Continuous Intersection Joins Over Moving Objects

Rui Zhang[1], Dan Lin[2], Kotagiri Ramamohanarao[3], Elisa Bertino[4]

[1,3]*Department of Computer Science and Software Engineering, University of Melbourne*
Carlton Victoria 3053, Australia
[1]rui@csse.unimelb.edu.au
[3]rao@csse.unimelb.edu.au

[2,4]*Department of Computer Science, Purdue University*
305 N. University St., West Lafayette, IN, USA
[2]lindan@cs.purdue.edu
[4]bertino@cs.purdue.edu

Abstract— **The continuous intersection join query is computationally expensive yet important for various applications on moving objects. No previous study has specifically addressed this query type. We can adopt a naive algorithm or extend an existing technique (TP-Join) to process the query. However, they compute the answer for either too long or too short a time interval, which results in either a very large computation cost per object update or too frequent answer updates, respectively. This motivates us to optimize the query processing in the time dimension. In this study, we achieve this optimization by introducing the new concept of time-constrained (TC) processing. Further, TC processing enables a set of effective improvement techniques on traditional intersection join algorithms. With a thorough experimental study, we show that our algorithm outperforms the best adapted existing solution by several orders of magnitude.**

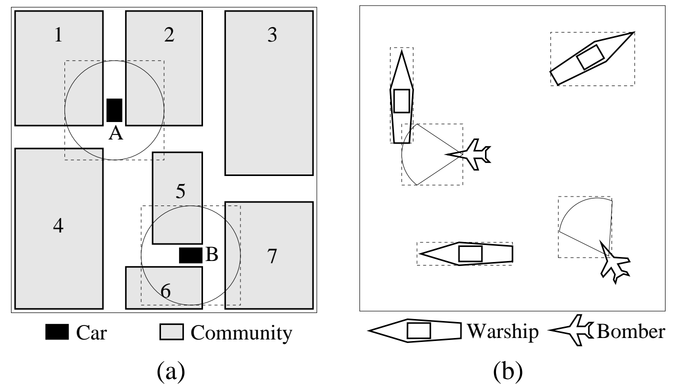


Fig. 1. Motivating examples

I. Introduction

Management of moving objects has become an imperative task recently due to the increasing need for real-time information in highly dynamic environments. In many previous studies, moving objects such as mobile phone users or vehicles have been modelled as points. The reason is that the objects' extents are negligible compared to the size of the whole region of interest. For example, ignoring the extents of vehicles does not hurt much if we want to have an idea of how many cars are in the central business district by performing a window query. However, there are also many scenarios described as follows where the extents of objects cannot be neglected. As shown in Figure 1(a), a number of police cars (filled black rectangles) are driving around in a city. Every police car can cover a circle-shaped region in case of emergency calls. We need to continuously keep track of the communities (gray rectangles) covered (that is, intersected) by each police car. For another example shown in Figure 1(b), a fleet of warships on the sea is fighting an enemy bomber squadron. The attack range of a bomber is a sector-shaped region in front of the bomber. We need to continuously report those warships whose bodies intersect any bomber's attack range so that the warships can be alerted to possible attack. In a military simulation, there can be up to 100,000 objects that are moving [1] and a primitive data management requirement is *interest management*, which is actually an intersection join of the interest ranges of objects [2], [1]. Furthermore, envision the example of Figure 1(b) in a large-scale online game where the number of fighters or players may reach hundreds of thousands, the system has to show each player possible enemy attacks all the time. These are applications representing the execution of intersection joins over moving objects (of nonzero extents) continuously.

To the best of our knowledge, no previous study has specifically addressed the *continuous intersection join query over moving objects with updates*. The only available way to support this query type is through extending a previous technique which was designed for other types of queries [3] (details are in Section III). Our experiments show that even with a small number (1,000) of objects, this extended algorithm is still too slow to produce the result in real time. In this paper, we address the problem of efficiently processing continuous intersection joins over moving objects and make the following contributions:

- Based on the key insight that the join result between any two objects only needs to be valid until the next update on any of the two objects, we propose the time-constrained (TC) processing technique for the continuous intersection join query and show how to optimize the technique. Unlike previous works, which optimize from the spatial aspects, this is the first attempt to optimize continuous spatio-temporal queries in the time dimension.
- We investigate a set of effective improvement techniques

on traditional intersection join algorithms, enabled by TC processing.
- We integrate the above techniques with carefully designed structures into a robust and scalable solution.
- We performed an extensive experimental study, which shows that our algorithm outperforms the best adapted existing solution by several orders of magnitude.

The rest of the paper is organized as follows. Section II gives the problem definition and a naive algorithm. In Section III, we extend a previous technique to support the continuous intersection join query. Then, we present out technique in Section IV have a discussion on it in Section V. Section VI reports the experimental study and Section VII reviews related work. Finally, Section VIII concludes the paper.

II. PRELIMINARIES

In this section, we define the problem and then describe TPR/TPR*-trees [4], [5] since we use them as the underlying access methods. Subsequently, we provide a naive algorithm for solving the problem.

A. Problem Formulation

We follow the common approach of representing positions of moving objects, that is, by linear functions of time. Such representations require less updates with position changes. We consider objects with nonzero extents, instead of restricting to only moving points. An object of irregular shape is represented by its *MBR (minimum bounding rectangle)*, whose sides are parallel to the axes of the 2-dimensional space[1]. Specifically, a moving object O in a 2-dimensional space is described by its MBR $\langle O_{Rx-}, O_{Rx+}, O_{Ry-}, O_{Ry+}\rangle$ ("$-$" and "$+$" stand for lower bound and upper bound, respectively) at reference time t_{ref} and its *VBR (velocity bounding rectangle)*, $\langle O_{Vx-}, O_{Vx+}, O_{Vy-}, O_{Vy+}\rangle$.

The join is performed on two moving object sets, A and B. Each object has a unique ID among all the objects in $A \cup B$. A management system maintains the information of the objects and process queries on them. With the consideration that the size of the data may be large and also in line with previous studies [6], [7], [4], [5], we have implemented our techniques assuming the data are disk resident, **although our techniques are applicable even if the data are held in main memory**. Each set of objects is indexed by a TPR-tree (actually the variant TPR*-tree) due to TPR-trees' efficient management of moving objects with nonzero extents. An update is sent to the management system when the difference between the object's actual parameters (position or velocity) and parameters maintained in the management system exceeds some threshold. Following many previous studies [8], [9], [10], if an object's actual parameters do not change for a long time, the system still requires the object to update at least once every T_M timestamps. We call T_M the *maximum update interval*, which is the longest time interval between two consecutive updates of an object. The reason for the maximum update interval is as follows. Updates not only keep the objects' movement information up to date, but also serve as heartbeat signals in practice. Without the maximum update interval requirement, if an object does not communicate with the management system for a long time, it is hard to know whether the object keeps moving in the same way or has disappeared accidentally without being able to notify the management system. T_M is a system parameter, which is the same for all objects.

