2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware

Interval-Based Nearest Neighbor Queries Over Sliding Windows from Trajectory Data

Yan Huang, Chengyang Zhang
Department of Computer Science and Engineering
University of North Texas
Denton, TX U.S.A
Email: {huangyan,chengyang}@unt.edu

***Abstract*—This paper proposes a new type of query for moving object trajectories – Continuous Interval-based Nearest Neighbor (CINN) Query. We clearly define the CINN in the context of streaming trajectory data. To efficiently process CINN queries, we first propose a spatial hashing algorithm (*SH*). Then we show that a new temporal hashing algorithm (*TH*) using speed constraints can save substantial computation cost. To reduce memory cost, we further propose the temporal hashing with dropping optimization (*THwD*) algorithm. Extensive experiment results on large trajectory datasets show that CINN queries can be effectively answered using our proposed algorithms. With realistic speed constraints, the *TH* optimization can save the computation time by nearly an order of magnitude compared with the *BF* algorithm, and by 5 times compared with the *SH* algorithm. The *THwD* algorithm can further save the memory space by nearly an order of magnitude.**

I. Introduction

The continuing proliferation of location-enabled devices and mobile communication enables the collection of large trajectory dataset from moving objects in real time. Nearest neighbor (NN) query is one of the most important queries in both static spatial databases and moving object databases. However, most NN queries are what we termed as *instant-based* queries, which do not consider distances accumulated over a time interval. In this paper, we define the *Interval-based* Nearest Neighbor (INN) of a trajectory. The INN is extended to Continuous Interval-based Nearest Neighbor (CINN) where the time interval is a sliding window.

A trajectory is a sequence of time stamped locations. The distance between two trajectories P_1 and P_2 is defined as the summation of distances from each point of P_1 to each point of P_2 with the same timestamp. A trajectory's interval-based nearest neighbor (INN) is its nearest neighbor trajectory based on the above distance definition. INN is useful in identifying objects that move in close spatial proximity over time. For example, in Ecology, locating devices can be attached to animals to study their social structures and migrating behaviors in real time. To track the closest companion of an animal such as a caribou, an ecologist may be interested in:"Find the nearest neighbor of the caribou named Flo in the past two days, or past two months continuously". The sliding window can be specified by the user. If the window is "now", then the CINN query degenerates into a continuous *instant-based* NN. If the window is "unbounded", then the CINN query will involve historical data migrated to disks.

Although k nearest neighbor (k-NN) queries [15], [16], [10], continuous k-NN (CKNN) [23], [17], [19], [4], [22], [21], [14], and recently k-NN over sliding windows [13] have been investigated extensively in literature, they all focus on *instant-based* queries and the work in *interval-based* NN is scarce. The closest work on interval-based k-nearest neighbor that we are aware of is the paper [9]. However, this work only considers the historical trajectories. Incremental query processing over sliding window is not addressed.

In this paper, we deal with processing and optimizing continuous interval-based nearest neighbor (CINN) queries over sliding window in real time. Answering CINN queries from large number of moving objects that report location updates in real time is challenging. The goal is to design efficient algorithms to save computation cost for answering multiple CINN queries while at the same time reduce memory consumption. The paper makes the following contributions:

- We define a new type of query called Continuous Interval-based Nearest Neighbor (CINN) query over sliding window. The definition is *interval-based* as opposed to *instant-based*. These queries are useful for identifying trajectories with close spatial proximity over time;
- We focus on movement-based location update (MLU) model, and propose *lazy location update* and *incremental distance computation* to provide basic optimizations.
- We propose several new algorithms and optimization techniques to process real time CINN queries. We start with a spatial hashing (*SH*) algorithm. Then we propose a temporal hashing (*TH*) algorithm that uses realistic speed constraints to save computational cost. Further, we propose the temporal hashing with dropping optimization (*THwD*) to save memory space.
- We conduct extensive experiment study over a large moving object dataset generated over a real road network. The results show that CINN queries can be effectively answered using our proposed algorithms. With realistic speed constraints, the *TH* optimization can save the computation time by nearly an order of magnitude compared with the *BF* algorithm, and by 5 times compared with the *SH* algorithm. The *THwD* algorithm can further save the memory space by nearly an order of magnitude with similar computational cost as that of *TH*.

978-0-7695-3650-7/09 $25.00 © 2009 IEEE
DOI 10.1109/MDM.2009.32

IEEE computer society

The rest of the paper is organized as follows. The related work is first reviewed in section II. Then CINN is precisely defined with related terms in section III. Algorithms *SH, TH, THwD* are presented in section IV. We compare the algorithms using extensive experiment studies in section V. The paper is concluded in section VI.

II. Related Work

There is a large body of related work on nearest neighbor (NN) queries on static spatial objects, e.g. [15], [16], [10]. With the increasing popularity of location sensory devices, NN query on moving objects [23], [17], [4], [18], [19], [22], [21], [14] has become an important research topic. Work in [23], [17], [18] addresses NN queries from moving objects on static objects. Algorithms are designed to provide information about the validity of the result in order to reduce computational and communication costs. In [18], given a moving object, the NN results up to a future timestamp are returned in the format of: $< R_i, T_i >$, where R_i is the set of NNs during future interval T_i. Other methods assume that the query and the data objects move linearly with known velocities [19], [4] and utilize dynamic disk-based indexes on moving objects (e.g. TPR-tree [20]) to process and optimize the queries.

Our work is related to similarity search in time series data. Generally, dimension reduction methods [3], [11], [7] and index structures [3] are used to reduce IO costs for efficient time series retrieval and similarity search. The methods may be adapted for retrieving INN from historical trajectories residing on disk. However, the overhead of indexing and dimension reduction is likely to eclipse the savings for in-memory processing of INNs over sliding window in real time. Our work selectively retains trajectories necessary for answering CINN queries in memory to speedup query responses.

In a stream data management system, e.g. [6], [2], [1], data arrives in the form of concurrent and continuous data streams. Queries on data streams are typically continuous monitoring queries, and emitting real time streaming data as results. Moving object databases handle streaming location updates, support queries on them [8], and can be seen as geo-stream databases. We review the NN query processing work in this area, focusing more on hashing based algorithms due to their close relevance to our work.

Several work [22], [21], [14] achieves low running time by handling location updates only from objects that fall in the vicinity of some queries. A grid-based data structure is proposed in [22] to index the current locations of moving objects and support NN queries. The first NN is identified through an iterative search around the query point using the grid. Subsequent maintenance is performed by only searching the region centered at the query point, with a radius of the distance from the query point to the previous NN. SEA-CNN [21] also uses grid to index the current locations. In addition, each cell stores information about whether it intersects with the answer region of a query q. To incrementally maintain the NN of q, a small search region SR around q is constructed considering the movements of the query and the objects in and out of the answer region. CPM [14] applies sophisticated conceptual partitioning of the space, using rectangles that indicate their proximity to a query q to facilitate NN search.

Continuous monitoring of NN queries over sliding window is investigated in [13]. The sliding window is used to prune NN events far away in the past. Two algorithms were proposed: conceptual partitioning and skyline maintenance in the distance-time space. The definition of the NN in this work is what we term as *instant-based* metric, and does not include an interval component as defined in this paper.

