Index Based Processing of Semi-Restrictive Temporal Joins *

Donghui Zhang, Vassilis J. Tsotras†
CSE Department, University of California, Riverside, CA 92521
{*donghui, tsotras*}*@cs.ucr.edu*

Abstract

Temporal joins are important but very costly operations. While a temporal join can involve the whole time (and/or key) domain, we consider the more general case where the join is defined by some time-key rectangle from the whole space (i.e., when the user is interested in joining portions of the –usually large– temporal data). In the most restrictive join, objects (within this rectangle) are joined together based on *key equality* and *interval intersection*. This paper concentrates on *semi-restrictive* joins, i.e., when either the key equality (equi-join) or the interval intersection (time-join) predicates are used. Given the large relations created by the ever increasing time dimension, we assume that each temporal relation is indexed and examine efficient ways to process semi-restrictive temporal joins. Utilizing an index is helpful since it directs the join towards the objects that are within the time-key rectangle. A straightforward approach is to perform an *unsynchronized* join. An index selection query on each relation identifies all objects within the time-key rectangle which are then joined. Although simple, this approach ignores the data distribution in the other relation. Instead, in a *synchronized* join, both indices are concurrently traversed as the join is computed. Synchronized semi-restrictive join algorithms can be performed utilizing traditional indices like B+-trees or R-trees. The drawback of this approach is that traditional indices do not achieve good temporal data clustering. Better clustering is achieved by temporal indices through record copying. Nevertheless, record copies can greatly affect the correctness and effectiveness of join performance. In this paper we introduce correct and efficient algorithms for performing semi-restrictive temporal joins using temporal indices. An extensive experimental comparison shows that the newly proposed algorithms have the best performance. While the paper concentrates on using the Multiversion B+-tree, our algorithms apply equally to other efficient tree-based temporal indices.

1 Introduction

A temporal record has a key, some attributes and a time interval during which the record is valid. Temporal join predicates may involve the key and/or time spaces [GS91]. Examples include the *T-Join* (join two records if their intervals intersect), the *E-Join* (join two records in their keys are equal), and the *TE-Join* (keys are equal and intervals intersect).

Due to large volume of temporal data, a user may be interested in a portion rather than the whole time-key data space. This portion is typically retrievable via a *range-interval selection query*: "find all records with keys in range r and whose intervals intersect interval i". By intergrating the range-interval selection condition into the join predicate, we get the *general* T-Join, E-Join and TE-Join (in

*category: regular paper, **track:** Time Management in Databases, **topics:** temporal database systems, indexing of temporal data.

†This work was partially supported by NSF (115-9907477, EIA-9983445) and the Department of Defense.

short, GT-, GE- and GTE-Joins). Note that the *plain* T-Join, E-Join and TE-Join are then special cases when the query range r is the whole key space and the query interval i is the whole time space.

Temporal join research has focused on non-indexed algorithms for plain joins [GS91, LM93, RF93, SSJ94, RS96, SE96, Zur97] and typically involves a sequential scan of both relations. This is prohibitive for general temporal joins. Instead, an indexing scheme is beneficial since it can quickly direct the join towards the objects of interest. With an index present in each joined relation, a straightforward approach is to perform *unsynchronized* joins. Each index is first used to identify the objects of the relation that is within the query rectangle. The retrieved objects are then joined (using existing algorithms for plain temporal joins). A drawback of this approach is that each selection is performed independently, ignoring the data selectivity of the other relation. Instead, in a *synchronized* join, the selection and the join phases are integrated. Both indices are traversed concurrently, taking advantage of the join selectivities and thus leading to more robust performance. For example, a synchronized traversal algorithm can quickly identify that no pair of records should be reported (if for example no record from the first relation qualifies) thus finishing the join fast.

When considering the index used on each temporal relation, various possibilities exist. One approach is to use traditional indices like B+-trees and R-trees [BKS+90]. Such indices are widely implemented and lead to straightforward synchronized joins. However, such joins suffer from the ineffectiveness of the B+-tree and (to a lesser extent) the R-tree on clustering temporal data. Temporal data is inherently multidimensional having typically long intervals on the time dimension. Even the R-tree (and its most efficient variation, the R*-tree [BKS+90]) are known to be problematic in clustering long intervals [KS91, ST99]. Better clustering is achieved by temporal access methods that create many copies for records with long time intervals [LS89, BGO+96]. This leads to fast processing of range-interval selection queries, but record copies can greatly affect join processing (for example, duplicate join results).