Orenstein [11] suggested that an intersection join on irregular shapes should be processed in two steps: (1)*Filter Step*: Find all the object pairs whose MBRs intersect each other; (2)*Refinement Step*: For all the object pairs found in the filter step, check whether the actual shapes of the objects intersect. We focus on the filter step.

Definition 1: Let A, B be moving object sets, and let Mbr be a function that returns the MBR of an object. The ***continuous intersection join query*** *is to find every pair $\langle a, b\rangle$ for every timestamp, $a \in A, b \in B$, that satisfies $Mbr(a) \cap Mbr(b) \neq \emptyset$.*

Since the join result has to be presented all the time, we assume that it can always be held in main memory. Producing the continuous join result consists of two phases: computing the initial join pairs (*initial join*) and then maintaining the join result continuously as objects are updated (*maintenance*). The initial join is performed only once, therefore the maintenance has significantly higher weight in the total cost.

B. The TPR/TPR*-tree

We assume that the reader is already familiar with the R*-tree [12]. The TPR-tree [4] extends the R*-tree [12] by attaching time parameters to node regions so that the nodes can bound moving objects. A leaf node of a TPR-tree is a moving object whose MBR (VBR) bounds the MBRs (VBRs) of the data objects inside. A non-leaf node of a TPR-tree is a moving object that bounds inside its children, either leaf nodes or other non-leaf nodes. The TPR*-tree [5] uses a set of improved algorithms to build the TPR-tree and achieves an almost optimal tree.

C. Processing Continuous Intersection Joins Naively

Recall that processing a continuous join (we omit "intersection" when the context is clear) consists of two phases: the initial join and the maintenance. For the initial join, we can use the naive algorithm described below to compute all the possible join pairs from now to the infinite timestamp. For the maintenance, whenever there is an object update, we need to perform an *answer update* as follows. First, we remove all the pairs involving the updated object from the current result; then we join the object with the other dataset (still using the naive algorithm) from the current timestamp to the infinite timestamp and the newly found pairs are added to the current

[1]For ease of presentation, we focus on 2-dimensional spaces, though the proposed techniques are applicable to higher-dimensional spaces.

Algorithm **NaiveJoin** (N_A, N_B)
1 **for** every e_A in N_A
2 **for** every e_B in N_B with
$([t'_s, t'_e] \leftarrow intersect(e_A, e_B, t_c, \infty)) \neq$NULL
3 **if** N_A is a leaf node
4 output $\langle e_A, e_B, t'_s, t'_e \rangle$;
5 **else**
6 ReadPage($e_A.ptr$); ReadPage($e_B.ptr$);
7 NaiveJoin($e_A.ptr$,$e_B.ptr$);
End NaiveJoin

Fig. 2. Algorithm NaiveJoin

join result. Next, we give the naive algorithm for computing join pairs.

Each dataset is indexed by a TPR-tree (tr_A and tr_B for A and B, respectively). The basic idea is to use the bounding relationship between a node of the TPR-tree and the entries inside it. Let N_A (N_B) be a node from tr_A (tr_B). If N_A does not intersect N_B, then none of the entries in the subtree rooted at N_A could intersect[2] any of the entries in the subtree rooted at N_B, therefore we need not visit the subtrees. Otherwise, there could be intersections between entries in the subtrees and we should check the entries in them. This intersection-or-not checking is performed recursively on both trees in a top-down manner, until all possible intersections are explored. It is a synchronous traversal on both trees. This algorithm is named ***NaiveJoin*** and summarized in Figure 2.

The function $intersect(e_A, e_B, t_s, t_e)$ in line 2 determines whether two entries e_A and e_B intersect each other during the time interval $[t_s, t_e]$. If yes, the time interval for the intersection, $[t'_s, t'_e]$, is returned; otherwise, NULL is returned. The details of the function $intersect()$ are shown in [13]. In NaiveJoin, the time interval $[t_c, \infty)$ (∞ denoting the *infinite timestamp*) is input to the function $intersect()$ so that we find all possible join pairs in the future in one (synchronous) tree traversal.

III. Extending Time-Parameterized Joins for Continuous Joins

In this section, we extend a previous technique, the *time-parameterized join* algorithm [3] to support the continuous join query. The purpose is for us to learn from the inefficiency of the extended algorithm and to use it for comparison in the experimental study.

In [3], Tao and Papadias presented a set of spatio-temporal queries called time-parameterized (TP) queries, including the TP (intersection) join query. While the TP join query does not answer the continuous (intersection) join query directly, it can be extended to support the continuous join query. Next, we first show how a TP join query is processed, and then show how it can be extended for the continuous join query.

A TP query returns: (i) the *objects* that satisfy a certain spatial query; (ii) the *expiry time* of the result given in (i); (iii) the *event* that changes the result. That is, the answers are in the

[2]Actually the MBRs of the entries intersect each other. We omit "MBR" when the context is clear.

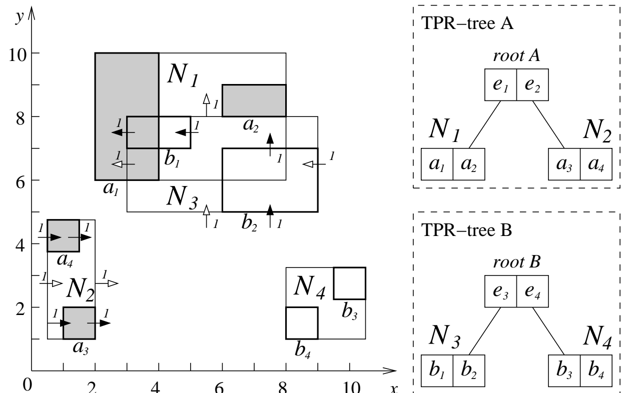


Fig. 3. A running example

format of triples, *(objects, expiry time, event)*. Figure 3 shows a TP intersection join query example. A consists of objects $\{a_1, a_2, a_3, a_4\}$ and B consists of objects $\{b_1, b_2, b_3, b_4\}$. The current result is $\{\langle a_1, b_1 \rangle\}$. Suppose the current timestamp is 0. The first result change happens at timestamp 1 when b_2 starts to intersect a_2, so the expiry time of the current result is 1 and the event causing this change is $\{\langle a_2, b_2 \rangle\}$. Therefore, the answer for the TP join query at the current timestamp is the triple $(\{\langle a_1, b_1 \rangle\}, 1, \{\langle a_2, b_2 \rangle\})$. At any timestamp, there is a "next event" that will change the result and the corresponding timestamp is called the *influence time* of the event. In this example, when a_2 intersects b_2 at timestamp 1, the next event is b_1 leaving a_1 at timestamp 3, denoted by $(\langle a_1, b_1 \rangle, 3)$ where 3 is the influence time. The subsequent events are $(\langle a_2, b_2 \rangle, 4)$, $(\langle a_3, b_4 \rangle, 6)$ and $(\langle a_3, b_4 \rangle, 8)$.

