation R at the partition boundary t_i, denoted as $s_{t_i}(R)$, is stored at processor p_i and contains:

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

where $P|_r$ is the derived state predicate for the relation R. That is, all qualified tuples (based on $P|_r$) whose lifespan intersects with the partition interval $[t_i, t_{i+1})$ and are not stored at processor p_i will be replicated at processor p_i. □

Given a user query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, the state predicates can be derived from the query qualification P using the Algorithms (2) - (4) resulting $P|_x$ and $P|_y$. As soon as the

[15] Similar constraint propagation algorithms appear in [Ull82, Chak84, Jar84].

state information of all operand relations at all partition boundaries have been constructed, the join phase, which is the focus of the following subsection, can proceed.

4.3 Join Phase & Merge Phase

For a user 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 (and eliminating duplicates) from all processors:

$$\bigcup_{1 \leq i \leq n_{pi}} \{ \sigma_P(X_i, s_{t_i}(Y)) \cup \sigma_P(Y_i, s_{t_i}(X)) \cup \sigma_P(X_i, Y_i) \}$$

where n_{pi} is the total number of partition intervals.

Theorem 2 Parallel Join Strategy: Given a query $Q \equiv \sigma_P(R_1, \cdots, R_m) \in TSJ_1$, the join strategy is a parallel join of local relation fragments and the state information at each processor:

$$\bigcup_{1 \leq i \leq n_{pi}} \{ \sigma_P((R_{1,i} \cup s_{t_i}(R_1)), \cdots, (R_{m,i} \cup s_{t_i}(R_m))) \}$$

where n_{pi} is the total number of partition intervals, and $R_{j,i}$ is the ith fragment (i.e. partition $[t_i, t_{i+1})$) of the relation R_j, $1 \leq j \leq m$, which is stored at processor p_i. □

Proof We sketch the proof of the correctness of this parallel join strategy for TSJ_1 queries (Theorem 2):

> Given a query $Q \equiv \sigma_P(R_1, \cdots, R_m) \in TSJ_1$. By the definition of TSJ_1, each m-tuple $<r_1, r_2, \cdots, r_m>$, where $r_j \in R_j$ for $1 \leq j \leq m$, that satisfies the join condition must have a common time point (denoted as t_c), as shown in Figure 11. That is, for each participating tuple $r_j \in R_j$, r_j satisfies the derived state predicate $P|_{R_i}$ (otherwise r_j will not be a component of the m-tuple $<r_1, r_2, \cdots, r_m>$). Without loss of generality, we assume that t_c falls into a partition interval $[t_i, t_{i+1})$ which is assigned to a processor p_i [16]. Specifically, "t_c between $[t_i, t_{i+1})$" holds. For every participating tuple, $r_j \in R_j$, $1 \leq j \leq m$, exactly one of the following conditions must hold:

[16] It is possible that the operand tuples share a common time interval that spans multiple partition intervals. In this case, the m-tuple may be produced by more than one processor. The final merge phase would eliminate the duplicates.

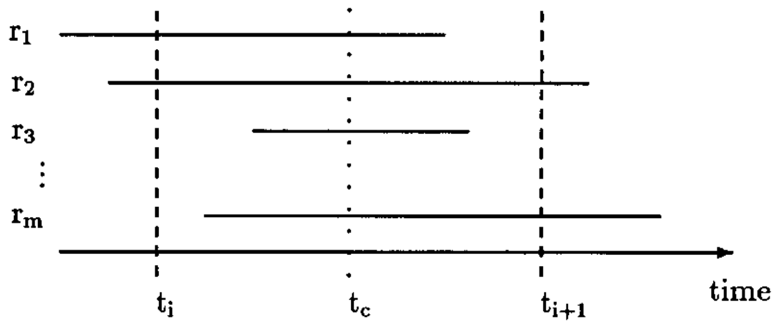


Figure 11: A m-tuple $<r_1, r_2, \cdots, r_m>$ that satisfies a TSJ_1 join condition

1. The relation R_j is TS range-partitioned and the tuple r_j starts during the partition interval $[t_i, t_{i+})$. That is, rmr_j is a tuple in the relation fragment stored at processor p_i.
2. The relation R_j is TS range-partitioned and the tuple r_j starts earlier than t_i and span the partition boundary t_i. That is, r_j should have been replicated at processor p_i as the state information.
3. The relation R_j is TE range-partitioned and the tuple r_j ends during the partition interval $[t_i, t_{i+})$. That is, rmr_j is a tuple in the relation fragment stored at processor p_i.
4. The relation R_j is TE range-partitioned and the tuple r_j ends at or later than t_{i+1} and span the partition boundary t_{i+1}. That is, r_j should have been replicated at processor p_i as the state information.

Therefore, each component in the m-tuple $<r_1, r_2, \cdots, r_m>$ can be found either in the state information or in the relation fragment. Hence, the m-tuple $<r_1, r_2, \cdots, r_m>$ is produced by the processor p_i. In other words, every m-tuple in the query response is produced by at least a processor in the parallel join processing strategy. Q.E.D.

4.4 Reducing State Information

The definition of the state information of a relation at a partition boundary as qualified tuples (based on the derived state predicates) that span the partition boundary is general enough to support the parallel query processing strategy for any query in TSJ_1 whose operand relations are homogeneously range-partitioned. In this subsection, we discuss some optimization opportunities in which the number of tuples replicated as state information can be reduced.

Asymmetry Property

First, we define the *asymmetry* property of operands in a TSJ_1 join query with respect to the TS and TE timestamps.

Definition 16 Given a TSJ_1 query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m)$. The relation R_k, where $1 \leq k \leq m$, has the asymmetry property with respect to the *TS* timestamp if the following condition is satisfied:

$$P(R_1, \cdots, R_m) \Rightarrow R_k.TS \geq R_i.TS, \text{ for all } 1 \leq i \leq m.$$

□

Definition 17 Given a TSJ_1 query $Q \equiv \sigma_{P(R_1,\cdots,R_m)}(R_1, \cdots, R_m)$. The relation R_k (where $1 \leq k \leq m$) in Q has the asymmetry property with respect to the *TE* timestamp if the following condition is satisfied:

$$P(R_1, \cdots, R_m) \Rightarrow R_k.TE \leq R_i.TE, \text{ for all } 1 \leq i \leq m.$$

□

For each m-tuple $<r_1, \cdots, r_m>$ that satisfies the join condition P, where $r_i \in R_i$ for $1 \leq i \leq m$, the asymmetry property with respect to the TS timestamp means that the tuple $r_k \in R_k$ must have the largest 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 while the relation X does not. Similarly, the asymmetry property with respect to the TE timestamp means that the tuple r_k must have the smallest 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 of the relation is redundant and thus the replication phase for that particular relation can be eliminated.

Theorem 3 Property of Redundant State Information: Given a TSJ_1 query $Q \equiv \sigma_P(R_1, \cdots, R_m)$, the condition under which the state information of a relation R_k (where $1 \leq k \leq m$) is redundant is:

The relation R_k has the asymmetry property with respect to the partitioning attribute (i.e. TS or TE timestamp).

□

Proof We sketch the proof for the Theorem 3 as follows:

We consider the case of the TS range-partitioning; the argument for the case of the TE range-partitioning is similar. Given a TSJ_1 join query, all the components of each m-tuple $<r_1, \cdots, r_m>$ where $r_i \in R_i$ $(1 \leq i \leq m)$ that satisfies the join condition must have a common time point. Also, given that relation R_k has the asymmetry property, the component r_k in the m-tuple must have the largest TS value and therefore the r_k.TS value must also be a common time point. Since the relation R_k is TS range-partitioned, the m-tuple must have been produced in the partition where the tuple r_k is stored. Hence, the tuple r_k need not be replicated to other partitions for the join process. Q.E.D.