In [ZTS02], we have proposed efficient synchronized join algorithms that utilize temporal indices for the GTE-Join (i.e., when both the key-equality and interval-intersection conditions are applied). As a follow-up to that work, in this paper we address the *semi-restrictive* GT-Join and GE-Join. That is, we look at temporal joins when either the key-equality (GE-join) or the interval-intersection (GT-join) predicates are used. Due to the inherent difference among the join conditions of the three problems, the GTE-Join algorithms proposed in [ZTS02], either do not apply or need to be modified so as to solve the GT- and GE-join problems. In the rest of the paper we use the Multiversion B+-tree (MVBT) [BGO+96] as a temporal index. However, the proposed algorithms can be applied to other efficient temporal access methods [LS89, VV97].

The main contributions of this paper are summarized below:

- We propose synchronized, temporal index based join algorithms for the GT-Join and GE-Join. Top-down and sideways traversal algorithms are presented.
- We examine other approaches, including the unsynchronized approach, the B+-tree and R-tree based synchronized approaches, and an approach based on spatial partitioning (using the TP-Index [SOL94]). Results from an extensive experimental evaluation are also presented.

Our experimental study shows that the sideways, link-based, synchronized traversal is the most robust among the examined methods. Depending on the join characteristics, other methods can be competitive and should be considered by a temporal database optimizer. The rest of this paper is organized as follows. Section 2 formally defines the join problems that we address. Section 3 reviews related work. Section 4 proposes algorithms for the GT-Join, while section 5 addresses the GE-Join.

Section 6 shows the experimental results over all compared approaches while section 7 concludes the paper.

2 Problem Definition

In the following, a key range r is specified by its $r.low$ and $r.high$ keys while a time interval i is described by its $i.start$ and $i.end$ time instants. A range and an interval create a *rectangle* in the 2-dimensional key-time space. A record contains a key, a time interval and various attributes that may change over time. We assume the First Temporal Normal Form [SS88] which implies that no two records exist in a given temporal relation that have equal keys and intersecting intervals. A record with time interval i is called *alive* for all time instants in i. Moreover, we assume the *transaction-time* model [JS99] which implies that record updates arrive in increasing time order. When a data record is inserted in a relation at time t, the end time of its interval is yet unknown and is thus initiated to *now* (a variable representing the ever increasing current time). Record deletions are logical, i.e., records are marked as deleted and are retained in the relation. Hence, if a record is deleted, its end time is changed from *now* to its deletion time. An attribute update to record with key k at time t is treated by a (logical) deletion of the old record at t and the subsequent insertion of a new record –with key k, but updated attributes– and an interval starting at t.

The temporal joins examined in this paper are defined as:

1. *General T-Join (GT-Join)*: given temporal relations X, Y, key ranges r_1, r_2, and time interval i, find all (x, y) where $x \in X$ and $y \in Y$ such that: (1) $x.key \in r_1$ and $x.interval$ intersects i; (2) $y.key \in r_2$ and $x.interval$ intersects i; and (3) $x.interval$ intersects $y.interval$.

2. *General E-Join (GE-Join)*: given temporal relations X, Y, key range r, and time intervals i_1, i_2, find all (x, y) where $x \in X$ and $y \in Y$ such that: (1) $x.key \in r$ and $x.interval$ intersects i_1; (2) $y.key \in r$ and $x.interval$ intersects i_2; and (3) $x.key = y.key$.

Note that in general, the key ranges (respectively, time intervals) restricting each of the two relations in the GT-Join (respectively, GE-Join) can be different. An example GT-Join query is: "*find employees whose last names start with 'B' and who co-worked during 1995 with the employees whose last names start with 'S'*". An example GE-Join query is: "*find the 1998 IBM employees who were UC Riverside students in 1995*".

