Maintenance of *K*-nn and Spatial Join Queries on Continuously Moving Points

GLENN S. IWERKS
The University of Maryland at College Park/The MITRE Corporation
HANAN SAMET
The University of Maryland at College Park
and
KENNETH P. SMITH
The MITRE Corporation

Cars, aircraft, mobile cell phones, ships, tanks, and mobile robots all have the common property that they are moving objects. A kinematic representation can be used to describe the location of these objects as a function of time. For example, a moving point can be represented by the function $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ is its velocity vector. Instead of storing the location of the object at a given time in a database, the coefficients of the function are stored. When an object's behavior changes enough so that the function describing its location is no longer accurate, the function coefficients for the object are updated. Because the location of each object is represented as a function of time, spatial query results can change even when no transactions update the database. We present efficient algorithms to maintain k-nearest neighbor, and spatial join queries in this domain as time advances and updates occur. We assume no previous knowledge of what the updates will be before they occur. We experimentally compare these new algorithms with more straight forward adaptations of previous work to support updates. Experiments are conducted using synthetic uniformly distributed data, and real aircraft flight data. The primary metric of comparison is the number of I/O disk accesses needed to maintain the query results and the supporting data structures.

Categories and Subject Descriptors: E.2 [**Data Storage Representations**]: *Object representation*; H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.3.3 [**Information Storage and**

The work of K. P. Smith and G. S. Iwerks was supported in part by a grant from the Human Brain Project (R01-MH64417), which is funded by the National Institute for Mental Health, and by funds from the MITRE Technology Program (project 07MSR204).
The work of H. Samet was supported in part by the National Science Foundation under grants IIS-00-86162, EIA-00-91474, and CCF-0515241, and by Microsoft Research.
Authors' addresses: G. S. Iwerks and K. P. Smith, The MITRE Corporation, 7515 Colshire Drive, McLean, VA 22102; email: {iwerks;kps}@mitre.org; H. Samet, Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland at College Park, College Park, MD; email: hjs@umiacs.umd.edu.

Retrieval]: Information Search and Retrieval—*Selection process*; J.7 [**Computers in Other Systems**]: *Command and control*

General Terms: Algorithms, Design, Experimentation, Measurement, Performance, Theory

Additional Key Words and Phrases: Moving object databases, temporal databases, materialized view maintenance, k-nearest neighbor, spatial join, continuously moving objects

1. INTRODUCTION

Consider the following queries. For a cell phone, keep track of the nearest cell tower. For a suspect getaway car, keep track of the nearest police cruiser. For a robot explorer, keep track of the nearest maintenance robot. For a ship, keep track of the nearest sonar tracking station. For each airplane, keep track of every other airplane that is too close for safety. For each tank, keep track of each target that is within firing range. For each robot explorer in a swarm of robots, keep track of all neighboring robots that are within radio range. For each unmanned air vehicle, keep track of every observation target within 5 miles.

Cars, aircraft, mobile cell phones, ships, tanks, and mobile robots all have the common property that they are moving objects. They can all be modeled conceptually as values that change as a function of time. In particular, this article addresses geo-located objects whose position is represented as a function of time $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ its velocity vector. These function coefficients $\overrightarrow{x_0}$, t_0, $\overrightarrow{v}$, are stored in a database. When the error, between the object's actual location and the function in the database describing its location, exceeds a predefined threshold, the database is updated (i.e., the object's course and speed are updated). Updates consist of a set of insertions and deletions to the base relations. These are applied incrementally to a query result in a manner similar to incremental materialized view maintenance [Gupta et al. 1993].

The motion of objects can change the query result independently of updates. An event is said to occur when the motions of a set of objects cause the query result to become incorrect. An event is characterized by a set of objects and the time at which the motion of those objects causes the query result to become incorrect. One or more events can be precomputed and then processed in temporal order to maintain the query result. When an event occurs, the information about the objects involved in the event is used to correct the query result. It is important to note that an update is not an event as it is defined here, but both events and updates can change the query result.

One may observe that the first four of our motivational examples belong to the category of nearest neighbor queries, while the other four are to spatial join queries, and all of them need their answers maintained over time. This is precisely the goal of this paper, to address the efficient maintenance of spatial join and k-nearest neighbor queries on moving points.

In maintaining queries, we assume no advance knowledge of what changes may occur before they occur, as is the case in a real time system. Additionally, we do not keep track of any past information about a point's motion beyond the last update.

Previous work on spatial k-nearest neighbor queries on static data includes Arya et al. [1998], Hjaltason and Samet [1999], and Roussopoulos et al. [1995], and for spatial join on static data includes Arge et al. [2000], Brinkhoff et al. [1993], Hjaltason and Samet [1998], Lo and Ravishankar [1996], and Patel and DeWitt [1996]. Objects represented as a function of time have been studied in other domains such as simulation [Fujimoto 1990; SCIS 1996], and computational geometry [Basch et al. 1997]. For moving object databases, past research includes indexing methods [Agarwal et al. 2000; Agarwal and Procopiuc 2002; Jensen and Saltenis 2002; Mokbel et al. 2003; Pfoser 2002; Saltenis et al. 2000; Tao et al. 2003a; Tayeb et al. 1998], ad-hoc queries [Saltenis et al. 2000; Sistla et al. 1997], and continuous queries such as continuous window [Saltenis et al. 2000; Sistla et al. 1997], within [Iwerks et al. 2003; Tao and Papadias 2003], nearest neighbor and k-nearest neighbor (k-nn) [Benetis et al. 2002; Iwerks et al. 2003; Mokhtar et al. 2002; Raptopoulou et al. 2003; Song and Roussopoulos 2001; Tao and Papadias 2003], spatial join [Tao and Papadias 2003], spatial semijoin [Iwerks et al. 2004], selectivity estimation [Tao et al. 2003b], and uncertainty [Pfoser and Jensen 1999; Wolfson et al. 1998]. Temporal databases [Özsoyoğlu and Snodgrass 1995] are a related topic to moving object databases. See Güting and Schneider [2005] for a detailed survey of moving object databases.

Although there is a large body of works that have addressed problems similar to the ones that we tackle in this paper (see Section 5), the distinct characteristic of our work is that we assume settings in which the information about the moving objects is represented function of time. This representation and, in particular, the nature of the application domains and the impact that they have on the updates of the moving objects, pose unique challenges for the efficient maintenance of the correctness of query results. To the best of our knowledge, none of the existing approaches would be appropriate for the domain in which we conduct our experiments, that is, continuously moving aircraft data.

Our main contributions in this article are query maintenance algorithms for k-nn and spatial join queries on continuously moving points that support updates to the base relations. Although the examples given in this article are 1-dimensional (1D) and 2-dimensional (2D) and show static query points, the techniques and algorithms are general and applicable to higher dimensions and moving query objects.

The rest of this article is organized as follows: Section 2 contains definitions and background on event-driven query processing. Section 3 presents event-driven query maintenance algorithms for k-nearest neighbor and spatial join queries. Section 4 presents experimental results. Section 5 describes some previous related work in this area. Conclusions and directions for future work are discussed in Section 6. The Appendix gives more detailed pseudocode for the algorithms presented in this article.

2. EVENT DRIVEN QUERY PROCESSING FOR MOVING POINTS

2.1 Data Types

A common representation for moving points are sampled locations (sometimes known as discretely moving points) [Nascimento et al. 1999]. For example, the motion of an aircraft can be represented by sampling its location using radar every 6 seconds and updating its position in the database each time the location of the aircraft changes. The problem with this representation is that the costs of updating the location of every aircraft in flight in a database every 6 seconds, and maintaining queries between updates, are prohibitive.

Kinematic points are an alternative to the sampled location representation described in the previous paragraph. The location of a *kinematic* [Nakamura and Yamane 2000][1] point is modeled as a function of time. In particular, the motion of a linear kinematic point is represented by the linear function $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ its velocity vector. The coefficients of this function are stored in the database for each point. When the actual location of the point diverges from the function by some error threshold, the database is updated. For example, if the function indicating the position of a particular aircraft shows it to be more than 2 nautical miles from the position detected by radar, then the database is updated.

2.2 Events

Events are used to maintain query results on kinematic points as time advances. Events are processed to keep the query result consistent as the points move. To support the maintenance of the results of these queries, we distinguish between two basic types of events: *within events* (w-event), and *order change events* (oc-events).

The first basic event type, the *within event* (w-event), occurs when a point moves to a specified distance (say distance d) from a query point. If a point is moving closer to the query point, then the w-event is called an *enter* event. If a point is moving farther away from the query point, then the w-event is called an *exit* event. For example, imagine a circle of radius d centered on a query point in a 2D space. An enter event occurs when a point moves from the outside of the circle to the inside, and the reverse is an exit event. For a moving point, the time of a within event is based on solving the Euclidean distance equation $|p(time), q(time)| = d$ for $time$, where p and q are two moving points. This results in a closed form quadratic equation. See Raptopoulou et al. [2003] and Tao and Papadias [2003] for more details on the computation of events between pairs of moving points.

The other basic type of event is the *order change event* (oc-event). The oc-event occurs when two points change order with respect to their distance from a query point. For query point q, and two other points p_1 and p_2, the time of their oc-events is based on solving the equation $|p_1(time), q(time)| = |p_2(time), q(time)|$

[1]Kinematics is defined as the branch of mechanics that studies the motion of a body or a system of bodies without giving any consideration to its mass or the forces acting on it.

for *time* (see Raptopoulou et al. [2003] and Tao and Papadias [2003] for details). A special case of an oc-event is a *nearest neighbor event* (nn-event). Given a query point and its current kth neighbor, the next nn-event is the soonest oc-event to occur in the future out of all possible future oc-events between the query point, the current kth neighbor, and any other point in the data set. For example, suppose that q is a query point, p_k is its current kth neighbor, and S is a set of moving points $S = \{s_1 \cdots s_n\}$. For each point $s_i \in S$, if s_i is closer to q than p_k, then the next oc-event e_i of point s_i occurs the next time when s_i moves to become farther from q than p_k. On the other hand, if s_i is farther from q than p_k, then the next oc-event e_i of point s_i occurs the next time when s_i moves to become closer to q than p_k. The next nn-event for q, p_k, and S is the soonest oc-event e_i of all future oc-events $\{e_1 \cdots e_n\}$. The time of the next nn-event is the next time in the future when the kth neighbor of q will change.

2.3 Notation

A particular instance of a kinematic point is denoted as $\mathsf{pt}(x_0, v, t_0)$, where x_0 is the start location, t_0 is the start time, and v is the velocity vector. We assume an object-relational database environment in which a kinematic point data type is an attribute in a relation r, referred to as a *moving point*, or simply a *point*. The object attribute for the kinematic moving point object stores the last known position (x_0 at time t_0) and velocity (v) of a point. When an update occurs, the old x_0, v and t_0 are discarded, and replaced by new values. No previous information about a point's movement prior to the last update is saved, just as no future information (i.e., future updates) is known or stored in the relation. For simplicity, and without loss of generality, we consider relations having one moving point attribute. The instance of a moving point attribute for some tuple $\tau \in r$ is denoted $\mathsf{P}(\tau)$. Each instance of a moving point attribute value has its own unique identifier. This allows us to index a relation on the point's id and retrieve the tuple to which the instance belongs. To indicate the tuple containing some point instance p we write $\mathsf{Tuple}(p)$. The Euclidean distance between point instances p and q at time t is $\|p, q, t\| = |p(t), q(t)| = \sqrt{(q(t) - p(t))^2}$.

A w-event instance is denoted as $\mathsf{w}(p, t)$ where p is the moving point, and t is the time of the event. It is important to remember that the query point and distance are part of the query, and not explicitly represented in the w-event notation.

An oc-event instance is denoted as $\mathsf{oc}(p, t)$ where p is the point involved, and t is the time. This notation is only meaningful in the context of a k-nn query where the current state of the query is known at time t. For example, consider three points **q**, **a** and **b** where **q** is the query point for a 1-nn query. Now suppose point **a** is the nearest neighbor to **q** and that point **b** is farther from **q** than **a** just before time t. Also suppose that point **b** becomes the new nearest neighbor after time t. Points **a** and **b** are equidistant from **q** at time t. This nn-event is an order change event denoted as $\mathsf{oc}(\mathbf{b}, t)$. There is no need to explicitly represent query point **q** or the nearest neighbor **a** in the oc-event notation because they are understood in the context of the query.

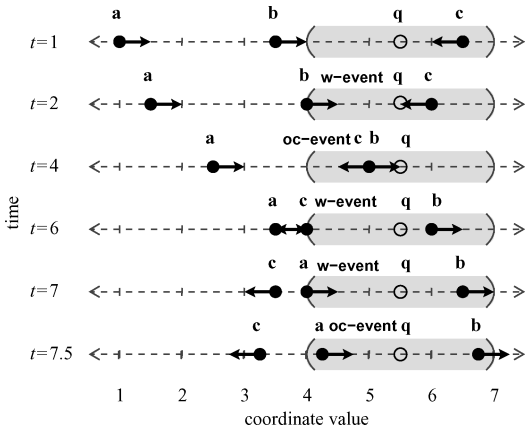


Fig. 1. Example snapshots of 1D moving point attributes and events for time interval $1 \leq t \leq 7.5$. Arrow lengths indicate the distances traveled in one time unit.

For a given event e (either a w-event or an oc-event), $\mathsf{P}(e)$ denotes the moving point explicitly represented in that event (e.g., $\mathsf{P}(\mathsf{w}(p,t)) = p$). The time of an event is denoted as $\mathsf{Time}(e)$ (e.g., $\mathsf{Time}(\mathsf{w}(p,t)) = t$). For a null event (denoted $e = \emptyset$), $\mathsf{Time}(\emptyset) = \infty$.

2.4 Event Example

For illustration, we present an example from Iwerks et al. [2003]. Figure 1 shows snapshots of a 1-dimensional data set $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ of moving points, and a query point $\mathbf{q}$ at different instances of time where $\mathbf{a} = \mathsf{pt}(1, 0.5, 1)$, $\mathbf{b} = \mathsf{pt}(3.5, 0.5, 1)$, and $\mathbf{c} = \mathsf{pt}(6.5, -0.5, 1)$. The query point is $\mathbf{q} = \mathsf{pt}(5.5, 0, 1)$, and the query distance is $d = 1.5$. The shaded area around $\mathbf{q}$ indicates the region along the line that is within distance $d = 1.5$ of $\mathbf{q}$. A w-event, $\mathsf{w}(\mathbf{b}, 2)$, takes place at time $t = 2$ when point $\mathbf{b}$ comes within distance $d = 1.5$ of query point $\mathbf{q}$. An oc-event $\mathsf{oc}(\mathbf{b}, 4)$ is shown at time $t = 4$ when point $\mathbf{b}$ is moving closer to $\mathbf{q}$ than point $\mathbf{c}$. At time $t = 4$, point $\mathbf{c}$ is the nearest neighbor prior to the event.

2.5 Event Driven Query Processing Without Updates

In this section we present some simple algorithms to maintain the correctness of the queries when there are no updates to the database over the duration of the query. This section serves as a tutorial to give the reader a better sense of how events are used in query processing, and of the properties of the different types of events. Section 3 describes more sophisticated algorithms for query maintenance that support updates.

Event-driven query processing is used to maintain queries on kinematic data types. This is similar to event-driven simulation [Fujimoto 1990], but instead of maintaining a simulation state, events are used to maintain query results as time advances. Events are processed in turn to keep query results consistent as points move.

```
    procedure Simple_Within(r, q, d, t)
1.    Q ← ∅, W ← ∅
2.    for each tuple τ ∈ r do
3.      if ||P(τ), q, t|| < d then W ← W ∪ {τ}
4.      e ←next_w_event(P(τ), q, d, t)
5.      if Time(e) < ∞ then Q ← Q ∪ {e}
6.    end for-each
7.    while Q ≠ ∅ do
8.      e ←Dequeue(Q),   t ←Time(e)
9.      if e is an enter event then
10.       W ← W ∪ {Tuple(P(e))}
11.       e ←next_w_event(P(e), q, d, t)
12.       if Time(e) < ∞ then Q ← Q ∪ {e}
13.     else W ← W \ {Tuple(P(e))}
14.   end while
```

Fig. 2. Simple_Within().

Event-driven within query processing is performed by examining all within events in temporal order while updating the result appropriately.[2] Figure 2 gives a simple event-driven algorithm Simple_Within() for maintaining a within query. For example, consider the 1D scenario in Figure 1, and a query to find all points within distance $d = 1.5$ of q. Initially, relation r containing tuples with moving point attributes $\{a, b, c\}$[3] is scanned (lines 2–6) in order to find the initial result at time $t = 1$, $W = \{c\}$, and the next w-event for each point. The w-events are inserted into priority queue Q, so that $Q = \{\mathsf{w}(b, 2), \mathsf{w}(c, 6), \mathsf{w}(a, 7)\}$. Function Dequeue($Q$) removes the next event from Q and returns it. Function next_w_event(p, q, d, t) returns the next event after time t when point p will be at distance d from q, or it returns a null event with time stamp ∞ if no such event exists. This is a simple computation based on solving the equation $|p(time), q(time)| = d$ for *time*. Since the positional components of a moving point are expressed as linear functions of time, for dimensionality greater than 1, this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. Events are processed one-by-one (Figure 2, lines 7–14). Lines 9–12 process enter events by adding the incoming point to the within result and computing the next exit event. Line 13 processes exit events by removing the point from the within result. No new events are generated by exit events. Figure 3 shows a trace of the event processing portion of the algorithm (lines 7–14) up to time $t = 7$ for the 1D example in Figure 1.

