Maintenance of Spatial Semijoin Queries on Moving Points

Glenn S. Iwerks [≀‡†] Hanan Samet [‡*] Kenneth P. Smith [≀†]

[‡]Computer Science Department,
Center for Automation Research, and
Institute for Advanced Computer Studies
University of Maryland at College Park
{iwerks,hjs}@umiacs.umd.edu

[≀]The MITRE Corporation
7515 Colshire Dr.
McLean, Virginia 22102
{iwerks,kps}@mitre.org

Abstract

In this paper, we address the maintenance of spatial semijoin queries over continuously moving points, where points are modeled as linear functions of time. This is analogous to the maintenance of a materialized view except, as time advances, the query result may change independently of updates. As in a materialized view, we assume there is no prior knowledge of updates before they occur. We present a new approach, *continuous fuzzy sets* (CFS), to maintain continuous spatial semijoins efficiently. CFS is compared experimentally to a simple scaling of previous work. The result is significantly better performance of CFS compared to previous work by up to an order of magnitude in some cases.

1 Introduction

We consider the following queries. For each moving firetruck, keep track of the nearest mobile police unit. For each airplane, keep track of the nearest airport. For each cell phone, keep track of the nearest airborne relay station. For each tank, keep track of the nearest target. For each robot explorer in a swarm of robots, keep track of the nearest maintenance robot. For each unmanned air vehicle, keep track of the nearest observation objective. For each ship, keep track of the nearest sonar tracking station.

[*]This work was supported in part by the National Science Foundation under grants EIA-99-00268, IIS-00-86162, and EIA-00-91474.

[†]This work was supported in part by the National Institute for Mental Health and the National Science Foundation under NIH grant R01-MH64417-01.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

These are all examples of spatial semijoin queries on moving objects. Many are examples where all the objects are moving simultaneously and continuously. All must update the query result as the objects move in real time. None know how the object will move ahead of time.

In this paper, we address the maintenance of spatial semijoin queries over continuously moving points. Given two sets of moving points Q and D, we define semijoin $Q \ltimes_k D$ as all the pairs $\langle q, p\rangle$, $q \in Q \wedge p \in D$, that are in Cartesian product $Q \times D$, and p is one of the k nearest neighbors of q. Set Q is the set of query points, and D is the set of data points. This amounts to a massive scaling of a continuous nearest neighbor query for all query points. Traditionally, a semijoin returns tuples from only one join relation. However, we relax this constraint to make the result meaningful in light of the examples above.

Data sets Q, and D are updated through insertions and deletions to the sets. There is no prior knowledge of what the updates will be in advance of each update occurrence. This is analogous to the maintenance of a materialized view [5], with the difference being that the query result may change as a result of the motion of points represented in the database as well as updates to the database.

Points are modeled as linear functions of time, as opposed to samples of an objects location that are updated as an object moves. Therefore, as time advances, the query result may change independently of updates. Access methods and operators for this data type have been studied extensively including indexing methods [17, 21], ad-hoc queries [17, 19], and continuous queries such as window [17, 19], and within [10, 20], and finding the k nearest neighbors [2, 10, 15, 20] (k-nn) to an object.

To our knowledge there has been no previous work to perform continuous spatial semijoin queries on moving objects so that any of the examples queries given in the first paragraph above for data sets of significant size can be answered. Some work on scaling k-nn queries on point data represented as samples (e.g., [14]) has been done, but not on the scale needed to perform semijoins.

In this paper, we present a new approach, termed *con-*

tinuous fuzzy sets (CFS), to perform spatial semijoins. This approach is most similar to a continuous window k-nn algorithm presented in [10]. However, CFS is not just a simple scaling of this previous work. As we will show, previous work (e.g., [10, 20]) does not scale well. CFS is compared experimentally to a simple scaling of the time-parameterized k-nn algorithm presented in [20]. The result is a significant better performance of CFS compared to this previous work by up to an order of magnitude in some cases.

The rest of this paper is organized as follows. Section 2 reviews previous work and the background necessary to understand this paper. Our CFS algorithm is presented in Section 3. Section 4 discusses some performance issues. Experimental results are described in Section 5. Section 6 contains some concluding remarks.

2 Background and Previous Work

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* [18] query returns all objects within a given distance $d \geq 0$ from a query object. A *window* query can be thought of as a special case of a within query where the query object is a hyper-rectangle and the distance is zero. A *spatial join* [8] 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 *k*-nn query [16] returns the closest $k > 0$ spatial objects to a given query object. A *spatial semijoin* [8] 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. Another way to define this form of spatial semijoin is for every object $a \in A$, to find the nearest neighbor $b \in B$ and report $\langle a, b \rangle$. Using this definition, we can relax the 1-nn constraint and find the k nearest neighbors for every object $a \in A$, denoted $A \ltimes_k B$.

The *incremental distance query* [9] returns all the objects within a given distance d of a query object q, one at a time, in increasing order of distance from q. The incremental distance query algorithm can be used for both within queries, and k-nn queries. Retrieving all the objects from q within distance $d < \infty$ is a within query. Retrieving the first k objects and then stopping, with $d = \infty$, is a k-nn query. The incremental distance query algorithm assumes a spatial tree index where, as in the case of the R-tree [6] 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 sorted by distance from q. The queue is initialized with the root BB of the index. Objects 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, or there are no more elements in the queue.

Motion of a spatial object can be represented in several ways. The most common is samples, or discretely moving points [12]. For example, the motion of an aircraft can be represented by sampling its location using radar every 6 seconds. The problem with this representation is that the costs of updating every aircraft's location in a database every 6 seconds, and maintaining queries between updates, are prohibitive.

A *kinematic*[1] representation is an alternative to sampling. Kinematics represents the extent and location of an object as a function of time. In particular, a moving point can be 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 speed or direction of an object changes, the database is updated. For example, if an aircraft moving east at 500 miles per hour turns to head south, then the the function describing its motion is updated with a new velocity vector to reflect the new direction of travel. Errors that may arise due to discrepancies between the kinematic model of the objects motion, and the actual location of an object are beyond the scope of this paper. Kinematic data types have been studied in other domains such as simulation [13] (i.e, dead reckoning) to reduce network traffic in distributed simulations, and computational geometry [1] (i.e., kinematics). Kinematic data types, along with an event-driven query processing algorithm are discussed further in [10].

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 [6], or a spatial decomposition, such as the quadtree [18]. More recently, the indexing of moving objects has also been addressed [17, 21].

The *TPR-tree* [17] 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 defining 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

[1]Kinematics is 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.

by the TPR-tree update algorithms. As an update occurs, the BB is adjusted to be minimal at the update time. 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.

Event-driven query processing is used to maintain queries on kinematic data types. This is similar to event-driven simulation [4], 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 objects move.

