# On Spatially Partitioned Temporal Join

*Hongjun Lu* *Beng-Chin Ooi* *Kian-Lee Tan*
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge, Singapore 0511
Internet: {luhj,ooibc,tankl}@iscs.nus.sg

## Abstract

This paper presents an innovative partition-based time join strategy for temporal databases where time is represented by time intervals. The proposed method maps time intervals to points in a two dimensional space and partitions the space into subspaces. Tuples of a temporal relation are clustered into partitions based on the mapping in the space. As a result, when two temporal relations are to be joined over the time attribute, a partition in one relation only needs to be compared with a predetermined set of partitions of the other relation. The mapping scheme and the join algorithms are described. The use of spatial indexing techniques to support direct access to the stored partitions is discussed. The results of a preliminary performance study indicate the efficiency of the proposed method.

## 1 Introduction

Research in temporal databases has largely focused on extensions of existing data models to handle temporal information [EW90, Gad88, GY88, SS87]. An important aspect of these extensions is the introduction of new temporal operators that facilitate the retrieval of factual data corresponding to some past states of the database and/or of the real world. More recently, many researchers have expressed interests in the viable implementation of these temporal operators. To this end, a number of specialized storage structures or indexes have been proposed to provide support for the efficient retrieval of temporal data [EWK90, SOL94]. However, only a few work address the issues of efficient implementation of one of the most expensive operations – temporal join [GS91, SSJ94].

Intuitively, temporal joins are needed in applications which require finding events that happen at the same time. Some examples include queries to find the employees who work in the same department or work on the same project in a company database; to find the nodes being connected (disconnected) during certain period of time in a communication network; to find suspects that appeared at the same location at the same time in a criminal database system; and etc. In order to answer such queries, temporal join is used to find the overlap among the time intervals of different tuples. Unlike joins in traditional relational database systems where equi-joins are the commonest form, joins on time intervals are non-equijoins for which less research work has been reported [DNS91, GS91, SSJ94].

In this paper, we address the issues of efficient processing of temporal joins. In particular, we will concentrate on the partition-based join methods where two source relations are partitioned and the original join is decomposed into joins among the partitions. Although it is a well known fact that partition-based algorithms outperform the nested-loops join and sort-merge join algorithms in most cases, previous results indicated that, for temporal join, the nested-loops and sort-merge methods are in fact reasonably good choices, and the partition-based methods only win in certain cases. Our initial studies also showed similar results. The performance bottleneck of partition-based temporal join method is that the partitions containing so-called "long-lived" tuples (tuples whose time intervals span over a long period) need to be compared with al-



**Proceedings of the 20th VLDB Conference**
**Santiago, Chile, 1994**

most all the other partitions. This defeats the purpose of partitioning, i.e. to limit the comparisons among corresponding partition pairs. As a result, the savings obtained from the reduction of comparisons among partitions may not be able to offset the overhead of partition-based methods – the cost of partitioning.

To solve the problem, we proposed in this paper a partition-based temporal join method with the following unique features:

- Time intervals are mapped to points in a two-dimensional space;
- The tuples are clustered on the time attributes according to their positions in the space;
- Time join is performed using partition-based approach. In other words, a partition of one relation is only compared with certain partitions of the other relation to find matches.

Our performance study indicates that the proposed method outperforms previously proposed methods.

The rest of this paper is organized as follows. Section 2 provides the background information to our study. In Section 3, we look at the basic partition-based time join algorithm and present a new partition scheme. Section 4 discusses how spatial index structures can be used to support the proposed join method. In Section 5, we present the results of a performance study that compares our algorithm with the nested-loops and sort-merge algorithms. Section 6 reviews some related work. Finally, we conclude in Section 7 with discussions on possible extensions to this work.

## 2 Preliminaries

There are a number of models proposed for temporal database. We would like to ignore the modeling issues in this paper and adopt a simple view of the problem specified in this short section so that we can concentrate ourselves on the essential issues related to efficient processing of joins involving time attributes.

Following [GS91, RF93] and others, we consider the time dimension as a sequence of discrete time instants where consecutive time instants differ exactly by one time unit. Time unit is assumed to be the same throughout the time dimension. Attributes of a temporal relation can be non-time varying attributes (such as employee id, name, sex), time-varying attributes (such as salary, qualifications) and time attributes that indicate the time interval that the given values of the time-varying attributes are valid. The *time interval*, denoted $[T_S, T_E]$, $T_E > T_S$, where $T_S$ is the start time and $T_E$ the end time, semantically represents the lifespan of the tuple in question. The *time dimension* is represented as a time interval $[0, T_{now}]$, where 0 represents the starting time of the application and $T_{now}$ refers to the current time which is continuously increasing. Moreover, all relations are assumed to be in first temporal normal form [SS88]. As such, there are no two intersecting time intervals for a given surrogate instance. We say that two tuples, $r$ and $s$, *intersects* if and only if their time intervals overlap, i.e. $r.T_S \leq s.T_E \wedge r.T_E \geq s.T_s$. We also say that an interval $[T_S, T_E]$ *contains* another interval $[t_s, t_e]$ if and only if $T_S \leq t_s \wedge t_e \leq T_E$.

**Example 1.** Consider a database that keeps record of a list of persons and their visits to the United States. A simplified version of **visitor** relation for this database is shown in Table 2. Every person in this database is assigned a unique **pid**. The attribute **entry_pt** stores the entry point used to gain entrance to the United States. Arrival and departure time is normalized against some reference point (day 0) and hence the duration of each person's visit can be represented as a time interval in the time dimension $[0, T_{now}]$. Each tuple is assigned a unique **tid**, that serves as a surrogate, to facilitate references to it.

Table 1: The **visitor** relation.

| tid | pid | entry_pt | $T_S$ | $T_E$ |
|---|---|---|---|---|
| t1 | p1 | NY | 0 | 3 |
| t2 | p2 | SFO | 0 | 5 |
| t3 | p3 | LA | 0 | 7 |
| t4 | p4 | NY | 2 | 3 |
| t5 | p5 | NY | 2 | 11 |
| t6 | p6 | LA | 4 | 8 |
| t7 | p1 | NY | 4 | $T_{now}$ |
| t8 | p2 | LA | 5 | 11 |
| t9 | p7 | SFO | 6 | 8 |
| t10 | p7 | NY | 8 | 9 |
| t11 | p8 | LA | 8 | $T_{now}$ |
| t12 | p6 | LA | 10 | $T_{now}$ |
| t13 | p3 | LA | 11 | $T_{now}$ |
| t14 | p9 | NY | 12 | $T_{now}$ |

A time join, denoted $\bowtie^T$ on two temporal relations $R$ and $S$, consists of the concatenation of all tuples $r \in R$ and $s \in S$ such that the time attribute values in $r$ and $s$ intersect. The start and end times of a resulting record, say $z$, are given as follows:

$$z.T_S = \max(r.T_S, s.T_S) \text{ and } z.T_E = \min(r.T_E, s.T_E)$$

For ease of reference in sequel, we use the term "join" to refer exclusively to the time join, and the term "relation" to mean temporal relation, unless otherwise stated.