The TP join algorithm (*TP-Join*) is described as follows. Each set of objects is indexed by a TPR-tree. A depth-first (or best-first) traversal is performed on each tree synchronously starting from the root. Suppose e_A and e_B to be two entries in non-leaf nodes, one from each TPR-tree. The traversals go down the subtrees pointed by e_A and e_B if one of the following conditions hold: (i) the MBRs of e_A and e_B intersect; or (ii) $T_{INF}(e_A, e_B)$ is less than or equal to the minimum influence time of all object pairs seen so far, where $T_{INF}(e_A, e_B)$ means the influence time of the pair $\langle e_A, e_B \rangle$. Condition (i) finds the current join pairs and condition (ii) identifies the next event. The traversals stop when leaf levels are reached for both trees.

In the same paper [3], Tao et al. suggested a way to extend TP-Join to produce answers for the continuous join query. The extended algorithm *ETP-Join* is described as follows. First, TP-Join is run to obtain the current answer and the next event. As time goes to the next event and the result changes, an answer update is performed by running TP-Join to get the new next event (no need to search for the new current answer since they can be computed from the previous answer and the event).When there is an update on object O, an answer update is also performed by traversing the tree to find the object's influence time $T_{INF}(O)$. If $T_{INF}(O)$ is before the current expiry time, then $T_{INF}(O)$ becomes the current expiry time and O becomes the next event; otherwise, the update is simply

ignored (the tree already been traversed). By this means, join pairs can be obtained for all the time.

IV. Our Approach

We first analyze the NaiveJoin and ETP-Join algorithms, and then present our approach to the problem, namely time-constrained query processing.

A. Analysis

To process the join continuously, ETP-Join needs an answer update (effectively, a tree traversal) every time there is an object update or a *change in the result*. In highly dynamic environments, result changes happen frequently even if there is no object update. For the example in Figure 3, four (synchronous) tree traversals are performed during the time interval [0,5] (at timestamps 0, 1, 3, 4). Therefore, ETP-join has to perform very frequent answer updates, which causes high cost.

For NaiveJoin, answer update only has to be performed upon an object update. Therefore, the answer updates are much less frequent. However, the per-update computation cost of NaiveJoin is much higher than that of ETP-Join, the reason being the following. NaiveJoin computes all possible join pairs from now to the infinite timestamp, while a run of TP-Join will stop once the unvisited part of the tree can be pruned by the candidate event's influence time. Unless the velocities of the objects are highly skewed (e.g., all moving in the same direction), an MBR will expand in all four directions (-x, +x, -y, +y), so two MBRs must intersect sometime in the future. This causes a whole tree being accessed per answer update, which is really expensive. For the example in Figure 3, NaiveJoin obtains the same continuous join result as the ETP-Join in just one traversal, but with more node accesses and entry comparisons in the traversal. In particular, NaiveJoin compares $root\ A$ with $root\ B$, N_1 with N_3 and N_2 with N_4, while ETP-join only compares $root\ A$ with $root\ B$ and N_1 with N_3 in its first TP-join run. NaiveJoin accesses two more pages (N_2, N_4) and has more join computations (comparing the entries in N_2 with those in N_3).

On one hand, ETP-Join has a low computation cost per answer update but too frequent answer updates. On the other hand, NaiveJoin has low-frequency answer updates but too high computation cost per answer update. This contrast is even clearer if we look at the time domain. ETP-Join has to run TP-Join frequently because updates and changes of results are frequent. The problem of ETP-Join is computing the result for **too short** a time interval in each run. NaiveJoin has a high computation cost per run because it returns the answer up to the infinite timestamp. The problem of NaiveJoin is computing the result for **too long** a time interval in each run. This motivates us to optimize the query processing in the time dimension. The crux of the problem is to choose a "good" time interval for each join run. In what follows, we introduce the new concept of time-constrained (TC) processing to solve this problem.

B. Time-Constrained Processing

Our key insight is that the join result between any two objects only needs to be valid until the next update on any of the two objects. Actually, if an object issues an update, all the predictions about this object's intersection with other objects in the future may become invalid immediately. We have to perform a join between the updated object with the other dataset anyway. In other words, an update of an object invalidates the object's join result starting from the update timestamp to the future. Therefore an ideal time interval for computing join pairs for an object is from the current timestamp to the object's next-update timestamp. This ideal case is impossible in reality because we could not know in advance an object's next-update timestamp. However, fortunately we have an upper bound of an object's next-update timestamp, that is, T_M from now. T_M is the maximum update interval described in Section II-A. For an object, we only need to find its join pairs with the other dataset during the period $[t_c, t_c + T_M]$. Before $t_c + T_M$, this object will have to issue an update and we will then find its join pairs with the other dataset again for another T_M period. By this means, we can obtain correct answers for this object continuously. One question remains: while doing this on one object seems correct, can we do this on all objects and still get correct join pairs between any two objects and for all the time? Theorem 1 below gives a positive answer to the question.

Theorem 1: Let O be an object in one set and $otherset(O)$ be the set O does not belong to. Let t_u be the update (or insertion) timestamp of O. For any O, if we always process the join between O and all the objects in $otherset(O)$ for the time interval $[t_u, t_u + T_M]$ whenever there is an update (or insertion) of O, the union of all the produced join pairs is the correct answer for the continuous join query for all the time.

The proof is given in [13]. This theorem indicates that, whenever we process the join, either for the initial answer or for the updates, we only need to compute join pairs for the time interval $[t_u, t_u + T_M]$ instead of $[t_u, \infty]$. It effectively imposes a constraint on the query processing in time. Therefore we call it ***time-constrained*** (***TC***) query processing. To apply it on the NaiveJoin algorithm, we simply change $intersect(e_A$, e_B, t_c,$\infty)$ in line 2 of the algorithm to $intersect(e_A$, e_B, t_c, $t_u + T_M)$. We call the resultant algorithm ***TC-Join***.

