Cost-Based Predictive Spatiotemporal Join

Wook-Shin Han, *Member*, *IEEE Computer Society*, Jaehwa Kim,
Byung Suk Lee, *Member*, *IEEE Computer Society*, Yufei Tao, Ralf Rantzau, and Volker Markl

Abstract—A *predictive* spatiotemporal join finds all pairs of moving objects satisfying a join condition on *future* time and space. In this paper, we present CoPST, the first and foremost algorithm for such a join using two spatiotemporal indexes. In a predictive spatiotemporal join, the bounding boxes of the outer index are used to perform window searches on the inner index, and these bounding boxes enclose objects with increasing laxity over time. CoPST constructs globally tightened bounding boxes "on the fly" to perform window searches during join processing, thus significantly minimizing overlap and improving the join performance. CoPST adapts gracefully to large-scale databases, by dynamically switching between main-memory buffering and disk-based buffering, through a novel probabilistic cost model. Our extensive experiments validate the cost model and show its accuracy for realistic data sets. We also showcase the superiority of CoPST over algorithms adapted from state-of-the-art spatial join algorithms, by a speedup of up to an order of magnitude.

Index Terms—Spatial databases, temporal databases.

1 Introduction

Moving object database systems managing spatiotemporal objects have been an area of active research and have applications in areas like telematics, location-based services, air traffic control systems, etc. Particularly in a wireless environment, techniques for managing a large number of moving objects effectively and efficiently are becoming increasingly important. Spatiotemporal queries used in a moving object database system can be classified into *historical queries* [1], [2], [3], which handle past information, and *predictive queries* [4], [5], which extrapolate the past information into the future. Each type of queries has its own application areas and pertinent research issues.

In this paper, we consider *predictive* queries with a focus on *spatiotemporal join* operations. A predictive spatiotemporal join is a key query operation in a moving object database system. The formal definition of the predictive spatiotemporal join is given as follows:

Definition 1 [6]. *Given two sets R and S of spatiotemporal objects, a future time stamp t_q, and a distance threshold d, a predictive spatiotemporal join finds all pairs of objects $<o_1, o_2>$ such that $o_1 \in R$, $o_2 \in S$, and the distance between the objects o_1 and o_2 at t_q is shorter than d.*

Fig. 1 shows an example predictive spatiotemporal join. Fig. 1a shows the positions of airplanes (moving objects) at the current time *now*, and Fig. 1b shows the positions of airplane moving objects 5 minutes from now. An example join query is "find all pairs of airplanes that will come closer than 10 miles from each other 5 minutes from now [6]." Note that the time condition in Definition 1 is specified on a time point. This type of the predictive spatiotemporal join is called a predictive time-stamp spatiotemporal join. A predictive spatiotemporal join is called a predictive *time-interval* spatiotemporal join instead if the time condition is specified on a future time interval $[t_1, t_2]$. An example time-interval join query is "find all pairs of airplanes that will come closer than 10 miles from each other between 21:00 and 21:15." We focus on the time-stamp join in this paper.

The objective of this paper is to develop an efficient predictive spatiotemporal join algorithm. To the best of our knowledge, this is the first research done with this objective. We assume that indexes are available on both input files. The index on an input file can be any predictive spatiotemporal tree of which the formats of node entries can be converted to conservative bounding boxes (CBBs). Example trees are STRIPES [7], the B^x-tree [8], the B^{dual}-tree [9], TPR-tree [10], and TPR*-tree [5]. A CBB stores information about the velocities and the bounding boxes of moving objects. This way, a CBB encloses moving objects conservatively (i.e., predicting the largest possible expansion at a future point in time) and hence encloses the objects with increasing laxity over time. Fig. 2 illustrates a predictive spatiotemporal tree R. Fig. 2a shows four CBBs, $R_1 \sim R_4$, which enclose seven moving objects, $o_1 \sim o_7$, at time T_c. Arrows near the CBBs and the objects represent their velocities. Fig. 2b shows the same CBBs expanded by the time $T_c + 1$. We can see that the CBBs at the time $T_c + 1$ do not enclose objects tightly.

An intuitive approach for achieving the objective is to apply the existing index-based *spatial* join algorithms, such as depth-first [11], breadth-first [12], and transformation-view-based [13] algorithms, to predictive spatiotemporal indexes.

- *W.-S. Han and J. Kim are with the Database Laboratory, Department of Computer Engineering, Kyungpook National University, 1370 Sankyuk-dong, Book-gu, Daegu 702-701, Korea. E-mail: wshan@knu.ac.kr, jhkim@www-db.knu.ac.kr.*
- *B.S. Lee is with the Department of Computer Science, University of Vermont, Burlington, VT 05405. E-mail: bslee@cems.uvm.edu.*
- *Y. Tao is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, New Territories, Hong Kong. E-mail: taoyf@cse.cuhk.edu.hk.*
- *R. Rantzau is with the IBM Silicon Valley Laboratory, 555 Bailey Ave., San Jose, CA 95141. E-mail: rrantzau@acm.org.*
- *V. Markl is with the Technische Universitat Berlin, Einsteinufer 17, 10587 Berlin, Germany. E-mail: volker.markl@tu-berlin.de.*

Manuscript received 2 May 2007; revised 11 Apr. 2008; accepted 29 June 2008; published online 22 July 2008.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2007-05-0194.
Digital Object Identifier no. 10.1109/TKDE.2008.159.

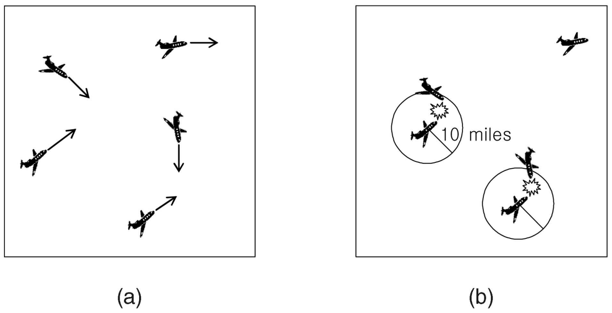


Fig. 1. An example predictive spatiotemporal join. (a) Positions of airplanes at the current time. (b) Positions of airplanes 5 minutes from now.

Here, we can speed up the adapted join algorithms using an optimization, which we call *local tightening*. That is, when traversing down a tree during a join operation, we can tighten the CBB for the current node being accessed, as in Fig. 2c. For example, when we access R_2, we know the exact positions of $o_1 \sim o_3$ at time $T_c + 1$, and thus, we can tighten the CBB for R_2 on the fly. This way, we can reduce the dead space caused by CBBs. However, it is clear that we still have considerable dead space after applying local tightening. Hence, using these adapted algorithms still causes the performance to degrade as time passes.

Our join algorithm resolves this problem by constructing globally tightened bounding boxes "on the fly" during join processing, as illustrated in Fig. 2d. We call our technique *global tightening* in contrast to local tightening. The algorithm works in three steps: 1) sort the leaf entries of the outer index based on a new space-filling curve called the *adaptive cell-based row-major* (ACRM) order (*index-assisted sorting*), 2) construct tightened bounding boxes of the sorted entries at runtime based on the *object density*, and 3) perform *window searches* on the inner file (through the other index) by using the bounding boxes as windows. These three steps can be pipelined unless the sorting is disk based.

The index-assisted sorting utilizes a priority queue to sort the outer index in *one pass*, unless the available memory is insufficient. However, when we deal with a predictive spatiotemporal join, the priority queue may become too large to fit in the available buffer; this is due to an increasing number of overlapping bounding boxes over time.

To tackle this problem, we need to automatically decide how the algorithm chooses between a main-memory resident queue and a disk-based queue. The decision is made based on the cost estimate. For this, we have built *probabilistic cost models* (hence the name *cost-based predictive spatiotemporal join (CoPST)* for our join algorithm). This is a challenging task, due to the effect of page buffering.

Extensive experiments show that in all experiment cases, our join algorithm incurs lower I/O and CPU costs than algorithms adapted from state-of-the-art spatial algorithms. This performance advantage is particularly large when the buffer size is small and/or the data distribution is nonuniform. This advantage is important since objects are typically nonuniform in their distribution and because the buffer size is usually small compared to the input size in a moving object database, especially in a multiuser environment

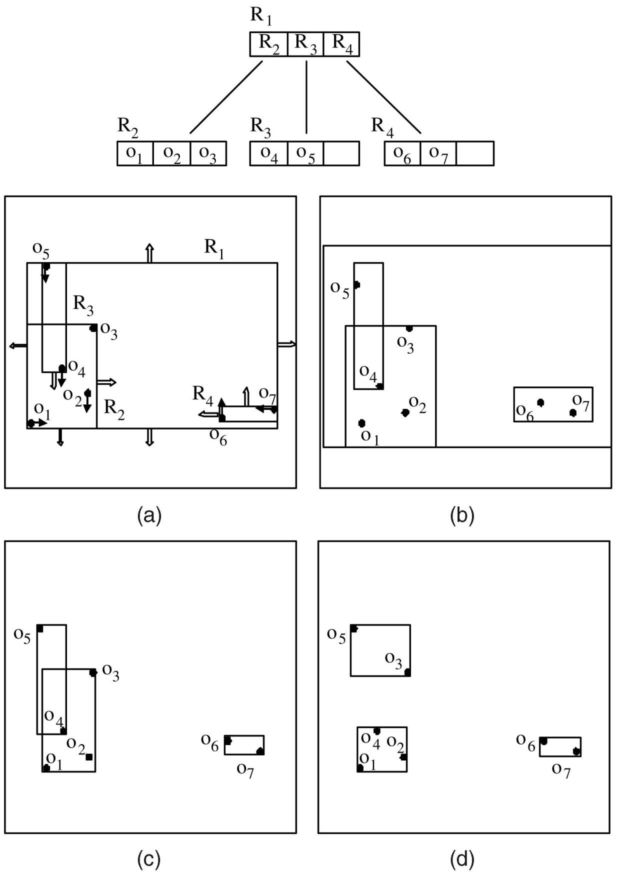


Fig. 2. A motivating example. (a) Tree R at the current time T_c. (b) Tree R at $T_c + 1$. (c) Local tightening at $T_c + 1$. (d) Global tightening at $T_c + 1$.

where many joins are executed together. Even for medium and large buffer sizes, CoPST performs at least two times faster than existing algorithms. This is because by using tightened bounding boxes, CoPST reduces CPU time, as well as I/O time, in contrast to the adapted existing algorithms. With those buffer sizes, CPU time plays a major role in performance. In the extreme case where indexes are resident in the main memory, CoPST still outperforms all the other algorithms by at least a factor of two.

