Continuous Spatiotemporal Trajectory Joins

Petko Bakalov[1] and Vassilis J. Tsotras[1]

Computer Science Department, University of California, Riverside
{pbakalov,tsotras}@cs.ucr.edu

Abstract. Given the plethora of GPS and location-based services, que- ries over trajectories have recently received much attention. In this paper we examine trajectory joins over streaming spatiotemporal data. Given a stream of spatiotemporal trajectories created by monitored moving objects, the outcome of a *Continuous Spatiotemporal Trajectory Join* (CSTJ) query is the set of objects in the stream, which have shown similar behavior over a query-specified time interval, relative to the current timestamp. We propose a novel indexing scheme for streaming spatiotemporal data and develop algorithms for CSTJ evaluation, which utilize the proposed indexing scheme and effectively reduce the computation cost and I/O operations. Finally, we present a thorough experimental evaluation of the proposed indexing structure and algorithms.

1 Introduction

The abundance of position locators and GPS devices enables creation of data management systems that monitor streaming spatiotemporal data, providing multiple services to the end users. The basic architecture of a monitoring system working with spatial streaming data consists of multiple tracing devices which continuously report their location thus forming a spatiotemporal stream. Such streams are collected to the base station (server) where users submit their queries and continuously receive query results based on the current state of the data. Unlike traditional snapshot queries that are evaluated only once continuous queries require continuous revaluation as the query result becomes obsolete and invalid with the change of information for the objects.

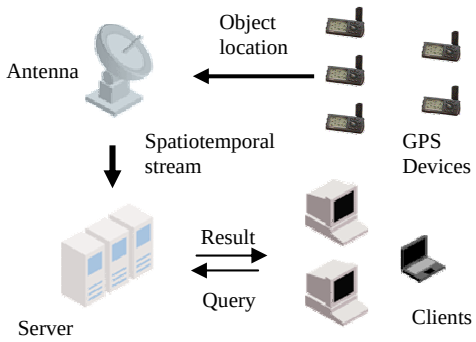

Fig. 1. A streaming spatiotemporal architecture.

Recent research efforts have focused mainly on simple queries (i.e., having just a single spatial Range [7][23] or Nearest Neighbor queries [33] predicate). However in many real life monitoring queries there is need for more complex continuous spatial predicates. For example, users may be interested in discovering pairs of moving objects which follow similar movement pattern for specified period of time. Consider a security system inside a building which is tracing person movement. The security officer may want to continuously check for any security violations or suspicious activities over the stream of spatiotemporal data coming from the sensors inside the building. One suspicious activity for example can be "Identify the pairs of (security officers,visitors) that have followed each other in the last 10 minutes." since it can be sign for someone trying to study the security personal.

In this paper we address a novel query for streaming data, called a *trajectory join*, i.e., the problem of identifying all pairs of similar trajectories between two datasets. The trajectory similarity is defined by their spatial closeness (expressed in a condition by a spatial threshold ϵ around each trajectory) which should last at least for an interval with duration δt. So the trajectory join can be expressed as a complex query predicate involving both the spatial and temporal constraints over the object trajectories. In the definition of the trajectory join problem for a static data set scenario [2][3] the temporal constraint δt is an absolute one (For example "Between 2 a.m and 3 a.m"). The absolute temporal constraints however do not make sense in a continuous environment since the result for them never changes. More useful for continuous queries are the relative time constraints. A relative time constraint uses the current time instance as a reference point. (Example: between 2 and 3 hours ago). As the time passes, the value of the current timestamp changes which makes the relative time constraint slide along the the temporal axis.

Because of the constant reevaluation of the result and the use of relative time constraints instead of absolute ones the extension of the static join solutions in the continuous environment is not efficient. A trivial extension of the existing static algorithms to continuous version can be repetitive execution of the static algorithm every time when the result has to be refreshed. However this is very expensive as the query evaluation starts from the beginning every time when the result has to be refreshed.

The prevailing strategy for efficient continuous query evaluation is the incremental approach [34, 33]. This approach implies that the query processor reuse as much as possible the current result and data structures for future iterations of the evaluation algorithm.