Finally, spatio-temporal pattern (STP) queries introduced in [9] specify an STP as a sequence of distinct spatial predicates, where the predicate temporal ordering matters. The STP queries that contain only NN predicates are defined as finding the trajectory that minimizes the sum of the distances from a sequence of given points. This definition is similar to the definition of INN in this paper. However, sliding-window-based queries and real time INN queries are not addressed.

III. Preliminaries and Problem Definition

We present the location update models, the key terms, and the problem definition formally in this section.

A moving object may report its location over time in two ways: constant location update (CLU) and movement-based location update (MLU). In the CLU model, a moving object reports its location at every time unit. In the MLU model, a moving object only report its location when it moves noticeably. If an object does not report its location, the server assumes that the object's location is the same as its last reported one. The MLU model saves communication cost when many objects are (nearly) static. This paper is focused on MLU model. However, the approach can also be applied to CLU model with minimal modifications.

Definition 1 (Trajectory): A trajectory P is a sequence of time stamped locations. We use $P[t]$ to represent the location of P at time t. If P is not defined at time t, $P[t] = \perp$.

In Table I, there are four trajectories with each represented by a sequence of stamped locations in the format of $< (x, y), t >$, where x and y are two-dimensional coordinates, and t is the timestamp.

Definition 2 (Streaming Trajectory): A streaming trajectory Q_t^w is the sub-trajectory of a trajectory Q over sliding window w that ends at time t, where t is typically the current time or a constant shift from the current time. As a result, the window w moves without explicit updates and so does Q_t^w .

Let the current time be 3, and the sliding window size be 3, i.e. $t = 3$ and $w = 3$, the streaming trajectory Q_t^w can be found in Table I. We have $Q_t^w = (< (2,1), 1 >, < (0,3), 2 >, < (3,3), 3 >)$. Subsequently, Q_t^w becomes $(< (0,3), 2 >, < (3,3), 3 >, < (4,3), 4 >)$, and $(< (3,3), 3 >, < (4,3), 4 >, < (5,3), 5 >)$ respectively.

Definition 3 (Distance): Given a streaming trajectory Q_t^w, the distance $dist(Q_t^w, P)$ from Q_t^w to another trajectory P is defined as $dist(Q_t^w, P) = \sum_{i \in (t-w,t]} dist(Q[i], P[i])$, where $dist(Q[i], P[i])$ is the distance between two point locations $Q[i]$ and $P[i]$. And $dist(Q[t], \perp) = \infty$. Note that although

TABLE I
ILLUSTRATION OF TRAJECTORIES, DISTANCES, AND INN

Trj. ID	Trajectory $< (x, y), t >$
Q	$(< (2,3), 0 >, < (2,1), 1 >, < (0,3), 2 >, < (3,3), 3 >, < (4,3), 4 >, < (5,3), 5 >)$
P_1	$(< (1,4), 0 >, < (2,5), 1 >, < (0,6), 2 >, < (3,7), 3 >, < (4,8), 4 >, < (5,9), 5 >)$
P_2	$(< (5,3), 0 >, < (6,2), 1 >, < (7,3), 2 >, < (8,3), 3 >, < (9,4), 4 >, < (10,3), 5 >)$
P_3	$(< (2,2), 0 >, < (2,2), 1 >, < (2,3), 2 >, < (2,4), 3 >, < (3,3), 4 >, < (4,3), 5 >)$
P_4	$(< (0,1), 0 >, < (1,1), 1 >, < (1,1), 2 >, < (1,1), 3 >, < (1,1), 4 >, < (1,2), 5 >)$

$t = 3,$ $w = 3$

$dist(Q_t^w, P_1) = 4 + 3 + 4 = 11$
$dist(Q_t^w, P_2) = \sqrt{17} + 7 + 5 = 16.12$
$dist(Q_t^w, P_3) = 1 + 2 + \sqrt{2} = 4.41$
$dist(Q_t^w, P_4) = 1 + \sqrt{5} + \sqrt{8} = 6.06$
CINN of Q_t^w at time 3 is P_3

we may handle the undefined points using its history or future information, it requires a formal statistical model, which is out of scope of this paper.

In Table I, when $t = 3, w = 3$, $dist(Q_t^w, P_1) = \sum_{i \in (0,3]} dist(Q[i], P_1[i]) = 4 + 3 + 4 = 11$. Starting with $t = 3, w = 3$, the distance from Q_t^w to P_1 is 12 at time 4, and 15 at time 5.

Definition 4 (CINN): The continuous interval-based nearest neighbor (CINN) trajectory of Q_t^w is the trajectory P whose distance from Q_t^w is the minimal among all the trajectories different from Q at t. Both t and w move over time.

Example 1: In Table I, at time $t = 3$ with $w = 3$, $dist(Q_t^w, P_1) = 11$, $dist(Q_t^w, P_2) = 16.12$, $dist(Q_t^w, P_3) = 4.41$. Therefore the CINN of Q_t^w at $t = 3$ is trajectory P_3. Starting with $t = 3, w = 3$, the CINN of Q_t^w is still P_3 at time 4 and time 5.

The problem of processing and optimizing CINN query is to *efficiently identify the CINNs of multiple streaming trajectories from moving objects in real time, with the goal of minimizing both computational cost and memory space*.

IV. CINN SEARCH OVER SLIDING WINDOW

We present algorithms for monitoring the streaming trajectories in real time to answer CINN queries. The challenge is to identify continuously the INN of multiple streaming trajectories with one scan of the incoming trajectories under the constraint of limited memory. We focus on the more realistic movement-based location update (MLU) model. To better illustrate the ideas of our proposed approaches, we will use the example in Table I throughout the paper to explain the details of each algorithm.

A. Brute Force Algorithm (BF)

If we are given unlimited memory, the *brute-forth algorithm* (*BF*) can be implemented as follows: We store each moving object's trajectories in a circular queue of size w (maximum window size). At any time stamp, if the moving object associated with P has a location update, the queue is updated with its reported location. Otherwise its last reported location is used to update the queue. The update operation always overwrite the earliest value in the queue. We then compute Q_t^w's CINN at time t by computing the distance $dist(Q_t^w, P)$ for each P and choose the P that minimizes the distance.

However, in the *BF* approach, the computational cost is high and is at least $O(n \cdot m \cdot w)$, where n is the total number of trajectories, m is the total number of CINN queries, and w is the maximum window size. Furthermore, the memory requirement for this simple algorithm is prohibitive and is at least $O(n \cdot w)$. When using incremental distance computation (as discussed shortly), additional memory space of $O(m \cdot n)$ is required to store the historical distances.

The *BF* algorithm is not computationally efficient because: 1) It is not necessary to update location queue every time when the report frequency is low. 2) Distances over window w can be incrementally calculated if we store the historical distance. 3) For any given object, only a small subset of the trajectories can be the candidates of its CINN. Therefore a full scan of all the trajectories is unnecessary. The *BF* algorithm is also memory prohibitive because the location information associated with every trajectory is stored in the system, although only a subset of them is useful to answer the queries.

Based on above observations, we first propose the *lazy location update* and *incremental distance computation* mechanisms. Utilizing these two mechanisms, the *BF* algorithm's computational cost can be substantially reduced. Then we explore spatial and temporal hashing methods to further reduce computational cost by scanning a subset of the trajectories at any time. Finally we will explore the speed limits of the moving objects to substantially reduce memory space.