The contributions made throughout this paper are summarized as follows:

1. We propose a novel algorithm, CoPST, which is the first to perform a *predictive* spatiotemporal join. CoPST resolves the loose-bounding-box problem by globally tightening bounding boxes. To maximize buffer utilization, CoPST leverages the ACRM order, which adaptively controls the order of pages to be joined depending on the available buffer size. Note that the ACRM order is "conscious" of the buffer size, unlike the well-known Z order [14] and Hilbert order [15].
2. We develop a *probabilistic cost model* that our join algorithm uses to decide whether to store its intermediate data structure (i.e., a priority queue) in the main memory or disk. By incorporating the effect of page buffering into the cost, our model can accurately estimate the cost for a varying buffer

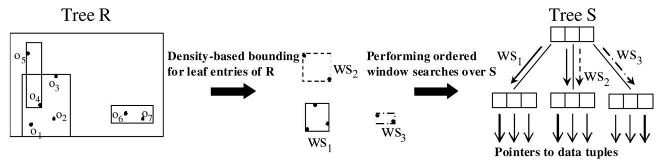

Tree R
o5
o4
o3
o2
o1
o6
o7
Density-based bounding for leaf entries of R
WS2
WS1
WS3
Performing ordered window searches over S
Tree S
WS1
WS2
WS3
Pointers to data tuples

Fig. 3. High-level view of the basic CoPST.

size. Furthermore, by exploiting histograms, our model can support nonuniform data without losing accuracy.

3. Through extensive experiments, we demonstrate the performance advantage of our algorithm over the algorithms adapted from current state-of-the-art spatial join algorithms.

The remainder of this paper is organized as follows: Section 2 describes the basic CoPST algorithm for a memory-resident priority queue, and Section 3 presents a probabilistic cost model of the algorithm. Section 4 enhances the basic algorithm to handle a disk-based priority queue. Section 5 presents the performance evaluations. Section 6 reviews related work, and Section 7 concludes the paper.

2 Basic CoPST Algorithm

2.1 Basic Concepts

CoPST is a variation of the ordered index-based nested loop join. Let us refer to the indexes accessed in the outer and inner loops as R and S, respectively. CoPST first sorts the leaf entries of R using a space filling curve (SFC), based on their spatial locations at the join time. Then, it organizes the sorted leaf entries into windows based on the object density and performs a window search on S for each window. Searching S for the *window* of entries instead of each entry improves the join performance over conventional index-based nested loop joins. Moreover, performing *ordered* window searches improves the buffer utilization by increasing the probability of accessing the same S node that was accessed in the previous window search. Fig. 3 shows a high-level view of the basic CoPST.

CoPST keeps packing entries, say, $o_1, o_2, \ldots$, into a window as long as their density is higher than the threshold density. Thus, CoPST guarantees that the density of each sequential subset starting from o_1 is higher than the threshold density. This approach considers both the spatial proximity of the entries and the buffer utilization. Additionally, the number of entries is limited to the blocking factor in order to prevent a case in which a large number of entries are included in a large window although the resulting density is higher than the threshold.

The threshold density used in our work is the average density of all the leaf nodes of R. The average density can be approximately computed at a low cost by accessing only the nonleaf nodes with the assumption that each leaf node has the same number of entries ($= \frac{\#\ of\ total\ objects}{\#\ of\ leaf\ nodes}$). (This cost is included in the join cost for our performance evaluations in Section 5.) The maximum error of this approximate average density is only 6 percent for the various data distributions we tested.

2.2 Algorithm Description

Algorithm 1 shows the basic CoPST algorithm. The function $Next\_OrderedLeafEntry$ returns the leaf entries in the order specified with SFC. CoPST thus sorts the leaf entries according to the SFC. Then, the leaf entries are bound in that sorted order into windows, and then, a window search is performed on S. Specifically, each leaf entry returned from $Next\_OrderedLeafEntry$ is inserted into $windowSearchSet$ until the object density of the window becomes lower than the threshold D_{thr} or the number of entries exceeds the blocking factor bf (lines 3-6). Here, $windowSearchSet$ is a set of leaf entries used to perform a window search. Then, CoPST performs a window search on S by calling the function $WindowSearch$ with $windowSearchSet$ as input (line 7) and repeats the loop with $windowSearchSet$ initialized to the current leaf entry. The function $WindowSearch$ traverses down the tree S using the bounding box of $windowSearchSet$ and finds the matching pairs (i.e., pairs satisfying the join predicate) between the entries in $windowSearchSet$ and the entries in the leaf nodes of the tree S. We use the plane sweep technique [11] to reduce the computation time of finding the matching pairs. In $WindowSearch$, the "distance" predicate is used as a join predicate, but other join predicates such as "overlap," "contain," and "disjoint" predicates can be used as well. The iteration terminates when there is no more leaf entry returned from $Next\_OrderedLeafEntry$. Finally, the last window search is performed for the set of any entries remaining in $windowSearchSet$ (lines 10-11).

Algorithm 1. $Basic\_CoPST(rootR, rootS, t, SFC, D_{thr}, bf)$

```
Require: $rootR$: root node of tree $R$, $rootS$: root node of tree $S$, $t$: join time stamp, $SFC$: space filling curve, $D_{thr}$: density threshold, $bf$: blocking factor
Ensure: The join result returned by the function $WindowSearch$.
1: initialize $windowSearchSet$ to an empty set $\emptyset$.
2: for each leaf entry $e$ returned from
   $Next\_OrderedLeafEntry(rootR, rootS, t, SFC)$ do
3:   if ($Density(windowSearchSet \cup \{e\}) \geq D_{thr}$) and
     ($Cardinality(windowSearchSet \cup \{e\}) \leq bf$) then
4:     insert $e$ into $windowSearchSet$
5:     continue {skip the rest of this for loop.}
6:   end if
7:   $WindowSearch(windowSearchSet, rootS, t)$
8:   initialize $windowSearchSet$ to the current leaf entry $e$
9: end for
10: if $windowSearchSet$ is not empty then
11:   $WindowSearch(windowSearchSet, rootS, t)$
12: end if
```

Note that we add entries to the current $windowSearchSet$ (the current bounding box used to perform a window search) "on the fly" by ordering them using the SFC (the ACRM order in Section 2.3) and by using the density threshold. Here, the ACRM order ensures that the generated bounding boxes look like squares, and the density threshold ensures that the entries in the current bounding box preserve the spatial proximity. Both the ACRM order and the density threshold enable us to construct tightened bounding boxes "on the fly" during join processing.

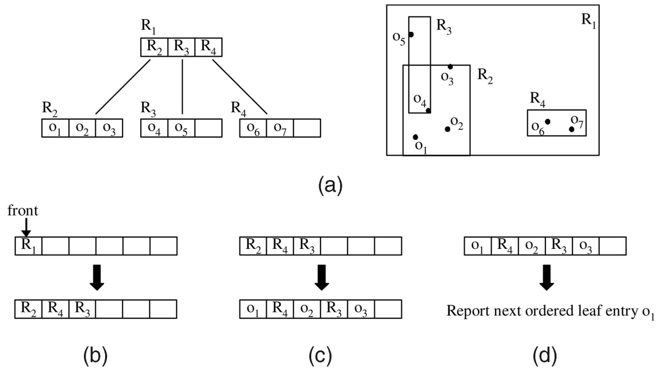


Fig. 4. Example of $Next\_OrderedLeafEntry$. (a) Tree R at $T_c + 1$. (b) When the root node R_1 is popped. (c) When the leaf node R_2 is popped. (d) When the leaf entry o_1 is popped.

Algorithm 2 shows the algorithm of the function $Next\_OrderedLeafEntry$. This function returns the leaf entries of the tree R, one for each call, in the spatial order specified by SFC. It maintains a priority queue for this purpose. The priority queue is initialized with the root node of R (lines 1-5). If the current entry e, removed from the priority queue, is a leaf entry, then the entry is returned (lines 8-9). Otherwise, from all the entries in the node the current entry is pointing to, those whose regions match the region of the root of S are found and inserted into the priority queue (lines 10-15). Here, the function $CBB(node)$ obtains a CBB of $node$; the function $MBB(CBB, t)$ constructs a minimum bounding box (MBB) of CBB at time stamp t. Note that the algorithm eliminates unnecessary page accesses by not inserting entries that do not match the region of the root of S (line 12).

Algorithm 2. $Next\_OrderedLeafEntry(rootR, rootS, t, SFC)$
Require: $rootR$: root node of tree R, $rootS$: root node of tree S, t: a join time stamp, SFC: a space filling curve
Ensure: The next entry of R, matching the region of S, ordered according to SFC.
Variable: $priorityQueue$: a global data structure whose elements are maintained in an order sorted by SFC

```
 1: if $priorityQueue$ has not been initialized then
 2:   $rootEntry.CBB = CBB(rootR)$
 3:   $rootEntry.ref = rootR$
 4:   $Enqueue(priorityQueue, rootEntry, t, SFC)$
 5: end if
 6: while $priorityQueue$ is not empty do
 7:   $e = Dequeue(priorityQueue, t, SFC)$
 8:   if $e$ points to an object then {$e$ is a leaf entry}
 9:     return $e$
10:   else {$e$ points to a node}
11:     for each entry $e'$ in the node pointed to by $e.ref$ do
12:       if $JoinPredicate(MBB(e'.CBB, t),$
          $MBB(CBB(rootS), t))$ then
13:         $Enqueue(priorityQueue, e', t, SFC)$
14:       end if
15:     end for
16:   end if
17: end while
```

Fig. 4 illustrates the algorithm $Next\_OrderedLeafEntry$. Consider the tree R at time stamp $T_c + 1$, shown in Fig. 4a.