An event-driven k-nn query processing algorithm finds the soonest oc-event to occur in the future out of all possible oc-events. This is called the *nearest neighbor event* (nn-event) because it will cause the k-nn query result to change. A nn-event is an oc-event, but not every oc-event is a nn-event. Figure 4 outlines a simple event-driven algorithm to maintain a simple nearest neighbor query.

The algorithm first scans r to find the nearest neighbor nn. The algorithm then examines every point to find the next oc-event for that point. Function next_oc_event(p, q, nn, t) returns the next oc-event for p after time t with respect

[2]There are cases when a point may only "touch" the event threshold and then move back (closer or farther) to its former state. To simplify the presentation, these cases are not discussed here.
[3]For brevity, only the moving attributes of tuples are shown.

line #	e	Q	W	t
8	w$(b,2)$	{w$(c,6)$,w$(a,7)$}	$\{c\}$	2
10	”	”	$\{b,c\}$	”
11	w$(b,8)$	”	”	”
12	”	{w$(c,6)$,w$(a,7)$,w$(b,8)$}	”	”
8	w$(c,6)$	{w$(a,7)$,w$(b,8)$}	”	6
13	”	”	$\{b\}$	”
8	w$(a,7)$	{w$(b,8)$}	”	7
10	”	”	$\{a,b\}$	”
11	w$(a,13)$	”	”	”
12	”	{w$(b,8)$,w$(a,13)$}	”	”

Fig. 3. A trace of the Simple_Within() algorithm for the example in Figure 1 through time $t = 7$.

```
    procedure Simple_Nearest_Neighbor(r, q, t)
1.    nn ← ∅, done ← false
2.    for each tuple τ ∈ r do
3.      if nn = ∅ ∨ ||P(τ), q, t|| < ||P(nn), q, t|| then
4.        nn ← τ
5.    end for-each
6.    while ¬done do
7.      e ← ∅
8.      for each τ ∈ r ∧ τ ≠ nn do
9.        e′ ←next_oc_event(P(τ), q,P(nn), t)
10.       if e = ∅∨Time(e′) <Time(e) then e ← e′
11.     end for-each
12.     if e ≠ ∅ then t ←Time(e), nn ←Tuple(P(e))
13.     else done ← true
14.   end while
```

Fig. 4. Simple_Nearest_Neighbor().

to query point q, and the nearest neighbor nn. This is a simple computation based on solving the equation $|p(time), q(time)| = |nn(time), q(time)|$ for $time$. Once again, for dimensionality greater than 1 this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. If no such event exists, then next_oc_event() returns a null event with time stamp ∞. Recall that a null event e is denoted by $e = \emptyset$, and the time of a null event is Time$(\emptyset) = \infty$. When the next nn-event comes due, the algorithm again examines every point and computes their oc-events to find the next nn-event. Figure 5 shows a trace of the event processing portion of the algorithm (lines 6–14) up to time $t = 7.5$ for the 1D example in Figure 1.

Note that Simple_Nearest_Neighbor() does not have a queue for events. This is because the nearest neighbor changes on each nn-event thereby rendering previously computed oc-events irrelevant. The asymptotic running time for Simple_Nearest_Neighbor() is $\mathcal{O}(E_{nn} * N)$ where E_{nn} is the number of nn-events processed throughout the course of the query maintenance, and N is the cardinality of r. The asymptotic running time for Simple_Within() is $\mathcal{O}(N + E_w)$ where E_w is the number of w-events processed throughout the course of the query maintenance.

These simple algorithms serve to illustrate the fundamental differences in processing nn-events vs. processing w-events. The oc-events from which the nn-event is chosen are dependent on the query result which changes when an nn-event occurs. This makes all previous oc-events computed with respect to the old query result irrelevant. This requires computing new oc-events when

line #	τ	e	e'	nn	t
7	-	$\emptyset$	-	c	1
8	a	”	”	”	”
9	”	”	$\mathsf{oc}(a, 6.5)$	”	”
10	”	$\mathsf{oc}(a, 6.5)$	”	”	”
8	b	”	”	”	”
9	”	”	$\mathsf{oc}(b, 4)$	”	”
10	”	$\mathsf{oc}(b, 4)$	”	”	”
12	”	”	”	b	4
7	”	$\emptyset$	”	”	”
8	a	”	”	”	”
9	”	”	$\mathsf{oc}(a, 7.5)$	”	”
10	”	$\mathsf{oc}(a, 7.5)$	”	”	”
8	c	”	”	”	”
9	”	”	$\mathsf{oc}(c, \infty)$	”	”
12	”	”	”	a	7.5

Fig. 5. Simple_Nearest_Neighbor() algorithm trace for the example from Figure 1 through time $t = 7.5$.

the query result changes. On the other hand, pending w-events do not become irrelevant when the query result changes because w-events are independent of the query result.

3. ALGORITHMS

In this section we describe event-driven query maintenance algorithms for k-nn and spatial join queries on moving points with updates. We assume no prior knowledge of updates before the updates occur.

Our Continuous Windowing (CW) algorithm presented in Iwerks et al. [2003] filters the points considered for the k nearest neighbors using a circular window query of fixed size. The size of the circular window must be large enough to have at least k points inside the window throughout the entire life of the query. Consequently, the size of the window needed is data dependent. The size of the window used by the algorithm presented in Iwerks et al. [2003] was chosen in an ad-hoc manner and remained fixed throughout the life of the query. This leads to a window size that may be much larger than needed in many cases.

In this section, we present an improved CW algorithm (iCW) that dynamically adjusts the window size when underflow occurs (see Figure 6). Additionally, we also support updates to the query point in the iCW algorithm which was not supported in the old CW algorithm presented in Iwerks et al. [2003]. The iCW algorithm is compared experimentally with the ETP algorithm from Iwerks et al. [2003] which has also been enhanced to support updates. A detailed version of the iCW algorithm is given in Appendix A.2, and a detailed version of the ETP algorithm is given in Appendix A.1. In Section 4 we compare the performance of these algorithms using both real data as well as synthetic data. This extends the quality of the experimental observations reported in Iwerks et al. [2003] which used only synthetic data, not real data.

In this section, we also present two new algorithms for the maintenance of spatial join queries on moving points with updates. Maintenance of spatial join

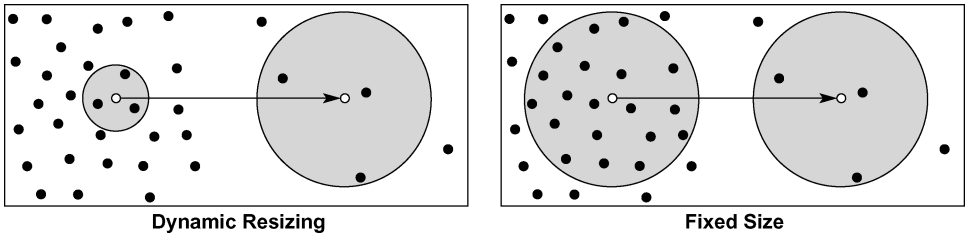


Fig. 6. Dynamic window resizing (left) allows the window to be small when query point starts in a dense region of points. The window becomes larger when the query point moves to a less dense region. A fixed window size (right) must be large enough from the beginning to never underflow when the query point moves.

queries was not addressed in Iwerks et al. [2003] nor in Iwerks et al. [2004]. Iwerks et al. [2004] addressed spatial semi-join queries. A semi-join query can be thought of as a massively scaled nearest neighbor query. That is, for every moving point in one relation (a relation of query points) we find the nearest neighbor (or k-nn's) in the other relation. On the other hand, the variant of the spatial join query that we consider in this article finds all pairs of moving points, one from each relation, that are within a given query distance of each other.

The approach for the spatial semi-join algorithm presented in Iwerks et al. [2004] used a filtering technique similar to CW to find all the points within a circular window, or about to enter a circular window centered on each query point called the *fuzzy set*. This circular window is a different size for each query point, and must be resized when underflow occurs (i.e., less than k points are in the window). A spatial join can be conceptualized similarly in that for each query point (all the points in one relation), all the points are found from the other relation that are within a circular window around each query point. The main difference in these problems is that (1) in the spatial join, the circular window is the same size for each query point, and (2) in the spatial join there is no underflow condition to worry about. If there are no points in the circular window, then there are no tuples in the output for a given query point.

Another significant difference between our semi-join algorithm presented in Iwerks et al. [2004] and the spatial join algorithms presented in this article is that the semi-join algorithm presented in Iwerks et al. [2004] does not use a w-event queue. The spatial join algorithms presented in Section 3.3 below use a w-event queue to process w-events as they come due. This is used to keep the query result as up-to-date as possible. On the other hand, the spatial semi-join algorithm in Iwerks et al. [2004] stores w-events in a B-tree index. The index is used to keep track of the fuzzy sets for each query point. The fuzzy sets are examined when nn-events come due, not when w-events come due.

3.1 Extended TP (ETP)

In this section we describe the *extended TP* (ETP) algorithm originally presented in Iwerks et al. [2003] for maintaining the query result of a k-nn query. The ETP algorithm is our extension of the continuous TP KNN algorithm

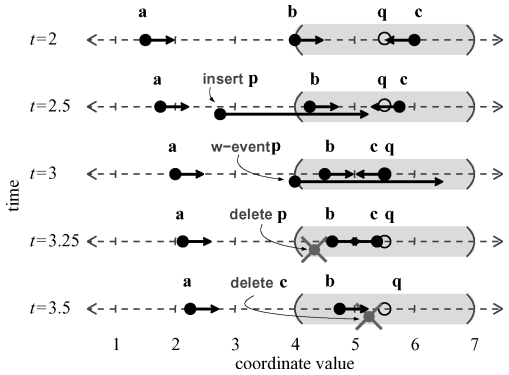


Fig. 7. Example snapshots in time of 1D moving points, events, and updates up to time $t = 3.5$. The arrow length indicates distance traveled in one unit of time. The shaded area indicates the window used by the iCW algorithm presented in Section 3.2.

[Tao and Papadias 2003] wherein we add support for updates to the data points. We further modified the algorithm here slightly from Iwerks et al. [2003] to support updates to the query point. Figure 7 shows a 1D example.[4] The ETP algorithm is given below.[5]

procedure ETP(r, q, k)
1. Find the k nearest neighbors of query point q and the next nn-event using an index on the points in relation r.
2. **while** true **do**
3. **if** a tuple is inserted into r **then**
4. **if** the new point in the tuple is closer to the query point q than the current kth neighbor **then**
5. Report the change to the query result (the kth neighbor just became the $k^{\text{th}} + 1$ neighbor).
6. Find the new next nn-event (changing the kth neighbor invalidated the old next nn-event).
7. **else** replace the current nn-event if the new point introduces a sooner nn-event.
8. **else if** a tuple is deleted from r **then**
9. **if** the deleted point is in the query result **then**
10. Compute the k nearest neighbors and report any changes from the previous query result, the new next nn-event, and update the query result.
11. **else if** the deleted point is part of the next nn-event **then**
12. Find the new next nn-event.
13. **end if-then**
14. **else if** q is inserted **then**
15. Find and report the k nearest neighbors of q and the next nn-event.
16. **else if** q is deleted **then**
17. Report the query result as the empty set, and set the next nn-event to the null event.
18. **else if** the next nn-event comes due **then**

[4]In our examples, the query point is not moving, but the ETP and iCW algorithms support moving query points.

[5]We assume that there is no more than one query point at any given time.

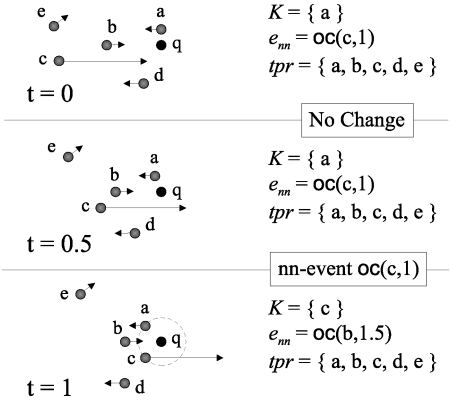


Fig. 8. ETP Events (k = 1).

19. **if** one of the points involved in the next nn-event is not already in the query result **then**
20. Report the change to the query result.
21. **end if-then**
22. Find the new next nn-event.
23. **end if-then**
24. **end while**

The ETP algorithm employs an index to find the k nearest neighbors, or to find the next nn-event (i.e., lines 1, 6, 10, 12, 15 and 22). In particular, the TPR-tree [Saltenis et al. 2000] index can support k nearest neighbor queries and next nn-event queries on moving point objects (see Sections 5.1.1, 5.2, and 5.4 for more details). The query result can be derived without resorting to the index in the case of line 5 where we already have the $k+1$ nearest neighbors, the old query result, and the new point. Likewise, the query result can be computed without using the index in line 19 since this is when a new point enters the set of k nearest neighbors, and the old kth neighbor becomes the $k+1$ neighbor.

An example nn-event and updates are given in Figures 8 and 9, respectively, where K is the k-nn query result, e_{nn} is the nn-event, and tpr are the points in the index (see Section 2.3 for notational conventions).

A trace of the ETP algorithm is given in Figure 10. At time $t = 2.5$, point $\mathbf{p} = \mathsf{pt}(2.75, 2.5, 2.5)$ is inserted. The oc-event for the new point $\mathbf{p}$ comes before the oc-event of any other point in the entire data set, so $\mathbf{p}$ becomes the new nn-event $e_{nn} \leftarrow \mathsf{oc}(\mathbf{p}, 3.5)$. Point $\mathbf{p}$ is deleted at time $t = 3.25$. This means the oc-events of the points in the data set must be examined again to find the next to occur. In this case, the oc-event for $\mathbf{b}$ becomes the next nn-event. When the current nearest neighbor $\mathbf{c}$ is deleted at time $t = 3.5$, point $\mathbf{b}$ becomes the new nearest neighbor. The only other point in the data set now is point $\mathbf{a}$ with nn-event $e_{nn} = \mathsf{oc}(\mathbf{a}, 7.5)$.

3.2 Improved Continuous Windowing KNN (iCW)

In this section we describe the *Improved Continuous Windowing* k-nn algorithm (iCW). The iCW is an improvement over the old CW algorithm presented

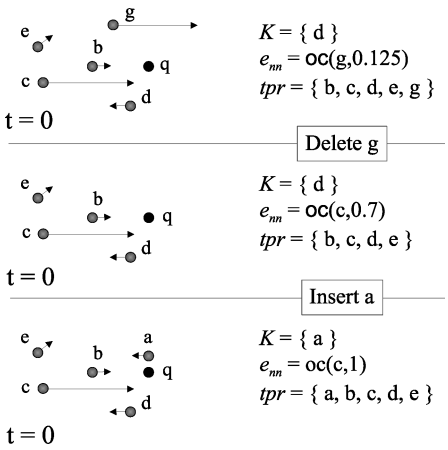


Fig. 9. ETP Updates (k = 1).

t	update or event	nn	e_{nn}
2	-	**c**	oc(**b**, 4)
2.5	insert **p**	**c**	oc(**p**, 3.5)
3.25	delete **p**	**c**	oc(**b**, 4)
3.5	delete **c**	**b**	oc(**a**, 7.5)

Fig. 10. ETP trace for the example in Figure 7. Column 1 indicates the current time for each row. Column 2 shows the update or event (if any) at time t. Column 3 shows the nearest neighbor (nn) at time t. Column 4 gives the next nn-event e_{nn}.

in Iwerks et al. [2003]. Here we expand on the work done in Iwerks et al. [2003] by introducing the ability to dynamically resize the circular query window when too few points are contained in it. This leads to an overall smaller window size resulting in fewer w-events on the queue. In addition, the iCW algorithm also extends the CW algorithm to support updates to the query point.

The iCW algorithm is based on the observation that window queries are easier to maintain on moving points than k-nn queries, because w-events are fundamentally less expensive to process than oc-events (see Section 2 for more background on event processing). The iCW algorithm filters points to be considered as nearest neighbor candidates using a within query around the query point. The within query must select at least $k + 1$ points as otherwise underflow occurs. Figure 11 gives an illustrative example. Only the points within this circular window are considered when finding the k nearest neighbors and for computing the next nn-event. The iCW algorithm is given below.[6]

[6]We assume that there is no more than one query point at any given time.