Nevertheless the CSTJ problem differs form all other continuous spatial predicates in that it also involves historical data from the stream.

In order to adopt efficiently the incremental approach for evaluation of the CSTJ queries we need an indexing structure for streaming data, which is able to:

- Answer queries about previous states of the spatiotemporal stream.
- Provide approximation for the object trajectory.
- Support the incremental approach for query evaluation.

To the best of our knowledge there is no indexing schema proposed, which has all these properties. In this paper we propose a novel indexing structure for spatial streams

which is able to store information about previous states of the spatiotemporal stream and develop algorithms for CSTJ evaluation, which utilize this indexing structure.

2 Related Work

Many join and self-join algorithms have been designed and proposed in the past for different data types and more specifically for spatial data [5] [19] [21] [27] [13] [31] [10] [35] [25] [18] [20] [1] [8]. However, these algorithms are not applicable in the case of spatio-temporal trajectories because they are based on intersections between MBRs while spatiotemporal trajectory join conditions are much more complex with constraints in both the spatial and temporal domain.

Recent work in the area of spatiotemporal streams has led to multiple indexing techniques and processing algorithms. They can be divided generally in three groups.

In [7] [23] [26] [24] the use of a simple grid structure is proposed for indexing the location of the objects inside the spatiotemporal stream. Every single grid cell is associated with a list of the objects currently residing inside it. Clearly such approach is very efficient from a computational point of view since the maintenance of the index structure is straightforward. It can handle very effectively issues like frequent updates, high arrival raters, the infinite nature of the data and so on. However it can be used only for a queries focused on the current state of the stream.

Multiple algorithms have been proposed for answering range predicates with the grid based indexing solutions. Gedik and Liu [7] propose a distributed system for range queries called ”mobieyes”, and it is assumed that the moving clients can process and store information. The client receive information about the moving range query from the server and notifies the server when it is going to enter or leave this query region. Mokbel et al [23] implements SINA, a scalable shared execution and incremental evaluation system, where the execution of continuous range spatiotemporal queries is abstracted as a spatial join between the set of objects and the set of queries. Continuous evaluation of nearest neighbor queries have also received a lot of attention lately using grid structures [12] [34] [33] [24]. Koudas et al [12] propose DISC a technique for answering ϵ-approximate k nearest neighbor queries over a stream of multidimensional data. The returned k^{th} Nearest neighbor lies at most on distance $d + \epsilon$ from the query point where d is the distance between the actual k^{th} Nearest neighbor and the query point. Yu at al.[34] and Xiong at al. [33] propose similar approaches for answering continuous NN queries using different distances for pruning. Finally Mouratidis et al. [24] introduced conceptual partitioning which archives a better performance than the previous approaches by handling updates from objects which fall in vicinity of the query.

The second group of indexing methods uses different tree-like structures. There are structures based on B+-trees [17] [11], R trees [14] [15] and TPR-trees [32] [16] [29] [30]. The main objective is to improve the update performance of the indexing structure since it is the most frequent operation in streaming environment. In [14] the reduction of the update cost is done trough avoiding the updates for objects that do not move outside of their MBRs. Later in [15] this technique is generalized trough a bottom - up update strategy which uses different levels of reorganization during the updates and in this way avoids the expensive top-down updates. The minimization of

the update time in [17] [11] is achieved trough the use of B+ trees, which have better update characteristics, instead of traditional multidimensional index structures like R-tree [9] [4]. This is achieved trough linearization of the representation of the moving objects locations using space filing curves like the Peano [6] or Z curve.

The last group [28, 22] of query evaluation methods for streams tries to avoid the expensive maintenance of index structures over the data. These methods are based on the notion of "safe" regions, created around the data [28] or uncertainty regions around the query [22]. If the object doesn't leave its safe region no further processing is required. And the reverse - in [22] objects are considered only if they fall inside the query region or its uncertainty regions.

All these indexing structures, discussed so far, try to improve the performance by minimizing the update rate. To the best of our knowledge there has not been any approach to improve the index performance from point of view of the query evaluation strategy. Later in this paper we propose a novel indexing structure which has fast object update rate and is oriented towards the incremental evaluation (i.e. an approach that reuses the result from the previous step).