Example 9: Consider the overlap-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TS< X.TE \wedge X.TE<Y.TE$". Suppose that the relation Y is TS range-partitioned. Only the state information of the relation X (but not relation Y) has to participate in the join phase. Therefore one has to replicate only tuples of relation X as state information, and the construction phase of state information of the relation Y can be eliminated. □

There are some interesting observations. First, the relation Y in the contain-join(X,Y) has the asymmetry property with respect to both TS and TE timestamps. For this reason, the state information of relation Y need not be constructed regardless of whether the relation is TS or TE range-partitioned. Second, the above two conditions have already accounted for the *equality* join predicates on timestamps. For example, the relation X in the meet-join(X,Y), whose join condition is "$X.TE=Y.TS$", has the asymmetry property with respect to the TE timestamp while the relation Y has the asymmetry property with respect to the TS timestamp.

Statistics

We discuss the definition and use several simple statistics on the data in each partition. We show how these statistics can further reduce data movement and discuss their pros and cons. Again, for the sake of simplifying our discussion, we focus on joins of two relations — X and Y.

Suppose that the query optimizer keeps the maximum and minimum of the TS and TE timestamp values for every relation fragment. In other words, the TS and TE timestamp values of each relation fragment Y_i are bounded by the intervals: $[Y_i.TS_{max}, Y_i.TE_{max})$, $[Y_i.TS_{min}, Y_i.TE_{min})$. Together with the user query qualification P, these statistics may further reduce data movement of the relation X. To illustrate this point, let us consider several examples. Suppose that the fragment Y_i of a relation Y is stored at processor p_i, and that the partition interval assigned to p_i is $[t_i,t_{i+1})$.

- 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 partition boundary t_i and whose TE values are smaller than $Y_i.TS_{min}$ 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 $Y_i.TE_{max}$.
- Consider the contain-join(X,Y) whose join condition is "$X.TS<Y.TS \wedge Y.TE<X.TE$". Intuitively, tuples in the relation X that span t_i and whose TE values are smaller than $Y_i.TE_{min}$ need not be sent to p_i as state information.
- Consider the meet-join(X,Y) whose join condition is "$X.TE=Y.TS$". We note that 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.

To eliminate redundant tuples from being replicated as state information, one can make use of the constraint propagation algorithm that we presented earlier. For example, consider the overlap-join(X,Y) whose constraint graph is shown in Figure 12. After the upper and lower bounds have been propagated, the TS values of relation X are bounded by the interval $[0,Y_i.TS_{max}-1)$ while the TE values are bounded by $[Y_i.TS_{min}+1,Y_i.TE_{max}-1)$. The state information of relation X at the partition boundary t_i therefore contains:

$$\{\, x \mid x \in X \wedge x.TS<t_i \wedge t_i \le x.TE \wedge P|_x(x) \wedge x.TS<Y_i.TS_{max}-1 \wedge Y_i.TS_{min}<x.TE \wedge x.TE<Y_i.TE_{max} \,\}$$

where the derived state predicate $P|_x$ is actually "true" for the overlap-join(X,Y). Note that the predicate "$x.TS<Y_i.TS_{max}-1$" is subsumed by the fact that "$x.TS<t_i$" and "$t_i \le Y_i.TS_{max}$" hold.

The tradeoffs for these optimizations include that keeping these statistics consistent for every relation fragment incurs some overhead. For example, suppose the state information of relation Y in the above example (see Figure 12) has to be constructed, the four statistics ($Y_i.TS_{max}$, $Y_i.TE_{max}$, $Y_i.TS_{min}$ and $Y_i.TE_{min}$) do not reflect the actual bounds on the timestamp values. This means that the actual bounds have to be computed after the state information of that relation has been completely constructed, and the results have to be broadcast to all senders (i.e. the processors which send tuples X). This may require substantial coordination overhead between processors. Moreover, determine if a tuple should be

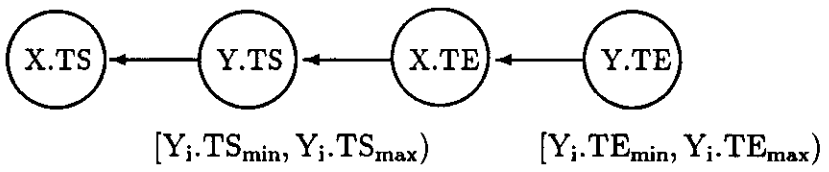

X.TS
Y.TS
X.TE
Y.TE
$[Y_i.TS_{min}, Y_i.TS_{max})$
$[Y_i.TE_{min}, Y_i.TE_{max})$

Figure 12: The overlap-join(X,Y): "X.TS<Y.TS ∧ Y.TS<X.TE ∧ X.TE<Y.TE"

sent to a processor requires the evaluation of a more complex qualification. However, for the operand relation that has the property of redundant state information discussed earlier, the four statistics of this relation do represent the actual bounds.

4.5 Participant Processors

For the parallel processing strategies that we discussed earlier, it can be determined a priori that for some TSJ_1 queries (whose operand relations are homogeneously range-partitioned), some processors necessarily return a null response when they join their local fragments using state information. Similarly, it can also be determined a priori that some processors need not participate in the replication phase. These situations may occur when the user query qualification contains some comparison predicates involving timestamps. To illustrate the idea, we first define the notion of *replication-intervals* and *join-intervals.*

Definition 18 A replication-interval is defined as the minimal interval with the property that only tuples whose partitioning attribute (i.e., TS or TE timestamps) value falls in the interval may be participated as state information [17]. □

Definition 19 A join-interval is defined as the minimal interval with the property that only tuples whose partitioning attribute (i.e., TS or TE timestamps) value falls in the interval can possibly contribute to the join result. □

With these two types of interval, we distinguish four types of processors: *join*, *non-join replication* and *non-replication* processors.

[17] 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).

Definition 20 A join processor is referred to as a processor that has to participate in the join phase (of our parallel processing strategy), i.e., the processor which has a partition interval that intersects with the join-intervals. Otherwise, it is referred to as a non-join processor which necessarily returns a null response. □

Definition 21 A replication processor is referred to as a processor that has to has to participate in the replication phase (of our parallel processing strategy), i.e., the processor which has a partition interval that intersects with the replication-intervals. Otherwise, it is referred to as a non-replication processor. □

To develop the mechanism which determine join-intervals and replication-intervals, we need to clarify two issues:

1. The relationship between the comparison predicates involving timestamps and the temporal join predicates.
2. The relationship among:
   - the TS and TE range-partitioning functions,
   - the join-interval and replication-intervals, and
   - the property of redundant state information.

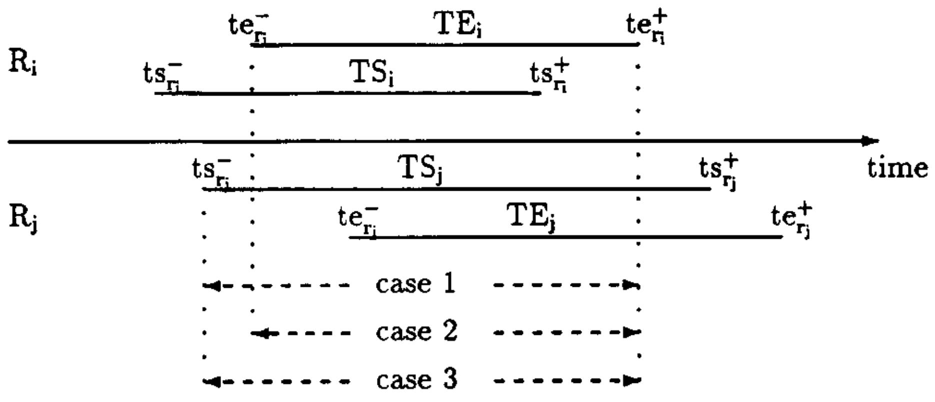
The first issue has been addressed in an earlier section; the upper and lower bounds on the TS and TE values of each individual relation can be determined by propagating constraints between relations using the constraint graph. Below we address the second issue.

