Efficient Temporal Join Processing using Indices

Donghui Zhang* Vassilis J. Tsotras† Bernhard Seeger‡

Abstract

We examine the problem of processing temporal joins in the presence of indexing schemes. Previous work on temporal joins has concentrated on non-indexed relations which were fully scanned. Given the large data volumes created by the ever increasing time dimension sequential scan is prohibitive especially when the temporal join involves only parts of the joining relations (e.g., data related to a given time interval instead of the whole timeline). Utilizing an index becomes beneficial as it directs the join to the data of interest. A straightforward way to process an indexed temporal join is to first collect the data of interest from each relation –using the relation's index– and then join them. This approach is penalized by the cost of storing/retrieving intermediate results. Instead, a *synchonized* algorithm combines the join phase with index traversal. We consider synchronized temporal join algorithms for three representative indexing schemes, namely a B+-tree, an R*-tree and a temporal index, the Multiversion B+-tree (MVBT). Both the B+-tree and R*-tree indices result in simple but not efficient join algorithms. This is because neither index achieves good temporal data clustering. Better clustering is maintained by the MVBT index through record copying. Nevertheless, copies can greatly affect the correctness and effectiveness of the join algorithms. We identify these problems and propose efficient solutions. In particular we propose four synchronized join algorithms using the MVBT. Moreover, we introduce two optimization techniques to further improve their performance. We perform an extensive comparison of all index based temporal joins, using a variety of datasets and query characteristics. Our experiments show that the MVBT based join algorithms are consistently faster. In particular the *link-based* algorithm has the most robust behavior.

*Computer Science Department, University of California, Riverside, CA 92521. donghui@cs.ucr.edu

†Computer Science Department, University of California, Riverside, CA 92521. tsotras@cs.ucr.edu This work was partially supported by NSF (IIS-9907477, EIA-9983445) and the Department of Defense.

‡Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany. seeger@Mathematik.Uni-Marburg.de

1 Introduction

A temporal join is an important but costly operation for applications that maintain time-evolving data (data warehouses, temporal databases, etc.). A typical temporal record has a key, one or more time-varying attributes and a time interval representing the record's time validity. Various temporal join predicates have been proposed ([OS95]). Examples include the T-Join (join two records if their intervals intersect) and the more general TE-Join (join two records if their keys are equal and their intervals intersect) ([GS91]). Here we examine a generalization of the TE-Join (the $GTEJ$ join) which also specifies that joined records must have their keys in range r while their intervals should intersect interval i. Such joins allow the user to concentrate on interesting rectangles in the key-time space. As an example, consider two temporal relations, one maintaining the graduate students at UC Riverside and the other the IBM summer interns. A GTEJ query is: "find the students at UC Riverside with last names starting with 'B' that during 1995-2000 were also working at IBM".

Previous temporal join research has focused on non-indexed joins ([GS91, LM93, RF93, SSJ94, RS96, SE96, Zur97]). The common approach is to scan both relations and select the records that need to be joined. Even though a plain scan of both relations will take advantage of sequential I/O, it becomes prohibitive when the temporal relation sizes are large (as in a data warehouse) and the join selectivity is high (as in the GTEJ query). It is then advantageous to utilize existing indices (which most probably are present given the large volumes of temporal datasets).

In this paper we present efficient algorithms to process GTEJoins in the presence of indexing schemes. Due to limited space, algorithms for processing other interesting join predicates using indices are reported in ([ZTS00]). We assume that both relations are indexed and consider three representative access methods, namely, the B+-tree, the R*-tree ([BKS+90]) and a temporal index, the Multiversion B-tree (MVBT, [BGO+96]).

A straightforward approach to process an indexed GTEJ query is by first performing a range-interval selection query on each index and then joining the query results via some non-indexed algorithm.

While this *unsynchronized* approach is simple, it is penalized by the cost of storing and retrieving the selection results. Nevertheless, if the results of the selection queries are small and fit in main memory, this approach should be considered. In our experimental evaluation we include a pipelined version of an unsynchronized join algorithm based on the MVBT index. The MVBT was chosen since it has the fastest range-interval selection query time among the three indices ([ST99]).

More robust is the *synchronized* approach which combines the index traversal with the join phase. We thus present various synchronized join algorithms, using the three representative indices. The advantages of the B+-tree and the R*-tree is that they are widely used indices and lead to simple synchronized algorithms. However, these algorithms suffer from the ineffectiveness of the B+-tree and the R*-tree (to a lesser extent) in clustering temporal data. Temporal data is inherently multidimensional, with the time dimension having typically long intervals. Even the R*-tree is known to be problematic in clustering records with long temporal intervals ([KS91, ST99]).

Better clustering is achieved by a temporal index which typically creates many copies of records with long intervals. Effectively, a long interval is split into smaller but easily manageable intervals. This leads to fast processing of selection queries, but copies can drastically affect join processing. For example, copies of the same record can create duplicated join results that need to be filtered, etc. Moreover, to achieve fast updates, a temporal index typically updates only the latest copy of a given record, leaving the previous copies with an incorrect interval (the *incorrect end-time* problem). This does not affect selection queries, but a naive synchronized join algorithm may find records with incorrect intervals and thus report erroneous join results. Clearly, a temporal index based join algorithm needs to take into consideration the characteristics of temporal indexing.

The main contributions of this paper are summarized below:

1. We identify and solve the *incorrect end time* problem in temporal indexing which affects the correctness of join algorithms.

2. We propose four synchronized, temporal index based join algorithms. Moreover, we introduce

two optimization techniques that further improve join performance. Although we concentrate on the *Multiversion B+-tree (MVBT)* ([BGO+96]), our techniques and optimizations apply to other efficient tree-based temporal indices, as well ([LS89, VV97]).

3. We present an extensive performance study that compares the proposed algorithms with (a) the unsynchronized MVBT join algorithm, (b) the synchronized B+-tree algorithms, and, (c) the synchronized R*-tree algorithms. Our performance results show that the MVBT approaches are consistently faster. In particular, the newly proposed *link-based* MVBT algorithm, is universally the best for the GTE-Join, for a variety of datasets and join characteristics.

The rest of this paper is organized as follows. Section 2 summarizes the synchronized B+-tree and R*-tree approaches. Section 3 presents the new synchronized MVBT-based join algorithms. It also identifies and solves the incorrect end-time problem and presents the two optimization techniques. The results from the experimental comparison appear in section 4. Section 5 discusses previous work on temporal joins while section 6 presents conclusions and future work. An overview of the MVBT appears in the Appendix.

