Noname manuscript No.
(will be inserted by the editor)

A Highly Optimized Algorithm for Continuous Intersection Join Queries over Moving Objects

Rui Zhang · Jianzhong Qi · Dan Lin · Wei Wang · Raymond Chi-Wing Wong

the date of receipt and acceptance should be inserted later

Abstract Given two sets of moving objects with non-zero extents, the continuous intersection join query reports every pair of intersecting objects, one from each of the two moving object sets, for every timestamp. This type of queries is important for a number of applications, e.g., in the multi-billion dollar computer game industry, massively multiplayer online games (MMOGs) like *World of Warcraft* need to monitor the intersection among players' attack ranges, and render players' interaction in real time. The computational cost of a straightforward algorithm or an algorithm adapted from another query type is prohibitive and answering the query in real time poses a great challenge. Those algorithms compute the query answer for either too long or too short a time interval, which results in either a very large computation cost per answer update or too frequent answer updates, respectively. This observation 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. Finally, we provide a method to find the optimal value for an important parameter required in our technique, the maximum update interval. As a result, we achieve a highly optimized algorithm for processing continuous intersection join queries on moving objects. With a thorough experimental study, we show that our algorithm outperforms the best adapted existing solution by several orders of magnitude. We also validate the accuracy of our cost model and its effectiveness in optimizing the performance.

Rui Zhang
University of Melbourne
E-mail: rui@csse.unimelb.edu.au

Jianzhong Qi
University of Melbourne
E-mail: jiqi@csse.unimelb.edu.au

Dan Lin
Missouri University of Science and Technology
E-mail: lindan@mst.edu

Wei Wang
University of New South Wales
E-mail: weiw@cse.unsw.edu.au

Raymond Chi-Wing Wong
Hong Kong University of Science and Technology
E-mail: raywong@cse.ust.hk

Keywords Spatial databases · moving objects · continuous intersection join

1 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 modeled 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 where the extents of objects cannot be neglected. For example, Fig. 1(a) describes a scenario where we monitor the movements of vessels and storms on the sea, and notify vessels of possible encounters with the storms. As shown in the figure, every vessel has an alert zone (a dotted rectangle) and there are two regions in the sea covered by storms. Navigation systems on vessels should continuously report those vessels whose alert zones are intersected by the storm regions, so that the vessels can be alerted to the possible impact. For another example shown in Fig. 1(b), in a massively multiplayer online game (MMOG), two teams of players are in a battle. Each player has a sector-shaped region in front

of her as her attack range. The MMOG server needs to continuously keep track of the intersection among players' attack ranges at about the graphics frame rate, so that combats between players can be processed and then rendered in almost real time. The high frequency of intersection result updates brings in a critical challenge to the server's performance and the immediate requirement of new query processing techniques [11,42]. The current systems only allow dozens of players and can not handle hundreds of thousands players in a battle. Military simulations have similar requirements as MMOGs do. In a military simulation, there can be up to 100,000 objects that are moving [25] and a primitive data management requirement is *interest management*, which is actually an intersection join of the interest ranges of objects [10,25].

The above applications represent the execution of *continuous intersection join query over moving objects (with nonzero extents) with updates*, which monitors two sets of moving objects and reports every pair of intersecting objects, one from each of the two sets, for every timestamp. Here, updates refer to changes in the spatial attributes (position or velocity) of the moving objects. They cause changes in the result of the query. To help understand the nature of this query, we express it in a SQL style as follows (for the MMOG example).

```
SELECT P_1.id, P_2.id, t
FROM   Player AS P_1, Player AS P_2
WHERE  Intersect(P_1.rg, P_2.rg, t)
       AND P_1.tid ≠ P_2.tid
UPDATE_RATE ur;
```

In this query, `Player` is a table of MMOG players; `id`, `tid` and `rg` denote the ID, team ID and attack range of a player; `Intersect()` is a boolean function that examines whether one player intersects another player's attack range at timestamp `t`, where `t` is a parameter that tells the timestamp when an answer update is triggered; and `UPDATE_RATE ur` gives the frequency that the query processor updates the answer. The parameter `rg` and the function `Intersect()` are supported in spatial databases to represent a region covered by a spatial object and to determine whether two spatial objects intersect each other, respectively. The parameters `t` and `ur` show that this is a continuous query, and they tell when the join results should be presented. These two parameters are not supported by the SQL language. However, in a spatial-temporal database where continuous queries were supported, `t` would be an intrinsic parameter and `ur` would be an input parameter that tells the database system how frequently the answer set should be updated. In a moving object management system, the continuous intersection join result will update very frequently. It is this frequent answer update that differs continuous intersection join query from traditional spatial join queries.

To the best of our knowledge, no previous study has specifically addressed the continuous intersection join query

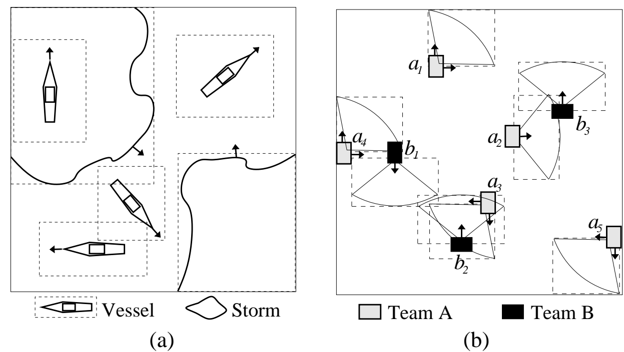


Fig. 1 Motivating examples

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 such as time-parameterized (TP) queries [39] (details are in Section 4). 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 article, we address the problem of efficiently processing continuous intersection joins over moving objects with updates 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 studies, 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 techniques to reduce the CPU cost of traditional intersection join algorithms, enabled by TC processing. We also provide a few techniques to reduce the I/O cost of the algorithms.
- We provide a model for estimating the cost of the continuous intersection join query. This model allows us to find the optimal value for an important parameter of the moving object monitoring system, the *maximum update interval*.
- We performed an extensive experimental study, which shows that our algorithm outperforms the best adapted existing solution by several orders of magnitude. We also validate the accuracy of our cost model and its effectiveness in optimizing the performance.

This article is an extended version of our earlier paper [45]. There we presented the TC processing technique and some improvement techniques to reduce CPU cost of the join algorithm. In this article, we present a few new techniques to reduce the I/O cost of the algorithms. More importantly, we model the cost of our join algorithm and thereby we can find the optimal value for the maximum update interval. As a re-

sult, we obtain a highly optimized approach to the continuous intersection join query. We have performed additional experiments. The results show (i) the effectiveness of the techniques for reducing the I/O cost of our algorithms, and (ii) the accuracy of our cost model and its effectiveness in optimizing the performance.

The rest of the article is organized as follows. Section 2 reviews related work. Section 3 gives the problem definition and a naive algorithm. In Section 4, we extend a previous technique to support the continuous intersection join query. Section 5 presents the TC processing technique and the improvement techniques for reducing CPU cost of our algorithms. Section 6 presents some techniques to reduce the I/O cost of our algorithms. In Section 7, we provide a method for finding the optimal value of the maximum update interval. Section 8 reports the experimental study and finally, Section 9 concludes the article.

2 Related Work

We discuss three categories of work below: work on moving objects in general, work on spatial joins on static objects, and work on other types of continuous spatial join queries on moving objects.

Moving objects in general: A traditional way to represent a moving object is to use its sampled locations on its trajectory. One of the earliest work that manages moving objects [27] samples object locations for each timestamp and then indexes such location data with an R-tree. Following this work, there are a few studies [34,38] proposing different index structures on sampled locations to support various queries on moving objects.

The sampling based approaches require frequent position updates, which impose a heavy workload on the system. Thus, instead of sampling and updating continuously, Sistla et al. [37] model moving objects using a linear function of time t: $P(O,t) = P(O,t_{ref}) + (t - t_{ref})V(O,t_{ref})$ where $P(O,t_{ref})$ and $V(O,t_{ref})$ are an object O's position and velocity at a reference timestamp t_{ref}. Under this representation, updates are only required when objects change their velocities. This reduces the number of updates since objects tend to move in a linear fashion for short periods. Civilis et al. [9] report that using such representation can reduce the number of updates by a factor of three for some vehicle data. Following Sistla et al.'s work [37], many studies [18,32,35, 43,40] on querying moving objects model them by linear functions of time. In this study, our problem considers moving objects with frequent updates. Thus, we model a moving object using a linear function of time.

Other than modeling moving objects, many studies on moving objects have focused on processing predictive queries. Kollios et al. [20] use the dual transform to map a line (1-dimensional trajectory) to a point and then exploit a spatial point access method to process window queries. Agarwal et al. [1] address the problem in two and higher dimensions and propose algorithms with good asymptotic performance. Zhang et al. [44] introduce the Transformed Minkowski Sum to determine whether a moving bounding rectangle intersects a moving circular query region, which enables traditional tree traversal algorithms to process window and kNN queries. While the above results are mainly theoretical, other studies aim at structures that yield good performance in practice. For example, the TPR-tree [35] is an extension of the R*-tree [4] to manage moving objects, and later, the TPR*-tree [40] enhances the TPR-tree through using a set of improved construction algorithms. The B^x-tree [18] indexes moving objects by a B^+-tree using space-filling curve (Hilbert, in particular) [6] values of the objects' positions as keys. As we use the TPR/TPR*-tree as the basic structure, we will have a closer look at them in Section 3.2. Besides predictive queries, querying historical spatio-temporal data is addressed in several studies [21,34,7]. There are also studies on continuously querying the current states of moving objects. For example, Ali et al. [2] study continuously monitoring the 3D objects around a moving object. Nutanong et al. [28] study continuous detour queries on objects moving in spatial networks. A number of other studies [15,26,29, 30] address continuous kNN queries over moving objects. A more relevant work is the time parameterized join algorithm [39], details of which are given in Section 4.

Spatial joins on static objects: Early spatial join algorithms transform objects to 1-dimensional values to avoid the difficulties in joining them because of their spatial extent and dimensionality. For example, the first known spatial join algorithm [31] indexes the objects with B^+-trees using objects' space-filling curve (z-ordering) values as the keys, and then performs the spatial join in a sort-merge join fashion.

Later, Brinkhoff et al. [8] propose the R-tree Join (RJ) algorithm and investigate techniques to improve both CPU and I/O time of the R-tree matching based spatial intersection join algorithm for objects indexed in R*-trees [4]. The two relevant improvement techniques, the plane sweep and intersection check techniques, will be discussed in Section 5.4. They were used for static object indexes like R*-trees by Brinkhoff et al.; in this paper, we analyzed them and proposed ways to take advantage of them on moving object indexes with the consideration of constrained processing time.

Studies after RJ focus on joining datasets where at least one dataset is not indexed. Lo and Ravishankar [23] propose Seeded Trees for cases where only one of the joining datasets is indexed by an R-tree apriori. A Seeded Tree is built using the existing R-tree as a skeleton, and then joined with the existing R-tree. Patel and DeWitt [33] propose the Partition Based Spatial-Merge Join (PBSM), which is a generalization of the sort merge join algorithm. The algorithm uses a rectangular grid to partition the space, and hashes objects in both joining datasets into the partitions. It then joins

objects in the same partitions using the sort merge join algorithm. Sevcik and Koudas [36] propose the Filter Trees and subsequently they propose the Size Separation Spatial Join algorithm [22] for cases where no index is available. The algorithm organizes objects hierarchically based on their sizes and assigns an object into only one partition. In the join phrase, objects in one partition are joined with objects in multiple partitions to form the join result. We assume our joining datasets are both indexed in TPR/TPR*-trees. Thus, these techniques are not applicable.

Other types of continuous spatial join queries on moving objects: Despite many efforts devoted into moving objects and spatial joins, there is little work specifically addressing continuous intersection joins over moving objects with updates. Mokbel et al. [24] use shared computation to process multiple continuous queries on moving objects. They do not address spatial 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 article (Section 3.3).

There are studies on other types of joins over moving objects. Iwerks et al. [16] address *continuous semijoins* over moving points. The semijoin on two datasets A and B is defined as all the pairs $\langle a, b\rangle, a \in A \cap b \in B$, that are in Cartesian product $A \times B$, and b is one of the k nearest neighbors of a. The k nearest neighbors of a are bounded by a circle that centers at a, which is named the *fuzzy set circle* of a. The points in the *fuzzy set circle* of a, and the points that will enter the circle sometime in the near future, form the *fuzzy set* of a. Fuzzy sets provide answers to the continuous semijoin. When there are updates in the datasets, the radii of the fuzzy set circles change, which in turn cause changes of fuzzy sets. In the continuous intersection join over moving objects with nonzero extents, we may use the fuzzy set circle of an object O to bound all the objects intersecting O. However, this circle will have a large radius, since there is no limit on the sizes of objects. A lot of objects not intersecting O will also be contained in this circle, which makes it inefficient to maintain the circle. Therefore, the fuzzy set circle based method does not apply.

Arumugam et al. [3] address closest-point-of-approach joins over moving object histories. They find the closest point pair between two historical trajectories, which is a totally different problem from the continuous intersection join.

U et al. [41] propose the exclusive closest pairs (ECP) join on point data and further address the problem of continuous monitoring ECP pairs with updates on the datasets. ECP join tries to match up the objects in two datasets A and B according to the distances between objects. In the resultant object pair set, any object in A or B will only appear in at most one pair. Thus, ECP join can first compute a set of nearest neighbors for every object in A, and then pick objects from the nearest neighbor sets one at a time to form the result set. Further, ECP join assumes small datasets (e.g., matching cars with parking slots). While the intersection join can be used to compute the nearest neighbor sets for ECP join, it does not have the above assumptions. Thus, ECP join does not solve our intersection join problem.