Definition 22 Given two intervals $[ts_1, te_1)$ and $[ts_2, te_2)$, we define:

$$\text{I-intersect}([ts_1, te_1),[ts_2, te_2)) \equiv [\max(ts_1, ts_2),\min(te_1, te_2))$$

That is, the time interval that the I-intersect operator produces is the intersection of the two operand intervals. The left-end value of the produced interval ought to be smaller than the right-end value; otherwise the operator produces a null interval. □

Having determined the upper and lower bounds of the timestamp values, we can now study the impact of TS and TE range-partitioning schemes on join-intervals and replication-intervals. For a TSJ_1 join query Q with relations R_i and R_j as shown in Figure 13, there are three cases to be considered.



Constraints:

$ts^-_{r_i} \le TS_i < ts^+_{r_i}$ $\quad$ $ts^-_{r_j} \le TS_j < ts^+_{r_j}$

$te^-_{r_i} \le TE_i < te^+_{r_i}$ $\quad$ $te^-_{r_j} \le TE_j < te^+_{r_j}$

$TS_i < TE_i$ and $TS_j < TE_j$ for all tuples

Figure 13: Determining the join-interval and the replication-intervals

Case 1: R_i and R_j are TS range-partitioned

The join-interval is:

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

In Figure 13, the join-interval is $[ts^-_{r_j}, te^+_{r_i})$. If the join-interval is null, the join response is also null. The replication-interval of relation R_k, where k is either i or j, is:

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

Tuples that start during the replication-interval are replicated on join processors for the join phase. If the replication-interval of a relation is a null interval, no tuples will be replicated as state information.

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

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

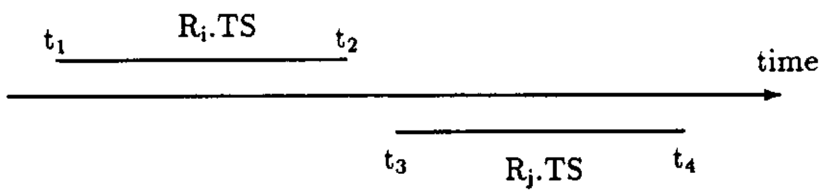


Figure 14: Constraints on TS timestamps of relations R_i and R_j: $t_2<t_3$

The replication-interval of R_i is a null interval while that of R_j is:

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

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

$$\text{I-intersect}(\ [ts^-_{r_i}, ts^+_{r_i}),\ [ts^-_{r_j}, ts^+_{r_j})\ )$$

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

Example 10: Suppose both relations R_i and R_j are TS range-partitioned. Consider the following query:

$$\sigma_{R_i.TS \textbf{ between } [t_1,t_2) \wedge R_j.TS \textbf{ between } [t_3,t_4) \wedge \text{intersect-join}(R_i,R_j)}(R_i,R_j).$$

The state predicate for relation R_i is "$R_i.TS$ **between** $[t_1,t_2)$ $\wedge$ $R_i.TE \geq \mathsf{max}(t_1,t_3)$" while that for relation R_j is "$R_j.TS$ **between** $[t_3,t_4)$ $\wedge$ $R_j.TE \geq \mathsf{max}(t_1,t_3)$". The join-interval is $[\mathsf{max}(t_1,t_3),\mathsf{max}(t_2,t_4))$. Suppose that t_2 is smaller than t_3 as shown in Figure 14, the join-interval becomes $[t_3,t_4)$. The replication interval of R_i is $[t_1,t_2)$ and that of R_j is a null interval (i.e. the state predicate for R_j becomes "false"). In other words, a tuple $r_i \in R_i$ will be replicated to the join processors as state information if "$r_i.TS$ **between** $[t_1,t_2)$ $\wedge$ $r_i.TE \geq t_3$" holds, and the join phase involves only joining the state information of R_i and the local fragments of R_j that start during $[t_3,t_4)$ [18]. □

<u>Case 2: R_i and R_j are TE range-partitioned</u>

[18] One can obtain tighter bounds by examining (and thus accessing) local fragment of R_j that start during $[t_3,t_4)$. In a later section, we will discuss this alternative in more detail.

When both relations are TE range-partitioned, the join-interval and replication-intervals are simply "mirror-images" of those for the above TS range-partitioning case. To simplify our presentation, we consider only the situation when both relations do not have the property of redundant state information. That is, the join-interval is:

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

In Figure 13, the join-interval is $[te^-_{r_i}, te^+_{r_i})$. The replication-interval of R_k, where k is either i or j, is:

$$\text{I-intersect}(\ [\max(te^-_{r_i}, te^-_{r_j}), \max(te^+_{r_i}, te^+_{r_j})), [te^-_{r_k}, te^+_{r_k})\ ).$$

Case 3: R_i is TS range-partitioned and R_j is TE range-partitioned

The join-interval is:

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

In Figure 13, the join-interval is $[ts^-_{r_i}, te^+_{r_j})$. The replication-interval of R_i is:

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

while that of R_j is:

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

If only relation R_i has the property of redundant state information, the join-interval becomes:

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

The replication-interval of R_i is a null interval while that of R_j is:

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

Conversely, if only relation R_j has the property of redundant state information, the join-interval becomes:

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

The replication-interval of R_j is a null interval while that of R_i is:

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

In summary, the above syntactic mechanism can be used to determine which processors have to send data as state information and which processors have to receive data as state information. Doing so allows us to eliminate redundant work. One may wonder if we can move tuples from join-interval to replication-interval for the join phase. In general, all tuples in the join-interval would have to be replicated to all partitions in the replication-intervals which is very expensive and should be avoided. In the following subsection, we discuss the overhead of constructing the state information.

4.6 Quantitative Analysis

We present a first-cut quantitative analysis on the overhead associated with constructing state information of a relation. We measure the overhead 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). Earlier we denoted the rate of insertion of tuples into a relation as λ and the average tuple lifespan as $\overline{T_{ls}}$. We also denoted the relation lifespan as TR_{ls}. Using Little's result [Lit61], the average number of active tuples of a relation at a particular time is:

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

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

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

Suppose that the selectivity of the state predicate q that is used to construct the state information of 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 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 partition boundaries at which state information has to be constructed (i.e. the partition intervals that overlap with the join-interval that we discussed in the previous section). Note that "$1 \leq n_p < n_{pi}$" holds where n_{pi} is the total number of partition intervals. We define the overhead as the ratio of the number of tuples to be copied over the total number of tuples in the relation:

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

This 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.

Note that the above analysis does not depend on whether the time-varying attribute is continuous or non-continuous. In case of continuous time-varying attribute, the average number of active tuples at a particular time is:

$$\overline{n} \;=\; \lambda \cdot \overline{T_{ls}} \;=\; \overline{T_{ls}} \cdot (\overline{Ns}/\overline{T_{ls}}) \;=\; \overline{Ns}$$

where $\overline{Ns}$ is the average number of surrogates in the relation.

5 Implementation Issues

In this section, we discuss several implementation issues that are pertinent to the TS and TE range-partitioning schemes, issues involving storing state information statically and late updates. We also discuss the application of these range-partitioning schemes to situations where the access patterns to current and history data are different, and when temporal data are modeled as non first normal form relations. Finally, we discuss the extension of the notion of state information for aggregate functions.

