

Close Pair Queries in Moving Object Databases

Panfeng Zhou, Donghui Zhang, Betty Salzberg, and Gene Cooperman
Northeastern University
Boston, MA, USA
zhoupf@ccs.neu.edu,
donghui@ccs.neu.edu,
salzberg@ccs.neu.edu,
gene@ccs.neu.edu *

George Kollios
Boston University
Boston, MA, USA
gkollios@cs.bu.edu

ABSTRACT

Databases of moving objects are important for air traffic control, ground traffic, and battlefield configurations. We introduce the (historical and spatial) *range close-pair query for moving objects* as an important problem for such databases. The purpose of a range close-pair query for moving objects is to find pairs of objects that were closer than ϵ during time interval I and within spatial range R , where ϵ , I and R are user-specified parameters.

This paper solves the range close-pair query using two components: the *retrieval component* and the *close-pair identification component*. The retrieval component breaks up long trajectories into trajectory segments, which are produced in increasing time order, without the need for sorting. The retrieval component takes advantage of a new index mechanism, the Multiple TSB-tree. The segments are then pipelined to the close-pair identification component. The identification component introduces a novel spatial sweep that sweeps by time and one spatial dimension at the same time. Extensive experimental results are provided, demonstrating the advantages of the new approach when considering close pairs.

Categories and Subject Descriptors

H.2.2 [Information Systems]: DATABASE MANAGEMENT—*Physical Design, Access methods*; H.2.8 [Information Systems]: DATABASE MANAGEMENT—*Database applications, Spatial databases and GIS*

General Terms

Algorithms, Experimentation, Performance

*This work was partially supported by the NSF under Grants IIS-0073063 and ACIR-0342555

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GIS'05, November 4, 2005, Bremen, Germany.

Copyright 2005 ACM 1-59593-146-5/05/0011 ...\$5.00.

1. INTRODUCTION

Database systems that manage the location of moving objects have received considerable attention in recent years due to the proliferation of wireless and GPS-based devices and to advances in embedded systems and microelectronics. Additional application areas that will benefit from these advances include air traffic control, battlefield configurations and intelligent transportation systems. These last applications motivate the need for indexing and efficient record retrieval for close pairs.

As an example, consider the range close-pair query: “Which airplanes were closer to each other than 10 miles during the past month in Massachusetts”? This example is motivated by the *Heinrich Pyramid Principle* for ensuring air-traffic safety [19]. The Heinrich Pyramid Principle predicts that for every fatal accident, there will be three to five non-fatal accidents, ten to fifteen incidents, and hundreds of unreported occurrences. Under FAA reporting requirements, the case of two airplanes flying closer than five miles at the same altitude constitutes an incident. But the case that two planes flew 5.1 miles apart is merely an unreported occurrence. By also monitoring air logs for unreported occurrences, one can hope to further reduce both incidents and accidents.

We assume that the given database stores trajectories of moving objects, and we wish to find the objects that are close at some point in time. Two objects are considered close if their Euclidean distance is below a user-specified threshold ϵ . The user further specifies a spatial range R and a time interval I , and we focus on the parts of trajectories that were inside R during I . The problem is denoted the *range close-pair query*. Along with each identified close pair of objects, we are also interested in finding the time interval(s) when they are close.

Almost all of the existing algorithms for indexing and efficient retrieval of moving objects concentrate on simple window (range) queries and nearest neighbor queries. Such algorithms are inadequate when the concept of interest is close pairs. The key contributions of this paper are as follows:

1. We address the problem of how to perform range close-pair queries on a historical trajectory data set. The range close-pair query is important in real-life applications, but it has not been studied before. We propose a storage scheme (Multiple TSB-tree) and a retrieval method based on the MTSB-tree. The MTSB-tree can output a stream of trajectories in increasing time order

without sorting after the retrieval of the range query. We develop an algorithm which combines trajectory segments retrieved from multiple TSB-trees without duplicates in an on-line fashion.

2. We propose a plane-sweep algorithm which takes the stream of trajectory segments and computes close-pairs on-the-fly. The improvement over existing solutions is that we can sweep on both time and space in the same run. This allows us to filter out segment pairs whose lifespans intersect each other when projected into the spatial dimension, but whose spatial locations are far from each other at any given point of time.
3. We report our experimental results comparing the retrieval algorithms based on the 3D R-tree [18] and the MTSB-tree. We also compare our new sweep algorithm with the existing time-sweep algorithm.

Not only are close pairs of objects of interest in themselves, but they may be used to derive further information. Pairs of objects that co-occur in space and time can be used to find associations and correlations between objects as in [21].

In this paper we focus on polyline trajectories. To store the exact trajectory of a moving object would require storage of the location for each time instant during the moving object’s lifespan. This would produce a potentially huge volume of data. Hence, in many applications, each object trajectory is approximated as a *polyline*, a sequence of line segments with each segment connected to the next.

3D R-tree. Our proposed MTSB-tree is compared with the *3D R-tree* of Porkaew et al. [18]. The 3D R-tree is an index supporting a spatio-temporal access method, and in particular, supporting range-query capability. (However, it was not designed specifically for the close-pair queries of this paper.) The 3D R-tree quickly finds all objects that were in the query spatial region during some portion of the query time interval. The idea of the 3D R-tree is to recursively enclose object trajectories (or segments of them) into minimum bounding rectangles (MBR). When a range query is performed, if the query range does not intersect a higher-level MBR, there is no need to examine the sub-tree since the objects stored in the sub-tree are fully enclosed in the MBR.

The 3D R-tree can be adapted to answer the close-pair query. A naive approach is to first perform a range query and then perform a self-join on the range query result. This nested-loop join is expensive.

A better approach is a *plane-sweep*. Sweeping on the temporal dimension is preferable to sweeping on a spatial dimension. If two objects have trajectories whose time intervals do not intersect, they are never close pair. We can use the plane-sweep based temporal join algorithms [9] to filter out pairs of trajectories whose time intervals do not intersect.

