A Comparison of Indexed Temporal Joins

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

Vassilis J. Tsotras*↑
Computer Science Department
University of California, Riverside
Riverside, CA 92521
tsotras@cs.ucr.edu

Bernhard Seeger
Fachbereich Mathematik und Informatik
University of Marburg
Marburg, Germany
seeger@Mathematik.Uni-Marburg.de

Abstract

We examine temporal joins in the presence of indexing schemes. Utilizing an index when processing join queries is especially advantageous if the join predicates involve only a portion of the temporal relations. This is a novel problem since temporal indices have various characteristics that can affect join processing drastically. For example, temporal indices commonly introduce record copies to achieve better clustering of records with long intervals. We first identify such issues and show that naïve approaches do not work. We concentrate on common temporal join queries and examine various index-based algorithms for processing them. Two optimization techniques for indexed joins are proposed, namely, the *balancing condition optimization* and the *virtual height optimization*. While we use the Multiversion B+-tree as the temporal index, our results apply to other efficient tree-based temporal indices. We also compare against using a traditional R^*-tree as the join index and a Spatially Partitioned Join approach. Based on the temporal data characteristics and the type of temporal join query, we identify the best approach to process the join, a useful outcome for a temporal query optimizer.

1 Introduction

Temporal databases capture the evolution of a modeled reality over time by maintaining many data states (past, current or even future) [OS95]. Data records are accompanied by time intervals, each interval representing the time validity for its associated record. While supporting temporal states increases functionality, it also increases the size of the database. It is thus important to maintain an index that can efficiently access temporal data [LS89, KTF98, BGO+96, VV97]. Temporal indices cluster records both on their keys and on their temporal attributes and optimize selection queries of the form: "find all objects with keys in range *r* and whose intervals intersect interval *i*" (also called *range-interval* queries). An efficient temporal index can also speed up temporal joins, especially if the join predicate involves only a portion from a joined relation. This paper assumes that both relations are indexed and examines efficient ways to process temporal joins using these indices. Various temporal access methods have been proposed recently [ST99]. Here we concentrate on the Multiversion B+-tree (MVBT) [BGO+96] since it optimally solves the range-interval query and its code was readily available. However, our techniques apply to other efficient tree-based temporal indices, like the Time-Split B-Tree [LS89] or the Multiversion Access Structure [VV97].

Temporal join query predicates involve both the key and time spaces. Examples include the T-Join (join two records if their intervals intersect) and the TE-Join (join two records if their keys are equal and their intervals intersect) [GS91]. We define a *generic* join which takes as parameters the query time intervals, the query key ranges and two conditions. The first condition specifies whether the keys of joining records should be equal and the second whether their time intervals should intersect. Other general joins can be easily defined (by specifying whether one interval contains or is after the other, etc.). However, for the purposes of this paper we concentrate on key-space equality and time-space intersection as these are among the most commonly used temporal joins.

There are various characteristics of temporal indices that make the problem examined here novel. A major concern in temporal indexing is how to cluster records with long intervals. A common approach is to create copies of such records, effectively splitting a long interval into smaller, more manageable ones. This leads to fast processing for selection queries, but copies can drastically affect join processing. For example, copies of a given record are stored at various parts of the index introducing large overhead in join processing. Moreover, to

---

* ICDE 2001 PC member

↑ This work was partially supported by NSF (115-9907477) and the Department of Defense.

achieve fast updates, temporal indices usually update only the latest copy of a record, if this record is updated in the modeled reality. This can lead to erroneous join results.

Recently, temporal join research has focused on non-indexed joins [GS91, LM93, RF93, SSJ94, RS96, SE96, Zur97]. A straightforward join algorithm assuming no indices is to scan through both relations, select the desired subsets and then join them. If the sizes of the joining relations are large and the join selectivity is high, indexed join algorithms will be more efficient as they avoid scanning the whole relations.

A straightforward index-based approach is to perform on each of the indices a range-interval selection query and then join the query results via some non-indexed algorithm. We consider this approach, however we also propose algorithms that combine the selection and join phases into one. This saves the cost of first storing and then retrieving intermediate results, which is large if many records satisfy the range-interval selection query. In particular we examine breath-first, depth-first and linked-based temporal join algorithms. We introduce two optimization techniques, the *balancing condition optimization* and the *virtual height optimization*. It should be noted that the virtual height optimization applies to any balanced-tree, indexed join algorithm.

Another straightforward approach is to consider the temporal dimension as just another dimension and use approaches proposed for spatial joins [Gun93, BKS93, HJR97, APR+98, APR+00]. Since records are intervals in the two-dimensional key-time space, they can be indexed using any spatial index (R*-tree, etc.). Spatial join algorithms can then directly be applied. Yet another approach is the spatially partitioned temporal join proposed in [LOT94]. An interval is mapped to a point in a two-dimensional space, which has been partitioned spatially. This partitioning identifies the partitions of the two relations that need to be joined. We compare both the spatial join (using an R*-tree as the index) and the spatially partitioned temporal join with the MVBT join algorithms. Our extensive experimentation shows that the MVBT based joins have overall more robust performance than R-tree joins and the spatially partitioned join. Since there are many parameters that can affect a temporal join (the kind of join, the length of the relation intervals, the size of the query rectangles, etc.) we identify the best choices among the MVBT based join algorithms for each case. That is, we identify the algorithms that a temporal query optimizer should definitely implement for efficient join processing.

The rest of this paper is organized as follows. Section 2 provides an overview of the MVBT as well as previous work. Section 3 presents the generic temporal join condition and concentrates on three common join queries. Section 4 discusses the various MVBT based join algorithms and the proposed optimizations. Section 5 shows the experimental results over all compared approaches. Finally, section 6 presents conclusions and future work.

2 Background

2.1 Preliminaries

Each data record in a temporal relation contains a key, a time interval and various attributes that may change over time. We follow the *First Temporal Normal Form (1TNF)* ([SS88]) which specifies that there are no two records in a temporal relation with equal keys and intersecting intervals. The time interval has the form: [*start*, *end*) where *start*, *end* are integers and $1 \leq start < end \leq max\_time$. An interval may reduce to a time point (when $end=start+1$). Similarly, we assume the record keys are positive integers from [1, *max_key*). A key range has the form [*low*, *high*) where $1 \leq low < high \leq max\_key$. For an interval *i*, a data record *x* is called *i-alive* if *x.interval* intersects *i*. Similarly, given a range *r*, a data record *x* is *r-overlapping* if *x.key* belongs to *r*.

An index record differs from a data record in that it contains a key range instead of a single key and it has a pointer pointing to a page. An index record *e* is called *r-overlapping* if *e.range* overlaps *r* for some range *r*. Given an index record *e*, *page(e)* denotes the page this record points to. A page also has a range and an interval. The range of a page is the key range between the smallest and the largest keys are stored in the page. The interval of a page is defined as the time interval between the time the page is created and the time the page is copied (split). The range and interval of a page create the page's rectangle.

We will differentiate between two types of indices. The *primary index* stores the actual data records, thus it controls the physical storage of the relation. The *secondary index* stores data records that contain the indexed attribute(s) and a pointer to the actual data record. The MVBT may be used as either type of index. The benefit of using it as a primary index is that no additional access is required to retrieve the entire data records. Given

that the MVBT performs many record copies, the benefit of using it as a secondary index is space efficiency, since only a small part of the actual data record is copied.