5.1 Continuously Expanding Time Axis

Although we regard time points as integers, a major difference between the time domain and an integer domain (such as department number) is that the time dimension is continuously expanding. Moreover, it is advancing in one direction, i.e., the current time is getting larger. This leads to some design issues as described below.

For the range-partition function presented earlier, the processor $p_{n_{pi}}$ keeps tuples of the last partition interval $[t_{n_{pi}}, \mathit{now})$. Assume that, for the moment, tuples are evenly distributed among all processors. As time evolves, more tuples with start time greater than $t_{n_{pi}}$ are inserted at processor $p_{n_{pi}}$. Eventually, the workload at processor $p_{n_{pi}}$ will be much higher than other processors. The inherent problem is due to the fact that time is continuously advancing and we have a *fixed* number of partition intervals. We now briefly discuss several solutions:

Variable # of partitions This approach basically splits the last partition interval into one or more intervals. That is, the total number of partitions increases as time advances.

Dynamic splitting This approach is based on balancing the number of tuples kept at each processor. Suppose that each processor should keep about the same number of tuples. When the number of tuples in $p_{n_{pi}}$ exceeds this threshold, the last partition interval $[t_{n_{pi}}, \mathit{now})$ is then split into two regions — $[t_{n_{pi}}, t_{n_{pi}+1})$ and $[t_{n_{pi}+1}, \mathit{now})$. The latter interval is assigned to any processor, denoted as $p_{n_{pi}+1}$; a simple approach, called "round-robin" [DeW90], is to choose processor p_j where j equals $n_{pi}+1$ modulo n. Tuples which belong to the partition $[t_{n_{pi}+1}, \mathit{now})$ are moved from $p_{n_{pi}}$ to $p_{n_{pi}+1}$.

Static splitting This approach is based on static partition intervals. The idea is to choose the partition intervals a priori such that the last partition boundary $t_{n_{pi}}$ is large enough that no update time would exceed this value. As time evolves to a certain point, we split the last interval $[t_{n_{pi}}, \mathit{now})$ into one or more intervals — $[t_{n_{pi}}, t_{n_{pi}+1})$, $[t_{n_{pi}+1}, t_{n_{pi}+2})$, $\cdots$, $[t_{n_{pi}+j}, \mathit{now})$. These new intervals are then assigned to some processors. This kind of static splitting has the drawback that tuples may not be evenly distributed among processors, but re-organization (as in dynamic splitting) seldom takes place, if it ever does.

Recall that the overhead of constructing state information of a relation at all partition boundaries is:

$$\mathrm{Ov} = \sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}} / TR_{ls}$$

where σ_q is the selectivity of the state predicate q, n_{pi} is the total number of partition intervals, $\overline{T_{ls}}$ is the average tuple lifespan, and TR_{ls} is the relation lifespan. The rate of change of the overhead is given by:

$$\frac{d\,\mathrm{Ov}}{dt} = \sigma_q \cdot \overline{T_{ls}} \cdot \frac{TR_{ls} \cdot \frac{d\,n_{pi}}{dt} - (n_{pi}-1) \cdot \frac{d\,TR_{ls}}{dt}}{TR_{ls}^2}$$

which remains constant if the numerator equals 0. That is,

$$TR_{ls} \cdot \frac{dn_{pi}}{dt} - (n_{pi} - 1) \cdot \frac{dTR_{ls}}{dt} = 0$$

Note that the relation lifespan is continuously advancing and the rate of change of the relation lifespan (i.e. $\frac{dTR_{ls}}{dt}$) is 1. To obtain an intuitive interpretation, we let the current relation lifespan be:

$$TR_{ls} = (n_{pi} - 1) \cdot T_{ls}$$

If we create a new partition boundary (i.e. $\frac{dn_{pi}}{dt}$) at a rate of $1/\overline{T_{ls}}$ (i.e. the length of partition interval is $\overline{T_{ls}}$), the overhead (Ov) will remain the same. On the other hand, the overhead increases if we create partition boundaries faster. Again, this is consistent with our intuition that for shorter partition intervals, more qualified tuples may span multiple partition intervals and therefore the overhead increases. As a rule of thumb, the length of partition interval should be greater than the average tuple lifespan. Given the average length of partition intervals (denoted as $\overline{T_{pt}}$), the average number of tuples in a partition interval is:

$$\lambda \cdot \overline{T_{pt}}$$

Fixed # of partitions This approach is to dynamically re-adjust the partition function to maintain a fixed number of partition intervals. The idea is to split the last partition $[t_{n_{pi}}, now)$ into $[t_{n_{pi}}, t_{n_{pi}+1})$ and $[t_{n_{pi}+1}, now)$, choose a particular processor p and relocate its tuples to its neighbor(s), and the partition $[t_{n_{pi}+1}, now)$ is assigned to processor p. We discuss two alternatives to relocate tuples:

2-processors Choose two processors, p_i and p_{i+1}, whose partition intervals are $[t_i, t_{i+1})$ and $[t_{i+1}, t_{i+2})$ respectively, and move tuples that belong to the partition $[t_i, t_{i+1})$ from p_i to p_{i+1}. Processor p_i is responsible for the partition interval $[t_{n_{pi}+1}, now)$ while p_{i+1} is responsible for the merged partition $[t_i, t_{i+2})$. Note that the number of tuples in p_{i+1} is roughly twice as many as it was before the re-adjustment.

3-processors Choose processors p_i, p_{i+1} and p_{i+2} whose partition intervals are $[t_i, t_{i+1})$, $[t_{i+1}, t_{i+2})$ and $[t_{i+2}, t_{i+3})$ respectively, and move half of the tuples of the partition $[t_{i+1}, t_{i+2})$ to p_i and the rest to p_{i+2}. That is, the partition intervals for p_i and p_{i+2} are $[t_i, t_{i'})$ and $[t_{i'}, t_{i+3})$ respectively, where $t_{i'}$ is a time point between t_{i+1} and t_{i+2}. Processor p_{i+1} is responsible for the new partition $[t_{i+1}, now)$. Note that the numbers of tuples in p_i and p_{i+2} increase roughly by half.

There are several tradeoff factors. First, dynamic re-adjustment requires moving tuples between processors and therefore the frequency of re-adjustment and the associated overhead should be kept minimal. For example, in the 2-processors scheme, re-adjustment probably occurs less frequently as the fragment size doubles after the re-adjustment. Second, tuples should be more evenly distributed after the re-adjustment. For example, the 3-processors scheme may produce a more even distribution of tuples as the fragment is split into two halfs.

In spite of potentially expensive re-adjustment overhead, there are some advantages. Firstly, the number of partition intervals is fixed. Consequently, less state information will have to be constructed during the processing of complex temporal queries. Moreover, in the other two schemes, the number of partition intervals becomes larger as time evolves, and eventually some kind of re-organization is needed to keep these partition intervals manageable (and thus accessing the range-partition function is efficient) [DeW90]. The dynamic re-adjustment scheme avoids this situation.

5.2 Storing State Information Statically

As opposed to dynamically constructing of state information, an alternative is to statically store some state information. The approach is to pick a frequent query whose query qualification is P_x for a relation X [19], and use P_x to create and store the state information at each partition boundary. That is, the static state information of relation X at the partition boundary t_i contains [20]:

$$\{ x \mid x \in X \wedge x.TS < t_i \wedge t_i \le x.TE \wedge P_x(x) \} \text{ if X is TS range-partitioned}$$
$$\{ x \mid x \in X \wedge x.TS < t_{i+1} \wedge t_{i+1} \le x.TE \wedge P_x(x) \} \text{ if X is TE range-partitioned}$$

Similarly, we create and statically store the state information of the relation Y using a predicate P_y. In order to process a user query $Q \equiv \sigma_P(X,Y) \in TSJ_1$, we require that "$P|_x \Rightarrow P_x$" and "$P|_y \Rightarrow P_y$" hold, where $P|_x$ and $P|_y$ are the state predicates derived from P. The implications are required because the static state information has to contain all relevant tuples in order to process Q by each processor independently.