Iwerks et al. [17] consider continuous range joins, which can be viewed as intersection joins on circles. This is probably the closest work to ours. However, there are many cases where ranges of objects are more tightly bounded by rectangles rather than circles such as the storms, vessels and attack ranges of MMOG players in Fig. 1. Therefore, we still need to study intersection joins on rectangular ranges. The algorithms proposed by Iwerks et al. [17] work as follows. A range join is computed to get the initial result set. Events that will cause the query result to change (i.e., an object moves into or moves out of the join range of another object) are computed and enqueued in an event queue prioritized by the event time. These events are then processed to keep the join result set updated. This approach resembles the ETP join algorithm (Section 4) in that they both generate large amount of events; the difference is that this approach generates multiple events at a time, while the ETP join only generates the next event after one event is processed. As the discussion in Section 4 will show, these event based approaches are inefficient due to the huge number of events generated continuously. It is difficult for them to provide real time query results to the continuous intersection join on large datasets with frequent object updates.

3 Preliminaries

In this section, we define the problem and then describe TPR/TPR*-trees [35,40] since we use them as the underlying access methods. Subsequently, we provide a naive algorithm for solving the problem.

3.1 Problem Formulation

Representing moving objects: We follow the most popular approach of representing positions of moving objects, i.e., by linear functions of time. An object of irregular shape is represented by its *MBR (minimum bounding rectangle)*. The sides of an MBR are parallel to the axes of the 2-dimensional space[1]. The movement of an object is represented by its *VBR (velocity bounding rectangle)*. The VBR describes how each side of the object's MBR moves. At timestamp t, the MBR of a moving object O is denoted as $Mbr(O, t)$.

$$Mbr(O, t) = Mbr(O, t_{ref}) + (t - t_{ref})Vbr(O, t_{ref}),$$

where $Mbr(O, t_{ref}) = \langle O_{Rx-}, O_{Rx+}, O_{Ry-}, O_{Ry+}\rangle$ (in the subscript, "$-$" and "$+$" stand for lower bound and upper bound, respectively) is the MBR of O at a reference timestamp t_{ref} and $Vbr(O, t_{ref}) = \langle O_{Vx-}, O_{Vx+}, O_{Vy-},$

[1] We focus on 2-dimensional spaces, although the proposed techniques are applicable to higher-dimensional spaces.

$O_{Vy+}\rangle$ is the VBR of O since t_{ref}. We call this representation of O a Time-Parameterized Bounding Rectangle (TPBR) of O. Such representations require less updates with position changes since an object's VBR can usually stay unchanged for a short while (e.g., a person walking on the street or a car driving on the road).

Representing object updates: The join is performed on two moving object sets, A and B. Each object in $A \cup B$ has a unique ID. A moving object 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 [16,24,35,40], 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. It is represented in the form of (oid, Mbr, Vbr, t_u), where oid, Mbr and Vbr denote the unique ID, the new MBR and the new VBR of the updating object, respectively, and t_u denotes the timestamp when the update is issued.

Following many previous studies [18,20,32,35,40,43], 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 allowed 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 impossible 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.

Problem definition: Orenstein [31] 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.

A formal definition of the *continuous intersection join query* is given as follows.

Definition 1 Let A, B be moving object sets, t_c be the current timestamp, and $Mbr(o,t)$ be a function that returns the MBR of a moving object o at timestamp t. The *continuous intersection join query* finds every pair $\langle a, b\rangle$ for every timestamp t, $a \in A, b \in B$, $t \in [t_c, \infty)$, that satisfies $Mbr(a,t) \cap Mbr(b,t) \neq \emptyset$.

Overview of the System: Our continuous join processing system consists of two components: a *join processor* and a *result presenter* as shown in Fig. 2. The join processor is responsible for producing intermediate results in the form of (O_i, O_j, t_s, t_e), where $\langle O_i, O_j\rangle$ is an intersecting pair and $[t_s, t_e]$ is the period when this intersecting pair is valid. The intermediate results are passed to the result presenter and maintained in an intermediate result list (denoted by irl), sorted on t_s. A B$^+$-tree can be used to implement it. When there is an updated object O, we need to remove intersecting pairs involving O from irl. To facilitate finding pairs involving an object, a hash table (denoted by ht) is maintained using the object ID as the key. Every entry of the hash table contains an object ID oid and a linked list of pointers, pointing to the intermediate results involving object oid. At every timestamp t, we check every entry e from the beginning of irl to the last entry with t_s equal to t (cf.Fig. 2). If e has $t_e < t$ (this means e has expired), then e is discarded. Otherwise, e is reported as an entry in the current join result.

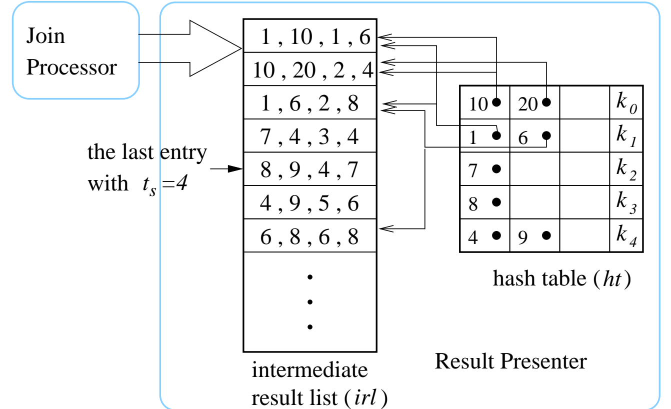


Fig. 2 System overview

Since the join result has to be presented all the time, we assume that it can always be held in main memory. Compared with the result presenter, the join processor requires much more computation. It is the focus of this paper and will be investigated in detail in following sections. Processing the continuous join 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.

As our algorithms are based on TPR-trees, we describe them before discussing the join processing algorithms.

Table 1 summarizes the frequently used symbols.

3.2 The TPR/TPR*-tree

The TPR-tree [35] extends the R*-tree [4] by attaching time parameters to node regions so that the nodes can bound moving objects. Following the popular linear function of time

Table 1 Frequently Used Symbols

Symbol	Explanation
O	A moving object
$Mbr(O,t)$	The MBR of O at timestamp t
$\langle O_{Rx-}, O_{Rx+}, O_{Ry-}, O_{Ry+}\rangle$	The MBR of O
$\langle O_{Vx-}, O_{Vx+}, O_{Vy-}, O_{Vy+}\rangle$	The VBR of O
A, B	Two moving object datasets
n_A	The number of objects in A
tr_A	A TPR-tree on A
N, e	A tree node and an entry
p_v	Voluntary update probability
C	Average per timestamp join cost
f	Average per update join cost
$SR(N,q_T)$, $A_{SR}(N,q_T)$	The region and its area swept by N during time interval q_T
t_c	Current timestamp
t_u	Update timestamp
T_M	Maximum update interval

representation, in the TPR-tree, a moving object O is represented by its MBR at reference timestamp t_{ref}, $Mbr(O,t_{ref})$, and its VBR since t_{ref}, $Vbr(O,t_{ref})$. Moving data objects are indexed in the leaf nodes of the tree. 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. Fig. 3 gives an example for a node N in a TPR-tree. Node N indexes objects a and b at timestamp 0. Its MBR bounds the two objects' MBRs, and its VBR is formed by finding the maximum and minimum speeds of the two objects on each of the dimensions (i.e., x-dimension and y-dimension).

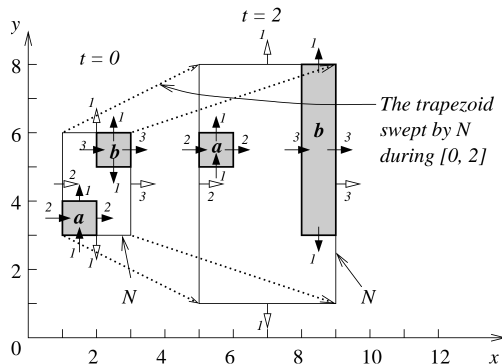


Fig. 3 A TPR-tree node

The TPR-tree supports time-slice queries as well as time-range queries, i.e., queries on the status of the indexed objects at a given timestamp and during a given time range, respectively. To illustrate how these two types of queries are processed, we use the window queries as the examples. To process a time-slice window query at timestamp t, the MBR of an object O in the TPR-tree is computed as $Mbr(O,t) = Mbr(O,t_{ref}) + (t-t_{ref})Vbr(O,t_{ref})$, and then compared with the query window to determine whether they intersect each other. To process a time-range window query with time range $[t_1,t_2]$, the region swept by an object O in the TPR-tree during $[t_1,t_2]$ is first computed (a trapezoid, cf. Fig. 3), and then compared with the query window to find the time range when they intersect each other.

The insertion, deletion and update procedures of the TPR-tree are similar to those of the R*-tree. Details are in Saltenis et al.'s paper [35]. The TPR*-tree [40] uses a set of improved algorithms to build the TPR-tree and achieves an almost optimal tree.

3.3 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 a 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 containing 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 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 sub-tree rooted at N_A could intersect[2] any of the entries in the sub-tree rooted at N_B, therefore we need not visit the sub-trees. Otherwise, there could be intersections between entries in the sub-trees 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 Fig. 4. Throughout this article, we assume that the two TPR-trees have the same height for brevity. If they do not and the traversal reaches the leaf level of one tree first, say tr_A, then we only read the node on the next level of tr_B and join this new node of tr_B with the leaf node entry of tr_A, i.e., we perform NaiveJoin(e_A,$e_B.ptr$).

The function $intersect(e_A,e_B,t_c,\infty)$ in line 2 determines whether two entries e_A and e_B (from tr_A and tr_B, respectively) intersect each other during time interval $[t_c,\infty)$, where t_c denotes the current timestamp and ∞ denotes the infinite timestamp. If yes, the time interval for the intersection, $[t'_s,t'_e]$, is returned; otherwise, `NULL` is returned. The function is based on the observation that for two objects to intersect, in every dimension, one object's upper bound must be larger than or equal to the other object's

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

```
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] ← intersect(e_A, e_B, t_c, ∞)) ≠NULL
3          if N_A is a leaf node
4              output ⟨e_A, e_B, t'_s, t'_e⟩;
5          else
6              ReadPage(e_A.ptr); ReadPage(e_B.ptr);
7              NaiveJoin(e_A.ptr, e_B.ptr);
End NaiveJoin
```

Fig. 4 Algorithm NaiveJoin

lower bound. Thus, to compute the intersection time interval for objects a and b, $intersect()$ first compute a time interval when $a(b)$'s upper bound is larger than or equal to $b(a)$'s lower bound for every dimension. Then, the intersection of all the resultant time intervals is the time interval when a and b intersect. Fig. 5 shows the details of function $intersect()$. In this function, a and b denote the

```
Function intersect (a, b, t_s, t_e )
1  [t_{1-}, t_{1+}] ← solutions of a_{R_{x+}(t)} ≥ b_{R_{x-}(t)};
2  [t_{2-}, t_{2+}] ← solutions of b_{R_{x+}(t)} ≥ a_{R_{x-}(t)};
3  [t_{3-}, t_{3+}] ← solutions of a_{R_{y+}(t)} ≥ b_{R_{y-}(t)};
4  [t_{4-}, t_{4+}] ← solutions of b_{R_{y+}(t)} ≥ a_{R_{y-}(t)};
5  [t'_s, t'_e] ← [t_s, t_e] ∩ (∩_{i=1}^{4} [t_{i-}, t_{i+}]);
6  return [t'_s, t'_e];
End intersect
```

Fig. 5 Function intersect

two objects to be checked for intersection, t_s (t_e) denotes the starting (ending) timestamp of the time interval to be checked, and the subscript $R_{x+}(t)(R_{x-}(t))$ denotes the upper (lower) bound of the MBR in dimension x at timestamp t, likewise for dimension y. Lines 1 to 4 compute the time interval when one MBR's upper bound is greater than the other's lower bound for each dimension. There is only one interval in the solution of each of these four inequalities since we assume VBRs of the objects do not change until a new update is issued. Then the time interval for MBRs' intersection is the intersection of the time intervals obtained from previous inequalities. Fig. 6 gives an example. Assuming that the current timestamp is timestamp 0, to compute

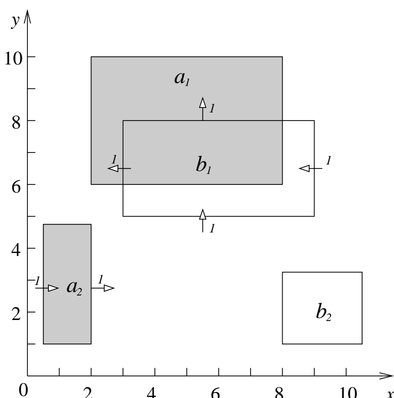


Fig. 6 Intersection time interval computation

the intersection time interval for a_1 and b_1, we first compute the time interval when $a_1(b_1)$'s upper bound is larger than or equal to $b_1(a_1)$'s lower bound for the x-dimension, which is $[0,\infty)([0,7])$. Similarly, we compute two time intervals for the y-dimension, which are $[0,5]$ and $[0,\infty)$. Subsequently, we compute the intersection time interval of a_1 and b_1, which is $[0,\infty)\cap[0,7]\cap[0,5]\cap[0,\infty)=[0,5]$.

In NaiveJoin, the time interval $[t_c,\infty)$ is input to the function $intersect()$ so that we find all possible join pairs in the future in one (synchronous) tree traversal. We call the time interval for which the algorithm needs to compute the join pairs the ***processing time interval***. Here, the processing time interval is $[t_c,\infty)$. If e_A intersects e_B and they are not leaf nodes, then the algorithm traverses to the next level of both trees synchronously, i.e., to retrieve both the pages pointed to by e_A and e_B (through pointers associated with them, $e_A.ptr$ and $e_B.ptr$, respectively).