3 Related Work

In our previous work [ZTS02], we proposed synchronized join algorithms based on temporal indices for the General TE-Join: given temporal relations X, Y, key range r, and time interval i, find all (x, y) where $x \in X$ and $y \in Y$ such that: (1) $x.key \in r$ and $x.interval$ intersects i; (2) $y.key \in r$ and $x.interval$ intersects i; and (3) $x.key = y.key$ and $x.interval$ intersects $y.interval$. Except for [ZTS02], research on temporal joins has focused on non-indexed algorithms. [RF93] assumed that the smaller relation fits in memory and proposed seven nested-loop (plain) T-Join algorithms. [GS91] provided sort-merge (plain) T-Join and TE-Join algorithms when one or both relations are sorted. [LM93] assumed that the relations are sorted on the start time of the record intervals and discussed how to merge them in a *stream-processing* manner. Each iteration of the algorithm reads in buffer one record whose start time is the smallest among non-read records. This record is joined with the in-buffer records and the in-buffer records which will not join with further records are removed. [RS96] also assumed the relations are sorted and discussed how to merge them.

Besides the nested-loop and sort-merge temporal join algorithms, partition-based algorithms have also been proposed. In static partitioning [Zur97], a record is copied to all partitions that intersect its interval. A partition of records in one relation needs to join with one partition of the other relation. In dynamic partitioning [SSJ94], a record is assigned only to one partition (the last partition that intersects the record's interval). After a pair of partitions is joined, the records that may possibly join with some records in the unprocessed partitions are retained in the join buffer. [SE96] used this dynamic partitioning algorithm while utilizing the *Time Index* [EWK90] to determine the exact partitioning intervals so that each partition fits in memory. [LOT94] proposed a (plain) T-Join algorithm based on spatial partitioning. Here a record's interval i is mapped to a point ($i.start$, $i.end$ - $i.start$) in a two-dimensional space. These points are then indexed by an R-tree like method (the *TP-Index* [SOL94]) which partitions the space. However, a partition in one relation may be joined with many partitions in the other relation.

When an R-tree is used as an index, a temporal join can be considered as a special case of a spatial join. [BKS93] presents a depth-first while [HJR97] proposes a breath-first synchronized R-tree join algorithm. [Gun93] proposed join algorithms based on *Generalization Trees*. [APR+00] developed a plane-sweeping algorithm that unifies the index-based and non-index based approaches. The plane-sweeping phase of the algorithm needs to read records from the joining relations in non-decreasing order regarding one dimension (and then the stream processing of [LM93] can be applied). For the non-indexed environment, an initial sorting is sufficient. When the R-tree index exists, it is exploited to directly extract the data in sorted order according to the plane-sweep direction. This algorithm is an extension to the *scalable sweeping-based spatial join (SSSJ)* [APR+98] to the case of indexed inputs.

4 Synchronized GT-Join Algorithms

4.1 GT-Join based on Traditional Indices

A one-dimensional index like the B+-tree, clusters data primarily on a single attribute. Consider first a B+-tree that clusters on the interval start time. Because of the transaction-time environment, records are inserted in increasing time order; thus such an index can easily take advantage of sequential I/O. However, this scheme will be clearly inefficient for the GT-Join. Given a query interval i, a record rec may intersect i as long as $rec.start < i.end$. To answer a GT-Join query we have to scan the B+-tree for all such records, although most of them do not intersect i (since they are completely to the left of i). Clustering primarily on record *end* times is similarly inefficient.

Alternatively, we can utilize a B+-tree which clusters by keys. Nevertheless, synchronized join algorithm is not possible in this case. Besides the range-interval selection query, the join condition of the GT-Join only specifies that the joining records must have intersecting intervals. Since the B+-trees do not cluster records on time attributes, the range-interval selection query will give un-sorted output (in the sense that the selection result is not clustered by time attribute, either). This implies finding the records first and then join them in a separate phase (as in an unsynchronized approach).

With a multidimensional index like the R-tree, records are clustered by both key and time. Then a temporal join can be addressed as a special case of a spatial join. The depth-first [BKS93] and breath-first [HJR97] R-tree join algorithms follow the *Synchronized Tree Traversal (STT)* scheme. Initially the pair of root nodes is pushed into the stack. To process a pair of nodes that is popped from the stack, every record in the first node is joined with every record in the second node (if they satisfy a given condition). Eventually, at the leaf level, records in leaf nodes are joined. The R-tree join algorithms can be used to solve the GT-Join, with the following modifications:

- The original algorithms [BKS93, HJR97] join two complete R-trees. In our case, we are interested in records within the *query rectangle* defined by range r and interval i. Thus whenever a page is examined, only records that intersect the query rectangle are considered.