Using the quantitative analysis that we presented earlier, the overhead of statically storing state information of a relation X at every partition boundaries is:

$$\sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}} / TR_{ls}$$

where σ_q is the selectivity of the state predicate P_x, n_{pi} is the total number of partition intervals, $\overline{T_{ls}}$ is the average tuple lifespan and TR_{ls} is the lifespan of relation X.

In general, the more selective the state predicate is, the less space is required for storing state information statically. On the other hand, the class of queries that can be processed

[19] Equivalently we can pick a query that subsumes a set of frequent queries.

[20] Note that if retroactive update is supported, it may be necessary to "refresh" the static state information as the updated tuple may span different partition intervals.

in parallel without moving *additional* data (i.e. constructing state information dynamically) becomes smaller. Below, we will discuss some mechanisms in reducing the storage requirement of state information and their tradeoffs.

Ratio of Partition Boundaries & State Information

The approach can be regarded as reducing the state information along the processor dimension. Specifically, only a subset of partition boundaries are selected for storing state information statically. That is, only a subset of processors will store static state information. For example, one can store state information at every other partition boundary. The tradeoffs include that not necessarily all processors have to participate in processing some queries. For example, suppose that relations are TS range-partitioned. Consider that a simple "as of" query such as finding the data value of a time-varying attribute of an object as of a particular time, e.g., $\sigma_{S=10}$ (X) as of t_i. In general, at least processors from p_1 to p_i would have to participate in processing the query. With state information stored at every other partition boundary, at most two processors (p_{i-1} and p_i) are required to perform the search.

Suppose that we statically store state information at one out of g partition boundaries. The overhead of storing state information statically in this approach is:

$$\sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}}/(g \cdot TR_{ls})$$

Interestingly enough, one can view this approach from a different angle. In the original scheme, each processor stores a relation fragment and some state information. In the above approach, a portion of the relation fragment is statically moved and stored at another processor. From this point of view, the query processing workload is spread among the processors.

Partition Boundaries

Another alternative is to select partition boundaries such that fewer tuples span the boundaries. For example, one can choose a time point t' as the partition boundary where there are many new tuples inserted right after time t'. If we had chosen a time point after t', many tuples (which span t') may have to be included as state information.

One can utilize this idea in a different approach. Consider two processors p_{i-1} and p_i which store partitions $[t_{i-1}, t_i)$ and $[t_i, t_{i+1})$ respectively. Instead of storing qualified tuples that span t_i as state information at p_i, we choose a time point t' where $t_{i-1} < t' \leq t_i$, and store only qualified tuples that span both t' and t_i. To process queries in parallel as before, processor p_i has to read the partition $[t_{i-1}, t_i)$ from p_{i-1} but only the portion (t', t_i). Practically, processor p_{i-1} would have to move qualified tuples to p_i that belong to the partition (t', t_i) and span t_i. In other words, the tradeoff for this scheme is to efficiently access these tuples, e.g. by indexing.

Suppose that the average distance between t' and t_i is $\bar{t}$ for all partition boundaries t_i, and that the tuple lifespan is exponentially distributed with $\overline{T_{ls}}$ as the mean value. The overhead of storing state information statically at all partition boundaries is:

$$P[y>\bar{t}] \cdot \sigma_q \cdot (n_{pi} - 1) \cdot \overline{T_{ls}}/TR_{ls}$$

where y is a random variable representing the tuple lifespan and $P[y>\bar{t}]$ is the probability that a tuple would span both time points t^- and t. The probability $P[y>\bar{t}]$ equals:

$$1 - P[0<y\leq\bar{t}] = 1 - \int_0^{\bar{t}} \frac{1}{\overline{T_{ls}}} \cdot e^{y/\overline{T_{ls}}}\, dy$$
$$= e^{-\bar{t}/\overline{T_{ls}}}$$

5.3 Late Updates

Although we assume that time points are monotonically increasing, update times are not necessarily so [21]. This may lead to a potential problem if tuples are statically replicated as state information. Consider the TS range-partitioning scheme and a current tuple $r<s,v,t_2,\mathit{now}>$ stored at processor p_2. The issue is: since the TE timestamp of tuple r can be set to any time point between $[t_2+1,\mathit{now})$, should tuple r be statically replicated in all processors from p_3 to $p_{n_{pi}}$ in their state information?

There are several ways to tackle this problem. The simpliest solution is to assume that we are dealing with transaction times, i.e., the update times are also monotonically increasing. Ensuring the monotonicity can be readily achieved using a system clock.

Another straightforward solution is this. The tuple r is statically replicated in all other processors as state information. When the TE value of tuple r is updated to a specific value (say at t_j), the duplicates at processors whose partition boundary is greater than t_j are removed from the state information while the duplicates at processors whose partition boundary is smaller than or equal to t_j will be updated accordingly. This approach is optimistic in the sense that if most updates occur at a time greater than the last partition boundary ($t_{n_{pi}}$), not many duplicates will be removed. If we assume that "*now*" is the largest current time point, "*now*" would be greater than the latest partition boundary $t_{n_{pi}}$ and thus no duplicate will be removed.

[21] Interesting enough, the Time Index proposed in [Elm90] did not address this issue. It appears that the authors implicitly assumed that update times are monotonically increasing.

An alternative is that all current tuples (i.e. TE value is *now*) are not replicated; only history tuples (i.e. TE value is not *now*) are replicated as state information. In the next section, we further discuss this alternative, keeping in mind that current tuples are more frequently accessed via surrogate or attribute values instead of the timestamps.

5.4 Current Tuples and History Tuples

In the previous sections, we focused on processing of complex temporal pattern queries and snapshot queries. Although one should not ignore these queries, the most frequently accessed tuples may be predominantly the current tuples especially in conventional business-oriented types of applications. This type of access pattern may appear often for several reasons. First, many users may be interested in only current tuples and might view the temporal database as if it were a "static" current database. Second, updating the value of a time-varying attribute will modify the current tuple by setting its TE value ("*now*") to a specific time point. Thus, accessing current tuples occurs more frequently. One of the characteristics of the access pattern in business-oriented applications is that current tuples are often accessed via a given surrogate (or key) value or a time-varying attribute value, e.g., accessing department records by name or department number. This suggests that a different fragmentation strategy for current tuples (instead of range-partitioning on timestamps) may provide more efficient accesses.

Hybrid Fragmentation Scheme

To support efficient access to current tuples based on non-time attributes and to facilitate temporal query processing at the same time, we propose the following hybrid scheme:

- Current tuples are distributed among the processors using any fragmentation method. For the sake of explanation, we use hashing in our discussion.
- History tuples are range-partitioned on a timestamp attribute (TS or TE) as proposed before.

That is, we partition temporal relations into two *logical* disjoint fragments: "current" and "history" fragments. The idea of this hybrid scheme is rather simple: as the temporal database is being updated, history tuples are "migrated" to a particular processor based on their timestamp values.