NaiveJoin is also used for processing updates. When an object O updates, we first remove from the current answer those join pairs that contain O. Then, O is treated as a TPR-tree of a single node with a single object and joined with the other dataset (the dataset that O is not in) by NaiveJoin; this is effectively a *window query* [35] on the other dataset using O as the window and $[t_c,\infty]$ as the query time interval.

4 Extending Time-Parameterized Joins for Continuous Joins

In this section, we extend a previous technique, the *time-parameterized join* algorithm [39] to support the continuous join query. To the best of our knowledge, this is the only existing way to support the continuous intersection join query. However, since this previous technique is originally designed for some other type of queries, extending it does not result in an efficient solution to our problem. Therefore, it is still important to design algorithms that can efficiently solve our problem, and this extended technique will only serve as a baseline technique.

Tao and Papadias presented [39] 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 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 format of triples, *(objects, expiry time, event)*. Fig. 7 shows a TP intersection join query example. Moving object set A consists of objects $\{a_1,a_2,a_3,a_4\}$ and moving object set 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

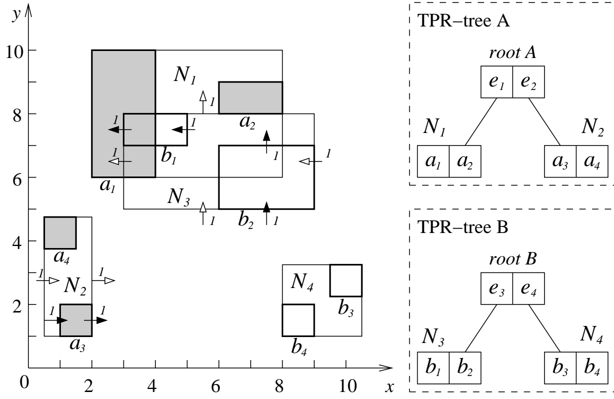


Fig. 7 A running example

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 are two entries in non-leaf nodes, one from each TPR-tree. The traversals go down the sub-trees pointed to 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. Fig. 8 summarizes the algorithm, where T_{MINF} denotes the minimum influence time of all object pairs seen so far and NE denotes the corresponding object pair (the event at T_{MINF}). The parameters T_{MINF} and NE are gradually updated during the execution of the algorithm (lines 10 and 12). When the algorithm stops, T_{MINF} becomes the time of the next event that changes the result and NE becomes the next event. In Fig. 7, at timestamp 0, e_1 intersects e_3 so nodes N_1 and N_3 are accessed. Assuming a depth-first traversal here, after all entries in N_1 are compared to those in N_3, the algorithm obtains (i) the current join pair $\langle a_1, b_1\rangle$ and (ii) the minimum influence time of all object pairs visited so far, which is 1 caused by $\langle a_2, b_2\rangle$. None of the other entry pairs from $rootA$ and $rootB$ has influence time less than 1, so the algorithm terminates the tree traversals and returns the result $(\{\langle a_1, b_1\rangle\}, 1, \{\langle a_2, b_2\rangle\})$.

Algorithm **TP-Join** (N_A, N_B, NE, T_{MINF})
1 **for** every e_A in N_A
2   **for** every e_B in N_B
3   **if** $e_A.mbr \cap e_B.mbr \neq \emptyset$
4   **if** N_A is a leaf node
5   output $\langle e_A, e_B\rangle$;
6   **else**
7   ReadPage($e_A.ptr$); ReadPage($e_B.ptr$);
8   TP-Join($e_A.ptr$,$e_B.ptr$, NE, T_{MINF});
9   **else if** $T_{INF}(e_A, e_B) < T_{MINF}$
10         $T_{MINF} \leftarrow T_{INF}(e_A, e_B)$;
11   **if** N_A is a leaf node
12             $NE \leftarrow \langle e_A, e_B\rangle$;
13   **else**
14   ReadPage($e_A.ptr$); ReadPage($e_B.ptr$);
15   TP-Join($e_A.ptr$,$e_B.ptr$, NE, T_{MINF});
End TP-Join

Fig. 8 Algorithm TP-Join

In the same paper [39], 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 has already been traversed). By this means, join pairs can be obtained for all the time. The ETP-Join algorithm is summarized in Fig. 9, where $wq()$ is a function that performs a window query on the other joining tree using the updated object O as the query window to compute the influence time of O, $T_{INF}(O)$, and the corresponding event $NE(O)$ (line 8). In Fig. 7, first the answer $\{\langle a_1, b_1\rangle\}$ is obtained for continuous join for the period $[0, 1)$ from the TP Join result at timestamp 0, $\{\langle a_1, b_1\rangle, 1, \langle a_2, b_2\rangle\}$. The continuous join answer becomes $\{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle\}$ at timestamp 1. At this moment, TP-Join is run to get the next event $(\langle a_1, b_1\rangle, 3)$. Then we know the continuous join answer during the period $[1, 3)$ keeps to be

Algorithm **ETP-Join** ($root\ A$, $root\ B$)
1 $NE \leftarrow \emptyset$; $T_{MINF} \leftarrow \infty$;
2 TP-Join($root\ A, root\ B, NE, T_{MINF}$);
3 **while** `true`
4   Wait until next answer update;
5   **if** the answer update is a change of the result
6   TP-Join($root\ A, root\ B, NE, T_{MINF}$);
7   **else** // the answer update is an object update
8         $\langle NE(O), T_{INF}(O)\rangle \leftarrow wq(O$, the other joining tree);
9   **if** $T_{INF}(O) < T_{MINF}$
10             $T_{MINF} \leftarrow T_{INF}(O)$; $NE \leftarrow NE(O)$;
End ETP-Join

Fig. 9 Algorithm ETP-Join

$\{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle\}$ and it changes to $\{\langle a_2, b_2\rangle\}$ at timestamp 3. Again, TP-join is run at timestamp 3 to find the next event and the influence time, which are $(\langle a_2, b_2\rangle)$ and 4, respectively. During period $[3, 4)$, the continuous join answer stays to be $\{\langle a_2, b_2\rangle\}$. At timestamp 4, TP-join is triggered again by the event of b_2 leaving a_2. The answers for subsequent timestamps can be obtained similarly.

5 Our Approach

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

5.1 Analysis

Given a period T, the computational cost of any of the continuous intersection join algorithms described in this paper is determined by the cost of processing an answer update and the number of answer updates in T. Here, the cost of processing an answer update is proportional to the number of nodes processed by the algorithms. Let p_x be the number of nodes processed for an answer update by algorithm X and u_x be the number of answer updates performed in algorithm X. Then, the computational cost of algorithm X is $O(p_x u_x)$. Using this notation, the computational cost of NaiveJoin and ETP-Join are denoted as $O(p_n u_n)$ and $O(p_e u_e)$, respectively.

We first compare the number of answer updates of the two algorithms. NaiveJoin needs an answer update upon every object update, while ETP-Join needs an answer update upon not only every object update but also every *change in the result*. In highly dynamic environments, result changes happen frequently. There usually are multiple result changes between two object updates. For the example in Fig. 7, ETP-Join performs four (synchronous) tree traversals during the time interval [0,5] (at timestamps 0, 1, 3, 4) for changes in the result, while NaiveJoin does not perform any traversal since there is no object update. Therefore, ETP-Join has a much larger number of answer updates, i.e., $u_e \gg u_n$.

On the other hand, the per update computational cost of NaiveJoin is much higher than that of ETP-Join. The reason is as follows. In NaiveJoin, the algorithm in Fig. 4 is executed for each answer update, which is effectively a time-range window query with the updated object O as the query window and $[t_c, \infty)$ as the processing time interval. The algorithm needs to process all the nodes that intersect O in $[t_c, \infty)$. In ETP-Join, algorithm TP-Join (a component of ETP-Join) is executed for each answer update, which performs a time-range window query using the updated object as the query window if it is triggered by an object update, or joins two TPR-trees if it is triggered by a change in the result. The processing time interval is $[t_c, T_{INF})$, where T_{INF} is the minimum influence time of all object pairs seen so far in the algorithm. If TP-Join is triggered by an object update, then it needs to process all the nodes that intersect the updated object in $[t_c, T_{INF})$. If TP-Join is triggered by a change in the result, then it needs to process all the nodes that intersect each other in $[t_c, T_{INF})$. Since ∞ is usually much larger than T_{INF}, the processing time interval of NaiveJoin is much larger than that of ETP-Join. 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 all the tree nodes being accessed per answer update for NaiveJoin (i.e., p_n is large). Therefore, $p_n \gg p_e$. For the example in Fig. 7, 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. For trees with more nodes, this difference will be much more significant.

To summarize, 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, i.e., having a too short processing time interval. 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, i.e., having a too long processing time interval. This motivates us to optimize query processing in the time domain. The crux of the problem is to choose a "good" processing time interval for each join run. Next, we introduce the concept of time-constrained (TC) processing, based on which we propose our MTB-Join algorithm to achieve a similar per update cost to that of the ETP-Join algorithm while retain the same small number of answer updates as that of the NaiveJoin algorithm.

5.2 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, i.e., T_M from now. T_M is the maximum update interval described in Section 3.1. 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 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.

Proof Omitted due to space limit. □

This theorem indicates that, whenever we process the join, either for the initial answer or for the updates, we can use the processing 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 $intersect(e_A,\, e_B,\, t_u,\, t_u + T_M)$. We call the resultant algorithm ***TC-Join***.

TC-Join has the advantages of both ETP-Join and NaiveJoin, i.e., 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 Fig. 7, 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.

5.3 Improving TC Processing with the MTB-tree

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 $\frac{T_M}{2}$. Therefore, one may ask: can we obtain better processing time interval 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.

Proof Omitted due to space limit. □

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.

Theorem 1 is a special case of Theorem 2. If there is at least one object update on $otherset(O)$ at every timestamp, we get $t(lu(otherset(O))) = t_u$. Therefore, in this case, $t_u + T_M$ is the optimal upper bound of the time intervals for which the join between O and all the objects in $otherset(O)$ is processed whenever there is an update (or insertion) of O.

The problem now 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 [18] 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*. Updates in the MTB-tree are handled as follows. When an object updates, we first identify which time bucket the object is currently indexed 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. The cost of updating an object in the MTB-tree is almost the same as the cost of updating an object in a regular TPR-tree, because even if the objects are indexed by a regular TPR-tree, an object update still involves deleting an object from the tree first and then insert the updated object. The only overhead of an MTB-tree object update compared to a TPR-tree object update is identifying the time bucket in which the updated object is currently indexed, which is done by a simple modulus operation and hence the overhead is negligible. Typically the length of a time bucket can divide T_M exactly. Fig. 10 shows an example where the

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

length of a time bucket is $\frac{T_M}{2}$ and the current timestamp t_u is in the third time bucket $[T_M, \frac{3T_M}{2}]$. Updates result in dele-

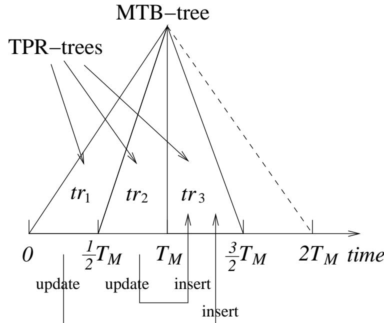


Fig. 10 The MTB-tree

tions from tr_1 or tr_2 and insertions to tr_3. Here, *lut* for the whole dataset is t_u $(t_u > T_M)$, 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 an 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 $[t_u, t_{eb} + T_M]$, where t_{eb} denotes the end of the time bucket of the TPR-tree. Since the join is between an object and a TPR-tree, it is effectively a time-range window query on the TPR-tree using the MBR of the updated object as the query window and $[t_u, t_{eb} + T_M]$ as the query time range. Suppose the MTB-tree in Fig. 10 is for B, then we join the updated object with tr_1, tr_2 and tr_3 for the time interval $[t_u, \frac{3T_M}{2}]$, $[t_u, 2T_M]$ and $[t_u, \frac{5T_M}{2}]$, respectively. We call the above method ***MTB-Join***.

We use m to denote the number of time buckets that T_M is divided into. Then there are at most m+1 TPR-trees in the MTB-tree. The choice of the value of m affects the performance of join operation, i.e., time-range window queries between updated objects and TPR-trees. A large m can reduce the cost of a single time-range window query since it reduces the processing time interval. However, it also increases the number of time-range window queries to process an update because it increases the number of TPR-trees in an MTB-tree. Also, more TPR-trees in an MTB-tree means less objects in a tree, which may lead to worse clustering of objects and hence worse performance. Due to the various complicated factors affected by the value of m and many of these factors being system dependent, it is difficulty to have an accurate theoretical model to determine an optimal value of m. Following a previous study [18], we take an empirical approach and find a suitable value of m through experiments. As shown by our experiments, $m = 2$ gives the best performance and this result accords with the result of the previous study [18].

5.4 Computational Improvements Enabled by TC Processing

Besides cutting the workload in the time dimension, TC processing enables a set of techniques that can help the traditional intersection join algorithm perform better by reducing the number of entry pairs to be checked for intersection when joining two nodes. We explore these improvement techniques below. First, we adopt the plane sweep (PS) technique, which sorts the entries with respect to their coordinates in a certain dimension and can prune some of the entry pairs from being checked for intersection according to the sorting result. Then we provide a method to choose the dimension for entry sorting based on entry speed. Adding to the PS technique, another improvement technique called Intersection Check is explored to filter the non-intersecting entries from entering the PS process, and hence fewer entry pairs are required to be checked for intersection.