B. Lazy Update and Incremental Distance Computation

Our basic data structure to handle location update is a circular queue of size w for each trajectory. For any trajectory P, only sliding window of size w that ends at time t is meaningful to queries issued by trajectory Q_t^w. For multiple queries, the size of the circular queue is the maximum window size of all the queries so that all queries can share the same location information. Since the query window size can change dynamically over time, we allow circular queues to expand or shrink according to w. This can be performed efficiently during the location update stage by passing the increment/decrement requests to the queue structure.

Intuitively, at any given time t, the circular queue of P has the content of $(< P[t - w + 1] >, < p[t - w + 2] >, ..., < p[t] >)$ after location update. For the objects that do not report their locations, it would be unnecessary to update their location queues. Therefore we propose the following *lazy location update* approach to update the location queue.

We only update P's location queue when there is a location update from P. Let $P.lastUpdate$ represent the last update time of P. At time t, we receive a new location update $P[t]$ from P. We first update next $min(t - P.lastUpdate - 1, w - 1)$

slots of the queue using $P[P.lastUpdate]$. Then we update next slot of the queue using $P[t]$, and set $P.lastUpdate$ to t. This will avoid unnecessary multiple location updates with the same value. The extra cost is an additional memory chunk to store each trajectory's last update time.

Example 2: In Table I, after time $t = 1$, $P_4.lastUpdate = 1$ and the location queue for P_4 is $(\perp, < 0, 1 >, < 1, 1 >)$. At $t = 2, 3$, or 4, there is no update from P_4. At time $t = 5$, we first update the next $min(5 - 1 - 1, 3 - 1) = 2$ slots of the queue using value $P_4[1] =< 1, 1 >$. Then we update the next slot of the queue with the value $P_4[5] =< 1, 2 >$. Now the queue will become $(< 1, 1 >, < 1, 1 >, < 1, 2 >)$ at $t = 5$ and $P_4.lastUpdate = 5$.

Note that using this approach, at time t after location update, if $P.lastUpdate \neq t$, the value of $P[P.lastUpdate]$ stored in the current slot of the queue will be used $min(t - P.lastUpdate + 1, w)$ times in distance calculation. An optimization can be performed to calculate the distance incrementally: The $dist(Q_t^w, P)$ can be computed by $dist(Q_t^w, P) = dist(Q_{t-1}^w, P) + dist(Q[t], P[t]) - dist(Q[t - w], P[t - w])$. We know that $P[t] = P[P.lastUpdate]$. Therefore we only need to get the value of $P[t - w]$ (To avoid $P[t - w]$ being overwritten during update, we store the value of last slot to be replaced in each update).

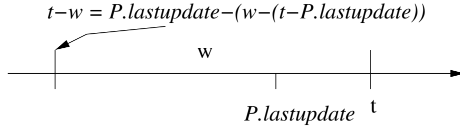


Fig. 1. Incremental Distance Calculation in MLU

$P[t-w]$ is $max(0, w-(t-P.lastupdate))$ back from the current location slot as shown in Figure 1. The formula considers the situation that the location queue is not updated for $t-P.lastupdate$ time and $P[t-w]$ is not exactly w back from the current location slot. However, when $t = P.lastupdate$, we need to go back w slot which returns to the current slot in the circular queue of size w. In this case, the $P[t - w]$ would have been stored by the update process. Thus we need to use the stored value when $t = P.lastudpate$.

Example 3: In Table I, at time $t = 4$, the content of P_4's queue is $(\perp, < 0, 1 >, < 1, 1 >)$. The distance from Q to P_4 is $dist(Q_4^3, P_4) = dist(Q_3^3, P_4) + dist(Q[4], P_4[4]) - dist(Q[1], P_4[1]) = 6.06 + \sqrt{13} - 1 = 8.67$, where $P_4[4]$ can be obtained from the current slot of the queue and $P_4[1]$ is the value in the $max(0, 3-(4-1)) = 0$ slots back of the current slot of the queue.

The lazy location update will be used in the spatial and temporal algorithms to be described. The incremental distance computation will be adapted as well.

C. Spatial Hashing Algorithm (SH)

We now use a spatial hashing method [22], [21], [14] to avoid the brute-force computation to answer CINN queries. We will explore a temporal hashing method in the next section.

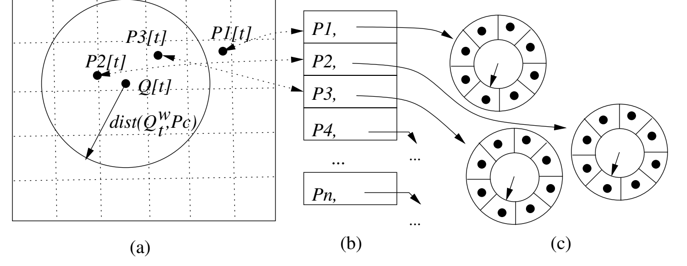


Fig. 2. Spatial Hashing

As shown in Figure 2 (a)[1], we maintain a grid-based spatial hashing for the current locations of the moving objects similar as the methods for processing *time-instant-based* continuous NN queries [22], [21], [14]. The location queue of each trajectory is shown in Figure 2 (c). The *SH* algorithm has the following main steps:

- At each time instant t, when processing a location update $P[t]$, P's location is also updated in the hashing grid.
- Let P_c be the INN of Q_{t-1}^w, then the *influence region* of $Q[t]$ at t is defined as $dist(Q_t^w, P_c)$.
- After processing of location update, we examine the *influence region* of a query Q_t^w at time t. We use the location hashing grid to locate all the trajectories in the influence region of the $Q[t]$.
- The INN of $Q[t]$ is updated to be the trajectory with the minimal distance from $Q[t]$ in $Q[t]$'s influence region.

Example 4: In Table I, at time $t = 4$, we first update the location of P_1, P_2, P_3 and P_4 in the spatial hash table. Since at $t = 3$, the INN of Q is $P_c = P_3$. We compute the radius of the influence region at $t = 4$ as $dist(Q_4^3, P_3) = 4.41$. Since $Q[4] =< 4, 3 >$, through the spatial hashing grid, we find that neither P_1's location $< 4, 8 >$ nor P_2's location $< 9, 4 >$ at time 4 is inside the influence region. But P_4 is within $Q[4]$'s influence region, and $dist(Q_4^3, P_4) = 8.67 > 4.41$. Therefore P_3 is still the CINN at $t = 4$.

Lemma 1 (Influence Region): Given a query Q_t^w, and P_c as Q's INN at time $t-1$, then the INN of Q_t^w at time t is within a circular region centered at $Q[t]$ with a radius of $dist(Q_t^w, P_c)$.