It should be emphasized that the two logical fragments (current and history fragments) can be stored in different file structures, although we might want to treat them as a single logical entity for discussion purposes. Such separation enables us to access the current fragment more efficiently. For example, one can create an index on the current fragment as in conventional databases. Moreover, one can also avoid storing the special markers "*now*" in current tuples as they are really redundant in the current fragment. Nonetheless, for a temporal relation R that is TS range-partitioned, processor p_i logically keeps tuples specified as:

$$\begin{aligned} R_i = \{\, r \mid r \in R \wedge (\, (\, r.TE = \mathit{now} \wedge h(r) = i \,) \vee \\ (\, r.TE \neq \mathit{now} \wedge r.TS \textbf{ between } [t_i, t_{i+1}) \,) \,) \,\} \end{aligned}$$

where h(r) is a hash function to be applied on attribute(s) of R such as the surrogate.

Note that since temporal relations are logically partitioned into current and history fragments, the "append-only" update model that we presented at the beginning of this paper has to be slightly adjusted to incorporate the fact that updating a current tuple will generally migrate the corresponding history tuple between processors.

State Information & Temporal Query Processing

The parallel processing strategies that we presented earlier basically remains unchanged in the hybrid scheme. The state information is dynamically constructed using both current and history fragments. Since the current fragment is neither TS nor TE range-partitioned, the current tuples are re-distributed differently; the state information at the partition boundary t_i contains the following current tuples:

$$\{\, x \mid x \in X \wedge t_i \leq x.TE \wedge x.TS \leq t_{i+1} \wedge P_x(x) \,\}$$

As in the case of TS or TE range-partitioning schemes, history tuples are also replicated as state information accordingly. After the replication phase, each processor can individually work its local history fragments and the state information.

For state information to be stored statically, we adopt a somewhat different strategy in that only qualified history tuples (i.e. no current tuples) are replicated and stored. For example, for the relation R that is TS range-partitioned, processor p_i statically keeps tuples specified as:

$$\begin{aligned} R_i = \{\, r \mid r \in R \wedge (\, (\, r.TE = \mathit{now} \wedge h(r) = i \,) \vee \\ (\, r.TE \neq \mathit{now} \wedge (\, (\, r.TS \textbf{ between } [t_i, t_{i+1}) \,) \vee \\ (\, r.TS < t_i \wedge t_i \leq r.TE \wedge P_r(r) \,) \,) \,) \,\} \end{aligned}$$

where P_r is the predicate for constructing the static state information.

A point to note regarding the asymmetry properties that we discussed earlier. Recall that one can avoid constructing the state information of a relation R_k if R_k is TS (and TE respectively) range-partitioned and R_k has the asymmetry property with respect to the TS (and TE respectively) timestamp. For the hybrid fragmentation scheme, these conditions do not hold because the current fragment is no longer partitioned based on the TS or TE timestamp. However, if only the history fragment is involved in the user query, one can still eliminate the construction phase of an operand that has the asymmetry property. The condition in which only history fragment of a relation R_k is involved is given by:

$$\mathrm{P} \Rightarrow \mathrm{R_k.TE} \neq \mathit{now}$$

where P is the user query qualification. If we assume that *now* is the largest current time point, the above condition is equivalent to "$\mathrm{P} \Rightarrow \mathrm{R_k.TE} < \mathit{now}$".

5.5 ¬1NF temporal relations

There are several common ways to model temporal data. The data model that we presented earlier uses a pair of timestamps to represent the tuple lifespan and thus can be regarded as *tuple-versioning* [Ahn86] or *first temporal normal form* (1TNF) [Seg88]. Another approach is *attribute versioning* [Ahn86], including the so-called *nested* or *non-first normal form* (¬1NF) temporal data models [Gad88, Tan89]. In the attribute versioning approach, a temporal relation may have more than one time-varying attribute; each attribute value is tagged with a pair of timestamps representing its lifespan. Consider a relation with three time-varying attributes A, B and C:

$$\mathrm{R(S,\ A, TS_A, TE_A,\ B, TS_B, TE_B,\ C, TS_C, TE_C)}$$

Typically, a tuple has a surrogate value and several timestamped data values for each attribute. For this reason, the tuple is in ¬1NF. For example, Figure 15 shows the attribute values of a tuple (s_1) over time. The corresponding ¬1NF tuple is shown in Figure 16 and its normalized version is shown in Figure 17. For ¬1NF relations, the database update model remains essentially unchanged — the difference is that the "append-only" policy is applied on the attribute level and thus no additional ¬1NF tuple is inserted when an attribute value is updated. In other words, the size of a ¬1NF tuple grows as more history attribute values are appended to it. It should be noted that a temporal relation in ¬1NF (e.g. Figure 16) is conceptually equivalent to a materialized join on surrogate value using the corresponding normalized relations (e.g. Figure 17), although a different file structure may be used for storing ¬1NF tuples.

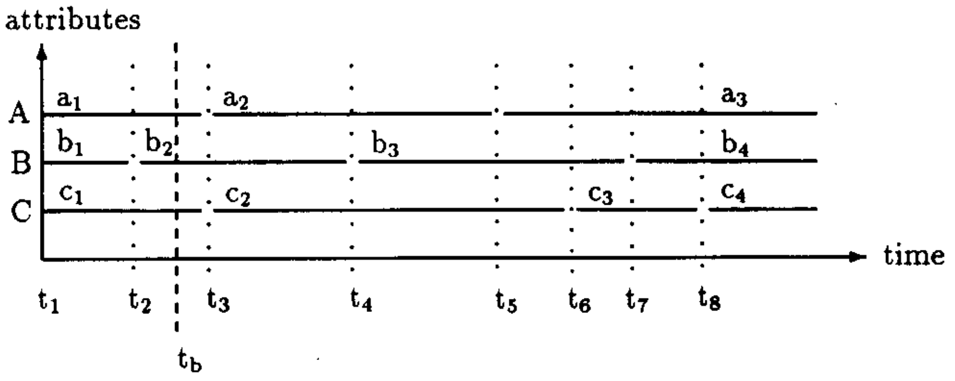


Figure 15: A temporal tuple (s_1) with multiple time-varying attributes

S	A $[TS_A, TE_A)$	B $[TS_B, TE_B)$	C $[TS_C, TE_C)$
s_1	a_1 $[t_1,t_3)$ a_2 $[t_3,t_5)$ a_3 $[t_5,$*now*$)$	b_1 $[t_1,t_2)$ b_2 $[t_2,t_4)$ b_3 $[t_4,t_7)$ b_4 $[t_7,$*now*$)$	c_1 $[t_1,t_3)$ c_2 $[t_3,t_6)$ c_3 $[t_6,t_8)$ c_4 $[t_8,$*now*$)$

Figure 16: A ¬1NF tuple for s_1

S	A	TS_A	TE_A
s_1	a_1	t_1	t_3
s_1	a_2	t_3	t_5
s_1	a_3	t_5	*now*

S	B	TS_B	TE_B
s_1	b_1	t_1	t_2
s_1	b_2	t_2	t_4
s_1	b_3	t_4	t_7
s_1	b_4	t_7	*now*

S	C	TS_C	TE_C
s_1	c_1	t_1	t_3
s_1	c_2	t_3	t_6
s_1	c_3	t_6	t_8
s_1	c_4	t_8	*now*

Figure 17: Normalization of tuple s_1

Range-partitioning ¬1NF relations on timestamp attribute is rather straightforward as before. Let us consider the hybrid scheme on TS range-partitioning:

- The current fragment of s_1 is represented by the following tuple, and is stored at a processor determined by a fragmentation method such as hashing:.

s_1	a_3 $[t_5, now)$	b_4 $[t_7, now)$	c_4 $[t_8, now)$

  Note that the start times of current attribute values are different.