- We modify the condition for two tree nodes to join with each other. The original R-tree join algorithms join two nodes as long as they intersect, with the goal to eventually find pairs of leaf records which intersect with each other. In our GT-Join case, we want to find pairs of records whose intervals intersect; thus at higher levels of the tree, we join two nodes as long as they intersect in the time dimension, regardless of whether they intersect in the key dimension.

The disadvantage of the R*-tree based approaches emanates from their difficulty in storing the typically long time-intervals. Such intervals tend to increase the size of their bounding rectangles which introduces overlapping and in turn affects the join performance.

4.2 GT-Join using Temporal Indices

Temporal indices like the MVBT [BGO+96] achieve better clustering by introducing record copies: long intervals are broken into smaller ones that are stored in multiple places. Thus special care is needed for the range-interval selection query algorithm to avoid duplicates, i.e. reporting multiple copies of the same record. [BS96] proposed two range-interval selection query algorithms, a top-down and a sideways (link-based) one. We also present a variation of the link-based approach, the plane-sweep algorithm.

4.2.1 Top-Down GT-Join

The idea of the top-down GT-Join algorithm using MVBT is to perform the depth-first range-interval selection query on both MVBTs synchronously. Note that an MVBT can be viewed as a forest of ordered trees. The root node of each tree corresponds to a time interval, where for any root node n_i except the last one, $n_i.end = n_{i+1}.start$. For more details of the MVBT, we refer to [BGO+96]. To join two relations indexed by MVBTs, we join every pair of root nodes, one from each MVBT, whose time intervals intersect each other and intersect the query interval i. To join each such pairs (n_1, n_2), we perform STT of the two trees rooted by n_1 and n_2. Here the condition for two pages to join is that their intervals should intersect.

Similar to the above depth-first join algorithm, we can have a breadth-first join algorithm, which finishes one level of the MVBT before going to the next level. These two algorithms are similar to the top-down algorithms for the GTE-Join [ZTS02] (with the exception of the different join condition) and are omitted. Furthermore, both the *balancing condition optimization* and the *virtual height optimization* proposed there can be applied.

4.2.2 Link-based GT-Join

The link-based range-interval selection algorithm of [BS96] utilizes the concept of *predecessors*. Each leaf page of the MVBT corresponds to a rectangle in the two-dimensional key-time space; if for two pages A and B, their key ranges overlap and A is right before B in the space (i.e. the *end* of A's time interval is equal to the *start* of B's interval), then A is a *predecessor* of B. The link-based range-interval selection algorithm first finds the leaf pages which intersect the right border of the query rectangle, and then follows the predecessor links to find the other leaf pages that intersect the query rectangle. Once these leaf pages are identified, the task to select records from them is trivial.

We propose a new link-based algorithm for the GT-Join. This is a simpler solution than the link-based approach used for the GTE-Join [ZTS02]. The idea of the algorithm is to perform the link-based selection algorithm synchronously on the two MVBTs. First, it finds pairs of data pages: (1) whose rectangles intersect the right border of the query rectangle; and (2) whose intervals intersect each other. Then it follows predecessor records synchronously to find other pairs of data pages intersecting with the query rectangle and whose intervals intersect. To follow predecessor records synchronously, the following procedures are utilized: to choose a pair of data pages to join, use a priority queue to choose the one whose *end* time is the latest. While a pair of data pages are examined, if their *start* times are different, the page with the smaller *start* time is joined with the predecessors of the other page; if their *start* times are equal, the predecessors of one page are joined with the predecessors of the other.

4.2.3 Plane Sweep GT-Join

The plane-sweep join algorithm is similar to the link-based join algorithm in that it also starts with finding the data pages intersecting with the right border of the query rectangle, and it also proceeds by following predecessor links to find the other data pages.

The difference is that instead of keeping pairs of data pages in a single priority queue, we now keep a priority queue of individual pages. Furthermore, we maintain a buffer of records which may join with records to be identified later. At each step, we locate the leaf page which has the largest *end* time and select records from the page to buffer, and we insert the predecessors of the leaf page into the priority queue. Once a record is put into the buffer, we join it with the in-buffer records of the other relation. Also, as leaf pages are taken out from the queue, we locate *garbage records* (whose *start* is larger than the largest *end* time of any page in the queue) which are removed from the buffer.