2 Synchronized Join Algorithms based on Traditional Indices

In the following, a key range r is specified by its $r.low$ and $r.high$ keys while a time interval is described by its *start* and *end* time. A range and an interval create a *rectangle* in the 2-dimensional key-time space. 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 GTE-Join is formally defined as: Given two temporal relations A and B, a key range r and an interval i, a record from A is joined with a record from B if, (i) their keys are in range r and are equal, and, (ii) their intervals intersect interval i and each other.

An 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, it will be clearly inefficient for the GTE-Join. Given a query interval i, the B+-tree will identify records with start times on or after $i.start$. Nevertheless, there may be many records that started long before $i.start$ and still intersect i. Clustering primarily on record end times is similarly inefficient. Instead we assume that each temporal relation uses a B+-tree that clusters first by key. This is useful given that a record can only join with records with equal 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 have the same key but should not be joined since their intervals do not intersect.

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 ([Gun93, BKS93, HJR97, APR+98, APR+00]). Both depth-first ([BKS93]) and breath-first ([HJR97]) synchronized tree join algorithms have been proposed. Initially the pair of tree 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. We implemented both the depth-first and the breadth-first joins. 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. 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 in turn affects the join performance.

3 Synchronized Join Algorithms using Temporal Indexing

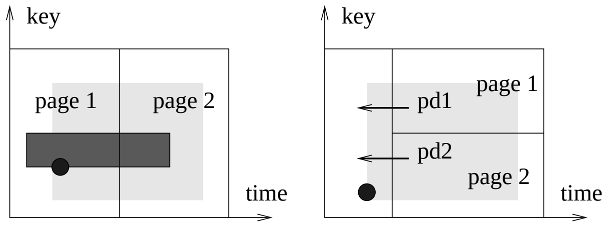
Important in processing joins with temporal indices (either in a synchronized or in an unsynchronized fashion) is how the range-interval selection query is performed. We start by discussing how this selection query is addressed with an MVBT (section 3.1). In section 3.2 we identify a problem with existing selection algorithms that affects the correctness of temporal joins and provide a solution. The synchronized MVBT approaches are categorized in top-down and sideways traversals. In section 3.3 we introduce the top-down synchronized traversals, namely, a depth-first and a breadth-first MVBT join algorithm. To improve their efficiency, we propose two optimization techniques (3.3.1, 3.3.2). Finally we present two sideways synchronized traversal algorithms, the link-based and the plane-sweep MVBT join algorithms (section 3.4). Note that the two optimization techniques apply to the sideways algorithms as well.

3.1 The Range-Interval Selection Query

An MVBT index record contains a key range, a time interval and a pointer to a page. The key range and time interval create a rectangle for this index record. In the following we say that two index records intersect, if their rectangles intersect.

A range-interval selection query (r, i) finds all records with keys in the range r and intervals intersecting interval i. A naive approach is to browse the MVBT in a depth-first manner and visit a record if it intersects with the query rectangle. This approach does find the correct answer set. Nevertheless, it suffers from the *duplicate result problem*: a data record may be reported multiple times. From the viewpoint of efficiency, a bigger problem is that a qualifying page may be visited more than once and the degree of redundancy has an exponential dependence in the height of the tree. The duplicate result problem occurs because of the many copies a given record may have. To avoid duplicate results, only one of these copies should be visited.

[BS96] provides a refined depth-first algorithm (DF_{ref}). The idea is to compute a reference point for each record, which is defined as the lower-left intersection of the record with the query rectangle. Let $page(e)$ denote the page pointed by an index record e. Given index record e, a record se in page(e) and a query rectangle r, record se is visited only if its reference point with respect to r is inside the rectangle of e. Consider figure 1a where the light-gray rectangle is the query rectangle and the dark-gray rectangle represents a record which is stored both in page 1 and in page 2. The reference point (the black dot) is a unique point in the key-time space among all the copies of a record. Since the reference point resides only in page 1 (and not in page 2), the record is visited only while examining page 1.



(a) The RP condition (b) The PD condition

Figure 1: The RP condition and the PD condition to avoid duplicates

[BS96] also provides a link-based range-interval algorithm ($Link_{ref}$) which conceptually searches the MVBT index *sideways* and is based on the predecessor/successor concept. Consider two data pages A and B in a MVBT, where A has been copied to B. Then A is called a *predecessor* of B and B is a *successor* of A. An index record called the *predecessor record* is saved in B pointing to A. Link_{ref} first finds all data pages that intersect the right border of the range-interval query rectangle. It then follows predecessor records to find all the other data pages whose rectangles intersect the query rectangle.

Finally, within each data page thus found, it reports all the records that intersect the query rectangle and whose reference point is inside this page's rectangle.

Due to the splitting policy of the MVBT, a data page can have at most two successors. To avoid visiting a page twice (while following the predecessor record in each successor), the Link_{ref} algorithm also utilizes a reference point. This point is defined as the lower left intersection of the query rectangle and the predecessor page rectangle. Then the $predecessorcondition(PD)$ is defined as: "given record e, a predecessor record pd in page(e) and a query rectangle r, the predecessor record is visited only if its reference point with respect to r falls in the key range of e". For example, in figure 1b, 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.

3.2 The Incorrect End Time Problem

Consider the example in figure 2. At time t_1, the page that contains record x_1 is split and thus x_1 is copied to x_2 in a successor page. Later on the end time of x_2 is updated to t_2, but the end time of x_1 is still kept as now. (The MVBT and the other temporal access methods do not update any previous copy for efficiency). Suppose the query rectangle is the whole key-time space. Both the DF_{ref} and Link_{ref} range-interval selection algorithms from [BS96] will report x_1 but not x_2. Let y be a record in the other relation. Since $x_1.end = now$, any join algorithm utilizing these selection algorithms will assume that $x_1.interval$ intersects $y.interval$, which is obviously wrong. We call this problem the *incorrect end time* problem. To solve it, we propose to replace the reference points of [BS96] with the *right reference time (RRT)*. The RRT of a record se with respect to a query rectangle r is the mininum time between $se.end$ and $r.interval.end$. The $RRTcondition$ is defined as: "given index record e, a data record se in page(e) and a query rectangle r, record se is visited if its RRT with respect to r is inside $e.interval$". For example, in figure 2, the RRT for x_1 is now, which is

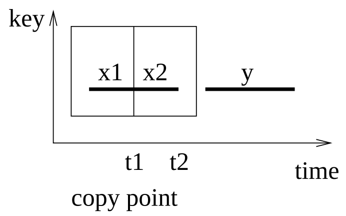


Figure 2: The incorrect end time problem

not in the page containing x_1. Instead, the RRT for x_2 is t_2, which is in the page containing x_2. So the modified range-interval selection algorithms will report x_2, which has the correct end time. We call the modified depth-first and link-based range-interval algorithms DF_{right} and $Link_{right}$ respectively.