- The history fragment is range-partitioned based on the TS timestamp. To explain the idea, we consider a partition boundary t_b as shown in Figure 15. History attribute values which start prior to t_b are stored at one processor, say p_1, while those after t_b are stored at another processor, say p_2. For tuple s_1, processor p_1 stores the following portion:

s_1	a_1 $[t_1,t_3)$	b_1 $[t_1,t_2)$	c_1 $[t_1,t_3)$
		b_2 $[t_2,t_4)$	

  while p_2 stores:

s_1	a_2 $[t_3,t_5)$	b_3 $[t_4,t_7)$	c_2 $[t_3,t_6)$
			c_3 $[t_6,t_8)$

  Note that history attribute value may not have started during the partition interval.

- State information can be constructed and stored statically. Consider the most general case in which the state predicate to construct the static state information is "true". The portion of tuple s_1 stored in processor p_2 becomes:

s_1	a_1 $[t_1,t_3)$	b_2 $[t_2,t_4)$	c_1 $[t_1,t_3)$
	a_2 $[t_3,t_5)$	b_3 $[t_4,t_7)$	c_2 $[t_3,t_6)$
			c_3 $[t_6,t_8)$

- Updating an attribute value will also migrate the history attribute value. For example, suppose the attribute A of tuple s_1 is updated to a_4 at time t_9 which is later than t_8. The history value a_3 is moved to processor p_2 which would then store (without state information):

s_1	a_2 $[t_3,t_5)$	b_3 $[t_4,t_7)$	c_2 $[t_6,t_8)$
	a_3 $[t_5,t_9)$		c_3 $[t_6,t_8)$

5.6 Temporal Aggregate Functions

The notion of state information is not limited to qualified tuples that span partition boundaries, and can be further generalized in the context of stream processing. Recall that the state information of a stream processor at a particular time t' represents a summary of the history of a computation on the portion of data streams that have been read before t'. Generally speaking, it may be very difficult to characterize the state information (and therefore its storage requirement) for an arbitrary computation. However, the notion of state information can be easily defined for aggregate functions.

Suppose that we ask an aggregate query: find the weekly sales volume on the daily sales records [22]. We further suppose that the data is range-partitioned on a yearly basis among processors, e.g. processors p_{i-1} and p_i store the partitions for the years of t_{i-1} and t_i respectively. The state information at processor p_i can be defined as tuples that started during the last week of the year of t_{i-1}. To process the aggregate query in parallel, each processor p_i has to send (at most) tuples in the last week of its local fragment to its "successor" processor. For example, p_i send some tuples of its local fragment (i.e. year of t_i) to p_{i+1} as state information. Note that the number of tuples in the state information for this aggregate query is bounded — at most one week of daily sales records. Also note that the temporal data is not necessarily time-interval tuples.

6 Other Parallel Temporal Join Strategies

The strategies that we presented earlier may not be applicable in some situations, e.g. TSJ_2 queries. In this section, we discuss several alternative parallel strategies that may be suitable for temporal joins.

6.1 TSJ_2 Queries

In the previous sections, we discussed the parallel processing strategies that are suitable for only TSJ_1 queries. Here we explain why these strategies cannot be utilized for a more generally class of join queries — TSJ_2, and suggest some alternatives in processing these queries. Recall that TSJ_2 are join queries whose join conditions do not require all participating tuples to

[22] Several temporal aggregation operators have been proposed and defined in [Sno86, Seg87]. An example taken from [Seg87] is: "Get a series of 7-day moving averages of book sales."

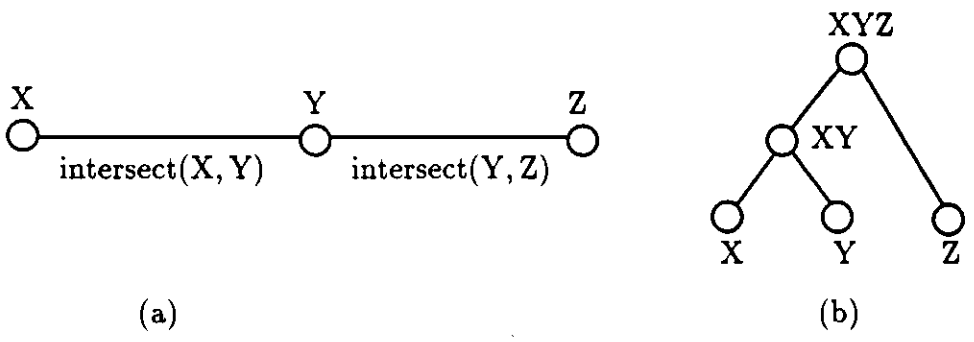


Figure 18: Join graph for query in Example 11

share a common time point (but the participating tuples must overlap).

Example 11: Given temporal relations X, Y and Z. The following query belongs to TSJ_2 (but not TSJ_1):

$$\sigma_{\text{intersect-join}(X,Y) \wedge \text{intersect-join}(Y,Z)} (X,Y,Z)$$

and its join graph is shown in Figure 18(a). □

Consider Example 11. As in the case for TSJ_1, we assume relations are homogeneous range-partitioned on timestamp attributes. Given a tuple triplet <x,y,z> that satisfies the join condition, the tuples x and z may not overlap with each other and may actually reside on different processors. In other words, even if sufficient state information for *all* operand relations have been constructed, the query may not be processed in parallel without moving additional data. The essential difference here is that the state information of temporary tables may have to be constructed dynamically during the join sequence while it is not required for TSJ_1 queries.

Processing TSJ_2 queries can be somewhat more complicated than TSJ_1 queries in general. For example, we can first construct a join sequence for pairwise join execution using the join graph built from the join query qualification. In Example 11, it is "X-Y-Z". Suppose the join sequence is to join relations X and Y first, and then join the result with relation Z, as illustrated in Figure 18(b) [23]:

[23] Obviously, there is a tradeoff in choosing which join to be processed first. For example, joining Y and

Replication Phase I Construct the state information for relations X, Y and Z, denoted as $s_{t_i}(X)$, $s_{t_i}(Y)$ and $s_{t_i}(Z)$ respectively, at every partition boundary t_i, as described before.

X-Y Join Execute the join between X and Y in parallel, the result being a temporary table (denoted as XY) which is also fragmented among processors. The fragment of XY stored at p_i is:

$$XY_i = \sigma_{Pxy}(X_i, s_{t_i}(Y)) \cup \sigma_{Pxy}(Y_i, s_{t_i}(X)) \cup \sigma_{Pxy}(X_i, Y_i)$$

where X_i (and Y_i respectively) is a relation fragment of X (and Y respectively) stored at processor p_i, and Pxy is "intersect-join(X,Y)".

Replication Phase II Construct the state information for relation XY at every partition boundary. That is, A tuple pair <x,y> in XY is copied to processor p_i as state information, denoted as $s_{t_i}(XY)$, if its y component spans t_i.

XY-Z Join The final query response becomes joining the Z fragment with XY fragment and $s_{t_i}(XY)$. That is,

$$Q = \bigcup_{1 \leq i \leq n} \{ \sigma_{Pyz}(XY_i, s_{t_i}(Z)) \cup \sigma_{Pyz}(Z_i, s_{t_i}(XY)) \cup \sigma_{Pyz}(XY_i, Z_i) \}$$

where Z_i is a relation fragment of Z stored at processor p_i, and Pyz is "intersect-join(Y,Z)".

6.2 Sequential/Pipelining

This strategy assumes that relations are homogeneously TS range-partitioned. In conventional databases, "pipelining" often refers to the paradigm in which data "flow" through relational operators such as join and select without being stored in temporary files. The sequential/pipelining strategy here refers to the dataflow among processors and to the fact that state information is sent from one processor to another sequentially. The approach stems from the observation that the processor p_1, which stores tuples in the first partition $[t_1, t_2)$, can immediately proceed to execute the join since we assume that the state information at t_1 (equal to 0) is empty. While the local join in processor p_1 proceeds, qualified tuples that span the next partition boundary t_2 are copied to processor p_2. Processor p_2 can start its execution at any time but it can not finish the execution until it has received all the necessary state information from p_1. This execution mechanism continues for p_2 until $p_{n_{pi}}$ which stores the last partition $[t_{n_{pi}}, \mathit{now})$.