3 Problem Definition

Consider a system that continuously monitors the locations of a set of moving objects. Location updates arrive as a stream of tuples $S = \langle u_1, u_2, \ldots, u_l, \ldots \rangle$ where $u_i = \langle o_i, l_i, t_i \rangle$, and o_i is the object issuing the update while l_i is the new location of the object on the plane and t_i is the current time stamp. $l_i \in \mathbb{R}^d, o_i \in \mathbb{N}$ (for simplicity we can assume a two dimensional plane).

Trajectory $T(o_i)$ of an object o_i in a stream S is a sequence of pairs $\{\langle l_1, t_1 \rangle, \ldots, \langle l_n, t_n \rangle\}$, where $l_i \in \mathbb{R}^d, t_i \in \mathbb{N}$. Let t_{now} denote the ever increasing current time instant ($t_{now} \in \mathbb{N}$). The definition of the CSTJ query follows:

Given trajectory sets r and s, the CSTJ query continuously returns all trajectory pairs $\langle T(o_{ri}), T(o_{si}) \rangle$ which have been spatially close (within threshold ϵ) for some time period δt ending at the current timestamp (i.e. the temporal constraint uses as a reference point the current timestamp). An example of such a relative time constraint is the restriction "in the last 30 minutes". In contrast, absolute time constraints (e.g. "between 2:30pm and 3:40pm") produce a query result that is static and does not change with time. In a continuous query environment, as the current time proceeds some objects will expire from the observed period while others will be introduced, thus continuously changing the join result.

Continuous Spatiotemporal Trajectory Join Given two sets of moving objects o_r and o_s, a spatial threshold ϵ and a (relative) time period δt ($\delta t \in \mathbb{N}$), the CSTJ returns continuously the set of pairs $\langle o_{ri}, o_{sj} \rangle$ such that for every time instance t_i between t_{now} and $t_{now} - \delta t$ the spatial distance between the trajectories $T(o_{ri})$ and $T(o_{sj})$ is less than the threshold ϵ.

4 Evaluation Framework

The basic idea behind the evaluation algorithms for the static version of the problem [2][3] is to find a way to prune as many trajectory pair similarity evaluations as possible. There are two major elements needed for the efficient evaluation of a trajectory join, namely:

- First we need a compact object trajectory approximation. This requirement is necessary to make the index structure which stores the trajectory approximations small and thus fit into the main memory for efficient access. It is assumed that the raw spatiotemporal stream data is too large to be kept in the main memory and has to be stored on a secondary storage devices. We further assume that the raw trajectory data is stored in lists of data pages per trajectory where each data page has a pointer to the next one in the list.
- Second, we require an easy to compute lower bound distance function between the trajectory approximations.

Using trajectory approximations and lower bound distance functions we can prune a large number of the pairs from the Cartesian product between the object trajectory sets $T(o_r)$ and $T(o_s)$. Because we work with trajectory approximations instead of the actual (full) trajectory data a verification step is also needed, where the pairs of trajectories, not pruned away by the distance function are then verified to satisfy the join criteria using their actual trajectory data. The lower bound distance function defined in this paper guarantees that we may have only "false positives" in the verification step (i.e., some trajectory pairs not pruned away by the lower bound distance may still not satisfy the join criteria) but no "false negatives" (i.e., no join result is missed). To remove these false positives in the final result we need the extra verification step which access the raw trajectory data on the secondary storage device and verifies for each pair that it indeed satisfies the join criteria. Hence the total cost of a single evaluation iteration will comprise of two parts:

- The cost of computing the lower-bounding distances.
- The cost of executing the verification step.

For the continuous version of the problem there are also additional requirements. The trajectory approximation should be easy to compute and maintain. This requirement is needed since the approximation is created on the fly as the streaming data enters the server. For example the static approximation discussed in [2] [3] does not satisfy this condition because it makes very expensive aggregations over the raw trajectory data in both the temporal and spatial domains. Moreover the lower bound distance function should be defined in such way that allows the application of the incremental approach. This means that it should be possible to reuse the results from one iteration to another.