There are two efficiency problems with this approach. First, long trajectories (or long segments) will cause the 3D R-tree to have large MBRs. This in turn results in extensive overlapping between sibling sub-trees and in poor query performance. Second, to perform a sweep on time, we need to sort the range query results.

MTSB-tree. In order to solve the two efficiency problems, we introduce the *MTSB-tree*. To address the extensive overlapping problem, we follow the practice of SETI [5]. We

partition the space into cells, and maintain a temporal access method corresponding to each cell. A trajectory is stored in all cells it intersects. The search algorithm will first find all the cells that intersect with the spatial query range. Then in each such cell, the corresponding temporal access method is utilized to locate the trajectories. Even though such an approach increases the index space, we feel that it is more appropriate for supporting the close-pair query in moving objects databases, since the query algorithm does not suffer from the extensive-overlapping problem. The challenge of this approach is to provide an algorithm that combines the trajectory segments generated from multiple cells into a stream of trajectories in increasing time order.

To avoid sorting after the retrieval of the range query, we use the Time Split B-tree (TSB-tree [15]) to index the time dimension within each cell. Because the data are stored in time order in the TSB-tree, the range query result is already ordered in time order. In [5], an R^* -tree is used to index the time dimension within each cell. The retrieval results need to be sorted to generate a stream of trajectories in increasing time order. Furthermore, during the time key split, the duplicate pointers to the same page can be created. During the interval queries, this might lead to duplicate visit to the same page via different parents. The R-tree based methods cannot solve this problem because there is overlap in the R-tree. The TSB-tree based method can solve this problem by using reference point method.

To further improve close-pair query performance, we propose a new plane-sweep algorithm that takes into consideration not only time but also space. The motivation is that there may be many trajectories whose time intervals intersect but whose spatial locations are far from each other. Comparing with the time-sweep algorithm, this new algorithm, which sweeps both on time and on space, is capable of further filtering out the comparison of such pairs.

Organization. The rest of the paper is structured as follows: Section 2 formally defines the problem we address in the paper. Section 3 outlines existing methods for processing close pair queries and indexing moving objects. Section 4 presents the details of the algorithm. Section 5 experimentally evaluates the different techniques on a simulated data set, and demonstrates the advantages of MTSB-trees for close-pair queries. Section 6 presents the conclusions.

2. PROBLEM DEFINITION

A commonly used model for moving objects is to store each trajectory as a set of line segments. Each line segment represents part of an object’s trajectory for some time period. When the object’s velocity (speed, direction) changes beyond a threshold, a new line segment is used. We focus on the *polyline trajectory dataset*. Each object o in the dataset has the following fields: (oid, p_1, p_2, \dots) . The trajectory is described by a sequence of 3-D points, where each p_i corresponds to (t, x, y) . For clarity we consider one time dimension (t) plus two spatial dimensions (x,y). The algorithms we give can be extended to more than two spatial dimensions in a straightforward manner.

DEFINITION 1. *Given a polyline trajectory dataset D , a spatial range R , a time interval I and a threshold ϵ , the **Range Close-Pair Query** finds all pairs of object IDs $(o_1.oid, o_2.oid)$ such that at some time $t \in I$, $o_1, o_2 \in D$ are both located inside R and $d(o_1, o_2) < \epsilon$. Here $d()$ is the*

2-D Euclidean distance function. Along with each close pair, the time interval(s) during which the two objects are close is also reported.

The goal of this paper is to solve the range close-pair query in moving objects databases. The first part of the task is to locate the parts of object trajectories that are inside the query spatial range R during time interval I . The second part will identify the close pairs from the query result of the first part. Furthermore, to efficiently identify close pairs in the next step, it is preferable to get the parts of trajectories in ascending order of time. More formally:

DEFINITION 2. *Given a polyline trajectory dataset D , a spatial range R , and a time interval I , the **Ordered Range Query** finds the parts of trajectories that are in R during I as a list of line segments (oid, p_1, p_2) . Here the list should be sorted in ascending order of $p_1.t$.*

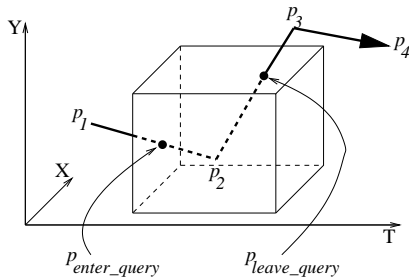


Figure 1: Illustration of the ordered range query.

Figure 1 shows a trajectory $(oid, p_1, p_2, p_3, p_4)$. The cube in the figure represents the query range, which consists of a spatial range in the X - Y plane and a time interval. The contribution of the trajectory to the ordered range query is two segments: $(oid, p_{enter_query}, p_2)$ and $(oid, p_2, p_{leave_query})$. The first one should appear earlier in the result list since the 3-D point p_{enter_query} has a smaller time than p_2 .

The MTSB-tree is proposed to process the *Ordered Range Query* and a time plane sweep algorithm is used to identify the close pairs efficiently. The MTSB-tree and the time plane sweep algorithm together are used to perform the *Range Close-Pair Query*. More details will be illustrated in the following sections.

3. RELATED WORK

Mobile data management has received increased attention in the past few years and methods to model and index moving object databases have been proposed. Database systems have been built to manage moving objects [30]. A data model for representing and querying the current and future locations of moving objects has been developed in [31]. Other proposals for modelling historical spatio-temporal databases include [10, 22]. In these proposals, the trajectory of moving objects is represented as a function of time. Also related are [29] which discusses management of moving objects in real time and [28] which discusses modelling moving objects in a road network.