In [10], two basic types of events are defined. One basic type, the *within event* (w-event) occurs when two objects move to be at a given distance d to a query object. For a linear point kinematic data type, 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 [15, 20] for more details on the computation of events between pairs of moving spatial objects. The other basic type of event is the *order change event* (oc-event). The oc-event occurs when two objects change order with respect to their distance to a query object. 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 $|p_1(time), q(time)| = |p_2(time), q(time)|$ for *time* (see [15, 20] for details). A special case of an oc-event is a *nearest neighbor event* (nn-event). Given a query object and its current k^{th} neighbor, the nn-event is the soonest oc-event to occur in the future out of all possible future oc-events among the objects in the data set. For example, suppose that q is a query point, p_k is its current k^{th} neighbor, and S is a set of kinematic data points $S = \{s_1 ... s_n\}$. 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. 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 $\{e_1 ... e_n\}$. The time of the next nn-event is the next time in the future the k^{th} neighbor of q will change.

An *incremental within event query* is similar to an incremental distance query, except that an event time metric is used instead of a distance metric [20]. 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. The 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 [9] (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.

A *next nearest neighbor event query* finds the next nearest neighbor given a query object and its current nearest neighbor. In [20], Tao and Papadias describe a method for finding the next nn-event given a query object, the current k^{th} neighbor, and a set of data points indexed in a tpr tree. To find the next event, the bounding box (BB) of each node is examined and the node is placed on a global priority queue sorted by the oc-event time of its BB. Processing starts with the root node of the tpr tree. The first object on the queue is dequeued and expanded, repeating the process recursively. When the first 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. If the next BB on the queue has an oc-event sooner than the candidate nn-event, then it is expanded. And objects in a subsequent leaf node with an oc-event sooner than the candidate nn-event replaces the candidate. When the oc-event time of the next node on the queue is later than the candidate, then the candidate oc-event is returned as the next nn-event and processing stops.

The next nn-event query supports the time parameterized k-NN algorithm (TP KNN) presented in [20]. This computes a k-nn query on kinematic objects, and then finds the next nn-event that will change the result. An even more efficient continuous k-nn algorithm is presented in [20] for finding many subsequent nn-events. Neither of these algorithms support updates.

The *continuous windowing* (CW) k-nn algorithm presented in [10] uses the w-event and oc-event types, the incremental within event query [20], along with an event-driven query processing algorithm to maintain k-nn queries. The idea is to filter the points from the data set with a circular window query centered at a moving query point, and then maintain the k nearest neighbors from the filtered result. At least k points must be selected in the filtering step. The window query is maintained by processing within events to keep a running set of points W that are within a given distance d of the query point. The k neighbors and next nn-event are computed from the points in W. The motivation for this approach is that within events are much cheaper than nn-events to process. This is because, an nn-event changes the k^{th} neighbor rendering all previous oc-events involving the old k^{th} neighbor obsolete. Within events, on the other hand, are independent of the query result.

3 Approach

The *continuous fuzzy set* (CFS) semijoin algorithm maintains a semijoin query result $Q \ltimes_k D$ on the sets of kinematic points Q and D as time advances and updates occur.

The main algorithm is a simple event-driven query processing algorithm that supports updates similar to the one presented in [10]. Events are placed on a priority queue sorted by time and dequeued one at a time for processing. There is one and only one nn-event or underflow event (described below) on the event queue for each query point in Q. Updates (insertions and deletions) are also processed as they occur. The assumption on updates is that there is no priori knowledge of updates, such as is the case in a real-time system.

The *fuzzy set* of a query point $q \in Q$ consists of all the points $S = \{s_1 ... s_n\}$, where $s_i \in D$, that are now or will be within some given distance of q sometime in the near future. This maintains a cloud of points around each query point. The next nn-event for any given point $q \in Q$ is computed from q's fuzzy set.

A fuzzy set is determined by a circle (or hypersphere for higher dimensional data) centered at q and with radius r known as the *query circle*. Radius r is chosen so that there are at least k points within distance r of q. The points in the circle, along with points that will enter the circle sometime in the near future, make up the fuzzy set of q. Scalar value r is generally a different value for each query point $q \in Q$. This region around the query point is denoted $\mathsf{circle}(q, r)$.

Time is divided up into uniform segments of time called fuzzy-set-intervals. The *fuzzy-set-interval* determines which points entering $\mathsf{circle}(q, r)$ belong to the fuzzy set. Only points that enter $\mathsf{circle}(q, r)$ during the current fuzzy-set-interval are in q's fuzzy set. At the start of each new fuzzy-set-interval, each query point's fuzzy set is updated (see Update_Fuzzy_Set() below).

Figure 1 illustrates an example fuzzy set for a single query point q in set Q, and data points $\{\mathbf{a,b,c,d,e,f,g,h,i}\} \in D$. Assume for this example, that the length of the arrows in the figure indicate how far each point will travel in one minute. In this example, the query point is not moving for simplicity. Also, assume for this example, that the current fuzzy-set-interval will end in one minute. In this example, all the points in $\mathsf{circle}(\mathbf{q}, r)$, and all the points that will enter $\mathsf{circle}(\mathbf{q}, r)$ within the next minute (up to the end of the current fuzzy-set-interval), are in the fuzzy set of point **q**. The length of each fuzzy-set-interval is a system parameter. Points $\{\mathbf{a,b,c,f,g,i}\}$ are in the fuzzy set of query point **q**. Note that point **d** is closer to the circle than point **c**, but it is moving slower and will not enter the circle during the next minute.

An *underflow event* occurs when the k^{th} neighbor of some query point q leaves $\mathsf{circle}(q, r)$. When this happens, r has to be increased to encompass more data points in $\mathsf{circle}(q, r)$. Underflow events are denoted by $\mathsf{uf}(q, p_k, t)$, where $q \in Q$ is the query point, $p_k \in D$ is the current k^{th} neighbor of q, and t is the underflow event time. For example, suppose that the query for Figure 1 is $Q \ltimes_3 D$, that is, for each point in Q find the 3 nearest neighbors in D. Point **g** is currently the 3^{rd} nearest neighbor from

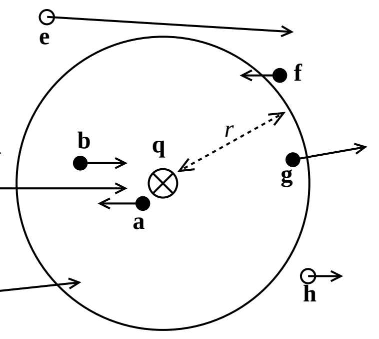

Figure 1: Example fuzzy set, where $\otimes$ is the query point **q**, $\bullet$ indicate points in **q**'s fuzzy set, and $\circ$ indicate points not in the fuzzy set.

point **q**. Recall from Section 1, the subscript k for a semi-join $Q \ltimes_k D$ denotes the number of nearest neighbors in D to be found for every point in Q. Suppose that point **g** will leave $\mathsf{circle}(\mathbf{q}, r)$ at time $t_{\mathbf{g}}$, then the underflow event is $\mathsf{uf}(\mathbf{q}, \mathbf{g}, t_{\mathbf{g}})$. The component members of an underflow event are denoted by $\mathsf{query\_pt}(\mathsf{uf}(q, p_k, t)) = q$, $\mathsf{kth\_pt}(\mathsf{uf}(q, p_k, t)) = p_k$, and $\mathsf{time}(\mathsf{uf}(q, p_k, t)) = t$. For the sake of consistency with nn-event notation (described below) we define $\mathsf{other\_pt}(\mathsf{uf}(q, p_k, t)) = p_k$.