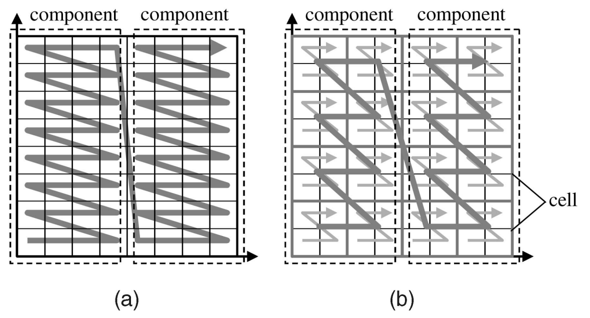


Fig. 5. Example of (a) ARM order and (b) ACRM order.

Here, we use the row-major (RM) order as SFC. We assume that the root node R_1 has already been pushed in the priority queue (lines 1-5). When the root node is popped (i.e., e is not a leaf entry), R_2, R_4, and R_3 are placed in this order in the priority queue in the RM order, as in Fig. 4b. When the leaf node R_2 is popped, we push o_1, o_2, and o_3 into the priority queue (lines 10-16), as in Fig. 4c. When the leaf entry o_1 is popped, we return o_1 (lines 8-9), as in Fig. 4d.

2.3 Space Filling Curve

It is important to use a good space filling curve for SFC, since the node access order is determined by the curve. In this paper, we propose a new space filling curve, the *ACRM* order. This curve has been modified from our earlier work [13], the adaptive RM (ARM) order, which was shown to perform better than Z order and Hilbert order for a spatial join [13]. The ARM order divides the space into k (≥ 1) one-pass regions along the x-axis and follows the RM order within each RM component, and the RM component with a smaller x precedes the one with a larger x. Fig. 5a shows an example of ARM when there are two RM components.

In CoPST, since we sort the leaf *entries* (not leaf nodes) of R, the ARM order tends to generate long horizontal striplike windows. This becomes problematic if there exist a small number of components (in the extreme case, only one component exists), as it can result in many matching entries of S, thereby increasing the computation time. This motivates us to propose a novel SFC, the ACRM order.

The ACRM order consists of $k\,(\geq 1)$ cell-based RM (CRM) components of minimum size (ϵ) along the x-axis. It divides the space into n^2 ($n > 1$) hypothetical grid cells and follows the RM order within each cell and the ARM order among the cells. The space is divided into cells in order to bound leaf entries in the form of a cell. The value of n for bounding all the leaf entries of R with the smallest number of pages satisfies

$$(n-1)^2 < \frac{|R_{leafentry}|}{bf} \leq n^2$$

if we assume that the leaf entries are uniformly distributed, where $|R_{leafentry}|$ denotes the number of leaf entries of R. Fig. 5b shows an example of ACRM when there are two CRM components.

Fig. 6 illustrates the advantage of ACRM over ARM. The horizontal striplike bounding boxes B_1 and B_2 of the ARM match six ($S_7 \sim S_{12}$) and eight ($S_8 \sim S_{11}$ and $S_{14} \sim S_{17}$) leaf nodes of S, respectively. In contrast, the squarelike

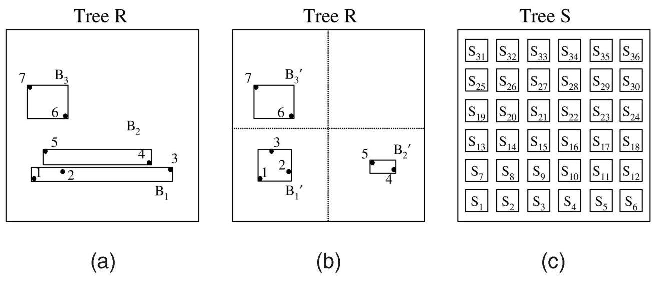


Fig. 6. Horizontal striplike windows in the ARM order. (a) Bounding boxes by ARM order. (b) Bounding boxes by ACRM order. (c) Leaf nodes of Tree S.

bounding boxes B_1' and B_2' of the ACRM match four (S_7, S_8, S_{13}, and S_{14}) and two (S_{11} and S_{12}) nodes of S, respectively.

We have compared the difference in the CoPST performance using the ACRM order and using others such as the Z order and the Hilbert order, as well as the ARM order. The results, obtained using the same data sets used in Section 5, show that with ACRM, the I/O cost has been reduced by up to 1.27, 6.36, and 1.27 times compared with the Z order, the Hilbert order, and the ARM order, respectively. The improvement over the ARM order is larger for the CPU cost. Thus, for instance, the elapsed time $(= \text{CPU cost} + \text{IO cost})$ has been reduced up to 3.02 times compared with the ARM order when it has only one component. This is due to the long horizontal striplike window that causes excessive CPU computation time.

3 Cost Model of the Basic CoPST Algorithm

In this section, we present the cost model of the CoPST algorithm and validate the precision of the model through experiments. The cost model is based on the following assumptions. First, objects are distributed in a two-dimensional space [0, 1] × [0, 1]. Second, the priority queue is stored in a separate main memory space. (We will discuss handling a large priority queue in Section 4.) Third, the LRU buffer page replacement algorithm is used.

In the cost model, we use $Cost_{WQ}(S, q)$ to denote the cost of a window search query q on a predictive spatiotemporal tree S. The specific expression of $Cost_{WQ}(S, q)$ depends on the predictive spatiotemporal tree (e.g., TPR*-tree [5], B^{dual}-tree [9]).

In order to model the random effect of page buffering on the cost, we introduce a random variable $\mathbf{X}$, which denotes the number of node accesses between two repeated accesses of the same node. The probability distribution of $\mathbf{X}$ will be discussed below. It is straightforward to see that given the buffer size of b pages, $P[\mathbf{X} \geq b]$ is the probability that the node does not exist in the buffer when accessed again (i.e., has been replaced by a different page). Here, we assume one disk page per node, without loss of generality. The cost model of CoPST is presented in the following theorem:

Theorem 1. *In CoPST, the total cost of performing the join algorithm Basic_CoPST using two index trees R and S given the buffer of size b, $Cost_{Basic}(R, S, b)$, is expressed as follows:*

$$Cost_{Basic}(R,S,b) = |R_{node}| + |S_{node}| + \left(\sum_{i=1}^{|WQ|} Cost_{WQ}(S, wq_i) - |S_{node}| \right) \times P[\mathbf{X} \geq b], \quad (1)$$

where $|R_{node}|$ and $|S_{node}|$ are the number of nodes in R and S, respectively, $|WQ|$ is the total number of window searches performed during the join, and wq_i is the ith window search query. This wq_i is expressed as $\langle M_{wq_i}, t\rangle$, where M_{wq_i} denotes the MBB at the join time stamp t.

Proof. Since the window search cost varies, depending on which nodes of the tree S accessed in a previous query are still in the buffer, $Cost_{WQ}(S, q)$ should be derived with the buffer taken into consideration. Let us separate the cost $Cost_{WQ}$ into the cost of accessing the nodes of the tree S accessed previously (and may or may not be in the buffer), $Cost_{WQ_p}$, and the cost of accessing the nodes of the tree S the first time, $Cost_{WQ_f}$. That is,

$$Cost_{WQ}(S,q) = Cost_{WQ_p}(S,q) + Cost_{WQ_f}(S,q). \quad (2)$$

CoPST accesses each node of R once and only once to sort the leaf entries (using the priority queue). Then, it accesses each node of S at least once to perform window searches on S using the windows' bounding sorted leaf entries. There is an additional cost for evaluating a window query, depending on the available buffer size. Thus, the total join cost is expressed as

$$|R_{node}| + |S_{node}| + \left(\sum_{i=1}^{|WQ|} Cost_{WQ_p}(S, wq_i) \right) \times P[\mathbf{X} \geq b]. \quad (3)$$

This can be rewritten as follows using (2):

$$= |R_{node}| + |S_{node}| + \left(\sum_{i=1}^{|WQ|} Cost_{WQ}(S, wq_i) - \sum_{i=1}^{|WQ|} Cost_{WQ_f}(S, wq_i) \right) \times P[\mathbf{X} \geq b]. \quad (4)$$

Here, $\sum_{i=1}^{|WQ|} Cost_{WQ_f}(S, wq_i)$, the summation of the costs of accessing the nodes of S for the first time, is the same as the number of nodes in S, $|S_{node}|$. Hence, (4) can be rewritten as (1). □

Now, let us discuss the probability distribution of the random variable $\mathbf{X}$. We can regard $\mathbf{X}$ values as interarrival times. Here, following common practice [16], [17], we measure time intervals in terms of the number of page accesses between two repeated accesses of the same page in the page reference string. Consider a stochastic process $\{S_t = \sum_{i=1}^{t} X_i | t = 1, 2, \ldots\}$, where X_i is i.i.d. drawn from $F(\mathbf{X})$, and S_t is the time to the tth event. Additionally, consider $N(t)$ defined as the number of events occurring in $(0, t]$. Due to the ordered window searches, CoPST shows high locality of page accesses so that the probability of X_t being greater than a certain time period l decreases dramatically with increasing l. This, along with the i.i.d. property of X_t, indicates that $N(t)$ is a Poisson process, and thus, $\mathbf{X}$ follows the exponential distribution. The probability density function $f(\mathbf{X})$ is thus expressed as

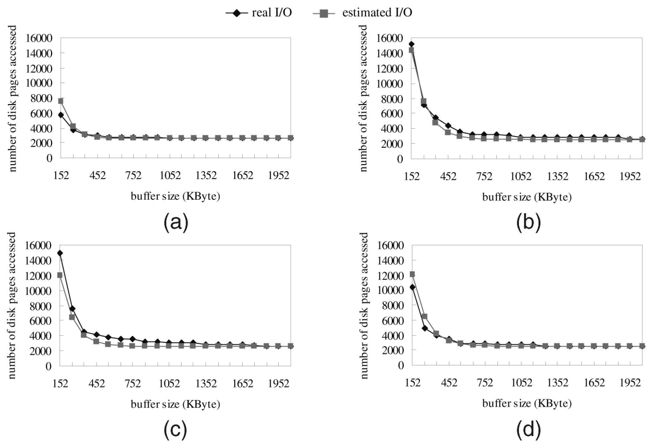


Fig. 7. Precision of the cost model at join time stamp 30. (a) Uniform distribution. (b) Gaussian distribution. (c) Skewed distribution. (d) Road network distribution.