Indexing moving objects for simple range and nearest neighbor queries has also received a lot of interest. In this case,

the queries are ranges in space and time. The index is used to find the objects that are contained inside the queried space during the specified time interval. The time interval may refer to the past (historical queries) or to the future (predictive queries). For the historical queries, a number of indexing methods based on traditional spatial access methods (mainly R-trees [1]) have been proposed including: 3D R-trees [18, 25], Historical R-trees [23], multi-version (or time-split) R-trees [11, 24]. These indices are used to answer spatio-temporal range or nearest neighbor queries. Indexing methods for answering other types of queries have also been proposed recently [17]. Indexing methods for the predicted location of moving objects given the current locations and their movement functions have also been proposed [14, 20].

Other topics that are related to our work are Spatial Join and Closest Pair queries in spatial databases. Efficient spatial join algorithms based on the R*-tree have been proposed in [3, 12]. An improved method based on a parallel hardware platform was introduced in [4]. Brinkhoff et al. [7] addressed data redundancy and duplicate detection in spatial join. Methods to answer closest pair queries using spatial access methods have appeared in [6, 13].

There is also some work that first divides the space into cells and then indexes each cell separately as we do. Chakka et al. [5] indexes each spatial cell by a sparse R-tree based only on the time dimension. We use the TSB-tree [15] to index each cell because the range query result is in time ascending order without sorting. In addition, we focus on the close pair query while [5] solves the time interval and time slice query.

4. FINDING CLOSE PAIRS

This section presents the solution for range close-pair queries. Section 4.1 gives an overview of our scheme. The scheme has two components: a retrieval component and a close-pair identification component. The two components are discussed in Sections 4.2 and 4.3, respectively.

4.1 Scheme Overview

Figure 2 illustrates our algorithm for finding close pairs. The trajectories are stored using some indexing structures which support range queries. At query time, when a spatial range R and time interval I are given, the retrieval component interacts with the indexing structure to locate the parts of object trajectories that intersect R during I . The located trajectory segments are pipelined in increasing time order to the identification component, which finds close pairs.

4.2 The Retrieval Component

Similarly to SETI [5], we divide the space into disjoint cells. Then for each cell we maintain a temporal index for the trajectory segments that intersect the spatial cell. SETI uses a sparse R-tree in each cell to index the time dimension and the retrieval result is not in time ascending order. The temporal index we choose for each cell is the TSB-tree [15]. The Time-Split B-tree (TSB-tree) is a temporal access method which can effectively find the data within a query time interval. The TSB-tree clusters its data in disk pages with respect to both time and key. Each page in the TSB-tree describes a rectangle in time-key space. Unlike the situation in the R-tree, the time-key spaces of distinct data pages in the TSB-tree are disjoint. The TSB-tree is chosen

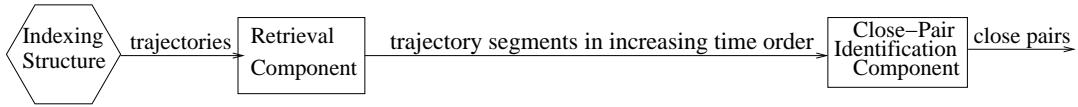


Figure 2: Overview of the scheme to find close pairs.

instead of the R^* -tree for three reasons: First, the TSB-tree is an efficient temporal access method. Second, the range query result of the TSB-tree is already sorted by time order. Third, the TSB-tree based method can avoid duplicate visits by using the reference point method [27] (more details will be illustrated later). The R-tree based methods cannot use the reference point method because there is overlap in the R-tree. Since our structure consists of a set of TSB-trees, one for each spatial cell, it is called the Multiple TSB-tree (MTSB-tree). The insertion algorithm of the MTSB-tree is similar to that of the SETI structure (i.e., if a trajectory segment covers multi-cells, it will be divided and stored in all the cells it covered). Due to space limitations, we omit the insertion algorithm of the MTSB-tree.

The intersection between a polyline trajectory and the region of a spatial cell is a possibly smaller polyline. For clarity, let's assume that each line segment is maintained as a separate record. Thus, besides the object ID, we index the time interval of each line segment. In each cell, the TSB-tree clusters the trajectories by time and objects ID.

To answer the ordered range query using the Multiple TSB-tree, we first locate the cells whose extents intersect the spatial query region. For each such cell, one or several buffers are allocated to retrieve trajectory segments from it. The buffers are used to load all the disk pages whose lifespan intersects the query time interval. The usage of the TSB-tree is twofold. First, it allows us to focus on the disk pages whose time intervals intersect the query time interval. Second, it allows us to examine such disk pages in increasing page-start-time order.

For each (part of an) object trajectory we examine in a buffer, we check every line segment of the trajectory. Even though we can scan through such line segments from one individual page in increasing order of time, it is not straightforward how to globally output the range query result in order. There are two challenges:

1. Consider disk pages A and B (either from the same cell or not). Let the start time of A be smaller than the start time of B . When we examine page A , we cannot output all line segments in it yet, because it is possible that B may contain a line segment whose start time is smaller than that of some line segment in A .
2. We need to handle: duplication from different cells and duplication in the same cell.

To address the first challenge, when we retrieve line segments from buffer pages, we do not send them to output directly. Rather, we store trajectory segments in a priority queue ordered by QET (Query Entry Time, the time that trajectory segment enters the query region). Each cell which intersects the query spatial range has one slot in the priority queue. At the beginning of the algorithm, we insert into the priority queue the trajectory segments whose QET s

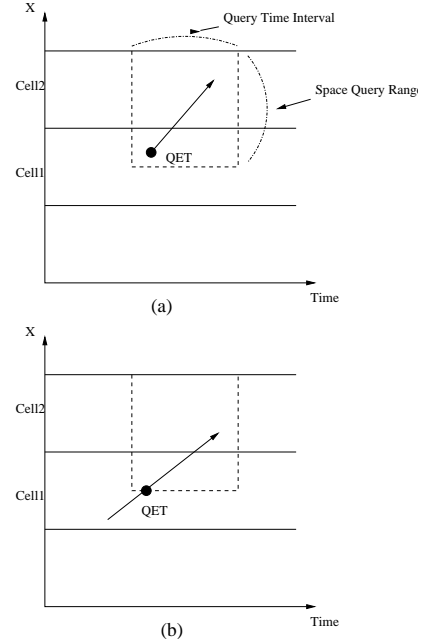


Figure 3: Duplication in different cells.

are the earliest from each cell. Then we output the first record in the priority queue to the close-pair identification component. We then replace it by the next trajectory segment from the same cell. If all trajectory segments of a page from a cell have been output to the close pair identification component, we load one more page from the same cell into the buffer until the start time of the next page is outside the query time interval. The output of the algorithm is a stream of objects in increasing order of QET .