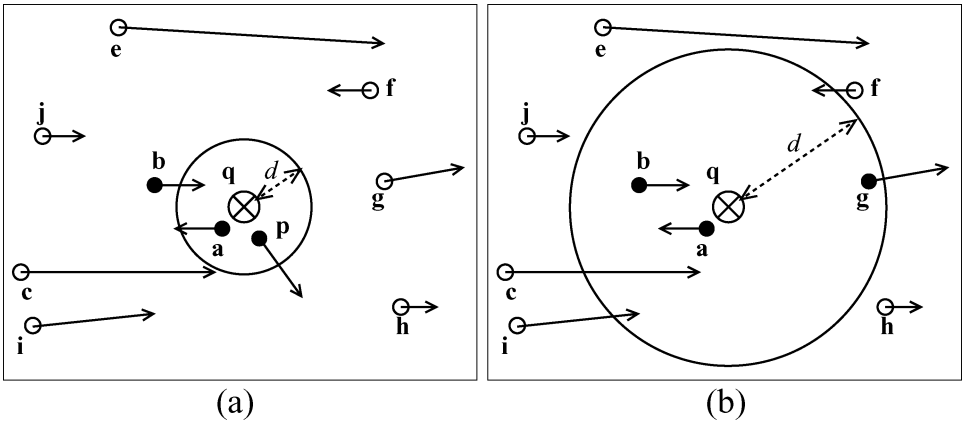


Fig. 11. 2D example illustrating the iCW approach, where $\otimes$ is the query point $\mathbf{q}$, d indicates the radius of the circular window, $\bullet$ denote points inside the circular window, and $\circ$ denote points outside the window. Figure (a) shows an example just before point $\mathbf{p}$ is deleted. Figure (b) shows the same example just after point $\mathbf{p}$ is deleted leading to underflow and subsequent expansion of the circular window.

procedure iCW(r, q, k, x)
1. Create a circular window containing $k + x$ points centered on query point q in relation r.
2. Compute and report the k–nn result, and compute the next nn-event (from the points in the window).
3. Enqueue the next w-event, if any, for each point in r.
4. **while** true **do**
5. **if** a tuple is inserted into r **then**
6. Enqueue the next w-event, if any, for the new point in the new tuple.
7. **if** the new point is inside the circular window **then**
8. **if** the new point is closer to q than the current kth neighbor **then**
9. Report the change in the k–nn result, and compute the next nn-event.
10. **else if** the new point introduces a sooner nn-event **then**
11. replace the next nn-event.
12. **end if-then**
13. **end if-then**
14. **else if** a tuple is deleted from r **then**
15. **if** the deleted point is inside the circular window **then**
16. **if** the deleted point causes the circular window to underflow **then**
17. Enlarge the circular window to contain $k + x$ points.
18. Compute and report the k–nn result, and compute the next nn-event.
19. Clear the queue and enqueue the next w-event, if any, for each point in r.
20. **else**
21. Remove any w-event involving the deleted point from the queue.
22. **if** the deleted point is in the k–nn result **then**
23. Report the change in the k–nn result, and compute the next nn-event.
24. **else if** the deleted point is involved in the next nn-event **then**
25. Compute a new next nn-event.
26. **end if-then**
27. **end if-then**
28. **else** remove any w-event involving the deleted point from the queue.

29. **else if** q is inserted **then** initialize the k–nn query result and data structures, as in lines 1, 2, and 3.
30. **else if** q is deleted **then** clear the k–nn query result and data structures.
31. **else if** a w-event comes due **then**
32. **if** the w-event is an enter event **then**
33. Enqueue the exit w-event for the point entering the circular window.
34. Replace the next nn-event if the entering point introduces an nn-event that will occur sooner.
35. **else if** the w-event is an exit event that causes the window to underflow **then**
36. Enlarge the circular window to contain $k + x$ points.
37. Compute the next nn-event (the k-nn result won't change).
38. Clear the queue and enqueue the next w-event, if any, for each point in r.
39. **end if-then**
40. **else if** the next nn-event come due **then**
41. Report the change to the k-nn result, and compute the next nn-event.
42. **end if-then**
43. **end while**

When the circular window is created or enlarged (lines 1, 17, 29, and 36), its radius is computed as the average distance to the $k + x$ closest points and the $k + x + 1$ closest point to the query point q. For example, $k = 1$ and $x = 2$ in Figure 11. These points are found by scanning the base relation and retaining the $k + x + 1$ closest points. All the points in the window, or that will enter the window in the future are kept in memory so that the k-nn query result and the next nn-event can be computed from them without going back to the index (lines 2, 9, 11, 18, 23, 25, 29, 34, 37, 41). A second scan of the base relation is needed whenever the window is created or enlarged to compute the next w-event for each point in r (lines 3, 19, 29, 38). The w-events can not be computed during the first scan since the size of the circular window is not known until the first scan is complete. Although there can be more than one future w-event for any point in r, only the next w-event to occur for any given point is enqueued. There is at most one w-event for each point in r on the queue. If a point in r is outside the window, and will not enter the window sometime in the future, then no w-event for that point will be enqueued. In general, a smaller window leads to a smaller event queue since fewer points will enter it, so we can expect the number of events on the queue to be significantly less than the number of points in r. Note that there is only one pending nn-event at any given time, whereas there are multiple pending w-events on the queue. Additionally, we assume there can be 0 or at most 1 query point q at any given time (line 29).

Figure 12 shows an example trace of the iCW algorithm. At time $t = 2.5$, a tuple with moving point $\mathbf{p} = \mathsf{pt}(2.75, 2.5, 2.5)$ is inserted into relation r. Location $\mathbf{p}(2.5)$ is farther from $\mathbf{q}$ than $d = 1.5$ so a new w-event $\mathsf{w}(\mathbf{p}, 3)$ is added to the priority queue Q. At time $t = 3$, the w-event is processed and $\mathbf{p}$ is added to the set of points within the circular window W. Since the oc-event of $\mathbf{p}$ comes before the oc-event of any other point in W, it becomes the new nn-event $e_{nn} \leftarrow \mathsf{oc}(p, 3.5)$. At time $t = 3.25$, $\mathbf{p}$ is deleted, and the new nn-event, $e_{nn} \leftarrow \mathsf{oc}(b, 4)$, is computed by examining the remaining elements of $W = \{\mathbf{b}, \mathbf{c}\}$. At time $t = 3.5$, the nearest neighbor $\mathbf{c}$ is deleted.

t	update or event	nn	e_{nn}	Q
2	-	$\mathbf{c}$	oc($\mathbf{b}$, 4)	⟨w($\mathbf{c}$, 6), w($\mathbf{a}$, 7),w($\mathbf{b}$, 8)⟩
2.5	insert $\mathbf{p}$	$\mathbf{c}$	oc($\mathbf{b}$, 4)	⟨w($\mathbf{p}$, 3),w($\mathbf{c}$, 6), w($\mathbf{a}$, 7),w($\mathbf{b}$, 8)⟩
3.0	w-event	$\mathbf{c}$	oc($\mathbf{p}$, 3.5)	⟨w($\mathbf{p}$, 4.2),w($\mathbf{c}$, 6), w($\mathbf{a}$, 7),w($\mathbf{b}$, 8)⟩
3.25	delete $\mathbf{p}$	$\mathbf{c}$	oc($\mathbf{b}$, 4)	⟨w($\mathbf{c}$, 6), w($\mathbf{a}$, 7),w($\mathbf{b}$, 8)⟩
3.5	delete $\mathbf{c}$	$\mathbf{b}$	$\emptyset$	⟨w($\mathbf{a}$, 7),w($\mathbf{b}$, 8)⟩

Fig. 12. A trace of the iCW algorithm applied to the example in Figure 7 where $d = 1.5$. Column 1 indicates the current time for each row. Column 2 shows the update or event (if any). Column 3 shows the nearest neighbor (nn). Column 4 gives the next nn-event e_{nn}. Column 5 shows the w-events on Q.

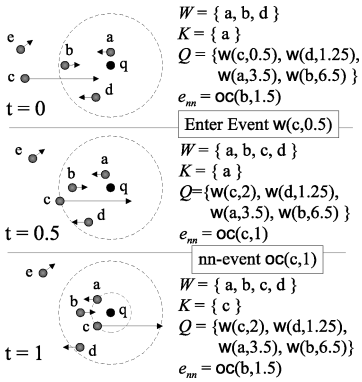


Fig. 13. iCW Events (k = 1).

Figure 13 gives another example illustrating how events are processed by the iCW algorithm. Point $\mathbf{q}$ is the query point. The set of points inside the circular window is W. The query is $k = 1$, so the result set has one element $K = \{\mathbf{a}\}$. Q is the w-event queue, and e_{nn} is the next nn-event. The length of the arrows indicate the direction and distance each point will move in one unit of time. At time $t = 0.5$, point $\mathbf{c}$ enters the query circle. When this happens, point $\mathbf{c}$ is added to set W, and its exit event is enqueued in Q. The next nn-event, e_{nn} is also replaced since $\mathbf{c}$ will become the next nearest neighbor before $\mathbf{b}$. At time $t = 1$, the nn-event oc(c, 1) comes due. The result set K is updated, and the next nn-event is computed.

Figure 14 illustrates how updates affect the data structures. First, the point $\mathbf{g}$ is deleted. Since $\mathbf{g}$ is in the circular window, it is removed from W. A check is made to see if $\mathbf{g}$ was involved in the next nn-event e_{nn}. Since it was, a new nn-event must be computed from the points in the circular window. If $\mathbf{g}$ was not within the circular window at the time of deletion, then the nn-event would not need to be checked. The example also shows an insertion of point $\mathbf{a}$ processed after the deletion of point $\mathbf{g}$. Since point $\mathbf{a}$ is closer to the query point than the

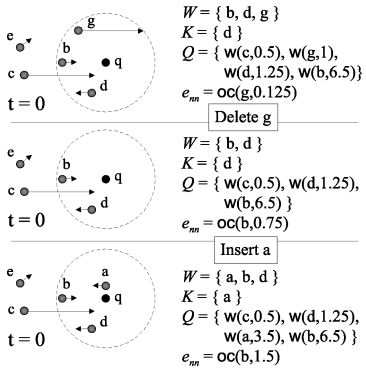


Fig. 14. iCW Updates (k = 1).

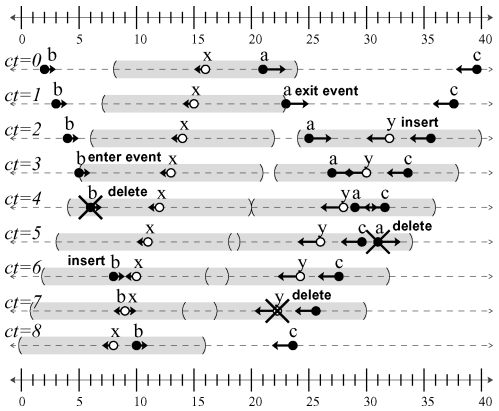


Fig. 15. Example spatial join of 1-dimensional moving points.

previous nearest neighbor **d**, point **a** becomes the new nearest neighbor, and a new nn-event is computed.

3.3 AE and NE Algorithms

In this section, we use the general event-based query maintenance algorithm approach to maintain spatial join queries on moving points while the base relations are updated throughout the duration of the query. Given relations $l(L)$ and $r(R)$, a spatial join is the join $l \bowtie_{\|\alpha_l,\alpha_r,\text{now}\|\leq d} r$ where $\alpha_l \in L$ and $\alpha_r \in R$ are the moving point spatial attributes in their schemas, $d \geq 0$, and $\|\alpha_l, \alpha_r, \text{now}\|$ is the Euclidean distance metric at the current time. As with the k-nn algorithms, we assume no previous knowledge about the updates prior to the arrival of the update.

Figure 15 is an example of the dynamics involved in the maintenance of the continuous spatial join on two relations, l and r, of 1-dimensional moving points. Initially, $l = \{a, b\}$ and $r = \{x\}$. Each row in the figure shows the state of the moving points, as well as the modifications to the underlying relations, as the current time ct advances. The join distance is illustrated by the shaded

time ct	event or update	J
0	initial join	$\{\langle a, x\rangle\}$
1	exit event	$\{\}$
2	insert y into r	$\{\langle a, y\rangle, \langle c, y\rangle\}$
3	enter event	$\{\langle a, y\rangle, \langle c, y\rangle, \langle b, x\rangle\}$
4	delete b from l	$\{\langle a, y\rangle, \langle c, y\rangle\}$
5	delete a from l	$\{\langle c, y\rangle\}$
6	insert b into l	$\{\langle c, y\rangle, \langle b, x\rangle\}$
7	delete y from r	$\{\langle b, x\rangle\}$
8	no change	$\{\langle b, x\rangle\}$

Fig. 16. Trace of the example join query given in Figure 15. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J after the event or update occurs.

areas extending for 8 distance units around points in relation r. Points b, and x are moving at speed 1. Points a, c, and y are moving at speed 2. The arrows indicate their direction of movement. Although the example is 1-dimensional, the algorithms presented in this paper work for any dimension $D > 0, D \in \mathbb{N}$. Figure 16 shows how updates and events affect the query result as time advances. For brevity, only the moving point attributes of tuples are presented.

Recall from above that event-based query maintenance involves the processing of events and updates to maintain a consistent query result as time advances. For example, in the sample query in Figure 15, events occur at times 1 and 3 resulting in a change to the query result. The occurrences of one or more events are computed in advance and placed in a priority queue sorted by time. Thus, when the time of the event arrives, the query result is modified, and more events are possibly computed. Updates may also change the query result. For example, in Figure 15, updates occur at times 2, 4, 5, 6, and 7. The query result can be updated using techniques similar to incremental view maintenance techniques [Gupta et al. 1993] when an update occurs. Additionally, the events in the queue may also be modified when updates occur.

In this section we present the *All Events* (AE) and the *Next Event* (NE) algorithms. For both algorithms, the initial join is computed, then w-events that will cause the query result to change are computed and enqueued in a temporal priority queue. Time is divided into segments of equal length called *event generation cycles*. Only events that occur during the current event generation cycle are considered for processing. The first event generation cycle starts at the current time (i.e., now), and lasts for some finite time interval. The basic differences between AE and NE are the times at which events are computed and placed on the queue. The AE algorithm enqueues all events up to the end of the current event generation cycle. The NE algorithm enqueues only the next event to occur for each moving point in one designated join relation (by convention this is the left relation l). Neither algorithm enqueues any event that occurs beyond the end of the current event generation cycle.

3.3.1 *All Events* (AE). The AE algorithm maintains all currently pending events on the queue that occur between the current time and the end of the current event generation cycle. This means there can be up to $|l| \times |r|$ elements on the queue. This algorithm can be thought of as an extension of the continuous spatial join (CSJ) algorithm for future queries presented in Tao and Papadias

time ct	event or update	J	Q
0	initial join and event gen.	$\{\langle a, x\rangle\}$	$\langle \mathsf{w}(a, x, 1),\ \mathsf{w}(b, x, 3)\rangle,\ \mathsf{w}(b, x, 11)\rangle$
1	exit event	$\{\}$	$\langle \mathsf{w}(b, x, 3),\ \mathsf{w}(b, x, 11)\rangle$
2	insert y into $j.r$	$\{\langle a, y\rangle, \langle c, y\rangle\}$	$\langle \mathsf{w}(b, x, 3),\ \mathsf{w}(a, y, 5.75),\ \mathsf{w}(b, y, 8.\overline{6}),\ \mathsf{w}(b, x, 11)\rangle$
3	enter event	$\{\langle a, y\rangle, \langle c, y\rangle, \langle b, x\rangle\}$	$\langle \mathsf{w}(a, y, 5.75),\ \mathsf{w}(b, y, 8.\overline{6}),\ \mathsf{w}(b, x, 11)\rangle$
4	delete b from $j.l$	$\{\langle a, y\rangle, \langle c, y\rangle\}$	$\langle \mathsf{w}(a, y, 5.75)\rangle$
5	delete a from $j.l$	$\{\langle c, y\rangle\}$	$\langle\rangle$
6	insert b into $j.l$	$\{\langle c, y\rangle, \langle b, x\rangle\}$	$\langle \mathsf{w}(b, y, 8.\overline{6}),\ \mathsf{w}(b, x, 11)\rangle$
7	delete y from $j.r$	$\{\langle b, x\rangle\}$	$\langle \mathsf{w}(b, x, 11)\rangle$

Fig. 17. Trace of the *All Events* (AE) algorithm for the example from Figure 15 where the event generation cycle is 12 time units long. The first cycle starts at time $ct = 0$ and ends at time $ct = 12$. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J, and column 4 shows the contents of the event queue Q for the AE approach after each event or update is processed.

[2003] to support updates. The approach of this extension is to run the CSJ future query for some finite time in the future to find all the events in that time period and place them on an event queue. If an update occurs, modify the join result, and modify the event queue to reflect the changes introduced by the update. The AE algorithm is given below. A more detailed version of the AE algorithm is given in Appendix A.3.

procedure AE(l, r, d)
1. Compute and report the initial join result.
2. **while** true **do**
3. Enqueue *all* w-events that will occur in the current event generation cycle.
4. **while** not at the end of the current event generation cycle **do**
5. **if** a w-event comes due **then**
6. Report changes to the join result.
7. **else if** a tuple is inserted into l (r) **then**
8. Join the new point with r (l) and report the result as inserted to the join result.
9. Find *all* the w-events between the new point and points in r (l) that occur during the current event generation cycle and enqueue them.
10. **else if** a tuple is deleted from l (r) **then**
11. Join the deleted point with r (l) and report the result as deleted from the join result.
12. Remove all w-events involving the deleted point from the queue.
13. **end if-then**
14. **end while**
15. Advance to the next event generation cycle and enqueue all the w-events that will occur in it.
16. **end while**

It is assumed that an index is maintained for each relation on its moving points (e.g., the TPR-tree [Saltenis et al. 2000]). The indexes are used to compute joins (lines 1, 8, and 11), and w-events (lines 3, 9, and 15).