2.2 Review of the MVBT

The MVBT [BGO+96] is a graph structure that maintains the evolution of a B+-tree over time. It has many roots, each responsible for the subsequent part valid during a specific time interval. References to the different roots associated with the corresponding time intervals are kept in an additional data structure called *root**. The MVBT partitions the key-time space into rectangles where each rectangle is associated with exactly one data page. A data record is stored in all the data pages whose key-time rectangle contains the data record's key and intersects the record's interval. The page rectangles are created recursively. As records are inserted into a certain page of a MVBT, the page may overflow. At that time, this page's currently alive data records are copied to another page. The kind of copying is based on the number of alive records in the overflowed page. A *time-split* simply copies all alive records into a new page (Fig 1a). 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 (Fig 1b).

Data records are inserted in the MVBT in increasing time order (transaction-time is assumed [JS99]). When a data record is inserted at *t*, its deletion time is yet unknown and its interval is initiated to *[t, now)*; *now* is a variable representing the ever increasing current time. For implementation purposes, *now* is stored as *max_time*. When later (if ever) this data record is deleted or updated, the *end* time in its interval is updated from *now* to the deletion time.

An important feature of the MVBT is that it guarantees a minimum key density for every page. In particular, for any time *t* in the page's rectangle, the page contains at least *d t*-alive records, where *d* is linear to the page capacity. To achieve this, the MVBT uses yet another structural change: *merge*. If a *weak underflow* occurs after a deletion, i.e. the key density of the page where the deletion takes place drops below the threshold *d*, the alive records in the page and a sibling page are copied into a new page (Fig 1c). 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 bound 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 immediately performed (Fig 1d).

The MVBT optimally solves (in linear space) the range-snapshot query: "find all keys in range *r* that were alive at time *t*". To answer range-interval queries, the structure has to deal with the various copies that may intersect the query interval. The following is a naïve depth-first approach for a range-interval query.

Algorithm *DF$_{naïve}$* (Record *e*, Range *r*, Interval *i*)
1. *N* = Readpage(*e*); // Read the page pointed by record e.
2. for (each record *se* in *N*) do
3. if (RangeIntervalCond(*se*, *r*, *i*)) then
4. if (*N* is an index page) then
5. DF$_{naïve}$ (*se*, *r*, *i*);
6. else
7. output *se*;
8. endif
9. endif
10. endfor

end DF$_{naïve}$;

The algorithm visits record *se* if the following condition holds:

RangeIntervalCond(Record *se*, Range *r*, Interval *i*)
1. return (*se* is *r*-overlapping and *i*-alive);

That is, if *se* intersects the query rectangle.

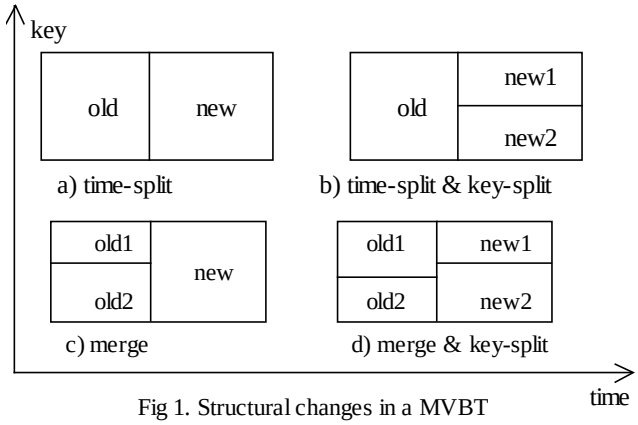


Fig 1. Structural changes in a MVBT

As pointed out in [BS96], the naïve approach does find the correct answer set; however, it suffers from the *duplicate result problem*, which means that 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 happens because many copies of a given record may exist. To avoid duplicates, only one of these copies should be visited. [BS96] provides a depth-first range-interval-query algorithm and a link-based one, both with duplicate

avoidance. The idea of the depth-first approach (***DF$_{ref}$***) is to compute a *reference point* for each record, which is defined as the lower-left intersection of the record and the query rectangle. As illustrated in Fig 2a, where the light-shadowed 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 spot) 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. More formally, DF_{ref} is a slight modification of $DF_{naïve}$ with step 3 being changed to: "if (RangeIntervalCond(*se, r, i*) and ***RefPointCond***(*se, e, r, i*)) then". Here the reference point condition (RefPointCond) means whether the reference point computed for record *se* lies in the page pointed by record *e*:

RefPointCond(Record *se*, IndexRecord *e*, Range *r*, Interval *i*)
1. if *se* is an index record then
2. *keylow* = *se.low*
3. else
4. *keylow* = *se.key*
5. endif
6. (ref_K, ref_T) = (max{*keylow*, *r.low*}, max{*se.start*, *i.start*});
7. return (e.*low* ≤ ref_K < *e.high* and *e.start* ≤ ref_T);

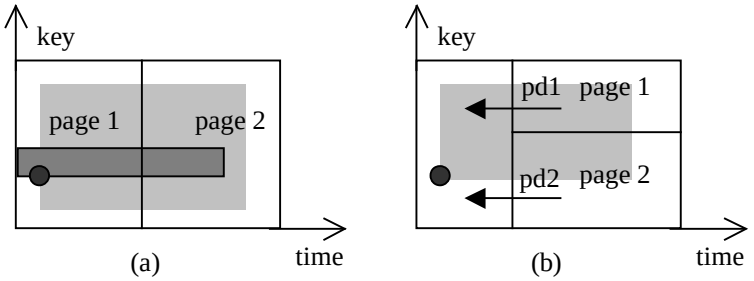


Fig 2. (a) RefPointCond and (b) PredCond

The link-based approach (***Link$_{ref}$***) is based on the following concepts: for two data pages *A* and *B* in a MVBT, if *A* is copied to *B*, *A* is called a *predecessor page* of *B* and *B* is called a *successor page* of *A*, and an index record called *predecessor record* is saved in *B* pointing to *A*. $Link_{ref}$ first finds all data pages that intersect the right border of the query rectangle. It then follows *predecessor records* to find all the other data pages whose rectangles intersect the query rectangle. Finally, for each data page thus found, it reports all the records satisfying RangeIntervalCond and RefPointCond.

Due to the splitting policy of the MVBT, a data page can be the predecessor of at most two other pages. To avoid visiting a predecessor page twice, the algorithm also utilizes a reference point (the lower left intersection of the query rectangle and the predecessor page rectangle). For example, in Fig 2b, a predecessor record *pd1* in page 1 and a predecessor record *pd2* in page 2 point to the same page. Since the reference point (the dark spot) lies in the key range of page 2 and not page 1, only *pd2* is followed. More formally, while examining a data page pointed by record *e*, $Link_{ref}$ follows a predecessor record *pd* stored in page(*e*) if the following condition holds:

PredCond(Record *pd*, Record *e*, Range *r*)
1. return *e.low* ≤ max{*pd.low*, *r.low*};

That is, if the reference point of *pd* record lies in the key range of the page(*e*).

2.3 Temporal Joins

Work on temporal joins has focused on non-indexed algorithms. They are classified similarly as the traditional equijoin algorithms: nested-loop, sort-merge and partition-based. [RF93] assumes the smaller relation fits in memory and proposes seven nested-loop join algorithms to solve the T-Join. Most sort-merge temporal joins assume some order in the input relations and focus on the merge step. [GS91] discusses T-Join and TE-Join when one or two relations are sorted. [LM93] assumes that the relations are sorted on the insertion time of the records and discusses how to merge them in a stream-processing manner. [RS96] discusses 1-, 2- and 3-dimensional intersection joins. In the 1-dimensional case, which is the temporal case, it also assumes the two relations are sorted.