5.4.1 Plane Sweep

Various studies [8, 33] 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 $c.x-$ $(c.x+)$ denote the lower (upper) bound of rectangle c in dimension x. Suppose $b_1.x- < a_1.x+$, then initially c is set to b_1. The rectangles in S_a are scanned until a rectangle e with $e.x- > b_1.x+$ is found. 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 c in dimension y, it is added to the join answer set. Now b_1 is done and marked as processed. Then, c moves on to the next rectangle with the smallest $x-$-value in $S_a \cup S_b$, say, a_1. At this time, 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. Lower bound lb is used to keep two sets of objects sorted in two sequences; and then they are accessed in increasing order of lb. While

Algorithm **PSIntersection** (S_a, S_b, t_0, t_1)
1 $i \leftarrow 1, j \leftarrow 1, S_c \leftarrow \emptyset$;
2 **while**($i \leq |S_a|$ and $j \leq |S_b|$)
3 **if** $a_i.lb < b_j.lb$
4 **while**($j \leq |S_b|$ and $b_j.lb \leq a_i.ub$)
5 **if**($[t'_s, t'_e] \leftarrow intersect(a_i, b_j, t_0, t_1) \neq$ NULL
6 Append $\langle a_i, b_j, t'_s, t'_e \rangle$ to S_c;
7 $j \leftarrow j + 1$;
8 $i \leftarrow i + 1$;
9 **else**
10 **while**($i \leq |S_a|$ and $a_i.lb \leq b_j.ub$)
11 **if**($[t'_s, t'_e] \leftarrow intersect(a_i, b_j, t_0, t_1) \neq$ NULL
12 Append $\langle a_i, b_j, t'_s, t'_e \rangle$ to S_c;
12 $i \leftarrow i + 1$;
13 $j \leftarrow j + 1$;
End PSIntersection

Fig. 11 Algorithm PSIntersection

an object is accessed, its ub is checked against lb of the objects from the other sequence. Two objects O_1 and O_2 may 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 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. The algorithm to compute intersections of two sets of moving objects using PS, called ***PSIntersection***, is presented in Fig. 11. In this algorithm, S_a (S_b) is the sequence of entries a_i (b_i) from node A (B) sorted on lb values, $[t_0, t_1]$ is the time interval the join is processed for, and S_c is a sequence to keep the join results in the output order.

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 $max(O_{Rx+}(t_0), O_{Rx+}(t_1))$ as ub because of the infinite timestamp. Further, the time constraint $[t_0, t_1]$ greatly reduces the chance of intersection and makes PS more effective than the static case.

5.4.2 Dimension Selection Based on Speed

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 Fig. 12. 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 Fig. 12 (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 Fig. 12(b), $a_1.ub > b_2.lb, b_3.lb, b_4.lb$, thus, 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 Fig. 12(b) have an intersection test cost three times that of Fig. 12(a). This cost difference is caused by the difference of their speeds. 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.

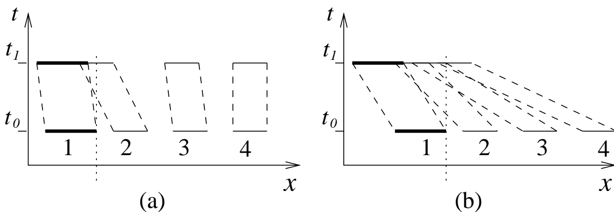


Fig. 12 Selecting sorting dimension

5.4.3 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 [8]. 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 $[t_s, t_e]$. Interval $[t_s, t_e]$ 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.

5.4.4 A Join Algorithm with the Improvement Techniques

All the techniques discussed above are integrated into one join algorithm ***ImprovedJoin***, shown in Fig. 13. Compared

Algorithm **ImprovedJoin** (N_A, N_B, t_0, t_1)
1 **for** all entries in N_A and N_B
2 Intersection check, $intersect(N_A, N_B, t_0, t_1)$,
and 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
8 output $\langle a_i, b_i, t_{si}, t_{ei} \rangle$;
9 **else**
10 ReadPage($a_i.ptr$); ReadPage($b_i.ptr$);
11 ImprovedJoin($a_i.ptr, b_i.ptr, t_{si}, t_{ei}$);
End ImprovedJoin

Fig. 13 Algorithm ImprovedJoin

with NaiveJoin, ImprovedJoin takes two additional parameters t_0 and t_1, which reflect the constrained processing time. First, we perform the intersection check. It returns $[t_s, t_e]$ as the time interval during which N_A intersects N_B. We can compute 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 the sorting dimension is selected, we sort both sequences of entries and perform PS to obtain join pairs.

5.5 Computational Cost Comparison between NaiveJoin, ETP-Join and MTB-Join

Following the discussion and notation in Section 5.1, the computational cost of MTB-Join is denoted as $O(p_m u_m)$.

The number of answer updates performed by MTB-Join is the same as that by NaiveJoin because both MTB-Join and NaiveJoin only have an answer update upon every object update, so $u_m = u_n$. As discussed in Section 5.1, $u_n \ll u_e$. Therefore, $u_m = u_n \ll u_e$.

The number of nodes processed for an object update by MTB-Join is similar to that by ETP-Join as explained below. On one hand, ETP-Join's processing time interval ($[t_c, T_{INF}]$) is smaller than that of MTB-Join ($[t_c, t_c + T_M]$) because T_{INF} is the time for the next result change, which is usually earlier than the time for the next object update. The shorter processing time interval of ETP-Join means accessing fewer tree nodes. However, on the other hand, an ETP-Join run needs to join two trees while an MTB-Join only performs a window query using the updated object as the query window. Joining two trees obviously requires accessing much more nodes than window querying one tree. The above two aspects results in similar numbers of nodes processed for an object by MTB-Join and by ETP-Join, so $p_m \approx p_e$. As discussed in Section 5.1, $p_e \ll p_n$. Therefore, $p_m \approx p_e \ll p_n$.

In summary, $u_m = u_n$ and $p_m \ll p_n$, so $O(p_m u_m) \ll O(p_n u_n)$; $p_m \approx p_e$ and $u_m \ll u_e$, so $O(p_m u_m) \ll O(p_e u_e)$. The computational cost of MTB-Join is much smaller than that of both NaiveJoin and ETP-Join. We will further validate the performance comparison in the experimental study.

5.6 Applicability of TC processing

The core idea of TC processing is that the result of a continuous query on moving objects determined by an object O only needs to be valid until O's next update. After O updates, the query result has to be updated anyway. We utilize this forced result updating property and propose that, when an object updates, we compute the result of a continuous query only to the object's next-update timestamp instead of to the infinite timestamp. Since we can not predict an object's next-update timestamp, we use the maximum update interval T_M, which is the longest time period between an object's two consecutive updates. For an object that updates at timestamp t_c, we compute whether it satisfies the query predicate 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 update the query result for another T_M period. By this means, we can obtain correct answers for the query continuously.

The above query processing procedure 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 proposed by Benetis et al. [5] 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 proposed by Iwerks et al. [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 $[t_c, t_c + T_M]$.

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.

6 Improvements on Node Access Performance

In previous sections, we have focused on techniques which improve computational efficiency of the intersection join algorithm. In this section, we present two techniques to improve node access performance. The first one provides better pruning performance during Intersection Check; the second one achieves node access reduction by processing updates in a group fashion.

6.1 Improved Node Accessing Order in Intersection Check

During Intersection Check, even if node N_A intersects N_B, it is still possible that no entry of N_A intersects any entry of N_B. For example, in Fig. 14, entries of node N_4 do not intersect $N_2.mbr \cap N_4.mbr$. Thus, we can discard this pair without accessing N_2 and hence save one node access. However, this pruning strategy dose not work if we check N_2 first.

The above pruning strategy motivates us to modify the node accessing order as follows when joining two intersect-

ing nodes (N_A, N_B). Between N_A and N_B, we always access first the one which is most recently accessed. Since this node is most recently accessed, its probability of still being in buffer is high (we consider an LRU buffer due to its popularity). Thus, we can perform Intersection Check on it without any additional I/O cost and find out whether it satisfies the pruning criterion. If it does, we then successfully avoid accessing the other node and hence save one node access.

To find out the node between N_A and N_B that is most recently accessed, we consider the process of generating intersecting node pairs by PS. Take Fig. 14 as an example. PS generates intersecting node pairs in the order of (N_1, N_4), (N_2, N_4) and (N_3, N_4). Between every two adjacent node pairs, there is a node in common. We call this node a *master node*. For example, node N_4 is a master node for the first two node pairs. Finding a master node can be easily done by comparing two adjacent node pairs. When joining a pair of nodes, we always access the master node first. We name such node accessing order the *improved accessing order*. This guarantees that the node accessed first within a pair is in the buffer (the only exception is when joining the first node pair generated by PS). By using this order in conjunction with the aforementioned pruning strategy, nodes in Fig. 14 are accessed in the order of N_1, N_4, N_4 and N_4, while without using these improvement techniques, the order is N_1, N_4, N_2, N_4, N_3 and N_4.

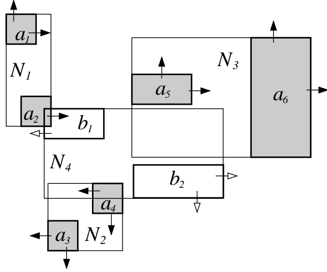


Fig. 14 Example for IOImprovedJoin

The above described improvement techniques are integrated into ImprovedJoin algorithm and result in the algorithm ***IOImprovedJoin*** (cf. Fig. 15), where the improved accessing order is achieved by putting a master node as the first parameter of a function call (lines 16 to 19) and performing Intersection Check on the first function parameter at the first place (lines 2 to 3).

6.2 Group Processing of Updates

In the phase of maintenance, we need to find new intersection pairs for updated objects, which is processed as window queries and requires traversals on the MTB-trees. To reduce the number of traversals and hence improve node access performance, we process updates in a group fashion as follows.

At every timestamp, we group all updated objects of a dataset into a set S and join it with the other joining tree. During this joining process, we recursively construct a subset S_n of S for every node N in the other joining tree and join S_n with N, where the set S_n only contains all the objects in S that intersect N. We describe the detailed process of constructing S_n in the algorithm ***GroupJoin*** as shown in Fig. 16, which is also used for the group processing of object updates. This algorithm is a modified version of IOImprovedJoin. While IOImprovedJoin is used in the phrase of initial join, GroupJoin is used in the phrase of maintenance. GroupJoin resembles IOImprovedJoin in that S_n is viewed as a tree node and joined with N. The difference is that, instead of joining an entry b_i of N with every one of its intersecting objects in S_n separately, GroupJoin constructs a subset S_n^i of S_n, which contains all the objects in S_n intersecting b_i, and then uses S_n^i as a new subset of S to join with the child node pointed to by $b_i.ptr$ (lines 8 to 18). When constructing S_n^i for b_i, we also progressively compute an MBR to bound all the objects in S_n^i, denoted as mbr_i (line 15), and an interval $[min\{t_{sj}\}, max\{t_{ej}\}]$, in which $b_i.mbr$ intersects at least one object in S_n^i (line 16). These are then used to perform Intersection Check on the child node pointed to by $b_i.ptr$. We do not perform Intersection Check on S_n since its construction process guarantees that every object in it will pass this check.

Algorithm **IOImprovedJoin** (N_A, N_B, t_0, t_1)

```
1  ReadPage(N_A);
2  for all entries in N_A
3       Intersection check, let S_a be the entries from N_A
        which intersects N_A.mbr ∩ N_B.mbr;
4  if S_a = φ return;
5  ReadPage(N_B);
6  for all entries in N_B
7       Intersection check, let S_b be the entries from N_B
        which intersects N_A.mbr ∩ N_B.mbr;
8  if S_b = φ return;
9  Determine sorting dimension;
10 sort(S_a); sort(S_b);
11 S_c ← PSIntersection(S_a, S_b, t_s, t_e);
12 for every entry ⟨a_i, b_i, t_si, t_ei⟩ ∈ S_c
13      if N_A is a leaf node
14           output ⟨a_i, b_i, t_si, t_ei⟩;
15      else
16           if a_i.ptr is a master node
17                IOImprovedJoin(a_i.ptr, b_i.ptr, t_si, t_ei);
18           elseif b_i.ptr is a master node
19                IOImprovedJoin(b_i.ptr, a_i.ptr, t_si, t_ei);
End IOImprovedJoin
```

Fig. 15 Algorithm IOImprovedJoin

7 Choosing the Maximum Update Interval

In previous sections, we have assumed that the maximum update interval, T_M, is a given parameter. In practice, a system may allow to set T_M to any value within a reasonable range based on the application requirements. In this section, we examine the problem of finding an optimal T_M value in the sense that it minimizes the average query processing cost. Towards that end, we first model the cost of our contin-

Algorithm **GroupJoin** (S_n, N, t_0, t_1, mbr)

1 ReadPage(N);
2 **for** all entries in N
3 Intersection check, let S_b be the entries from N which intersects mbr
4 **if** $S_b = \phi$ return;
5 Determine sorting dimension;
6 sort(S_n); sort(S_b);
7 $S_c \leftarrow PSIntersection(S_n, S_b, t_s, t_e)$;
8 **for** every b_i in entries of S_c
8 find a subset S'_c of S_c, which includes all entries containing b_i;
9 $S^i_n \leftarrow \emptyset$, $mbr_i \leftarrow (0, 0, 0, 0)$;
10 **for** every entry $\langle a_j, b_i, t_{sj}, t_{ej}\rangle \in S'_c$
11 **if** N is a leaf node
12 output $\langle a_j, b_i, t_{sj}, t_{ej}\rangle$;
13 **else**
14 $s^i_n \leftarrow S^i_n \cup \{a_j\}$;
15 Enlarge mbr_i with $a_j.mbr \cap b_i.mbr$ during $[t_{sj}, t_{ej}]$;
16 Update $min\{t_{sj}\}$, $max\{t_{ej}\}$;
17 **if** $S^i_n \neq \emptyset$
18 GroupJoin(S^i_n, $b_j.ptr$, $min\{t_{sj}\}$, $max\{t_{ej}\}$, mbr_i);
End GroupJoin

Fig. 16 Algorithm GroupJoin

uous intersection join algorithm for a given value of T_M, and then provide methods to find the optimal T_M value based on the cost analysis.