A trace of this algorithm is given in Figure 17, for the example presented in Figure 15. The figure shows w-events placed on priority queue Q sorted by time in the context of the spatial join query $J = (l \bowtie_{pred} r)$, where the join predicate is $pred = (\|\alpha_l, \alpha_r\| \leq d)$. A w-event is denoted by $\mathsf{w}(p_l, p_r, t)$, where p_l represents an instance of moving point attribute α_l, p_r represents an instance of moving point attribute α_r, and t is the time at which p_l and p_r move

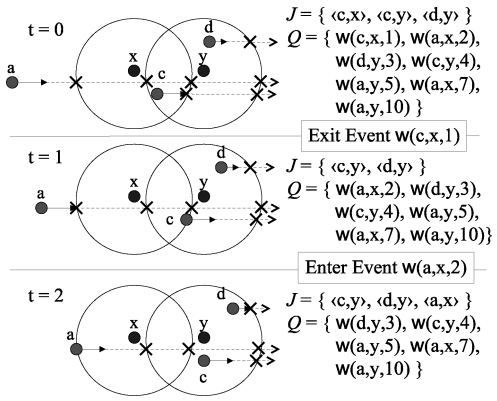


Fig. 18. AE Events.

to be at the distance d of each other. For example, the enter event in line $ct = 3$ of Figure 15 is denoted as $\mathsf{w}(b, x, 3)$, and the exit event in line $ct = 1$ of Figure 15 is denoted by $\mathsf{w}(a, x, 1)$. Notice that when y is inserted into r at time $ct = 2$, there is no exit event inserted in the queue for point b. This is because the exit event for b and y does not occur until the next event generation cycle. Similarly, the enter and exit events between point c and point x do not occur during the current event generation cycle. Note also that no exit event between point c and point y exists because the points have the same velocities.

Figure 18 shows a 2D example illustrating how events are handled in the AE approach. The figure shows the spatial join of two data sets. Relation r initially contains the points $\{x, y\}$. Relation l contains the points with names from the beginning of the alphabet, such as $\{a, b, c, d\}$. The join distance is indicated by the circles around the points in relation r. J is the query result. Q is the w-event queue. All w-events between a point in l, and a point in r, occurring in an immediate and finite future time period, are computed in advance an placed on Q. At time $t = 1$, event $\mathsf{w}(c, x, 1)$ occurs. After dequeuing, the type of event (enter or exit) is checked. In this case it is an exit event, so the pair $\langle c, x\rangle$ is removed from the query result J. At time $t = 2$, enter event $\mathsf{w}(a, x, 2)$ results in the addition of pair $\langle a, x\rangle$ to J. Figure 19 illustrates how updates affect the data structures in the AE approach.

3.3.2 *Next Event* (NE). The NE algorithm enqueues just the next pending w-event for each moving point from one designated join relation (relation l by convention) that occurs during the current event generation cycle. In other words, the queue Q contains at most one event, the next w-event, for each point in relation l. This can be thought of as a scaled up version of the window query of the iCW algorithm the differences being that (1) there is no notion of underflow, (2) the window size never changes throughout the life of the query, and (3) the NE algorithm enqueues only the next w-event for a given point if it occurs during the current event generation cycle. The iCW algorithm, on the other hand, enqueues the next w-event no matter how far into the future it

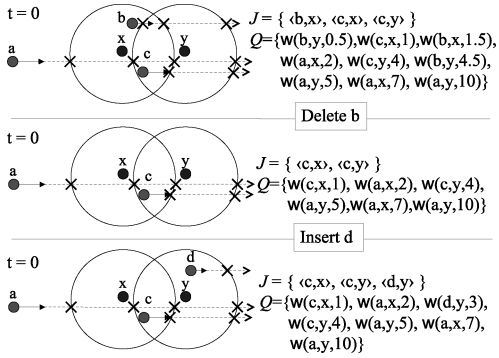


Fig. 19. AE Updates.

may occur. The NE algorithm is given below. A more detailed version of the NE algorithm is given in Appendix A.4.

procedure NE(l, r, d)
1. Compute and report the initial join result.
2. **while** true **do**
3. Enqueue the *next* w-event for each point in l occurring during the current event generation cycle.
4. **while** not at the end of the current event generation cycle **do**
5. **if** a w-event comes due **then**
6. Report changes to the join result.
7. Enqueue the *next* w-event between the point in l involved in the current event and points in r if the event occurs during the current event generation cycle.
8. **else if** a tuple is inserted into l **then**
9. Join the new point with r and report the result as inserted to the join result.
10. Enqueue the *next* w-event between the new point and points in r if they occur during the current event generation cycle.
11. **else if** a tuple is inserted into r **then**
12. Join the new point with l and report the result as inserted to the join result.
13. **for each** w-event e between the new point and points $p_l \in l$ **do**
14. **if** e is now the next w-event for p_l and occurs in the current event gen. cycle **then**
15. Enqueue it replacing any old w-event for p_l on the queue.
16. **end if-then**
17. **end for-each**
18. **else if** a tuple is deleted from l **then**
19. Join the deleted point with r and report the result as deleted from the join result.
20. Remove any w-event involving the deleted point from the queue.
21. **else if** a tuple is deleted from r **then**
22. Join the deleted point with l and report the result as deleted from the join result.
23. Remove all w-events involving the deleted point from the queue.
24. **for each** point $p_l \in l$ that was involved in the deleted w-events **do**
25. Enqueue the next w-event for p_l if it occurs during the current event generation cycle.

time ct	event or update	J	Q
0	initial join and event gen.	$\{\langle a, x\rangle\}$	$\langle \mathsf{w}(a, x, 1),\ \mathsf{w}(b, x, 3)\rangle$
1	exit event	$\{\}$	$\langle \mathsf{w}(b, x, 3)\rangle$
2	insert y into $j.r$	$\{\langle a, y\rangle, \langle c, y\rangle\}$	$\langle \mathsf{w}(b, x, 3),\ \mathsf{w}(a, y, 5.75)\rangle$
3	enter event	$\{\langle a, y\rangle, \langle c, y\rangle, \langle b, x\rangle\}$	$\langle \mathsf{w}(a, y, 5.75),\ \mathsf{w}(b, y, 8.\overline{6})\rangle$
4	delete b from $j.l$	$\{\langle a, y\rangle, \langle c, y\rangle\}$	$\langle \mathsf{w}(a, y, 5.75)\rangle$
5	delete a from $j.l$	$\{\langle c, y\rangle\}$	$\langle\rangle$
6	insert b into $j.l$	$\{\langle c, y\rangle, \langle b, x\rangle\}$	$\langle \mathsf{w}(b, y, 8.\overline{6})\rangle$
7	delete y from $j.r$	$\{\langle b, x\rangle\}$	$\langle \mathsf{w}(b, x, 11)\rangle$

Fig. 20. Trace of the *Next Event* (NE) algorithm for the example from Figure 15 where the event generation cycle is 12 time units long. The first cycle starts at time $ct = 0$ and ends at time $ct = 12$. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J, and column 4 shows the contents of the event queue Q for the NE approach after each event or update is processed.

26. **end for-each**
27. **end if-then**
28. **end while**
29. Advance to the next event generation cycle queueing the *next* w-event for each point in l that will occur in it.
30. **end while**

Like the AE algorithm, it is assumed that an index is maintained for each relation on its moving points, and is used to compute joins (lines 1, 9, 12, 19, 22), and w-events (lines 3, 7, 10, 13, 25, 29). Updates to relation r incur the most penalty in terms of index overhead. This is because a point in r may be involved in any number of next w-events for the points in l. All the points in relation l that are affected by an update to r must be reexamined, possibly resulting in many index queries and queue updates.

Figure 20 shows a trace of the NE approach with respect to the example given in Figure 15. Note that no w-event for point c in relation l is enqueued because the next w-event for point c does not occur in the current event generation cycle ($0 \leq ct < 12$).

Figures 21 and 22 illustrate how events and updates, respectively, are processed in the NE approach. These are the same examples as shown in Figures 18 and 19. The only difference is in the contents of the data structure Q. Whereas all events in the near future were enqueued in Q for the AE algorithm, only the next event for a given point in l is enqueued in the NE algorithm. As a result, the size of Q is smaller, but when an event is processed, a new event must be found and enqueued. For example in Figure 21, when within event $\mathsf{w}(a, x, 2)$ occurs, the next event $\mathsf{w}(a, y, 5)$ for point **a** must be computed and enqueued.

4. EXPERIMENTS

In this section we present the results of experiments comparing the ETP algorithm with the iCW algorithm, and the AE with the NE algorithm. Experiments were conducted using both real aircraft data and synthetic data. Our primary metric for cost is the number of disk accesses needed to compute and maintain a query. This is because accessing data on disk is several orders of

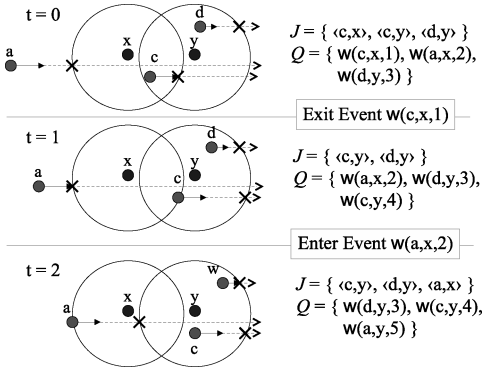


Fig. 21. NE Events.

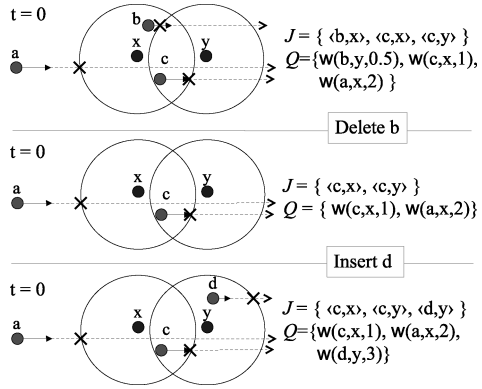


Fig. 22. NE Updates.

magnitude slower than accessing data in memory. Implementation issues are also discussed.

4.1 Data Sets

We used the same data sets as those described in Iwerks et al. [2004]. Both real aircraft flight data and synthetic uniformly distributed data are used in our experiments. One significant difference between the real and synthetic data is in the size of the data set at any given time. The number of points for the synthetic data stays constant, but the real flight data changes as flights land and take off.

Data sets consist of an initial set of moving points described as a linear function of time ($p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$), and updates to the function coefficients ($\overrightarrow{x_0}$, t_0, $\overrightarrow{v}$) over time. A data set is characterized by the mean and standard deviation in the number of moving points (cardinality) at any given time, the period of time covered by the data set, and the average update interval. The average update interval (UI) is the average length of time between updates for any given point.

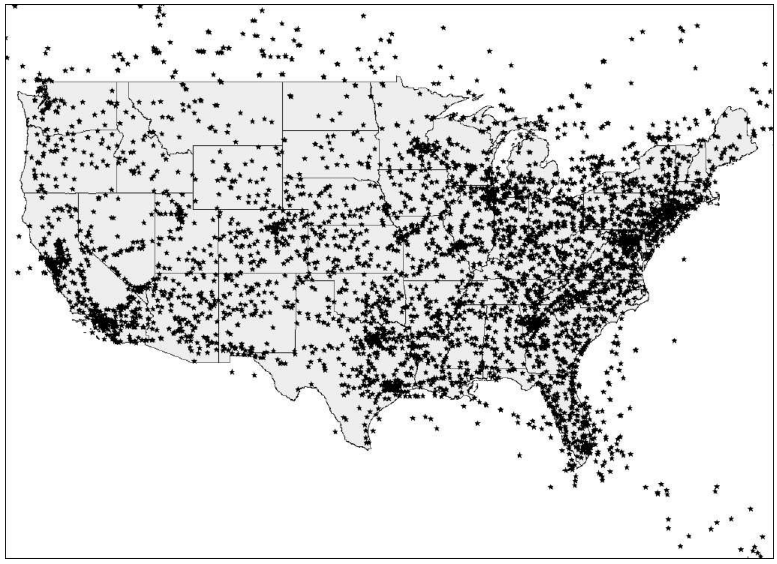


Fig. 23. Snapshot of the aircraft flight data.

All synthetic uniformly distributed data sets were generated using a data generation tool developed by Saltenis et al. [2000]. The synthetic moving points are uniformly distributed over a 1000×1000 coordinate space. The speed of each point is uniformly distributed between 0 and $3/60 = 0.05$ coordinate distance units per time unit. All synthetic moving points are inserted at the start time of the dataset. Updates change the velocity, but not the current location of each point. The number of moving points stays constant. The average update interval (UI) for our synthetic data is 600 time units. Each synthetic data set covers 3600 time units. The UI and speed relative to the size of the coordinate space of the synthetic data were chosen to be similar to the aircraft flight data for comparability.

Real commercial aircraft flight data was acquired as location data sampled at one minute intervals. Figure 23 shows an example snapshot in time to see how the data is clustered. The latitude-longitude of sampled locations were converted to linear functions describing aircraft motion by first applying the Douglas–Peucker line simplification algorithm [Douglas and Peucker 1973] to the 2D latitude-longitude points forming a polyline from earliest to latest sampled location in time. In our application of the Douglas–Peucker algorithm, we used a maximum error bound of $0.0\overline{6}$ degrees. Distortions introduced by the latitude-longitude projection onto the Earth's surface were ignored. The resulting vertices serve as the start locations for each update. Each vertex has an associated time stamp. The line segment to the next vertex divided by the time difference between their time stamps gives the velocity vector for each update. The result was an average update interval of 700–735 seconds. The aircraft data sets cover a window $[20^\circ, 60^\circ]$ latitude by $[-135^\circ, -60^\circ]$ longitude. Since only about 5000 aircraft are in the air at any one time, larger data sets are generated by combining flights on different days during the same time

μ	1097	2212	3334	4453	9021	12690	17106	21020	29551	41855	50822
σ	81.9	165.4	148.3	330.8	680.8	962.4	1293	1591	2223	3386	4381
UI	683.3	699.4	699.3	700.7	712.8	725.1	734.6	729.0	724.6	727.5	726.6

Fig. 24. Statistics on the aircraft data sets. Each column corresponds to a different aircraft data set. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds.

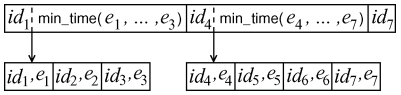


Fig. 25. Example EB-tree with one root node, and two leaf nodes.

period. Figure 24 shows statistics for the aircraft flight data sets used in the experiments.

Finally, subsets of the aircraft data were derived by dividing the flight data into 12 subsets starting at different times along the temporal dimension. The time duration covered by each subset overlapped neighboring subsets. The time domain of each subset was then transformed to start at time 0. The cross product of these subsets was taken to generate pairs of data sets as input for experiments. For the k-nn experiments, the query point is taken randomly from one dataset to find the nearest neighbors in the other data set. For the spatial join experiments, a join was performed on each pair of data sets.

4.2 Implementation

We used the code provided by Saltenis et al. [2000][7] from their original implementation of the TPR-tree. We extended this with implementations of all the index operations used by our algorithms. In Iwerks et al. [2003], we did not find a significant difference between *depth first* and *best first* versions of the TPR-tree index search algorithms (i.e., incremental distance query, next nn-event query), (see Iwerks et al. [2003] for more details) so we use only the *best first* versions of the algorithms in the experiments presented here.

To implement the iCW priority queue, we use a B+-tree variant of a priority search tree called the *Event B-tree* (EB-tree). In our implementation, every point has a unique id. The priority queue is a B+-tree ordered by the point ids. In addition to propagating the min-max key up the B+-tree, the earliest event time of all events in each subtree is also propagated up to the root. The earliest event in the tree is found by following the minimum event time down the branches of the tree to the leaf in which it is stored. The time of the next event can be found by just examining the root node. Figure 25 shows an example EB-tree.

For the spatial join, two B+-trees are used to support deletion of events based on an id from either relation. One B+-tree is sorted by ids (id_l) of objects from the first join relation l, while the second B+-tree is sorted by ids (id_r) of objects from the second join relation r. This results in a mapping of the form $id_l \rightarrow \{id_r, t\}$, and $id_r \rightarrow \{id_l, t\}$. Like the EB-tree, both the minimum

[7]A special thanks to Saltenis et al. and Tao et al. for making their code available for use and study.

and maximum *id* keys and the minimum event time are propagated up the trees.

Both the TPR-tree and B+-trees were implemented using the generalized search tree (GiST) [Hellerstein et al. 1995] version 0.9beta1 code. The code was compiled using gcc 2.95. The experiments were run on several VLSI 80686 CPU based machines running Linux.

4.3 ETP and iCW Experiments

4.3.1 *Performance Issues for ETP and iCW.* Analysis of algorithms for kinematic data is difficult without making many simplifying assumptions. Performance is dependent on many factors such as data set size, location distribution, velocity distribution, distribution of updates over space, and update frequency distribution. Instead of attempting a rigorous analysis on an overly constrained subset of these factors, this section discusses some key performance issues of the ETP and iCW algorithms, and how these factors play a part in the performance of each algorithm.

We assume that for large data sets, the majority of the data is stored on disk. Accesses to disk are orders of magnitude slower than accesses to memory, so cost is measured in number of disk accesses. For the sake of this discussion, we assume that all moving point data and query points share the same location, velocity, and update rate distributions. Ignoring esoteric cases, assume that all points are moving relative to the query point, and that they are not all moving in the same direction and at the same speed. For the purpose of this discussion, we assume a B+-tree is used to implement the iCW event priority queue (Q_{iCW}), and a TPR-tree for the ETP algorithm index.

Initial Build. Both methods require at least one initial scan of some relation r. ETP scans r to build the TPR-tree index. iCW scans the relation once to find the query result and then a second time to find pending w-events.

Data Structure Size. Let n be the size of the point data set. The asymptotic upper bound for the TPR-tree, and Q_{iCW} data structures is $\mathcal{O}(n)$. The lower bounds for each data structure are not the same. The best case for iCW is when no points will enter the circular window in the future. Let W be the set of points inside the circular window. In this case, the only points involved in events in Q_{iCW} are those inside the circular window giving a lower bound of $\Omega(|W|)$. In general, the size of Q_{iCW} is proportional to the selectivity of the circular window over all time [Tao et al. 2003b]. We assume that $|W| \ll n$. Given this assumption, it is likely that the size of Q_{iCW} will be much smaller than the TPR-tree.