Partition-based algorithms are classified by whether the partitioning technique is *static partitioning*, *dynamic partitioning* or *spatial partitioning*. In static partitioning ([Zur97]), a record is copied to all partitions that intersect its interval. One partition needs to join with only one partition in the other relation. When two partitions are joined, two records do not need to join unless the start time of at least one of them falls in the partition. 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.

In spatial partitioning ([LOT94]), a record's interval *p* is mapped to a point (*p.start*, *p.end- p.start*) in a two-dimensional transformed space. The *x*-axis corresponds to insertion time while the *y*-axis corresponds to the interval length. A partition in one relation is joined with many partitions in the other relation. The paper also gives a join algorithm (the *SpatiallyPartitionedJoin (SPJ)*) that uses the *Time Polygon Index (TPI)* ([SOL94]) to index points in the transformed space. Note that SPJ is an index-based temporal join algorithm.

2.4 Indexed Spatial Joins

A common indexed spatial join technique is *Synchronized-Tree-Traversal (STT)*. Either depth-first or breadth-first can be utilized to traverse both trees synchronously. Initially the pair of root nodes is pushed into a 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 the two records satisfy some condition. [BKS93] presents a depth-first R-tree join while [HJR97] discusses breadth-first R-tree join. Since in the breadth-first case, each level is completely processed before proceeding to the next level, global optimizations are possible. One such optimization sorts the result of one level, or the so-called *intermediate join index (IJI)*, before proceeding to the next level [HJR97]. If memory is large, the breadth-first join is better than the depth-first one [HJR97]. Another work is [Gun93] which proposes join algorithms when *Generalization Trees* are used.

3 The Generic Temporal Join Query

We first propose a generic temporal join condition and then identify three important special cases.

Definition. Given temporal relations *X* and *Y*, key ranges *r1* and *r2*, intervals *i1* and *i2* and boolean variables *key_must_equal* and *interval_must_intersect*, the ***Generic Temporal Join*** query reports all record pairs (*x*, *y*) where $x \in X$ and $y \in Y$ such that: (1) *x* is *r1*-overlapping and *i1*-alive, (2) *y* is *r2*-overlapping and *i2*-alive, (3) *x.key* = *y.key* if *key_must_equal* is true, and, (4) *x.interval* intersects *y.interval* if interval*_must_intersect* is true.

In other words, the range-interval-query results of the two joining relations are joined according to the *key_must_equal* and *interval_must_intersect* conditions. Three important special cases of the generic join are:

(1) *Generic TE-Join (GTEJ)*: find records in the two relations that intersect the same query rectangle and join them if their keys are equal and their intervals intersect.

(2) *Generic T-Join (GTJ)*: the two query rectangles have the same interval; a record in relation one that intersects the first rectangle joins with a record in relation two that intersects the second rectangle if their intervals intersect.

(3) *Generic Equi-Join (GEJ)*: the two query rectangles have the same key range; a record in relation one that intersects the first rectangle joins with a record in relation two that intersects the second rectangle if their keys are equal.

4 MVBT-based Temporal Joins

We first describe how record copies created by the MVBT can affect the correctness of join algorithms and we provide a solution that avoids this problem. This solution applies to both unsynchronized and synchronized index traversal join algorithms. For synchronized traversal, we first concentrate on top-down traversals and present the depth-first and breadth-first join algorithms. Next, we consider the link-based traversals that examine the leaf level of the trees mainly sideways (from right to left).

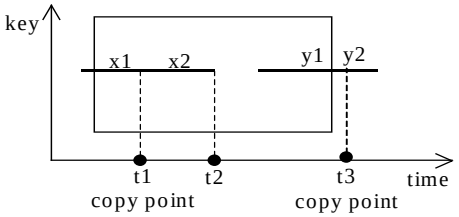


Fig 3. The incorrect deletion time problem. The rectangle is the query rectangle for both relations. Records x1 and x2 are copies and records y1 and y2 are copies.

4.1 The Incorrect Deletion Time Problem

Consider the example in Fig 3. At time *t1*, the page that contains record *x1* is split and thus *x1* is copied to *x2* in a successor page. The deletion time of *x2* is *t2*, while the deletion time of *x1* is *now*, since at *t1*, the deletion time of *x1* was unknown. (The MVBT and the other

temporal access methods do not update any previous copy about the actual deletion time of a record as this does not affect selection queries but it would largely increase update time). Both DF_{ref} and $Link_{ref}$ will report *x1* but not *x2*. Let *y1* be a record in the other relation. Since *x1.end=now*, any join algorithm utilizing the range-interval-query algorithms will assume that *x1.interval* intersects *y1.interval*, which is obviously wrong. We call this problem the *incorrect deletion time problem*. It affects records that have been deleted during the query interval and have copies in that interval. (Note that if a record, e.g. record *y1* in Fig 3, is alive at the *end* time of the query interval, even though it may be deleted some time later, it still provides the correct join result if the deletion time of the record is reported as *now*). To solve the problem, we propose to replace the RefPointCond with:

RightRefTimeCond(Record *se*, Record *e*, Interval *i*)
1. return min{*se.end*, *i.end*}≤ page(*e*).*end*;

This condition means to choose the smaller time between the *end* time of record *se* and the *end* time of the query interval and returns whether this time is no later than the *end* time of the page containing *se*. This condition also helps the range-interval-query algorithms DF_{right} and $Link_{right}$ to avoid duplicates. Among various copies of the same record which intersects the query rectangle, only the rightmost one satisfies this condition. Fig 4 shows the difference between the "right reference time" and the "reference point". The dark rectangle illustrates a record that is inserted in page *1* and is copied to page *2*. The light rectangle is the query rectangle. The reference point for both copies of this record is shown as the black dot. The right reference time is *t2* for the first copy and *t1* for the second copy. Among the various copies of a record, the RefPointCond leads the range-interval-query algorithms to visit the first one while the RightRefTimeCond leads the algorithms to visit the last one. We call the modified range-interval-query algorithms $\boldsymbol{DF_{right}}$ and $\boldsymbol{Link_{right}}$. It can be proved that join algorithms utilizing these modified algorithms will not suffer from the incorrect deletion time problem.

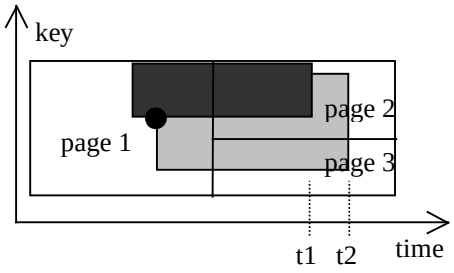


Fig 4. The right reference time versus the reference point

4.2 Unsynchronized Tree Traversal

A straightforward approach to solve the generic temporal join query is to perform range-interval queries on each relation asynchronously and then join the results. In our experiments, $Link_{right}$ is used to perform the range-interval queries since it always outperforms DF_{right}. As for the algorithm to join the range-interval-query results, we experimented with sort-merge and block-nested-loop algorithms.

4.3 Synchronized Depth-first and Breadth-first Approaches

We proceed with top-down Synchronized Tree Traversal join algorithms for the MVBT. The idea is to perform range-interval queries using DF_{right} for the two MVBTs synchronously. This discussion applies to all three joins identified in section 3. The depth-first join algorithm is given below.