Proof: We prove by contradiction. Assume that the current location of the INN of Q_t^w at time t, denoted by $Q_t^w.INN$, is more than $dist(Q_t^w, P_c)$ away from $Q[t]$.

$$\begin{aligned} dist(Q_t^w, Q_t^w.INN) = &\sum_{i \in (t-w,t]} dist(Q[i], Q_t^w.INN[i]) \\ &\geq dist(Q[t], Q_t^w.INN[t]) \end{aligned}$$

Since $dist(Q[t], Q_t^w.INN[t]) > dist(Q_t^w, P_c)$, we will have $dist(Q_t^w, Q_t^w.INN) > dist(Q_t^w, P_c)$. This means P_c is closer to Q_t^w than the $Q_t^w.INN$, which is a contradiction. ■

We now discuss how to adapt the incremental nearest neighbor calculation as that used in the *BF* algorithm. Please note that the historical distances may not be available for some trajectories in the *influence region* of Q_t^w if these trajectories

[1]For better visualization, the illustration figures in this paper do not have same parameter settings as the running examples.

were not in the influence region of the previous time stamp. This will bring a problem to incremental distance computation. To deal with this problem, we record the historical distance of each trajectory and its associated timestamps whenever a trajectory is inside the *influence region*. Then incremental distance calculation is used whenever possible.

The *SH* algorithm can save computational cost by using influence regions. The spatial hashing grid requires additional memory which is negligible compared with memory needed for location queues and historical distances. Our next goal is to exploit the physical limitations of moving objects in terms of traveling speeds to devise a temporal hashing algorithm that uses similar memory but is computationally efficient.

D. Temporal Hashing Algorithm (TH)

We design a method to hash the trajectories into future time slots where they may start to influence the result of the query. With temporal hashing, we only need to process trajectories in one time slot (the current time slot) at a time. The base rationale is that some trajectories are too far away from the query trajectory and will not be the INN for some time given the traveling speed limitations of "non-magical" moving objects.

Let v denote the upper bound of the relative speed of any two moving objects in the application domain. We will now use v to devise our temporal hashing algorithm. In practice, the moving objects in CINN queries are often homogeneous (e.g. cars or pedestrians). Therefore the value of v will not vary significantly between different objects. Even if the objects are heterogeneous, we can still use different levels of speed constraints to prune the trajectories according to the type of the objects. When the trajectory of a moving object can be described by a function for some time interval, our algorithm can be modified accordingly to accommodate.

Definition 5 (Dormant Time): Let P_c denote a candidate interval-based nearest neighbor of Q_t^w. Let P denote another trajectory. If P will not be closer to $Q_{t+\delta t}^w$ than P_c for δt time instants starting from t, we call δt the dormant time $DT(P, P_c, Q_t^w)$ of P with respect to P_c for query Q_t^w.

The dormant time allows us to put a trajectory in a future time slot and ignore it during the dormant time until they may start to be the INN of a query. It can assume many values. However, the larger the value is, the more savings we can get from hashing the trajectory to the time slot further into the future. We now look at how to find a large dormant time.

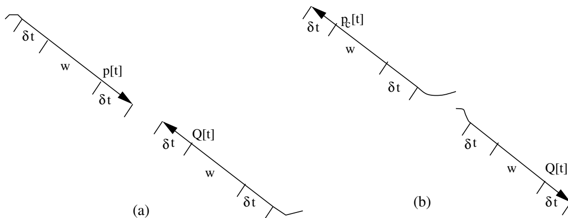


Fig. 3. When $\delta \in (0, w)$ (a) Lower Bound of $dist(Q_{t+\delta t}^w, P)$ (b) Upper Bound of $dist(Q_{t+\delta t}^w, P_c)$

We first consider when $\delta t \in (0, w)$. At time t, if we know the distance from Q_t^w to P and in the subsequent time instants $t + \delta t$, $dist(Q_{t+\delta t}^w, P)$ can be calculated by adding the new distances from t to $t+\delta t$ and subtracting the old distances from $t - w$ to $t - w + \delta t$. Because of the maximal relative speed v of Q and P, at time $t+i$ for $i \in [1, \delta t]$, $dist(Q[t+i], P[t+i])$ can not be less than $\max(0, dist(Q[t], P[t]) - i \cdot v)$. The lower bound of the distance is achieved when P and Q head to each other from t to $t + i$ until they meet. Furthermore, we know $\max(0, dist(Q[t], P[t]) - i \cdot v) \geq dist(Q[t], P[t]) - i \cdot v$.

Similarly, at time $t - w + i$ for $i \in [1, \delta t]$, $dist(Q[t - w + i], P[t-w+i])$ can not be more than $dist(Q[t], P[t]) + (w-i) \cdot v$. This upper bound is achieved when P and Q head directly to each other from $t - w + i$ to t until they come to their positions $P[t]$ and $Q[t]$ at time t. Combining the pieces, we have:

$$dist(Q_{t+\delta t}^w, P) \geq dist(Q_t^w, P) + \sum_{i \in [1, \delta t]} (dist(Q[t], P[t]) - i \cdot v) - \sum_{i \in [1, \delta t]} (dist(Q[t], P[t]) + (w - i)v) = dist(Q_t^w, P) - w\delta t \cdot v \quad (1)$$

Figure 3 (a) illustrates when the lower bound of $dist(Q_{t+\delta t}^w, P)$ is achieved. Using similar arguments of speed limits, we can find the upper bound of $dist(Q_{t+\delta t}^w, P_c)$ with the situation shown in Figure 3 (b).

$$dist(Q_{t+\delta t}^w, P_c) \leq dist(Q_t^w, P_c) + \sum_{i \in [1, \delta t]} (dist(Q[t], P[t]) + iv) - \sum_{i \in [1, \delta t]} (dist(Q[t], P[t]) - (w - i)v) = dist(Q_t^w, P_c) + w \cdot \delta t \cdot v \quad (2)$$

From inequalities 1 and 2, we can deduce that when:

$$\delta t \leq \frac{dist(Q_t^w, P) - dist(Q_t^w, P_c)}{2w \cdot v} \quad (3)$$

We have $dist(Q_{t+\delta t}^w, P) \geq dist(Q_{t+\delta t}^w, P_c)$. We choose $DT(P, P_c, Q_t^w) = \min(w - 1, \frac{dist(Q_t^w, P) - dist(Q_t^w, P_c)}{2w \cdot v})$ to be the dormant time for now. When $DT(P, P_c, Q_t^w) = w - 1$, we will continue to the following condition test where $\delta t \in [w, \infty)$ to see if we can extend the dormant time further. Otherwise, the dormant time obtained will be final. This is because when $DT(P, P_c, Q_t^w) < w - 1$ we do not have sufficient evidence that when $\delta t \in (t + DT(P, P_c, Q_t^w, w), t + w)$, P won't be the INN of $Q_{t+\delta t}^w$. So we will have to hash P to $t + DT(P, P_c, Q_t^w)$.