4.1 Trajectory Approximation and Indexing

To produce the trajectory approximation with the required properties we decided to use of a uniform spatial grid to discretize the spatial domain. Each object location l_i

in the stream can be approximated with the grid cell in the boundaries of which it is. Example of a one dimensional trajectory is shown on figure 2. The trajectory shown can be approximated as a sequence of grid cell numbers 1112222333. We can write this approximation in more a compact form $< 1, 3 >< 2, 7 >< 3, 10 >$ by compressing the consecutive object location approximations in the same grid cell $\bar{x}_i$. They are replaced by record $< \bar{x}_i, t_j >$ where t_i is the last time instant in the list of consecutive object location approximations for grid cell $\bar{x}_i$. For example the trajectory on figure 2 has three consecutive location/time instances $u_i = \langle o_i, l_i, t_i \rangle$ all of them in the grid cell 1. These are time instances 1, 2 ad 3. Instead of having a sequence 111 in the approximation we put a record with the grid cell number and the last time instance in the sequence $< 1, 3 >$. More formally a trajectory approximation can be defined as:

Trajectory approximation Given trajectory $T(o_i) = \{< l_1, t_1 >, \ldots < l_n, t_n >\}$ of length n, a trajectory approximation is a sequence $\bar{T}(o_i) = \{< \bar{x}_1, \bar{t}_1 >, \ldots, < \bar{x}_m, \bar{t}_m >\}$ where the spatial values contained inside each time frame $(t_{i-1}; t_i)$ are approximated with the grid cell number $\bar{x}_i$ and $1 \leq m \leq n$

Note that this approximation scheme is different from the scheme proposed for the static version of the join problem [2][3], since it approximates the trajectory data only in the spatial domain. We thus avoid the costly approximation along the temporal axis which can be too expensive for the continuous environment.

We proceed with the description of the indexing structure. We propose the creation of a 2-dimensional index space where both dimensions are temporal (”from” and ”to” axes) On the “from” temporal axis we plot the time when the object enters a given cell while the “to” temporal axis depicts the time when the object leaves the grid cell. Between these two timestamps the object does not move outside the boundaries of the grid cell. For every record $< \bar{x}_i, \bar{t}_i >$ in the trajectory approximation we place a two dimensional point I_i in the indexing space. We refer to these points as indexing points. A more formal definition of an index point is:

Index point is a tuple $I_i = \langle o_i, \bar{g}, t_f, t_t, p \rangle$, where o_i is the moving object, $\bar{g}$ is a grid cell number such that $\cap(\bar{g}, T_{o_i}) = true$ for $\forall t \in (t_f, \ldots t_t)$ and p is a pointer to pages on the disk containing the raw trajectory data for time period $(t_f; t_t)$.

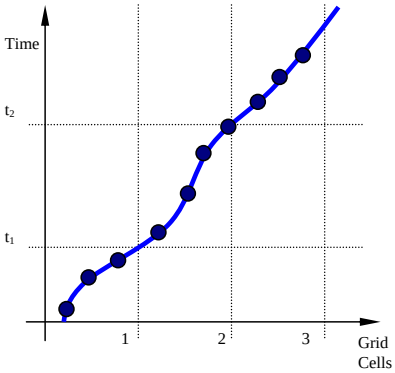

Fig. 2. One dimensional example.

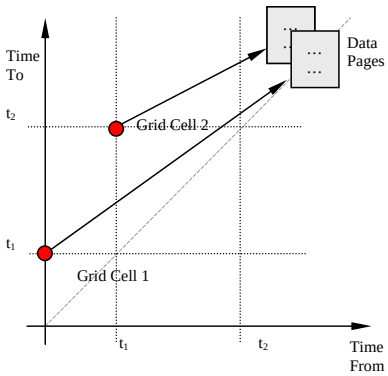

Fig. 3. Indexing space.