An nn-event is denoted by $\mathsf{nn}(q, r, p_k, o, t)$, where q is the query point, r is the radius of $\mathsf{circle}(q, r)$, p_k is the k^{th} neighbor of q, o is the other data point that will become the new k^{th} neighbor at event time t. For example, suppose that the query for Figure 1 is $Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point **a** is currently the nearest neighbor of point **q**. Let the time for the next oc-event between points **a**, **c**, and **q** be time $t_{\mathbf{a,c}}$. Also, suppose that this is the next oc-event out of all the oc-events among the points in **q**'s fuzzy set. In this case, the nn-event for point **q** is $\mathsf{nn}(\mathbf{q}, r, \mathbf{a}, \mathbf{c}, t_{\mathbf{a,c}})$. To support fuzzy sets, the radius of the query circle is stored with the nn-event. The radius is not part of the definition of the nn-event itself, but it will be needed when the nn-event is processed. The component members of an nn-event are denoted by $\mathsf{query\_pt}(\mathsf{nn}(q, r, p_k, o, t)) = q$, $\mathsf{radius}(\mathsf{nn}(q, r, p_k, o, t) = r$, $\mathsf{kth\_pt}(\mathsf{nn}(q, r, p_k, o, t)) = p_k$, $\mathsf{other\_pt}(\mathsf{nn}(q, r, p_k, o, t)) = o$, and $\mathsf{time}(\mathsf{nn}(q, r, p_k, o, t)) = t$.

3.1 Data Structures

The event queue E-queue is a priority queue of events (underflow and nn-events) sorted by time. It is made up of three data structures. The first is a B+-tree variant called the *nearest neighbor event B-tree* (NN-B-tree). Every point p is assumed to have an associated unique id denoted $\mathsf{id}(p)$. The NN-B-tree B+-tree is sorted on the key $\mathsf{id}(\mathsf{query\_pt}(e))$ and yields the event values e (i.e., $\mathsf{id}(\mathsf{query\_pt}(e)) \rightarrow e$). In addition to implementing a range tree on $\mathsf{id}(\mathsf{query\_pt}(e))$, the B+-tree is augmented to implement a heap in the event times. In particular, in addition to the minimum and maximum keys, the minimum event time for a subtree in the NN-B-tree is propagated up to the root. Thus the result is a variant of a priority search tree [11]. Figure 2 shows an example NN-B-tree. The next event time is found by examining the root node, and returning the minimum event time in the root. To obtain the next event, the tree is traversed from its root to the leaf by following the minimum event time down the branches of the tree. The NN-B-tree allows efficient updates based on the id of query points from Q. However, in order to efficiently perform updates using a data point id as a key, additional data structures are needed. The K-B-tree is a standard B+-tree sorted by key id(kth_pt(e)) yielding the value id(query_pt(e)) (i.e., id(kth_pt(e)) $\rightarrow$ id(query_pt(e))). The O-B-tree is a standard B+-tree sorted by key id(other_pt(e)) yielding the value id(query_pt(e)) (i.e., id(other_pt(e)) $\rightarrow$ id(query_pt(e))).

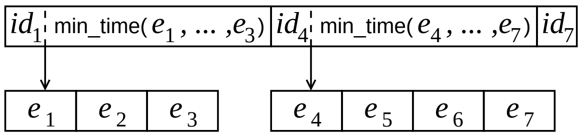
id_1 min_time(e_1, ... ,e_3)
id_4 min_time(e_4, ... ,e_7)
id_7
e_1 e_2 e_3
e_4 e_5 e_6 e_7

Figure 2: Example NN-B-tree with one root node, and two leaf nodes, where id_i = id(query_pt(e_i)).

Together, these three data structures NN-B-tree, K-B-tree, O-B-tree and their algorithms form the event queue E-queue. The algorithm to insert an event e is given in Figure 3. The algorithm to delete an event, given a query point q, is given in Figure 4. These are straight-forward since there is one and only one event for each query point in Q. The algorithm to delete all events involving a given data point p is more complicated since there may be many events involving p in the queue. The algorithm is given in Figure 5. First, all query points q for which p is the k^{th} neighbor are considered (line 1). The event for each q is found (line 2), then all the entries involving q are removed from the three B+-trees; O-B-tree, NN-B-tree, K-B-tree (lines 3–5). Since id(q) is unique, there is only one entry for a given q in each B+-tree. Second, all query points q are considered where p is the other point involved in q's event (line 7). Likewise, each entry for each of these query points are deleted from the three B+-trees (lines 8–11).

```
procedure E-queue_Insert($e$)
1. Insert id(query_pt($e$)) $\rightarrow e$ into NN-B-tree.
2. Insert id(kth_pt($e$)) $\rightarrow$ id(query_pt($e$)) into K-B-tree.
3. Insert id(other_pt($e$)) $\rightarrow$ id(query_pt($e$)) into O-B-tree.
```

Figure 3: E-queue_Insert()

```
procedure E-queue_Delete_QueryPt($q$)
1. Find entry id($q$) $\rightarrow e$ in NN-B-tree.
2. Delete entry id(kth_pt($e$)) $\rightarrow$ id($q$) from K-B-tree.
3. Delete entry id(other_pt($e$)) $\rightarrow$ id($q$) from O-B-tree.
4. Delete entry id($q$) $\rightarrow e$ from NN-B-tree.
```

Figure 4: E-queue_Delete_QueryPt()

To manage fuzzy sets, another set of data structures is

```
procedure E-queue_Delete_All_DataPt($p$)
1. foreach id($p$) $\rightarrow$ id($q$) in K-B-tree do
2.     Find entry id($q$) $\rightarrow e$ in NN-B-tree.
3.     Delete id(other_pt($e$)) $\rightarrow$ id($q$) from O-B-tree.
4.     Delete entry id($q$) $\rightarrow e$ from NN-B-tree.
5.     Delete entry id($p$) $\rightarrow$ id($q$) from K-B-tree.
6. end foreach
7. foreach id($p$) $\rightarrow$ id($q$) in O-B-tree do
8.     Find entry id($q$) $\rightarrow e$ in NN-B-tree.
9.     Delete entry id(kth_pt($e$)) $\rightarrow$ id($q$) from K-B-tree.
10.    Delete entry id($q$) $\rightarrow e$ from NN-B-tree.
11.    Delete entry id($p$) $\rightarrow$ id($q$) from O-B-tree.
12. end foreach
```

Figure 5: E-queue_Delete_All_DataPt()

used. The fuzzy set index (FS-index) keeps track of points in fuzzy sets, and the time the points will expire from each fuzzy set. The FS-index utilizes two B+-trees. The first B+-tree is the FS-B-tree. It is sorted by id(q) yielding the value $\{q, p, t\}$ (i.e., id(q) $\rightarrow \{q, p, t\}$), where $q \in Q$, $p \in D$, and t is the *expiration time* for p. The expiration time is the time when p leaves circle(q, r) and is no longer part of q's fuzzy set.