TC-Join has the advantages of both ETP-Join and NaiveJoin, that is, it has a small computation cost per object update ($[t_u, t_u + T_M]$ is much smaller than $[t_u, \infty]$) and only needs to update the answer when there is an object update. For the example in Figure 3, suppose $T_M = 5$. During the time interval [0,5], TC-Join only performs one tree traversal; for this traversal, it only compares $root\ A$ with $root\ B$ and N_1 with N_3 (TC-Join does not access N_2 and N_4 because it knows they will not intersect in the time interval [0,5] by comparing e_2 and e_4). TC-Join is better than both ETP-Join, which has

four tree traversals, and NaiveJoin, which performs one tree traversal but with all nodes accessed. This clearly shows the benefit of TC processing.

C. Making the Most of TC Processing

Since T_M is the maximum time interval between two updates of an object, the actual time interval between two updates may be much shorter than T_M. If we consider a uniform distribution, the average update time interval between two updates is $T_M/2$. Therefore, one may ask: can we obtain better time constraint than $[t_u + T_M]$? The answer is again positive based on theorem 2 below. We reuse the notation for Theorem 1. In addition, if there is an update on any object in set Z, we say that there is an update on Z. Let $lu(Z)$ denote the latest update on Z before the current timestamp.

Theorem 2: For any O, if we always process the join between O and all the objects in $otherset(O)$ for the time interval $[t_u, t(lu(otherset(O))) + T_M]$ whenever there is an update (or insertion) of O, the union of all the produced join pairs is the correct answer for the continuous join query for all the time.

The proof is given in [13]. An example for Theorem 2 is as follows. Suppose $T_M = 5$, the current timestamp is 7, and we know that all the objects in B were updated before timestamp 4. Then for an update on A at the current timestamp, we only need to compute its join pairs with B until timestamp 9 (9=4+5), which means the processing time interval is [7,9]. This is even shorter than [7, 12] (12=7+5). $t(lu(otherset(O)))$ is the *latest update timestamp* (***lut***) of $otherset(O)$ before O is updated. The smaller the *lut*, the stricter the time constraint for processing the query. The problem is how to reduce the *lut* for a set of objects. Given a set of objects, we cannot change the *lut* of it. However, part of the set could have smaller *lut* and if we can separate them from those that have large *lut*, then we can still achieve stricter time constraint for processing that part of the set. We propose to group objects into time buckets based on their latest updates; therefore the set of objects in each time bucket (except the last one) has a smaller *lut* than that of the whole dataset. To group objects into time buckets for TPR-trees, a similar idea as used in the B^x-tree [8] can be exploited. Particularly, we divide the time axis into equi-length time buckets; for each time bucket, a TPR-tree is used to index all the objects whose latest update time fall in the bucket. This results in a group of TPR-trees based on multiple time buckets, which we call the *MTB-tree*.

To handle updates in the MTB-tree, we first identify which time bucket the object is currently stored from its last update timestamp[3]. We delete the object from the TPR-tree in that time bucket and insert it into the current TPR-tree. Typically the length of a time bucket can divide T_M exactly. Figure 4 shows an example where the length of a time bucket is $\frac{T_M}{2}$ and

[3]We assume that the last update timestamp is sent together with the update information.

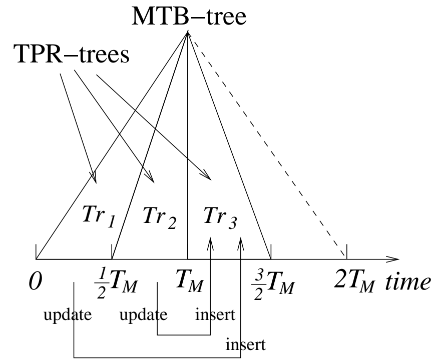


Fig. 4. The MTB-tree

the current timestamp is in the third time bucket $[T_M, \frac{3T_M}{2}]$. Updates result in deletions from Tr_1 or Tr_2 and insertions to Tr_3. Here, *lut* for the whole dataset is $\frac{3T_M}{2}$, while the *lut* for the objects in Tr_1 and Tr_2 are $\frac{T_M}{2}$ and T_M, respectively. Thereby we reduce *lut* for many objects in the set.

The continuous join is processed as follows. The initial join is still performed on two single TPR-trees. After the maintenance phase begins, we start to divide the time axis into time buckets and change the single TPR-tree into a MTB-tree. When there is an object update on A, it is first updated on the MTB-tree on A; then it is joined with the MTB-tree on B. Specifically, the object is joined with each TPR-tree of B using the TC-Join algorithm, but for an even shorter period $[t_c, t_{eb} + T_M]$, where t_{eb} denotes the end of the time bucket of the TPR-tree. Suppose the MTB-tree in Figure 4 is for B, then we join the updated object with Tr_1, Tr_2 and Tr_3 for the time interval $[t_c, \frac{3T_M}{2}]$, $[t_c, 2T_M]$ and $[t_c, \frac{5T_M}{2}]$, respectively. We call the above method ***MTB-Join***. TC-Join is a special case of MTB-Join when the whole time dimension is one time bucket.

If T_M is m times the length of a time bucket, there are at most m+1 TPR-trees in the MTB-tree. Larger m results in more TPR-trees and smaller *lut* for each tree, but also incurs more tree maintenance cost and increases the number of combinations between two joining MTB-trees. Following the rationale of the B^x-tree [8], we used $\frac{T_M}{2}$ as the length of a time bucket in our implementation.

D. Improvement Techniques

Besides cutting the workload in time dimension, TC processing enables a set of effective improvement techniques on traditional intersection join algorithms. We explore these improvement techniques below.

1) Plane Sweep: Various studies [14], [15] have shown that the plane sweep (PS) technique provides a good order of accessing two sets of rectangles and hence saves computation for processing spatial joins on static rectangles. However, no study has shown how to apply this technique to moving rectangles. The traditional PS is not applicable since the rectangles not intersecting each other at a timestamp may intersect later due to their movements. In what follows, we

will first describe PS for static rectangles and then discuss how to adapt PS to moving rectangles for a constrained time interval.