Algorithm ***MVBT_DF$_{right}$***(MVBT *mvbt1*, *mvbt2*, Range *r1*, *r2*, Interval *i1*, *i2*, Boolean *key_must_equal*, *interval_must_intersect*)
1. for (every *i1*-alive record *e1* in the root* of *mvbt1*) do
2. for (every *i2*-alive root *e2* in the root* of *mvbt2*) do
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. *N1* = Readpage(*e1*); *N2* = Readpage(*e2*);
10. for (every record *se1* in *N1* and every record *se2* in *N2*) do
11. if (RangeIntervalCond(*se1*, *r1*, *i1*) and RightRefTimeCond(*se1*, *e1*, *i1*) and RangeIntervalCond(*se2*, *r2*, *i2*) and RightRefTimeCond(*se2*, *e2*, *i2*) and JoinIndexCond(*se1*, *se2*, *key_must_equal*, *interval_must_intersect*))
12. then
13. Push(*Stack*, [*se1*, *se2*]);
14. endif
15. endfor
16. else if (only *e1* points to index page) then
17. *N1* = Readpage(*e1*);

```
18.         for ( every record se1 in N1 ) do
19.           if ( RangeIntervalCond(se1, r1, i1) and RightRefTimeCond(se1, e1, i1) and
                   JoinIndexCond(se1, e2, key_must_equal) )
20.           then
21.             Push( Stack, [se1, e2] );
22.           endif
23.         endfor
24.       else if ( only e2 points to index page ) then
25.         // similar; omit
26.       else // both e1 and e2 point to data pages
27.         N1 = Readpage( e1 ); N2 = Readpage(e2);
28.         for ( every record se1 in N1 and every record se2 in N2 ) do
29.           if ( RangeIntervalCond(se1, r1, i1) and RightRefTimeCond(se1, e1, i1) and
                   RangeIntervalCond(se2, r2, i2) and RightRefTimeCond(se2, e2, i2) and
                   JoinDataCond(se1, se2, key_must_equal, interval_must_intersect) )
30.           then
31.             Output( [se1, se2] );
32.           endif
33.         endfor
34.       endif
35.     endwhile
end MVBTJ_DF_right;
```

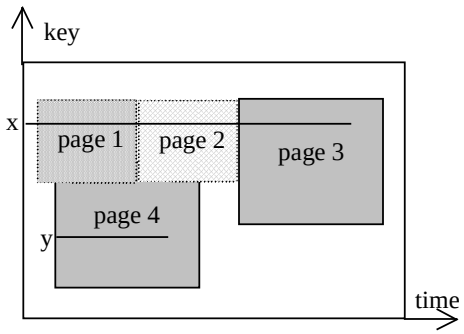


Fig 5. The balancing condition optimization. Records x and y belong to separate relations.

The conditions to join two index records and two data records are:

```
JoinIndexCond(IndexRecord se1, se2, Boolean key_must_equal)
1.  return ( not key_must_equal or se1.range overlaps se2.range ) );

JoinDataCond(DataRecord se1, se2, Boolean key_must_equal, interval_must_intersect))
1.  return ( ( not key_must_equal or se1.key = se2.key ) and
           ( not interval_must_intersect or se1.interval intersects se2.interval ) );
```

The breadth-first join algorithm MVBT_BF$_{right}$ is similar and is omitted (the full version appears in [ZTS00]).

4.3.1 The Balancing Condition Optimization

The reader may have noticed that the JoinIndexCond above does not require that the intervals of joining index records should intersect. The reason is illustrated in Fig 5. Suppose *interval_must_intersect* is true and *key_must_equal* is false. Obviously, record *x* and *y* should join. However, *x* can only be visited while examining page 3 and *y* can only be visited while examining page 4. In order for (*x, y*) to be joined, page 3 and page 4 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. Below we give an optimization technique that leads to a far more efficient solution where pairs of pages are only pushed on the stack when the page rectangles intersect. Here we assume that *interval_must_intersect* is always true (otherwise we have to join records whose intervals do not intersect and the discussion does not apply).

The idea of the *balancing condition optimization (BCO)* technique is to balance between the following two conditions: (1) requiring that the RightRefTimeCond holds for both joining records (this condition is required in MVBT_DF$_{right}$; e.g. in step 29, in order for records *se1* and *se2* to join, both RightRefTimeCond(*se1, e1, i1*) and RightRefTimeCond(*se2, e2, i2*) should hold); and (2) requiring the intervals of joining index records to intersect (this condition is not required in MVBT_DF$_{right}$). 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 RightRefTimeCond is false (under certain conditions). For example, in Fig 5, the improved join algorithms do not join page 3 with page 4 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 4 are joined and when page 2 and page 4 are joined, since both pages 1 and 2 contain *x* and intersect page 4 (note that the RightRefTimeCond is false for *x* in both cases). To make sure that (*x, y*) is not joined multiple times, we require that the RightRefTimeCond holds for at least one of the joining records (in the example of Fig 5, RightRefTimeCond(*y*, record pointing to page 4, *i2*) is true) and we also require that the *end* time of the page containing *x* (where the RightRefTimeCond is false) should be no less than that of page 4. Since page 2 satisfies this but page 1 does not, the algorithm joins (*x, y*) only when page 2 and page 4 are joined.

4.3.2 The Virtual Height Optimization

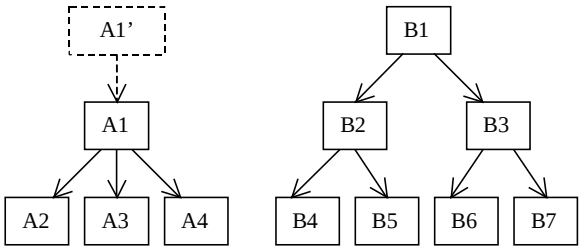


Fig 6. The virtual height optimization

The v*irtual height optimization (VHO)* technique can be utilized to improve the performance while joining any two balanced trees. The idea is illustrated in Fig 6. The node with a dashed rectangle (A1') 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 <*A1, B1*>. At the middle level, we need to join: <*A2, B2*>, <*A3, B2*>, <*A4, B2*>,<*A2, B3*>, <*A3, B3*> ,<*A4, B3*>. Finally, at the leaf level, we join every leaf node from tree A with every leaf node in B. With the VHO, what needs to be joined at the top level and at the leaf level remains unchanged. However, at the middle level only <*A1, B2*> and <*A1, B3*> are joined.

Clearly, the bigger the difference in the height of the two trees, the more significant the benefit of this optimization should be.

4.4 Synchronized Link-based Approach

The link-based join approach is specific to the MVBT. The idea is to perform range-interval queries on both trees using $\text{Link}_{\text{right}}$ synchronously. It is necessary however to separate the discussions for the three joins as each case has a different algorithm. Both the BCO and the VHO can be applied. Due to space limitations, in the following we only discuss the main ideas of the algorithms. For details of the algorithms, refer to [ZTS00].

4.4.1 Link-based GTJ

To find all the pairs of records whose intervals intersect, the proposed algorithm first identifies all pairs of data pages whose intervals intersect and then joins them. To join two data pages is trivial. Hence we focus on how to locate the pairs of data pages whose intervals intersect. The algorithm uses two steps: (1) it finds pairs of data pages whose intervals intersect and each of which intersects the right border of the query rectangle; (2) it follows predecessor records synchronously to find other pairs of data pages whose intervals intersect. The first step is straightforward. To follow predecessor records synchronously, the following procedure is utilized: while a pair of data pages is 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. As done in $\text{Link}_{\text{right}}$, in each of the MVBTs, the PredCond is used to avoid duplicates (i.e. to avoid following two predecessor records pointing to the same page).