3.3 Top-Down Approaches

The idea of the depth-first join algorithm (*MVBT_DF*) is to perform range-interval queries using $\mathrm{DF}_{\mathrm{right}}$ for the two MVBT indices synchronously.

Algorithm *MVBT_DF*(MVBT $mvbt_1$, $mvbt_2$, Rectangle r)

```
 1. for every root e1 of mvbt1 which intersects r
 2.     for every root e2 of mvbt2 which intersects r
 3.         Push( Stack, [e1, e2] );
 4.     endfor
 5. endfor
 6. while ( not IsEmpty(Stack) ) do
 7.     [e1, e2] = Pop( Stack );
 8.     if both e1 and e2 point to index pages, then
 9.         for every record se1 in page(e1) which intersects r and satisfies the RRT condition
10.             for every record se2 in page(e2) which intersects r and satisfies the RRT condition
11.                 if the key ranges of se1 and se2 intersect, Push( Stack, [se1, se2] );
12.             endfor
13.         endfor
14.     else if only e1 points to an index page
15.         for every record se1 in page(e1) which intersects r and satisfies the RRT condition
16.             if the key ranges of se1 and e2 intersect, Push( Stack, [se1, e2] );
17.         endfor
18.     else if only e2 points to index page, then
19.         // similar; omit
20.     else // both e1 and e2 point to data pages
21.         for every record se1 in page(e1) which intersects r and satisfies the RRT condition
22.             for every record se2 in page(e2) which intersects r and satisfies the RRT condition
23.                 if the keys of se1 and se2 are equal and the intervals of them intersect, then
24.                     Output( [se1, se2] );
25.                 endif
26.             endfor
27.         endfor
28.     endif
29. endwhile
```

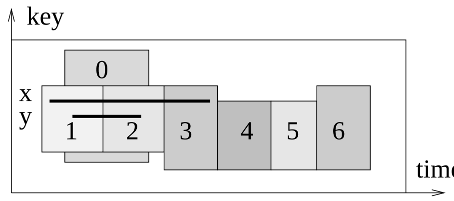
Steps 1-5 pushes the pairs of root pages which intersect the query rectangle r into the stack. After that, for every pair of records popped from the stack, three cases are differentiated. Steps 8-13 correspond to the case when both records point to index pages. The algorithm examines every pair of records, one from each index page. Every pair (se_1, se_2) is pushed into the stack if both se_1 and se_2 intersect r, both satisfy the RRT condition, and the key ranges of them intersect. Steps 14-19 correspond to the second case when one record (e.g. e_1) points to an index page and the other (e.g. e_2) points to a data page. The algorithm checks e_2 against every record se_1 in page(e_1). To decide whether to push (se_1, e_2) into the stack, the same condition as in case 1 is used. The last case (steps 20-27) is to join two data pages. Every pair of data records are joined as long as they satisfy the selection condition and the join condition.

In the breadth-first join algorithm the join proceeds one level at a time. All pairs of records to be joined at one level are found and processed before going into the next level. The details are omitted but can be found in [ZTS00].

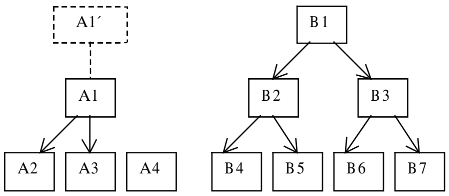
3.3.1 The Balancing Condition Optimization

In steps 11 and 16 of algorithm MVBT_DF, a pair of index records are pushed into the stack as long as their key ranges intersect. That is, it is not required that their time intervals should intersect as well. The reason is illustrated in figure 3a. Suppose record x and pages 1 through 6 belong to the first MVBT, while record y and page 0 belong to the second MVBT. Both x and y have the same key (for ease of illustration, y is drawn below x) and thus they should join. However, according to $\mathrm{DF}_{\mathrm{right}}$, x can only be visited while examining page 3 and y can only be visited while examining page 0. In order for the join algorithm to report (x, y) in the join result, page 3 and page 0 have to be joined, even though their intervals do not intersect. This is necessary in order not to lose join results. But it greatly affects the efficiency of the join algorithms because too many pairs of index records are joined (e.g. page 0 has to join with pages 1 through 6). Below we give an optimization technique that leads to a far more efficient solution, where a page joins only with pages which it intersects.

The idea of the *balancing condition optimization (BCO)* technique is to balance between the following two conditions: (1) requiring that the RRT condition holds for both joining records (currently required in MVBT_DF; and (2) requiring that the intervals of joining index records should intersect (currently not required). In our approach, to increase efficiency, we always require the intervals of joining index records to intersect, while we allow the algorithms to visit a record even if the RRT condition is false (under certain constraints to be discussed next). For example, in figure 3a, the optimized join algorithms do not join page 3 with page 0 since they do not intersect. In order not to lose join result, the algorithm should join (x, y) somewhere else. The candidate places are when page 1 and page 0 are joined and when page 2 and page 0 are joined, since both pages 1 and 2 contain x and intersect page 0 (note that the RRT condition is false for x in both cases). To make sure that (x, y) is not reported multiple times, we set two more constraints. First, we require that the RRT condition holds for at least one of the joining records. For example, in figure 3a, the RRT condition for y is true. Second, we require that the page containing the other record (where the RRT condition is false) should have a larger *end* time. For example, since page 2 (and not page 1) has an *end* time larger than that of page 0, the algorithm joins (x, y) only when page 2 and page 0 are joined.



(a) The balancing condition optimization



(b) The virtual height optimization

Figure 3: Two optimization techniques.