To address the second challenge, we use a reference point approach similar to that of [27]. Duplicated loading from different cells is analyzed first. Then duplicated loading from one cell is addressed, finally, the general page loading algorithm is presented.

Since one trajectory segment can be stored in different cells, it may be loaded from different cells. This can cause a duplicate computation for the same trajectory segment. We illustrate this in Figure 3. The time-x plane is chosen to explain the different cases. The query range includes a query time interval and a query space (which is an interval in the one-dimensional x-coordinate space). The dotted line indicates the query range. The spatial part of the query range is not a simple union of cells. As we can see in Figure 3, one trajectory segment can enter multiple cells, but it is only in one cell when it enters the query region. This is the cell from which the trajectory segment is loaded into the priority queue. If a trajectory segment starts inside the

query range, then the QET is equal to the start time of the trajectory segment (see Figure 3a). On the other hand, if a trajectory segment does not start inside the query range, then the QET is equal to the intersection point of trajectory segment and query range (see Figure 3b).

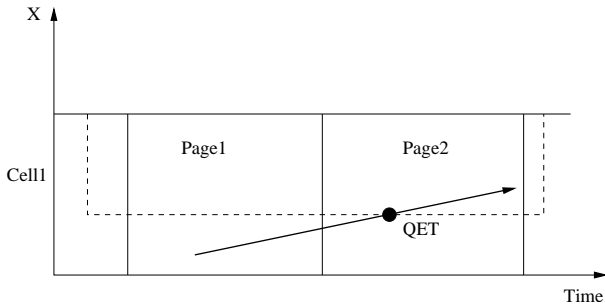


Figure 4: Duplication in the same cell.

In addition, due to the time-split and time-key split of the TSB tree, the same line segment can be stored in different pages of the same cell. This duplication is introduced (also in other temporal access methods) to achieve better clustering and thus better query performance. As an example, in Figure 4, a segment is stored in both page 1 and 2. Since the time interval of the query range (the dashed rectangle) intersects with the lifespans of both these pages, the trajectory segment will be seen twice. We know that in a TSB-tree, if multiple pages contain copies of the same trajectory segment, these pages will have non-intersecting lifespans. The QET regarding to the segment will be in one and only one of the page lifespans (in our example, the QET is in the lifespan of page 2 but not page 1). Thus we send the line segment to the priority queue only while examining that page.

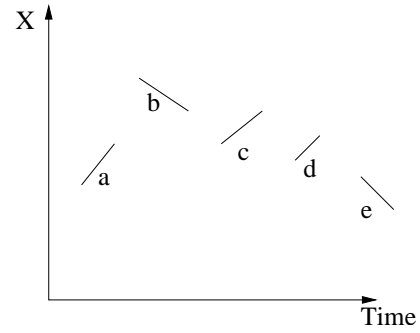
From the analysis above, the general duplicate avoidance rule which takes into consideration both the case of multiple cells and the case of multiple pages within one cell is as follows. When examining a line segment l stored in page P in cell C , we compute the QET. If l is in the cell at QET and QET is in the time interval of P , we load l from P .

4.3 The Close-Pair Identification Component

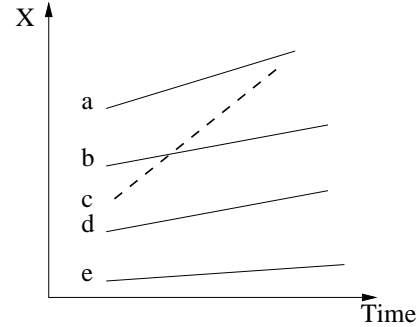
Let there be n trajectories returned by the retrieval component. A straightforward method of identifying close pairs is to compare every trajectory with every one else, resulting in an expensive $O(n^2)$ algorithm. Our close-pair identification component follows a filter-refinement process. The filter step identifies a relatively small set of candidate pairs of trajectories. Then the refinement step eliminates false positives. We focus on the filter step as this is more complex and expensive. We propose to implement the filter step by a new time-space sweep algorithm, which is better than the traditional time sweep algorithm.

The input to the close-pair identification component is a stream of line segments in increasing start time order and it comes from the retrieval component. For clarity, we first present our algorithm by assuming each line segment corresponds to a different object. We will extend the solution to the polyline case at the end of the section.

Since the retrieval component can provide the data stream in increasing time order as objects enter the query range, we



(a) Segments do not intersect in time



(b) Segments intersect in time

Figure 5: Motivation of the algorithm to sweep on both time and space.

can perform a time-sweep over the stream of objects. The idea is similar to the temporal join algorithm proposed in [9]. We keep a buffer which is initially empty. For each segment we see in the stream of input, we compare it with all segments already in the buffer, and we add the new segment into the buffer. If no segment is ever thrown out of the buffer, this algorithm becomes the naive nested-loop join where we compare every object with everything else. The improvement of the time-sweep is that whenever a new segment s is read, we can discard any in-buffer segment v whose end time is smaller than the start time of s . The reason is that all segments to be read later will have a start time larger than that of s and thus will not intersect v in time.

This time-sweep algorithm is very efficient if most of the trajectory segments do not intersect in time (Figure 5a). In the case that many trajectories intersect in time but are far from one another in some spatial dimension (Figure 5b), the time-sweep algorithm is not efficient since every segment needs to be compared with every other segment. This observation motivates our algorithm to sweep both on time and on space. The idea is described below. For the segments that intersect each other in space, if a segment b is below a , and if b and a are vertically far from each other (their distance is more than ϵ in the X dimension at all time instants), any segment below b can not be close to a . In the example of Figure 5b, there is no need to compare d or e against a . An exception is c , which swaps order with b . In such a case, b and c will intersect in the TIME- X plane. Our algorithm keeps track of such intersection events.