We use the number of node accesses to estimate the cost of query processing due to two reasons: (i) the node access cost is a significant part of the total cost; (ii) the CPU cost is roughly proportional to the number of node accesses, since the same routine will be executed on similar number of entries for every accessed node. We focus on the maintenance phase of the continuous join since this part dominates the total cost. The maintenance phase essentially deals with updates of objects (insertions and deletions can also be viewed as updates). We aim at minimizing the *average per timestamp cost*, C, which is the average cost for processing all the updates per timestamp. In what follows, we will first show that C is a function of T_M. Then we discuss how to find the optimal value of T_M.

7.1 Average per Timestamp Cost

At each timestamp, an object may need to update itself due to the change of speed or moving direction. We call this type of updates *voluntary updates*. It is reasonable to assume that the probability of performing a voluntary update is constant, since we are modeling the average behavior of a large number of objects whose movements are random and independent of each other. We denote the probability of a voluntary update as p_v. On the other hand, if an object has not updated voluntarily in the last T_M timestamps, it is forced to perform an update to satisfy the requirement of the maximum update interval. We call this type of updates *forced updates*.

Let n_A be the number of objects in tr_A and f_A the average cost of updating one object of tr_A. Consider a period of T_M timestamps. The amount of objects not updated voluntarily after T_M consecutive timestamps is $n_A(1-p_v)^{T_M}$. In other words, there are $n_A(1-p_v)^{T_M}$ forced updates during T_M. Adding the number of voluntary updates in T_M, $n_A p_v T_M$, we obtain the total amount of objects updated during T_M, $n_A p_v T_M + n_A(1-p_v)^{T_M}$. It is reasonable to assume that, after a long time, the system will reach a stable state in which the number of forced updates is distributed uniformly in each timestamp. Thus, the average cost for processing all the updated objects in tr_A per timestamp, denoted by C_A, is given by the following equation.

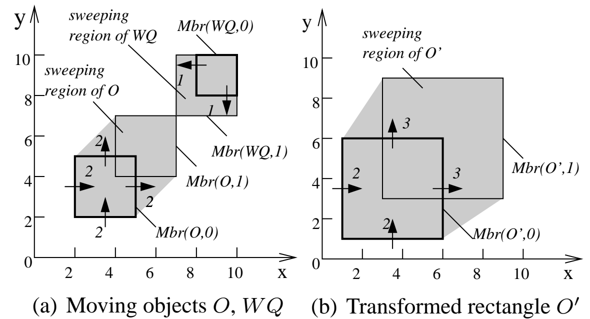
$$C_A = \left(n_A p_v + \frac{n_A(1-p_v)^{T_M}}{T_M} \right) f_A \quad (1)$$

The average cost for processing all the updated objects in tr_B per timestamp, denoted by C_B, follows a similar formula. Then we have the total average per timestamp cost $C = C_A + C_B$. We focus on how to obtain the value of C_A in the remainder of this section and C_B can be obtained in the same way.

In Equation (1), p_v is determined by parameters in real applications such as the road network and traffic conditions which do not change dramatically in a short time. Thus, we can derive p_v from the statistics on updates in a recent time window. We also know n_A. Therefore, C_A is a function of T_M multiplied by f_A. Next, we show how to derive f_A, which is also a function of T_M.

7.2 Average per Update Cost

In order to estimate f_A, we make use of the cost model for window queries on the TPR-tree [40], which is explained in Section 7.2.1. We use this cost model to estimate the cost of an individual object update. Based on this, we then show how to derive the average cost for processing all the object updates during a span of T_M timestamps in Section 7.2.2.



(a) Moving objects O, WQ (b) Transformed rectangle O'

Fig. 17 Sweeping region of moving rectangle

7.2.1 Cost Model

Consider a moving object O and a moving window query WQ for the time interval [0,1] as shown in Fig. 17(a). The *sweeping regions* of O and WQ are the regions swept by O and WQ during the time interval [0,1] (the gray regions shown in Fig. 17(a)). To determine whether object O intersects WQ, we first define the *transformed rectangle* O' with

respect to WQ as follows: the MBR of O' in the ith dimension is $\langle O_{Ri-} - |WQ_{Ri}|/2, O_{Ri+} + |WQ_{Ri}|/2 \rangle$; the VBR of O' in the ith dimension is $\langle O_{Vi-} - WQ_{Vi+}, O_{Vi+} - WQ_{Vi-} \rangle$. To check whether object O intersects WQ during the time interval [0,1] is equivalent to check whether the transformed rectangle O' intersects the center of WQ (which is a point) during the time interval [0,1]. Therefore, the probability of O intersecting WQ (which is the probability of object O being accessed by the query WQ) during the time interval [0,1] is the same as the probability of O' intersecting the center of WQ during the time interval [0,1], which equals the area of the sweeping region of O' in the time interval [0,1] (the gray region shown in Fig. 17(b)) assuming that the MBR of WQ uniformly distributes in the data space and the data space has a unit extent in each dimension. Adding up this probability for every node of the tree, we obtain the expected number of node accesses for the window query WQ as

$$\sum_{every\ node\ N\ in\ the\ tree} A_{SR}(N', q_T) \tag{2}$$

where N denotes a node in the TPR-tree; N' is the transformed rectangle of N with respect to WQ; q_T is the query time interval; $A_{SR}(N', q_T)$ is the area of the sweeping region of N' during q_T.

In the continuous join, when an object O is updated, we first remove from the current answer those join pairs that contain O. Second, O is treated as a window query with the processing time interval $[t_0, t_1]$ on the other joining dataset to find new join pairs containing O (recall that $[t_0, t_1]$ is the processing time interval used in the join algorithms). The first step, finding join pairs containing O, can be done very efficiently by looking up the object in the hash table ht in the result presenter (cf. Fig. 2). Moreover, all the join pairs in the answer are always held in main memory. In comparison, the second step is much more expensive, which involves accessing tree nodes and searching for all the new intersected objects. Therefore, we focus on the cost of the second step. As discussed above, the second step is essentially a window query, so we can use Equation (2) to estimate the cost of an individual update with q_T being $[t_0, t_1]$.

7.2.2 Update Cost for All Objects in T_M

We have shown that Equation (2) can be used to estimate the cost of an individual update. However, there are still several challenges if we want to estimate the average per update cost for all object updates.

First, the cost of an object update should be estimated using different parameters for Equation (2) at different timestamps due to the following reasons. Based on our MTB-tree scheme, the processing time interval q_T for joining the updated object with the other dataset is different at different timestamps, and the other dataset may change over time due to its own object updates at different timestamps. Therefore, we need to treat each timestamp differently. We will show later in this subsection how to derive the processing time interval q_T for a given timestamp and T_M value. To obtain the average per timestamp update cost, we compute the total update cost at each timestamp, sum it up for infinite timestamps in q_T and then divide it by the number of timestamps in q_T. Since it is infeasible to derive the update cost for *all* timestamps, we consider only T_M timestamps assuming that when the system reaches a stable state, the average update behavior will occur periodically every T_M timestamps.

A second challenge lies in how to estimate the total cost of all the updates occurring at a given timestamp. Note that to use Equation (2) to estimate the cost of an object update, we need to check whether the updated object intersects with each of the tree nodes and this requires a tree traversal. The cost of performing the check is too high if we do it on all the updated objects. To reduce this cost, we propose to estimate the cost of a set of *equivalent object updates*, which has a small cardinality and approximates the average update behavior of all the objects in terms of the number of node accesses. We will explain how to construct the equivalent object updates later in this subsection.

With the above discussion, now we can give an overview of our method of deriving the average update cost of all the objects in T_M. At every timestamp t_i in T_M, we (i) derive the q_T value given t_i and T_M, (ii) perform a tree traversal and compute the number of node accesses through Equation (2) for every update in the set of equivalent object updates, and (iii) record the average value as the average per update cost at t_i, denoted as f_i. After T_M timestamps, the average value of these average per update costs (the f_i's) is used as an approximation of f_A. The overhead of our approach is only one tree traversal at each timestamp, regardless how many updates there are. Next, we show how to derive q_T and how the set of equivalent object updates is constructed.

Deriving q_T as a function of T_M: First, we consider the case where the MTB-trees for sets A and B are single TPR-trees. As Theorem 2 shows, at a given timestamp t_u, the processing time interval for an update on object O is $[t_u, t(lu(otherset(O)))+T_M]$, i.e., $q_T = [t_u, t(lu(otherset(O))) + T_M]$. We assume large datasets and there are updates on every timestamp. Therefore, the latest update timestamp of $otherset(O)$ is the update timestamp, i.e., $t(lu(otherset(O))) = t_u$ and $q_T = [t_u, t_u + T_M]$.

Next, we derive q_T for the case where there are multiple TPR-trees in the MTB-trees for sets A and B. Following the previous notation, suppose the length of each time bucket is $\frac{T_M}{m}$ and the current time is in the $(m+1)^{th}$ time bucket as shown in Fig. 18. Suppose object O in tr_A is updated, at timestamp t_u. Among the first m TPR-trees of the MTB-tree, the ith one indexes objects updated in the time

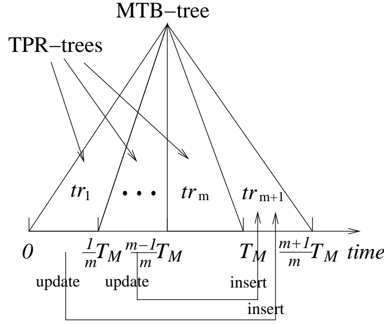


Fig. 18 An MTB-tree with n buckets

interval $(\frac{i-1}{m}T_M, \frac{i}{m}T_M]$. Hence *lut* for the ith TPR-tree is $\frac{i}{m}T_M$, and q_T on the ith sub-tree is $[t_u, \frac{i}{m}T_M + T_M]$ according to Theorem 2. The $(m+1)^{th}$ TPR-tree indexes objects updated in time interval $(T_M, t_u]$. Hence its *lut* is t_u and q_T on it is $[t_u, t_u + T_M]$. More generally, at timestamp t_u ($t_u \in (kT_M + \frac{j-1}{m}T_M, kT_M + \frac{j}{m}T_M], k, j = 1, ...m$), we can derive q_T for the ith TPR-tree among the first m TPR-trees of tr_B as $[t_u, \frac{i+j-1}{m}T_M + kT_M]$, and q_T for the $(m+1)^{th}$ TPR-tree as $[t_u, t_u + T_M]$. For example, the q_T's for tr_1, tr_2 and tr_3 in Fig. 10 are $[t_u, \frac{T_M}{2} + T_M]$, $[t_u, 2T_M]$ and $[t_u, t_u + T_M]$, respectively.

Based on the above derivation, we can see that, given a timestamp t_u, q_T is a function of T_M.

Equivalent object updates: We construct a set of equivalent object updates to approximate the average behavior of all the object updates in T_M in terms of the number of node accesses. Assume that the objects (and hence also the updates) are uniformly distributed in the data space and their velocities are also uniformly distributed within a range. A straightforward way of constructing the set of equivalent object updates is to have just one object update which is positioned at the average location (i.e., the center of the data space), has the average MBR size of all the objects, and has the VBR with the average velocity. Since the velocities are uniformly distributed, the average is 0, so the VBR of the above constructed object update is $\langle 0, 0, 0, 0 \rangle$. We call this way of constructing the set of equivalent object updates the **Zero-VBR method**. However, this method has a problem as illustrated by the example in Fig. 19. Suppose O_1 and O_2 are two object updates with the same MBR size in tr_A, and node N is in tr_B. The VBRs of O_1, O_2 and N are $\langle 0, 0, -v, -v \rangle$, $\langle 0, 0, v, v \rangle$ and $\langle 0, 0, 0, 0 \rangle$, respectively. According to the cost model, the transformed rectangles of N with respect to O_1 (O_2) is N'_1 (N'_2), which has the VBR of $\langle 0, 0, v, v \rangle$ ($\langle 0, 0, -v, -v \rangle$). If we use the Zero-VBR method to construct an equivalent object update O_3, then O_3 has the same MBR size as O_1 and O_2, and the zero VBR. The transformed rectangle of N with respect to O_3 is N'_3, which also has the zero VBR. Recall that the area of the transformed rectangle corresponds to the probability of an object update intersecting a node. We can see that the average area of N'_1

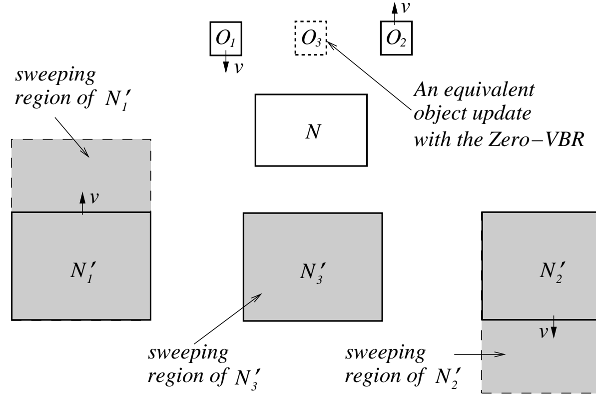


Fig. 19 Problem of the Zero-VBR method

and N'_2 is much larger than the area of N'_3. This shows that the Zero-VBR method is not accurate. The reason is that velocities of opposite directions cancel out each other in the Zero-VBR method, but actually they both add to the area of the transformed rectangles.