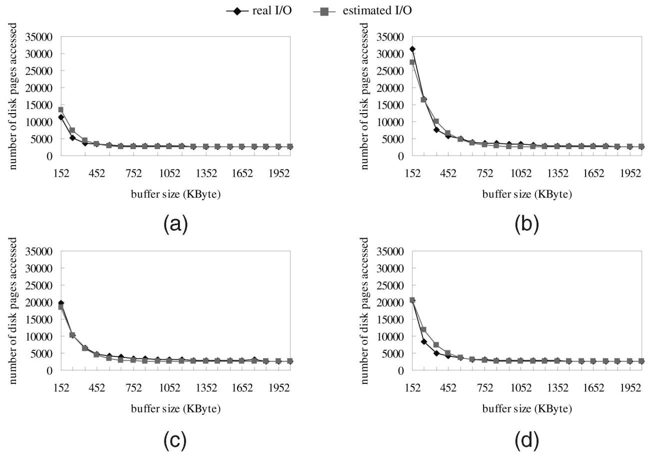


Fig. 8. Precision of the cost model at join time stamp 50. (a) Uniform distribution. (b) Gaussian distribution. (c) Skewed distribution. (d) Road network distribution.

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0, \end{cases} \tag{5}$$

where $\lambda > 0$ and the parameter λ is equal to the inverse of the mean μ of $\mathbf{X}$. $P[\mathbf{X} \geq b]$ is expressed as follows:

$$P[\mathbf{X} \geq b] = 1 - \int_0^b f(\mathbf{X}) = 1 - (1 - e^{-\lambda b}) = e^{-\lambda b}.$$

Similar approaches are used to estimate the probability distribution for the depth-first spatial join [18], [19].

To model $\mathbf{X}$ as an exponential random variable indicates that most of the $\mathbf{X}$ values are less than $\mu \times c$ $(c \geq 1)$, where c is a constant. When $c = 2$, $P[\mathbf{X} \geq \mu \times c] = e^{-2} \simeq 0.14$; when $c = 3$, $P[\mathbf{X} \geq \mu \times c] = e^{-3} \simeq 0.05$. In CoPST, 1) a page fault rate decreases significantly in case the buffer size is larger than the number of pages accessed during a window search, and 2) between any two consecutive window searches WQ_1 and WQ_2, a set of pages accessed by WQ_1 significantly overlap with a set of pages accessed by WQ_2. Thus, we estimate μ as the average number of nodes accessed during each window search on the tree S. That is,

$$\mu = \frac{\left(\sum_{i=1}^{|WQ|} Cost_{WQ}(S, wq_i)\right)}{|WQ|}. \tag{6}$$

Then, $P[\mathbf{X} \geq b]$ is expressed as follows:

$$P[\mathbf{X} \geq b] = e^{-\frac{|WQ|}{\sum_{i=1}^{|WQ|} Cost_{WQ}(S,wq_i)} \times b}. \tag{7}$$

Now, we estimate $|WQ|$ and $\sum_{i=1}^{|WQ|} Cost_{WQ}(S, wq_i)$. To do so, we construct a two-dimensional histogram, where each bucket corresponds to a cell in the ACRM order. To calculate the number of objects in the buckets, we first obtain a set of CBBs, $leafCBBs$, of leaf nodes of R by only accessing the parent nonleaf nodes of the leaf nodes. Second, for each CBB in $leafCBBs$, we find buckets that overlap the CBB and evenly distribute the number of objects in the CBB to the overlapping buckets. Here, we assume that each leaf node has the same number of entries, as in Section 2.1. After these two steps, if there is a histogram bucket whose size is larger than the blocking factor, we partition the histogram bucket along the y-axis to simulate the ACRM order so that the size of each partitioned bucket is less than the blocking factor. After constructing the histogram, we can estimate $|WQ|$ as the number of buckets in the histogram. To estimate $\sum_{i=1}^{|WQ|} Cost_{WQ}(S, wq_i)$, we perform window searches using the buckets on the tree S. In these window searches, we only need to access nonleaf nodes of S. Note that all these estimation costs, which are included in the join cost throughout all experiments, constitute less than 1 percent of the total performance cost.

3.1 Validation of the Cost Model

We have conducted various experiments to verify the precision of this cost model using various distributions. We used the TPR*-tree as a predictive spatiotemporal tree for all experiments. In the experiment, the data sets used are Groups 3 $\sim$ 6 (see Section 5), and the current update time for the TPR*-tree is set to 20.

Figs. 7 and 8 show the results obtained for a varying buffer size when the join time stamps are 30 and 50, respectively. The figures show that the cost model is very precise for the entire range of buffer size. For the join time stamp 30, the average errors for the uniform, Gaussian, skewed, and road network distributions are 4.24 percent, 10.79 percent, 13.38 percent, and 5.80 percent, respectively. For the join time stamp 50, the average errors for the uniform, Gaussian, skewed, and road network distributions are 6.86 percent, 10.80 percent, 11.64 percent, and 8.33 percent, respectively. This shows that the cost model is precise regardless of the join time stamp value and the data distribution. Experiment results in Section 5 confirm that CoPST works very well for all these distributions.

4 General CoPST Algorithm

4.1 Basic Concepts

Up to this point, it has been assumed that the priority queue used in the join algorithm is resident in a main memory space separate from the buffer space used for the

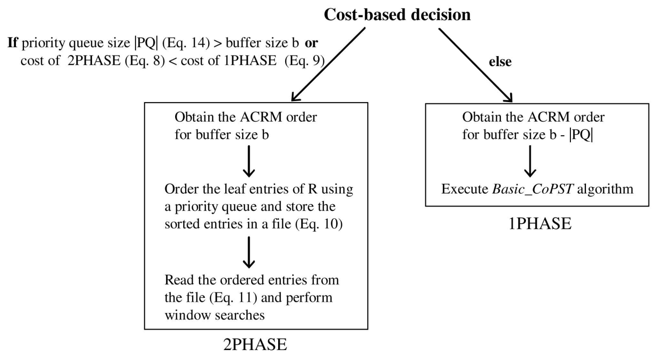

Cost-based decision
If priority queue size |PQ| (Eq. 14) > buffer size b or cost of 2PHASE (Eq. 8) < cost of 1PHASE (Eq. 9)
else
Obtain the ACRM order for buffer size b
Order the leaf entries of R using a priority queue and store the sorted entries in a file (Eq. 10)
Read the ordered entries from the file (Eq. 11) and perform window searches
2PHASE
Obtain the ACRM order for buffer size b - |PQ|
Execute Basic_CoPST algorithm
1PHASE

Fig. 9. Illustration of how the General CoPST algorithm works.

join. In fact, however, the storage space needed for a priority queue should share the same buffer space needed for the nodes of the trees (i.e., R and S). The size of a priority queue can become very large in an environment where overlaps occur frequently, like moving object databases. Thus, in this section, we discuss enhancing the $Basic\_CoPST$ algorithm to handle a large priority queue efficiently.

The size of the priority queue has a significant influence on the buffer utilization efficiency during join processing. Especially, in case the priority queue size is larger than the buffer size, most of the tree nodes accessed in the previous window search are replaced before the next window search due to buffering of the priority queue. This decreases the buffer utilization efficiency drastically.

In order to resolve this problem, we consider an alternative algorithm that separates the algorithm $Basic\_CoPST$ into two phases. The first phase of the algorithm orders the leaf entries of the tree R using a disk-based priority queue and saves the ordered entries in a file (called the *leaf entry file*). The second phase reads the ordered entries from the file and performs window searches. We call this algorithm the *two-phase algorithm* ($2PHASE$) and the original $Basic\_CoPST$ the *one-phase algorithm* ($1PHASE$).

With both $1PHASE$ and $2PHASE$ available, the criterion to choosing between them becomes an issue. For this purpose, we use cost models of each algorithm and choose one based on the cost estimates at join time. Fig. 9 shows how the algorithm makes a cost-based decision to pick either $1PHASE$ or $2PHASE$ algorithms. In this figure, we also add equation numbers, which reference the different parts of the cost model in the next section.

4.2 Cost Models of $2PHASE$ and $1PHASE$ and the Selection Rule

The cost of $2PHASE$ is expressed as a sum of the cost of ordering the leaf entries of R using a priority queue PQ and storing the sorted entries in a leaf entry file F ($Cost_{PQsort}$), the cost of scanning the leaf entry file ($Cost_{scan}$), and the cost of executing the $Basic\_CoPST$ algorithm ($Cost_{Basic}$):

$$\begin{aligned} Cost_{2PHASE}(R,S,b,PQ,F) = & Cost_{PQsort}(R,b,PQ,F) \\ & + Cost_{scan}(F) \\ & + (Cost_{Basic}(R,S,b) - |R_{node}|), \end{aligned} \tag{8}$$

where $|R_{node}|$ is the number of nodes in the trees R, and b is the buffer size. Note that $|R_{node}|$ should be subtracted from $Cost_{Basic}$ because $Cost_{Basic}$ includes the cost of scanning the tree R while it is already included in $Cost_{PQsort}$. See (1) for $Cost_{Basic}$. $Cost_{PQsort}$ and $Cost_{scan}$ will be shown in (10) and (11), respectively, below.