For illustration consider the trajectory shown in figure 2. The object stays inside grid cell 1 between time instances 0 and t_1 and has 3 location/time instances $u_i = \langle o_i, l_i, t_i \rangle$ inside this grid cell then it moves to grid cell 2 and stays there between time instances t_1 and t_2. Finally it moves inside grid cell 3. Figure 3 depicts the corresponding indexing space for this example. The object movement is approximated with two *index* points in the indexing space. Both of them show the time period for which the object was inside a given grid cell. For example index point 1 in this 2-dimensional space shows that the moving object was in grid cell 1 in the time interval $(0; t_1)$. Respectively object point 2 shows that moving object was in grid cell 2 in the time interval $(t_1; t_2)$ and so on. A trivial observation is the fact that all index points I_i will be placed above the dashed line on figure 3 which bisects the angle between the two temporal axes (that is because the timestamp when an object leaves an grid cell is bigger than the timestamp when the object enters a grid cell e.g. $\forall I_i; I_i.t_f < I_i.t_t$).

With the above approximation, an object trajectory is transformed to a set of index points in the 2 dimensional indexing space. Inside each index point we keep a pointer to the data page on the secondary level storage which stores the raw trajectory data approximated with this indexing point I_i. For example for index point 1 we keep a pointer to the data page on the disk which has the raw data for time instances 1, 2 and 3. These pointers are used in the verification step when we have to check if the objects indeed satisfy the join criteria using the raw data. Instead of accessing all records for the given trajectory we access only those data pages which have the data for the period of interest. To make the access to the indexing space more efficient we can now use variety of tree-like spatial indexes (R tree or kdb tree) build over all index points in the indexing space.

There are two major advantages of the proposed indexing structure over simple solutions like keeping a trajectory tail for the last δt time instances. First the size of this index is expected to be smaller than the size of an in memory data structure, which holds the fresh trajectory tail. This is because in the proposed index we keep information only for the moments when an object changes its location grid cell instead of keeping all location/timestamp pairs for a period δt. The second advantage is that by issuing a range query inside the index space we can efficiently locate all moving objects which change their location grid cell for the specified time period without accessing all trajectory tails (This will be discussed in detail in section 5).

As time passes, more and more index points will be added to the indexing space. The tree structure built over this space will grow and it performance will eventually deteriorate. Moreover we would like to keep the indexing space and the tree structure over it small enough to fit in main memory for fast access. Possible solution to this problem is to delete all index points I_i from the index space which are too ”old”. A data older than the time period δt cannot participate as a result so we can safely prune these regions of the indexing space. For example if the time period is $\delta t = 2$ hours then there is no need to keep data older than 2 hours in the structure. The index points, which we can safely remove from the index structure, will be in the shaded region, shown on figure 4.

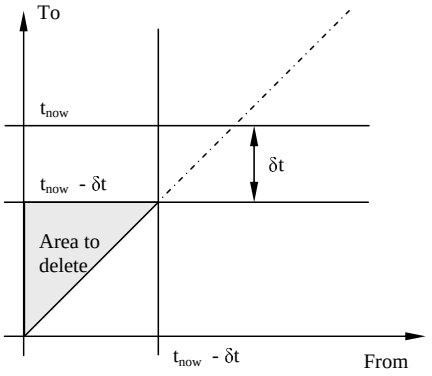

Fig. 4. Indexing space to remove.

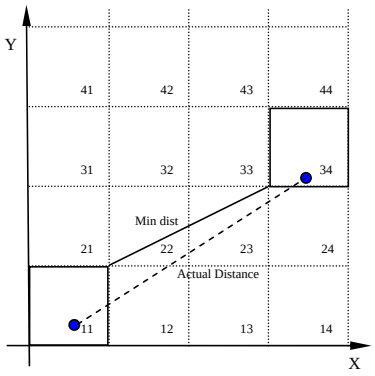

Fig. 5. Min. distance between two grid cells.

4.2 Lower bound distance function

Having defined the trajectory approximation we need an appropriate lower bound distance function between the approximated trajectories. Given that the minimal distance between two grid cells is a lower bound of the actual distance between the object locations ($d(\bar{x}_i, \bar{x}_j) \leq d(l_i, l_j)$), as it is shown on figure 5, we can define a lower bound function between the trajectory approximations using the minimal Euclidean distance between the grid cells, i.e.:

$$\bar{\mathcal{D}}_{\delta t, t_i}(\bar{T}(o_{ri}), \bar{T}(o_{si})) = \sqrt{\sum_{i \in (t_i; t_i - \delta t)} d(\bar{r}_i, \bar{s}_i)^2}$$

5 Query evaluation

We now proceed with the CSTJ evaluation algorithm which assumes a spatiotemporal stream of moving objects approximated in an index structure built as described in section 4.1. There are two major processes in a continuous query evaluation framework that are working in parallel. The first process is keeping the indexed space consistent with the spatiotemporal stream. The second process is responsible for the continuous reevaluation of the CSTJ queries in the system. During its lifetime, a CSTJ query goes through two phases, namely:

- Phase 1. Initial formation of the query result.
- Phase 2. Continuous query reevaluation.