Now we consider when $\delta t \in [w, \infty)$. Given $P[t]$ and $Q[t]$, if P and Q head directly towards each other after time t until time $t + \delta t$, then $dist(Q_{t+\delta t}^w, P)$ will be minimal as shown in Figure 4 (a). At time $t + \delta t - w$, the lower bound of $dist(Q_{t+\delta t-w}^w, P)$ is $dist(Q[t], P[t]) - (\delta t - w) \cdot v$. In the subsequent i time instant until $t + \delta t$ which consists of the current sliding window w from $t + \delta t$, the lower bound of the distance between $Q[t + (\delta t - w) + i]$ and $P[t + (\delta t - w) + i]$

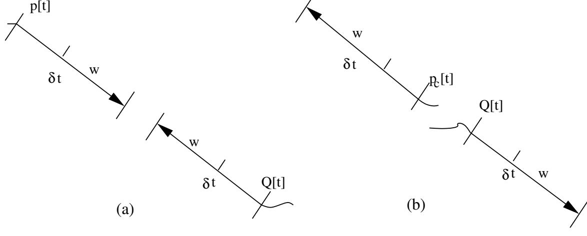


Fig. 4. When $\delta \in [w, \infty)$ (a) Lower Bound of $dist(Q^w_{t+\delta t}, P)$ (b) Upper Bound of $dist(Q^w_{t+\delta t}, P_c)$

is $dist(Q[t], P[t]) - (\delta t - w) \cdot v - i \cdot v$. So we have:

$$dist(Q^w_{t+\delta t}, P) \geq \sum_{i \in (0,w]} (dist(Q[t], P[t]) - (\delta t - w)v - iv)$$
$$= w(dist(Q[t], P[t]) - (\delta t - w)v) - \frac{w(w+1)}{2}v \quad (4)$$

On the other hand, if P_c and Q head directly opposite against each other after time t until $t + \delta t$ as shown in Figure 4 (b), then $dist(Q^w_{t+\delta t}, P_c)$ will be maximal. So, we have:

$$dist(Q^w_{t+\delta t}, P_c) \leq \sum_{i \in (0,w]} (dist(Q[t], P_c[t]) + (\delta t - w)v + iv)$$
$$= w(dist(Q[t], P_c[t]) + (\delta t - w)v) + \frac{w(w+1)}{2}v \quad (5)$$

Combining the given condition of $\delta t \in [w, \infty)$ and inequalities 4 and 5, we can deduce that when:

$$\delta t \leq \frac{dist(Q[t], P[t]) - dist(Q[t], P_c[t])}{2v} - \frac{w-1}{2}$$

We have:

$$dist(Q^w_{t+\delta t}, P) \geq dist(Q^w_{t+\delta t}, P_c) \quad (6)$$

So, if $w \leq \frac{dist(Q[t],P[t]) - dist(Q[t],P_c[t])}{2v} - \frac{w-1}{2}$ which is equivalent to $w \leq \frac{dist(Q[t],P[t]) - dist(Q[t],P_c[t]}{3v} + \frac{1}{3}$, the dormant time of P will be $DT(P, P_c, Q^w_t) = \frac{dist(Q[t],P[t]) - dist(Q[t],P_c[t])}{2v} - \frac{w-1}{2}$. Otherwise, it will keep the original value which is $w - 1$.

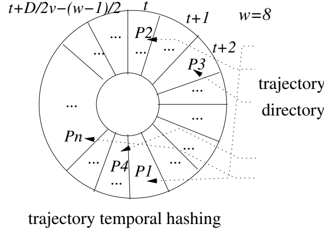


Fig. 5. Temporal Hashing

We can hash a trajectory P to a future time slot $DT(P, P_c, Q^w_t)$ away from t given an existing INN P_c because we know Q will not be closer to P than P_c any time earlier.

Now the temporal hashing algorithm works as follows and is formally given in Algorithm 1:

- At time t, we first process location update as before. Then we retrieve the trajectories in time slot t in the data structure.
- The trajectory ID is hashed to the time slot $DT(P, P_c, Q^w_t)$ away from the current slot t, where P_c is the INN from the previous time snapshot.
- At the same time, we check if any of them are the INN of Q^w_t and update the INN of Q^w_t if needed.

Example 5: In Table I, at $t = 3$, we first process location update for all the trajectories. We know that $P_c = P_3$, and $dist(Q^w_t, P_c) = 4.41$. Assuming $v = 1$, given that $min(w-1, \frac{dist(Q^w_t, P_1) - dist(Q^w_t, P_c)}{2w \cdot v}) = min(3-1, \frac{11-4.41}{2*3*1}) = 1.1$, therefore $DT(P_1, P_c, Q^3_3)$=1.1. P_1 can be hashed to the time slot $3+2 = 5$. Similarly, P_2 and P_4 can be hashed to time 5 and time 4 respectively. Now assume $v = 0.5$, for P_1 we will have $min(w-1, \frac{dist(Q^w_t, P_1) - dist(Q^w_t, P_c)}{2w \cdot v}) = w - 1 = 2$. Then we find that $\frac{dist(Q[t], P_1[t]) - dist(Q[t], P_c[t]}{3v} + \frac{1}{3} = 4.7 > w = 3$, we have $DT(P_1, P_c, Q^3_3)$=5.6. P_1 can be hashed to new time slot 9. Similarly P_2 and P_4 can be hashed to time 15 and 4 respectively.

For multiple queries, a trajectory will be hashed to the nearest dormant time of all the queries. The trajectories in the current time slot will need to be checked against all queries.

The maximum size of the temporal hashing table is $D/v + (w-1)/2$ (the maximal dormant time) where D is the longest distance between any two points in the space and v is the upper bound of the speed between any two objects.

Algorithm 1 shows the details of the TH approach and ThwD approach (as discussed shortly). Lines 1-2 initialize the data structure. Lines 2-18 process the location update. Lines 19-24 compute the first CINN for each trajectory. Then lines 25-37 update the CINN and compute the hashing value for each trajectory based on the formula (3) and (6). Lines 38-40 map each trajectory to the temporal hashing table. Subsequently lines 41-53 process the CINN query similarly for future time stamps. As in *SH* algorithm, the incremental distance calculation is used whenever possible.

The computational cost of the *TH* algorithm depends on how far we can hash the trajectories into the future. Compared with the *SH* algorithm, the *TH* algorithm considers the speed constraints of the moving objects, and is more likely to achieve a tighter bound to filter out faraway trajectories. On the other hand, similar constraints can not be easily applied to the *SH* algorithm. The memory consumption of the *TH* algorithm however, is still more than that in *BF* algorithm because of the additional temporal hashing table.

E. Temporal Hashing with Dropping Algorithm (THwD)

A deeper look at the above algorithms indicates that the memory consumption mainly consists of memory required for historical distances and location circular queues, which has the size of $O(n \cdot m)$ and $O(n \cdot w)$ respectively. The spatial hashing grid and temporal hashing table has a fixed size, and pointers to ids of trajectories that consume relatively less space.

Algorithm 1 Temporal Hashing Algorithm (*TH*) and Temporal Hashing with Dropping Algorithm (*THwD*)