3.3.2 The Virtual Height Optimization

The *virtual height optimization (VHO)* technique can be utilized to improve the performance while joining any two balanced trees. The idea is illustrated in figure 3b. The node with a dashed rectangle (A_1') is a virtual node so that the two trees appear as if they had the same height. Suppose every node in tree A joins with every node in tree B. Without the VHO, we first need to join the pair $\langle A_1, B_1\rangle$. At the middle level, we need to join pairs: $\langle A_2, B_2\rangle$, $\langle A_3, B_2\rangle$, $\langle A_4, B_2\rangle$, $\langle A_2, B_3\rangle$, $\langle A_3, B_3\rangle$, $\langle A_4, B_3\rangle$. Finally, at the leaf level, we join every leaf node in tree A with every leaf node in B. With the VHO, what is joined at the top level and at the leaf level remains unchanged. However, at the middle level, only $\langle A_1, B_2 \rangle$, $\langle A_1, B_3 \rangle$ are joined. Clearly, the bigger the difference in the height of the two trees, the more significant the benefit of this optimization should be.

Although the MVBT is not a tree but a graph, the VHO still applies. The reason is that the MVBT is a *balanced forest*: it has multiple root nodes and the sub-tree under any root node is a balanced tree.

3.4 Sideways Approaches

In contrast to the top-down approaches, the sideways approaches first join data pages that intersect the right border of the query rectangle and then follow predecessor records synchronously to examine the rest data pages. Both the BCO and the VHO optimizations apply.

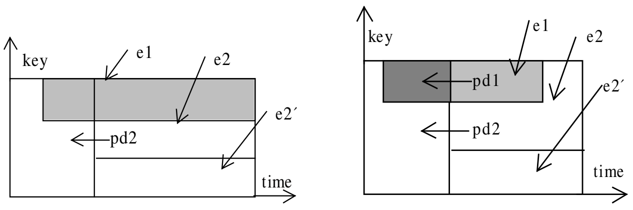
3.4.1 Link-based Join Algorithm

The idea of the link-based join algorithm (*MVBT_Link*) is to synchronously perform range-interval queries on both trees using $\text{Link}_{\text{right}}$. The detailed algorithm is outlined below.

Algorithm *MVBT_Link*(MVBT $mvbt_1$, $mvbt_2$, Rectangle r)

1. Browse down $mvbt_1$ and $mvbt_2$ synchronously to find every pair of data pages (e_1, e_2) such that: (1) both e_1 and e_2 intersect the right border of r; and (2) e_1 intersects e_2.
2. Follow predecessor records synchronously to find the other pairs of data pages intersecting each other and each of which intersects r. To do so, while examining a pair of data pages (e_1, e_2), if $e_1.start = e_2.start$, join the predecessors of e_1 with the predecessors of e_2. If $e_1.start < e_2.start$, join e_1 with the predecessors of e_2. If $e_2.start < e_1.start$, join e_2 with the predecessors of e_1.
3. To join each pair of data pages, use steps 21 through 27 of the depth-first algorithm MVBT_DF.

An important issue that arises is how to guarantee that each pair of data pages which should be joined is joined once and only once. First of all, since two pages may have the same predecessor, the selection algorithm $\text{Link}_{\text{right}}$ utilizes the PD condition to make sure that the predecessor is visited only once. We could utilize the PD condition, too. This successfully avoids duplicates. However, without extra care, this method misses some join result. This is illustrated in figure 4a. Record e_1 points to a data page (the shadowed page) in one MVBT. Records pd_2, e_2 and e_2' point to data pages in the other MVBT. The query rectangle is the key-time space. The page that pd_2 points to is a predecessor page of both page(e_2) and page(e_2'). The PD condition ensures that page(pd_2) is checked only when examining page(e'_2). However, e_1 does not join with e'_2 since they do not intersect. So if we always require the PD condition to hold, page(e_1) and page(pd_2) will not be joined. To solve the problem, we need to identify the cases where we need to *release* the PD condition, that is, we follow a predecessor record even if the PD condition is false. Below we discuss two such cases, assuming that the current joining pair is (e_1, e_2).



(a) Illustration of case 1 (b) Illustration of case 2

Figure 4: Two cases where the PD condition needs to be released.

Case 1: Let $e_1.start < e_2.start$ (the case when $e_1.start > e_2.start$ is similar). We need to examine whether to join e_1 with each predecessor record pd_2 in page(e_2). If $e_1.low \geq e_2.low$ (figure 4a), the PD condition on pd_2 needs to be released, since $e_1.low \geq e_2.low$ means that e_1 does not intersect the other page (pointed by e'_2) which contains a predecessor record pointing to page(pd_2).

Case 2: $e_1.start = e_2.start$. We need to examine whether to join each predecessor record pd_1 in page(e_1) with each predecessor record pd_2 in page(e_2) where pd_1, pd_2 intersect each other and both intersect the query rectangle. When the PD condition is true for only one of the predecessor records, say for pd_1, the PD condition for pd_2 needs to be released if $e_1.low \geq e_2.low$ (figure 4b). When the PD condition is false for both pd_1 and pd_2, there is no need to join them (as proved in [ZTS00]). With the above modifications the MVBT_Link join algorithm will provide the correct join result.

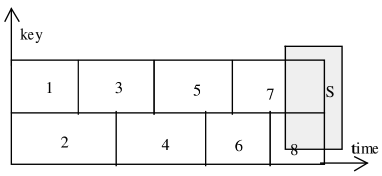


Figure 5: Motivation for the order-by-time optimization

In order to improve the MVBT_Link performance, we propose the *order-by-time optimization*. To motivate the need for this optimization, consider figure 5. The numbered rectangles represent the data pages in $mvbt_1$. The shadowed rectangle S represent a data page in $mvbt_2$. MVBT_Link starts by pushing pairs (7, S) and (8, S) into the stack. Then it pops (8, S) from the stack and joins the two pages. After that, it joins the predecessors of them and so on, until the left border of the query rectangle is reached. When the algorithm eventually pops (7, S) from the stack, chances are that page S is already switched out of memory and needs to be read again, causing an extra I/O. To avoid this phenomenon, the order-by-time optimization keeps all pairs of data pages not in a stack, but in a priority queue, ordered in decreasing order of the end time of data pages. For example, in figure 5, the optimization ensures that after (8, S) is joined, instead of joining the predecessors of S with page 8 and its predecessors, page 7 and S are joined since they have the largest overall end time.