The cost of $1PHASE$ is the cost of executing $Basic\_CoPST$ with the buffer of size $b - |PQ|$, where $|PQ|$ denotes the size of the priority queue (a conservative estimation of $|PQ|$ will be shown in (14)):

$$Cost_{1PHASE}(R,S,b,PQ) = Cost_{Basic}(R,S,b-|PQ|). \tag{9}$$

4.2.1 PQ-Based Sorting Cost and Leaf Entry File Scan Cost

In our work, we have implemented the priority queue using a B$^+$-tree (due to its simplicity and good performance). If $b \geq |PQ|$, then all the B$^+$-tree nodes reside in the buffer, and thus, the only disk I/O costs are for scanning the nodes of R and writing the output leaf entry file:

$$Cost_{PQsort}(R,b,PQ,F) = |R_{node}| + |F|, \tag{10}$$

where f is the average fill factor of a node, and $|F|$ is the size of the leaf entry file.

Here, $|F|$ equals the number of pages required to store all the leaf entries, that is, equals $|R_{leafentry}|/bf$, where $|R_{leafentry}|$ is the number of leaf entries in R, and bf is the blocking factor of a page. Thus, (10) is rewritten as

$$\begin{aligned} Cost_{PQsort}(R,b,PQ,F) &= |R_{node}| + |R_{leafentry}|/bf \\ &= |R_{node}| + |R_{leafnode}| \times f. \end{aligned}$$

The leaf entry file scanning cost is expressed as

$$Cost_{scan}(F) = |R_{leafnode}| \times f. \tag{11}$$

4.2.2 Predicting the Priority Queue Size

Now, we explain the method for predicting the priority queue size $|PQ|$. We use a maximum size to consider the worst case. The leaf entries in the priority queue are the entries in the leaf nodes that overlap the leaf entries ordered so far. In case the data are distributed uniformly and the sorting is done in the ACRM order, the number of entries in the priority queue is the maximum if the ACRM order consists of *two* CRM components. Fig. 10 illustrates such a case, where the dots depict the leaf entries in the priority queue.

Thus, we can predict the number of leaf entries stored in a priority queue, $|PQ_{leafentry}|$, by computing the ratio of the area occupied by those leaf entries to the total area:

$$|PQ_{leafentry}| = |R_{leafentry}| \times (1 \times s \times c + 0.5 \times s \times c), \tag{12}$$

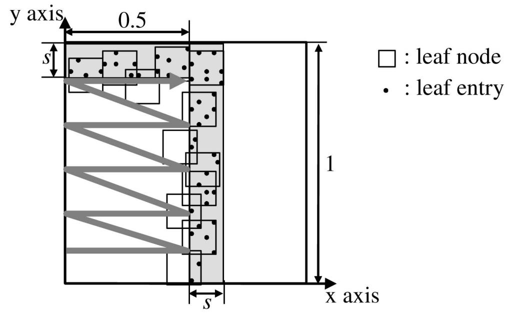


Fig. 10. An illustration of a priority queue with the maximum number of leaf entries.

where s denotes the average side (i.e., dimension axis) length of the leaf nodes, and c denotes the fraction of the leaf entries that are in the priority queue. In the worst case, the priority queue has all the entries of the leaf nodes overlapping the leaf entries sorted so far. The value of c is 1.0 in this case. In this paper, we use a value in the middle between the average case ($= 0.5$) and the worst case ($= 1.0$), that is, $c = 0.75$. Our experiments with various data sets have shown that the cost estimation with c equal to 0.75 is slightly more robust with respect to the data distribution than with 0.5 or 1.0.

By extending (12) to take all entries into account, the maximum number of entries in the priority queue, $|PQ_{entry}|$, is expressed as follows:

$$|PQ_{entry}| = \sum_{i=1}^{h} \big(|R_{i,entry}| \times 1.5 \times s_i \times c\big), \qquad (13)$$

where h denotes the tree height, $|R_{i,entry}|$ denotes the number of entries in all the nodes at level i (leaf nodes are at level 1), and s_i denotes the average side length of the nodes at level i. (The value of s_i can be obtained through a nonleaf node search; this cost will be included in all the experiments in Section 5.) Let c_p denote the average number of nodes that can be stored in the B$^+$-tree representing the priority queue. Then, the maximum size of the priority queue, $|PQ|$, is the number of the B$^+$-tree nodes storing the $|PQ_{entry}|$ entries, computed as follows:

$$|PQ| = \sum_{i=1}^{h'} \frac{|PQ_{entry}|}{{c_p}^i}, \qquad (14)$$

where $h' = 1 + \lceil \log_{c_p} \frac{|PQ_{entry}|}{c_p} \rceil$.

4.3 Putting Them All Together

Algorithm 3 shows the algorithm $General\_CoPST$, which automatically chooses either $1PHASE$ or $2PHASE$, depending on the estimated cost. It first estimates the size of the priority queue (using (14)), estimates the costs of $2PHASE$ and $1PHASE$ (using (8) and (9)), and then chooses between $2PHASE$ and $1PHASE$ (line 1). Then, it configures the ACRM order according to the chosen algorithm and performs the join using the resulting ACRM (either lines 2-3 or lines 5-6).

```
Algorithm 3. $General\_CoPST(rootR, rootS, t, D_{thr}, bf, b)$
Require: $rootR$: root node of tree $R$, $rootS$: root node of Tree $S$, $t$: join time stamp, $D_{thr}$: density threshold
$bf$: blocking factor, $b$: buffer size
1: if $b < |PQ|$ or $Cost_{2PHASE} < Cost_{1PHASE}$ then
2:    obtain the ACRM order $acrm$ for buffer size $b$
3:    $2PHASE(rootR, rootS, t, acrm, D_{thr}, bf)$
4: else
5:    obtain the ACRM order $acrm$ for the buffer size
      $b - |PQ|$
6:    $1PHASE(rootR, rootS, t, acrm, D_{thr}, bf)$
7: end if
```

5 Performance Evaluation

We evaluate the performance of our $General\_CoPST$ algorithm by comparing the join processing cost between this algorithm and the three algorithms based on the depth-first, the breadth-first, and the transformation-view-based spatial join algorithms (see Section 6 for an outline of these algorithms). We refer to these four algorithms as CoPST, DFSJ, BFSJ, and TVSJ from here on. BFSJ uses as an ordering key the lower x-coordinate of the node of the tree R (called OrdOne), which is reported to be the best ordering choice in BFSJ [12]. The main objective of the experiments is to compare the join processing cost of CoPST against DFSJ, BFSJ, and TVSJ and evaluate the effect of the cost-based switch-over between 1PHASE and 2PHASE in CoPST for a varying buffer size. In Section 5.1, we describe the setup for the experiments, and in Section 5.2, we present the experiments conducted and their results.

5.1 Experiment Setup

We generate experimental data sets using the *Generate Spatiotemporal Data (GSTD)* tool [20]. The GSTD tool is a data generator used broadly in the performance evaluations on moving objects and is able to generate moving objects with various distributions [21]. In our experiments, we use GSTD to generate data sets with the uniform, Gaussian, and skewed distributions. The maximum speed of an object is set to 2 km/minute.

We also use the same data set as the one generated by Saltenis et al. [10]. This data set simulates a road network; it contains imaginary objects moving on a road. Each object randomly chooses its source and destination points from 200 fixed points randomly distributed in a 1,000 km $\times$ 1,000 km space. Once the objective arrives at the destination, then it randomly chooses the next destination and moves to there again. The maximum speed of each object is one among 0.75 km/minute, 1.5 km/minute, and 3 km/minute. While moving from a source to a destination, an object accelerates in the first 1/6 of the distance, moves at the maximum speed in the next 2/3, and decelerates in the last 1/6. During this move, each object reports its speed and location every 20 minutes on the average.

The data sets indexed by R and S are point objects, and a join using R and S produces all pairs of point objects that are within less than a certain distance (Dt in Table 2) from each other. For simplicity, both data sets have the same number of objects (i.e., $|R_{object}| = |S_{object}|$); this makes the

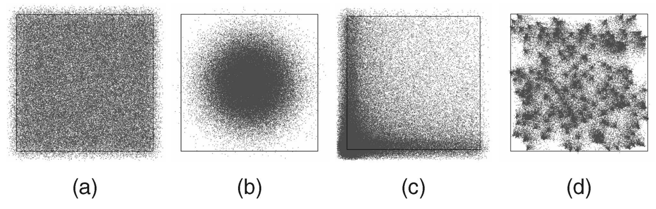


Fig. 11. Distributions of the objects indexed by the tree R at the current time $(= 20)$. (a) Uniform. (b) Gaussian. (c) Skewed. (d) Road network.

analysis of experimental results easy. Fig. 11 depicts the distributions of the point objects used in the experiments. For the road network data set, we use the same data set for building both R and S; this simulates a self-join.

We use TPR*-trees for the tree R and tree S. Table 1 shows the statistics of the two TPR*-trees and the moving object data set for each join group. Four different values of the TPR*-tree horizon (H) are used, and two different page sizes (4 Kbytes and 8 Kbytes) are used for road network data sets, as shown in the table. The last column shows the number of pairs of objects in the join result. "Distance" is used as the spatial join predicate in all experiments.

All experiments are done on Window Server 2003 PC with 512 Kbytes unified (data and instruction) L2 cache, 512 Mbytes RAM, and Pentium IV 2.8-GHz CPU. We use LRU as the buffer page replacement algorithm.

5.2 Experiments and Results

As already mentioned, our objective is to compare the join processing costs of different algorithms for different parameter values. We have conducted the experiments for different values of the parameters listed in Table 2. (The parameter values in boldface are the default values used for fixed parameters in each experiment.)

The experiment results show that CoPST is more efficient than the other three algorithms in all the cases tested. Moreover, the results show that CoPST is particularly more efficient than the others when the buffer size is small or the data distribution is nonuniform. Both of these are typical for a moving object database. That is, a spatiotemporal database is typically very large compared with the available buffer size (this is more so in a multiuser environment), and many real data distributions are skewed or clustered.