4.4.2 Link-based GTEJ

To solve the GTEJ query, we were originally tempted to use the GTJ algorithm, with the slight modification that two data pages join if their page rectangles intersect. Unfortunately, this straightforward approach is wrong, as illustrated in Fig 7. Record *e1* points to a data page (the shadowed page) in one MVBT. Records *pd2*, *e2* and *e2'* point to data pages in the other MVBT. The query rectangle is the key-time space. For records that intersect the right border of the query rectangle, *e1* and *e2* are joined but *e1* and *e2'* are not. If we require that the PredCond is true in order for a predecessor record to be followed (as we do in the link-based GTJ algorithm), there will be no way page(e1) and page(pd2) can be joined, since PredCond(*pd2, e2, r*) is false. To solve the problem, we need to 'release' the PredCond, that is, we still follow the predecessor record, even if the PredCond is false. Below we discuss the places where the PredCond needs to be released. Assume that the current joining pair is (*e1, e2*). We differentiate between two cases.

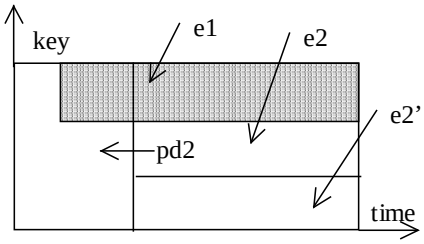


Fig 7. Using the GTJ algorithm to solve the GTEJ query does not work.

Case 1: the *start* times of *e1* and *e2* are different. Let *e1.start<e2.start*. We need to examine whether to join *e1* with each predecessor record *pd2* in page(*e2*) where *pd2* intersects the query rectangle and *e1*. Fig 7 illustrates the scenario. If *e1.low≥e2.low*, the PredCond on *pd2* needs to be released, since *e1.low≥e2.low* means that *e1*

does not intersect the other page (pointed by *e2'*) which contains a predecessor record pointing to page(*pd2*). If the PredCond on *pd2* is still required, page(*e1*) and page(*pd2*) will fail to be joined.

Case 2: the start time of *e1* and *e2* are equal. We need to examine whether to join each predecessor record *pd1* in page(*e1*) with each predecessor record *pd2* in page(*e2*) if they intersect the query rectangle and with each other. We differentiate two subcases.

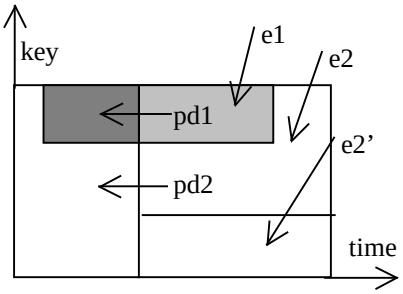


Fig 8. Illustration of subcase 2.1.

Subcase 2.1: The PrefCond is true only for one of the two predecessor records. Let PredCond(*pd1, e1, r*) be true and PredCond(*pd2, e2, r*) be false. Fig 8 illustrates this scenario. Again, if *e1.low*≥*e2.low*, the PredCond on *pd2* needs to be released.

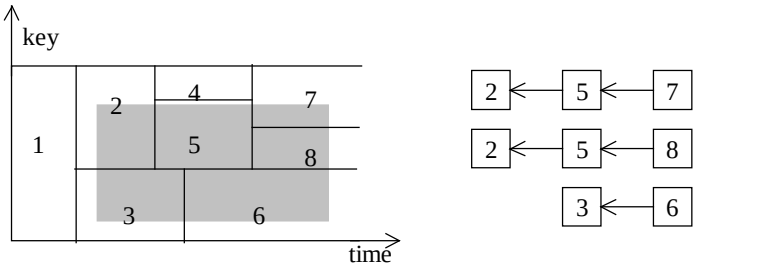
Subcase 2.2: The PredCond is false for both predecessor records. Analysis shows that in this subcase, there is no need to release the PredCond ([ZTS00]).

4.4.3 Link-based GEJ

The link-based GEJ algorithm differs from the link-based GTJ and GTEJ mainly in that two data pages join even if their intervals do not intersect. Consider two linked lists, one from each MVBT, where the *head* node is some data page intersecting with the right border of the query rectangle and the *next* pointer is some predecessor record (Fig 9 shows the linked lists corresponding to the data pages in a 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. The idea is to join every node in linked list *A* with the whole linked list *B*. The algorithm is as follows.

Algorithm *LinkedListJoin* (Linked-list *A*, *B*)
1. push(*Stack*, [*A.head*, *B.head*]);
2. while(not IsEmpty(*Stack*)) do
3. [*a*, *b*] = Pop(*Stack*);
4. Join *a* with *b*;
5. Push(*Stack*, [*a*, *b.next*]) if (*b.next* exists);
6. Push(*Stack*, [*a.next*, *b*]) if (*a.next* exists and *b* = *B.head*);
7. endwhile

end LinkedListJoin;



(a) Layout of data pages in a MVBT (b) The corresponding linked lists

Fig 9. The set of data pages intersecting the query rectangle (the shadowed area) form a set of linked lists, where nodes are linked via predecessor records.

By following this approach, the link-based GEJ algorithm is able to find every pair of data pages that should be joined. The issue remaining is how to avoid duplicates (as illustrated in Fig 9b, two linked lists corresponding to one MVBT may contain the same node). As we do in the link-based GTEJ algorithm, we use the PredCond to avoid duplicates, but the condition sometimes needs to be released in order not to lose join results. Below we discuss the places where the PredCond needs to be released.

Step 5 and step 6 in LinkedListJoin are the only steps that correspond to reading some predecessor records. Assume that the join algorithm is joining (*e1, e2*) where *e1* and *e2* point to data pages in the two MVBTs. Step 5 corresponds to pushing (*e1, pd2*) into the stack, where *pd2* is a predecessor record in page(*e2*), *pd2* intersects the query rectangle and *e1.range* overlaps *pd2.range*. If *e1.low*≥*e2.low*, PredCond(*pd2, e2, r*) needs to be released, since otherwise the join algorithm will fail to join page(*e1*) with page(*pd2*). Step 6 corresponds to pushing (*pd1, e2*) into the stack, where *pd1* is a predecessor record in page(*e1*), *pd1* intersects the query rectangle, *e2* intersects the right border of the query rectangle and *pd1.range* overlaps *e2.range*. If *e1.low*≥*e2.low*, PredCond(*pd1, e1, r*) needs to be released.

5 Performance Analysis

We compared the performance of the unsynchronized and synchronized MVBT-based join algorithms with the R*-tree-based join and the Spatially Partitioned Join (SPJ) algorithms for the three generic join queries of section 3 (namely the GTEJ, GEJ and GTJ). The notations used in the performance graphs are as follows:

Notation:	Meaning:		Notation:	Meaning:
mvbt_df	Synchronized, depth-first traversal using MVBT		*mvbt_sm*	Unsynchronized, sort-merge traversal using MVBT
mvbt_bf	Synchronized, breadth-first traversal using MVBT		*r*_df*	Synchronized, depth-first traversal using R*-tree
mvbt_link	Synchronized, link-based traversal using MVBT		*r*_bf*	Synchronized, breadth-first traversal using R*-tree
mvbt_nl	Unsynchronized, nested-loop traversal using MVBT		*SPJ*	Spatially Partitioned Join

Since SPJ does not involve the key space it applies only for the GTJ query. We examined the performance while using the indices as primary and as secondary. For the GTEJ and GEJ queries the secondary indices perform universally better than their primary counterparts, while for the GTJ query the primary indices prevail. For the same relation, the primary index is larger than the index part of the secondary index, so when joining the index parts, using the primary index takes longer. On the other hand, the secondary index requires a second step to retrieve the actual records. For the GTEJ and GEJ queries the joining records must have equal keys thus the join selectivity is high. Hence, the time needed to retrieve the actual data records from secondary indices is relatively small and the overall performance of secondary indices is better. For the GTJ query, the join selectivity is low (more records join since no key predicate is present) and the secondary indices have high retrieval cost. We thereby report the performance of only the secondary indices for the GTEJ and GEJ and the primary indices for the GTJ query (the complete set of results appear in [ZTS00]). In the graphs we use subscript 1 to denote a primary index (mvbt1_df) and subscript 2 for a secondary index (r*2_bf).