5 Synchronized GE-Join Algorithms

5.1 GE-Join based on Traditional Indices

Using the B+-tree to cluster by time attributes is not efficient for the same reason as discussed in section 4.1. However, unlike the GT-Join case, it is possible to perform synchronized GE-Join based on B+-tree indices which cluster by keys. Since tree leaf pages are linked and records in them are ordered, the join algorithm starts with the leaf page in each tree that contains *r.low*, and proceeds by performing a sort-merge join until leaf pages with keys larger than *r.high* are met. The major shortcoming of this simple join algorithm is that it may encounter many records that do not participate in the join since their intervals do not intersect the query interval.

If the relations are indexed by R-trees, we can use the modified R-tree join algorithms. The modification from the original version [BKS93, HJR97] is similar to our discussion in the GT-Join case (section 4.1). The difference is that in order for two tree nodes to join, instead of requiring them to intersect in the time dimension, we now require them to intersect in the key dimension.

5.2 GE-Join using Temporal Indices

5.2.1 Top-Down GE-Join

Again, we can have depth-first and breadth-first join algorithms based on the MVBT. The algorithms are similar to the top-down approach of the GT-Join and are omitted. The difference is that to join two pages, instead of using the condition that they should intersect in the time dimension, we now require that they should intersect in the key dimension.

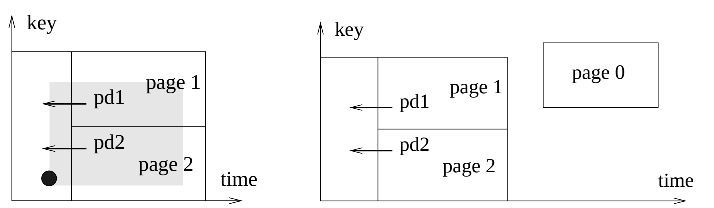
5.2.2 Link-based GE-Join

The link-based GE-Join algorithm is completely different from the link-based GT-Join algorithm. The reason is that now two leaf pages join even if their intervals do not intersect (as long as their key ranges overlap). Consider the leaf pages in a MVBT that intersect the query rectangle. They form a set of linked lists where the 'next' pointer between each pair of nodes is a predecessor link between two leaf pages; and the 'head' of each linked list is a leaf page which intersects the right border of the query rectangle. Consider the joining of two such linked lists, one from each joining MVBT. Every node in one linked list should be joined exactly once with every node in the other. We proceed with discussing an algorithm to join two linked lists and then relate the link-based GE-Join problem with it. The idea of joining linked lists is to join every node in linked list A with the whole linked list B. More formally, we give the algorithm as follows.

Algorithm *LinkedListJoin*(Linked-list A, B)

Push(*Stack*, [*A.head*, *B.head*]);
while(not IsEmpty(*Stack*)) do
    [a, b] = Pop(*Stack*);
    Join a with b;
    Push(*Stack*, [a, *b.next*]) if (*b.next* $\neq$NULL);
    Push(*Stack*, [*a.next*, b]) if (*a.next* $\neq$NULL and $b = B.head$);
endwhile

Consider each leaf page from the first MVBT which intersects the right border of the query rectangle. By following the predecessor links from it until the left border of the query rectangle is reached, we get a sequence of nodes. We consider such a sequence as a linked list. Similarly, we can get a linked list from the second MVBT. We thus can join the two lists using the above algorithm. If we consider every possible linked list, we can find every pair of leaf pages whose key ranges overlap. The issue which remains is how to avoid duplicates. This important issue arises due to the fact that a leaf page in the MVBT may have more than predecessors. Correspondingly, different linked-lists may share nodes. We need to make sure that the same pair of nodes are not joined multiple times.



(a) The PD condition (b) Case to release the PD condition

Figure 1: The PD condition and when it should be released.

To avoid duplicates, we borrow from [BS96] the concept of *reference point* and *predecessor condition (PD)*. The reference point is defined as the lower left intersection of the query rectangle and the predecessor page rectangle. Then the *predecessor condition (PD)* is defined as: "given record e, a predecessor record pd in page(e) and a query rectangle $rect$, the predecessor record is visited only if its reference point with respect to $rect$ falls in the key range of e". For example, in figure 1a, a predecessor record pd_1 in page 1 and a predecessor record pd_2 in page 2 point to the same page. Since the specified intersection point (the black dot) lies in the key range of page 2 and not page 1, only pd_2 is followed.