Yet another optimization technique is to utilize sequential I/O. Typically a random disk access costs at least 30 times more than a sequential I/O. The motivation is that the MVBT creates pages in increasing time order and the link-based algorithm accesses data pages in decreasing time order. So when a data page A is needed, it is very likely that the data pages created right before A was created will be needed very soon. So in the link-based join algorithm, whenever a page miss happens, we read in multiple consecutive pages instead of one. E.g. if a page with number 1070 is needed, we read pages 1041 through 1070. This optimization applies to the *plane-sweep* algorithm to be presented in the next section as well.

3.4.2 Plane-Sweep Join Algorithm

The plane-sweep join algorithm (*MVBT_PS*) 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.

Algorithm *MVBT_PS*(MVBT $mvbt_1$, $mvbt_2$, Rectangle r)

1. Browse down both $mvbt_1$ and $mvbt_2$ to find data pages intersecting the right border of r;
2. Find the records in these pages and store them in buffer;
3. Store the predecessors of these pages, in decreasing order of page end time, in priority queues PQ_1 and PQ_2 for $mvbt_1$ and $mvbt_2$, respectively;
4. Set $largest\_end_1$ and $largest\_end_2$ to be the largest end time of any page in PQ_1 and PQ_2, respectively;
5. Join the in-buffer records;

6. Remove records from buffer whose *start* is no less than *largest_end* of the other index;
7. while at least one out of PQ_1 and PQ_2 is not empty
8.   if $largest\_end_1 \geq largest\_end_2$, then
9. $e = \text{Pop}(PQ_1)$;
10.   Add in PQ_1 the predecessors of e which intersect r and which satisfy the PD condition;
11.   Adjust $largest\_end_1$;
12.   Add to buffer the qualifying records from page(e);
13.   Join the added records with in-buffer records of $mvbt_2$;
14.   Remove such in-buffer record rec of $mvbt_2$ that $rec.start \geq largest\_end_1$;
15.   else
16.   // similar, omit
17.   endif
18. endwhile

Steps 1 through 6 do initialization. Records from both MVBTs which reside in data pages intersecting the right border of the query rectangle are located and joined. Out of these records, only the ones that may join with future records are retained in the buffer. In order to detect the *garbage records*, which for sure will not join with any record to be reported later in the execution of the algorithm, the following property of $\text{Link}_{\text{right}}$ is utilized. "Given a data page A which intersects the query rectangle r but does not intersect with the right border of r. When $\text{Link}_{\text{right}}$ examines A, any record it reports has end time within the interval of A ". Since all data pages of $mvbt_1$ to be examined later will have end time no larger than $largest\_end_1$, all $mvbt_1$ records to be found later will also have end time no larger than $largest\_end_1$. So, as step 6 reveals, an in-buffer $mvbt_2$ record becomes garbage if its start time is no less than $largest\_end_1$.

In the while loop from step 7 to step 18, the other data pages intersecting the query rectangle are examined. Each loop reads in one data page and adds qualifying records from the page into memory buffer. Next, the newly added records are joined with in-buffer records. Last, since one *largest_end* may become less, step 14 eliminates garbage records.

Discussion: There are three differences between MVBT_Link and MVBT_PS. (1) While MVBT_Link keeps pairs of data pages in the priority queue, MVBT_PS keeps individual data pages. (2) MVBT_PS needs to keep data records in memory. (3) MVBT_PS visits each data page intersecting the query rectangle only once.

The differences (1) and (2) reveal an interesting tradeoff of the memory utilization between MVBT_PS and MVBT_Link. Compared with MVBT_Link, MVBT_PS uses more memory to maintain the data records already found which may be used later. When there are too many such records, some of them are written to disk, causing extra I/O. On the other hand, since MVBT_Link keeps pairs of data pages (actually pairs of index records pointing to data pages) rather than individual data pages, and since one data page may appear in multiple pairs, MVBT_Link tends to have a larger priority queue. Difference (3) shows another advantage of MVBT_PS. However, this advantage is not significant since that MVBT_Link has been optimized with the order-by-time optimization.

Note that the MVBT_PS algorithm resembles the stream processing approach of [LM93] and the *priority query driven R-tree join (PQ)* of [APR+00]. A difference is that when adding records to the in-memory buffer, MVBT_PS adds a set of records at a time, while PQ adds one at a time. Another difference is that the MVBT_PS focuses on leaf pages, while PQ examines the whole tree.

4 Performance Analysis

4.1 Implemented Algorithms

Table 1 lists the algorithms we implemented.

Notation:	Access Method:	Meaning:	Section:
mvbt_df	MVBT	Synchronized, depth-first traversal	3.3
mvbt_bf	MVBT	Synchronized, breadth-first traversal	3.3
mvbt_link	MVBT	Synchronized, link-based traversal	3.4.1
mvbt_ps	MVBT	Synchronized, plane-sweep traversal	3.4.2
mvbt_sm	MVBT	Unsynchronized, selection query followed by sort-merge join	4.1
b+	B+-tree	Synchronized, find the start point using index and sort-merge on leaf pages	2
r^*_df	R^*-tree	Synchronized, depth-first traversal using R^*-tree	2
r^*_bf	R^*-tree	Synchronized, breadth-first traversal using R^*-tree	2

Table 1: Implemented Algorithms

Since the performance of an unsynchronized approach depends on how fast the range-interval selection query is performed, we used the MVBT as the index (because it answers this query faster than the other indices). After the records in the query rectangle are selected they can be joined using traditional methods (sort-merge, block-nested-loop, etc.). One benefit of the unsynchronized approach is that it can utilize sequential I/O, since the selection results can be written to disk and later retrieved from disk sequentially. Furthermore, we utilize a pipelining technique for additional speed-up. After qualifying records are selected from the first index (A), a portion of the selection result is kept in memory. As each record x from the second index (B) is selected, it is joined with the in-memory A records. Record x is then discarded if it is certain that it will not join with any on-disk A record. After the selection on B is finished, if there are still some on-disk records from both A and B, they are joined using traditional techniques. We actually implemented a sort-merge and a block-nested-loop version of the unsynchronized MVBT algorithm. However, in all our experiments, the sort-merge version (*mvbt_sm*) outperformed the block-nested-loop which is omitted.

For the synchronized B+-tree join algorithm (denoted as $b+$), the records are actually clustered first by key and then by *start* time. For the synchronized R*-tree join algorithms we implemented both the depth-first and the breadth-first joins (r^*_df, r^*_bf).

4.2 Experimental Setup

The algorithms are implemented in C and C++ using GNU compilers. The programs run on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.6. To compare the performance of the various algorithms we use 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 measure 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 is counted as 10ms on average. A sequential access is counted as 1/30 of a random access time.