The other B+-tree used by the FS-index is the ID-B-tree. It is sorted by id(p) yielding id(q) (i.e., id(p) $\rightarrow$ id(q)). It serves a similar purpose as the K-B-tree or O-B-tree for the E-queue to support deletions of data points. Together, the FS-B-tree and the ID-B-tree form the FS-index. The algorithms to insert and delete objects in the FS-index are similar to those for the E-queue, but simpler since there are only two B+-trees involved.

Two tpr indexes [17] are used by the CFS algorithm. One index is on the query circles circle(q, r) rather than the query points in set Q. A second tpr index used by CFS is on the set of data points D.

3.2 CFS Algorithms

The *main loop* of the event-driven algorithm processes events and updates as they occur to maintain the query result over the moving points. The main loop invokes procedures Process_Event(), Update_Fuzzy_Set(), Insert_Data_Point(), Delete_Data_Point(), Insert_Query_Point(), and Delete_Query_Point() as needed (see below). Updates are not known in advance of their occurrence as we are assuming a real-time system. Processing continues indefinitely.

When an event on the E-queue comes due, it is dequeued and passed to Process_Event() (Figure 6). Every query point has either an nn-event, or an underflow event associated with it in the event queue, E-queue, even if the event time is ∞. This is done so that every query point and its k^{th} neighbor can be found simply by examining the queue. For an nn-event (line 2), if other_pt(e) was not previously part of the k-neighbor-set for q (line 3), then it pushes the current k^{th} neighbor out of the set and other_pt(e) becomes the new k^{th} neighbor. This necessitates an update to the query result. The query re-

```
procedure Process_Event($e$)
 1. Point $q \leftarrow$ query_pt($e$)
 2. if $e$ is an nn-event then
 3.    if kth_pt($e$) is becoming $k+1$ neighbor of $q$ then
 4.       update the query result
 5.    Get all entries for $q$ from FS-index, and remove
       expired points to get fuzzy set $S$.
 6.    if count of expired points > expired_threshold then
 7.       remove all expired entries for $q$ from FS-index
 8.    Enqueue_Event($S, q$,radius($e$),kth_pt($e$))
 9. else if $e$ is an underflow event then
10.    Handle_Underflow($q$)
```

Figure 6: Process_Event()

sult is updated by reporting $\langle$kth_pt$(e), q\rangle$ deleted, and $\langle$other_pt$(e), q\rangle$ inserted (line 4). For example, suppose that the query for Figure 1 is $Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point **a** is currently the nearest neighbor of point **q**. The event on the queue for point **q** is nn$(\mathbf{q}, r, \mathbf{a}, \mathbf{c}, t_{\mathbf{a},\mathbf{c}})$, where time $t_{\mathbf{a},\mathbf{c}}$ is the time points **a** and point **c** will be equidistant from point **q**. When this event comes due and is processed, point **c** pushes point **a** out of the k-neighbor-set (in this case the 1-neighbor-set) of point **q**, and becomes the new nearest neighbor.

If, on the other hand, other_pt(e) was already part of the k-neighbor-set, then it simply becomes the new k^{th} neighbor, and the current k^{th} neighbor becomes the $k-1$ neighbor. For example, suppose that the query for Figure 1 is $Q \ltimes_2 D$, that is, for each point in Q find the 2 nearest neighbors in D. Point **b** is currently the 2^{nd} nearest neighbor from point **q**. In this example, the nn-event for point **q** is nn$(\mathbf{q}, r, \mathbf{b}, \mathbf{a}, t_{\mathbf{b},\mathbf{a}})$, because point **a** will be the first to be equidistant with point **b** from point **q** before any other point in point **q**'s fuzzy set. When nn$(\mathbf{q}, r, \mathbf{b}, \mathbf{a}, t_{\mathbf{b},\mathbf{a}})$ comes due at time $t_{\mathbf{b},\mathbf{a}}$, point **a** becomes the new k^{th} neighbor (2^{nd} neighbor), but point **b** stays in the 2-neighbor-set of point **q**, so the query result does not change. However, since the 2^{nd} neighbor changed, a new nn-event for point **q** must be calculated and enqueued.

The new nn-event is calculated from q's fuzzy set. The fuzzy set S for q is stored in the FS-index. All points in the FS-index that have not expired are in q's current fuzzy set (line 5). A point expires from the fuzzy set when it leaves the circle around q. If the number of expired points exceeds a certain threshold, then all expired points for q are removed from the FS-index (line 6) This keeps down the number of expired entries in the FS-index. The fuzzy set S is used to compute the next event for q (line 8) (see description of Enqueue_Event() below). An *underflow event* occurs when circle(q, r) contains less than k points (line 9). When underflow occurs, the fuzzy set must be expanded (line 10) (see description of Handle_Underflow() below).

Enqueue_Event() (Figure 7), called from line 8 of Figure 6, computes the next event for a query point from its fuzzy set. The current k^{th} neighbor is removed from the fuzzy set S (line 1). The remaining points in fuzzy set S are each considered for the next nn-event by computing each of

```
procedure Enqueue_Event($S, q, r, p_k$)
1. $S \leftarrow (S - p_k)$
2. Find the next nn-event $e$ from among the points in $S$.
3. if kth_pt($e$) will expire before $e$ occurs then
4.    enqueue an underflow event for $q$ in E-queue.
5. else enqueue $e$ in E-queue.
```

Figure 7: Enqueue_Event()

their next occurring order change events (oc-events) in turn (line 2). Recall from Section 2 that an oc-event for a data point p occurs when p moves to be at the same distance to q as its current k^{th} neighbor. The soonest oc-event becomes the next nn-event (line 5), unless circle(q, r) underflows sooner. In that case, an underflow event is enqueued instead (line 4).

```
procedure Handle_Underflow($q$)
1. Remove all entries for $q$ from FS-index.
2. $n \leftarrow \lceil k * circle\_factor \rceil$
3. Get new set $S$ of $n+1$ neighbors around $q$.
4. $r \leftarrow (\|q, s_n\| + \|q, s_{n+1}\|)/2$, where $s_n, s_{n+1} \in S$
5. Remove $s_{n+1}$ from $S$.
6. Add points to $S$ that will enter circle($q, r$) during the
   current fuzzy-set-interval.
7. Insert points $S$, and their expiration times
   into FS-index.
8. Enqueue_Event($S, q, r, s_k$),
   where $s_k \in S$ is the $k^{\text{th}}$ neighbor of $q$.
9. Remove old circle centered at $q$ from query point tpr
   tree, and insert circle($q, r$).
```

Figure 8: Handle_Underflow()