5.1 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. The memory size is 512 MB. 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. Following the practice in [APR+00], we measure the CPU cost by adding the amounts of time spent in *user* and *system* mode as returned by the *getrusage* system call. We assume all disk I/Os are random, except those accessing the actual data records in a secondary index. A random access takes 10ms on average. Actual data records in a secondary index are stored sequentially and a sequential access takes 1/30 of the time of a random access.

The buffering schemes we use in the implementation of the algorithms are as follows. For the MVBT joins, we use a LRU buffer for each MVBT. 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 mvbt_bf and r*_bf, we use 1/6 of the memory buffer for storing and sorting the intermediate join results. For SPJ, we also use LRU buffering.

With the exception of the dataset used in section 5.2, all datasets are first created using the TimeIT software [KS98] and then transformed to add record keys. 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, 1 million). A dataset contains 50,000 unique keys where each key has an average of 10 intervals (Note that each such interval corresponds to a distinct record lifetime since a deleted record can be later reinserted by the application. This is unrelated to the copies the MVBT makes per record). We define three kinds of intervals: *short* (length about 1/1000 of the time space), *medium* (1/100 of the time space) and *long* (1/10 of the time space). We use the following collection of datasets:

Dataset:	Description:		Dataset:	Description:
uni-LM	keys uniformly distributed; 25% long and 75% medium intervals		*normal_M*	keys normally distributed; 100% medium intervals

uni-M	keys uniformly distributed; 100% medium intervals		*nexp_M*	keys negative-exponentially distributed; 100% medium intervals
uni_SM	keys uniformly distributed; 25% medium and 75% short intervals		*uni_LM-sk*	keys uniformly distributed; 25% long and 75% medium intervals; key space is [1, 100k)
uni_S	keys uniformly distributed; 100% short intervals		*uni-M-sk*	keys uniformly distributed; 100% medium intervals; key space is [1, 100k)

Due to space limitations we report results involving the following dataset combinations:

Dataset 1	Dataset 2	Purpose
uni-LM	uni-M	Join datasets with mainly "large" intervals.
uni-S	uni-SM	Join datasets with mainly "short" intervals.
normal-M	uni-M	Study the effect of normally distributed keys.
nexp-M	uni-M	Study the effect of negative-exponentially distributed keys.
uni-LM-sk	uni-M-sk	Study the effect of a smaller key space.

For simplicity, we assume that the query rectangles are the same for both relations. Each experiment reports the average response over six 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, we use the following default parameters:

parameter	default value		parameter	default value
page size	4KB		memory buffer size	1MB
R/I ratio	1 (square query)		QRS	10%

5.2 Improvement due to the VHO

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. Fig 10 shows the results using the VHO on the MVBT, for a GTEJ query that self-joins the above dataset (using QRS=100%). The depth-first (df) and breadth-first (bf) approaches are clearly improved for both primary (1_) and secondary (2_) indexing. The link-based algorithm has virtually no improvement. This is expected since the link-based algorithm focuses on data pages while the VHO helps only in the intermediate levels.

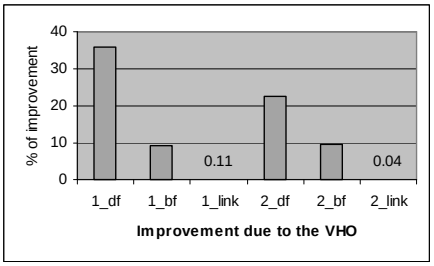


Fig 10

5.3 Improvement due to the BCO

The improvement due to the BCO is drastic, especially when the R/I ratio is small. Fig 11 shows the results of a GTEJ query between the uni-LM and uni-M datasets while varying the R/I ratio. 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 very many unnecessary joins of pages. The improvement of the link-based join algorithm is still substantial but smaller because it

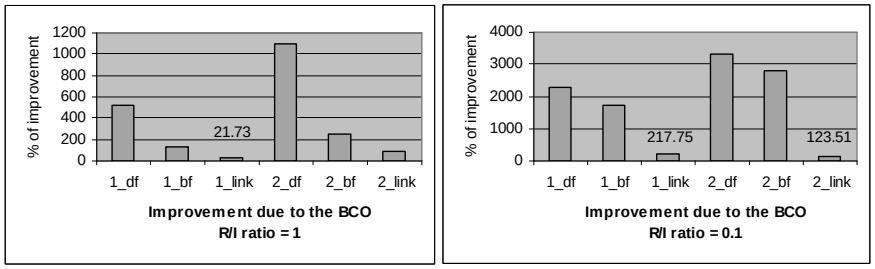


Fig 11. Improvement due to the BCO

examines only a few index pages. In the following experiments both the VHO and the BCO have been already applied.

5.4 The GTEJ query

We first examine the query performance for various values of the R/I ratio. The results for joining the uni-LM and uni-M datasets appear in Fig 12. Among the unsynchronized MVBT methods we report only the sort-merge approach since the nested-loop was clearly worse. We observe that the three synchronized MVBT-based traversal algorithms perform better than the unsynchronized sort-merge MVBT algorithm and the R*-tree-based algorithm. The sort-merge algorithm does not perform well because there are many records satisfying the range-interval query and thus it is expensive to maintain them (store to disk, perform external sort and then read in). The R*-tree based algorithms do not perform well because there are many overlapping records among sibling nodes in the tree. This is to be expected, because R*-trees are affected by the interval overlapping on the time dimension.

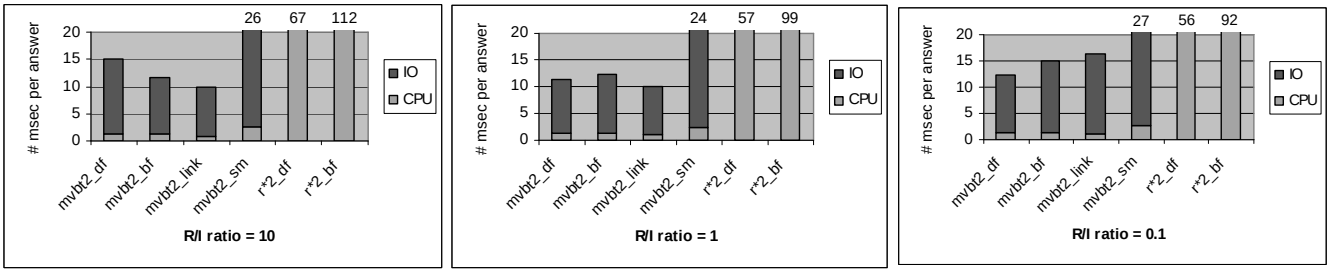


Fig 12. GTEJ, varying R/I ratio

Among the three synchronized MVBT traversal algorithms, for large R/I ratio mvbt2_link performs the fastest while for small R/I ratio mvbt2_df is the best. This result is quite interesting. We know that both the depth-first and the link-based approaches find pairs of intersecting data pages and join them. In other words, they read the same number of data pages. One may expect that the link-based algorithm should always have better performance since it reads less index pages. However, when the R/I ratio is small the depth-first algorithm utilizes the memory buffer better. Small R/I ratio implies that the query rectangle covers much of the time space and little of the key space. Hence there are more time splits and less key splits among the data pages touched by the query rectangle. An example is shown in Fig 13. Suppose the numbered rectangles are the data pages in one MVBT that intersect the query rectangle (clearly, this query rectangle covers more time space than key space). Let the shadowed rectangle *S* be the last data page in the other MVBT that is intersects the same query rectangle. The link-based algorithm starts by pushing (*7, S*) and (*8, S*) into the stack. Then it pops (*8, S*) from the stack and joins page *8* with page *S*. After that, it joins the predecessors of page *8* with the predecessors of page *S*. This continues until the left border of the query rectangle is reached. When the algorithm eventually joins page 7 with page *S*, since the predecessor record is long, chances are that page *S* is already switched out of memory and needs to be read in again. The depth-first algorithm tends to finish joining all the data pages within a time interval before it proceeds to the next time interval, and thus it utilizes the memory buffer better in this case.