However, to apply the PD condition without further consideration will result in loss of join results. Figure 1b reveals the scenario. Suppose pages 1 and 2 belong to the first MVBT and page 0 belongs to the second MVBT. Assume the query rectangle is the whole key-time space. Obviously, the GE-Join algorithm should join page 0 with page 1 and with page 1's predecessor page (their key ranges overlap). However, the PD condition specifies that page 1's predecessor page will only be visited while examining page 2. Since page 2 does not join with page 0 at all (their key ranges do not overlap), applying the PD condition in this case results in lose of joining page 0 with page 1's predecessor. In general, when joining leaf A with leaf B from the other MVBT, if A's low key is no less than B's low key, we should join A with B's predecessor page regardless of whether the PD condition is true (i.e. the PD condition is *released* in this case).

5.2.3 Pipelined Sort-Merge GE-Join

Unlike the GT-Join case, if we perform the plane sweep algorithm for the GE-Join, it would be very inefficient. The reason is that the plane sweep scheme sweeps on the time dimension (from right to left), while the GE-Join condition does not require the joining records to have intersecting intervals. For the previous GT-Join case, due to the fact that joining records must have intersecting intervals, we can remove records from memory buffer as long as the sweep line is moved to the left of their *start* time. In our GE-Join case, however, records join as long as their keys are equal, irrelavent to whether their time intervals intersect or not. Thus we have to maintain all records in buffer.

Instead, we consider the following sort-merge with pipelining technique. It can be thought of as a semi-synchronized approach. The idea is as follows.

1. Perform range-interval selection queries on the first relation, sort the results, and then fill the memory buffer with the sorted selection result. If the selection result is larger than the memory buffer, part of it should be kept on disk. It is trivial to calculate the key range of the on-disk records.
2. As the range-interval selection query is performed on the second relation, for each result that is generated, join it with the in-memory records from the first relation. Then, the record can simply be discarded if its key does not belong to the key range of the on-disk records from the first relation. Otherwise, retain it.
3. After the selection query on the second relation is performed, join the on-disk records from relation one with the retained records from relation two.

6 Performance Analysis

6.1 Implemented Algorithms

Notation:	GT-Join:	GE-Join:	Meaning:
mvbt_df	4.2.1	5.2.1	Depth-first traversal
mvbt_bf	4.2.1	5.2.1	Breadth-first traversal
mvbt_link	4.2.2	5.2.2	Link-based traversal
mvbt_ps	4.2.3		Plane-sweep traversal
mvbt_sm		5.2.3	Pipelined sort-merge join
b+		5.1	Synchronized, find the start point using index and sort-merge on leaf pages
r^**_df*	4.1	5.1	Depth-first traversal using R*-tree
r^**_bf*	4.1	5.1	Breadth-first traversal using R*-tree
spj	3		Spatially partitioned join [LOT94]

Table 1: Implemented Algorithms

Table 1 lists the algorithms we implemented. For each algorithm, the index structure used can be seen from the notation. The plane sweep algorithm based on MVBT is only implemented for the

GT-Join, while the pipelined sort-merge join and the B+-tree join are only implemented for the GE-Join. The reasons are discussed in sections 4 and 5. We also implemented a spatially partitioned GT-Join using the approach in [LOT94]. For each implemented algorithm, we mention the section where the algorithm is discussed.

6.2 Experimental Setup

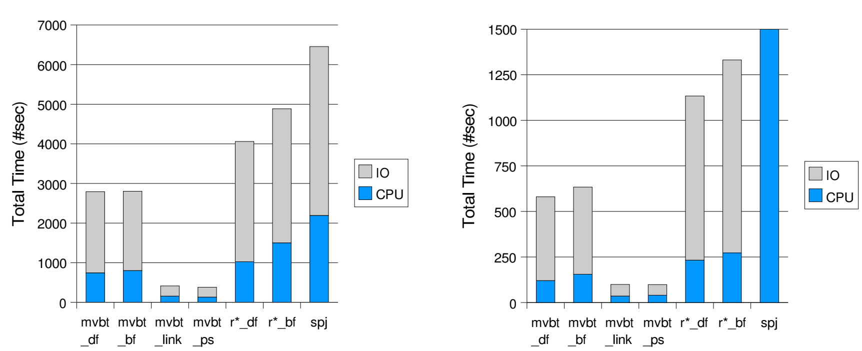
The algorithms were implemented in C and C++ using GNU compilers. The programs were run on a Sun Enterprise 250 Server machine with two UltraSPARC-II processors using Solaris 2.8. To compare the performance of the various algorithms we used the estimated running time. This estimate is commonly obtained by multiplying the number of I/O's by the average disk block read access time, and then adding the measured CPU time. We measured the CPU cost by adding the amount of time spent in *user* and *system* mode as returned by the *getrusage* system call. A random access was counted as 5ms on average.