A lesson we learn here is that velocities in different directions should be reflected (instead of cancelled out) in the set of equivalent set of object updates. Therefore, we may consider setting the VBR of the above described equivalent object update to $\langle -v_a, v_a, -v_a, v_a \rangle$, where v_a denotes the average speed of the objects in the dataset. We call this way of constructing the set of equivalent object updates the **Expanding-MBR method**. However, this method has the following problem. The size of the updated object keeps increasing, which is not a truthful reflection of the average behavior of the object updates: actually some objects' sizes increase and some others' sizes decrease, so the average size of all the objects should stay almost unchanged.

The reason the two previous methods fail is that they try to capture the average behavior of a large number of updates by just one object, which is hard to reflect all characteristics such as size and speed at the same time. To address the problem, we propose to use eight object updates as the set of equivalent object updates. Each of these eight updates still has the average MBR size and are positioned at the center of the data space, but has a VBR different from others. The VBRs of these eight object updates are $\langle v_a, v_a, 0, 0 \rangle$, $\langle v_a, v_a, -v_a, -v_a \rangle$, $\langle 0, 0, -v_a, -v_a \rangle$, $\langle -v_a, -v_a, -v_a, -v_a \rangle$, $\langle -v_a, -v_a, 0, 0 \rangle$, $\langle -v_a, -v_a, v_a, v_a \rangle$, $\langle 0, 0, v_a, v_a \rangle$ and $\langle v_a, v_a, v_a, v_a \rangle$. We call this way of constructing the set of equivalent object updates the **Eight-VBR method**. By using this method, velocities of opposite directions all contribute to the sum area of the transformed rectangles. At the same time, their average size remains unchanged. Therefore, the Eight-VBR method addresses the problems of both of the previous methods. At every timestamp, one tree traversal is performed to evaluate Equation (2) for these eight object updates concurrently and the average of their numbers of node

accesses is used as the estimate of the per update cost for that timestamp.

7.3 Finding the Optimal Value for T_M

Revisiting Equation (1), C_A is a function of T_M multiplied by f_A. We then find that f_A is also a function of T_M in Section 7.2. Therefore, C_A is a function of T_M. However, the method of obtaining the value of f_A requires tree traversals and the result of f_A depends on the actual updates over a span of time. Consequently, we do not have a simple closed-form formula for C_A as a function of T_M. Therefore, we adopt the following empirical approach to find the optimal value of T_M. We compute the values of C_A (and C_B in a similar way) for a set of candidate T_M values at the same time, and the candidate value with the smallest $C_A + C_B$ value is chosen as the optimal T_M value. These candidate values are chosen from a range of T_M values allowable by the real system. For example, the system may require T_M to be less than 3 minutes. Then we may have $\{30, 60, 90, 120, 150, 180\}$ as the set of candidate T_M values.

7.4 Overhead

Our scheme of finding the optimal T_M value involves estimating C_A (and C_B) for a set of candidate T_M values. At every timestamp, we need to traverse tr_B (and tr_A) once to compute the value of Equation (2) for the set of (eight) equivalent object updates and perform this computation for all the candidate T_M values. Please note that evaluating Equation (2) needs the MBRs and VBRs of all the *tree nodes*, which only requires traversing the non-leaf nodes of the tree. As the number of non-leaf nodes is much smaller than the total number of the tree nodes and the non-leaf nodes usually reside in the buffer since they are frequently accessed in the query processing, the overhead of the tree traversal is actually not large. The CPU cost involved in the above process is constant for each visited node and therefore also quite limited. We have also performed experiments (see results in Section 8) which show that the cost estimation brings a negligible overhead compared to the cost of processing the intersection join query, but it leads to a good choice of the T_M, which significantly reduces the overall query processing cost.

7.5 Impacts of System Parameters

In this subsection we discuss how the system parameters impact the optimal value of T_M.

According to Equation (1), the average per timestamp cost C_A is determined by the average number of updates at a timestamp u_m, $u_m = (n_A p_v + \frac{n_A(1-p_v)^{T_M}}{T_M})$ and the average per update cost f_A. With the increase of the value of T_M, u_m will decrease because a longer maximum update interval leads to fewer forced updates, while f_A will increase because a longer processing time interval leads to larger processing cost per update. Thus, a T_M value can not minimize u_m and f_A at the same time, and the optimal T_M value is a balance between optimizing u_m and f_A. The impact of a system parameter on the optimal value of T_M is determined by whether this parameter makes u_m or f_A the dominating factor in system optimization. For example, the **object update probability** p_v has a more significant influence on u_m. If p_v is small, then a larger T_M value can reduce the number of forced updates and thus reduce u_m. Consider an extreme case where objects never change their moving directions or speed at all (e.g. all static objects), i.e., $p_v = 0$. Then, $T_M = \infty$ is optimal because the join result is not changing anyway. On the other hand, when p_v becomes larger, the optimal value of T_M gets smaller. Also consider an extreme case where objects update voluntarily at every timestamp, i.e., $p_v = 1$. In this case, $T_M = 1$ is optimal because there is no forced update. Any larger T_M value can only increase f_A but not reduce u_m. The influence of **object moving pattern** is shown by its impact on the object update probability. In a moving pattern where the objects change their moving directions or speed very frequently, (e.g. particles in Brownian motion), there are a lot of voluntary updates and thus the optimal T_M value goes smaller. In a moving pattern where objects have unified moving routes (e.g. an army marching), objects do not update much, then the optimal T_M goes larger. Unlike the above two factors, the influence of **the number of moving objects** is more complex. Increasing the number of moving objects not only increases u_m but also increases f_A because the TPR-trees will have larger number of nodes and node sizes. Therefore, in cases where changing the number of moving objects affect u_m more significantly than it affects f_A, a T_M value that optimizes u_m more can optimize the system performance better. On the other hand, if the change affects f_A more significantly, then the optimal T_M value should optimize f_A more.

8 Experimental Study

In this section, we report the results of our experimental study. First, in Section 8.2, we evaluate the impact of the number of time buckets in T_M, m, on MTB-Join, and choose a best value of m for system implementation. Then, we evaluate the impact of TC processing, computational improvement techniques and node access improvement techniques on join algorithms in Sections 8.3, 8.4 and 8.5, respectively. After that, we compare the overall performance of MTB-Join with NaiveJoin and ETP-Join in Section 8.6. We investigate the validity of our cost model, the choice of an optimal T_M value and the overhead it may incur in Section 8.7.

8.1 Experimental Setup

All the experiments were conducted on a desktop computer with 3GB RAM and 2.66GHz CPU. The disk page size is 4KB. We use the TPR*-tree [40] to index moving objects, and an LRU buffer with 50 pages is used (suggested by

Tao and Papadias [39] for TPR-trees). We measure both the number of node accesses and CPU time.

We conduct experiments on both synthetic datasets and real datasets. Synthetic datasets are generated with a space domain of 1000×1000 using the data generator developed by Saltenis et al. [35]. 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 a maximum object speed. Five maximum speeds, 1, 2, 3, 4 and 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.

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 2, where values in bold denote default values used.

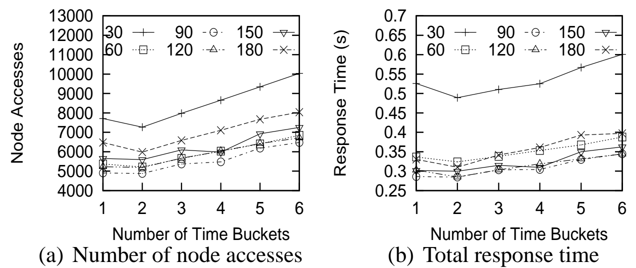
Table 2 Parameters for synthetic datasets and their settings

Parameter	Setting
Node capacity	**113**
Maximum update interval	30, **60**, 90, 120, 150, 180
Maximum object speed	**1**, 2, 3, 4, 5
Object size (% of space)	**0.5%**, 1%, 2%, 4%, 8%
Voluntary update probability	**1%**, 2%, 4%, 8%, 16%
Dataset size	1K, **10K**, 50K, 100K
Dataset	**Uniform**, Gaussian, Battlefield

We adopt two real-world trajectory datasets, a fleet of trucks and a fleet of school buses [13]. They consist of 276 and 145 trajectories, respectively. Each trajectory consist of location information for a truck/bus within a day, collected every 30 seconds. Because the number of trajectories in each dataset is small, following previous studies [12, 14, 19], we generate more objects moving on these trajectories as follows. Two groups of datasets are generated. One is based on the truck trajectories (Truck datasets), and the other is based on the bus trajectories (Bus datasets). Each dataset generated contains 10K moving objects. To generate a moving object, we first randomly pick a trajectory from the real dataset. Then, starting from a randomly picked location in the trajectory and with a randomly picked direction, a new object with the size of 0.5% (the default object size) of the space is generated to move on the trajectory. Every time the object reaches a location in the trajectory, it issues a voluntary update, i.e., the object issues a voluntary update every 30 seconds. When reaching one end of the trajectory, the object changes its direction and continues moving.

8.2 The Number of Time Buckets in T_M, m

The choice of the number of time buckets m in T_M affects the join performance. A large m can reduce the cost of a single time-range window query since it reduces the query time range. However, it also increases the number of time-range window queries to process an object update because it increases the number of TPR-trees in an MTB-tree. Also, more TPR-trees in an MTB-tree means less objects in a tree, which may lead to worse clustering of objects and hence worse performance. This experiment is to estimate a best value of m for the experimental setting considered. To observe only the effect of m values, no proposed improvement techniques or buffer pages is used in this experiment.



(a) Number of node accesses (b) Total response time

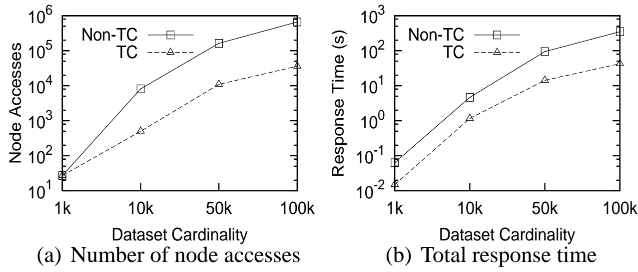
Fig. 20 Performance for varying m and T_M

Fig. 20 shows the average per timestamp cost for joining two 10K datasets with m ranging from 1 to 6 and T_M ranging from 30 to 180. From the figure, we can observe that MTB-trees with $m = 2$ perform the best at most times (only at some points, the performance of MTB-trees for $m = 1$ and $m = 2$ is similar). Therefore, $m = 2$ is used as the default setting in the following experiments.

8.3 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 5.4 or Section 6. Fig. 21 shows the performance for the initial join computation with and without imposing time constraints. The one denoted as "Non-TC" computes all possible join pairs from timestamp 0 to the infinite timestamp, which is NaiveJoin. The "TC" 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 "TC" join in this figure.

We observe that both the number of node accesses and the total response time of NaiveJoin are much higher (up to 10 times) than those of MTB-Join, which clearly shows the huge benefit we gain from TC processing. NaiveJoin performs worse 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



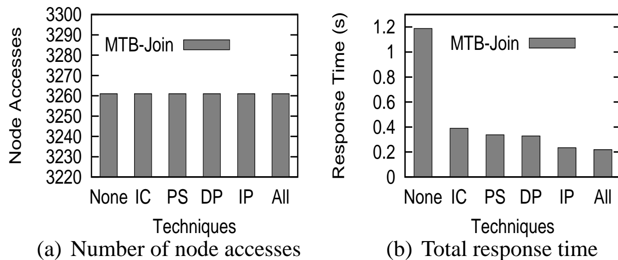
(a) Number of node accesses (b) Total response time

Fig. 21 Effect of TC processing

almost the same as the initial join, but 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, the object speed) also give similar results, and hence we omit them here.

8.4 Effect of Computational Improvements Enabled by TC Processing

In this section, we examine the impact of the computational 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. Fig. 22 shows the join performance when we use different combinations of the three techniques: PS(Plane Sweeping), DS(dimension selection) and IC(Intersection Check). "DP" means the combination of DS and PS, while "IP" means the combination of IC and PS. "None" means using none of the techniques and "All" means all techniques are used. The focus of this section is not in buffer utilization efficiency and thus no buffer is used in these experiments. From Fig. 22(a), we



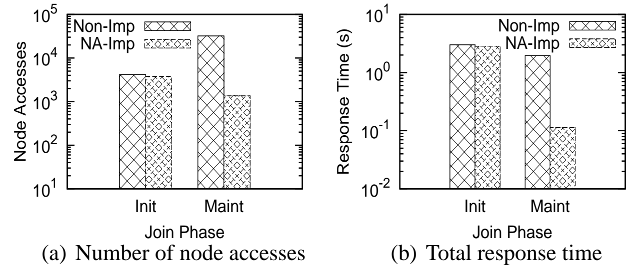
(a) Number of node accesses (b) Total response time

Fig. 22 Effect of computational improvement techniques

observe that no technique reduces the number of node accesses. From Fig. 22(b), we find that the total response time decreases as more and more techniques are applied. Therefore, we can conclude that all these techniques only affect the total response time. When all techniques are applied, the total response time is improved by the factor of about 5. Such behavior can be explained as follows. Despite PS provides a better order for comparing nodes in two trees, which saves CPU costs, it does not affect the number of intersection node pairs. Likewise, DS and IC can only reduce the CPU time since both of them aim at reducing the 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 "IP" improves the performance more than "DP" does. Again, the impact of these techniques on maintenance cost follow similar behavior and is omitted.