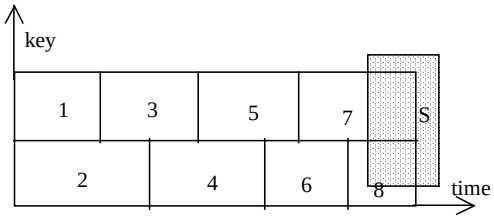


Fig 13. Illustration why mvbt2_df is better than mvbt2_Link when the R/I ratio is small or when joining short intervals.

When the pagesize is large (Fig 14), all the synchronized tree traversal algorithms (including both MVBT-based and R*-tree-based ones) perform a little worse, while the unsynchronized sort-merge algorithms improve. The effect of a large page size on the join performance is two-fold. On the one hand, it tends to improve the join performance since each I/O reads in more data and thus it takes less I/Os to read in the same amount of data. On the other hand, it tends to degrade the join performance since there are less number of pages of the available

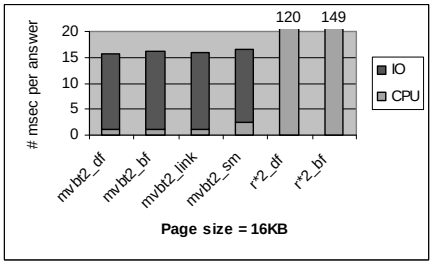


Fig 14. Large page size

memory buffer (assuming fixed buffer size) and thus a page might be needed to be read more times. The reason why the synchronized join algorithms perform worse is that a page from one index may need to be read more than once, since it may join with many pages in the other index. The reason why the unsynchronized algorithms perform better is that a page is read only once from the indices.

We also experimented with joins involving negative exponentially distributed keys. A nexp-M dataset was joined with a uni-M dataset. The results appear in Fig 15. Fig 16 shows the result when joining datasets with smaller key space (the uni-LM-sk and uni-M-sk datasets were used). In both figures the comparative behavior of all methods remains unchanged.

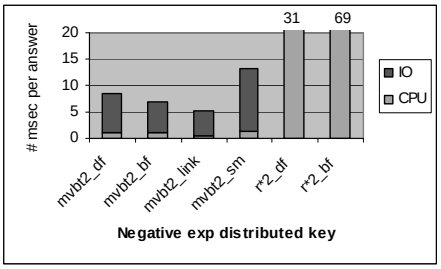


Fig 15

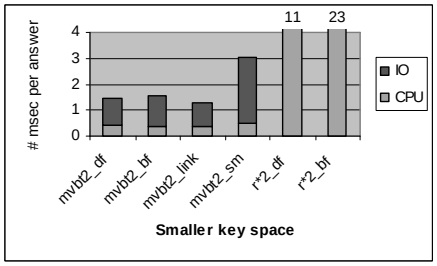


Fig 16

We also checked other QRS and the effect of normally distributed keys and observed similar behavior [ZTS00].

When joining short intervals (using datasets uni-S and uni-SM), the depth-first algorithm is the best (Fig 17). The reason is the same as the case when the R/I ratio is small, since having short intervals is similar to having a query rectangle with a small R/I ratio.

Last, we examine the join performance with varying buffer size (Fig 18). Again, the synchronized MVBT-based join algorithms perform better.

Clearly, in all the above experiments the breadth-first R*-tree join performs consistently worse than its depth-first counterpart. This was observed in the rest of our experiments. Hence we omit it from the remaining graphs.

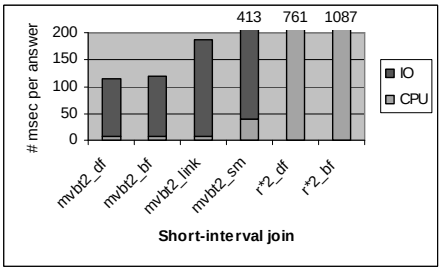


Fig 17

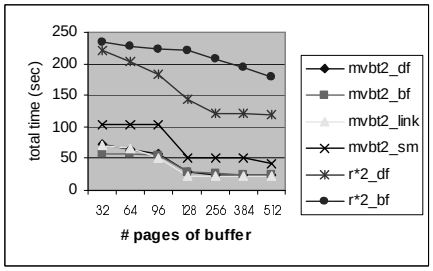


Fig 18

5.5 The GEJ Query

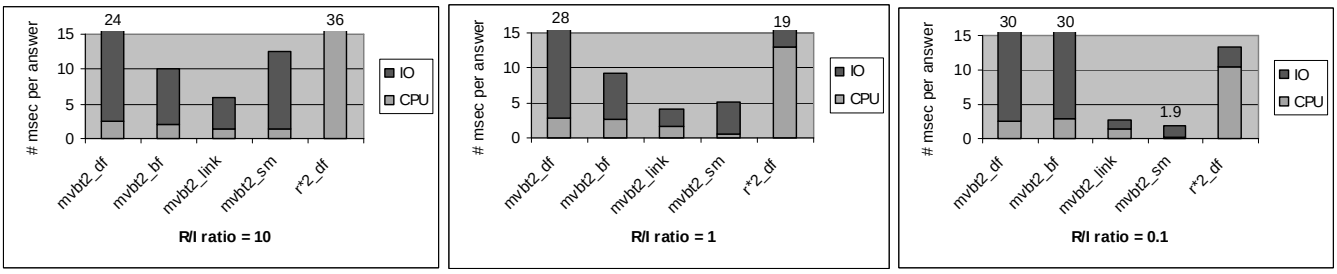


Fig 19. GEJ, varying R/I ratio

For the GEJ query, the link-based algorithm is almost always the best. The reason is that the depth-first and the breadth-first algorithms have to join too many index pages, since two pages join as long as their key ranges overlap and regardless of whether their intervals intersect or not. If an index pages joins with many index pages in the other relation, it is likely that the page is read many times from disk. The link-based algorithm is better because it focuses on the data pages. Fig 19 shows the results where the uni-LM and uni-M datasets were joined. Again, the performance of the unsynchronized nested-loop approach is not reported since it was much worse than its sort-merge counterpart.

Interestingly, we also observe that when the R/I ratio is small, the unsynchronized sort-merge algorithm (mvbt2_sm) becomes a competitor. The reason is that mvbt2_sm does not need to read a page from the indices more than once, while all the synchronized algorithms do (note that although the link-based algorithm avoids the extensive join of index pages, it does not avoid the extensive join of data pages). 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.

We experimented with joins involving negative exponentially distributed keys and joins with smaller key space. Again we noticed that the link-based algorithm was the best performer except when the R/I ratio is small, in which case the sort-merge algorithm was a close competitor. We also observed similar behavior when joining short intervals (using datasets uni-S and uni-SM) [ZTS00].