TABLE 1
Statistics of the Index Trees R and S and the Moving Object Data Set

H	Page size	Join group	TPR*-tree R			TPR*-tree S			$\|R_{object} \bowtie S_{object}\|$
			$\|R_{node}\|$	$\|R_{object}\|$	Data distribution	$\|S_{node}\|$	$\|S_{object}\|$	Data distribution	(Join timestamp=30)
40	4K	Group1	1270	100,000	road network	1270	100,000	road network	127010
70	4K	Group2	634	50,000	road network	634	50,000	road network	57670
	4K	Group3	1292	100,000	uniform	1276	100,000	uniform	1137
	4K	Group4	1287	100,000	Gaussian	1258	100,000	Gaussian	4148
	4K	Group5	1305	100,000	skewed	1287	100,000	skewed	4680
	4K	Group6	1269	100,000	road network	1269	100,000	road network	127010
	8K	Group7	619	100,000	road network	619	100,000	road network	127010
100	4K	Group8	1286	100,000	road network	1286	100,000	road network	127010
130	4K	Group9	1274	100,000	road network	1274	100,000	road network	127010

(current time = 20)

TABLE 2
Parameters Used in the Experiments

Parameter	Description	Values Used
Now	current time	**20**
Ds	distribution of objects	uniform, Gaussian, skewed, **road network**
N	number of objects	50,000, **100,000**
JTS	join timestamp	**30**, 40, 50, 60, 70
H	TPR*-tree parameter (horizon)	40, **70**, 100, 130
Dt	Distance	**0.2KM**, 1KM, 5KM, 25KM
P	Page size	**4K**, 8K

The values in bold face are the default used when the values are fixed in the experiments.

For applications that require quick responses, it may be desirable to have both indexes resident in the main memory. In this case, the CPU cache, with its limited size, amounts to the buffer of a disk-based environment. The Pentium IV CPU uses a "pseudo LRU" (a variant of LRU) as the cache entry replacement policy [22]. Our experiment shows that CoPST consistently outperforms the other methods even in a main-memory environment.

Now, we show the results for each case of the experiments. In some experimental results, the buffer size is varied in two separate ranges: small (152 Kbytes $\sim$ 1,052 Kbytes) and large (1,100 Kbytes $\sim$ 2,052 Kbytes), each corresponding to 1.5 percent $\sim$ 10.4 percent and 10.8 percent $\sim$ 20.2 percent of the total size of the two join index trees of the default data set (Group 6). In other experimental results (in which another parameter is varied), the buffer size is set to two values selected from each range, that is, 300 Kbytes and 600 Kbytes from the smaller range and 1,300 Kbytes and 1,600 Kbytes from the larger range.

Due to the page limitation, we explain only the graphs with the smaller buffer size, except for the case of different data distributions; the graphs of the large buffer size show the same trends (see Figs. 21, 22, 23, 24, 25, and 26). For the same reason, we show only the graphs of disk I/O costs; the graphs of elapsed time show the similar trends (see Fig. 12). We also show the results obtained in a main-memory resident environment.

5.2.1 For Different Values of the Parameters

Figs. 13 and 14 show the results for the four distributions (Ds) in Group 3 through Group 6, with the buffer size varying in the small range and the large range, respectively. We see that CoPST outperforms DFSJ, BFSJ, and TVSJ in the entire range of the buffer size regardless of the data distribution. In our experiments, CoPST switches over between 1PHASE and 2PHASE when the buffer size is

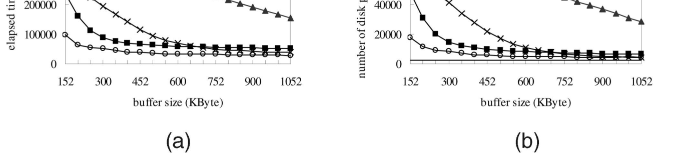


Fig. 12. Elapsed time and disk I/O cost for the road network distribution. (a) Elapsed time. (b) Number of disk pages accessed.

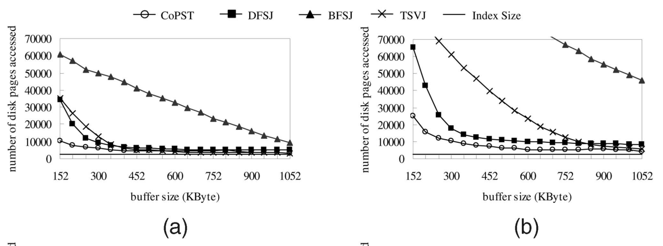
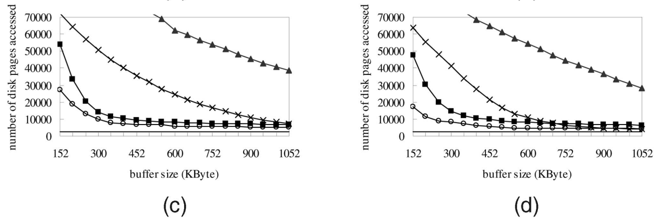



Fig. 13. Disk I/O costs for different data distributions (small buffer). (a) Uniform distribution. (b) Gaussian distribution. (c) Skewed distribution. (d) Road network distribution.

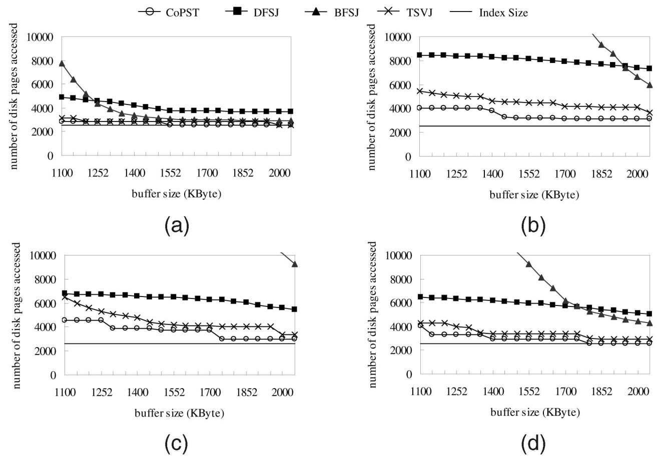


Fig. 14. Disk I/O costs for different data distributions (large buffer). (a) Uniform distribution. (b) Gaussian distribution. (c) Skewed distribution. (d) Road network distribution.

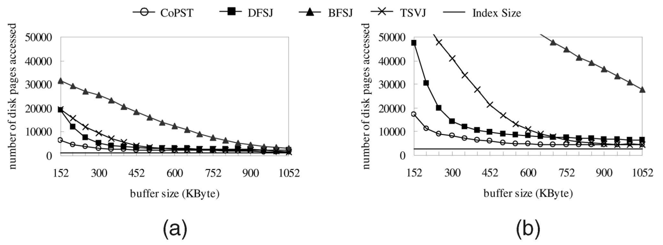


Fig. 15. Disk I/O costs for different numbers of objects. (a) 50,000 objects. (b) 100,000 objects.

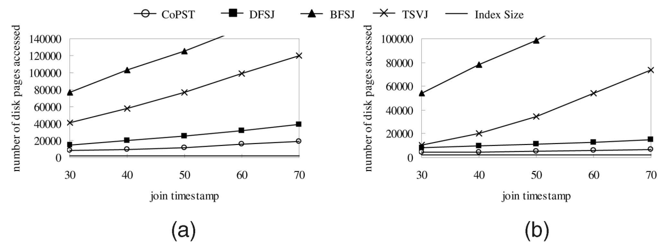


Fig. 16. Disk I/O costs for different join time stamps. (a) $\text{Buffer size} = 300$ Kbytes. (b) $\text{Buffer size} = 600$ Kbytes.

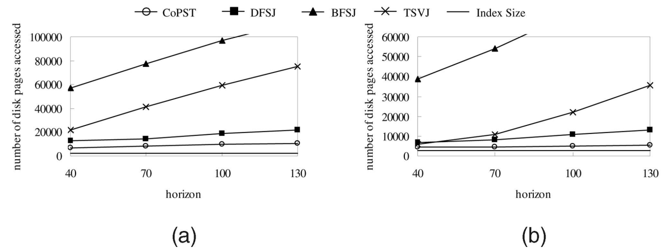


Fig. 17. Disk I/O costs for different horizons. (a) $\text{Buffer size} = 300$ Kbytes. (b) $\text{Buffer size} = 600$ Kbytes.

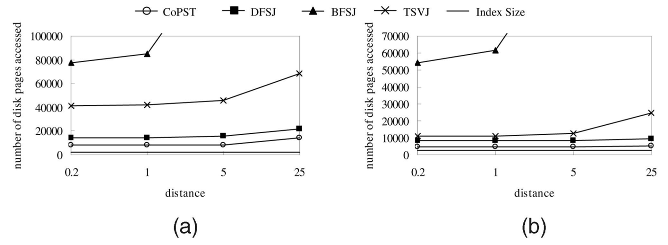


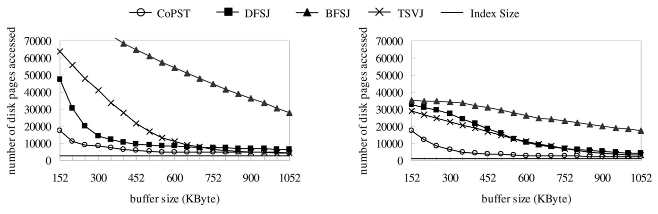
Fig. 18. Disk I/O costs for different distances. (a) $\text{Buffer size} = 300$ Kbytes. (b) $\text{Buffer size} = 600$ Kbytes.

around 800, 852, 800, and 1,000 Kbytes for the uniform, Gaussian, skewed, and road network distributions, respectively. Due to the high accuracy of our cost model, the CoPST curve in the figure shows a smooth transition between the two phases. Fig. 15 shows the results for the two different numbers of objects (N), using the data sets of Group 2 (50,000 objects) and Group 6 (100,000 objects). Naturally, when the number of objects is larger, the join cost is higher for all the algorithms.