8.5 Effect of Improvement Techniques on Node Access Performance

The impact of the improvement techniques on node access performance is examined in this section. We join two 50K datasets using $T_M = 60$ and maintain the join result for 360 timestamps, during which updates are processed. MTB-join algorithms with and without node access improvement techniques are used independently to process the join and their performance in both phase of initial join and maintenance are presented in Fig. 23, where "NA-Imp" and "Non-Imp" means using and not using node access improvement techniques, respectively. From this figure, we observe that



(a) Number of node accesses (b) Total response time

Fig. 23 Effect of node access performance improvement techniques

the number of node accesses and the total response time decreases as node access improvement techniques are applied. In the phase of maintenance, group update processing results in a significant decrease in both the number of node accesses and the response time. Compared to that of group update processing, the effect of improvement on Intersection Check is relatively small (please note the logarithmic scale; this technique still saves about 10% of initial join cost). This is because in a dataset of uniformly distributed objects, the probability for two intersecting nodes to have no entry intersecting the intersection area of these two nodes is small.

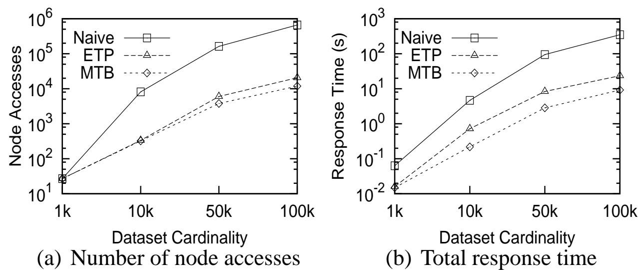
8.6 Overall Performance Comparison

We now compare our technique, MTB-join (using all improvement techniques) with NaiveJoin (Section 3.3) and ETP-Join (Section 4) by evaluating two phases of the continuous join processing: initial join and maintenance.

8.6.1 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.

Fig. 24 shows the effect of varying the dataset size. We observe that NaiveJoin has extremely high cost compared

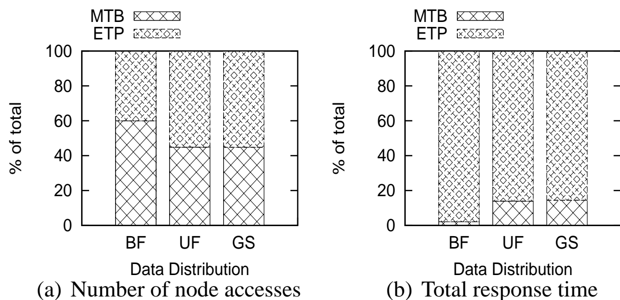


(a) Number of node accesses (b) Total response time

Fig. 24 Initial join cost when varying dataset size

to MTB-Join and ETP-Join, and the gap between their total response time increases as dataset size increases. When the dataset size is 100K, the initial join time of NaiveJoin is about 6 minutes, which is intolerable. Due to such an uncompetitive fact of NaiveJoin, we do not consider it in the remaining experiments of the phase of initial join on synthetic datasets. Compared to Fig. 21, here MTB-Join performs far better than NaiveJoin because of the use of all the improvement techniques in MTB-Join.

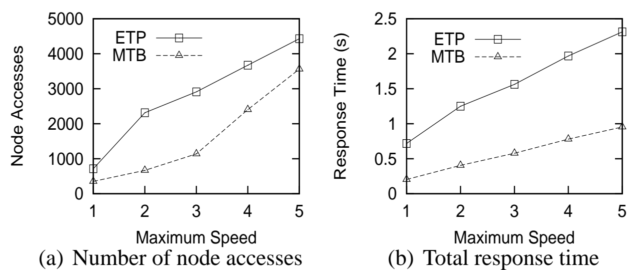
It is interesting to see that the total response time of MTB-Join is still much less (please note the logarithmic scale) than that of ETP-Join even though MTB-Join may need to compute join results for a longer time interval in each tree traversal. In particular, MTB-Join outperforms ETP-Join by up to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



(a) Number of node accesses (b) Total response time

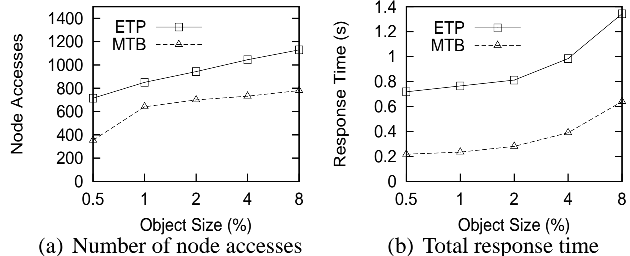
Fig. 25 Initial join cost when varying data distribution

Fig. 25 shows the effect of the data distribution. "BF", "UF" and "GS" represent using the Battlefield datasets, Uniform datasets and Gaussian datasets, respectively. We can see that MTB-Join is superior to ETP-Join for datasets of all the three types of distribution in terms of total response time. The total response time saving is high (up to 90% for the battlefield dataset). These improvements are again attributed to the improvement techniques on join algorithms. ETP-Join shows better performance in the number of node accesses for battlefield datasets. This is mainly because for battlefield datasets, experiment starts with objects of the two datasets clustering on opposite sides of the space, which means the first condition for ETP-Join's traversal to continue is not satisfied. Thus, the traversal ends quickly. Even in this extreme case, MTB-Join's node access efficiency is close to ETP-Join and MTB-Join still has a much better total response time due to the computational improvement techniques.



(a) Number of node accesses (b) Total response time

Fig. 26 Initial join cost when varying the maximum object speed

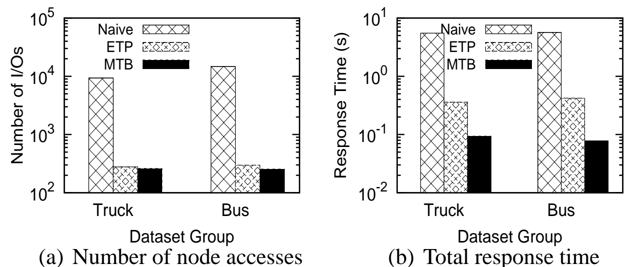


(a) Number of node accesses (b) Total response time

Fig. 27 Initial join cost when varying moving object sizes

The results of the experiments where we vary the maximum object speed and the object size are shown in Fig. 26 and Fig. 27, respectively. MTB-Join outperforms ETP-Join in all cases for the same reasons as stated above.

We also conduct initial join experiments on real datasets, and the results shown in Fig. 28 confirm that MTB-Join shows better performance than NaiveJoin or ETP-Join does (please note the logarithmic scale).



(a) Number of node accesses (b) Total response time

Fig. 28 Initial join cost when using different real datasets

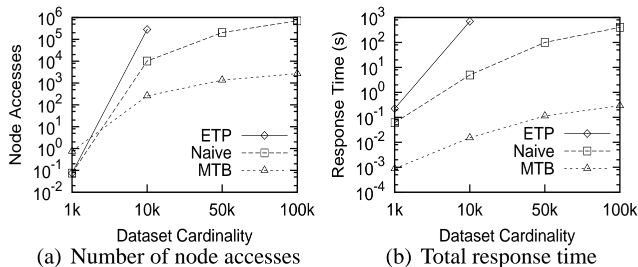
8.6.2 Maintenance

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 that the timestamp for the initial join is 0. The intension is to wait for the TPR-trees for the first few time buckets to be built up so that we are comparing a fully functioning MTB-Join with NaiveJoin and ETP-Join.

The first set of experiments evaluates the performance with respect to dataset size. Fig. 29 shows the average maintenance cost per timestamp during [60, 360] (by default, T_M=60). NaiveJoin and ETP-Join have smaller numbers of node accesses for 1K datasets because their tree nodes are all buffered while MTB-Join keeps removing and creating tree nodes so that the nodes can not be entirely buffered. Even so,

MTB-Join has a smaller response time. Other than this special case, MTB-Join achieves significant improvement over NaiveJoin and ETP-Join in terms of both the number of node accesses and the total response time. The gap among them increases with the increase of dataset size.

Further, we observe that even for very small datasets (1K objects), the per-timestamp response time of ETP-Join is not small (0.23 seconds). Considering the capability of human perception, 0.1 seconds may be a preferable choice for a timestamp [25]. Then ETP-Join is far inferior and is unable to produce the result in time. What's more, the response time of ETP-Join grows so dramatically with the increase of dataset size that it is unable to be measured accurately. Thus, there is no experimental result presented for ETP-Join using 50K or 100K datasets. For NaiveJoin, though it can produce the result at each timestamp for 1K datasets within about 0.07 second, its processing time also rises rapidly with the increase of dataset size. It requires about 5 seconds for 10K datasets, which is not acceptable. As for MTB-Join, it only takes about 0.9 milliseconds to produce the join result at each timestamp for 1K datasets. Even for 100K datasets, the processing time is only about 0.3 seconds. With some upgrade in hardware and slightly longer T_M, it is still realistic for MTB-Join to produce the result in real time. Therefore, we reach the following conclusion. While it is impossible to obtain the continuous join result in real time using NaiveJoin or ETP-Join, MTB-Join makes this difficult task realistic, even for large 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.

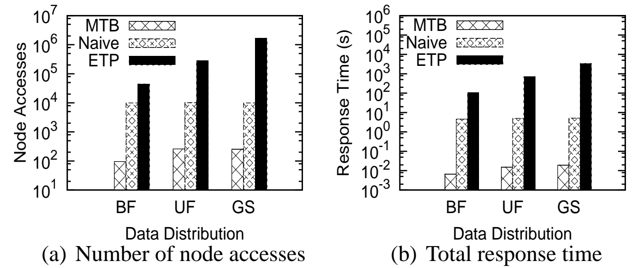


(a) Number of node accesses (b) Total response time

Fig. 29 Maintenance cost with the effect of dataset sizes

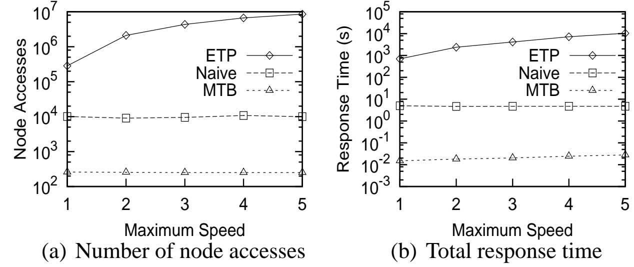
We varied other parameters in the experiments such as data distribution, maximum object speed, object size and voluntary update probability. We also conduct experiments on real datasets. The results of all these experiments show very similar behavior, as shown in Fig. 30, 31, 32, 33, and 34.

Recall that maintenance has significantly higher weight in the total cost of a continuous join. Therefore, how MTB-Join compares to NaiveJoin and ETP-Join in maintenance cost means more than their comparison in initial join. Based on this rationale and the results above, we say that MTB-Join outperforms NaiveJoin and ETP-Join by several orders of magnitude.



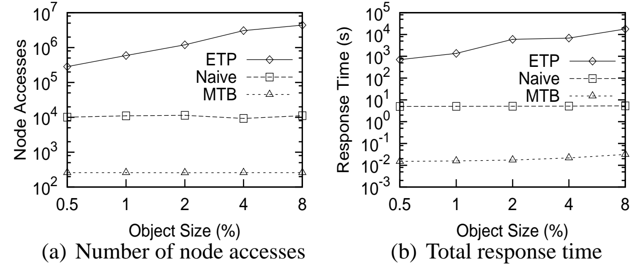
(a) Number of node accesses (b) Total response time

Fig. 30 Maintenance cost with the effect of data distribution



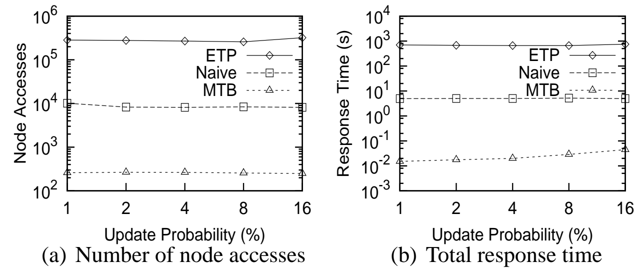
(a) Number of node accesses (b) Total response time

Fig. 31 Maintenance cost with the effect of maximum object speed



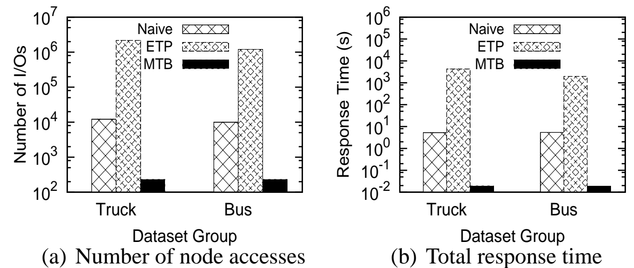
(a) Number of node accesses (b) Total response time

Fig. 32 Maintenance cost with the effect of object sizes



(a) Number of node accesses (b) Total response time

Fig. 33 Maintenance cost with the effect of voluntary update probability



(a) Number of node accesses (b) Total response time

Fig. 34 Maintenance cost with different real datasets

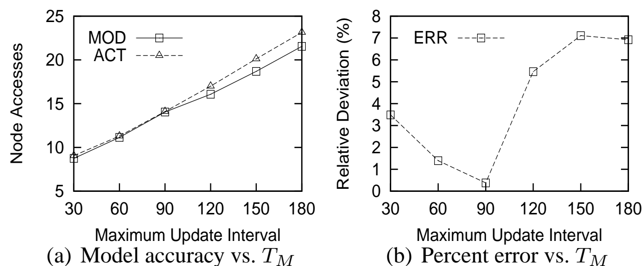
8.7 The Optimal T_M Value

In this subsection, we empirically verify our cost model for the join process and evaluate its effectiveness for finding the optimal value of T_M.