5.6 The GTJ Query

As mentioned earlier, for the GTJ query the primary indices are faster than their secondary counterparts. Here, we also include the performance of MVBT-based nested-loop join (mvbt_nl) and of SPJ.

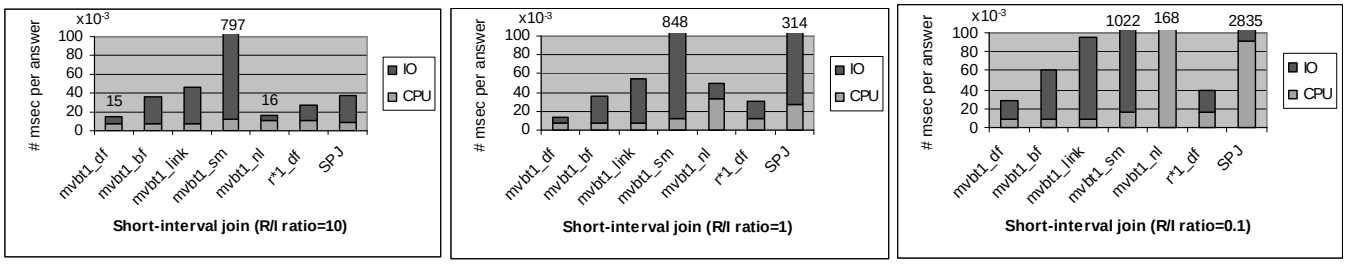


Fig 20. GTJ, joining short intervals, varying R/I ratio

We first examine joining short intervals (Fig 20). The synchronized depth-first algorithm (mvbt_df) is the fastest. The reason is the same as for the short-interval GTEJ. For a large R/I ratio, the nested-loop algorithm becomes competitive, too. This is because for a large R/I ratio, basically every record (in one range-interval-query result) joins with every record (in the other). Clearly, when the join selectivity is very low, the block nested-loop algorithms perform well and the sort-merge algorithm does not (this also explains why the unsynchronized sort-merge algorithm does not perform well for the GTJ query). When joining long intervals, the join selectivity is very low and thus the unsynchronized nested-loop algorithm prevails (Fig 21).

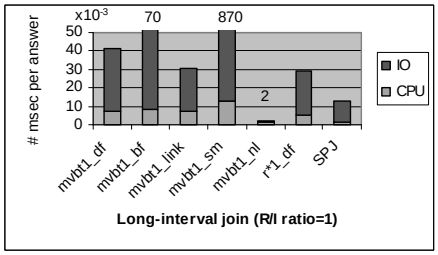


Fig 21. GTJ of long intervals

The SPJ does not perform well since a partition in one relation needs to be joined with many partitions in the other relation. Also, the performance of SPJ worsens as the R/I ratio reduces. This is because the query rectangle covers fewer key space while the SPJ joins the whole key space. Even when the whole key space is covered by the query rectangle (e.g. when R/I ratio=10 and QRS=10%, as shown in Fig 20a), SPJ is still not as efficient as the depth-first MVBT algorithm and the nested-loop algorithm.

We also examined the join performance for other query rectangle sizes and observed similar behavior [ZTS00].

6 Conclusions & Future Work

We studied the problem of efficiently processing temporal joins, when a preexisting index is available on both relations. Many serious problems arise because of temporal index characteristics, like the introduction of duplicated records. Unfortunately, techniques known from selection queries are not applicable to solve these problems for temporal join queries. We identified the problems and provided efficient solutions. While we have concentrated on using the MVBT, our findings apply to other temporal indices as well. We also presented two optimization techniques (BCO and VHO). Both techniques improve the MVBT algorithms, especially the BCO. We compared the MVBT-based joins with other approaches namely, R-tree based joins and a spatially

partitioned join. We experimented with various datasets, using primary and secondary indices and for three kinds of temporal joins (GTEJ, GEJ and GTJ). Our findings are summarized below:

(1) For joins requiring the keys of joining records to be equal (GTEJ and GEJ), the secondary indices have better performance than their primary counterparts. Primary indexing is better for plain interval-based joins (GTJ).

(2) The performance of the MVBT-based join algorithms is overall more robust than the R*-tree based join algorithms and the spatially partitioned join for all join queries examined.

(3) For the GTEJ query, the MVBT link-based algorithm has overall the best performance. When joining relations with mainly short intervals or when the R/I ratio is small, the MVBT depth-first algorithm is a good competitor.

(4) For the GEJ query, the MVBT link-based algorithm has again the best overall performance. However, when joining relations with mainly short intervals or when the R/I ratio is small, the MVBT sort-merge algorithm prevails.

(5) For the GTJ query, the MVBT nested-loop algorithm is the best, except when joining mainly short intervals, in which case the MVBT depth-first algorithm is the best.

This paper shows that while the proposed MVBT-based join algorithms are faster than the spatial R*-tree joins and the SPJ, there is no method among the MVBT-based ones that is universally the best. This is to be expected since the join parameters can vary drastically by interval length, QRS, R/I ratio and the type of join (GTJ, GTEJ, GEJ), etc. However a temporal query optimizer should definitely include the link-based MVBT join algorithm, since it consistently has the best or very good performance. Depending on the application, the optimizer can also be enhanced with the depth-first, nested-loop and sort-merge MVBT join algorithms. The choice will be based on the temporal join parameters. We are currently examining formal cost models that will enable a temporal query optimizer to choose automatically the best join algorithm.

References

[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.

[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.

[BGO+96] B. Becker, S. Gschwind, T. Ohler and B. Seeger, "An Asymptotically Optimal Multiversion B-Tree", VLDB Journal, vol. 5, No. 4, pp. 264-275, 1996.

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

[BS96] J. 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.

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

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

[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, Vol. 11, No.1, pp: 36-44, 1999.

[KS98] N. Kline and M. Soo, “Time-IT, the Time-Integrated Testbed”, URL: ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz, August 1998.

[KTF98] A. Kumar, V. Tsotras and C. Faloutsos, "Designing Access Methods for bitemporal Databases", TKDE, vol. 10, No. 1, pp. 1-21, 1998.

[LM93] T. Leung and R. Muntz, “Stream Processing: Temporal Query Processing and Optimization”, in A. Tansel, etc. (editors), Temporal Databases: Theory, Design, and Implementation, 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.

[LS89] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", ACM Transactions on Database Systems, pp. 315-324, 1989.

[OS95] G. Ozsoyoglu and R. Snodgrass, “Temporal and Real-Time Databases: A Survey”, TKDE, Vol. 7, No. 4, pp 513-532, 1995.

[RF93] S. Rana and F. Fotouhi, "Efficient Processing of Time-joins in Temporal Data Bases", DASFAA, pp. 427-432, 1993.

[RS96] S. Ramaswamy and T. Suel, "I/O-Efficient Join Algorithms for Temporal, Spatial, and Constraint Databases", Unpublished Bell Labs Tech Report, URL: http://www.bell-labs.com/user/sridhar/ftp/suelrep.ps.gz, pp. 1-11, 1996.

[SE96] D. Son and R. Elmasri, "Efficient Temporal Join Processing using Time Index", SSDBM, pp. 252-261, 1996.

[SOL94] H. Shen, B. Ooi and H. Lu, "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases", Proc. of ICDE, pp. 274-281, 1994.

[SS88] Segev and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment", 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. Tsotras, "A Comparison of Access Methods for Temporal Data", ACM Computing Surveys, Vol. 31, No. 2, 1999.

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

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

[ZTS00] D. Zhang, V. Tsotras and B. Seeger, “A Comparison of Indexed Temporal Joins”, Tech Report, UCR-CS-00-02, CS Department, UC Riverside, 2000.