Fig. 16 shows the results for different join time stamps (JTS) varying from 30 to 70 at the increment of 10. When the join time stamp is larger, the join cost is higher for all the algorithms because the CBBs expand more as the time goes farther from the current time. Fig. 17 shows the results for different horizons (H) varying from 40 to 130 at the increment of 30. When the horizon is larger, the join cost is higher for all the join algorithms because the CBBs expand more as the horizon is farther away. Fig. 18 shows the results for different distances (Dt) varying from 0.2 Km to 25 Km at the factor of five. When the distance is longer, the join cost is higher for all the join algorithms because there are more objects satisfying the join predicate.

Fig. 19 shows the results for the two page sizes: 4 Kbytes and 8 Kbytes. When the page size is larger, the I/O cost is lower for all the join algorithms, because fewer pages need to be fetched for the same amount of data. However, we cannot continue to increase the page size indefinitely, because poor performance will result; the extreme case is when we have one only page containing all moving objects. When we performed experiments using the page size of 16 Kbytes, the peak performances of all four algorithms became worse, due to larger CPU cost (i.e., the larger the bounding boxes are, the more comparisons are made).

In all these experiments, we see that CoPST gains more advantage over the other algorithms as the parameter value increases, with its ability to construct tightened bounding boxes dynamically while the join is performed at a future time stamp. Besides, while the join cost increases in the same way for all the join algorithms, the increase is the smallest for CoPST.



(a) (b)

Fig. 19. Disk I/O costs for different page sizes. (a) $\text{Page size} = 4$ Kbytes. (b) $\text{Page size} = 8$ Kbytes.

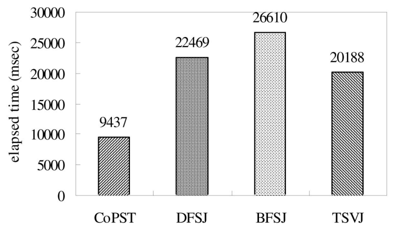
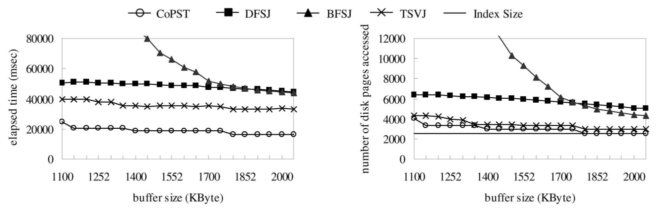
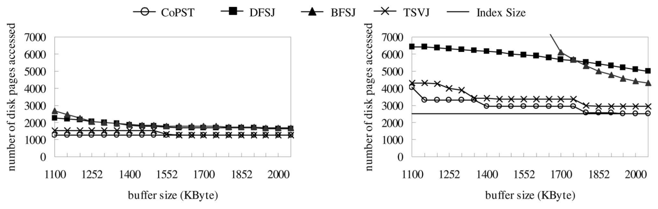


Fig. 20. Elapsed time for main-memory-based joins.



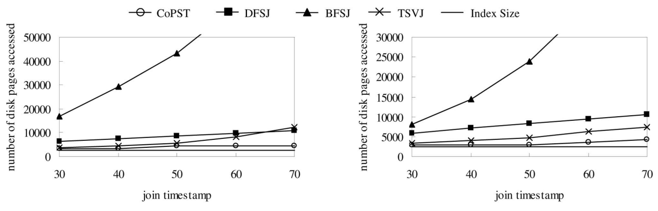
(a) (b)

Fig. 21. Elapsed time and disk IO cost. (a) Elapsed time. (b) Number of disk pages accessed.



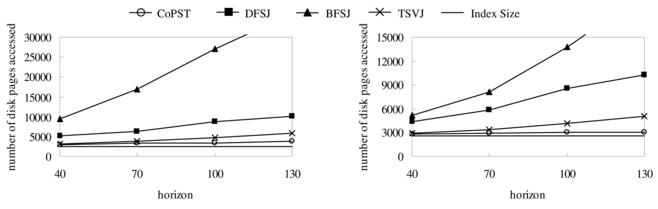
(a) (b)

Fig. 22. Disk I/O costs for different numbers of objects. (a) 50,000 objects. (b) 100,000 objects.



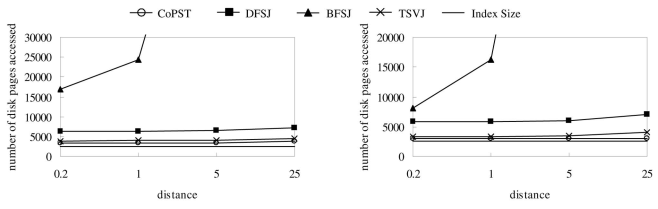
(a) (b)

Fig. 23. Disk I/O costs for different join time stamps. (a) $\text{Buffer size} = 1{,}300$ Kbytes. (b) $\text{Buffer size} = 1{,}600$ Kbytes.



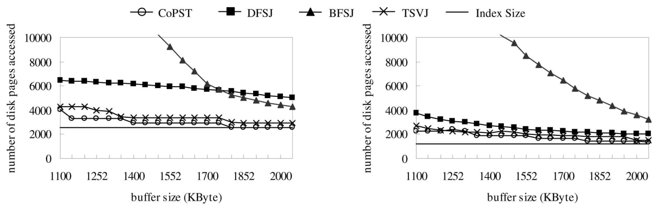
(a) (b)

Fig. 24. Disk I/O costs for different horizons. (a) $\text{Buffer size} = 1{,}300$ Kbytes. (b) $\text{Buffer size} = 1{,}600$ Kbytes.



(a) (b)

Fig. 25. Disk I/O costs for different distances.(a) $\text{Buffer size} = 1{,}300$ Kbytes. (b) $\text{Buffer size} = 1{,}600$ Kbytes.



(a) (b)

Fig. 26. Disk I/O costs for different page sizes.(a) $\text{Page size} = 4$ Kbytes. (b) $\text{Page size} = 8$ Kbytes.

As the buffer size becomes larger, the I/O cost gap between CoPST and the other algorithms (DFSJ, BFSJ, and TVSJ) decreases (see 22, 23, 24, 25, and 26). However, CoPST is still at least two times faster than the other algorithms consistently over all buffer sizes (see Fig. 21). This comes from the fact that when the buffer size is large the CPU cost is a major cost determining the elapsed time while using the tightened bounding boxes in CoPST reduces the CPU time (as well as the I/O time) in contrast to the other three algorithms.

5.2.2 For Main-Memory-Based Joins

Fig. 20 shows the elapsed times of the algorithms in a main-memory database environment. It is not possible to show the results for a varying cache size (in the same manner as in Figs. 13, 14, 15, 16, 17, 18, and 19) because the cache size is fixed for a given PC hardware. We see that the performance advantage of CoPST over the other algorithms is larger than in the disk-based environment. CoPST reduces the number of comparisons, as well as the number of CPU cache misses, and the number of comparisons plays a major role in the main-memory environment.

6 Related Work

To our knowledge, there has been no predictive spatiotemporal join algorithm in the literature. Sun et al. [6] and Tao et al. [4] researched on selectivity estimation of predictive spatiotemporal joins but did not address the cost formula of predictive spatiotemporal joins. Tao et al. [5] dealt with predictive *window queries*, but these are not join queries. Iwerks et al. [23] proposed a "spatial semijoin" algorithm, but the join time was the *present*, and the focus was on how to *maintain* join results as time passes. Jeong et al. [3] studied the performance of spatiotemporal join strategies, but they considered only *historical* spatiotemporal joins. Similarly, Arumugam and Jermaine [1] dealt with *historical* closest pair joins, and Bakalov et al. [2] dealt with a variant of *historical* spatiotemporal join.

As for the spatial join algorithms using indexes on both input files,[1] there are three representative spatial join algorithms, using R*-trees [27]. They are the DFSJ algorithm [11], the BFSJ algorithm [12], and the TVSJ algorithm [13]. The third one has been proposed recently and shown to perform better than the first two.

The depth-first algorithm finds the pairs of overlapping leaf entries through a depth-first traversal of each index. In this case, if there is no overlap between two nonleaf nodes (one node from each file), then there cannot be any overlap between all children nodes traversed recursively from them. By taking advantage of this property, the join algorithm reduces the number of nodes to be compared in the two input files. One problem is that the same page (with one page per node) may be read repeatedly from disk. In order to alleviate this problem, heuristic approaches such as local plane sweeping and pinning have been used in [11]. These heuristic approaches, however, do not guarantee a globally optimal node access order because they optimize the access order only for the children nodes of one pair of overlapping nonleaf nodes at a time and thus does not consider the access order for all the nodes that may overlap [13].

The breadth-first algorithm guarantees to find a globally optimal node access order. It first saves in an "intermediate join index" the information about all pairs of overlapping nodes found through a breadth-first traversal of the indexes. Then, it considers the access order of all the join candidate nodes in the intermediate join index. This generates a globally optimal node access order. However, we observe that as the number of the pairs of overlapping nodes becomes large, which is common in a moving object database, the size of the intermediate join index increases quadratically. In this case, it entails expensive I/O cost to materialize the join index and to generate the optimal access order.

The transformation-view-based algorithm first sorts the leaf nodes of one R*-tree by their spatial adjacency in a "transformed space." Then, it finds overlapping nodes by performing a window search of the other R*-tree using each node in the first R*-tree as the window. It improves the buffer utilization by allowing the nodes accessed in the previous window search to remain in the buffer as long as possible. We observe, however, that in a moving object database the buffer utilization is poor with small buffer if there is much overlap among the sorted leaf nodes. In [28], a variant of spatial join with a similar spirit is presented; it first sorts the leaf nodes of one R^*-tree in the Hilbert order and executes the nearest neighbor search for the other R^*-tree. This algorithm has the same problem.