## 3 Spatially Partitioned Time Join

### 3.1 Partition-Based Time Join

In relational systems, the partition-based join is one of the three major join methods. Compared to the other two methods, the nested-loops join and sort-merge join, the partition-based join is more efficient as only those "promising" tuple pairs, rather than all tuple pairs, are examined. As hashing is the major partition method, most partition-based join algorithms proposed in relational systems are hash-based. The basic hash partition-based equi-join operation of two relations $R$ and $S$ comprises the following two phases [DKO+84, Sha86]:

- **Partition Phase.** In this phase, a function (usually a hash function) is employed to split relation $R$ into $n$ disjoint partitions $R_1, R_2, \ldots, R_n$ based on the join attribute values such that

  $$\cup_{i=1}^{n} R_i = R \wedge \forall i, j,\ i \neq j,\ R_i \cap R_j = \emptyset$$

  Relation $S$ is also split into partitions $S_1, S_2, \ldots, S_n$ using the same (hash) function.

- **Join Phase.** In this phase, tuples of $R$ are joined with tuples of $S$. Since tuples in $R_i$ are not joinable with tuples in $S_j$ whenever $i \neq j$, $R_1$ only needs to join with $S_1$, $R_2$ with $S_2$, and so on. In other words,

  $$R \bowtie S = \cup_{i=1}^{n} R_i \bowtie S_i$$

  This phase performs the join for each corresponding pair of partitions one at a time using any of the existing join methods.

Performance studies have shown that partition-based techniques generally outperform sort-merge and nested-loops join algorithms for equi-join operation [DKO+84, Sha86]. Unfortunately, to perform temporal join efficiently using partition-based algorithm is not as straightforward. This is because in temporal database, the join operation requires comparison of intervals rather than point data. Unless all the tuples within an interval falls in one partition, or a tuple is duplicated in all the partitions that it overlaps, a partition has to be joined with more than one partition of the other relation. Therefore, the traditional partition-based join algorithms have to be modified to consider this factor.

Partition-based algorithms for temporal join proposed in the literature can be classified into the following two types:

- **Static Partitioning.** In this method, $R$ and $S$ are split into $n$ partitions in a similar manner as the conventional partition-based join methods, except that they are range-partitioned on the start timestamp ($T_S$) of the tuple.[1] However, while each tuple of $R$ appears in only one partition of $R$, a tuple of $S$ will appear in partition $S_i$ if its temporal interval intersects with the interval assigned to $R_i$. In other words, tuples of $S$ may be replicated across several partitions. Therefore,

  $$\sum_{i=1}^{n} \|R_i\| = \|R\| \quad \text{and} \quad \sum_{i=1}^{n} \|S_i\| \geq \|S\|$$

  The advantage of replicating $S$ is that each partition of $R$ needs to be joined with the corresponding partition of $S$ only.

- **Dynamic Partitioning.** In this case, relation $R$ is range-partitioned into $n$ non-overlapping intervals $I_i$, $1 \leq i \leq n$ that covers completely the time line. A tuple (of $R$ and $S$) must appear in the $i^{th}$ partition if its timestamp overlaps the interval $I_i$. To avoid replicating tuples of $R$ and $S$ into multiple partitions, Soo, *et. al.*, keeps each tuple in the last partition that the tuple overlaps [SSJ94]. The join computation is performed backward by processing partition $n$ first, followed by partition $n-1$, and so on. To compute the join results correctly, those tuples whose time intervals intersect more than one partition range will be retained in memory to be combined with tuples in the next partition. For example, a tuple $t_R$ whose time interval intersects partitions $k$ and $k-1$, is stored in partition $k$. After partition $k$ of $R$ has been joined with partition $k$ of $S$, tuples whose time intervals are contained in the interval of partition $k$ are swapped out to prepare for the join of partitions $R_{k-1}$ and $S_{k-1}$. However, $t_R$ will be kept in memory so that it can be joined with tuples in $S_{k-1}$. In other words, the partitions are dynamically adjusted during the join computation.

While the first method is simple, it introduces both storage and processing overhead. For example, in the worst case where all $S$ tuples have time interval $[0, T_{now}]$, each $S$ tuple appears in every partition. The actual tuples processed will be $n \cdot \|S\|$ for some $n \geq 1$, which may be much larger than $\|S\|$. Therefore it requires additional storage space to hold the replicated tuples. More importantly, the additional I/Os required to write and read the replicated tuples result in poor performance. The second method avoids such replication in partitions and it was shown to perform well when the number of so called long-lived tuples (tuples whose time intervals intersect more than one partition), is not large. However, it may perform poorly

[1] The partitioning can also be done using the end timestamp of the tuple.

when the number of long-lived tuples increases. In such a case, the number of partitions also increases drastically. Furthermore, retaining the tuples across partitions during join computation requires quite sophisticated memory management.

### 3.2 Spatial Mapping of Time Intervals

Both the above two methods require determining the partition to which a time interval belongs, which is not an easy task [SSJ94]. Furthermore, both methods need to replicate some tuples, either statically (the first method) or dynamically (the second method). Based on a time-space mapping scheme described in [HN83, SOL94], we propose that tuples in a temporal relation be partitioned using both their start timestamps and time intervals.

Tuples in a temporal relation are viewed as spatial objects in a multi-dimensional space, one or more of which is the time dimension. The following function is introduced to map time intervals to discrete data points in a two-dimensional space:

$$f: I \rightarrow N \times N \qquad \text{where } f([a,b]) = (a, b-a)$$

Applying the function $f$ on tuples of a temporal relation results in a *spatial rendition*. The spatial rendition obtained in this manner have the following nice properties:

- any tuple with a start time $T_S = a$ must be mapped to a point on the line $x = a$;
- any tuple with an end time $T_E = b$ must be mapped to a point on the line $x + y = b$; and
- any tuple with a total time duration of $c$ must be mapped to a point on the line $y = c$.

Figure 1 illustrates the result of applying function $f$ on the tuples of the **visitor** relation. For example, the tuple t4 is mapped to (2,1) since the time interval corresponding to tuple t4 is [2, 3].

Interestingly, the spatial rendition supports time-intersection operation effectively. Given an interval, we can easily determine the set of tuples that intersect it. For example, referring to Figure 1, the set of tuples that intersect t6 (with interval [4, 8]) is given by the shaded region as shown in Figure 2. Clearly, we have a highly visual representation of the tuples that are involved in an intersection operation in the form of a well-delineated search region.