Our algorithm results from extending the line-intersection detection algorithms of [2, 8]. For clarity, we present the

algorithm assuming there do not exist overlapping segments (as assumed in [2]). To handle such a degenerate case, we can use the same technique as in [8].

Here are some preliminary observations.

1. **Endpoints or Intersections.** In the TIME-X plane, if two trajectory segments do not intersect, they are closest at the start time or the end time of one of the two trajectory segments. Such a trajectory segments pair is entered as a candidate pair if the closest distance at some start or end time is less than ϵ . This is illustrated in Figure 6(b). In the TIME-X plane, if two trajectory segments do intersect, then the pair is always entered as a candidate pair. This is illustrated in Figure 6(a).
2. **Relative Positions.** In the TIME-X plane, if trajectory segment L_i 's start point is below trajectory segment L_j , L_i will be always below L_j until they intersect. If they never intersect, they will always keep their relative positions. If they intersect, they will switch the relative positions after intersection. This is also illustrated in Figure 6.
3. **Triangle Inequality.** For two objects O_i and O_j , if their relative distance is always larger than the threshold ϵ in either the TIME-X plane or the TIME-Y plane, the distance in TIME-X-Y space is always larger than ϵ , and the two objects are not candidates for close pairs. This is due to the triangle inequality.

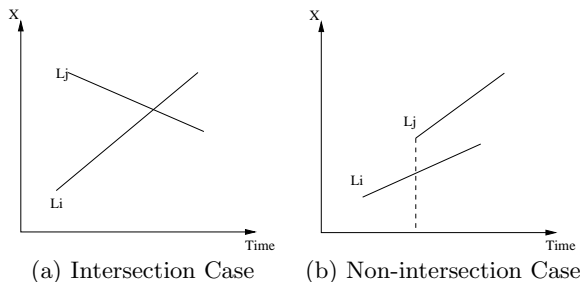


Figure 6: Relationship between two line segments in the TIME-X plane.

From the first observation, we conclude that we need only detect candidates for close pairs at intersections, end times and start times. We use an algorithm from Bentley [2] to find intersection points efficiently. We extend this algorithm to treat close pairs and to assume an incoming stream of trajectories, arriving in start time order.

There are two basic structures in the algorithm. One is the **Event List (EL)** which records all the start, end and intersection events in ascending time order. The other one is the **Sweep Line (SL)** which will record line segments' relative positions in the x dimension at each event time of EL.

From the second of the preliminary observations, we know that if we insert the objects' starting relative positions in SL in ascending order and switch their positions in SL after an intersection, and sweep on time, we always get the correct relative positions of objects in SL. The description of the algorithm is now given, followed by an example.

There are three kinds of events to process:

1. **Start event.** Let O be the object whose start event is processed. Insert the end time for O in EL. Insert O in SL. Calculate the distance to the other objects in SL in the x -dimension at the time of the start event. Detect all of O 's neighbors whose x -coordinate in SL are within ϵ . Then, find the current immediate neighbors of O in SL. Using the function of time for the x -coordinate of the trajectories, determine if O intersects with its immediate above or below neighbor in SL at any future time. If so, add that intersection time to EL.
2. **End event.** Let O be the object whose end event is processed. Detect any objects in SL which currently have their x -coordinate within ϵ of the x -coordinate of O . In addition, if O 's immediate above and below neighbors in SL intersect with each other in the future, we will add this intersection event to EL. Finally, delete O from SL.
3. **Intersection event.** Let P and Q be two objects that intersect. Switch their positions in SL. Suppose P is above Q before their intersection. If P 's former immediate above neighbor intersects with Q or if Q 's former immediate below neighbor intersects with P , add these intersection events to EL.

At the end of any of these events, any candidates for close pairs (those whose x -coordinates were within ϵ , or those which intersected in the TIME-X plane) are examined using a Euclidean distance calculation. Also the events are removed from EL.

Now we show how to schedule these events. Unlike Bentley [2], we assume an incoming stream from an external file. Thus at the beginning of the algorithm, not all start events are in EL.

If EL is empty, we will read in all the objects whose start time is the next time instant (using the MTSB-tree), and process start events for each of the new elements of SL. Then we must decide what event to process next.

Each time we make the decision of what event to process next, we compare the time of the next event in EL with the start time of the next trajectory from the MTSB-tree. We take the earliest of these two events.

If events from different trajectories have the same times, we process first the start events, then the intersection events, and last the end events. Thus we will be able to detect close pairs whose lifetimes only intersect when the end time of one is the start time of another.

We illustrate the algorithm using the examples in Figure 7. At the beginning, both EL and SL are empty. Then we read A from TSB file and initiate a start event for A . This places the end time 5 in EL. The object A is placed in the list SL. Since there is now only one object in SL, no future intersections are placed in EL and no candidates for close pairs are detected.

Now we compare the event times in EL (i.e. 5) with the next start time to come from the TSB-tree which is (2). We thus process the start event for B , which is earlier than the end event for A in EL.

Now we have in EL the times 5 (end time for A) and 8 (end time for B) and we have A below B in SL. No intersection

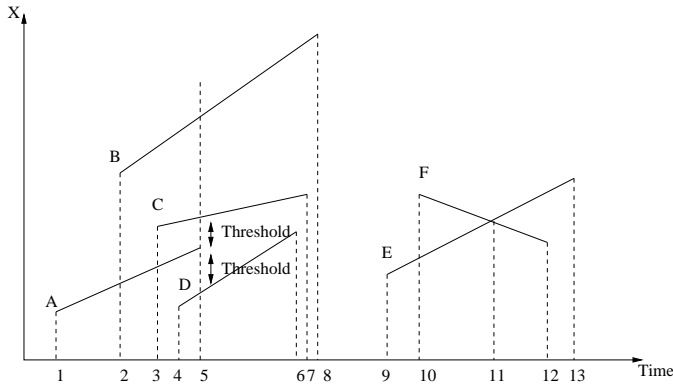


Figure 7: Plane sweep on time and x .

event has been detected because A and B do not intersect in their lifetimes.