Handle_Underflow() (Figure 8), called from line 10 of Figure 6, resizes the fuzzy set for a query point. The old fuzzy set needs to be removed from the FS-index, since the radius defining the expiration times for the points in the old fuzzy set will change (line 1). The circle around q is calculated to initially contain some multiple of k points. The global constant *circle_factor* > 1 is used to determine how many points to start with in a circle (line 2). An incremental distance algorithm [9] (see Section 2) is used to get the $n+1$ neighbors of q using the tpr index on the data points (line 3). The $n+1$ neighbor, s_{n+1}, is used to determine the radius of the new circle. The new radius is the average of the distances from q to the n^{th} neighbor, s_n, and q to s_{n+1} (line 4). Note that the Euclidean distance at the current time between two kinematic points q and p is denoted $\|q, p\|$. This technique for finding the radius helps to avoid the situation where points instantly leave the circle after it is resized. Once the radius is computed, s_{n+1} is discarded from the set S because it is outside the circle (line 5). The rest of the fuzzy set is found using an incremental within event algorithm [20] (see Section 2) on the tpr index on the data points (line 6). At this point S contains all the points in q's new fuzzy set. The points in fuzzy set S are inserted into the FS-index along with their expiration times (line 7). The next nn-event is computed from the points in S and enqueued (line 8). Finally, the tpr index on the query circles is updated (line 9).

For example, suppose that the query for Figure 1 is

$Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point **a** is currently the nearest neighbor of point **q**. Also suppose that the radius of the circle is not r as in the figure, but is smaller, and suppose that an underflow event has just occurred. In other words, the radius of the circle is at the distance from point **q** that point **a** is at right now, say r_{old}. Suppose also that *circle_factor* $= 3$. When Handle_Underflow() is invoked, we get $n = 1 * 3 = 3$ (line 2). We then find the $n+1$, or 4 nearest neighbors to point **q** (line 3). Set S now contains points $\{$**a, b, g, f**$\}$. The new distance r (the large circle in Figure 1) is calculated to be halfway between point **g** and point **f** from point **q** (line 4). Once r is computed, the 4^{th} neighbor of q is removed from S leaving $\{$**a, b, g**$\}$ (line 5). Suppose that the end of the current fuzzy-set-interval is one minute in the future. All the points that will enter circle$(\mathbf{q}, r)$ before the end of the current fuzzy-set-interval (e.g., within the next minute) are added to set S to give $\{$**a,b,c,f,g,i**$\}$ (line 6). In this case, point **f** ends up back in set S, but it would not if it were moving away from the circle. The points $\{$**a,b,c,f,g,i**$\}$, along with their expiration times are inserted into the FS-index (line 7). They are also used to find the next nn-event (line 8). The old circle circle$(\mathbf{q}, r_{old})$ is removed from the circle tpr tree and the new circle circle$(\mathbf{q}, r)$ is inserted (line 9).

```
procedure Update_Fuzzy_Set()
1.  foreach circle($q, r$) in the query circle tpr tree do
2.      Add new points to $q$'s fuzzy set $S$ that will enter
        circle($q, r$) during the current fuzzy-set-interval.
3.      Find the next nn-event $e$ from among the points in $S$.
4.      if $e$ occurs before the currently enqueued event for $q$
5.          then replace currently enqueued event with $e$.
6.  end foreach
```

Figure 9: Update_Fuzzy_Set()

Update_Fuzzy_Set() (Figure 9) is invoked at the start of each new fuzzy-set-interval to update the fuzzy set for each query point. This finds all the data points that will enter query circles during the new fuzzy-set-interval segment of time. The tpr index on the query circles is scanned to get all the query circle circle(q, r) (line 1). The fuzzy set S for each q is updated by finding all the new data points entering circle(q, r) using an incremental within event algorithm [20] (see Section 2) on the data point tpr index (line 2). The current k^{th} neighbor $s_k \in S$ is found, and then the nn-event e from the rest of the points in S is computed (line 3). This is done by considering each point $s_i \in S, i \neq k$ for the next nn-event by computing each s_i's next oc-event with respect to s_k and q. The soonest oc-event out of all is the next nn-event e. If e occurs before the event that is currently in the event queue E-queue for q, then e replaces the one on the queue for point q (line 5).

Insert_Data_Point() (Figure 10) is invoked when a new point p is added to the set of data points D in the semi-join query $Q \ltimes_k D$. The query circle tpr index is used to find all circle(q, r)'s that currently contain, or will contain

```
procedure Insert_Data_Point($p$)
1.  foreach circle($q, r$) with $p$ in $q$'s fuzzy set do
2.      if $p$ is in the $k$-neighbor-set of $q$ then
3.          Update the query result.
4.          Remove $q$'s event from the E-queue.
5.          Get all entries for $q$ from FS-index, and remove
            expired points to get fuzzy set $S$.
6.          Enqueue_Event($S, q, r, s_k$),
            where $s_k \in S$ is the $k^{\text{th}}$ neighbor of $q$.
7.      else if $p$ introduces a sooner nn-event for $q$ then
8.          Replace the nn-event for $q$ in E-queue.
9.      end if-else-if
10. end foreach
11. Insert $p$ into the data point tpr index.
```

Figure 10: Insert_Data_Point()

p between now and the end of the current fuzzy-set-interval (line 1). In particular, this entails the performance of two operations using the query circle tpr index. The first finds all the circles that currently contain point p using a within distance $d = 0$ query [9]. The second uses an incremental within event algorithm [20] (see Section 2) to find all the circles that will contain p before the end of the fuzzy-set-interval. Each query circle circle(q, r) is processed in turn. If p is closer to a given q than the k^{th} neighbor of q, then it is in the k-neighbor-set of q (line 2). Point q's entry in the event queue can be used to find the k^{th} neighbor of q since both nn-events and underflow events keep track of the k^{th} neighbor. When p is in the k-neighbor-set of q, then the current k^{th} neighbor becomes the $k+1$ neighbor. The entry involving the old k^{th} neighbor $\langle k^{\text{th}}, q\rangle$ is removed from the query result and the new entry $\langle p, q\rangle$ is added (line 3). When the k^{th} neighbor changes, the nn-event or underflow event changes as well, so the old event needs to be removed from E-queue (line 4). The new event is calculated from the fuzzy set of q (lines 5–6). If p is not in the k-neighbor-set, then it may still affect the next nn-event. If p's next oc-event occurs before the current nn-event for q, then the oc-event becomes the new nn-event, and replaces the old nn-event on the queue (lines 7–8). After all circles have been processed, the tpr index on the data points is updated (line 11).