First, the two sets of rectangles are sorted respectively based on their lower left corners in a dimension, say x, to obtain two sorted sequences S_a=$\langle a_1, a_2, ...\rangle$ and S_b=$\langle b_1, b_2, ...\rangle$. Then, all the rectangles in both sequences are processed in increasing order of their x-coordinates of the lower left corner. Let c be the current rectangle to be processed. Let $e.xl$ ($e.xu$) denote the lower (upper) bound of rectangle e in dimension x. Suppose $b_1.xl < a_1.xl$, then initially c is set to b_1. The rectangles in S_a are scanned until a rectangle e with $e.xl > b_1.xu$. The scanned rectangles in S_a must overlap b_1 in dimension x, so they are further checked for overlap with c in dimension y. If any of them also overlaps x in dimension y, it is added to the join answer set. Now b_1 is done and marked as processed. c moves on to the next rectangle with the smallest xl-value in $S_a \cup S_b$, say, a_1. Then S_b is scanned and compared with c similarly as above. This process continues until a sequence is processed completely.

We find that essentially PS needs two parameters to work. A lower bound lb and an upper bound ub. lb is used to keep two sets of objects sorted in two sequences; and then they are accessed in increasing order of lb. While an object is accessed, its ub is checked against lb of the objects from the other sequence. Two objects O_1 and O_2 must not intersect if $O_1.ub < O_2.lb$. This is the fundamental requirement for choosing the two parameters. As seen from the previous sections, our join algorithm has a time constraint $[t_0, t_1]$ as part of the input. This means we need to consider the movements of the rectangles in $[t_0, t_1]$. Suppose we decide to sort in the dimension x. Let $O_{Rx-}(t)$ (or $O_{Rx+}(t)$) denote O's lower (or upper) bound at timestamp t. We can use $min(O_{Rx-}(t_0), O_{Rx-}(t_1))$ as lb and $max(O_{Rx+}(t_0), O_{Rx+}(t_1))$ as ub since they satisfy the requirement described above. Then we obtain the algorithm to compute intersections of two sets of moving objects using PS, called **PSIntersection** (details are given in [13]).

Note that the constrained processing time $[t_0, t_1]$ is necessary to enable the lower/upper bound property for PS. Otherwise, if $[t_0, \infty]$ is the time interval for processing the intersection, then we will not be able to use $max(O_{Rx+}(t_0), O_{Rx+}(t_1))$ to serve as ub because of the infinite time stamp. Further, the time constraint $[t_0, t_1]$ greatly reduces the chance of intersection and makes PS more effective than the static case.

2) Dimension Selection: We need to sort the entries (moving rectangles) before running PSIntersection. The choice of sorting dimension also has an impact on the computation cost. Consider the two examples in Figure 5. Lines 1, 2, 3 and 4 are the projections of some entries on dimension x. The dashed lines show their movements as time goes from t_0 to t_1. Line 1 corresponds to entry a_1 from node N_A; Lines 2, 3 and 4 correspond to entries b_2, b_3 and b_4, respectively, from node N_B. For Figure 5 (a), $a_1.ub > b_2.lb$, $a_1.ub < b_3.lb, b_4.lb$, therefore we only check whether a_1 intersects b_2 during PS. For Figure 5(b), $a_1.ub > b_2.lb, b_3.lb, b_4.lb$, therefore we need to check whether a_1 intersects b_2, b_3 and b_4 during PS. Suppose a_1 intersects b_2, b_3 and b_4 in dimension y. Hence a_1 actually only intersects b_2 in both cases. However, the entries in Figure 5(b) have an intersection test cost three times that of Figure 5(a). This cost difference is caused by the difference of their speed. The larger the speed, the larger the region the entry moves, and hence the greater the chance that $b_i.lb$ is smaller than $a_1.ub$, and hence the more the intersection test costs. Based on this observation, we first compute the sum of the absolute values of the speed of all entries in each dimension. Then the dimension with the smallest sum is selected as the sorting dimension.

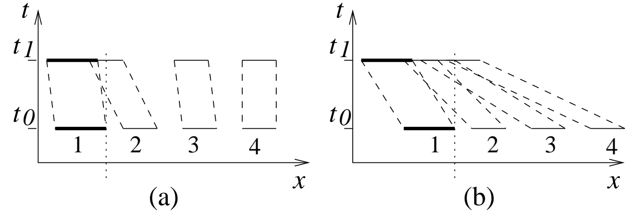


Fig. 5. Selecting sorting dimension

3) Intersection Check: Only the entries of N_A and N_B that intersect N_A.MBR$\cap N_B$.MBR could intersect each other. Therefore, before computing intersections of the entries from two nodes using PSIntersection, we first test whether the entries intersect $N_A \cap N_B$. We only run PSIntersection on entries that pass this test. This intersection check technique has been used before on static datasets [14]. Here, intersection is more effective because of the constrained processing time. Note that $N_A \cap N_B$ is a rectangle that moves in the constrained time interval $[t_0, t_1]$. Suppose they intersect during $[t_s, t_e]$. $[t_s, t_e]$ is actually an even stricter time constraint imposed on the intersection check. As we traverse the tree to a lower level, $[t_s, t_e]$ here serves as $[t_0, t_1]$ to the lower level. Because $[t_s, t_e] \subset [t_0, t_1]$, the time constraint becomes stricter and stricter. Therefore, the intersection check on moving objects have a stronger pruning power than that on static objects.

4) An Improved Join Algorithm: All the techniques presented in previous subsections are integrated into one join algorithm **ImprovedJoin**, shown in Figure 6. Compared with

Algorithm **ImprovedJoin** (N_A, N_B, t_0, t_1)
1 **for** all entries in N_A and N_B
2 Intersection check with $intersect(N_A, N_B, t_0, t_1)$,
let S_a (S_b) be the entries from N_A (N_B);
3 Determine sorting dimension;
4 sort(S_a); sort(S_b);
5 $S_c \leftarrow PSIntersection(S_a, S_b, t_s, t_e)$;
6 **for** every entry $\langle a_i, b_i, t_{si}, t_{ei}\rangle \in S_c$
7 **if** N_A is a leaf node
4 output $\langle a_i, b_i, t_{si}, t_{ei}\rangle$;
5 **else**
6 ReadPage($a_i.ptr$); ReadPage($b_i.ptr$);
7 ImprovedJoin($a_i.ptr, b_i.ptr, t_{si}, t_{ei}$);
End ImprovedJoin

Fig. 6. Algorithm ImprovedJoin

NaiveJoin, ImprovedJoin takes two additional parameters t_0 and t_1, which reflect the constrained processing time. First, we perform the intersection check. $[t_s, t_e]$ is returned as the time interval during which N_A intersects N_B. We can calculate the sum of the absolute values of the speed at the same time as the intersection check. Therefore we can avoid accessing the entries again for selecting the sorting dimension. After sorting dimension selected, we sort both sequences of entries and perform PS to obtain join pairs.