1: Initialize the temporal table TT, location queue LQ, history distance H
2: Initialize flag $flag = 1$ for TH, and $flag = 2$ for $THwD$
3: **for** Each incoming location $< (x, y), t >$ that belongs to trajectory $P(i)$ **do**
4:   **if** $flag == 2$ **then**
5:     Get Dormant Time $DP(i)$
6:     **if** $DP(i) - t < w$ **then**
7:       **if** $q(i) == null$ **then**
8:         Initialize $q(i)$
9:       **else**
10:         $q(i)$.update((x,y))
11:       **end if**
12:     **else**
13:       store $P[t]$
14:     **end if**
15:   **else**
16:     $q(i)$.update((x,y))
17:   **end if**
18: **end for**
19: **if** $t <= w - 1$ **then**
20:   **for** Each query $Q(j)$ **do**
21:     **for** Each trajectory $P(i)$ **do**
22:       compute $MinDist(j)$, update history distance H and $CINN(j)$
23:     **end for**
24:   **end for**
25: **else if** $t == w$ **then**
26:   **for** Each query $Q(j)$ **do**
27:     **for** Each trajectory $P(i)$ **do**
28:       compute $MinDist(j)$, update history distance H and $CINN(j)$
29:       compute hash value δt using Formula (3)
30:       **if** $\delta t == w - 1$ **then**
31:         compute hash value δt using Formula (6)
32:         **if** ($flag == 2$ and $\delta t \geq w$) **then**
33:           Drop location queue and the historical distances of $P(i)$.
34:         **end if**
35:       **end if**
36:     **end for**
37:   **end for**
38:   **for** Each trajectory $P(i)$ **do**
39:     $TT.map(i, min((t + \delta t))$
40:   **end for**
41: **else**
42:   **for** Each query $Q(j)$ **do**
43:     **for** Each trajectory $P(k)$ in $TT(t)$ **do**
44:       compute $MinDist(j)$, update history distance H and $CINN(j)$
45:       compute hash value δt according to Formula (3) and (6) similarly
46:       **if** ($flag == 2$ and $\delta t \geq w$) **then**
47:         Drop location queue and the historical distances of $P(k)$.
48:       **end if**
49:     **end for**
50:   **end for**
51:   **for** Each trajectory $P(k)$ in $TT(t)$ **do**
52:     $TT.map(k, min(t + \delta t))$
53:   **end for**
54: **end if**

To reduce the memory used for historical distances, we may drop the historical distances of trajectories that are not used in finding the CINNs. Similarly, to reduce the memory used for location circular queues, we may remove the queues of trajectories that are not useful in finding CINNs. Now we investigate a way to identify these unnecessary trajectories.

Lemma 2: At time t, if $DT(P, P_c, Q_t^w) \geq w$, we can drop location circular queue of P and historical distances associated with P without storing them in memory, given $P[P.lastUpdate]$ is already in memory. This condition is called the dropping condition for P with respect to Q_t^w.

Proof: The proof is based on the fact that $P[t]$ is only needed for w time starting from t before it permanently expires (not participate in any window w query). ∎

However, if for each location update $P[t]$ from P, we check whether the dropping condition is met with respect to P_c for Q_t^w in order to decide whether to drop the streaming location circular queue of P or not, the cost will be more than that in the *BF* algorithm. This is because to calculate the dormant time, we need to calculate $dist(Q_t^w, P)$ for each P.

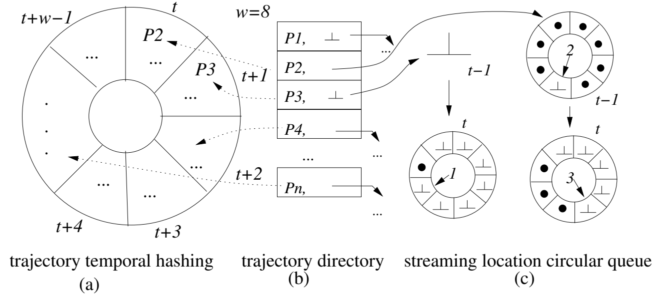


Fig. 6. Temporal Hashing with Dropping

Instead, we perform dropping during the rehashing process. We associate the storage of historical distances with the location queue of each trajectory P. If P is not hashed at current time slot, we do not need to update the queue or historical distances. Otherwise we update $P[t]$ in the queue (and create a new queue if the queue does not exist). Meantime we also update/create the historical distances from P to Q. Note however, we still need to store P's last report location in case P does not report its location again for a long time. The dormant time re-calculation and re-hashing are only performed for the trajectories in the current time slot.

The temporal hashing with dropping algorithm *THwD* can proceed as follows:

- For each location update $P[t]$, if P is hashed more than $w - 1$ away from t, then store $P[t]$ without further processing. Otherwise, proceed to the next step.
- If P already has a circular queue, store $P[t]$ to the next slot. Otherwise, create a new queue with the first slot storing the value of $P[t]$ and leave other slots undefined. Meanwhile update P's accumulated historical distance.
- Calculate the CINN of Q_t^w from all the trajectories in the current time slot t as shown in Figure 6 (a). At the same time, re-calculate the dormant time of P with respect to Q_t^w. Drop P 's queue and the historical distances associated with P if the dormant time is more than $w - 1$ away from t. According to dropping condition, a trajectory P is guaranteed to have all the values for the window w from now for distance calculation and re-hashing. Update P_c with the new CINN.

Example 6: In Example 5, we already know that at $t = 3$, if $v = 0.5$, we find that $DT(P_1, P_c, Q_3^3) = 5.6 > w - 1 = 2$, therefore we hash P_1 to time 9. Meanwhile we can safely drop the location queue and the historical distances associated with P_1. The similar conditions also applies to P_2. Then at $t = 4$, during location update, we find that P_1's next slot (i.e. 9) is still more than $w - 1 = 2$ away from current time. We only store P_1's location without further processing. The same process continues until time $t = 7$, although P_1 is still not

hashed at this time, we will need to store the location queue and historical distances associated with P_1 because they will soon be useful. P_4 does not satisfy the dropping condition and will be processed as before.

Figure 6 (a) shows the temporal hashing of trajectories in the future. The temporal hash has w slots. Figure 6 (c) shows each trajectory is associated with a streaming location circular queue with some being empty (represented by $\perp$). In Figure 6, at time t, $P_1[t]$ meets the low cost dropping condition, thus is not in the temporal hash and its circular queue is deleted if existed. $P_2[t]$ and $P_3[t]$ do not meet the low cost dropping condition and are updated to the right time slots. P_2 has a circular queue at $t-1$ with $w-1$ elements. $P_2[t]$ is stored in the next slot which complete its all its locations in the past window w and is ready for distance calculation and re-hashing. P_3 does not have a streaming location circular queue at time $t-1$. Thus a new queue with one element is created as shown in Figure 6 (c). Finally, all the trajectories in slot t of the trajectory temporal hash which includes P_2 are checked and the new INN may be found and the P_c is updated if needed. At the same time, these trajectories are re-hashed. Then, we move to the next time slot.

Algorithm 1 shows the detail of THwD approach when $flag$ is set to 2. lines 4-14 determines whether a dropped queue need to be recreated when the trajectory will soon become the candidate. Lines 32-34 and 46-48 determines the dropping condition and remove the queue and historical distances associated with the trajectory.

Since we drop the unnecessary trajectories from the memory, the memory space of the *THwD* algorithm can always be substantially reduced. On the other hand, the computational cost of the *ThwD* algorithm is not substantially higher than that of the *TH* algorithm because allocating and free memory operations are relatively less expensive compared with distance computation performed at each time stamp.

V. Experimental Results

In this section, we present our experiment results to compare the memory consumption and computational cost of our proposed algorithms. We will also compare the percentage of trajectories pruned by *SH* and *TH*. The trajectory data used in our experiments is generated on a real road network using the *Network-based Generator of Moving Objects* developed by Thomas Brinkhoff [5]. The number of trajectories generated ranges from 10K to 100K. Each trajectory contains 500 time stamps, which results in 5M to 50M point locations in space.