More formally, an interval $[T_S, T_E]$ will intersect all points in the region bounded by the following five lines:

$$x = 0, \quad y = 0, \quad x = T_E, \quad x + y = T_S, \quad x + y = T_{now}$$

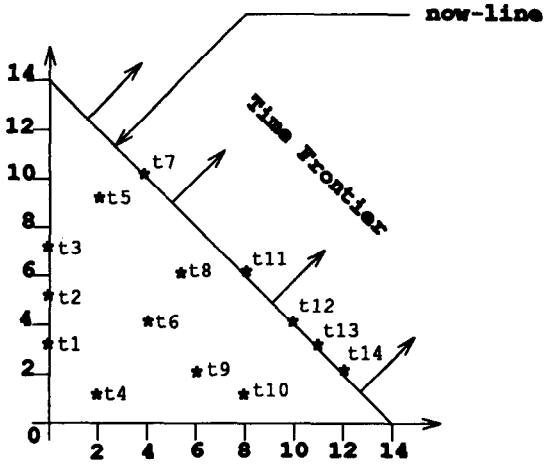



Figure 1: Spatial rendition of the **visitor** relation.

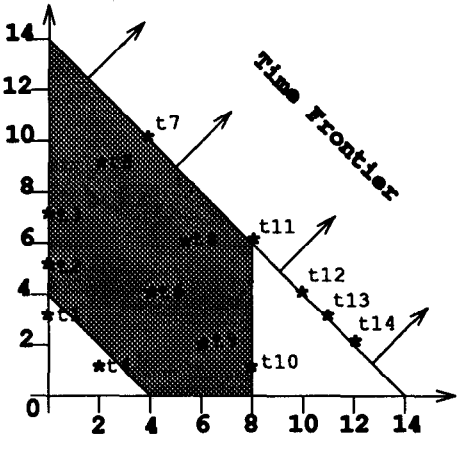
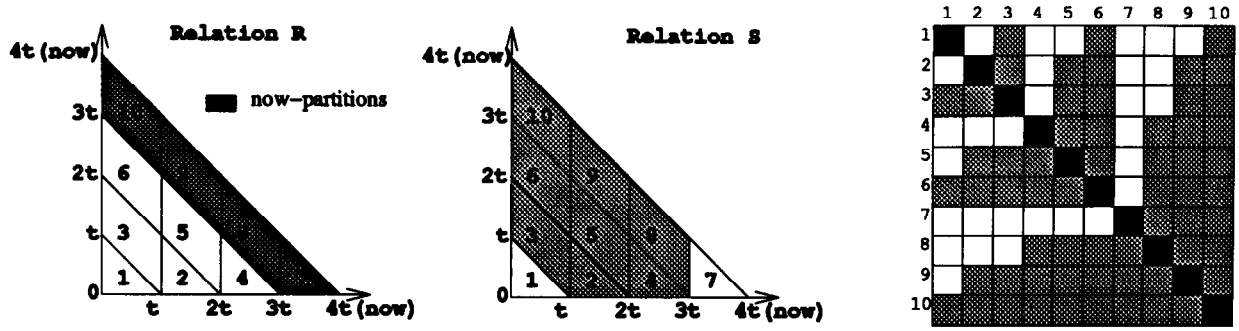



Figure 2: Set of tuples that intersect tuple t6.

### 3.3 Spatial Partitioning of a Temporal Relation

Based on the mapping defined above, we can partition a temporal relation on its time attributes represented by time intervals as illustrated in Figure 3(a):

1. The spatial rendition is split into $n$ *diagonal strips*. The $i^{th}$ strip is bounded by the lines $x = 0$, $y = 0$, $x + y = T_{i-1}$, $x + y = T_i$, where $T_0 = 0$ and $T_n \geq T_{now}$.
2. The strips obtained are split into partitions by the lines $x = 0$, $x = T_1, \ldots, x = T_n$. Thus, each partition is bounded by four lines: $x = T_i$, $x = T_{i+1}$, $x+y = T_j$, $x+y = T_{j+1}$, where $n \geq i, j \geq 0$. Given $n$ strips, there will be a total of $\sum_{i=1}^{n} i = n \cdot (n+1)/2$ partitions. In the Figure, we have $n = 4$ and hence 10 partitions. We distinguish the partitions associated with the *now-line* as the *now-partitions*. For simplicity, we assume that the




(a) Partitions of a spatial rendition. (b) Region to be joined for partition 5. (c) Join Processing of two partitioned renditions.

Figure 3: Partition-based join algorithm.

*partitioning interval* is the same for all partitions, that is $T_i - T_{i-1} = T_{i-1} - T_{i-2}$ for all $i$.

3. Each partition is uniquely identified by an integer value, the *partition_id* as shown in the Figure. Given an interval $[T_S, T_E]$, and a partitioning interval of $t$ units, finding its partition can be done using the following two mapping functions:

$$f: (T_S, T_E) \rightarrow (c, d) \text{ where } \begin{cases} c = \lceil \frac{T_E}{t} \rceil \\ d = \lfloor \frac{T_S}{t} \rfloor \end{cases} \quad (1)$$

and

$$g: (a, b) \rightarrow k \text{ where } k = \frac{a \cdot (a+1)}{2} - b \quad (2)$$

In other words, the *partition_id* of a time interval $[T_S, T_E]$ is given by $g(f(T_S, T_E))$.

### 3.4 Spatially Partitioned Join

We are now ready to present our partition-based join algorithm. Two source relations $R$ and $S$ are partitioned on the join time attributes. Without loss of generality, we assume that the two relations are partitioned using the same interval length $t$. We will discuss the effects later with non-equal interval length in partitioning.

Unlike partition-based natural join in the relational systems where only those corresponding partition pairs $R_i$ and $S_i$ need to be compared to find matching tuples, partition $R_i$ of a temporal relation needs to join with more than one partition of $S$. Let Figure 3(a) and (b) represent the partitions of two relations $R$ and $S$. Partition 5 in $R$, $R_5$, needs to be compared with all the partitions in the shaded region of Figure 3(b), i.e. partitions $S_2, S_3, S_4, S_5, S_6, S_8, S_9$ and $S_{10}$. More general, Figure 3(c) shows the set of partitions of $S$ that must be compared with partitions of $R$, indicated by the shadowed and black squares. For example, $R_1$ has to join with $S_1, S_3, S_6$ and $S_{10}$. In fact, Figure 3(c) explains why the time join is much more expensive than relational equi-join: For relational join, only those partitions in black needs to be compared, the number of which is much fewer than the number of partitions to be compared in the time join case where all shadowed and black partitions need to be joined. If the partition size is equal to the size of available memory, a partition has to be brought into memory a number of times.