The next comparison between EL and the incoming new start times compels us to process a start event for C , inserting the end time 7 in EL and placing C between A and B in SL.

We next process the start event for D placing its end time 6 in EL. Now the ordered list in SL is $[D, A, C, B]$.

At event 5, we find that C and D have x -coordinates within ϵ of A 's x -coordinate. Thus (A, C) and (A, D) are our first candidates for close pairs. At this point A is removed from SL.

After we processed end events 6, 7 and 8, EL and SL become empty again. We will read E from the TSB-tree, insert event 13 into EL, and add E to SL.

When we compare the next start time from the TSB-tree (10) with the next event time in EL, we see we must next process a start event for F . This time an intersection event is detected and we add the event with time 11 to EL. At event 11, we find the close pair candidate (E, F) . Later, events 12 and 13 are processed, leaving EL and SL empty again.

We next analyze the running time. The size of EL is $(n + k)$ where n is the number of trajectories currently alive and k is the number of intersections of those trajectories. Normally, in our variation of the algorithm, there will be n end events in EL, one for each live trajectory, and up to k intersections. (Some of the intersections for live trajectories may not yet be detected.)

All the insert and remove operations for SL can be done in $O(\log n)$ operations if we implement SL as a binary search tree. Similarly, the insert and delete operations for events in EL have complexity $O(\log(n + k))$. Then the total running time of this algorithm is $O((n + k) \log(n + k))$.

We have also explored the plane sweep algorithms that use more spatial dimensions (e.g., X and Y dimensions). Using more spatial dimensions can avoid more unnecessary comparisons between trajectory segments. But the cost of maintaining more spatial dimensions in the plane sweep algorithm exceeds the benefit of comparing less trajectory segments pairs. In later sections, we will focus on the TIME- X sweep algorithm (i.e., plane sweep on time dimension and one spatial dimension).

The above discussion assumed that each line segment corresponds to a different object. For polyline trajectories, mul-

iple line segments may belong to the same object. Our algorithm needs only some minor extensions to deal with this case. Since each segment in our case has an object ID, our algorithm is extended so that we do not compare pairs of segments with the same object ID.

Another difference is that when each object corresponds to a single line segment, there is only one time interval during which two objects may be close; but with polyline trajectories, the times when two trajectories are close may not be connected time intervals. Two trajectories may diverge and then approach one another later. In the latter case, our algorithm will report the pair of objects multiple times, each time with a different interval they are close.

5. EXPERIMENTAL RESULTS

In this section, we present a preliminary evaluation of the methods proposed in Section 4. Section 5.1 discusses the experimental set-up. In section 5.2 we give a comparison between the plane sweep on time and the plane sweep on time and space. In Section 5.3, we compare the 3D R-tree approach against the Multi-TSB tree for retrieving trajectory segments from external memory.

Our scheme has two components: a retrieval component and a close-pair identification component. The choice of which method to use in which component is independent. For example, both retrieval methods can use either the *Time Sweep* or the *TimeX Sweep* algorithm as the close-pair identification component. So we have measured the performance of each component separately. When measuring the close-pair identification component, we used elapsed time. When measuring the retrieval component, we use the usual metric for access methods: the number of pages accessed. Due to the space limitation, the performance of the insertion algorithm and the tree structure are not included.

5.1 Experimental Setup

All experiments were run on an Intel Pentium IV 2.66GHz CPU machine with 1GB main memory. To implement our algorithms we used the XXL(eXtensible and fleXible Library) [26], which is a Java-based library containing a rich infrastructure for implementing advanced query processing functionality. The spatial area is 10000 x 10000 miles and it is partitioned into uniform rectangular cells. In all the experiments, the disk page size is 16KB (following the suggestions of Lomet [16] for the B-tree). Values in all the dimensions, including the time dimension, are represented by 8-byte doubles. Each entry in the leaf node of the 3D R-tree (The R*-tree is used in the experiments) occupies 68 bytes, which includes a 4D rectangle (i.e., 3D for spatial dimensions and 1D for time dimension) and an object ID (4 bytes). A leaf node contains between 120 and 240 entries.

We have tried to obtain real air traffic data sets. But it is extremely hard to obtain those data sets after *September 11th*. We generated a synthetic data set (i.e., *Air Traffic Control(ATC) data set*) to the best of our knowledge from talking with air traffic controllers. First, random points (CITIES) are generated. In the real world, a flight between two cities will not be a straight line. The flight trajectory will be a zig-zag polyline from source city to destination city. In order to simulate this, for each trajectory, we first randomly select the source and destination points from CITIES, and then we generate a set of middle points between source and destination cities. The middle points represent small

changes in the direction and/or speed of objects. Thus, the trajectory is a polyline with line segments which link the source to the destination. Each trajectory is assigned an initial altitude. The altitude will not change until the trajectory reaches the next middle point or the destination point. The trajectory will be assigned a new altitude when it reaches a new middle point. This is a reasonable assumption because airplanes try to keep their altitudes for the sake of fuel saving and passenger comfortableness.

The ATC data set is denoted as $ATC(n_t, n_l)$ where n_t is the number of trajectories and n_l is the number of the line segments in each trajectory. In the following experiments, unless mentioned explicitly, $ATC(1000, 200)$ which includes 200,000 trajectory segments is used and the default threshold for closeness is two miles.