During the first phase the CSTJ query is introduced into the system and the initial result is computed from scratch. Once the initial result of the query is formed, the evaluation of the continuous query moves to the second phase where the query stays in till it is taken out of the system. In this phase the query is reevaluated regularly and the results from the reevaluations are constantly send to the end users. In the remaining of this section we look at each phase in detail.

5.1 Initial formation of the query result

When the query is first introduced into the system, the result has to be computed from scratch. For this phase we modify the "multiple-origin" static join algorithm discussed in [3] to be used with the described indexing scheme. In particular we need to find all pairs of trajectories in the time period $(t_n - \delta t; t_n)$ where the corresponding grid cells for every time instance are not further apart than the threshold ϵ, i.e. $d(\bar{r}_i, \bar{s}_i) \leq \epsilon$, for $i \in (t_n - \delta t; t_n)$.

Each trajectory approximation of length δt can be viewed as a δt-dimensional point in a transformed δt-dimensional space. Using distance function $\bar{\mathcal{D}}_{\delta t}$ defined over the trajectory approximations we can define an ordering of the points in δt-dimensional space by sorting them according to their distances from some set of origins $\bar{O}_i$.

An origin $\bar{O}_i$ is an approximated trajectory with length δt and can be selected arbitrarily. We assign to every trajectory approximation $\bar{T}(o_i)$ a set of q scores ($w_1, \ldots$, $w_j, \ldots w_q$), where each score w_j is simply the distance $\bar{\mathcal{D}}_{\delta t}(\bar{O}_j, \bar{T}(o_i))$, between the trajectory approximation $\bar{T}(o_i)$ and origin $\bar{O}_j$. Approximations with different distances from a given single origin $\bar{O}_j$ are considered to be dissimilar. The reverse however is not true: we can have approximations with the same distance from origin $\bar{O}_j$ which are still not spatially close. To reduce the probability of this happening we thus use multiple origins. The verification step however is still needed.

To compute the object scores ($w_1, \ldots, w_j, \ldots w_q$) we use the index space maintained over the spatiotemporal stream. We locate the portions of the trajectory approximations which belong to the time interval $(t_n - \delta t; t_n)$ by issuing spatial queries in this index space. Given a grid cell and a time interval $(t_f; t_t)$ we can partition the space in four regions, as shown in figure 6.

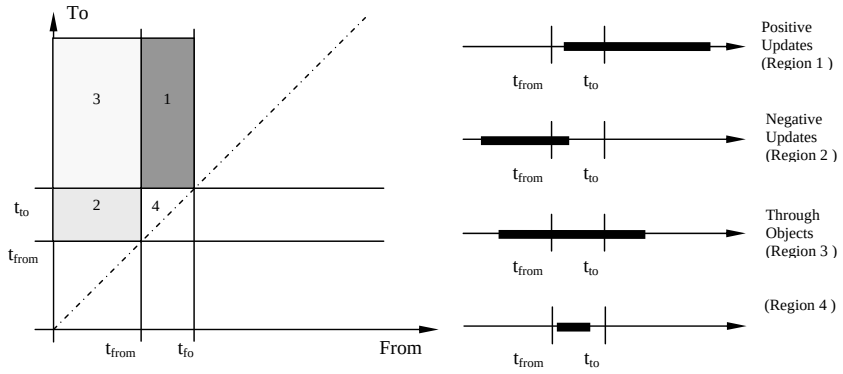

Fig. 6. Index space partitioning.