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

With the exception of the dataset used in section 4.3, all datasets 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. The default key space and time space are both defined as [1, 20 million). A dataset contains 50,000 unique keys where each key has on average 200 intervals. We define three kinds of intervals: *short* (length about 1/10000 of the time space), *medium* (1/1000 of the time space) and *long* (1/100 of the time space). We use the collection of datasets shown in table 2. To study the effect of joining mainly long intervals, a uni-LM and a uni-M datasets are joined. To study the effect of joining mainly short intervals, a uni-S and a uni-SM datasets are used.

Dataset	Description
uni-LM	keys uniformly distributed; 25% long and 75% medium intervals
uni-M	keys uniformly distributed; 100% medium intervals
uni-SM	keys uniformly distributed; 25% medium and 75% short intervals
uni-S	keys uniformly distributed; 100% short intervals

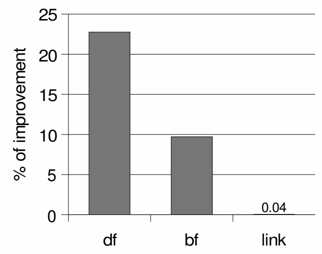
Table 2: Dataset Description

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 use are: buffer size = 10MB, page size = 8KB, QRS = 1% and R/I = 1.

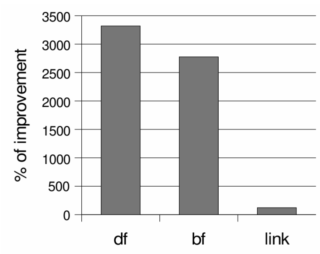
4.3 Improvement due to the optimization techniques

The virtual height optimization focuses on eliminating the number of intermediate index nodes visited during a join. It becomes important when the heights of the joined trees are substantially different.

To observe this, we create a dataset with 50K records that are never deleted. Hence, the height of the MVBT will increase as time proceeds, creating a large difference between the latest and the earliest B+-trees in the MVBT graph. Figure 6a shows the results using the VHO on the MVBT, for a GTEJ query that self-joins the above dataset where QRS=100%. The depth-first (*df*) and breadth-first (*bf*) approaches are clearly improved. The link-based (*link*) algorithm has little improvement. This is expected since the link-based algorithm focuses on data pages while the VHO helps only in the intermediate levels.



(a) Improvement due to the VHO



(b) Improvement due to the BCO

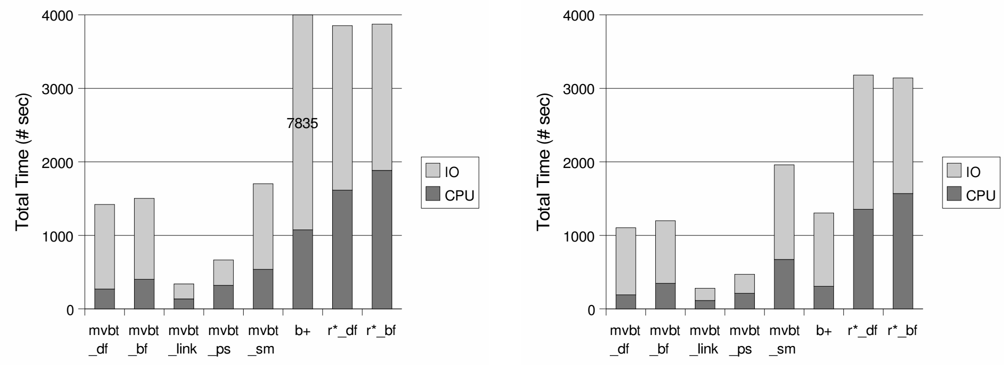
Figure 6: The VHO and BCO optimizations

The improvement due to the balancing condition optimization is drastic, especially when the R/I ratio is small. Figure 6b shows the results of a GTEJ query between a uni-LM and a uni-M dataset each of which has 0.5 million records, with R/I ratio being 0.1. With a small R/I ratio the query rectangle covers a large portion of the time space, and thus the algorithms without the BCO perform many unnecessary joins of pages. The improvement of the link-based join algorithm is also substantial but is relatively smaller because it examines only a few index pages. In the following experiments both the VHO and the BCO have been applied if applicable.

4.4 GTE-Join Performance

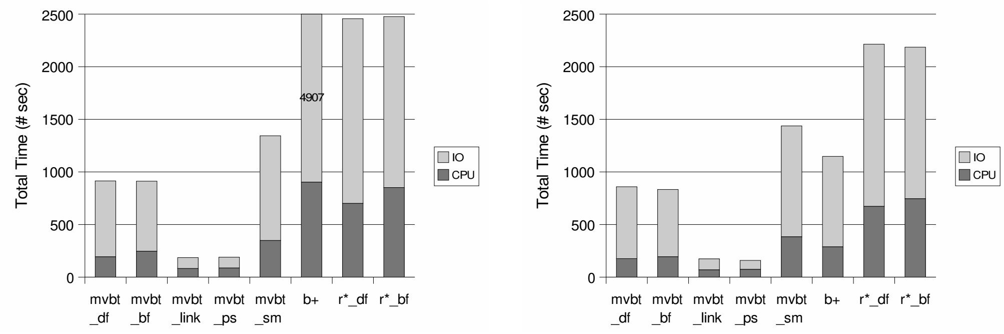
Figure 7 compares the join performance with varying R/I ratio, using datasets with mainly large intervals. The four MVBT-based synchronized approaches have overall more robust performance. Among them the two sideways approaches (*mvbt_link* and *mvbt_ps*) are the best.