Scanning the Base Relation. Let UI be the average time period between two updates for a single object. By experimentation, Saltenis et al. [2000] determined that the performance of the TPR-tree degrades after time UI because almost all the entries have been updated by that time thereby causing the ability of the index to distinguish between the data objects to degenerate due to increasing overlap of the index nodes. They conclude that the TPR-tree should be rebuilt when time UI is reached. Rebuilding the TPR-tree requires scanning the base relation, or scanning the entire TPR-tree to construct a new tree. Two scans of the base relation are required for the iCW algorithm each time the

query point is updated or the query circle underflows. For the iCW algorithm, assume that the expected time between updates to the query point is the same as that for the TPR-tree (i.e., UI). The number of scans needed overall for the iCW algorithm depends heavily on the characteristics of the data. We assume that updates to any given point happen on average once every UI time units. Given this assumption, we would expect the iCW algorithm to scan the base relations at least twice as many times as the TPR-tree.

Number of Events. Only the nn-events are processed in the ETP approach. This makes the ETP approach optimal in the number of events processed throughout the course of a query. There is only one event pending at any one time. iCW processes additional w-events. The number of w-events over the course of a query, or on the event queue at any one time, depends on the selectivity of the circular window and on the motion characteristics of the data.

Cost of Events. The cost of processing each event for each method is not the same. The iCW algorithm requires no disk accesses to examine other points when either a w-event or an nn-event is processed because all the points that need to be examined are in set W which is already in main memory. The only cost in disk accesses for the iCW algorithm is in updating the event queue which is $\mathcal{O}(\log n)$ (i.e., to delete an old w-event and, if needed, to insert a new w-event in the B+-tree). In the ETP approach, the cost of processing an event is $\mathcal{O}(n)$ for the *incremental distance query* [Hjaltason and Samet 1999] (see Section 5.1.1). The worst case for the *incremental distance query* happens when all points are at the same distance from the query point which is in practice an unlikely case for low dimensional data sets. The ETP method has no event queue. The entire cost of the ETP method lies in the TPR-tree operations.

4.3.2 *Results for ETP and iCW.* Default parameter values for each experiment, unless otherwise specified in the description of an individual experiment, are as follows. The duration of each experiment is 1000 time units (the time duration in Iwerks et al. [2003] was only 60 time units). Disk page size is 4096 bytes. Experiments that do not vary by k use $k = 1$. Experiments that do not vary by data set size are run on a data set of 50000 points for uniform data, and an average of 50822 points for aircraft data. The number of pages in the cache for the TPR-tree index is 64 pages. The event queue disk cache is 8 pages. Each cache uses a least recently used (LRU) page replacement policy. For the iCW algorithm, the number of extra points to find (parameter x in the iCW algorithm, Section 3.2) is $x = 4$. Disk accesses are computed as an average over 100 experiments for a given set of parameters. All results report statistics accumulated after the initial loading of the data structures, and flushing of the disk-based data structure caches.

For the iCW algorithm, whenever there is a query point update or an underflow of the window, the base relation is scanned twice. We assume one page access for every 186 moving point objects in the data set at the time the relation is scanned. This number was derived as follows. A 2D kinematic point is represented by 5 floating point numbers of 4 bytes each, two floats for the start location coordinates, two for the velocity vector, and one for the start time. Each moving point also has a unique identifier represented by a 2 byte short. This

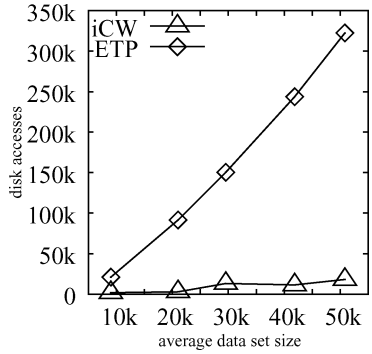


Fig. 26. Number of disk accesses vs. data set size for aircraft data.

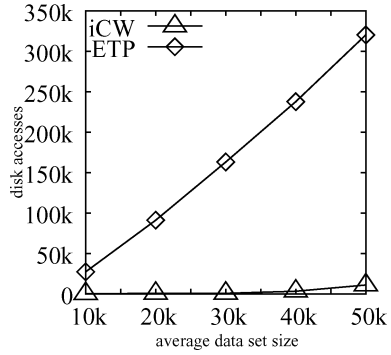


Fig. 27. Number of disk accesses vs. data set size for uniform data.

gives a total of $(5 * 4 \text{ bytes}) + 2 \text{ bytes} = 22$ bytes per object. Each page is 4096 bytes. Therefore $\lfloor 4096 \text{ bytes per page}/24 \text{ bytes per object} \rfloor = 186$ objects per page.

The purpose of the first experiment is to determine which algorithm, iCW or ETP, performs better in terms of disk accesses for different data set sizes. The results are given in Figures 26 and 27 for aircraft and uniform data respectively. For the aircraft data set, the iCW algorithm has over 17 times fewer disk accesses than the ETP algorithm for the largest data sets tested. For the uniform synthetic data, the iCW algorithm has over 29 times fewer disk accesses than the ETP algorithm for the largest data sets tested.

An additional experiment was conducted to examine the relative performance of the two algorithms when there are no updates. The results are given in Figure 28 for both an aircraft data set size of approximately 50k points, and uniform data set size of 50k points. The results were obtained by simply ignoring all subsequent updates once the experiment started and processing events only. Two pairs of vertical bars are shown for each data set. The black bar on the left indicates the number of disk accesses for the ETP algorithm, and the white bar on the right indicates the number of disk accesses for the iCW algorithm. When there are no updates, the ETP algorithm has 72 times fewer disk accesses than the iCW algorithm for aircraft data, and 57 times fewer disk

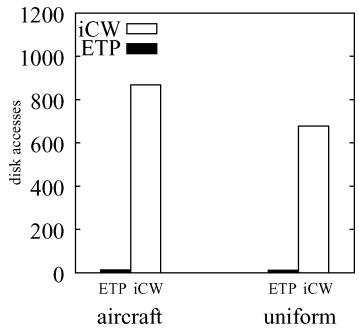


Fig. 28. Number of disk accesses when there are no updates.

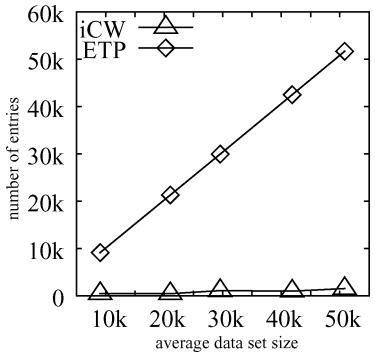


Fig. 29. Number of entries in the corresponding data structures vs. data set size for aircraft data.

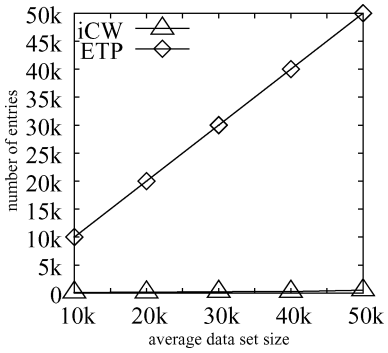


Fig. 30. Number of entries in the corresponding data structures vs. data set size for uniform data.

accesses for uniform data. This is because the ETP algorithm processes just nn-events, whereas the iCW algorithm processes both w-events and nn-events. However, the number of disk accesses even for the iCW algorithm, is relatively small ($<$1000).

The performance differences shown in Figures 26 and 27 are likely due to the size of the disk based structures obtained running each algorithm. Figures 29 and 30 plot the number of data structure entries entries vs. the data set size for each algorithm. This supports our suspicion discussed in Section 4.3.1 that the

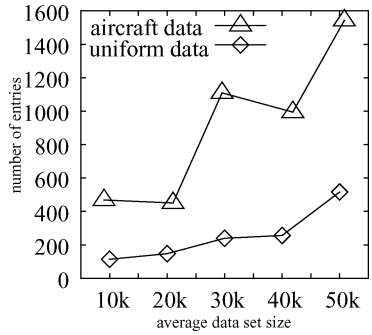


Fig. 31. Number of entries in the data structure vs. data set size for the iCW algorithm on uniform and aircraft data.

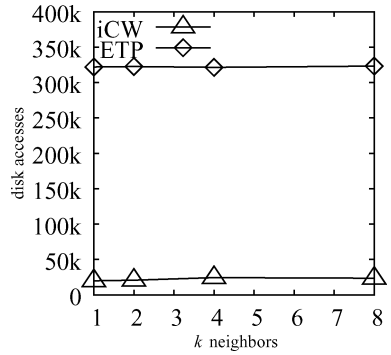


Fig. 32. Number of disk accesses vs. number of neighbors for aircraft data.

event queue for the iCW algorithm will generally be much smaller than the spatial index used by the ETP algorithm. This also seems to indicate that the cost of rescanning the base relations when the query point is updated, or when the iCW circular window underflows, is much less than the overhead of maintaining and querying the TPR-tree index. For the largest data set, the iCW event queue has 33 times fewer entries than the ETP TPR-tree index for the aircraft data, and 96 times fewer entries for the uniform data.

Figure 31 shows the same results as in Figures 29 and 30, but for only the iCW algorithm to emphasize how the different data set types affect the size of the event queue. The figure indicates that the non-uniformly distributed aircraft data results in more events placed on the queue on average. This may be due to initially large windows moving into regions of higher density data. Extending the algorithm to shrink the window size when an overflow level is reached may counteract this effect; however, the tradeoff would be additional scans of the base relations when resizing the window.

The purpose of the next experiment is to determine how the number of neighbors sought by the queries affect performance. The results are given in Figures 32 and 33 for aircraft and uniform data respectively. The results show that an increase in the number of neighbors has a relatively small effect on the overall performance.

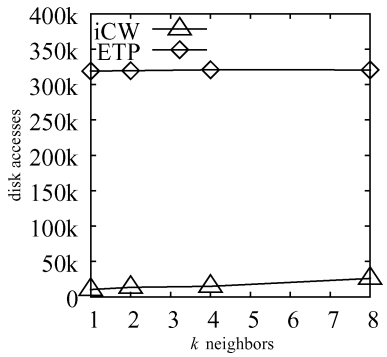


Fig. 33. Number of disk accesses vs. number of neighbors for uniform data.

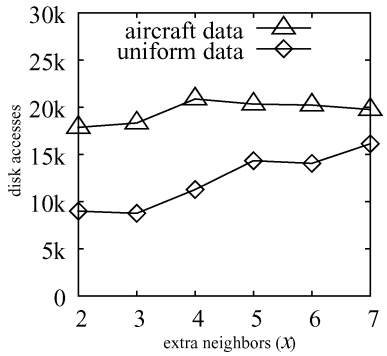


Fig. 34. Number of disk accesses vs. extra neighbors for the iCW algorithm.

The next experiment is designed to give us some insight into the tuning of the iCW algorithm by varying the number of extra points contained in the circular window when it is initially created or enlarged. Recall that parameter x of the iCW algorithm controls this behavior where the number of initial points in the window is $k + x$. Figure 34 gives the results of this experiment. For uniformly distributed data, increases in x seem to degrade the performance somewhat, but for the nonuniformly distributed aircraft data the performance remains relatively constant at a value of $x = 4$ or greater. This may reflect a balance between the selectivity of larger windows vs. the number of underflow states encountered. A larger initial window is less likely to underflow. A value of $x = 1$ is not used because this can lead to a degenerative state where the query circle repeatedly underflows.

The purpose of the next experiment is to determine how the size of the disk cache affects performance for each algorithm. Figure 35 shows the results of varying the cache size for the event queue of the iCW algorithm. Figure 36 shows the results of varying the cache size for the TPR-tree of the ETP algorithm. This experiment was used in choosing the default disk cache sizes for other experiments.

The purpose of the final experiment on the ETP and the iCW algorithms is to study the relative performance of the two algorithms as the average update

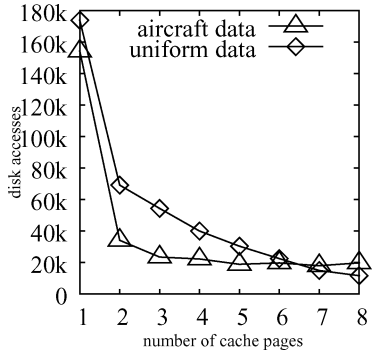


Fig. 35. Number of disk accesses vs. number of disk cache pages for the iCW algorithm.

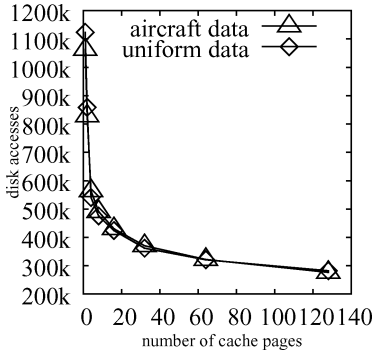


Fig. 36. Number of disk accesses vs. number of disk cache pages for ETP algorithm.

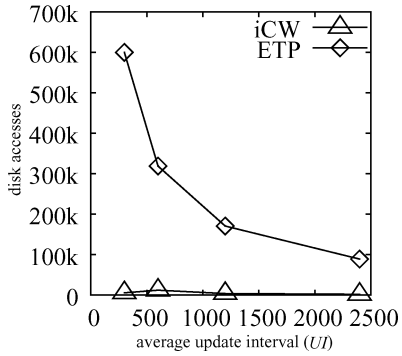


Fig. 37. Number of disk accesses vs. average update interval for uniform data.

interval (UI) changes. Figure 37 gives the results for uniform data. This experiment was only run on synthetic data since precise control of the average update interval is much easier for synthetic data than for real data. It appears from the graph that changes in UI result in nearly constant performance for the iCW algorithm. In contrast, the number of disk accesses for the ETP algorithm appears to grow quadratically as UI decreases. This represents a significant

improvement for the new iCW algorithm given here over the original iCW algorithm presented in Iwerks et al. [2003]. The CW algorithm given in Iwerks et al. [2003] showed quadratic growth as UI decreased. For the original CW algorithm given in Iwerks et al. [2003], the query window size was fixed at a relatively large size so that overflow would not occur. This resulted in many more entries in the event priority queue than necessary. A larger queue size increases the likelihood that a queue element needs to be modified when an update occurs. The underflow handling of the iCW algorithm presented in this paper allows for a much smaller initial window size. The smaller window results in fewer w-events and a smaller priority queue. This also means that updates to points involved in events on the queue are less likely. Many updates don't affect any events, and thus do not incur any disk accesses for the iCW algorithm. On the other hand, every update requires an update to the TPR-tree used by the ETP algorithm.

4.4 AE and NE Experiments

The size of the TPR-tree indexes are the same for both the AE and NE approaches, but the size of the AE event queue can be larger than the NE event queue. On the other hand, the AE approach is simpler in that it does not have the added overhead that the NE algorithm does when w-events come due in using the indexes to find the next event. The trade-off is between a large queue size for the AE algorithm vs. the overhead of additional index access in the NE algorithm.

The worst case for the AE algorithm arises when the join result is initially empty, and every pair of moving points in the Cartesian product of the join relations, $l \times r$, are in the result set and then leave the result set at some time in the future before the next event generation cycle. This leads to two events for each pair of points, or $\mathcal{O}(|l| * |r|)$ events in the queue. On the other hand, the maximum size for the NE queue is the size of one relation since only the next event is computed for each point in one join relation. There can be at most $\mathcal{O}(|l|)$ events in the queue in the worst case in the NE event queue.

The cost of accessing the event queue for the NE algorithm should be cheaper than for the AE algorithm with its smaller queue, especially at the beginning of an event generation cycle where the queues will be at their largest. The TPR-trees will be accessed more often in the NE algorithm since the NE algorithm needs them to compute the next event during event processing. The AE event algorithm does not use the TPR-tree indexes during event processing. This would seem to indicate that for large event generation cycles then AE queue can be large, so we might expect the cost of a large queue to dominate. For small event generation cycles the AE queue is small, so we might expect the cost of the index overhead to dominate. Given this trade-off between the size of the event queue, and the frequency of use of the TPR-tree indexes, plus the overhead of maintaining the indexes and queues during updates, it is not apparent which algorithm may be better through analysis alone.

4.4.1 *Results for AE and NE.* The experiments measure the total number of disk accesses over the duration of a query. Since we are concerned with the

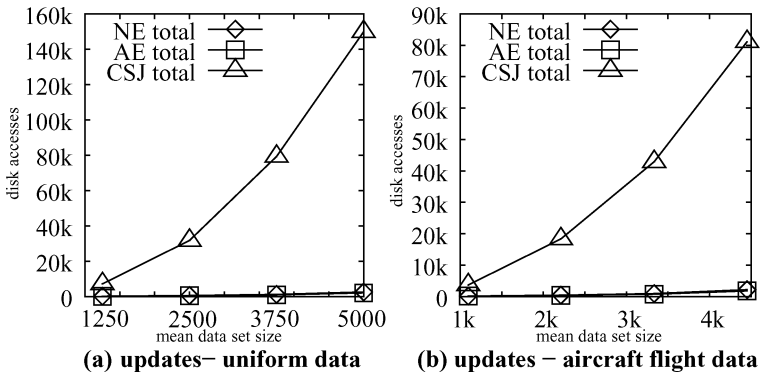


Fig. 38. Total number of disk accesses for our simple adaptation of Tao and Papadias's CSJ algorithm to support updates as compared to the NE and AE algorithms.

maintenance portion of the query, the disk accesses used to compute the initial join result are not included. Pairs of data sets were chosen randomly without replacement from all possible combinations for a total of 110 joins per query. The number of disk accesses was averaged to yield the experiment results for a given query.