This sequential/pipelining scheme can significantly reduce the interconnection network traffic congestion due to the simultaneous tuple shuffling among all processors. The initial

Z may produce a smaller temporary relation than joining X and Y, and therefore the overall cost may be cheaper if we join Y and Z first.

query response is faster as the first processor can start the execution without delay. However, processing the query takes longer as it has to go through from a processor to another, and the total query response time may be too long that pipelining becomes unacceptable especially when operand relations are range-partitioned into many fragments.

6.3 Semijoin & Join

This method can handle both homogeneous as well as heterogeneous range-partitioning. The semijoin is actually a pre-processing mechanism that can also reduce the number of tuples moved between processors.

Recall that in an earlier section, we discussed alternatives in reducing the amount of state information using four statistics: the maximum and minimum of the TS and TE timestamp values of a relation fragment X_i. The idea is that instead of keeping these four statistics, each processor p_i scans the operand fragment X_i and determine its *exact* fragment lifespan which is obtained by "concatenating" the lifespans of overlapping tuples in X_i. Note that the fragment lifespan may consist of more than one interval [24]. Using the fragment lifespan of X_i in processor p_i, each processor can determine which Y tuples should be moved to p_i for executing the join, for example, sending Y tuples which overlap with the fragment lifespan of X_i. Readers may note that only relation X is required to be range-partitioned on a timestamp attribute; other fragmentation strategies (such as hashing) can be used on relation Y.

One can compare this approach with the alternative that uses the four statistics discussed earlier. The above approach has to pay the overhead in determining fragment lifespans (e.g. scanning fragments once). However, fewer tuples may be sent between processors especially when the fragment lifespans are consisted of several intervals. That is, there are "gaps" in the fragment lifespan that can be used to avoid sending redundant tuples as state information.

7 Previous Work

The parallel processing schemes that we present in this paper is a substantial extension of the work on generalized data stream indexing [Leu92] — the notion of checkpointing the execution state of a query appears in both [Leu92] and this paper. In [Leu92], we proposed an indexing technique based on periodically checkpointing on data streams which are sorted on the TS

[24] If the query qualification has a comparison predicate on relation X (e.g. "X.V=v") and an index on that particular attribute exists, one can use the index to retrieve qualifying tuples and determine the "fragment lifespan".

timestamp. Checkpoints are statically stored and can be indexed on checkpoint times. The major extensions of the work [Leu92] include the following. In this paper, the state predicates are derived from a user query qualification via constraint propagation algorithm and thus the state predicates are more "restrictive". Temporal relations can be range-partitioned based on the TS or TE timestamp; in [Leu92], data streams are temporal relations sorted on the TS timestamp. The class of join query is more expressive in the sense that we can handle equality temporal join predicates (e.g. X.TE=Y.TS) easily. The proof of Theorem 2 can be used to show the data stream indexing scheme in [Leu92] is sound.

[Kar90] is apparently the first publication that appears to support temporal features in multiprocessor database machines. The discussion, however, is quite superficial. First, a number of standard temporal operators such as "when", "at" and "during" are proposed although these operators have been introduced in other literature. More importantly, the paper emphasizes only on translating user queries (with temporal operators) into conventional relational queries, and there is no discussion on query processing and optimization as well as fragmentation strategies. In short, the paper only proposed a front-end syntactic translator for a relational database system regardless whether or not the database system is residing on a multiprocessor database machine.

A partitioned storage for temporal databases was proposed in [Ahn88]. The idea is to split the storage structure into the *history store* and the *current store*, and is similar to the hybrid range-partitioning scheme proposed in this paper. The current store contains current versions and perhaps some history versions of temporal records while the history store contains only history versions. The major issues involved are 1) mapping temporal queries into queries on two storage structures, and 2) update procedures. Their emphasis was on select queries and equi-joins. The idea of archiving a portion of history tuples into optical disks also appears in [Sto87, Kol90].

In [DeW91] a "partitioned band" join algorithm was proposed to evaluate the so-called "band join" which is defined as [25]:

> 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.

The proposed partitioned band join algorithm is to split up relations R and S into partitions R_i and S_i, and compute the band join by joining R_i and S_i for each i. For every tuple *r* in R_i, it is required that all tuples of S that join with *r* appear in the partition S_i. With the

[25] One can think of this type of join as "fuzzy" equi-join. The sort-merge band join (modified sort-merge join) that is proposed in [DeW91] to process "band joins" is essentially a specific instance of stream processing algorithms that we presented in [Leu90].

assumption that the width of a "band" (i.e. c_1+c_2) is small such that the number of tuples that fit in the band will fit in the main memory, the major concern of the join algorithm is to choose the partition sizes (without sorting the relation) such that each of the R_i fits entirely into the buffer pool.

Our proposed parallel join strategies can also be used to process this type of band joins — if the relation R is "converted" into an interval relation using the constants c_1 and c_2. There are a number of differences between [DeW91] and this paper. In this paper, we address a larger class of temporal join queries and snapshot queries. Recall that a complex temporal join may involve several time-interval relations. Moreover, we address the optimization issues related to both TS and TE range-partitioning schemes, as well as query optimization issues. In addition, we do not assume any relationship between the size of buffer pool and relation (or fragment) size as the authors in [DeW91] assume for their band join algorithm.

8 Conclusions & Future Work

In this paper, we discuss parallel query processing strategies for complex temporal join queries and snapshot queries. We show that the strategies are sound for TSJ_1 queries. A number of optimization alternatives have been addressed. We have also discussed several data fragmentation strategies for temporal data.

There are many future research directions. The most challenging ones include the parallel query processing strategies for TSJ_1 queries whose operand relations that are heterogeneously range-partitioned, and for TSJ_2 queries.

References

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

[Ahn88] I. Ahn and R. Snodgrass. Partitioned Storage for Temporal Databases. *Information Systems*, 13(4):369–391, 1988.

[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.

[Elm90] R. Elmasri, G. Wuu, and Y. Kim. The Time Index: An Access Structure for Temporal Data. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 1–12, 1990.

[Gad88] S. Gadia and C. Yeung. A Generalized Model for a Relational Temporal Database. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 390–397, June 1988.

[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.

[Kol90] C. Kolovson. *Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems*. PhD thesis, University of California at Berkeley, November 1990. Memorandum No. UCB/ERL M90/105.

[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 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. Forthcoming.

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

[Nie84] J. Nievergelt, H. Hinterberger, and Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.

[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.

[Seg88] A. Segev and A. Shoshani. The Representation of a Temporal Data Model in the Relational Environment. In *Proc. of the Conf. on Statistical and Scientific Database Management*, pages 39–61, June 1988.

[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.

[Sno86] R. Snodgrass and S. Gomez. Aggregates in the Temporal Query Language TQuel. Technical Report 86-009, University of North Carolina, Chapel Hill, Dept. of Computer Science, March 1986.

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

[Sta88] R. Stam and R. Snodgrass. A Bibliography on Temporal databases. *Database Engineering*, 7(4):231–239, December 1988.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[Sto87] M. Stonebraker and L. Rowe. The POSTGRES Papers. Technical Report UCB/ERL M86/85, University of California at Berkeley, Electronic Research Laboratory, June 1987.

[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.

[Tan89] A.U. Tansel and L. Garnett. Nested Historical Relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 284–294, 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.