1. There exists another class of spatial join algorithms that do not use indexes on both files. Examples are the Partition-Based Spatial Merge Join [24], Spatial Hash Join [25], Iterative Spatial Join [26], and Slot Index Join [19]. Each class has its own advantages and applications.

These spatial join algorithms can be applied to predictive spatiotemporal trees [8], [7], [9], [10], [5]. However, the predictive spatiotemporal trees are designed to optimize the *overall* query performance for a duration H (horizon) from the index creation time. As a result, the bounding boxes do not bound objects optimally during an interval shorter than H, and hence, there are unnecessarily many pairs of overlapping nodes found. This causes a performance degradation. Specifically, in the depth-first case, the performance degrades due to the large number of overlapping node pairs, as well as the ineffectiveness of local optimization. In the breadth-first case, the performance degrades due to the high cost of using an intermediate join index, as the size of the intermediate index increases quadratically with the number of overlapping node pairs. In the transformation-view-based case, the performance degrades due to a low buffer utilization, which reduces the likelihood of a node accessed in the previous window query to remain in the buffer.

The fundamental reason for all these problems is that the predictive spatiotemporal indexes use bounding boxes that are not optimal at a future query time. The solution, therefore, is to tighten the bounding boxes at runtime.

7 Conclusion

In this paper, we proposed an efficient predictive spatiotemporal join algorithm called CoPST, which performs a join on future time and space. CoPST involves index-assisted sorting, density-based moving-object bounding, and a window search with a bounding box. These features are important for constructing the bounding boxes globally tight during join processing to minimize the overlap and the join processing cost. Next, we proposed ACRM, a new space filling curve used to determined the join order. The ACRM order adaptively controls the order of pages to be joined depending on the available buffer size. Next, we developed a novel probabilistic cost model by incorporating the effect of page buffering into the cost. Our model can accurately estimate the join cost regardless of the data distribution. Moreover, CoPST switches dynamically between the main memory and the disk for temporary buffering of the priority queue used in the index-assisted sorting phase, based on the probabilistic cost model. The resulting algorithm is a hybrid of one- and two-phase algorithms.

We conducted extensive experiments using various data distributions to evaluate our join algorithm for a wide range of buffer size. In all the experiments, our algorithm outperformed the algorithms adapted from state-of-the-art spatial join algorithms.

To our knowledge, CoPST is the first predictive spatiotemporal join algorithm. We believe more research will follow in this direction. The future work includes developing predictive spatiotemporal join algorithms that are usable when only one or neither input file is indexed, when the join condition is on the time interval (i.e., time-interval joins), and when parallelizing the algorithm.

ACKNOWLEDGMENTS

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-521-D00399). Yufei Tao was supported by grants CUHK 1202/06, 4161/07, 4173/08 from HKRGC.

REFERENCES

[1] S. Arumugam and C. Jermaine, "Closest-Point-of-Approach Join for Moving Object Histories," *Proc. 22nd Int'l Conf. Data Eng. (ICDE)*, 2006.

[2] P. Bakalov, M. Hadjieleftheriou, and V. Tsotras, "Time Relaxed Spatiotemporal Trajectory Joins," *Proc. 13th ACM Int'l Workshop Geographic Information Systems (GIS)*, 2005.

[3] S.-H. Jeong et al., "An Experimental Performance Evaluation of Spatio-Temporal Join Strategies," *Trans. GIS*, vol. 9, no. 2, 2005.

[4] Y. Tao, J. Sun, and D. Papadias, "Analysis of Predictive Spatio-Temporal Queries," *ACM Trans. Database Systems*, vol. 28, no. 4, 2003.

[5] Y. Tao et al., "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB)*, 2003.

[6] J. Sun, Y. Tao, D. Papadias, and G. Kollios, "Spatio-Temporal Join Selectivity," *Information Systems*, vol. 31, no. 8, 2006.

[7] J. Patel, Y. Chen, and V. Chakka, "Stripes: An Efficient Index for Predicted Trajectories," *Proc. ACM SIGMOD*, 2004.

[8] C.S. Jensen et al., "Query and Update Efficient B^{+}-Tree Based Indexing of Moving Objects," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB)*, 2004.

[9] M. Yiu et al., "The B^{dual}-Tree: Indexing Moving Objects by Space-Filling Curves in the Dual Space," *VLDB J.*, vol. 17, no. 3, 2008.

[10] S. Saltenis et al., "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD*, 2000.

[11] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *Proc. ACM SIGMOD*, 1993.

[12] Y.-W. Huang et al., "Spatial Joins Using R-Trees: Breadth-First Traversal with Global Optimizations," *Proc. 23rd Int'l Conf. Very Large Data Bases (VLDB)*, 1997.

[13] M. Lee et al., "Transform-Space View: Performing Spatial Join in the Transform Space Using Original-Space Indexes," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 2, Feb. 2006.

[14] J. Orensten, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD*, 1986.

[15] D. Hilbert, "Über Die Stetige Abbildung Einer Linie Auf Flächenstück," *Mathematische Annalen*, 1891.

[16] E.G. Coffman and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.

[17] E.J. O'Neil et al., "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD*, 1993.

[18] Y. Huang, N. Jing, and E. Rundensteiner, "A Cost Model for Estimating the Performance of Spatial Joins Using R-Trees," *Proc. Ninth Int'l Conf. Scientific and Statistical Database Management (SSDBM)*, 1997.

[19] N. Mamoulis and D. Papadias, "Slot Index Spatial Join," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 1, Jan./Feb. 2003.

[20] Y. Theodoridis, J. Silva, and M. Nascimento, "On the Generation of Spatiotemporal Datasets," *LNCS*, vol. 1651, 1999.

[21] D. Pfoser et al., "Novel Approaches to the Indexing of Moving Object Trajectories," *Proc. 26th Int'l Conf. Very Large Data Bases (VLDB)*, 2000.

[22] *A-32 Intel Architecture Optimization Reference Manual.* Intel Corporation, 2005.

[23] G. Iwerks, H. Samet, and K. Smith, "Maintenance of Spatial Semijoin Queries on Moving Points," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB)*, 2004.

[24] J. Patel and D. DeWitt, "Partition Based Spatial-Merge Join," *Proc. ACM SIGMOD*, 1996.

[25] M. Lo and C. Ravishankar, "Spatial Hash-Joins," *Proc. ACM SIGMOD*, 1996.

[26] E. Jacox and H. Samet, "Iterative Spatial Join," *ACM Trans. Database Systems*, vol. 28, no. 3, 2003.

[27] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD*, 1990.

[28] J. Zhang et al., "All-Nearest-Neighbors Queries in Spatial Databases," *Proc. 16th Int'l Conf. Scientific and Statistical Database Management (SSDBM)*, 2004.

Wook-Shin Han received the BS degree in computer engineering from Kyungpook National University, Daegu, Korea, in 1994 and the MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 1996 and 2001, respectively. He is currently a assistant professor in the Database Laboratory, Department of Computer Engineering, Kyungpook National University. In the past, he has worked as a postdoctoral researcher at IBM Almaden Research Center, working on parallel progressive optimization. His research interests include query processing and optimization, similarity search, XML databases, object-oriented/object-relational databases, and information retrieval. He is the workshop chair of CIKM 2009. He is an editorial board member of several international journals. He is a member of the IEEE Computer Society.

Jaehwa Kim is a member of the Database Laboratory led by Professor Wook-Shin Han in the Department of Computer Engineering, Kyungpook National University, Daegu, Korea. He is interested in join processing and similarity search.

Byung Suk Lee received the BS degree from Seoul National University, the MS degree from the Korea Advanced Institute of Science and Technology (KAIST), and the PhD degree from Stanford University. He held several positions in the industry and academia: previously at Gold Star Electric, Bell Communications Research, Datacom Global Communications, and the University of St. Thomas. He was also a visiting professor at Dartmouth College and a participating guest at Lawrence Livermore National Laboratory. He is currently an associate professor of computer science in the Department of Computer Science, University of Vermont, Burlington. His main research interests are database systems, data management, and query processing. He served on international conferences as a program committee member, a publicity chair, a special session organizer, and a workshop organizer, and also on the review panels of US federal funding agencies. He is a member of the IEEE Computer Society.

Yufei Tao joined the Department of Computer Science and Engineering, Chinese University of Hong Kong, New Territories, Hong Kong, in September 2006. Before that, he held positions at the Carnegie Mellon University and the City University of Hong Kong. He is engaged in research of database systems. He is particularly interested in index structures and query algorithms on multidimensional data and has published primarily on temporal databases, spatial databases, and privacy preservation. He received the Hong Kong Young Scientist Award in 2002. He has served the program committees of most prestigious database conferences such as ACM Sigmod, VLDB, and ICDE and is currently an associate editor of *ACM Transactions on Database Systems*. He is a member of the ACM.

Ralf Rantzau received the PhD degree from Universität Stuttgart. He is a researcher and a senior software engineer at IBM. He currently works on business intelligence, data integration, and RFID data management at the IBM Silicon Valley Laboratory, San Jose, California. Prior to this, he worked in the Intelligent Information Systems Research Group at the IBM Almaden Research Center on privacy technology for information systems, as well as context-based search problems. He has published more than 20 scientific papers, submitted several invention disclosures, and won the Best Paper Award at ICDE in 2006.

Volker Markl received the PhD degree from Technische Universität München. He is a full professor at Technische Universität Berlin, leading the Database Systems and Information Management Group. Prior to this, he led a research group at FORWISS, the Bavarian Research Center for Knowledge-Based Systems, and worked as a research staff member and project leader at the IBM Almaden Research Center. His research areas include indexing, query processing and optimization, information extraction, information integration, and cloud computing. He has given more than 100 invited talks at industry, conferences, and universities, has published more than 50 papers at world-class scientific venues, and has submitted more than 20 invention disclosures. He earned numerous prestigious awards, including the Information Society and Technology Price 2001 awarded by the European Union, an IBM Outstanding Technological Achievement Award, and the Pat Goldberg Best Paper Award.