Given a partition of $R$ with its unique partition_id $k$, the id's of the $S$ partitions to be joined can be computed by mapping its partition_id $k$, to its two-dimensional counterpart using the following function:

$$h: k \rightarrow (a, b) \text{ where } b = k - \frac{a \cdot (a+1)}{2} \quad (3)$$

where $a$ is the largest integer that satisfies $b > 0$. Given a partition $R_i$ of relation $R$ with its two-dimensional counterpart $(a_R, b_R)$, a partition $S_i$ of $S$ whose partition_id maps to $(a_S, b_S)$ is joinable with $R_i$ if and only if

$$(a_S > b_R) \wedge (a_R - 1 \geq b_S) \quad (4)$$

The join algorithm is summarized in Figure 4. The two major functions in the algorithm are GetPartitionID and ComputeMatchPartition. Function GetPartitionID computes the partition_id for a tuple using Equations 1 and 2. Function ComputeMatchPartition returns a set of partition_id's which is to be joined with a partition of $R$ using Equations 3 and 4.

## 4 Spatial Indexing Structures Supporting Partition-Based Join

In the previous section, we have seen how tuples of a temporal relation can be mapped into points in a two-dimensional time-space, and how a join can be processed for such spatially partitioned relations. In

```
Algorithm BaiscSpatiallyPartitionedJoin
for each tuple r in R do
    k := GetPartitionID(r, t)
    Output (r, k)
endfor
for each tuple s in S do
    k := GetPartitionID(s, t)
    Output (s, k)
endfor

for each partition i in R
    read in partition i
    SetS := ComputeMatchPartition(i)
    for each partition j in SetS do
        read in partition j
        for each tuple pair in i and j do
            if the two intervals intersect
            then output the tuple pair
        endfor
    endfor
endfor
```

Figure 4: The spatially partitioned time join algorithm.

this section, we further analyze how to improve the performance of the partition-based join.

### 4.1 The Bottleneck

Like the conventional partition-based join method, the basic algorithm presented in Figure 4 consists of two phases: the partition phase and the joining phase. During the joining phase, a partition is only joined with a set of predetermined partitions, which reduces the cost of join. Because of the nature of time join, however, the partition-based join over temporal attributes does not perform as well as that over non-time attributes. Refer to the example in Figure 3(c). For nested-loops join, the number of partition pairs to be joined is 100. If the join of $R$ and $S$ is over non-time attributes and partition-based algorithm is used, the number of partition pairs to be joined is 10. If $R$ and $S$ are to be joined over a time attribute, the number of partition pairs to be joined is 70. That is, if we compare the partition-based join with the nested loops join, the saving is only 30%. On the other hand, the partitioning process does incur overhead. In the same example, the partitioning process of one relation needs to read and write the whole relation at least once. A preliminary study was conducted to show the effect of memory size on the nested-loops and the partition-based join algorithms. Figure 5 shows the result of this study. The details of the experimental study will be discussed in Section 5. For the moment, it suffices for us to observe that the partition-based approach, as expected, outperforms nested-loops algorithm only at low memory size, and is inferior to nested-loops algorithm when the memory size is large. Previous work shows similar results [SSJ94].

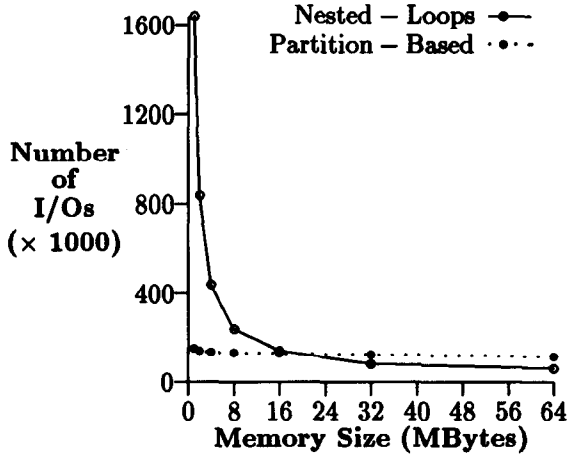



Figure 5: Preliminary study on effects of varying memory (mean lifespan = 1 unit).

Thus, the partition-based algorithm will not be very attractive unless the overhead of partitioning can be reduced.

### 4.2 Clustering Tuples To Avoid Partitioning

One effective method that can be used to avoid partitioning is to cluster the temporal data on the time attributes. In other words, when tuples are inserted, they are clustered on the time attribute in such a way that tuples belonging to the same partition are stored together. Furthermore, a spatial index is built to support direct access to the stored partitions. That is, we adopt the methodology used in spatial databases to treat temporal data. A large number of indexing techniques have been developed to store spatial data [LO93]. Most of these techniques can be used here for indexing spatially partitioned temporal relations with slight modifications.

When two temporal relations are clustered on their time attributes, the partitioning phase can be eliminated. For convenience, we use one recently proposed indexing technique, TP-index [SOL94], to illustrate how a spatial index can be used in the joining process to avoid the partitioning cost. The TP-index, like the $B^+$-tree index [Com79], is a dynamic, multi-level, tree-structured index. The leaf nodes of a TP-index contain pointers pointing to the data pages. Unlike the conventional $B^+$-trees where entries of index nodes contain numeric or alphabetical key values, an entry of

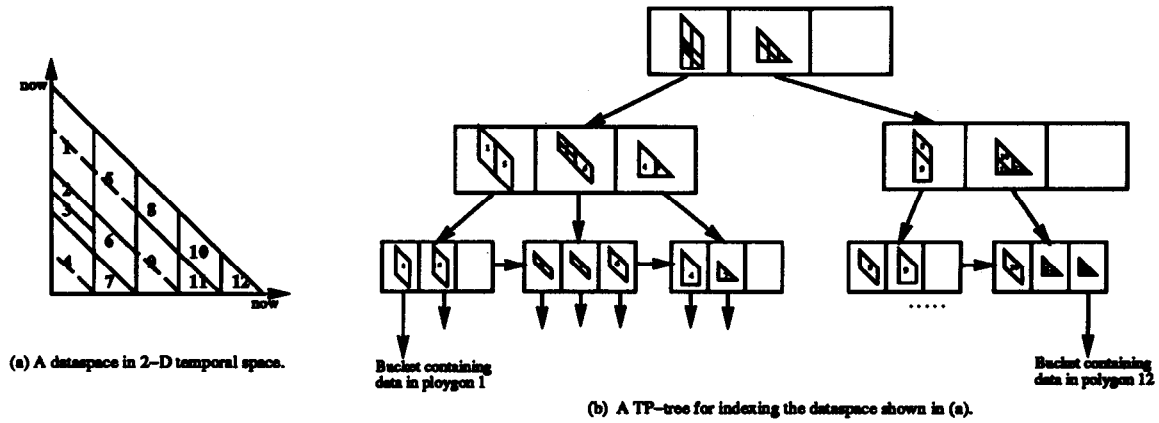



Figure 6: A temporal database in the 2-dimensional space organized by a TP-index.

the TP-index nodes represent the subspace comprised by the data pages in the subtree of which the entry is the root. Figure 6 illustrates a sample TP-index for a relation with 12 data pages, and each index node can accommodate three entries.[2] Each data page stores points in the corresponding areas as shown.