8.7.1 Cost Model Validation

To validate the cost model presented in Section 7.2.1, we measure the average value of per update cost estimated by Equation (2). We denote this value as f and compare it with the average value of actual per update cost, which is denoted by af. This means, when an update is issued, we compute a cost value with Equation (2). Meanwhile, we record the number of actual node accesses for updating the intersection result set. We sum these cost values up during a maximum update interval and then compute two average values f and af. The percent error between f and af, which is denoted by pe, $pe = \frac{|f-af|}{af}$, is presented.

To observe the stable state of the system, we collect data during time interval $[T_M+1, 2T_M]$. Fig. 35(a) shows the experimental results for running the system under different T_M values. In the figure, "MOD" means the model estimated per update cost values (f) and "ACT" means the actual per update cost values (af). It shows that, with the same datasets, f and af have very close values. They are both rising with the increase of the T_M value. Fig. 35(b) is the values of pe when varying the value of T_M. We can see that pe is less than 8%, which demonstrates the validity of the cost model.



(a) Model accuracy vs. T_M (b) Percent error vs. T_M

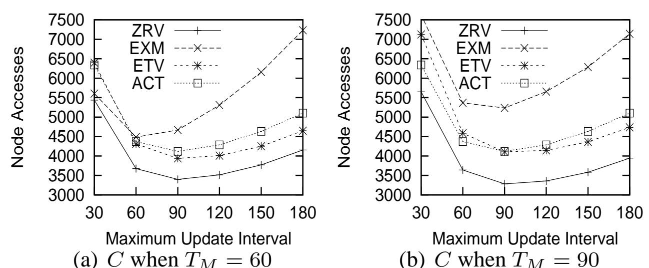
Fig. 35 Verification of the cost model

8.7.2 Finding the Optimal T_M Value

We implement the approach proposed in Section 7 to find an optimal T_M value among a set of candidate values $\{30, 60, 90, 120, 150, 180\}$. For each candidate value t_M, we run our test system once as follows. We initiate the test system by setting the current T_M value to be t_M. While the system is running, at every timestamp t_i, we compute the estimated per update cost, f_i, for each of the candidate values. To observe the stable state of the system, we start collecting these cost data at timestamp T_M and continue for T_M timestamps. After that, we will have an average value of estimated per update cost for each candidate value. Since we also know the average number of updates per timestamp, we then can compute an average value of estimated per timestamp cost, C, for every candidate value. The candidate value yielding the minimum estimated per timestamp cost will be chosen as the optimal T_M value.

Experimental results for our test system running on different T_M values are similar. Therefore, we only present two typical ones here. Fig. 36(a) is the experimental result of running the system with T_M value being 60, and Fig. 36(b) is that of setting T_M value to be 90. In these figures, "ZRV", "EXM" and "ETV" stand for computing the estimated per timestamp cost with equivalent object updates defined by the Zero-VBR method, the Expanding-MBR method and the Eight-VBR method, respectively. "ACT" means the average values of actual per timestamp cost. These values are derived directly from the average values of actual per update cost recorded in the experiments of the last subsection.

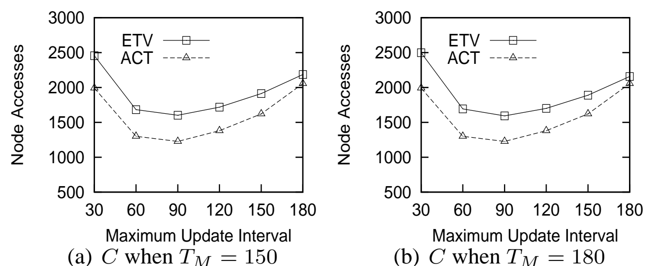
Both figures show that the equivalent object updates defined by the Eight-VBR method provide a best estimation accuracy. They also show that under our experimental settings, the T_M optimization approach will suggest T_M to be 90, which is the actual optimal T_M value, as shown by the "ACT" curve.



(a) C when $T_M = 60$ (b) C when $T_M = 90$

Fig. 36 The optimal T_M value for uniformly distributed datasets

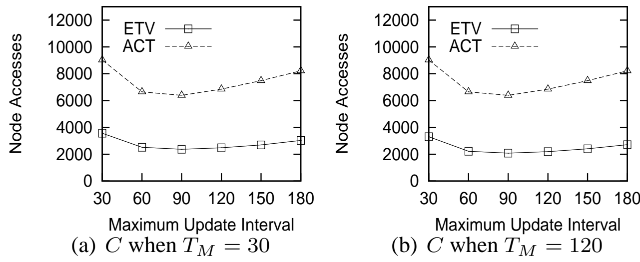
We have further performed experiments on non-uniformly distributed datasets. Fig. 37, 38, 39 and 40 show the comparison between the actual numbers of node accesses and the estimated numbers of node accesses using the Eight-VBR method, for the battlefield datasets, Gaussian datasets, Tuck datasets and Bus datasets, respectively.



(a) C when $T_M = 150$ (b) C when $T_M = 180$

Fig. 37 The optimal T_M value for battlefield datasets

We notice that in these results, the gaps between the actual values and the estimated values are larger than those of experiments on uniform datasets. This is because the distribution and the moving pattern of the objects in these datasets do not follow the assumption of the cost model. However, we also observe that the trends of the curves ascending/descending are the same, and the optimal T_M value chosen using our

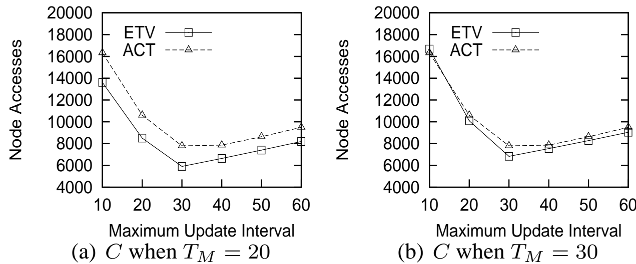


(a) C when $T_M = 30$ (b) C when $T_M = 120$

Fig. 38 The optimal T_M value for Gaussian datasets

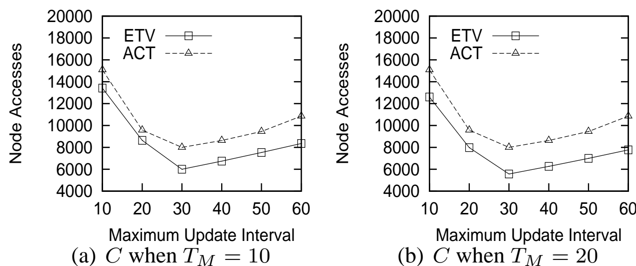
cost model still matches the actual optimal T_M values for these datasets of different distributions due to the same trend.

Note that in the real dataset experiments, we use T_M values ranging from 10 to 60. This is because the real dataset objects are issuing voluntary updates every 30 timestamps. If we only use T_M values that are larger than 30, then there will be no forced updates and thus a larger T_M value will always result in larger per timestamp update cost. Therefore, we need to test our system performance with T_M values that are smaller than 30 to see whether we can find a value that has better performance than 30 has.



(a) C when $T_M = 20$ (b) C when $T_M = 30$

Fig. 39 The optimal T_M value for the Truck datasets



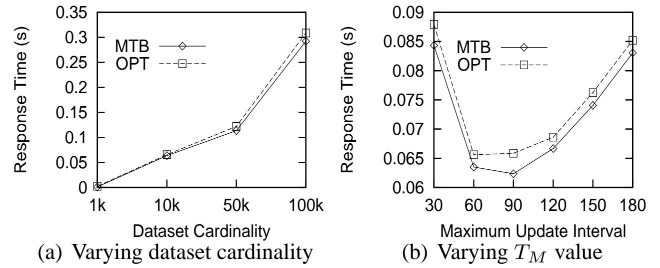
(a) C when $T_M = 10$ (b) C when $T_M = 20$

Fig. 40 The optimal T_M value for Bus datasets

8.7.3 Overhead of Finding the Optimal Value for T_M

We measure the response time overhead of our approach for finding the optimal T_M value. In the following experiments, we record and compare the average per timestamp response time of the test system with and without the T_M optimization process. The comparison results are shown in Fig. 41, where "OPT" and "MTB" denote the response time of the test system with and without the T_M optimization process, respectively.

Fig. 41(a) presents the overhead of the T_M optimization process performed on datasets of different cardinalities; we observe that this overhead is almost negligible compare to the cost of processing the join query. For example, the over-



(a) Varying dataset cardinality (b) Varying T_M value

Fig. 41 Total response time overhead

head for the 100K datasets is 0.016 seconds, which is about 5% of the time for processing the join query. Fig. 41(b) presents the overhead for the 10K datasets while varying the T_M value that the test system is running with. For all the T_M values used, the overhead is less than 0.004 seconds, which is less than 6% of the time for processing the join query. We compare this overhead with the performance gain that a well chosen T_M value can bring. From the experimental results of the last subsection, we can see that for the 10K datasets (cf. Fig. 36(a), the "ACT"curve), if the test system runs with a randomly chosen T_M value, say, 30, the average number of node accesses per timestamp is about 54% larger than that of running the system with the optimal T_M value, 90. In terms of response time, the T_M value optimization process brings a much larger performance gain compared to the overhead. Therefore, optimizing the T_M value is worthwhile.

9 Conclusions

In this article, 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 techniques enabled by TC processing to reduce the CPU cost of traditional intersection join algorithms and a few techniques to reduce the I/O cost of the algorithms. All these techniques are integrated together. Moreover, we derived a cost model for the continuous intersection join query, and showed that it can accurately predict the cost of processing the query and suggest optimal T_M values for moving object monitoring systems. We also performed an extensive experimental study. The results show the effectiveness of TC processing and the various improvement techniques enabled by it. 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 experiments also validates the accuracy of our cost model and its usefulness in choosing the optimal T_M value for our algorithm, which may provide significant

performance gain compared to using a badly chosen value.

Acknowledgments

This work is supported by the Australian Research Council's Discovery funding scheme (project numbers DP0880250 and DP0880215).

References

1. P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *PODS*, pages 175–186, 2000.
2. M. E. Ali, R. Zhang, E. Tanin, and L. Kulik. A motion-aware approach to continuous retrieval of 3d objects. In *ICDE*, pages 843–852, 2008.
3. S. Arumugam and C. Jermaine. Closest-point-of-approach join for moving object histories. In *ICDE*, page 86, 2006.
4. 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.
5. R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB Journal*, 15(3):229–249, 2006.
6. T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, 15:658–664, 1969.
7. V. Botea, D. Mallett, M. Nascimento, and J. Sander. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12:143–168, 2008.
8. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.
9. A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. In *MobiQuitous*, pages 164–173, 2004.
10. J. S. Dahmann, R. Fujimoto, and R. M. Weatherly. The department of defense high level architecture. In *Winter Simulation Conference*, pages 142–149, 1997.
11. A. J. Demers, J. Gehrke, C. Koch, B. Sowell, and W. M. White. Database research in computer games. In *SIGMOD*, pages 1011–1014, 2009.
12. H. Ding, G. Trajcevski, and P. Scheuermann. Omcat: optimal maintenance of continuous queries' answers for trajectories. In *SIGMOD*, pages 748–750, 2006.
13. E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Nearest neighbor search on moving object trajectories. In *SSTD*, pages 328–345, 2005.
14. R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB Journal*, 19(5):687–714, 2010.
15. G. S. Iwerks, H. Samet, and K. P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.
16. G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of spatial semijoin queries on moving points. In *VLDB*, pages 828–839, 2004.
17. G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of k-nn and spatial join queries on continuously moving points. *TODS*, 31(2):485–536, 2006.
18. C. Jensen, D. Lin, and B.C.Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.
19. H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *Proc. VLDB Endow.*, 1(1):1068–1080, 2008.
20. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *PODS*, pages 261–272, 1999.
21. G. Kollios, V. J. Tsotras, D. G., A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *TKDE*, 13(5):758–777, 2001.
22. N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.
23. M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, pages 209–220, 1994.
24. M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.
25. K. L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, 1996.
26. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.
27. M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *SAC*, pages 235–240, 1998.
28. S. Nutanong, E. Tanin, J. Shao, R. Zhang, and K. Ramamohanarao. Continuous detour queries in spatial networks. *To appear in TKDE*.
29. S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The V*-diagram: a query-dependent approach to moving knn queries. *Proc. VLDB Endow.*, 1(1):1095–1106, 2008.
30. S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. Analysis and evaluation of V*-kNN: an efficient algorithm for moving knn queries. *The VLDB Journal*, 19:307–332, June 2010.
31. J. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD*, pages 326–336, 1986.
32. J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An efficient index for predicted trajectories. In *SIGMOD*, pages 637–646, 2004.
33. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.
34. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.
35. S. Saltenis, C. S.Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.
36. K. C. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *VLDB*, pages 16–27, 1996.
37. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.
38. Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.
39. Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, pages 334–345, 2002.
40. Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.
41. L. H. U, N. Mamoulis, and M. L. Yiu. Computation and monitoring of exclusive closest pairs. *TKDE*, 20(12):1641 –1654, dec. 2008.
42. W. M. White, C. Koch, N. G. 0003, J. Gehrke, and A. J. Demers. Database research opportunities in computer games. *SIGMOD Record*, 36(3):7–13, 2007.
43. M. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal*, 17:379–400, 2008.
44. R. Zhang, H. V. Jagadish, B. T. Dai, and K. Ramamohanarao. Optimized algorithms for predictive range and knn queries on moving objects. *Inf. Syst.*, 35(8):911–932, 2010.
45. R. Zhang, D. Lin, R. Kotagiri, and E. Bertino. Continuous intersection joins over moving objects. In *ICDE*, pages 863–872, 2008.