Delete_Data_Point() (Figure 11) is invoked when a data point p is deleted from the set of data points D in the semi-join query $Q \ltimes_k D$. First, p is removed from the data point tpr index (line 1), and FS-index (line 2). All the circles that contain p, or would contain p during the current fuzzy-set-interval are processed in turn (line 3). These circles are found by applying an incremental distance [9], and incremental event algorithm [20] (see Section 2) on the data point tpr index. If p is the current k^{th} neighbor, or closer to a given q than its k^{th} neighbor, then the events and query result change (line 4). The current $k+1$ neighbor becomes the k^{th} neighbor. After the old event for q is removed from the event queue (line 5), the fuzzy set S is found (line 6), and checked for underflow (line 7). Underflow results in a resizing of the fuzzy set, and a new nn-event is enqueued (line 8). If the fuzzy set does not underflow, then a new

```
procedure Delete_Data_Point($p$)
 1. Remove $p$ from the data point tpr index.
 2. Remove all entries involving $p$ from FS-index.
 3. foreach circle$(q, r)$ with $p$ in $q$'s fuzzy set do
 4.    if $p$ is in the $k$-neighbor-set of $q$ then
 5.       Remove $q$'s event from the E-queue.
 6.       Get all entries for $q$ from FS-index, and remove
          expired points to get fuzzy set $S$.
 7.       if number of data points in circle$(q, r) < k$ then
 8.          Handle_Underflow($q$)
 9.       else
10.          Enqueue_Event($S, q, r, s_k$),
             where $s_k \in S$ is the new $k^{\text{th}}$ neighbor of $q$.
11.       Update query result.
12.    else if $p$ is involved in $q$'s enqueued event then
13.       Remove $q$'s event from the E-queue.
14.       Get all entries for $q$ from FS-index, and remove
          expired points to get fuzzy set $S$.
15.       Enqueue_Event($S, q, r, s_k$),
          where $s_k \in S$ is the new $k^{\text{th}}$ neighbor of $q$.
16.    end if-else-if
17. end foreach
```

Figure 11: Delete_Data_Point()

nn-event is enqueued given the new k^{th} neighbor (line 10). The result is updated by deleting the old k^{th} neighbor and inserting the new one (line 11). When p is not in the k-neighbor-set, but is involved in the nn-event for q (line 12), then the nn-event changes. In particular, the new nn-event is found from the points in q's fuzzy set S, replacing the old nn-event in the queue (lines 13–15).

```
procedure Insert_Query_Point($q$)
1. $n \leftarrow \lceil k * \textit{circle\_factor} \rceil$
2. Get new set $S$ of $n + 1$ neighbors around $q$.
3. $r \leftarrow (\|q, s_n\| + \|q, s_{n+1}\|)/2$, where $s_n, s_{n+1} \in S$
4. Remove $s_{n+1}$ from $S$.
5. Add points to $S$ that will enter circle$(q, r)$ during the
   current fuzzy-set-interval.
6. Insert points $S$, and their expiration times
   into FS-index.
7. Enqueue_Event($S, q, r, s_k$),
   where $s_k \in S$ is the $k^{\text{th}}$ neighbor of $q$.
8. Report $\langle s_i, q \rangle$ inserted to result for the closest $k$
   points $s_i \in S$ to $q$.
9. Insert circle$(q, r)$ into query circle tpr index.
```

Figure 12: Insert_Query_Point()

Insert_Query_Point() (Figure 12) is invoked when a query point is inserted into Q in the semijoin query $Q \ltimes_k D$. This procedure is similar to Handle_Underflow() except that there are no previous entries for q in FS-index or the query circle tpr index to remove. Lines 1–7 are identical to lines 2–8 of Figure 8. Before finishing, the k neighbors of q are added to the query result (line 8), and the query circle is indexed (line 9).

```
procedure Delete_Query_Point($q$)
1. Delete the current $k$ neighbors to $q$ from query result.
2. Remove any entries for $q$ from FS-index, E-queue,
   and the query point tpr index.
```

Figure 13: Delete_Query_Point()

Delete_Query_Point() (Figure 13) is invoked when a query point is deleted. It first updates the query result (line 1). This is done by applying an incremental distance algorithm [9] (see Section 2) on the data point tpr tree with q as the query point to determine what entries to delete. It then removes any entires involving the query point q from all the data structures (line 2).

4 CFS vs. CW

The CFS algorithm somewhat resembles the CW k-nn algorithm for one query point presented in [10]. However, there are significant differences. The similarity is that both approaches maintain a circular region around a query point with the constraint that it contain at least k points at all times. This filters the data points for candidates from which to select the k nearest neighbors.

The differences are in the other ways in which the circles are used. In the CW algorithm, the nn-event is computed from only those points found inside the query circle. In the CFS algorithm, points entering the circle in the near future are also considered for the next nn-event. This reduces the number of "false" nn-events that need to be changed before they occur when new candidates enter the widow of the CW algorithm. The CFS algorithm introduces the notion of fuzzy-set-intervals to limit which points are considered for the nn-event. Points entering the window in the distant future are not likely to be involved in the next nn-event. In the CW algorithm, within events are used to process points entering the window of a single query point. The CFS algorithm does not process within events as they occur. Instead, it only processes nn-events and underflow events for each query point. Within events are used in the CFS algorithm to determine when fuzzy set elements will expire.

To scale the CW algorithm to handle many query points at the same time, additional data structures would be needed to keep track of nn-events, the contents of each query circle, the size of each query circle, and underflow. This, in addition to the sheer number of within events that would need to be queued and processed makes scaling the CW algorithm an inferior solution to the CFS algorithm.

5 Experiments

For the purpose of evaluating our algorithm, we scale up an existing k-nn algorithm to perform semijoin queries. We then compare the simple scaling of the previous work to the CFS algorithm.

In [10], the TP k-nn algorithm [20] was extended to support updates (presented as the ETP algorithm in [10]). Here, we scale up the ETP algorithm to do semijoins in addition to updates. We call the extension to perform semijoins the TP-semijoin (TPS) algorithm. To scale the ETP algorithm to perform semijoins, an event queue containing an nn-event for each query point is added. If for some query point q, no such event exists, then a pseudo event $\mathsf{nn}(q, p_k, p_k, \infty)$ is added to keep track of the current k^{th} neighbor p_k. When an update occurs, the event queue is scanned to determine what part of the query result, and which events need to be modified. If the set of k neighbors changes due to an update, then new neighbors and events are found using a tpr index on the data points similar to what was done in the ETP algorithm in [10]. No tpr index for the query points is needed since all the query points are in the event queue.

As discussed above (Section 4), the CW algorithm also presented in [10] would not scale well because there would be too many within events to process. Note that a straightforward scaling of the CW algorithm given in [10] can be achieved by adding an nn-event queue in addition to the within event queue. In preliminary results, scaling of the CW algorithm was found to be significantly less efficient than the TPS algorithm.

5.1 Data Sets

We used both real aircraft flight data and synthetic uniformly-distributed data in our experiments. 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.

All synthetic uniformly-distributed data sets are generated using a data generation tool developed by Saltenis et. al. [17]. The synthetic moving points are uniformly distributed over a 1000x1000 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 speed, but not the current location of each point. No new points are introduced after the start time, nor are any removed. 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 14 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 [3] to the 2D latitude-longitude points forming a polyline from earliest to latest sampled location in time.[2] 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 was 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 $[20°, 60°]$ latitude by $[-135°, -60°]$ 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 period. Each aircraft data set covers a time period of two hours.