For every index, an LRU buffering was used. For the R*-tree joins, besides using a LRU buffer, for each tree we also buffered all the nodes along the path from the root to the most recently accessed node. For the breadth-first joins, we used 15% of the memory buffer for storing and sorting the intermediate join results.

We created and joined two datasets: a uni-S and a uni-SM dataset. They were were first created using the TimeIT software [KS98] and then transformed to add record keys. Each dataset has 10 million records. Each actual record is 128 bytes long. The *key*, *start* and *end* attributes are each 4 bytes long. A dataset contains 50,000 unique keys where each key has on average 200 intervals. The uni-S dataset contains only short intervals (length about 1/10000 of the time space), while the uni-SM dataset contains 25% medium (length about 1/1000 of the time space) intervals and 75% short intervals. Here the time space is from 1 to 20 million.

Each experiment reports the average response over 10 randomly generated query rectangles with fixed rectangle shape and size. The shape of a query rectangle is described by the *R/I ratio*, where R is the length of the query key range divided by the length of the key space and I is the length of the query time interval divided by the length of the time space. The *query rectangle size (QRS)* is described by the percentage of the query area in the whole key-time space. Unless otherwise stated, the default parameters we used are: buffer size = 10MB, page size = 8KB, QRS = 0.1% and R/I ratio = 1.

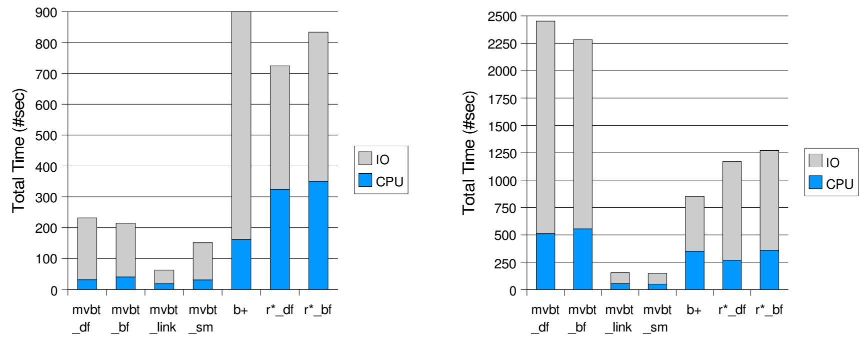
6.3 Join Performance



(a) R/I ratio = 10 (c) R/I ratio = 0.1

Figure 2: Performance of GT-Join, varying the R/I ratio.

Figure 2 compares the GT-Join performance. For both cases, the Link-based and the plane sweep MVBT algorithms have very similar performance (much faster than the competitors). The breadth-first and the depth-first MVBT based approaches do not perform well since they join two many index pages. The R-tree based join algorithms are much slower. The difference in clustering temporal data is apparent. The SPJ performs the worse since a partition in one relation needs to be joined with many partitions in the other relation. Also, the performance of SPJ deteriorates as the R/I ratio reduces. This is because the query rectangle covers less key space while the SPJ joins the whole key space.



(a) R/I ratio = 10

(c) R/I ratio = 0.1

Figure 3: Performance of GE-Join, varying the R/I ratio.

For the GE-Join query, the link-based MVBT algorithm performs the best. (Recall, that the plane-sweep MVBT algorithm is very inefficient for GE-Join – see section 5.4). Interestingly, we observe that when the R/I ratio is small, the pipelined sort-merge MVBT algorithm becomes a competitor. The reason is that it does not need to read a page from the indices more than once, while all the synchronized algorithms do. When the R/I ratio is small, the query rectangle intersects many pages from each relation with similar key ranges. Since the time attribute is not involved in the join predicate, most of these pages will join. Thus the problem for the synchronized algorithms worsens as the R/I ratio gets smaller. The performance of the other algorithms is drastically affected by the R/I ratio. The B+-tree algorithm performs relatively better as the R/I ratio reduces. But it is still not as good as the link-based one.