V. Discussions

TC processing can be applied to a wide range of continuous query types on moving objects such as continuous window queries and kNN queries. Take continuous window queries as an example. It is essentially computing the intersection between objects and query windows. Again, a naive algorithm would compute the intersection for the time interval $[t_c, \infty]$. We can apply the TC processing technique and only compute the intersection for $[t_c, t_c + T_M]$. Further, we can index the objects by a MTB-tree and use even tighter time constraints for each TPR-tree as we do in MTB-Join. Similarly, we can imagine applying TC processing to other queries and may enable other algorithmic improvements.

TC processing can also be easily grafted onto many existing continuous query algorithms on moving objects. This is because previous studies have focused on how to improve algorithms in the spatial aspects. Our work is the first attempt to optimize the processing in an **orthogonal** aspect, the time dimension. For example, the continuous kNN algorithm in [16] needs to compute kNN candidates for a time interval $[t_s, t_e]$ as traversing a TPR-tree. If $t_e > t_s + T_M$, we can apply TC processing and reduce the time interval to $[t_s, t_s + T_M]$. The continuous kNN and range join algorithms in [17] put all events in a queue and process them one by one. We can apply TC processing here and only process events that happen in $[t_c, t_c + T_M]$. More generally, TC processing can be applied to any continuous query algorithm as long as the data objects get updated and we can find an upper bound for the update time.

VI. Experimental Study

In this section, we report the results of our experimental study. First, we evaluate the impact of TC processing and the impact of the improvement techniques on join algorithms independently in Sections VI-B and VI-C, respectively. Then, we compare the overall performance of our techniques for the continuous intersection join with the naive algorithm, NaiveJoin, and the best possible competitor, ETP-Join, in Section VI-D.

A. Experimental Setup

All the experiments were conducted on a desktop with 2.6GHz Pentium IV CPU and 1GB RAM. The disk page size is 4K bytes, and an LRU buffer with 50 pages is used (this buffer size is suggested by [3]). We measure both the number of disk I/Os and CPU time.

Due to limited availability of real datasets of moving objects, we used the data generator developed by the authors of [4] to generate synthetic datasets with space domain of 1000×1000. We perform joins on two datasets with the same cardinality ranging from 1K to 100K. Objects are of square shape. We use the following three types of datasets: (i)*Uniform dataset*, where object positions and moving directions are generated randomly according to a uniform distribution; the speed of the objects is randomly distributed between 0 and the maximum object speed. Five maximum speeds 1, 2, 3, 4, 5 are used. (ii)*Gaussian dataset*, where object positions follow the Gaussian distribution. The speed of the objects are generated as in (i). (iii) *Battlefield dataset*, where objects of two datasets are first clustered on opposite sides of the space and then move toward the opposing party, simulating the scenario of a battlefield. By default, we use the uniform dataset.

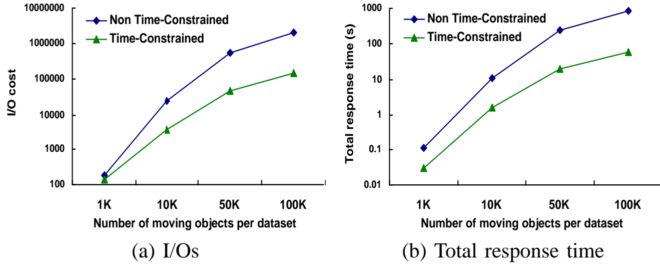
We use the TPR*-tree[5] as the underlying access method. For each dataset, we build a TPR*-tree at timestamp 0, and then keep updating it as follows. At every timestamp, we randomly change directions or speed of some objects to generate updates. Every object is required to be updated at least once during the maximum update interval T_M. The continuous join processing starts from timestamp 0. The parameters used in the experiments are summarized in Table I, where values in bold denote default values used.

Parameter	Setting
Node capacity	113
Maximum update interval	**60**, 120, 240
Maximum object speed	**1**, 2, 3, 4, 5
Object size (% of space)	**0.5%**, 0.1%, 0.2%, 0.4%, 0.8%
Dataset size	1K, **10K**, 50K, 100K
Dataset	**Uniform**, Gaussian, Battlefi eld

TABLE I
PARAMETERS AND THEIR SETTINGS

B. Effect of TC Processing

To evaluate the impact of imposing time constraints on query processing, we do not use any join improvement techniques presented in Section IV-D. Figure 7 shows the performance for the initial join computation with and without imposing time constraints. The one denoted as "Non Time-constrained" computes all possible join pairs from t_c to the infinite timestamp, which is NaiveJoin. The "Time-constrained" version computes join pairs for only the time interval $[0, 60]$. MTB-Join uses a single tree before getting the initial result, so it corresponds to the "Time-constrained" join in this figure. We observe that both the I/O cost and total response time of NaiveJoin are much higher (up to 15 times) than those of MTB-Join, which clearly shows the huge benefit we gain from TC processing. NaiveJoin performs worst mainly because it returns join pairs from the current timestamp to the infinite timestamp. Every node in one index overlaps with almost all nodes in the other index in some future time. For maintenance, the join processing is almost the same as the initial join, but



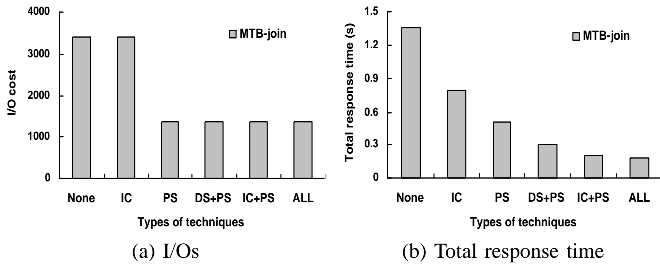
(a) I/Os (b) Total response time

Fig. 7. Effect of TC processing

on a smaller number of objects (the updated objects), so the impact of TC processing is very similar. The experiments on other settings (such as different data distributions, object speed) also give similar results, and hence we omit them here.

C. Effect of Improvement Techniques on Joins

In this section, we examine the impact of the improvement techniques on join algorithms independently of the effect of TC processing. We use the same time interval $[0, 60]$ for all techniques so that the time constraint does not have an effect on the relative performance. Figure 8 shows the join performance when we use different combinations of the three techniques: PS, DS(dimension selection) and IC(Intersection Check). "None" means using none of the techniques and "All" means all techniques are used.