Independent variables are mean data set size (μ), join distance (d), event generation cycle length (EG), query duration (t_Q), and disk cache size ($|cache|$). A disk page size of 1024 bytes was used in all experiments. For aircraft flight data, the defaults are $\mu = 9021$ flights, $d = 0.08$ degrees, $t_Q = 100$ seconds, and $|cache| = 32$ pages. For synthetic uniform data, the defaults are $\mu = 10000$ points, $d = 8$ distance units, $t_Q = 100$ time units, and $|cache| = 32$ pages. Values for EG, and values other than the defaults are stated for each individual experiment below.

Each TPR-tree index and each B+-tree has a cache of size $|cache|$. Every cache uses a least recently used (LRU) replacement policy. For a page size of 1024 bytes, leaf nodes of the TPR-tree (B+-tree) hold 50 (22) entries, and internal nodes hold up to 28 (66) entries.

The purpose of the first experiment is to establish a baseline for a performance comparison between a naive adaptation of the continuous spatial join (CSJ) algorithm presented in Tao and Papadias [2003], the NE algorithm, and the AE algorithm. The CSJ algorithm is a future query and does not support updates (see Section 5.5 for more details). A simple naive adaptation of the CSJ algorithm to support updates is to re-invoke the TP portion (the part that finds the events) of the algorithm at the time of the update to find the events for the remainder of the event generation cycle. Nondefault parameters for this experiment are EG = 11, $t_Q = 10$, and $|cache| = 12$. Figure 38 examines the total number of disk accesses (number of spatial index I/O's + number of event queue I/O's) vs. the mean number of objects. The figures show an advantage of the AE and NE algorithms over our simple adaptation of CSJ by nearly two orders of magnitude.

The purpose of the next experiment is to determine for which EG values does the overhead of computing the next event in the NE algorithm dominate

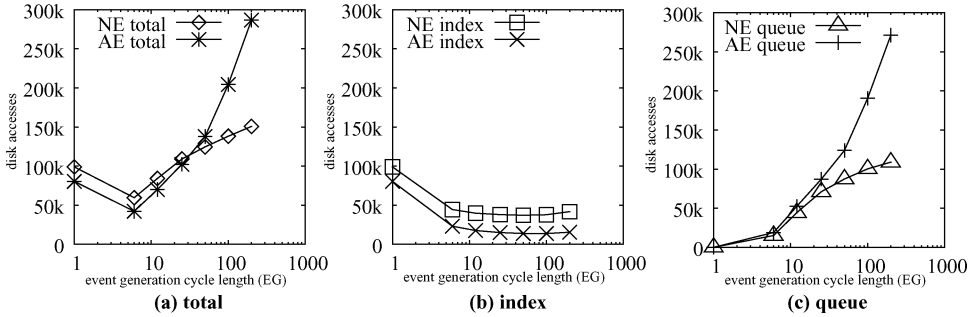


Fig. 39. Aircraft flight data (x-axis is log scale) (a) The total number of disk accesses for the aircraft flight data further broken down into (b) those used in conjunction with the TPR-tree index and (c) those used in the priority event queue.

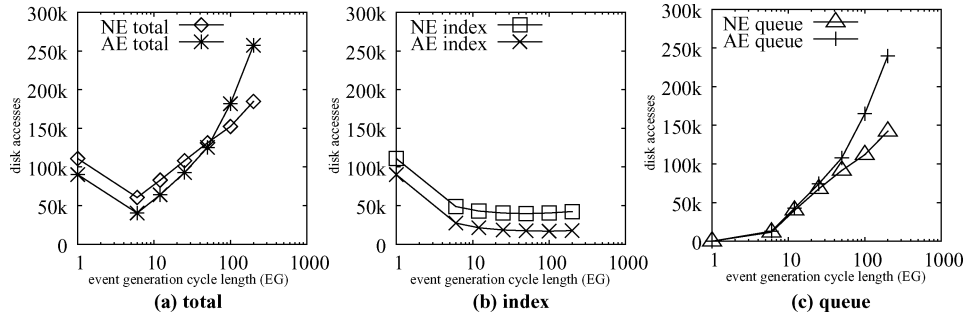


Fig. 40. Uniform synthetic data (x-axis is log scale) (a) The total number of disk accesses for the synthetic data further broken down into (b) those used in conjunction with the TPR-tree index and (c) those used in the priority event queue.

the overhead of having a larger event queue in the AE algorithm. This experiment was run using EG values of 1, 6, 12, 25, 50, 100, and 200. All other parameters were set to the default. The results are given in Figures 39 and 40. Graph (a) reveals that the optimal performance for both methods is around EG = 6. Graph (b) in each figure reveals that the extra TPR-tree overhead for finding the next event in the NE algorithm is nearly constant with respect to EG. Graph (c) shows that the cost of a larger queue for the AE approach grows rapidly as EG increases. The event queue cost for the NE approach grows as well, but at a slower rate than the AE algorithm. As we expect, the TPR-tree performance dominates for small EG value and the event queue dominates for larger EG values.

The next experiment shows how the join query distance affects the performance of the AE and NE approaches. Figure 41 shows the results for different join query distance for two different EG values as shown in Figure 41. For Aircraft data, distance is measured in degrees. For simplicity, distortions introduced by measuring distance in this way at different latitudes are ignored. These distortions are relatively small for the region covered by our experimental data. In a fielded system, degrees would need to be converted to another metric (i.e., kilometers, miles) for accuracy, but for our experiments the data

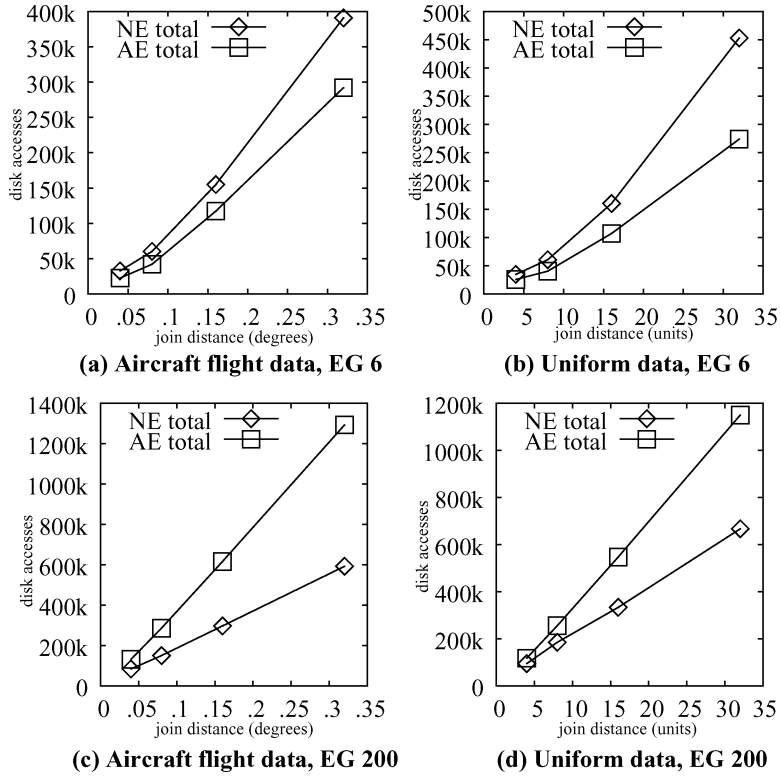


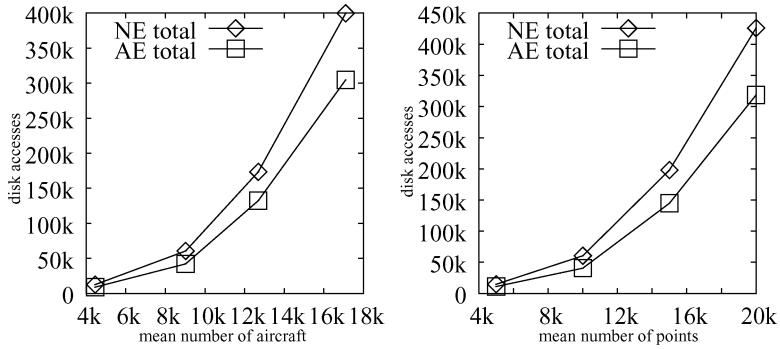
Fig. 41. Number of disk accesses vs. join distance for different data sets and different values of the event generation cycle (EG).

exhibits the desired characteristics of motion and distribution in either case. All other parameters were set to their default values. The number of disk accesses seems to vary quadratically with distance for small values of EG, but varies linearly with distance for larger values of EG. The complex interaction between the TPR-tree indexes, and the event queue makes it unclear why linear behavior is observed for larger EG values.

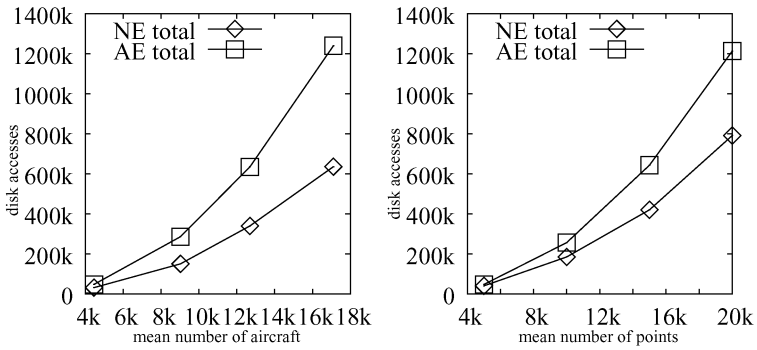
The final experiment shows how the size of the join relations affects the performance of the AE and NE approaches. Results are given in Figure 42. The number of disk accesses seems to follow a quadratic growth rate as a function of data set size. This is consistent with our expectations since the selectivity of a static join exhibits a quadratic growth rate with respect to join relation size.

5. PREVIOUS RELATED WORK

In this section, we present a detailed survey of related work, and how it is related to this article. In particular, some of this related work is use by our algorithms to answer subqueries, for example, the *incremental within event query* in Section 5.3. Other related work consists of solutions to problems similar to maintaining queries over time on moving objects.



(a) Aircraft flight data, EG 6 **(b) Uniform data, EG 6**



(c) Aircraft flight data, EG 200 **(d) Uniform data, EG 200**

Fig. 42. Number of disk accesses vs. the mean number of moving points (per relation) for different data sets and different values of the event generation cycle (EG).

5.1 Spatial Queries on Static Data

Some of the most widely researched queries on static spatial data include within, window, spatial join, k-nearest neighbor (k-nn), and spatial semijoin. A *within* [Samet 1990] query returns all objects within a given distance $d \geq 0$ from a query object. A *window* query is a special case of a within query where the query object is a hyper-rectangle and the distance is zero (i.e., returns all objects inside the hyper-rectangle). A $k$nn query [Arya et al. 1998; Roussopoulos et al. 1995] returns the closest $k > 0$ spatial objects to a given query object. A *spatial join* [Arge et al. 2000; Brinkhoff et al. 1993; Lo and Ravishankar 1996; Patel and DeWitt 1996] returns all pairs of objects in the Cartesian product of two relations that are within a given distance $d \geq 0$ of each other. A *spatial semijoin* [Hjaltason and Samet 1998] is a subset of a spatial join $A \bowtie B$ where a tuple in the result $\langle a, * \rangle$ appears only once for any given $a \in A$, denoted $A \ltimes B$. An additional constraint is imposed in the spatial context of semijoins which stipulates for any tuple $\langle a, b \rangle$ in the result that $b \in B$ is the closest neighbor to a out of all objects in B. Each of these spatial queries also has an incremental [Hjaltason and Samet 1998, 1999] version in which the query result is incrementally computed and reported one tuple at a time until some termination condition is satisfied.

5.1.1 *Incremental Nearest Neighbor and Incremental Distance Query.* The incremental nearest neighbor algorithm [Hjaltason and Samet 1999] returns objects sorted by distance from a query object q one at a time. This algorithm can be used for both within queries, and k-nn queries. Retrieving all the objects within distance $d < \infty$ from q is an *incremental within query* [Hjaltason and Samet 1999]. Retrieving the first k objects and then stopping, with $d = \infty$, is a k-nn query. The incremental nearest neighbor algorithm assumes a spatial tree index where, as in the case of the R-tree [Guttman 1984] spatial index, the internal nodes have bounding boxes (BB) that spatially contain all objects in the subtree. It makes use of a priority queue of objects and internal nodes sorted by distance from q. The queue is initialized with the root BB of the index. Spatial data objects and internal nodes are successively removed from the queue. Data objects are reported as they are dequeued. Internal nodes are expanded when they are dequeued by inserting each element in the node into the queue. This process continues until a maximum number of elements are reported, a maximum distance is reached, it is terminated by the user in an online session, or there are no more elements in the queue. The incremental nearest neighbor algorithm was used to implement some subqueries for our experiments. It was also expanded upon in Tao and Papadias [2003] to find next within events (see Section 5.3).

5.1.2 *Incremental Spatial Join Query.* An incremental spatial join [Hjaltason and Samet 1998] returns all pairs of objects within a given distance d of each other by increasing order of distance. Like the incremental nearest neighbor algorithm (see above), this algorithm uses a priority queue where each element on the queue is a pair of objects. Ordered by distance, closer pairs of objects come before further pairs on the queue. The algorithm assumes uses two spatial indexes, one for each set of objects to be joined. The queue is initialized with the pair of root bounding boxes from each index. Elements on the queue are processed in succession. If an element is a pair of spatial data objects, then they are reported to the user. If at least one of the element pair is an internal index node, then it is expanded and new pairs consisting of the children are enqueued. This process continues until the maximum distance is reached, or by some other means as in the incremental nearest neighbor algorithm. The incremental spatial join query was expanded upon in Tao and Papadias [2003] to develop a continuous spatial join query (see Section 5.5), which in turn is related to our spatial join algorithms.

5.2 Indexing

Spatial indexes are used to support spatial queries. They help aggregate objects and prune the search space by organizing objects either in an object hierarchy, such as the R-tree [Guttman 1984], or a spatial decomposition, such as the quadtree [Samet 1990]. More recently, the indexing of moving objects has also been addressed [Saltenis et al. 2000; Tayeb et al. 1998].

One index that has received much attention is the Time Parameterized R-tree (*TPR-tree*) [Saltenis et al. 2000]. The TPR-tree indexes moving objects described as a function of time. It is a disk-based object hierarchy R-tree variant.

In the R-tree, each node is stored in one disk page. Each node has an associated minimum bounding box (MBB). Leaf nodes contain the MBBs for the indexed objects themselves. Each internal node has an MBB for each subtree spatially bounding the objects in the subtree. In the TPR-tree, a bounding box (BB) is a moving hyper-rectangle specified by two moving points corresponding to opposite corners of the BB. The corner points are chosen so that the BB will always spatially contain the moving objects within it. The BBs in the TPR-tree rarely stay minimal, tending to grow faster than what would be the minimum bounding box at any given time. This is partly compensated for by the TPR-tree update algorithms. As an update occurs, the BB is adjusted to be minimal at the time of the update. Another compensatory action is that the TPR-tree insertion algorithm tries to insert objects moving in a similar manner (e.g., speed, direction), or to a similar destination, into the same leaf node. Saltenis et al. [2000] use the TPR-tree to support ad-hoc spatio-temporal window queries. The query algorithms presenting in Saltenis et al. [2000] did not maintain the query results over time as ours do. The TPR-tree was used in our implementations for indexing moving points.

5.3 Incremental within Event Query

An *incremental within event query* [Tao and Papadias 2003] is similar to an incremental distance query, except that an event time metric is used instead of a distance metric. An incremental within event query returns all the objects and the time at which they will enter the region within a given distance d around a query object q, one at a time, in increasing order of event time. If the distance $d = 0$, then the event time will be the time the objects will intersect, or cease to intersect one another. Tao and Papadias [2003] present such an algorithm. This algorithm assumes an object hierarchy tree index on the moving objects (e.g., the TPR-tree) for which internal nodes have bounding boxes (BB) that continually contain all the moving objects in each subtree. The algorithm is identical to the incremental distance query [Hjaltason and Samet 1999] (see above), except that the priority queue is sorted by within event time instead of by the distance from q. The within event time for an internal node BB will always be less than or equal to the within event times of the objects it contains. This algorithm can be used as is to maintain query results of within distance queries, but only if there are no database updates to the moving objects, that is, they never change course or speed. This algorithm was used in our implementation for some subqueries in our experiments.

5.4 Next NN Event Query

A *next nn-event query* finds the next nearest neighbor event given a query object and its current nearest neighbor. Tao and Papadias [2003] describe two methods, depth-first and best-first, for finding the next nn-event given a query object, the current kth-nearest neighbor, and a set of data points indexed in a TPR-tree. We describe just the best-first method here. To find the next nn-event, the bounding box (BB) of each child at the root is examined and the oc-event for the BB is computed. These nodes are then placed on a global priority queue

sorted by their oc-event time. The first node on the queue is then dequeued and expanded by computing the oc-event for the BB of each child node and inserting the children into the priority queue. This process is repeated recursively until a leaf node is encountered. When a leaf node is examined, the object in the leaf with the soonest oc-event time is saved along with its event as the candidate nn-event. Once the first candidate nn-event is found, the oc-event of the first node on the priority queue and the candidate nn-event are compared. If the first node on the queue has an oc-event sooner than the candidate nn-event, then it is expanded, and the process continues until a leaf node is again encountered, or there are no more nodes on the priority queue with an oc-event time sooner than the candidate nn-event. If another leaf node is encountered, then the oc-event time for each object in the leaf node is computed. If any oc-event time is sooner than the candidate nn-event, then the candidate nn-event is replaced by the object with the soonest oc-event time in the leaf. If, at any time, the next BB on the queue has an oc-event time later than that of the candidate nn-event, then processing stops, and the candidate nn-event is returned as the answer. As long as the oc-event time of the first node on the priority queue is sooner than that of the event time of the candidate nn-event, processing continues.