The unsynchronized sort-merge algorithm (*mvbt_sm*) does not perform well because there are many



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

Figure 7: Joining mainly large intervals



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

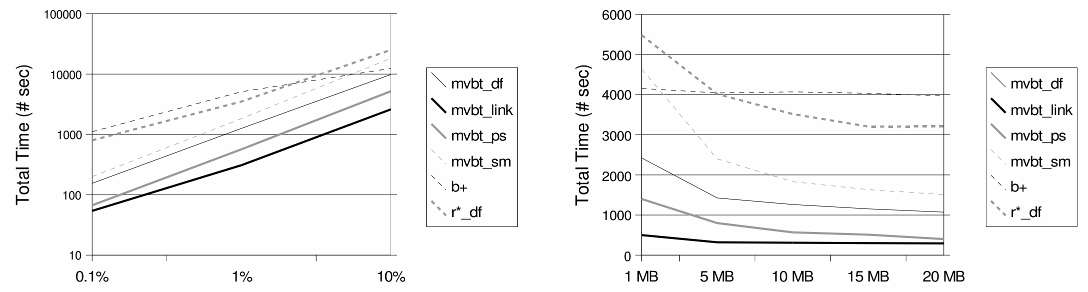
Figure 8: Joining mainly short intervals

records satisfying the range-interval query and thus it is expensive to maintain them (storing, sorting, etc.). Its performance improves as the R/I ratio becomes large. This is because the query rectangle covers relatively more of the key space and fewer intervals with the same key, and thus a record joins with fewer records in the other relation.

As expected, the synchronized B+-tree join does not perform well since it accesses records that should not be joined. This problem worsens with a large R/I ratio. Similarly, the R*-tree based algorithms perform worse than the MVBT ones. This is also expected, because R*-trees are affected by the interval overlapping on the time dimension. The R*-tree breadth-first join needs more CPU time but less I/O time than the R*-tree depth-first join. Their overall performance however is similar.

All algorithms perform better when joining short intervals since there are fewer join results (figure 8). The performance of the R*-tree based approaches improves with short intervals because the the degree of overlapping among the sibling pages is reduced.

Clearly from Figures 7 and 8 the synchronized sideways MVBT approaches perform better than their top-down counterparts. The reason is that top-down approaches join a lot of index pages, while the sideways approaches benefit by focusing on the data pages only. Moreover, the sideways approaches can take advantage of the sequential I/O optimization. Between the two sideways approaches, the link-based one (*mvbt_link*) is more robust. The reason is that the plane-sweep algorithm (*mvbt_ps*) keeps records in buffer as long as they are needed for joining with future records. If there are many such records, some may have to be kept on disk, which affects the join performance. This problem worstens especially when joining long intervals or with a large R/I ratio. The only case when the plane-sweep algorithm performs better than the link-based one is when joining short intervals while the R/I ratio is small (figure 8b). But even then, the advantage is only marginal.



(a) Varying QRS (b) Varying memory buffer size

Figure 9: Varying QRS and buffer size

Figure 9a compares the performance of the algorithms while varying the QRS while Figure 9b presents the performance when varying the memory buffer size. The datasets used are the uni-LM and uni-M, i.e. joining mainly long intervals. Since we have observed that the breadth-first joins (mvbt_bf, r*_bf) and their depth-first counterparts (mvbt_df, r*_df) have similar performance, for clarity we omit the performance of the breadth-first algorithms.

As the QRS varies (figure 9a), the link-based algorithm is always the best choice. When the QRS is small, the plane-sweep algorithm becomes a competitor, too. The reason is that for a smaller QRS, there are fewer records in the query rectangle with intervals containing any given time, which leads the plane-sweep algorithm to have fewer records to maintain in buffer. Another interesting observation is that when the QRS becomes larger, the synchronized B+-tree based join performs better than the synchronized R*-tree join and the unsynchronized sort-merge join. The reason is that with a larger QRS, the query rectangle covers more of the time space. Hence the leaf pages of the B+-tree have fewer records whose intervals do not intersect the query rectangle.

As expected, all algorithms improve as the available memory buffer increases (figure 9b). Nevertheless, the link-based algorithm is again the fastest. With large buffer size, the plane-sweep based algorithm becomes a competitor. With a large buffer, the plane-sweep algorithm can keep more records in memory and thus the I/O needed for the join is less. Interestingly, the B+-tree based join is rather independent from the size of available buffer. Since the leaf records of the B+-tree are already sorted, the join processing is basically synchronized scan of two sorted lists and not much buffer is needed.

We also experimented with various other cases: joins involving negative exponentially distributed keys, joins involving normally distributed keys, joins varying page sizes, etc. The results are omitted but can be found in [ZTS00]. However, in all experiments we performed, we got the same conclusion, i.e., the two synchronized MVBT sideways traversals are the best and among these two, the link-base algorithm has overall the more robust performance.

5 Previous Work

Research on temporal joins has focused on non-indexed algorithms. [RF93] assumes the smaller relation fits in memory and proposes seven nested-loop T-Join algorithms. [GS91] provides sort-merge T-Join and TE-Join algorithms when one or both relations are sorted. [LM93] assumes that the relations are sorted on the start time of the record intervals and discusses 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 assumes the relations are sorted and discusses 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] uses this dynamic partitioning algorithm while utilizing the *Time Index* ([EWK90]) to determine the exact partitioning intervals so that each partition fits in memory. [LOT94] proposes a 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 Time Polygon Index) which partitions the space. However, a partition in one relation may be joined with many partitions in the other relation ([ZTS00]).

In addition to ([BKS93, HJR97]) indexed spatial joins have been also considered in ([Gun93, APR+00]). [Gun93] proposes join algorithms based on *Generalization Trees*. [APR+00] develops 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.

6 Conclusions & Future Work

We studied the problem of efficiently processing temporal joins when pre-existing indices are available. We concentrated on GTE-Joins and argued that traditional indexing schemes, like a B+-tree or an R*-tree do not lead to efficient join processing, mainly due to their ineffectivenes in clustering temporal data. A better approach is to use a temporal index. Various problems arise due to the temporal index characteristics, like the introduction of record copies or the incomplete updating of previous record copies, and can affect the correctness and/or efficiency of the temporal join. Unfortunately, known techniques for range-interval queries do not apply when processing temporal joins. We identified these problems and provided efficient solutions. More specifically, we proposed four synchronized, temporal index based join algorithms. While we have concentrated on using the MVBT, our findings apply to other temporal indices as well. We also presented two optimization techniques that further improve the join processing. We compared the proposed synchronized indexed temporal join algorithms with (i) a straightforward unsynchronized (but optimized using the pipelining technique) join approach, (ii) a synchronized B+-tree based approach, and (iii) a synchronized R*-tree based join. Our experimental results verified that temporal index based joins are more efficient. In particular, the newly proposed link-based join algorithm has the most robust performance. This algorithm behaves similarly well for other temporal join queries (like a generalized T-Join and a generalized Equi-Join; we refer to [ZTS00] for a complete list of the performance results).