(a) I/Os (b) Total response time

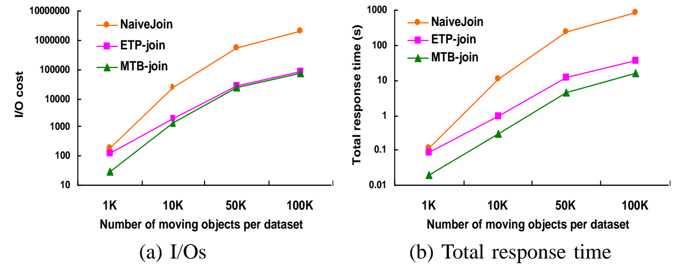
Fig. 8. Effect of improvement techniques

From Figure 8(b), we observe that the total response time decreases as more and more techniques are applied. From Figure 8(a), we find that only PS reduces I/O cost (about 60% compared to the algorithm using none of the techniques) while the other techniques only affect total response time time. When all techniques are applied, the total response time is improved by the factor of about 6. Such behavior can be explained as follows. PS provides a better order for comparing nodes in two trees, which saves both I/O and CPU costs. DS and IC mainly reduce the CPU time since both of them aim at reducing number of entries to be compared in two nodes. Specifically, DS chooses the dimension that needs less intersection comparisons for entries in two nodes. IC provides both space and time constraints to prune entries to be compared. This is also the reason why "IC+PS" improves the performance more than "DS+PS" does. Again, the impact of these techniques on maintenance cost follow similar behavior and hence we omit them here.

D. Overall Performance Comparison

We now compare our technique, MTB-join (using the ImprovedJoin algorithm in Section VI-C) with NaiveJoin (Section II-C) and ETP-Join (Section III) by evaluating two phases of the continuous join processing: initial join and maintenance.

1) Initial Join: We compare the initial join computation cost of the three approaches by varying the dataset size, data distribution, object speed and object size, respectively. When we vary one parameter, the other parameters are set to default values.



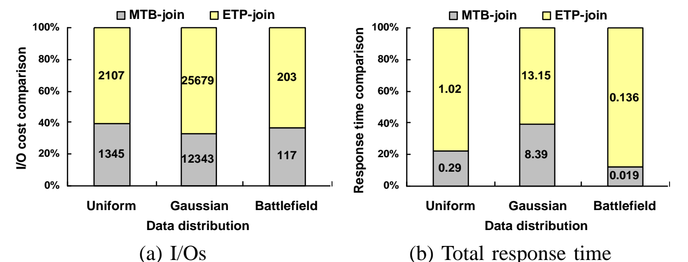
(a) I/Os (b) Total response time

Fig. 9. Initial join cost when varying dataset size

Figure 9 shows the effect of varying the dataset size. We observe that NaiveJoin has extremely high cost compared to MTB-Join and ETP-Join, and the gap between their total response time increases rapidly as dataset size increases. When the dataset size is 100K, the initial join time of NaiveJoin is about half an hour, which is intolerable. Also, NaiveJoin is much worse than MTB-Join in maintenance because NaiveJoin does not use any improvement technique and needs to compute the join to the infinite timestamp for each updated object. Due to such an uncompetitive fact of NaiveJoin, we do not consider it in the remaining experiments. Compared to Figure 7, here MTB-Join performs far better than NaiveJoin because of the use of all the improvement techniques in MTB-Join.

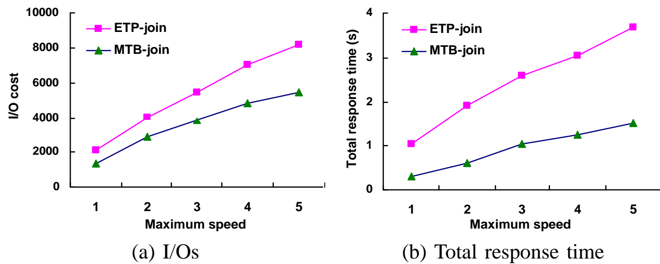
It is interesting to see that the total response time of MTB-Join is still much less (please note the logarithmic scale) than that of ETP-Join even though MTB-Join may need to compute join results for a longer time interval in each tree traversal. In particular, MTB-Join outperforms ETP-Join by up to 4 times in both I/O cost and total response time, which is mainly due to the improvement techniques on join algorithms.

Figure 10 shows the effect of the data distribution, where we can see that MTB-Join is superior to ETP-Join for all



(a) I/Os (b) Total response time

Fig. 10. Varying the data distribution

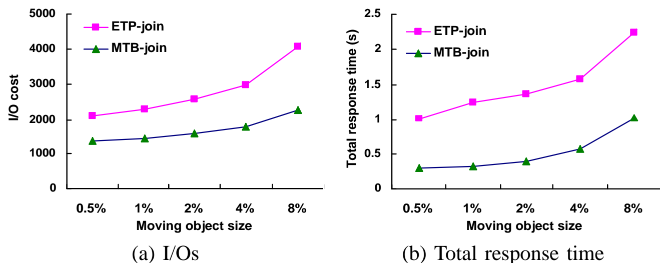


(a) I/Os (b) Total response time

Fig. 11. Varying the maximum object speed

three types of data distributions. MTB-Join saves about half of the I/O cost compared to ETP-Join for each case, and the total response time saving is even higher (up to 86% for the battlefield dataset). These improvements are again attributed to the improvement techniques on join algorithms.

The results of the experiments where we vary the maximum object speed and the object size are shown in Figures 11 and 12, respectively. MTB-Join outperforms ETP-Join in all cases for the same reasons as stated above.

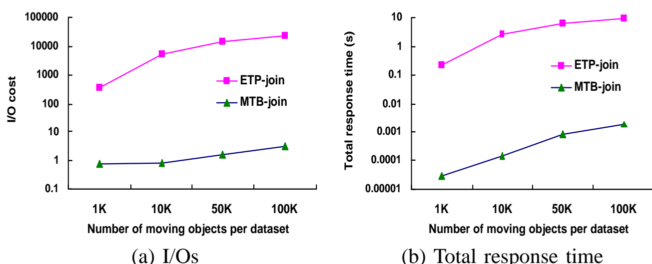


(a) I/Os (b) Total response time

Fig. 12. Varying moving object sizes

2) Maintenance: The maintenance cost is amortized by the number of updates at each timestamp. In all the subsequent experiments, we start measuring the average maintenance cost from timestamp T_M, assuming the timestamp for the initial join is 0.