The results to be shown are average results per time snapshot for movement-based update model. In the MLU model, we evaluate the computational costs and memory consumptions of the four algorithms with respect to: (1) the number of trajectories n, (2) the number of queries m, (3) the sliding window size w, (4) location report frequency f, and (5) the upper bound of relative speed v respectively. As a special case of MLU model, the constant location update model is evaluated for $f = 1$. The detailed settings for each parameter are listed in Table II. In all the experiments, we choose

TABLE II
Parameter Settings

	n	m	w	f	$v(mph)$
Effect of n	10k-100k	100-1k	10	0.3	100
Effect of m	100k	10-2k	10	0.3	100
Effect of w	100k	50	10-100	0.3	100
Effect of f	100k	1k	10	0.1-1	100
Effect of v	100k	1k	10	0.3	30-160

queries randomly from all the trajectories. The *lazy update* and *incremental distance computation* are applied to each approach (including the *BF* algorithm). The platform to execute the experiments is a dedicated dual-CPU Opteron system with 8G of RAM. It runs the latest Debian linux and JDK version 1.5.0. The code is all written in Java. Each experiment is executed three times and the results are averaged.

A. Effect of Number of Trajectories

We first evaluate the effect of number of trajectories. We set $n = 10K - 100K$, $w = 10$, $m = 1\% * n$, $f = 0.3$, and $v = 100\ mph$.

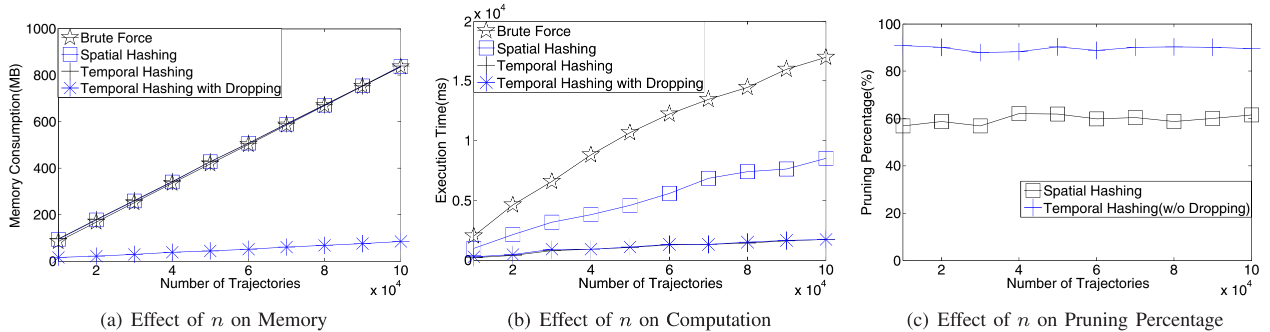
As shown from Figure 7(a), the memory consumptions are proportional to the number of trajectories for algorithms. Both the *SH*, *TH* algorithm have the similar memory consumption as the *BF* algorithm. However, the *THwD* can save nearly an order of magnitude of memory for 10k trajectories. On the other hand, Figure 7(b) shows that the *TH* and *THwD* algorithms are superior in terms of computational cost, which is again nearly more than one order of magnitude of saving compared with the *BF* algorithm, and nearly 5 times better than the *SH* algorithm. The pruning percentages in Figure 7(c) indicate that n does not have significant effect on pruning. The temporal hashing's pruning percentage is about 90% and much better than that of the *SH* algorithm(60%).

B. Effect of Number of Queries

Next we evaluate the impact of the number of CINN queries concurrently posted to the system. The numbers are reported when $N = 100K$, $w = 10$, $v = 100\ mph$, $f = 0.3$, and $m = 10 - 2K$. As demonstrated by Figure 8, the effect of m on memory consumption and computation cost of the *BF*, *SH* is similar to that of n. For *TH* and ThWD algorithm, the performance will generally degrade as the m increases because the dormant time is based on the smallest of all the queries. However, as 8 (a) and (b) shows, when $m = 2k$ (2% concurrent queries), the memory and computation savings are still close to an order of magnitude for the *THWD* algorithm, and the pruning percentage is slightly more than 90%, as compared with the *SH* algorithm's 60%.

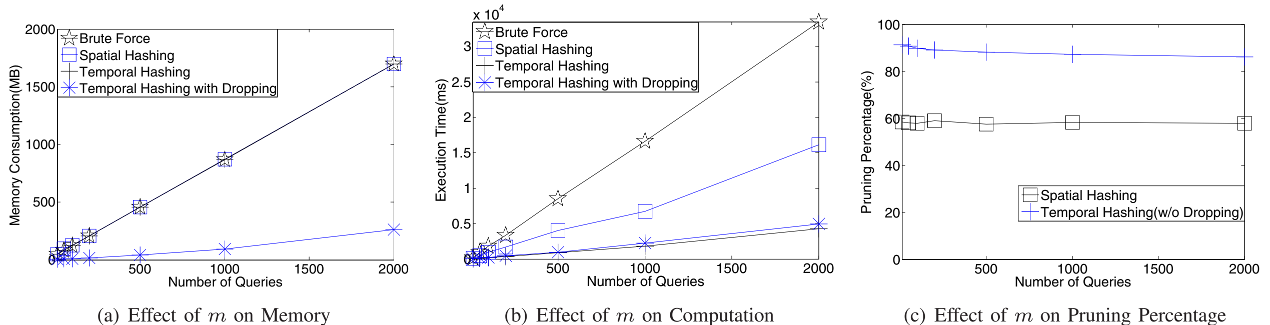
C. Effect of Maximum Window Size

We now evaluate the effect when sliding window size changes. We set $n = 100K$, $m = 50$, $v = 100\ mph$, $f = 0.3$ and $w = 10 - 100$. The effect of window sizes is shown in Figure 9. It is clear that although the performance of each algorithm degrades as w increases, the *THwD* algorithm is more scalable in both memory space and computation cost.



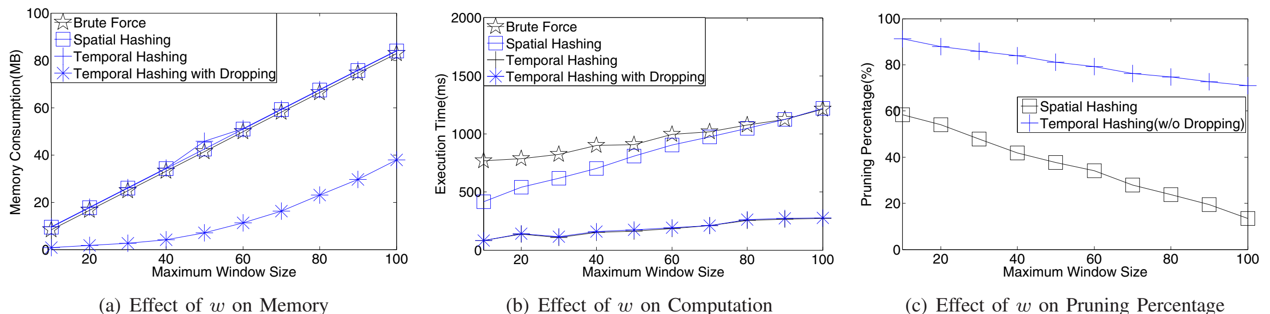
(a) Effect of n on Memory (b) Effect of n on Computation (c) Effect of n on Pruning Percentage

Fig. 7. Effect of Number of Trajectories (n = 10k-100k, $w = 10$, $m = 100 - 1$k, $f = 0.3$, $v = 100mph$)



(a) Effect of m on Memory (b) Effect of m on Computation (c) Effect of m on Pruning Percentage

Fig. 8. Effect of Number of Queries (n = 100k, $w = 50$, $m = 10 - 2$k, $f = 0.3$, $v = 100$ mph)



(a) Effect of w on Memory (b) Effect of w on Computation (c) Effect of w on Pruning Percentage

Fig. 9. Effect of Maximum Window Size (n = 100k, $w = 10 - 100$, $m = 50$, $f = 0.3$, $v = 100$ mph)

Note that in 9(b) and (c), when window size is large, the *SH* algorithm will perform nearly the same as the *BF* algorithm, and the pruning percentage is only about 10%, which means the pruning using spatial grid is not effective at all. However, the tempral hashing still has close to 70% pruning percentage.