It is obvious that two TP-indexed temporal relations can be joined over the time attributes: data pages of one relation are read into memory in batches. In our example, if the available memory for the outer relation is two pages, pages 1 and 5 are read in as one batch, followed by pages 2 and 3, 6 and 4, and so on. For each batch of pages read in, the pages from the other relation that are joinable with those pages in memory can be computed as described in the previous section. The TP-index of the inner relation is then used to retrieve the desired pages to perform the join.

Because the data points may not be uniformly distributed, the subspace covered by each data page are usually different. In other words, a partition shown in Figure 3 may consists of a number of data pages or one data page may cover a number of partitions. For ease of reference, we call the uniform partitions formed by lines $t, 2t, 3t, \ldots, x+y=t, x+y=2t, \ldots, x+y=nt$ *logical partitions*. Using the notation of logical partition, the join algorithm for two spatially indexed temporal relations $R$ and $S$ can be expressed as in Figure 7.

In the algorithm shown in Figure 7, the outer relation, $R$, is sequentially read in. $S$-pages are retrieved via an index on $S$ with a given search region.

## 5 Performance Study

In this section, we describe the performance study conducted in order to evaluate the proposed technique.

[2]Readers may notice that the partition lines shown here is different from that described in [SOL94]: In order to facilitate efficient processing of the partition-based join, we partition the time space using $y$-line instead of $x$-line. The basic properties, however, remain the same.

```
Algorithm SpatiallyPartitionedJoin
SetS := ∅
while there are unprocessed R pages do
    read in pages of R to fill memory
    SetR := logical partitions covered by R-pages
    for each logical partition i in SetR
        SetS := SetS ∪ ComputeMatchPartition(i)
    for each partition j ∈ SetS do
        read in pages of partition j
        for each S-page s read in do
            compare tuple pairs from s and
                R-pages joinable with j
            if two intervals intersect
            then output the tuple pair
        endfor
    endfor
endwhile
```

Figure 7: The time join algorithm for spatially indexed relations.

The nested-loops join and sort-merge join algorithms [GS91] are also used as references. As in [SSJ94], we do not assume any sort ordering of input tuples. However, both the nested-loops and sort-merge join algorithms were designed to utilize the main memory effectively.

For each algorithm, we simulate its execution on IBM RISC/6000 to obtain the number of random and sequential I/Os needed. Since the result size is the same for all algorithms, we ignore the I/Os for writing the result size to disks. All the algorithms are compared on the basis of the number of I/Os required.

Table 2 shows the parameters used and their default settings. Most of the parameters are self-explanatory. Each relation has 100,000 objects. The range of time is 0 – 100,000 units. The start time of each object follows a Poisson distribution. Each object has an average of 10 versions, the duration of each of which is determined by an exponential distribution, i.e. the lifespan

of each tuple is exponentially distributed. Thus, each relation has 1,000,000 tuples. The partitioning interval at which the relations are to be partitioned is fixed at 1,000 time units, and hence the proposed structure has a total of 5,050 logical buckets. We distinguish between the more expensive random I/Os from the less costly sequential I/Os [SSJ94], and assume that random I/O is 2 times more costly.

Table 2: Parameters used and their default settings.

| Parameter | Default |
|---|---|
| Lifespan of relation | [0, 100,000] |
| Number of objects per relation | 100,000 |
| Avg number of versions per object | 10 |
| Number of tuples per relation | 1,000,000 |
| Memory size | 16 MBytes |
| Page size | 4 KBytes |
| Partitioning interval (in time units) | 1,000 |
| Number of buckets $(\frac{(100+1)\times 100}{2})$ | 5,050 |
| Ratio of cost of random I/O to cost of sequential I/O | 2:1 |
| Avg number of (random) I/Os for index access per bucket access | 4 |

For the proposed partition-based join, to access a bucket, the index must be traversed. Therefore, the cost for the proposed algorithm includes the cost to access the index, and the cost for the join itself. For simplicity, as well as to avoid restricting our discussion to a particular index mechanism, we model the index cost using the *average number of I/Os for index access per bucket access* (denoted $\mathcal{I}$). In other words, for each bucket accessed, the cost incurred for traversing the index is equal to the value as given by $\mathcal{I}$. Note that these I/Os incurred are random I/Os. For example, if 10 buckets are accessed, then the index cost would be $10 \times \mathcal{I}$ random I/Os. When $\mathcal{I}$ is small, such as 0 or 1, it would imply that the index cost is negligible, and a high value of $\mathcal{I}$ would mean that it is expensive to access the buckets through the index.

For the following experiments, we use the following notations for the algorithms studied:

NL: Nested-loops join algorithm
SM: Sort-merge join algorithm
$P_i$: Partition-based join algorithm with $\mathcal{I} = i$.

Varying $\mathcal{I}$ allows us to model a family of partition-based join algorithms with different index mechanisms.

### 5.1 Experiment 1: Effect of Memory Size

Partition-based algorithms are sensitive to memory size [DKO+84, Sha86]. In this section, we study how sensitive the various algorithms are to the amount of memory available. We vary the memory size from 1 MBytes to 64 MBytes. We also vary $\mathcal{I}$ from 0 to 6. As *lifespan* of a tuple will affect the performance of a join to a large extent, two tests were conducted,

1. The lifespan of each tuple is small with the mean value set to 1 time unit. In this case, most of the tuples do not overlap other buckets.

2. The lifespan of each tuple is large with the mean value set to 4,000 time units, which means that on average, each tuple will overlap 4 buckets.

Figure 8 shows the results when the mean lifespan, denoted $\mathcal{L}$, is 1 time unit. We can see that the performance of all algorithms improves as memory increases. When the memory size is small, the nested-loops algorithm must repeatedly scan the inner relation a large number of times since the memory can contain only a few pages of the outer relation. This results in high I/O cost. As memory increases, the number of scans over the inner relation reduces quickly and the total number of I/Os required decreases.

On the other hand, when the mean lifespan is 1 time unit, the sort-merge algorithm is not as sensitive to the memory size. The memory affects mainly the sorting phase of the algorithm. For small memory, more sorted runs are generated, each of which has fewer pages of data. This results in more random I/Os to generate and merge the sorted runs. However, increasing the memory size allows fewer sorted runs to be produced, leading to fewer random I/Os. Since the mean lifespan is 1 time unit and the algorithm is optimized to utilize memory efficiently, there is little backing-up during the merging phase. Thus, the merging phase is virtually unaffected by memory size. However, because of the "fixed" overhead involved in sorting the relations, it is inferior to nested-loops join algorithm for large memory (> 10 MBytes) where the nested-loops algorithm can bring large portion of the outer relation into memory and small number of scans over the inner relation is required. This result coincides with the findings in [GS91, SSJ94].