5.2 Time Sweep vs Time Space Sweep

In this section, we compare the plane sweep algorithm on time and the plane sweep algorithm on both time and one of the spatial dimensions (X-axis). The plane sweep algorithm on both time and x-coordinate is denoted as the *TimeX Sweep* algorithm while the plane sweep algorithm on time is denoted as the *Time Sweep* algorithm. In order to better illustrate the trends caused by different factors, the *Ratio*, which is defined as the *Elapsed Time (CPU Time)* of the *Time Sweep Algorithm* divided by *Elapsed Time of the TimeX Sweep Algorithm*, is chosen as the metric for the two algorithms.

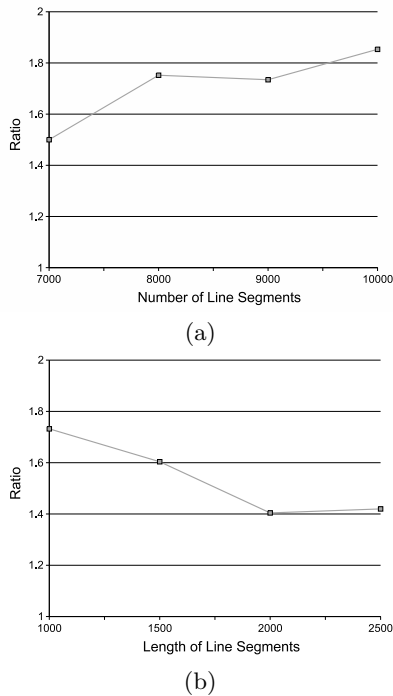


Figure 8: Varying the number and length of the trajectory segments.

In the first experiment, we examine how the number of trajectory segments affects finding close pairs. In this experiment, the length of the line segments is 1000 miles and the threshold for close pairs is 2 miles. The result is displayed in Figure 8(a). As the number of trajectory segments increases,

the space becomes more crowded. The *TimeX Sweep* is more efficient since it compares fewer pairs of line segments.

In the second experiment, we examine how the length of the trajectory segments affects close pair processing. We keep the number of trajectory segments constant (e.g., 10000), while we vary the length of the segments. The result is shown in Figure 8(b). The experiments with other numbers of trajectory segments have similar results. As the length of trajectory segments increases, the processing time for the *Time Sweep* algorithm will not change too much since the total number of line segments remains the same. But as the length of trajectory segments increases, there are more close pair candidates in the space and the space filter is less effective, so the *TimeX Sweep* algorithm becomes less efficient. (However, it is still more efficient than the *Time Sweep* algorithm.)

The experimental results in Figure 8 verify our theoretical analysis result at the end of section 4.3. If the average number of trajectory segments, whose life spans intersect with any given trajectory segment's life span, is large, then the *TimeX Sweep* algorithm will outperform the *Time Sweep* algorithm because the *TimeX Sweep* algorithm can avoid unnecessary closeness detection computations. But we also observe that if the average number of trajectory segments whose life spans intersect with any given trajectory segment's life span is small, then the *Time Sweep* algorithm will outperform the *TimeX Sweep* algorithm because of its simplicity.

5.3 3D R-tree vs MTSB-tree

In this section, the 3D R-tree and MTSB-tree are compared in different experimental settings. For each experiment, we run 100 random queries (each of these queries starts with a cold buffer pool) and calculate the average number of page accesses per query. For the first two sets of experiments in this section, a random query covers one percent of the total space area (10% in each dimension) and thirty percent of the total time interval. The last set of experiments we report varies the query range. The number of pages accessed is used as the metric for the two access methods.

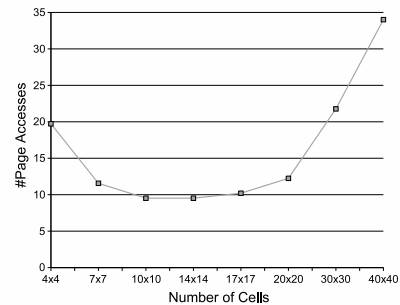


Figure 9: Varying the number of cells in the MTSB-tree.

In the first experiment, we study how the number of spatial partitions can affect the search performance. Figure 9 shows the results for the $ATC(1000, 200)$ data set, which includes 200,000 trajectory segments. If the whole area is divided into a small number of cells, each cell will cover

a large portion of the whole area. As a result, some trajectories which are not in the query range are also included, which decreases the performance of the spatial filtering. But when the whole area is divided into a large number of cells, each trajectory will be stored in many different cells. This will decrease the query performance since the trajectory in the query range will have to be examined and filtered from the loading process many times. Between 10×10 and 17×17 divisions, the performance change is gradual and is optimal. We observed similar behavior (i.e., page accesses will first decrease, then increase as the number of cells increases) for other queries and other data sets. Different data sets achieve their optimal performance for different numbers of cells. We keep the number of cells constant at $100(10 \times 10)$ in the following experiments. The results are in line with those reported in [5].

Next, we vary some of the properties of the data set itself and we measure the impact on the performance of the proposed algorithms. First we investigate how the number of line segments in each trajectory affects the query performance, while the number of trajectories is kept constant. More segments take up more bytes which will lead to more data pages. So the number of pages accessed for both methods increases. By observing Figure 10(a), the MTSB-tree clearly outperforms the 3D R-tree. Then we keep the number of line segments in each trajectory constant, but change the number of trajectories. This also results in more line segments in the same fixed area. The result is shown in Figure 10(b). The MTSB-tree outperforms the 3D R-tree in this case also.

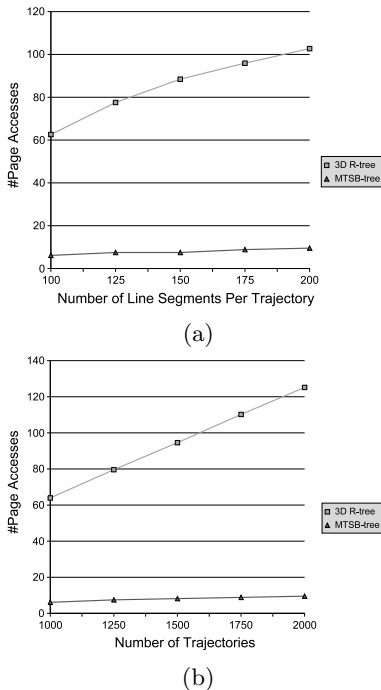


Figure 10: Varying the number of line segments per trajectory and the number of trajectories.