7 Conclusions

We studied the problem of efficiently processing semi-restrictive temporal joins (GT- and GE-join) when indices are available. We argued that traditional indexing schemes, like a B+-tree or an R*-tree do not lead to efficient join processing, due to their ineffectiveness in clustering temporal data. Instead we used a temporal index and we proposed various synchronized join algorithms. While we have concentrated on using the MVBT, our findings apply to other temporal indices as well. Our experimental results verified that temporal index based joins are more efficient than the B+-tree and R*-tree based joins. In particular, for both the GT-Join and GE-Join, the newly proposed link-based join algorithm has the most robust performance. It showed multi-fold improvement over the B+-tree/R*-tree joins.

References

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel and J. Vitter, "Scalable Sweeping-Based Spatial Join", *Proc. of VLDB*, 1998.

[APR+00] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold and J. Vitter, "A Unified Approach For Indexed and Non-Indexed Spatial Joins", *Proc. of EDBT*, 2000.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), 1996.

[BKS93] T. Brinkhoff, H. Kriegel and B. Seeger, "Efficient Processing of Spatial Joins using R-trees", *Proc. of SIGMOD*, 1993.

[BKS+90] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, "The R* tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, 1990.

[BS96] J. van den Bercken and B. Seeger, "Query Processing Techniques for Multiversion Access Methods", *Proc. of VLDB*, 1996.

[EWK90] R. Elmasri, G. Wuu and Y. Kim, "The Time Index: An Access Structure for Temporal Data", *Proc. of VLDB*, 1990.

[Gun93] O. Günther, "Efficient Computation of Spatial Joins", *Proc. of ICDE*, 1993.

[GS91] H. Gunadhi and A. Segev, "Query Processing Algorithms for Temporal Intersection Joins", *Proc. of ICDE*, 1991.

[HJR97] Y. Huang, N. Jing and E. Rundensteiner, "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations", *Proc. of VLDB*, 1997.

[JS99] C. Jensen and R. Snodgrass, "Temporal Data Management", *TKDE* 11(1), 1999.

[KS91] C. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data", *Proc. of SIGMOD*, 1991.

[KS98] N. Kline and M. Soo, "Time-IT, the Time-Integrated Testbed", ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz, Current as of August 18, 1998.

[KTF98] A. Kumar, V. J. Tsotras and C. Faloutsos, "Designing Access Methods for Bitemporal Databases", *TKDE* 10(1), 1998.

[LM93] T. Leung and R. Muntz, "Stream Processing: Temporal Query Processing and Optimization", in *Temporal Databases: Theory, Design, and Implementation*, (ed.) A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass, Benjamin/Cummings, 1993.

[LOT94] H. Lu, B. Ooi and K. Tan, "On Spatially Partitioned Temporal Join", *Proc. of VLDB*, 1994.

[LS89] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", *Proc. of SIGMOD*, 1989.

[RF93] S. Rana and F. Fotouhi, "Efficient Processing of Time-joins in Temporal Data Bases", *Proc. of Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, 1993.

[RS96] S. Ramaswamy and T. Suel, "I/O-Efficient Join Algorithms for Temporal, Spatial, and Constraint Databases", Bell Labs TechReport, URL: http://www.bell-labs.com/user/sridhar/ftp/suelrep.ps.gz, 1996.

[SE96] D. Son and R. Elmasri, "Efficient Temporal Join Processing using Time Index", *Proc. of SSDBM*, 1996.

[SOL94] H. Shen, B. Ooi and H. Lu, "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases", *Proc. of ICDE*, 1994.

[SS88] A. Segev and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment", *Proc. of SSDBM*, 1988.

[SSJ94] M. Soo, R. Snodgrass and C. Jensen, "Efficient Evaluation of the Valid-Time Natural Join", *Proc. of ICDE*, 1994.

[ST99] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data", *Computing Surveys* 31(2), 1999.

[VV97] P. Varman and R. Verma, "An Efficient Multiversion Access Structure", *TKDE* 9(3), 1997.

[Zur97] T. Zurek, "Optimization of Partitioned Temporal Joins", *Ph.D. thesis, University of Edinburgh*, 1997.

[ZTS02] D. Zhang, V. J. Tsotras and B. Seeger, "Efficient Temporal Join Processing using Indices" *Proc. of ICDE*, 2002.