When the index cost is small ($\mathcal{I} < 4$), the proposed partition-based join algorithm outperforms sort-merge and nested-loops join algorithms at all memory sizes. This is because each partition only joins with a few partitions, and hence the total number of reads of the inner relation is reduced dramatically. Moreover, the low index cost do not contribute significantly to the cost of the algorithm. However, as the index cost increases, the cost of the partition-based algorithm increases, and it becomes inferior to the nested-loops join method at large memory size (> 32 MBytes). This is expected since at large memory, the gain in the joining

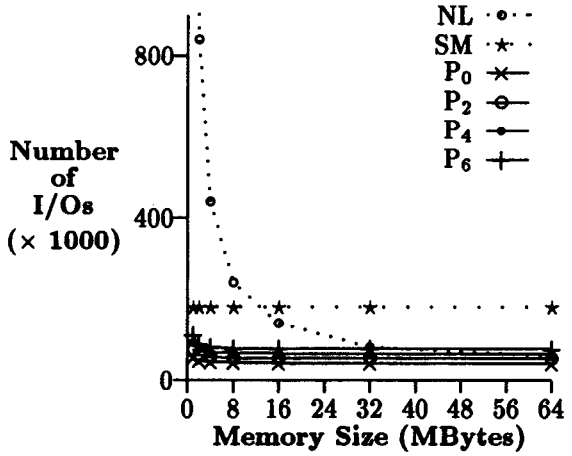



Figure 8: Effects of varying memory ($\mathcal{L} = 1$ unit).

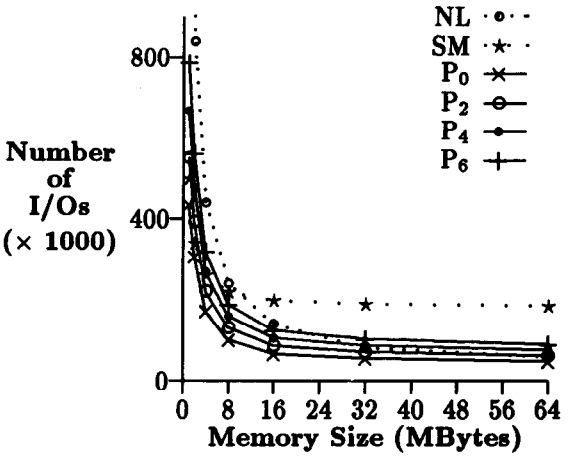



Figure 9: Effects of varying memory ($\mathcal{L} = 4000$ units).

cost of the partition-based algorithm over the nested-loops method is not significant as compared to its overhead in traversing the index.

With the mean lifespan is 4,000 units, the relative performance of the algorithms are similar. The result is shown in Figure 9.

The nested-loops algorithm is unaffected by the mean lifespan of the tuples, as all tuple pairs are compared anyway. However, for the sort-merge algorithm, it incurs a higher cost during the merging phase due to more backing up being performed. In some instances, especially for low memory size, a tuple of relation $R$ overlaps a large number of $S$ tuples. Since the number of matching tuples may not fit in memory, some portion of the relations may have to be re-read, resulting in higher I/O cost.

The curve for the partition-based join algorithm moves up when the mean lifespan is increased to 4,000 time units. The algorithm performs worse than the nested-loops join at a smaller memory size than that with mean lifespan of 1 time unit. In the next experiment, we can see this more clearly and explain the reasons.

### 5.2 Experiment 2: Vary the mean lifespan of tuples

To further study the effects of the mean lifespan of tuples on the algorithms' performance, we vary the mean lifespan from 250 time units to 16,000 time units and measure the number of disk I/O's under the default memory size (16 MBytes). The result is shown in Figure 10.

As before, the nested-loops algorithm is not sensitive to the mean lifespan of the tuples, and is included for comparison purpose. Performance of the sort-merge algorithm becomes worse as the mean lifespan increases. This is expected since more tuples overlap as the mean lifespan increases and leads to re-reading of some pages of the relations. Since the sort-merge algorithm is inferior to nested-loops join at 16 MBytes memory (from Figures 8 and 9), and increasing the mean lifespan increases the cost, it performs worse than the nested-loops join algorithm.

The number of disk I/O's required for the partition-based join algorithm also increases when the mean lifespan increases. This can be explained as follows: The number of $S$ partitions to be joined with each $R$ partition is fixed as shown in Figure 3. For example, when there are 10 partitions, $R_1$ should join with 4 $S$ partitions, $R_2$, $R_6$ and $R_{10}$ need to join with 8, 9, and 10 $S$ partitions, respectively. The general pattern is that, the $R$ partitions in the lower part of the triangle need to join with relatively fewer number of $S$ partitions. The partitions located at the upper corner of the space rendition need to join with larger number of partitions. Partition 10, for example, needs to join with all the $S$ buckets. When we increase the mean lifespan of tuples, more tuples have longer lifespan. Graphically, their mapping points in the space rendition move upwards. In other words, with longer lifespan, the partitions at the upper corner of the space rendition contain more tuples, hence more pages. Therefore, the number of $S$ partitions to be compared in fact increases which results in higher costs.

As observed earlier, as the index cost increases, the cost of the partition-based algorithm increases. In this experiment, we see that the partition-based algorithm is superior over the nested-loops join (and sort-merge) for up to an average of 4 I/Os for index accesses per bucket access.

For most indexes, the fan-out, i.e. the ratio be-

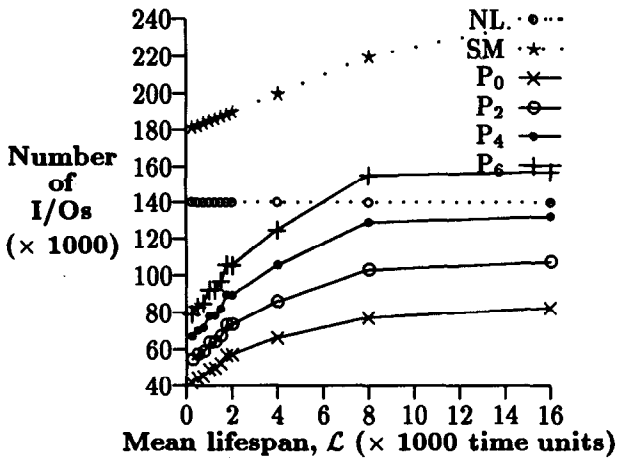



Figure 10: Effects of varying mean lifespan, $\mathcal{L}$.