D. Effect of Report Frequency

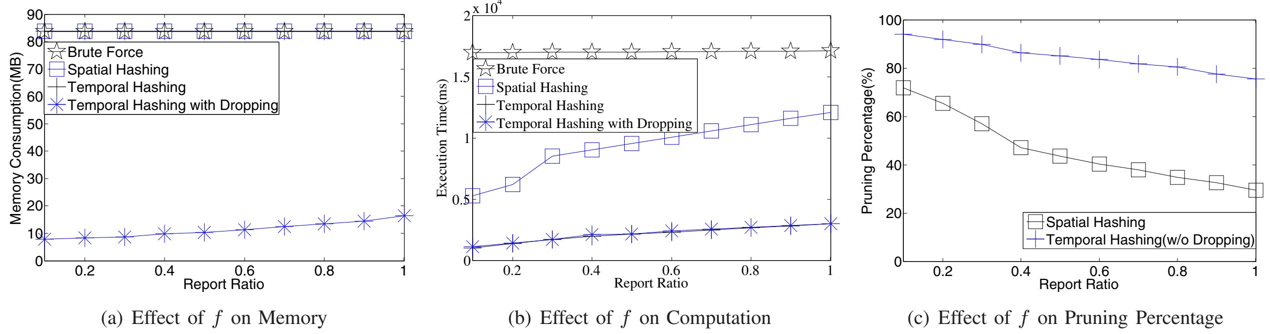
Next we evaluate the impact of the report frequency. We set $n = 100K$, $m = 1K$, $v = 100$ mph,$f = 0.1 - 1$ and $w = 10$. As shown by Figure 10(a), the report ratio has nearly no effect to the required memory space for the *BF*, *SH* and the *TH* algorithm. However, the space required for *THwD* increases as more location update comes. This is because when objects move more frequently, CINN is more likely to switch to different objects across many timestamps. Consequently less dropping can be performed. However, even under $f = 1$ (i.e. CLU model), the *THwD* algorithm can still save nearly 8 times of computation cost, and the pruning percentage is close to 80%, as demonstrated by Figure 10(b) and (c).

E. Effect of Maximum Speed

Finally we evaluate the effect of the upper bound of relative speed of moving objects. In our simulation, the default value of v is set as $v = 100$ mph. This is a realistic value considering that typical cars will not exceed this bound significantly in practice. Now we set n = 100k, m = 1k, $v = 30 - 160$, $f = 0.3$ and $w = 10$. Figure 11 shows that the effect of maximum speed is similar as the effect of report ratio. Even when realative speed has reached 160 mph, the computation saving is still nearly 7 times, and the pruning percentage is close to 80%.

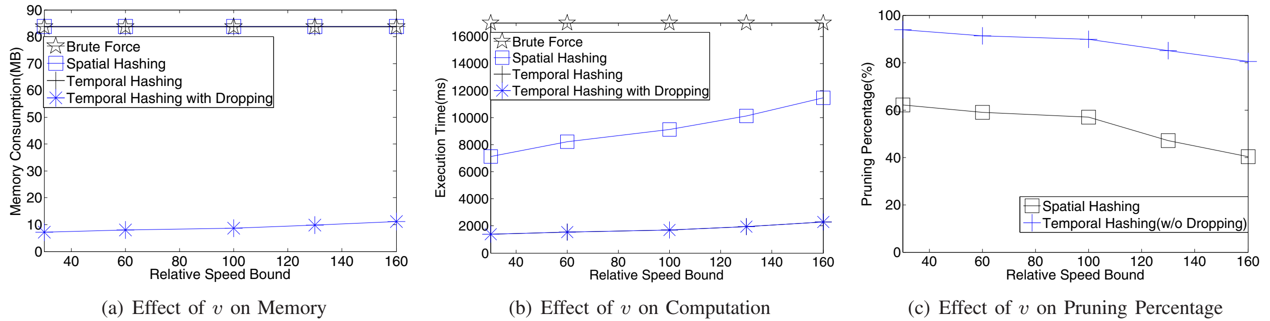
VI. Conclusion and Future Work

In this paper, we proposed a new type of nearest neighbor query called *interval-based* nearest query as opposed to *time-*



(a) Effect of f on Memory (b) Effect of f on Computation (c) Effect of f on Pruning Percentage

Fig. 10. Effect of Report Frequency ($n = 100$k, $w = 10$, $m = 1$k, $f = 0.1 - 1$, $v = 100\ mph$)



(a) Effect of v on Memory (b) Effect of v on Computation (c) Effect of v on Pruning Percentage

Fig. 11. Effect of Maximum Speed ($n = 100$k, $w = 10$, $m = 1$k, $f = 0.3$, $v = 30 - 160\ mph$)

instant-based NN queries. We define CINN which is INN in the context of stream processing with sliding windows. We proposed a spatial hashing, a temporal hashing, and a temporal hashing with dropping algorithm to reduce both computational cost and memory needed for computation.

In our future work, we plan to investigate the possibility of combining spatial hashing and temporal hashing approaches, as they exploit different aspects of the streaming trajectory. For multiple queries, the shared execution framework in [12] may also be utilized. We also plan to investigate CINN on static objects and historical trajectories. An example of a INN query on static objects is: Find the main water supply of a wild animal in the past month. We will also consider using moving prediction functions instead of speed constraints to achieve a better bound in temporal hashing.

REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), 2003.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: the stanford stream data manager (demonstration description). In *SIGMOD '03*, 2003.

[3] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT '08*, 2008.

[4] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J.*, 15(3), 2006.

[5] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2), 2002.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03*, 2003.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD '94*, 1994.

[8] R. H. Guting and M. Schneider. *Moving Objects Databases* . Morgan Kaufmann, 2005.

[9] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB '05*, 2005.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2), 1999.

[11] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 6(3), 2000.

[12] M. F. Mokbel and W. G. Aref. Sole: scalable on-line execution of continuous queries on spatio-temporal data streams. *The VLDB Journal*, 17(5):971–995, 2008.

[13] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6), 2007.

[14] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD '05*, 2005.

[15] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in r-trees. In *ICDE '97*, 1997.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD '95*, 1995.

[17] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, 2001.

[18] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD '02*, 2002.

[19] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM Trans. Database Syst.*, 28(2), 2003.

[20] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29(2), 2000.

[21] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE '05*, 2005.

[22] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE '05*, 2005.

[23] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD '03*, 2003.