The next nn-event query supports the time parameterized k-nn algorithm (TP KNN) presented in Tao and Papadias [2003]. This computes a k-nn query on kinematic objects, and then finds the next nn-event that will change the result. The algorithm presented in Tao and Papadias [2003] can then efficiently find many subsequent nn-events. This algorithm can be used to maintain the query result over time, but does not support updates as our iCW and ETP algorithms do. The TP KNN algorithm was used in our implementations to for some subqueries in our experiments.

5.5 Spatial Join Future Query

Tao and Papadias [2003] describe a *continuous spatial join* (CSJ) algorithm which is a continuous future spatial join query. Each event involves two data objects and the time when they either start intersecting, or stop intersecting each other. Events are retrieved in increasing order of event time until some termination condition is satisfied (e.g., maximum time, maximum number of events, etc.). The event processing component is an incremental distance join algorithm [Hjaltason and Samet 1998] (see Section 5.1.1) where the distance metric is replaced by event time as the metric. Like the TP KNN algorithm (Section 5.4), the CSJ algorithm can be used to maintain a query results over time, but it does not support updates to the query result as our AE and NE algorithms do.

5.6 Spatio-temporal and Moving Objects Databases

Mokbel et al. [2004] and Xiong et al. [2004, 2005] present a scalable method for processing range queries and k-nn queries called the *shared execution* paradigm. This approach works by converting many previously separate queries into one large spatial join query, joining together a set of data objects with a set of query objects. Moving objects are represented as points. These

points are updated as the objects move. Query objects are also represented as points, rectangles, or circles. Motion of queries is achieved by updating, or sampling, the locations of the query objects. In general, the shared execution approach updates the query result at periodic intervals. For example, the experiments in Mokbel et al. [2004] used a default query refresh rate of every 10 seconds, whereas the average update rate of a single object was once every 100 seconds. This gives an average of 10% of the object's locations updated between every query refresh. This is acceptable for relatively slow moving objects, or when a relatively large error between query results and the real situation is acceptable (e.g., people walking through an amusement park). This approach becomes problematic for fast moving objects where a small amount of error is required. For example, many ground tracking radar for commercial aircraft update every 6 seconds. This equates to 100% of the moving objects updating their locations every 6 seconds in the database. With a significant number of aircraft, this would most likely overwhelm any disk storage sampling based approach, and would not work for the data we used in our experiments. One advantage the shared execution approach has is the ability to handle erratically moving objects where their motion can not be described easily using functions of time.

A scalable adaptive approach to indexing moving objects represented by linear functions of time is presented in Gedik et al. [2004]. Although the motion of objects is represented as a linear function of time, the algorithms for spatial queries on the index only refresh periodically. In their experiments, this refresh interval was every 30 seconds. Because of this relatively long refresh time, this approach may also not adapt well to fast moving objects like the aircraft data used in our experiments.

A main memory solution to moving object queries is presented in Kalashnikov et al. [2004]. The algorithms presented in Kalashnikov et al. [2004] are scalable, and support moving data objects, but query objects are static. This algorithm also refreshes the query result periodically, but does so in just a few seconds in the experiments that were presented. This algorithm may be applicable for fast moving aircraft against stationary ground query objects, but would not be applicable if both data and query objects are moving. Our algorithms presented in this paper support both fast moving data objects and query objects.

In this article, we did not address queries pertaining to the complete past motion where the database has the complete information about the object's past trajectory. Processing of spatio-temporal queries in these settings has already been addressed in the literature. In particular, Güting et al. [2000] presents an in-depth analysis of the types and operators used, and Lema et al. [2003] gives the specification of the algorithms that implement the operators. Likewise, we did not address queries in systems that have full future knowledge of an objects' trajectory. This class problems is addressed in Trajcevski et al. [2004b]. Although the focus of Trajcevski et al. [2004b] is on the impact of the uncertainty on processing range queries, an important characteristics of this approach (full future trajectory knowledge) is that a particular future-abnormality, for example, an accident in a given road segment, may affect the parameters used in constructing the trajectories. This, in turn, has an impact to the answers to

the spatio-temporal queries that pertain to various geographic locations in the after-abnormality future (cf. Trajcevski et al. [2004a]).

6. CONCLUSION

In this article, we present several approaches to the problem of maintaining k-nn and spatial join query results on continuously moving points where the locations of points are represented as functions of time stored in a database. These functions can be updated in the database by changing the apparent velocity and position of these points as time advances. Our algorithms support these updates when no future knowledge of changes to the data is known, while maintaining constancy of the query results and supporting data structures when updates occur. Both data points and query points may be in motion.

An improved version of the Continuous Windowing k-NN algorithm (iCW) was presented in Section 3. Like the original CW algorithm introduced in Iwerks et al. [2003], the iCW algorithm filters the points considered for the nn-event using w-events to maintain the set of points close to the query point. We extended the original CW algorithm to support updates to the query point, and to dynamically adjust the size of the circular window when underflow occurs. The cost of adjusting the window size is expensive since the base relation must be scanned twice: once to find the $k + x + 1$ nearest neighbors to the query points, and once to find all the w-events. We also extended the ETP algorithm presented in Iwerks et al. [2003] to support updates to the query point for comparison. The ETP algorithm originally presented in Iwerks et al. [2003] is the Tao and Papadias [2003] continuous TP KNN algorithm extended by us to support updates. The ETP algorithm uses a TPR-tree index on the data points to find each subsequent nn-event.

The iCW method outperformed the ETP algorithm by more than an order of magnitude in experiments. Additionally, the dynamic window resizing results in an algorithm that exhibits nearly constant performance as the average update interval (UI) of the data set changes (see Figure 37). The original CW algorithm presented in Iwerks et al. [2003] showed a growth rate similar to that of the ETP algorithm as UI increased.

Event-based query algorithms to maintain spatial join queries on moving points were also presented in Section 3. These algorithms also support updates to the base relations. Two new approaches were compared that differ in the number of events placed on the queue. The All-Event (AE) approach can be thought of as an extension of the continuous spatial join (CSJ) algorithm presented in Tao and Papadias [2003] to support updates in the same way that the ETP algorithm is an extension of the continuous TP KNN algorithm [Tao and Papadias 2003] to support updates in k-nearest neighbor queries. The AE approach stores all within events to occur in the near future in a priority event queue. The more novel Next-Event (NE) approach only stores the next event to occur for each query point in the event queue. The time period considered for future events is limited to the current event generation cycle.

Both the AE and NE approaches outperform a simpler adaptation of the CSJ algorithm to support updates by up to two orders of magnitude. When the event

generation cycle is short, the AE approach results in fewer disk accesses than the NE algorithm. When the event generation cycle is long, the NE approach results in fewer disk accesses because the size of the event queue is kept small.

Updates to the TPR-tree and B+-tree based indexes are costly. Future work should focus on developing more update-efficient data structures to support these algorithms. Some work has been done towards this already [Jensen and Saltenis 2002], but more work is needed. For example, updates in our aircraft data usually consisted of changes in velocity. This was handled as a deletion followed by an insertion, but one can imagine more efficient methods for handling this case. With these algorithms to support the maintenance of spatial queries on point kinematic data types, more update-efficient disk-based data structures should help improve their performance and improve their scalability. Other spatial indexing structures such as the STAR-tree [Procopiuc et al. 2002], which maintains tighter bounding boxes on the index nodes, may also result in improved performance and are also worthy of future study.

APPENDIX A. DETAILED ALGORITHM IMPLEMENTATIONS

In the algorithms, r is a relation with a moving point attribute in its schema. Variable q is the query point. Variable k is the number of neighbors that are sought. Variable $x \geq 2$ is an integer. Variable e_{nn} is the next nn-event. Variable *tpr* is a TPR-tree index. Variable p is a moving point. Variable d is the window distance. Variable W is the window within query result set. Variable K is the k-nn query result. Variable Q is the w-event priority queue.

Function Dequeue(Q) removes the next event from Q and returns it. Function Kth(K) returns the kth neighbor in set K at the current time. Function next_w_event(p, q, d, t) returns the next event after time t when point p will be at distance d from q, or it returns a null event with time stamp ∞ if no such event exists (Section 2.5). Function next_oc_event(w_i, q,Kth(K),now) returns the next oc-event for point p after time t with respect to query point q, and kth neighbor *nn* (Section 2.5). Function Time(e) returns the time of event e. The Euclidean distance between point instances p and q at time t is $\|p, q, t\| = |p(t), q(t)| = \sqrt{(q(t) - p(t))^2}$.

The *next nn-event query* is the algorithm described in Section 5.4. The *incremental distance query* is the algorithm described in Section 5.1.1.

A.1 ETP Algorithms

procedure ETP(r, q, k)
1. Build TPR-tree index, *tpr* on the moving points in r.
2. Let $K \leftarrow$ first k points returned by the *incremental distance query* on index *tpr* for query point q.
3. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index *tpr* for query point q and kth neighbor Kth(K).
4. **while** true **do**
5. Sleep until there is an update, or event e_{nn} comes due.
6. **if** there is an update to r **then**
7. ETP_Update_Data_Relation(r, q, k, e_{nn}, tpr, K)
8. **else if** there is an update to q **then**
9. ETP_Update_Query_Point(q, k, e_{nn}, tpr, K)

```
10.     else ETP_Process_Nn_Evt(q, k, e_nn, tpr, K)
11.  end while
```

procedure ETP_Update_Data_Relation(r, q, k, e_{nn}, tpr, K)

```
 1.  Let p be the point just inserted or deleted in r.
 2.  if p was inserted into r then
 3.     Insert p into index tpr.
 4.     if there is no current query point then return.
 5.     if ‖q, p,now‖ < ‖q,Kth(K),now‖ then
 6.        K ← (K\Kth(K)) ∪ {p}.
 7.        Let e_nn ← result of next nn-event query on
           index tpr for query point q and new Kth(K).
 8.     else /* Kth(K) closer to q than p */
 9.        Let e_oc ←next_oc_event(p, q,Kth(K),now).
10.        if Time(e_oc) <Time(e_nn) then e_nn ← e_oc.
11.     end if-then
12.  else /* p was deleted from r */
13.     Remove p from index tpr.
14.     if there is no current query point then return.
15.     if ‖q, p,now‖ ≤ ‖q,Kth(K),now‖ then
16.        Let p_k ← kth point returned by incremental
           distance query on index tpr for query point q.
17.        K ← (K \ {p}) ∪ {p_k}.
18.        Let e_nn ← result of next nn-event query on
           index tpr for query point q and new Kth(K).
19.     else if p is involved in event e_nn then
20.        Let e_nn ← result of next nn-event query on
           index tpr for query point q and new Kth(K).
21.     end if-then
22.  end if-then
```

procedure ETP_Update_Query_Point(q, k, e_{nn}, tpr, K)

```
1.  if inserting q then
2.     Let K ← first k points returned by incremental
       distance query on index tpr for query point q.
3.     Let e_nn ← result of next nn-event query on
       index tpr for query point q and Kth(K).
4.  else /* deleting q */
5.     K ← ∅.
6.     e_nn ← ∅.
7.  end if-then
```

procedure ETP_Process_Nn_Evt(q, k, e_{nn}, tpr, K)

```
1.  Let p ← non-kth neighbor data point involved in e_nn.
2.  if p will be closer to q than Kth(K) after event then
3.     K ← (K \ Kth(K)) ∪ {p}.
4.  end if-then
5.  Let e_nn ← result of next nn-event query on index tpr
    for query point q and kth neighbor Kth(K).
```

A.2 iCW Algorithms

procedure iCW(r, q, k, x)

```
1.  iCW_Adjust_Window(r, q, k, x, d, W, Q)
2.  iCW_Compute_Knn_Result(q, k, e_nn, W, K)
3.  while true do
4.     Sleep until there is an update, or an event comes due.
```

```
5.      if there is an update to r then
6.          iCW_Update_Data_Relation(r, q, k, x, d, e_nn, W, K, Q)
7.      else if there is an update to q then
8.          iCW_Update_Query_Point(r, q, k, x, d, e_nn, W, K, Q)
9.      else if a w-event has come due then
10.         iCW_Process_Within_Evt(r, q, k, x, d, e_nn, W, K, Q)
11.     else iCW_Process_Nn_Evt(q, e_nn, W, K)
12.  end while

  procedure iCW_Adjust_Window(r, q, k, x, d, W, Q)
1.   Let W ← ∅; Q ← ∅
2.   Scan r and find the closest k + x + 1 points to
     q at the current time, and assign them to set S.
3.   Sort points in S by distance to q at the current time.
4.   Let d ← (‖q, s_{k+x},now‖ + ‖q, s_{k+x+1},now‖)/2.
5.   Let W ← first k + x points in S.
6.   for each point p_i ∈ r
7.      e ←next_w_event(p_i, q, d,now)
8.      if e ≠ ∅ then Q ← Q ∪ {e}
9.   end for-each

  procedure iCW_Compute_Knn_Result(q, k, e_nn, W, K)
1.   Let K be the closest k points to q in set W
     at the current time sorted by distance to q.
3.   e_nn ← ∅   /* note: Time(∅) = ∞ */
4.   for each point w_i ∈ W ∧ w_i ≠Kth(K) do
5.      e_oc ←next_oc_event(w_i, q,Kth(K),now)
6.      if Time(e_oc) <Time(e_nn) then e_nn ← e_oc
7.   end for-each

  procedure iCW_Update_Data_Relation(r, q, k, x, d, e_nn, W, K, Q)
1.   if there is no current query point then return.
2.   Let point p be the point just inserted or deleted from r.
3.   if p was inserted into r then
4.      if ‖q, p,now‖ ≤ d then
5.         Let Q ← Q ∪ {next_w_event(p, q, d,now)}.
6.         Let W ← W ∪ {p}.
7.         if ‖q, p,now‖ < ‖q,Kth(K),now‖ then
8.            Let K ← (K\Kth(K)) ∪ {p}.
9.            Let e_nn ← soonest oc-event from points in W after current time.
10.        else
11.           Let e_oc ←next_oc_event(p, q,Kth(K),now)
12.           if Time(e_oc) <Time(e_nn) then e_nn ← e_oc
13.        end if-then
14.     else if next_w_event(p, q, d,now) < ∞ then
15.        Let Q ← Q ∪ {next_w_event(p, q, d,now)}.
16.     end if-then
17.  else if p has a w-event in Q then
18.     Remove the w-event involving p from Q.
19.     if p in W then
20.        Let W ← W \ {p}.
21.        if |W| ≤ k then /* underflow */
22.           iCW_Adjust_Window(r, q, k, x, d, W, Q)
23.           iCW_Compute_Knn_Result(q, k, e_nn, W, K)
24.        else if p in K then
25.           iCW_Compute_Knn_Result(q, k, e_nn, W, K)
26.        else if p involved in e_nn then
```

```
27.            Let e_nn ← soonest oc-event from points in W after current time.
28.          end if-then
29.        end if-then
30.    end if-then
```

```
    procedure iCW_Process_Within_Evt(r, q, k, x, d, e_nn, W, K, Q)
 1.   e_w ←Dequeue(Q).
 2.   Let p be the data point involved in e_w.
 3.   if e_w is an enter event then
 4.      Let W ← W ∪ {p}.
 5.      Let Q ← Q ∪ {next_w_event(p, q, d,now)}.
 6.      Let e_oc ←next_oc_event(p, q,Kth(K),now).
 7.      if Time(e_oc) <Time(e_nn) then e_nn ← e_oc.
 8.   else /* e_w is an exit event */
 9.      W ← W \ {p}.
10.      if |W| ≤ k then /* underflow */
11.         iCW_Adjust_Window(r, q, k, x, d, W, Q)
12.      end if-then
13.   end if-then
```

```
    procedure iCW_Update_Query_Point(r, q, k, x, d, e_nn, W, K, Q)
 1.   if query point q was inserted then
 2.      iCW_Adjust_Window(r, q, k, x, d, W, Q)
 3.      iCW_Compute_Knn_Result(q, k, e_nn, W, K)
 4.   else let K ← ∅, Q ← ∅, e_nn ← ∅.
```

```
    procedure iCW_Process_Nn_Evt(q, e_nn, W, K)
 1.   Let p ← non-kth neighbor data point involved in e_nn.
 2.   if p will be closer to q than Kth(K) after event then
 3.      K ← (K \ Kth(K)) ∪ {p}.
 4.   end if-then
 5.   Let e_nn ← soonest oc-event from points in W
      after current time.
```

A.3 AE Algorithms

Input parameter j to the AE_Detailed() and the NE_Detailed() algorithms defines the continuous spatial join query. Associated with j are the two relations $j.l$ and $j.r$ to be joined. Relation $j.l$ has a TPR-tree index $j.tpr_l$ on moving point attribute $\alpha_l \in L$ where L is the schema of $j.l$. Relation $j.r$ has a TPR-tree index $j.tpr_r$ on moving point attribute $\alpha_r \in R$ where R is the schema of $j.r$. Events are stored in a queue $j.Q$ sorted by time in increasing order. The scalar value $j.dist$ is the join distance, and scalar $j.generation\_length$ is the event generation cycle duration.