While this paper concentrates on indexed temporal joins, an open question is whether hashing based schemes can be applied. One approach is to consider spatial hashing methods ([LR96, PD96]). However, spatial hash joins have two possible drawbacks. First, a record is copied in all buckets that intersect its interval. Temporal intervals tend to be long and will be copied in many buckets, which will affect performance. Second, such partition based joins are not very dynamic. Initial partitioning may deteriorate as records are updated. We plan to examine hashing approaches tailored to temporal data ([KT00]) as well.

References

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel and J. Vitter, "Scalable Sweeping-Based Spatial Join", *Proc. of VLDB*, pp. 570-581, 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*, pp. 413-429, 2000.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), pp. 264-275, 1996.

[BKS93] T. Brinkhoff, H. Kriegel and B. Seeger, "Efficient Processing of Spatial Joins using R-trees", *Proc. of SIGMOD*, pp. 237-246, 1993.

[BKS+90] N. Bechmann, H. Kriegel, R. Schneider and B. Seeger, "The R* tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, pp. 322-332, 1990.

[BS96] J. van den Bercken and B. Seeger, "Query Processing Techniques for Multiversion Access Methods", *Proc. of VLDB*, pp. 168-179, 1996.

[EWK90] R. Elmasri, G. Wuu and Y. Kim, "The Time Index: An Access Structure for Temporal Data", *Proc. of VLDB*, pp. 1-12, 1990.

[Gun93] O. Günther, "Efficient Computation of Spatial Joins", *Proc. of ICDE*, pp. 50-59, 1993.

[GS91] H. Gunadhi and A. Segev, "Query Processing Algorithms for Temporal Intersection Joins", *Proc. of ICDE*, pp. 336-344, 1991.

[HJR97] Y. Huang, N. Jing and E. Rundensteiner, "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations", *Proc. of VLDB*, pp. 396-405, 1997.

[JS99] C. Jensen and R. Snodgrass, "Temporal Data Management", *TKDE* 11(1), pp. 36-44, 1999. (also appear as *TimeCenter Tech Report 17*)

[KS91] C. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data", *Proc. of SIGMOD*, pp. 138-147, 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.

[KT00] G. Kollios and V.J. Tsotras, "Hashing Methods for Temporal Data", *to appear in TKDE*, 2001.

[KTF98] A. Kumar, V.J. Tsotras and C. Faloutsos, "Designing Access Methods for Bitemporal Databases", *TKDE* 10(1), pp. 1-20, 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, pp. 329-355, 1993.

[LOT94] H. Lu, B. Ooi and K. Tan, "On Spatially Partitioned Temporal Join", *Proc. of VLDB*, pp. 546-557, 1994.

[LR96] M-L. Lo and C.V. Ravishankar, "Spatial Hash-Joins", *Proc. of SIGMOD*, pp. 247-258, 1996.

[LS89] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", *Proc. of SIGMOD*, pp. 315-324, 1989.

[OS95] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *TKDE* 7(4), pp. 513-532, 1995.

[PD96] J.M. Patel and D.J. DeWitt, "Partition Based Spatial-Merge Join", *Proc. of SIGMOD*, pp. 259-270, 1996.

[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)*, pp. 427-432, 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 Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pp. 252-261, 1996.

[SS88] A. Segev and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment", *Proc. of Int. Conf. on Statistical and Scientific Database Management (SSDBM)*, pp. 39-61, 1988.

[SSJ94] M. Soo, R. Snodgrass and C. Jensen, "Efficient Evaluation of the Valid-Time Natural Join", *Proc. of ICDE*, pp. 282-292, 1994.

[ST99] B. Salzberg and V.J. Tsotras, "Comparison of Access Methods for Time-Evolving Data", *Computing Surveys* 31(2), pp. 158-221, 1999.

[VV97] P. Varman and R. Verma, "An Efficient Multiversion Access Structure", *TKDE* 9(3), pp. 391-409, 1997.

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

[ZTS00] D. Zhang, V.J. Tsotras and B. Seeger, "A Comparison of Indexed Temporal Joins", *Tech Report, UCR-CS-00-03, CS Dept., UC Riverside*, URL: http://www.cs.ucr.edu/research/tech_reports/, 2000.

A Appendix: An Overview of the MVBT

The MVBT ([BGO+96]) is a graph structure that maintains the evolution of a B+-tree over time. It has multiple roots, each root effectively accessing the B+-tree valid at subsequent time intervals. An additional data structure maintains the roots and their associated time intervals. The MVBT partitions the key-time space into rectangles where each rectangle is associated with exactly one data page. A data record x is stored in all the data pages whose rectangles contain $x.key$ and intersect $x.interval$. The page rectangles are created recursively. As records are inserted into a certain MVBT page this page may overflow. Then, the currently alive data records of the page (records whose end time is *now*) are copied to another page. The actual copying procedure is based on the number of alive records in the overflowed page. A *time-split* simply copies all alive records into a new page (figure 10a). If many alive records exist, the time-split is followed by a *key-split* that distributes them into two new pages according to the median of their key attribute (figure 10b).

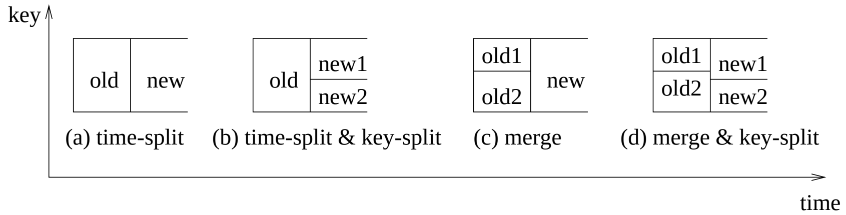


Figure 10: Structural changes in a MVBT

An important feature of the MVBT is that it guarantees a minimum key density for every page. Let i be the interval of page A; for any time instant t in i, there are at least d records in A whose intervals contain t. Here d is a portion of the page capacity. To achieve this, the MVBT uses yet another structural change: *merge*. If after a deletion update the number of alive records in a page drops below the threshold d *weak underflow* the alive records from this and a sibling page are copied into a new page (figure 10c). To avoid frequent merge/splits, the MVBT requires that when a new page is created, the number of records in it must be between a lower and a higher bound (*strong condition*). If the result page of a merge operation has too many records (more than the upper bound), a key split is performed immediately (figure 10d).

The MVBT has various attractive features. Despite the record copying it still uses linear space, has logarithmic update and optimally solves the range snapshot query (find all records alive at time t with keys in range r) [BGO+96].