In the last set of experiments, we check how the properties of the query can affect the query performance. In the first experiment, we vary the length of the query time interval.

In this experiment, the MTSB-tree is much better than 3D R-tree since it separates the temporal and spatial indexing and it only needs to access the space that intersects with the query spatial range. However, as the time interval gets longer, the duplication in the MTSB-tree causes some degradation in performance. The result is shown in Figure 11(a).

Next, the size of spatial query range is varied. Since the MTSB-tree stores some trajectory segments in several different cells, its performance will decrease as the query spatial range increases. The result is displayed in Figure 11(b).

From the experiments above, we can conclude that as long as the spatial range of the query is not too large (which is practical, since we seldom need to query the whole space), the MTSB-tree is more efficient than the 3D R-tree for range queries on trajectory data.

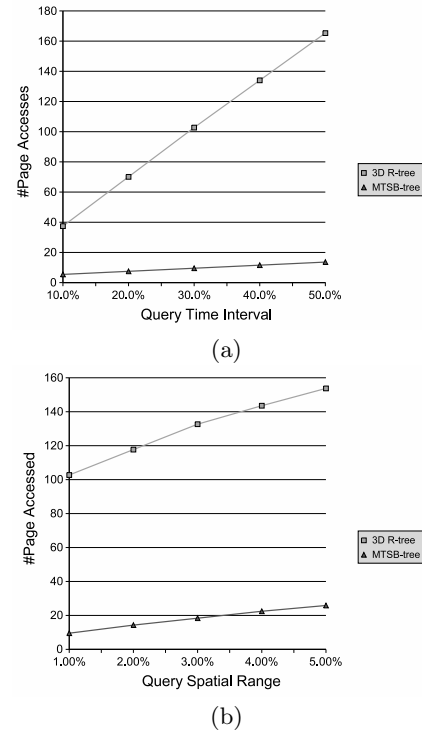


Figure 11: Varying the query time interval and query spatial range.

6. CONCLUSIONS

In this paper, we addressed the problem of close pair queries in a moving object database. We proposed solutions to efficiently retrieve segments of trajectories that are in a given spatial range during a given time interval. We proposed a storage scheme (MTSB-tree) and the corresponding query algorithm which avoids the sorting phase after retrieval. Methods to avoid duplicates are given. The proposed algorithms can return a stream of trajectory segments that are of interest in increasing time order. We then proposed an algorithm which takes this stream of trajectory segments as input and identifies close pairs of objects. This algorithm is a variation of plane-sweep. The improvement is that we can sweep on both the time dimension and a spatial dimension. This can be a big saving if the lifespans of

many objects intersect each other, but they are spatially far from each other. Finally, we experimentally compared the MTSB-tree approach and the 3D R-tree approach for retrieval, and we compared the new sweeping algorithm with the time-sweep algorithm. Experimental results show that our schemes outperform existing solutions.

7. REFERENCES

- [1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.
- [2] J. L. Bentley and T. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.
- [3] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 237–246, 1993.
- [4] T. Brinkhoff, H. Kriegel, and B. Seeger. Parallel Processing of Spatial Joins Using R-trees. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 258–265, 1996.
- [5] V. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Conference on Innovative Data Systems Research*, 2003.
- [6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 189–200, 2000.
- [7] J. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 535–546, 2000.
- [8] W. Freiseisen and P. Pau. A Generic Plane-Sweep for Intersecting Line Segments. In *RISC-Technical Report no 98-18*, 1998.
- [9] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 336–344, 1991.
- [10] R. Güting, M. Bohlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000.
- [11] M. Hadjieleftheriou, G. Kollios, V. Tsotras, and D. Gunopoulos. Efficient Indexing of Spatiotemporal Objects. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 251–268, 2002.
- [12] G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 237–248, 1998.
- [13] D. Zhang, J. Shan, and B. Salzberg. On Spatial-Range Closest-Pair Query. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 252–269, 2003.
- [14] G. Kollios, D. Gunopoulos, and V. J. Tsotras. On Indexing Mobile Objects. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 261–272, 1999.
- [15] D. Lomet and B. Salzberg. The Performance of A Multiversion Access Method. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 353–363, 1990.
- [16] David B. Lomet. B-tree page size when caching is considered. *SIGMOD Record*, 27(3):28–32, 1998.
- [17] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 395–406, 2000.
- [18] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 59–78, 2001.
- [19] FAA Report. Minutes of the Research, Engineering and Development Advisory Committee.
- [20] S. Saltenis, C. Jensen, S. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 331–342, 2000.
- [21] S. Shekhar and Y. Huang. Discovering Spatial Co-location Patterns: A Summary of Results. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 236–256, 2001.
- [22] J. Su, H. Xu, and O. Ibarra. Moving Objects: Logical Relationships and Queries. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 3–19, 2001.
- [23] Y. Tao and D. Papadias. Efficient Historical R-trees. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 223–237, 2001.
- [24] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-temporal Access Method for Timestamp and Interval Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 431–440, 2001.
- [25] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal Indexing for Large Multimedia Applications. In *Proceedings of International Conference on Multimedia Computing and Systems (ICMCS)*, pages 441–448, 1996.
- [26] J. van den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL- A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 39–48, 2001.
- [27] J. van den Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 168–179, 1996.
- [28] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 20–35, 2001.
- [29] O. Wolfson, L. Jiang, A. P. Sistla, S. Chamberlain, N. Rishé, and M. Deng. Databases for Tracking Mobile Units in Real Time. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 169–186, 1999.
- [30] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases for Moving Objects tracking. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 547–549, 1999.
- [31] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 111–122, 1998.