Function pair(e) joins the tuples that contain the moving points involved in the event e and returns them. Function Find_Within_Dist() is a within distance query that returning the tuples from $j.r$ with points indexed by $j.tpr_r$ that are within distance $j.dist$ of query point p_l. The function All_Within_Events($j.tpr_r$, p_l, $j.dist$, Δt) returns all the w-events at distance $j.dist$ from query point p_l during the next Δt with the moving points indexed by $j.tpr_r$. Function All_Within_Events() is implemented using the *incremental within event query* described in Section 5.3, stopping when the maximum time Δt is reached. Function Next_Within_Event($j.tpr_r$, p_l, $j.dist$, Δt) returns the next

w-event at distance $j.dist$ from query point p_l during the next Δt time units with the moving points indexed by $j.tpr_r$. It returns a null event if no such next event exists. Function Next_Within_Event() (not given here) is implemented using the *incremental within event query* described in Section 5.3, stopping after the first event is returned.

```
procedure AE_Detailed(j)
1.  Perform initial join and report result J.
2.  AE_Generate_Events(j)
3.  while true do
4.      Sleep until there is an update, or an event comes due,
        or the end of the event generation cycle is reached.
5.      if an event came due then
6.          AE_Process_Next_Event(j)
7.      else if there is an update then
8.          if tuple was inserted in j.l then
9.              AE_Insert_L(j)
10.         else if tuple was deleted from j.l then
11.             AE_Delete_L(j)
12.         else if tuple was inserted in j.r then
13.             AE_Insert_R(j)
14.         else /* tuple was deleted from j.r */
15.             AE_Delete_R(j)
16.         end if-then
17.     else /* end of event generation cycle was reached */
18.         AE_Generate_Events(j)
19.     end if-then
20. end while
```

```
procedure AE_Process_Next_Event(j)
1. Event e ←Dequeue(j.Q)
2. if event e is an enter event then
3.     Report pair(e) inserted into result J.
4. else if event e is an exit event then
5.     Report pair(e) deleted from result J.
6. end if-then
```

```
procedure AE_Insert_L(j)
1.  Let τ_l be the new tuple that was inserted into j.l.
2.  Let p_l be the instance of moving point attribute α_l in τ_l.
3.  Let Δt be the time between now and the end of the current event generation cycle.
4.  for each tuple τ ∈ Find_Within_Dist(j.tpr_r, j.r, p_l, j.dist)
5.      Report joined tuple τ_l τ inserted into result J.
6.  end for-each
7.  for each event e ∈ All_Within_Events(j.tpr_r, p_l, j.dist, Δt)
8.      Enqueue event e on queue j.Q.
9.  end for-each
10. Insert point p_l into index j.tpr_l.
```

```
procedure AE_Delete_L(j)
1.  Let τ_l be the tuple that was deleted from j.l.
2.  Let p_l be the instance of moving point attribute α_l in τ_l.
3.  Remove all events involving p_l from j.Q.
4.  for each tuple τ ∈ Find_Within_Dist(j.tpr_r, j.r, p_l, j.dist)
5.      Report joined tuple τ_l τ deleted from result J.
6.  end for-each
7.  Delete point p_l from index j.tpr_l.
```

AE_Insert_R() is symmetric with AE_Insert_L(j). In particular, replacing $\tau_l \tau$ with $\tau \tau_r$ in line 5 of AE_Insert_L(j), and then swapping symbols l with r yields the AE_Insert_R(j) algorithm. Similarly, AE_Delete_R() is symmetric with procedure AE_Delete_L(j). In particular, AE_Delete_R(j) is derived from AE_Delete_L(j) by replacing $\tau_l \tau$ with $\tau \tau_r$ in line 5, and then swapping symbols l with r.

procedure AE_Generate_Events(j)
1. Let Δt be the time between now and the end of the current event generation cycle.
2. **for each** leaf node $n \in j.tpr_l$
3. **for each** moving point $p_l \in n$
4. **for each** event $e \in$ All_Within_Events($j.tpr_r$, p_l, $j.dist$, Δt)
5. Enqueue event e in queue $j.Q$.
6. **end for-each**
7. **end for-each**
8. **end for-each**

A.4 NE Algorithms

procedure NE_Detailed(j)
1. Perform initial join and report result J.
2. NE_Generate_Events(j)
3. **while** true **do**
4. Sleep until there is an update, or an event comes due, or the end of the event generation cycle is reached.
5. **if** an event came due **then**
6. NE_Process_Next_Event(j)
7. **else if** there is an update **then**
8. **if** tuple was inserted in $j.l$ **then**
9. NE_Insert_L(j)
10. **else if** tuple was deleted from $j.l$ **then**
11. NE_Delete_L(j)
12. **else if** tuple was inserted in $j.r$ **then**
13. NE_Insert_R(j)
14. **else** /* tuple was deleted from $j.r$ */
15. NE_Delete_R(j)
16. **end if-then**
17. **else** /* end of event generation cycle was reached */
18. NE_Generate_Events(j)
19. **end if-then**
20. **end while**

procedure NE_Process_Next_Event(j)
1. Event $e \leftarrow$Dequeue($j.Q$)
2. Let p_l be the moving point in $j.l$ involved in e.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **if** event e is an exit event **then**
5. Report pair(e) deleted from result J.
6. **else if** event e is an enter event **then**
7. Report pair(e) inserted into result J.
8. Let e_{exit} be the exit event following e for the points involved in e.
9. **if** event e_{exit} occurs before now + Δt **then**
10. Let Δt be the time between now and e_{exit}.
11. **end if-then**
12. $e_{next} \leftarrow$ Next_Within_Event($j.tpr_r$, p_l, $j.dist$, Δt)
13. **if** event e_{next} is not null **then** enqueue e_{next} in $j.Q$.

procedure NE_Insert_L(j)
1. Let τ_l be the new tuple that was inserted into $j.l$.
2. Let p_l be the instance of the moving point in τ_l.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **for each** $\tau \in$ Find_Within_Dist($j.tpr_r$, $j.r$, p_l, $j.dist$)
5. Report joined tuple $\tau_l \tau$ inserted into result J.
6. Let e_{exit} be the exit event between the points in τ_l and τ.
7. **if** event e_{exit} occurs before now + Δt **then**
8. Let Δt be the time between now and e_{exit}.
9. **end for-each**
10. $e_{next} \leftarrow$Next_Within_Event($j.tpr_r$, p_l, $j.dist$, Δt)
11. **if** event e_{next} is not null **then** enqueue e_{next} in $j.Q$.
12. Insert point p_l into index $j.tpr_l$.

Procedure NE_Delete_L(j) is identical to AE_Delete_L(j) with the exception that there is at most one item to be removed from Q.

procedure NE_Insert_R(j)
1. Let τ_r be the new tuple that was inserted into $j.r$.
2. Let p_r be the instance of moving point attribute α_r in τ_r.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **for each** $\tau \in$ Find_Within_Dist($j.tpr_l$, $j.l$, p_r, $j.dist$)
5. Report joined tuple $\tau \tau_r$ inserted into result J.
6. **for each** $e \in$ All_Within_Events($j.tpr_l$, p_r, $j.dist$, Δt)
7. Let p_l be the point from $j.l$ that is involved in e.
8. **if** there is no event already in $j.Q$ involving p_l **then**
9. Enqueue event e in $j.Q$.
10. **else**
11. Let e_{prev} be the event involving p_l in $j.Q$.
12. **if** e_{prev} occurs after e **then**
13. Replace e_{prev} with e in $j.Q$.
14. **end if-then**
15. **end if-then**
16. **end for-each**
17. Insert point p_r into index $j.tpr_r$.

procedure NE_Delete_R(j)
1. Let τ_r be the tuple that was deleted from $j.r$.
2. Let p_r be the instance of moving point attribute α_r in τ_r.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. Delete point p_r from index $j.tpr_r$.
5. **for each** event w(p_l, p_r, t) $\in j.Q$
6. Remove w(p_l, p_r, t) from $j.Q$.
7. $e \leftarrow$Next_Within_Event($j.tpr_r$, p_l, $j.dist$, Δt)
8. Enqueue e in $j.Q$.
9. **end for-each**
10. **for each** $\tau \in$ Find_Within_Dist($j.tpr_l$, $j.l$, p_r, $j.dist$)
11. Report joined tuple $\tau \tau_r$ deleted from result J.
12. **end for-each**

procedure NE_Generate_Events(j)
1. Let Δt be the time between now and the end of the current event generation cycle.
2. **for each** leaf node $n \in j.tpr_l$

```
3.      for each moving point p_l ∈ n
4.        for each event e ∈ Next_Within_Event(j.tpr_r, p_l, j.dist, Δt)
5.          Enqueue event e in queue j.Q.
6.        end for-each
7.      end for-each
8.    end for-each
```

REFERENCES

AGARWAL, P. K., ARGE, L., AND ERICKSON, J. 2000. Indexing moving points. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems* (Dallas, TX). ACM, New York, 175–186.

AGARWAL, P. K. AND PROCOPIUC, C. M. 2002. Advances in indexing for mobile objects. *IEEE Bull. Tech. Comm. Data Eng. 25*, 2, 25–34.

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., VAHRENHOLD, J., AND VITTER, J. S. 2000. A unified approach for indexed and non-indexed spatial joins. In *Proceedings of the 7th International Conference on Extending Database Technology—EDBT'2000* (Konstanz, Germany). 413–429.

ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. Y. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM 45*, 6 (Nov.), 891–923.

BASCH, J., GUIBAS, L. J., AND HERSHBERGER, J. 1997. Data structures for mobile data. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, LA). ACM, New York, 747–756.

BENETIS, R., JENSEN, C., KARCIAUSKAS, G., AND SALTENIS, S. 2002. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)* (Edmonton, Ont., Canada). 44–53.

BRINKHOFF, T., KRIEGEL, H. P., AND SEEGER, B. 1993. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference* (Washington, DC). ACM, New York, 237–246.

DOUGLAS, D. AND PEUCKER, T. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartograph. 10*, 2, 112–122.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. *Commun. ACM 33*, 10 (Oct.), 30–53.

GEDIK, B., WU, K., YU, P., AND LIU, L. 2004. Motion adaptive indexing for moving continual queries over moving objects. In *Proceedings of the Conference on Information and Knowledge Management* (Washington, DC). 427–436.

GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. 1993. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference* (Washington, DC). ACM, New York, 157–166.

GÜTING, R. H., BÖHLEN, M. H., ERWIG, M., JENSEN, C. S., LORENTZOS, N. A., SCHNEIDER, M., AND VAZIRGIANNIS, M. 2000. A foundation for representing and querying moving objects. *ACM Trans. Datab. Syst. (TODS) 25*, 1 (Mar.), 1–42.

GÜTING, R. H. AND SCHNEIDER, M. 2005. *Moving Objects Databases*. Morgan-Kaufmann, San Francisco, CA.

GUTTMAN, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference* (Boston, MA). ACM, New York, 47–57.

HELLERSTEIN, J. M., NAUGHTON, J. F., AND PFEFFER, A. 1995. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases* (Zurich, Switzerland). 562–573.

HJALTASON, G. R. AND SAMET, H. 1998. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference* (Seattle, WA). ACM, New York, 237–248.

HJALTASON, G. R. AND SAMET, H. 1999. Distance browsing in spatial databases. *ACM Trans. Datab. Syst. 24*, 2 (June), 265–318. (Also University of Maryland Computer Science TR–3919).

IWERKS, G. S., SAMET, H., AND SMITH, K. 2003. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the 29th International Conference on Very Large Data Bases* (Berlin, Germany). 512–523.

IWERKS, G. S., SAMET, H., AND SMITH, K. 2004. Maintenance of spatial semijoin queries on moving points. In *Proceedings of the 30th International Conference on Very Large Data Bases* (Toronto, Ont., Canada).

JENSEN, C. S. AND SALTENIS, S. 2002. Towards increasingly update efficient moving-object indexing. *IEEE Bull. Tech. Comm. Data Eng. 25*, 2 (June), 35–40.

KALASHNIKOV, D. V., PRABHAKAR, S., AND HAMBRUSCH, S. E. 2004. Main memory evaluation of monitoring queries over moving objects. *Distrib. Para. Datab. 15*, 2 (Mar.), 117–135.

LEMA, J. A. C., FORLIZZI, L., GÜTING, R. H., NARDELLI, E., AND SCHNEIDER, M. 2003. Algorithms for moving objects databases. *Comput. J. 46*, 6, 680–712.

LO, M.-L. AND RAVISHANKAR, C. V. 1996. Spatial hash-joins. In *Proceedings of the ACM SIGMOD Conference* (Montréal, Ont., Canada). ACM, New York, 247–258.

MOKBEL, M. F., GHANEM, T. M., AND AREF, W. G. 2003. Spatio-temporal access methods. *IEEE Bull. Tech. Comm. Data Eng. 26*, 2 (June), 40–49.

MOKBEL, M. F., XIONG, X., AND AREF, W. G. 2004. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD Conference* (Paris, France). ACM, New York.

MOKHTAR, H., SU, J., AND IBARRA, O. 2002. On moving object queries. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems* (Madison, WI). ACM, New York, 188–198.

NAKAMURA, Y. AND YAMANE, K. 2000. Dynamics computation of structure-varying kinematic chains and its application to human figures. *IEEE Trans. Rob. Automat. 16*, 2 (Apr.), 124–134.

NASCIMENTO, M. A., SILVA, R., AND THEODORIDIS, Y. 1999. Evaluation of access structures for discretely moving points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management* (Edinburgh, UK), 171–188.

ÖZSOYOĞLU, G. AND SNODGRASS, R. T. 1995. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng. 7*, 4 (Aug.), 513–532.

PATEL, J. M. AND DEWITT, D. J. 1996. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD Conference* (Montréal, Ont., Canada). ACM, New York, 259–270.

PFOSER, D. 2002. Indexing the trajectories of moving objects. *IEEE Bull. Tech. Comm. Data Eng 25*, 2 (June), 3–9.

PFOSER, D. AND JENSEN, C. S. 1999. Capturing the uncertainty of moving-object representations. In *Proceedings of the Advances in Spatial Databases—Sixth International Symposium, SSD'99* (Hong Kong). Lecture Notes in Computer Science, vol 1651. Springer-Verlag, New York, 111–132.

PROCOPIUC, C. M., AGARWAL, P. K., AND HAR, S.PELED. 2002. Star-tree: An efficient self-adjusting index for moving objects. In *Proceedings of the Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments* (San Francicsco, CA). 178–193.

RAPTOPOULOU, K., PAPADOPOULOS, A. N., AND MANOLOPOULOS, Y. 2003. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica 7*, 2, 113–137.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference* (San Jose, CA). ACM, New York, 71–79.

SALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LOPEZ, M. A. 2000. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD Conference* (Dallas, TX). ACM, New York, 331–342.

SAMET, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA.

SCIS. 1996. *IEEE Std. 1278 1-1995*. IEEE Computer Society, Standards Committee on Interactive Simulation, USA.

SISTLA, A. P., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. 1997. Modeling and querying moving objects. In *Proceedings of the 13th IEEE Conference on Data Engineering (ICDE)* (Birmingham, UK). IEEE Computer Society Press, Los Alamitos, CA, 422–432.

SONG, Z. AND ROUSSOPOULOS, N. 2001. *K*-nearest neighbor search for moving query point. In *Proceedings of the Advances in Spatial and Temporal Databases (SSTD)* (Redondo Beach, CA). Lecture Notes in Computer Science, vol. 2121. Springer-Verlag, New York, 79–96.

TAO, Y. AND PAPADIAS, D. 2003. Spatial queries in dynamic environments. *ACM Trans. Datab. Syst. (TODS) 28*, 2 (June), 101–139.

TAO, Y., PAPADIAS, D., AND SUN, J. 2003a. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th International Conference on Very Large Data Bases* (Berlin, Germany). 790–801.

TAO, Y., SUN, J., AND PAPADIAS, D. 2003b. Selectivity estimation for predictive spatio-temporal queries. In *Proceedings of 19th IEEE International Conference on Data Engineering (ICDE)* (Bangalore, India). IEEE Computer Society Press, Los Alamitos, CA, 417–428.

TAYEB, J., ULUSOY, Ö., AND WOLFSON, O. 1998. A quadtree-based dynamic attribute indexing method. *Comput. J. 41*, 3, 185–200.

TRAJCEVSKI, G., SCHEUERMANN, P., WOLFSON, O., AND NEDUNGADI, N. 2004a. CAT: Correct answers of continuous queries using triggers. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (Heraklion, Greece). 837–840.

TRAJCEVSKI, G., WOLFSON, O., HINRICHS, K., AND CHAMBERLAIN, S. 2004b. Managing uncertainty in moving objects databases. *ACM Trans. Datab. Syst. (TODS) 29*, 3 (Sept.), 463–507.

WOLFSON, O., CHAMBERLAIN, S., DAO, S., JIANG, L., AND MENDEZ, G. 1998. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*. 588–596.

XIONG, X., MOKBEL, M. F., AND AREF, W. G. 2005. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the International Conference of Data Engineering* (Tokyo, Japan).

XIONG, X., MOKBEL, M. F., AREF, W. G., HAMBRUSCH, S. E., AND PRABHAKAR, S. 2004. Scalable spatio-temporal continuous query processing for location-aware services. In *Proceedings of the International Conference on Scientific and Statistical Database* (Santorini Island, Greece).

Received January 2005; revised August 2005; accepted November 2005