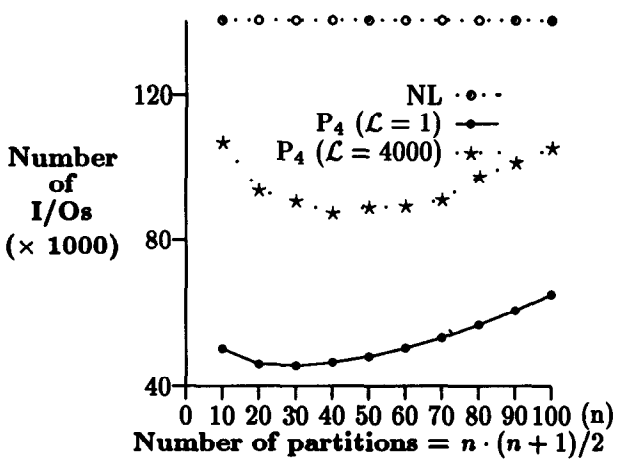



Figure 11: Effects of varying number of partitions.

tween the number of data pages and the number of index pages, is large. This is also true for spatial indexes. Furthermore, most systems will try to keep the first few levels and/or as many index pages as possible in buffer to reduce the cost of direct access to index pages. In the subsequent subsections, we only show the results with $\mathcal{I} = 4$, which represents reasonably large cost of accessing index pages and is unfavorable to the performance of the proposed algorithm.

### 5.3 Experiment 3: Vary number of buckets

Like all partition-based algorithms, the granularity of the bucket affect the performance of the algorithm. Recall that when the partitioning interval is $t$, we have $n = T_{now}/t$ diagonal strips, and there are a total of $n \cdot (n+1)/2$ buckets. With finer partitions, the number of $S$ buckets to be read for each $R$ bucket is controlled more closely. That is, fewer of the $S$ buckets read are wasteful or unnecessary. However, increasing the number of buckets also increases the number of random I/Os (for index access and bucket access), and fragmented pages. In this experiment, we study this tradeoff by varying the number of buckets. Both the relations have the same number of buckets. Since the sort-merge join performs worse than nested-loop join, we will ignore the sort-merge join from this point onwards, and compare the partition join algorithm with the nested-loops algorithm only.

Figure 11 shows the result of the experiment with mean lifespan 1 and 4,000 units respectively. Our first observation from the results is that there is an optimal number of buckets for the partition-based join algorithm. A small number of buckets (10) is not very effective as each bucket contains more pages which lead to unnecessary and redundant reads. As the number of buckets increases (until 30 or 40), the I/O cost is reduced. In other words, the gain from minimizing the wasteful reads outweighs the overhead of additional random I/O cost. However, further increase in the number of buckets results in poor performance again because the gain from minimizing the wasteful reads is not significant as compared to the overhead of additional random I/O cost. Second, performance improvement for the case with larger lifespan is more. This can be explained as follows: When the mean lifespan is small (1 time unit), the number of tuples that overlap several partitions is small and there are not too many unnecessary reads of $S$ buckets, hence the potential of improvement is limited. For large mean lifespan (4,000 time units) with small number of buckets, each partition consists of a large number of buckets. The number of unnecessary comparisons is large which leads to higher cost and to more space for improvement.

### 5.4 Experiment 4: Vary number of partitions of one relation

So far, we have assumed that both relations $R$ and $S$ are partitioned with the same interval (granularity). When two relations are partitioned on different partitioning interval, the proposed partition join algorithm is still applicable. All that is needed is to map the bucket in $R$ to a corresponding region in $S$. Then we can treat the join as if the two relations are partitioned using the same partition interval.

Let the partitioning intervals of $R$ and $S$ be $t_1$ and $t_2$ respectively. Suppose the lines that bound a bucket of $R$ are $x = k_1 \cdot t_1$, $x = (k_1 + 1) \cdot t_1$, $x + y = k_2 \cdot t_1$, $x + y = (k_2 + 1) \cdot t_1$, for some $k_1$ and $k_2$. Then the lines that bound the region of $S$ that contains this bucket

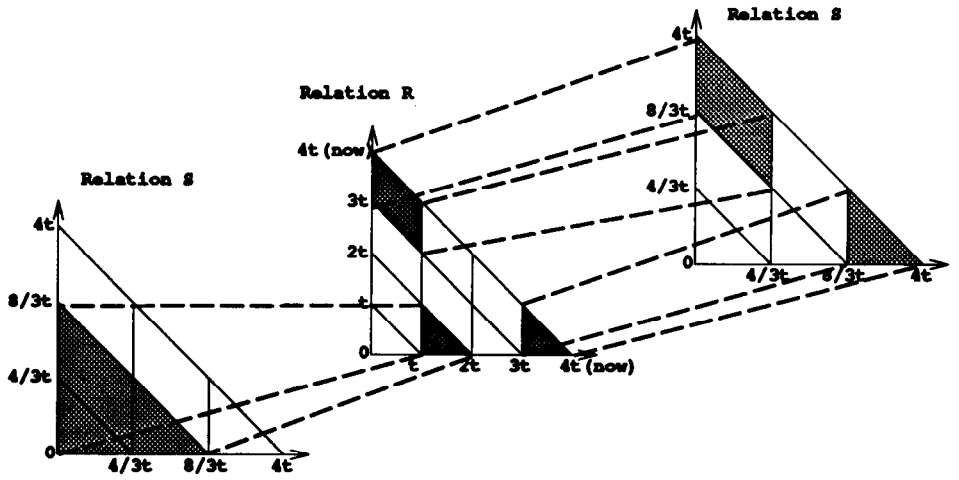



Figure 12: Mapping of partitions between two relations partitioned using different interval

of $R$ are: $x = k_3 \cdot t_2$, $x = k_4 \cdot t_2$, $x + y = k_5 \cdot t_2$, $x + y = k_6 \cdot t_2$, where $k_3$ is the largest integer such that $k_3 \cdot t_2 \leq k_1 \cdot t_1$, $k_4$ is the smallest integer such that $k_4 \cdot t_2 \geq (k_1 + 1) \cdot t_1$, $k_5$ is the largest integer such that $k_5 \cdot t_2 \leq k_2 \cdot t_1$, $k_6$ is the smallest integer such that $k_6 \cdot t_2 \geq (k_2 + 1) \cdot t_1$.

Figure 12 illustrates an example in which relation $R$ is partitioned into 10 buckets while relation $S$ is partitioned into 6 buckets. The join of 3 example partitions of $R$ can be mapped to the region of $S$ as shown.