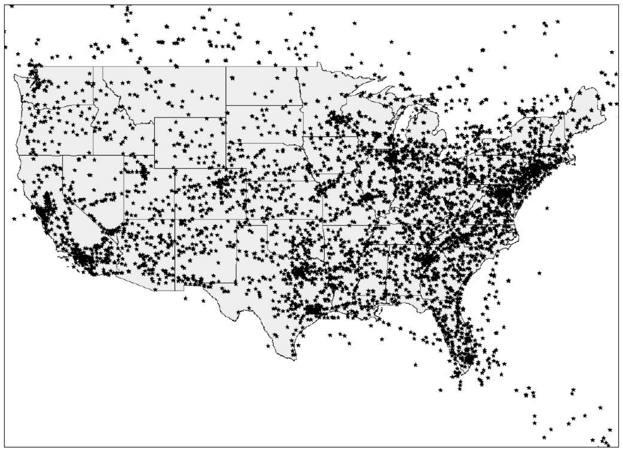
Figure 14: Snapshot of aircraft flight data.

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 in the number of aircraft as flights land and take off. Figure 15 shows the mean and standard deviation in the size of the data sets used over the entire 2 hour time interval covered by each data set. The figure also shows the average update interval (UI) for each aircraft data set.

μ	4453	9021	12690	17106
σ	330.8	680.8	962.4	1293
UI	700.7	712.8	725.1	734.6

Figure 15: Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. 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.

To make full use of the real data sets available, each data set was divided into 12 subsets starting at evenly spaced start times over the duration of the data set. For example, for a data set covering a time period of 900 units, the subset start times are 0, 75, 150, etc. If the duration of the experiment is longer than the time between subsets, then

[2]Although experiments were conducted on 2-dimensional data, the algorithms presented in this paper are applicable to higher dimensions.

the subsets overlaped. If needed, the spacing between start times was adjusted so the experiments didn't run past the time covered by the subset. For our 900 time unit example above, if the experiment duration is 100 time units, then the time between start times might be only 72 time units. The time domains of each subset were then transformed to start at time 0.

Each query was performed on combinations of these subsets, not including self semijoins. Pairs of subsets were chosen at random without replacement from all possible combinations for a total of 100 joins per query. Only subsets taken from the same original data set are used in a query, so the semijoin sets are approximately the same size for each query. In other words, the number of query points is about the same as the number of data points in each semijoin query. This technique was used on both the synthetic and real data sets for comparability.

5.2 Results

Experiments were conducted in a simulation of a real-time system in which semijoin queries are maintained over time as updates occur. The experiments measured the total number of disk accesses over the duration of a query. Since we are concerned primarily with the maintenance portion of the query, the number of disk accesses used to compute the initial join result are not included. The number of disk accesses over 100 trials was averaged to yield the experiment results for a given query.

The implementation of the event queue used the generalized search tree (GiST) [7] version 0.9beta1 code. The code was compiled using gcc 2.96. The experiments were run on several VLSI 80686 CPU based machines running Linux.

The primary independent variables for comparison are the mean data set size (μ), and number of neighbors (k) to find for each query point. For experiments where these do not vary, the defaults are $\mu = 9021$ for real aircraft data, $\mu = 10000$ for synthetic uniform data, and $k = 1$. Other general parameters, unless otherwise specified, are query duration of 130 seconds, disk page size of 4096 bytes, and disk cache size of 8 pages for each disk-based data structure.

Every cache page uses a least recently used (LRU) replacement policy except for the event queues. The event queues use a *Greatest Next Event* (GNE) replacement policy. GNE removes the page whose minimum next event time is the furthest in the future out of all pages in the cache. GNE worked better than LRU for small pages (e.g., 1024 bytes) and large caches (e.g., 32 pages). However, when the cache size was reduced, and the page size increased, we found nearly no difference between the LRU and GNE policies. Therefore, LRU can be used with nearly the same results as GNE.

Parameters specific to the CFS algorithm, unless otherwise specified, are *circle_factor* $= 2$, *expired_threshold* $= 25$ events, and fuzzy-set-interval duration of 128 time units to ensure that at least one call is made to $\mathsf{Update\_Fuzzy\_Set}()$ per each 130 second query. We found these particular settings for the CFS algorithm to be nearly optimal in our experiments.

The purpose of the first experiment is to determine which algorithm, TPS or CFS, performs better in terms of disk accesses for different data sets sizes. Figure 16 shows the results for (a) real aircraft flight data, and (b) uniform synthetic data. Parameter k is 1. The x-axis is the average data set size (see row 1 in Figure 15), and the y-axis is the number of disk accesses in millions (M). The points indicated by $\triangle$ symbols are the number of disk accesses for the CFS algorithm, while the $\diamond$ symbol indicates the number of disk accesses for the TPS algorithm. For the aircraft data, the CFS algorithm has 5 times fewer disk accesses than the TPS algorithm for the largest data sets tested. For the uniform synthetic data, the CFS algorithm has 10 times fewer disk accesses than the TPS algorithm for the largest data sets tested.

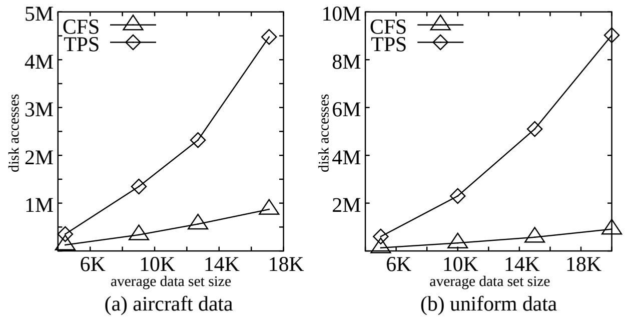

Figure 16: Disk accesses with respect to data set size.

The purpose of the second experiment is to determine the relative performance of the CFS algorithm to the TPS algorithm when k is varied. Figure 17 shows the results for (a) real aircraft flight data (data set size $\mu = 9021$), and (b) uniform synthetic data (data set size $\mu = 10000$). The x-axis is k, and the y-axis is the number of disk accesses in millions (M). The points indicated by $\triangle$ symbols are the number of disk accesses for the CFS algorithm, while the $\diamond$ symbol indicates the number of disk accesses for the TPS algorithm. The CFS algorithm has fewer accesses than the TPS algorithm, but the number of disk accesses for the CFS algorithm increases faster as the value of k is increased.