Figure 13 shows the average maintenance cost per update during [60, 240](by default, T_M=60) when varying dataset size. Observe that MTB-Join achieves significant improvement over ETP-Join in terms of both I/Os and total response time (10^4 times!) and the gap between them increases with the dataset size. Further, we observe that even for very small



(a) I/Os (b) Total response time

Fig. 13. Maintenance cost with the effect of dataset sizes

datasets (1K objects), the per-update response time of ETP-Join is quite large (0.26 second). Since each object is updated at least once during the maximum update interval (60 timestamps), the number of updates at each timestamp is at least 33 for the two 1K datasets. Thus, the total time required to process these updates by ETP-Join is 8.6 seconds for each timestamp. Considering the capability of human perception, 0.1 second may be a preferable choice for a timestamp [1]. Then ETP-Join is far inferior and is unable to produce the result in time. Even if the two datasets are held in main memory, ETP-Join still needs at least 6.3 seconds to produce the result for one timestamp. As for our algorithm MTB-Join, it only takes about 1 millisecond to produce result at each timestamp for the 1K datasets.

The reasons for MTB-Join's huge performance gain are highly constrained processing time (through grouping objects into different time buckets) and the improvement techniques. Further, ETP-Join has to perform a synchronous traversal on the trees whenever there is a result change or an update, while MTB-Join only needs to perform constrained joins upon updates.

We also varied other parameters in the experiments such as maximum update interval, data distributions, object speed and object sizes. The results have very similar behavior and their details are given in [13]. Recall that maintenance has significantly higher weight in the total cost of a continuous join, therefore, how MTB-Join compares to ETP-Join in maintenance cost means more than their comparison in the initial join. Based on this rationale and the results above, we say that MTB-Join outperforms ETP-Join by several orders of magnitude.

VII. Related Work

Despite many efforts devoted into moving object databases, such as index structures [8], [9], [10], [18] and other continuous queries [16], [19], [20], there is little work specifically addressing continuous intersection joins over moving objects with updates. Mokbel et al. [7] use shared computation to process multiple continuous queries on moving objects. They do not address join queries, but use a join of queries to achieve shared computation. If we view the queries as a set of objects joining with the real data objects, then their algorithm is very similar to NaiveJoin in our paper. The TP-join algorithm [3] is related and discussed in Section III.

There are works on other types of joins over moving objects. Iwerks et al. [6] address continuous semi-joins over moving points; Arumugam et al. [21] address closest-point-of-approach joins over moving object histories. Both of them are quite different from our problem of intersection joins between objects with nonzero extents. The most related work is by Iwerks et al. [17]. They address continuous range joins, which can be viewed as intersection joins on circles. However, there are many cases where ranges of objects are more tightly bounded by rectangles rather than circles such as the communities, ships and attack ranges of bombers in Figure 1. Therefore, we still need to study intersection joins

on rectangular ranges. The algorithm in [17] is not directly applicable to our problem, but our TC technique can be applied to their algorithm as discussed in Section V.

There is a rich literature on traditional intersection joins [22], [15]. Most of the techniques are not applicable to continuous joins on moving objects. Brinkhoff et al. [14] investigated several techniques to reduce join cost for objects indexed in R*-trees [12]. These techniques were designed for static object indexes. Some of them such as plane sweep can be adapted to moving objects, which we have discussed in Section IV-D.

VIII. CONCLUSIONS

In this paper, we addressed the problem of processing continuous intersection joins over moving objects by introducing the time-constrained (TC) query processing technique. Instead of processing the query for an overlong time, we only process it to a time point necessary to guarantee the correctness of the result. TC processing can be further optimized by grouping objects into time buckets. We also showed a set of effective improvement techniques on traditional intersection join algorithms, enabled by TC processing. All the techniques are integrated in a single algorithm and our experimental results demonstrate the effectiveness of TC processing. Our algorithm outperforms the best adapted existing solution by several orders of magnitude, making it realistic to process continuous intersection join queries in real time.

The TC processing technique is applicable to a wide class of continuous queries and can be grafted onto many other algorithms easily.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments that improved our paper. This work is supported by the ECR Grant provided by the University of Melbourne under Proposal RMS number 600106.

REFERENCES

[1] K. L. Morse, "Interest management in large-scale distributed simulations, Tech. Rep. ICS-TR-96-27, 1996.
[2] J. S. Dahmann, R. Fujimoto, and R. M. Weatherly, "The department of defense high level architecture," in *Winter Simulation Conference*.
[3] Y. Tao and D. Papadias, "Time-parameterized queries in spatio-temporal databases," in *SIGMOD*, 2002.
[4] S. Saltenis, C. S.Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," in *SIGMOD*, 2000.
[5] Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: An optimized spatio-temporal access method for predictive queries," in *VLDB*, 2003.
[6] G. S. Iwerks, H. Samet, and K. P. Smith, "Maintenance of spatial semijoin queries on moving points." in *VLDB*, 2004.
[7] M. F. Mokbel, X. Xiong, and W. G. Aref, "Sina: Scalable incremental processing of continuous queries in spatio-temporal databases." in *SIGMOD*, 2004, pp. 623–634.
[8] C. Jensen, D. Lin, and B.C.Ooi, "Query and update efficient B^+-tree based indexing of moving objects," in *VLDB*, 2004.
[9] G. Kollios, D. Gunopulos, and V. J. Tsotras, "On indexing mobile objects," in *PODS*, 1999.
[10] J. M. Patel, Y. Chen, and V. P. Chakka, "STRIPES: An efficient index for predicted trajectories," in *SIGMOD*, 2004.
[11] J. Orenstein, "Spatial query processing in an object-oriented database system," in *SIGMOD*, 1986, pp. 326–336.
[12] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990.
[13] R. Zhang, D. Lin, R. Kotagiri, and E. Bertino, "Continuous intersection joins over moving objects. A full version of this paper," *http://www.cs.mu.oz.au/∼rui/publication/TR_mj.pdf*.
[14] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees." in *SIGMOD*, 1993.
[15] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *SIGMOD*, 1996, pp. 259–270.
[16] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest and reverse nearest neighbor queries for moving objects." *VLDB Journal*, vol. 15, no. 3, pp. 229–249, 2006.
[17] G. S. Iwerks, H. Samet, and K. P. Smith, "Maintenance of k-nn and spatial join queries on continuously moving points." *TODS*, vol. 31, no. 2, pp. 485–536, 2006.
[18] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches in query processing for moving object trajectories," in *VLDB*, 2000.
[19] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring." in *SIGMOD*, 2005.
[20] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "Motion adaptive indexing for moving continual queries over moving objects." in *CIKM*, 2004.
[21] S. Arumugam and C. Jermaine, "Closest-point-of-approach join for moving object histories." in *ICDE*, 2006.
[22] M.-L. Lo and C. V. Ravishankar, "Spatial hash-joins," in *SIGMOD*, 1996, pp. 247–258.