We conducted an experiment to study the effect of partitioning two source relations using different partition interval. In the experiment, we keep the partition interval of $S$ as 1,000 time units, i.e. 5,050 buckets and vary the partition interval of $R$ so that the total number of buckets in $R$ varies from 55 (i.e. partition interval of 10,000 time units ) to 5,050 (i.e. partition interval of 1,000 time units). The results of this study is shown in Figure 13.

From the result, we see that partitioning two relations using different partition interval does affect the performance. While the two relations are partitioned with the same interval gives the best performance, there are some other choices that give similar performance. This is because the mapping between two partition schemes fits nicely and does not introduce extra comparison of buckets. However, if two partition schemes do not fit each other, in the sense that partitions of $R$ have to be mapped to partitions of $S$ with a lot of overlapping buckets, the performance will degrade.

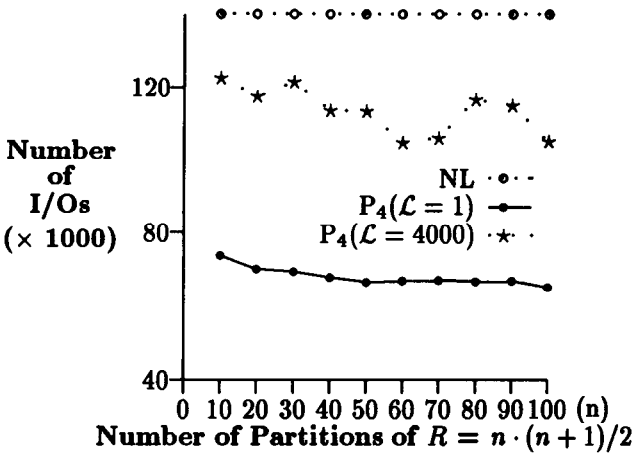



Figure 13: Effects of varying number of buckets of $R$.

## 6 Related Work

Most of the previous work in temporal join evaluation has concentrated on refinements of the basic nested-loops approach [GS91, LM90, RF93]. Gunadhi and Segev proposed three such join algorithms that, unlike the conventional nested-loops algorithm, require only partial scans of the inner and/or outer relations [GS91]. This is achieved by exploiting the sort-order of the input relations to determine if and when the scan of the inner and/or outer relations can be terminated. The algorithms are distinguished by its sort-order (whether the sort ordering is based on $T_S$ only or on $(T_S, T_E)$ pair) and the number of relations sorted (one or both). An analytical study showed that the proposed algorithms performed well when the average scan length through the inner relation is small; otherwise the traditional nested-loops algorithm is superior.

In [RF93], seven nested-loops-like algorithms were proposed. The main feature of these algorithms is that they minimize the number of unnecessary, and hence wasteful, tuple comparisons. This is done through sorting and/or providing additional pointers. Unfortunately, the algorithms assumed that the smaller rela-

tion fit in memory. Moreover, no performance analysis was presented.

Leung and Muntz considered stream processing techniques for processing temporal join [LM90]. They studied the effects of various sort orderings of the streams of input on the workspace requirement. Additional house-keeping must be done or a semi-join algorithm may also be used to preserve the order of an output stream to that of the input stream.

Partitioned-based algorithms have also been studied recently [LM92, SSJ94]. Leung's approach is essentially based on the the static partitioning approach. However, in his work, the algorithm is developed in a multiprocessor setting. Soo, *et. al.*, on the other hand adopted the dynamic partitioning strategy. Both algorithms were discussed in Section 3.

## 7 Conclusions

In this paper, we discussed the issues of efficient processing of temporal join, the join on time intervals. For such joins, the well-known efficient join method in relational systems, the partition-based join, does not perform very well compared to the nested-loops. The major reason is that temporal join is a non-equijoin operation by nature and a partition from one relation must be compared with several partitions from the other relation. The savings in reducing the number of comparisons may not be enough to offset the overhead incurred in the partitioning phase of the partition-based join algorithm.

To overcome the problem, we proposed in this paper a spatially partitioned time join method. Using this method, time intervals are mapped to points in a two-dimensional space. When tuples are inserted into the relation, they are clustered based on their corresponding points in the space. As such, the partition phase can be eliminated when join over the time attributes of two relations is to be performed. Hence, the join performance is much improved.

To provide direct access to the partitions to be joined with, certain spatial index must be used. Limited by space, we have not discussed this issue in detail here. One of our future work is to study existing spatial indexing mechanisms to see how well they can support the proposed partition based join method as well as other primitive operations in temporal databases. We would also like to compare our algorithm with other partition-base algorithms, such as the algorithm proposed in [SSJ94].

## References


[Com79] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979.

[DKO+84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *1984 SIGMOD*, Jun 1984.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *17th VLDB*, pages 443–452, Sept 1991.

[EW90] R. Elmasri and G.T.J. Wuu. A temporal model and query language for temporal databases. In *6th ICDE*, pages 76–83, Apr 1990.

[EWK90] R. Elmasri, G.T.J. Wuu, and Y.J. Kim. The time index: An access structure for temporal data. In *16th VLDB*, pages 1–12, Aug 1990.

[Gad88] S. Gadia. A homogeneous relational model and query language for ER databases. *ACM Transactions on Database Systems*, 13(4):418–448, Dec 1988.

[GS91] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *7th ICDE*, pages 336–344, Apr 1991.

[GY88] S. Gadia and C.S. Yeung. A generalized model for a relational temporal database. In *1988 SIGMOD*, Jun 1988.

[HN83] K. Hinrichs and J. Nievergelt. The grid file: A data structure designed to support proximity queries on spatial objects. In *1983 Workshop on Graphtheoretic Concepts in Computer Science*, pages 100–113, 1983.

[LM90] T.Y.C. Leung and R.R. Muntz. Query processing for temporal databases. In *6th ICDE*, pages 200–208, Apr 1990.

[LM92] T.Y.C. Leung and R.R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *18th VLDB*, pages 383–394, Aug 1992.

[LO93] H. Lu and B.C. Ooi. Spatial indexing: Past and future. *IEEE Data Engineering*, 16(3):16–21, Sept 1993.

[RF93] S.P. Rana and F. Fotouhi. Efficient processing of time-joins in temporal data bases. In *3rd DASFAA*, pages 427–432, Apr 1993.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, Sept 1986.

[SOL94] H. Shen, B.C. Ooi, and H. Lu. The tp-index: A dynamic and efficient indexing mechanism for temporal databases. In *10th ICDE*, pages 274–281, Feb 1994.

[SS87] A. Segev and A. Shoshani. Logical modelling of temporal data. In *1987 SIGMOD*, pages 454–466, May 1987.

[SS88] A. Segev and A. Shoshani. The representation of a temporal data model in the relational environment. *LNCS 339*, pages 39–61, 1988.

[SSJ94] M. Soo, R. Snodgrass, and C. Jenson. Efficient evaluation of the valid-time natural join. In *10th ICDE*, pages 282–292, Feb 1994.