The purpose of the third experiment is to study the effect of *circle_factor* on the performance of the CFS algorithm. Figure 18a shows the results for real aircraft flight data (data set size $\mu = 4453$), and $k = 1$. The x-axis is the *circle_factor*, and the y-axis is the number of disk accesses in thousands (K). The points indicated by $\triangle$ symbols are the number of disk accesses for the CFS algorithm. Although the TPS algorithm is not affected by *circle_factor*, for comparison purposes, we show the number of disk accesses ($\diamond$ symbol) for this data. From Figure 18a it can be seen that a *circle_factor* value of 2 yields the best perfor-

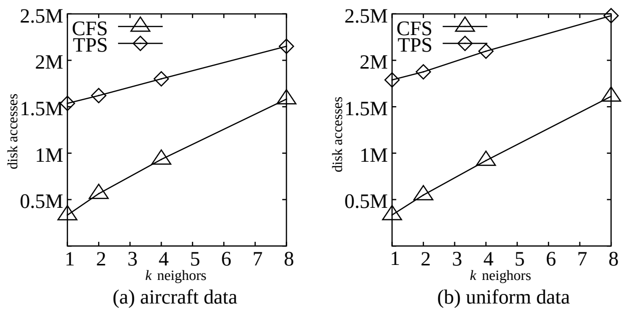

Figure 17: Disk accesses with respect to *k* with *circle_factor* = 2.

mance for the CFS algorithm with $k = 1$. A *circle_factor* < 2 for $k = 1$ is not meaningful since there needs to be at least $k + 1$ points inside a query circle when it is resized. As we see, larger *circle_factor* values do lead to more disk accesses for the CFS algorithm but this is still much lower than the number of disk accesses for the TPS algorithm.

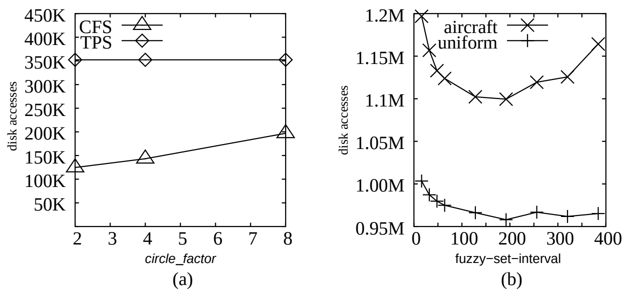

Figure 18: CW algorithm parameters. (a) Disk accesses vs. *circle_factor*. (b) Disk accesses vs. fuzzy-set-interval.

Figure 18b shows disk accesses (y-axis) versus different values for the CFS fuzzy-set-interval parameter (x-axis) for aircraft data ($\mu = 9021$), and $k = 1$. The $\times$ symbols indicate disk accesses for aircraft data, and the $+$ symbol indicates disk accesses for uniform data. Each point is an average over 50 trials. Small values (< 128) show increased disk activity due to more frequent calls to Update_Fuzzy_Set(). Larger values (> 192) show increased disk activity due to larger fuzzy sets for each point for the uniform data set. The reason why aircraft data does not exhibit the same performance characteristics as for uniform data for larger values of fuzzy-set-interval is unclear, and should be a topic for future research.

Data structure size: The implementation resulted in the following entry sizes. The tpr index entries were 32 bytes for internal node and leaf node entries. The NN-B-tree leaf node entries were 115 bytes, and internal node entries were 9 bytes. The FS-B-tree leaf node entries were 73 bytes, and internal node entries were 4 bytes. The K-B-tree, O-B-tree, and ID-B-tree leaf node entries were 8 bytes, and internal node entries were 4 bytes.

Given these numbers, we can estimate the size of the data structures under certain assumptions. Assume for query of $k = 1$ on data sets of size 20k, and page size of 4096 bytes, that 70% space utilization is achived. For each of the 20k query points, there is one event in the E-queue data structure. This gives $\lceil 20000/((4096 * 0.7)/115) \rceil = 803$ pages of leaf nodes in the NN-B-tree of the E-queue. and $\lceil 803/((4096 * 0.7)/9) \rceil = 3$ pages of internal nodes. This gives a total size of $(803 + 3) * 4069 = 3301376$ bytes on disk total for the NN-B-tree. For the K-B-tree and the O-B-tree we get $\lceil 20000/((4096 * 0.7)/8) \rceil = 56$ leaf node pages and 1 internal node page for a total of $(56 + 1) * 4096 = 233472$ bytes on disk for each. The total space taken by the NN-B-tree for this example is $3301376 + 262144 + 262144 = 3768320$ bytes.

To examine the FS-index, lets assume an average of 3 elements in each fuzzy set. This give $(20000 * 3) = 60000$ entries in the FS-index. For the FS-B-tree, this results in $\lceil 60000/((4096 * 0.7)/73) \rceil = 1528$ pages of leaf nodes, and $\lceil 1528/((4096 * 0.7)/4) \rceil = 3$ pages of internal nodes. This gives a total of $(1528 + 3) * 4096 = 6270976$ bytes on disk. For the ID-B-tree, we get $\lceil 60000/((4096 * 0.7)/8) \rceil = 168$ leaf node pages and 1 internal node page for a total of $(168 + 1) * 4096 = 692224$ bytes on disk. The total space take by FS-index for this example is $6270976 + 692224 = 6963200$ bytes.

Finally, for the tpr indexes, we get $\lceil 20000/((4096 * 0.7/32) \rceil = 224$ pages of leaf nodes, and $\lceil 224/((4096 * 0.7)/32) \rceil = 3$ pages of internal nodes. This gives a total $(224 + 3) * 4096 = 929792$ bytes on disk per tpr index.

The total disk space used for the E-queue, FS-index, and two tpr tree indices is $3768320 + 6963200 + (2 * 929792) = 12591104$ bytes in this example.

6 Concluding Remarks

Even with the improved performance over previous work, the number of disk accesses is still too high for the relatively small data sets to be practical. As can be seen in the example at the end of the last section, the size of the data structures is relatively small, yet the disk accesses are in the millions for small data sets over a short time interval. The main cost arises from updates to the data structures. In order to scale these algorithms up to large data sets (i.e., in the order of millions of objects) future work must focus on update efficient disk based data structures for indexing moving objects, event queues, and range trees.

In spite of these shortcomings, our experiments in Figure 16 show that the CFS algorithm clearly outperforms the TPS algorithm. In some cases, the difference can be as much as an order of magnitude (Figure 16b). The CFS algorithm is the first algorithm of its kind to maintain spatial semijoin results on kinematic data types, over an indefinite period of time, and with no prior knowledge of the updates that will be made.

References

[1] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, New Orleans, LA, January 1997.

[2] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 44–53, Edmonton, Canada, July 2002.

[3] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[4] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[5] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[7] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, September 1995.

[8] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. (Also University of Maryland Computer Science TR–3919).

[10] G. S. Iwerks, H. Samet, and K. Smith. Continuous *k*-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 512–523, Berlin, Germany, September 2003.

[11] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[12] M. A. Nascimento, R. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 171–188, Edinburgh, UK, September 1999.

[13] Standards Committee on Interactive Simulation (SCIS). *IEEE Std 1278.1-1995*. IEEE Computer Society, USA, March 1996.

[14] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, October 2002.

[15] K. Raptopoulou, A. N. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[17] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD Conference*, pages 331–342, Dallas, TX, May 2000.

[18] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[19] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13th IEEE Conference on Data Engineering (ICDE)*, pages 422–432, Birmingham, U.K., April 1997.

[20] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM Transactions on Databases Systems (TODS)*, 28(2):101–139, June 2003.

[21] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.