Region 2 contains index points for all these objects which were inside the given grid cell before the beginning of the interval t_f and which moved to another grid cell at some point during the interval $(t_f; t_t)$. Region 1 contains the objects which moved inside the given grid cell at some point during the interval $(t_f; t_t)$ and stayed there until the end of the time interval. Region 3 contains objects which were inside the given grid

cell all the time during the interval $(t_f; t_t)$ and region 4 contains objects which moved in and then moved out of the grid cell during the time interval. In regions 1, 2 and 3 for any time period we can have at most one index point I_i per object (for example, having two indexing points in region 3 would mean that the object was at the same time in two different grid cells during the specified period which is a contradiction).

Using spatial queries to locate the index points in these partitions of the index space for time interval $(t_n - \delta t, t_n)$ we can compute the trajectory scores by first computing the squared sum of distances between trajectory approximations and origin $\bar{O_j}$: Let I_1 be the set of indexing points for object o_i in region 1, I_2 the set of indexing points in region 2 and so on. The squared sum of distances between trajectory approximation $\bar{T}(o_i)$ and origin $\bar{O_j}$ will be:

$$\sigma_{t_i,\bar{O}_j}(o_i) = \sum \begin{cases} (I.t_t - (t_i - \delta t))d(I.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_1; \\ (t_i - I.t_f)d(I.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_2; \\ (\delta t)d(I.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_3; \\ (I.t_t - I.t_f)d(I.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_4. \end{cases}$$

$$w_j(o_i) = \bar{\mathcal{D}}_{\delta t, t_n}(\bar{T}(o_i), \bar{O}_j) = \sqrt{\sigma_{t_n,\bar{O}_j}(o_i)}$$

Since the lower bound distance $\bar{\mathcal{D}}$ is a metric, if two trajectory approximations have at least one w_j score larger than $\sqrt{\epsilon^2 \delta t}$, then there exists at least one time instance t_i in which the corresponding grid cells from the approximations are farther apart than the threshold ϵ. The distance between the grid cells is a lower bound between the actual position of the objects so the corresponding objects do not satisfy the join criteria.

To locate candidate join pairs we sort the trajectory approximations $\bar{T}(o_i)$ using the q scores, in order of $w_1, w_2, ...$, etc. That is, if there is a group of trajectory subsequences having the same value of w_1, they are further sorted on their w_2 score and so on.

Having the trajectory approximation scores $w_1, w_2, ...$, computed and the approximations sorted, we can locate the pairs for which could possibly join. This is done performed by a sliding window algorithm which passes over the sorted list of approximations. We set the size of the window to $2\sqrt{\epsilon^2 \delta t}$ and place the midpoint of the window on the first approximation from dataset s, say $\bar{T}(o_{sj})$, in the sorted list. For all approximations $\bar{T}(o_{ri})$ of dataset r falling inside the window, we compare their scores $w_1, ..., w_q$ with the corresponding scores of $\bar{T}(o_{sj})$. If all of them are within the threshold $\sqrt{\epsilon^2 \delta t}$ we save the pair $\langle o_{sj}, o_{ri} \rangle$ as possible join candidate. Then, we slide the window and place its midpoint on the next element in dataset s and and so on. At the end we verify the generated candidate pairs loading the raw data from the secondary storage. To reduce the number of I/O we follow the pointers inside the index points to locate the data pages storing information for this time period instead of having a full scan over the the pages storing raw trajectory data.

We illustrate the initial formation of the result with an example. Assume that we have 3 moving objects between time instances 1 and 11. Each object report its location every time instance (see figure 7 - the locations where an object reports its position are marked with a dot). We discretize the space with a grid 3x3 where each grid cell is a square with side 10. The minimal distance between the grid cells is given in table 6.1.

Algorithm 1 CSTJ - Initialization phase

Input: Query $\mathcal{Q} = \{o_r, o_s, \delta t, \epsilon\}$ current time instance t_i
Output: Set of pairs (o_{ri}, o_{sj}) where $T(o_{ri})$ and $T(o_{sj})$ are joined for the last δt time instances

Set $\sigma \leftarrow \emptyset$, $W \leftarrow \emptyset$, $V \leftarrow \emptyset$, $Res \leftarrow \emptyset$
Find Origins(O_1 .. O_m);
ComputeApproximations($\tilde{O_1}$...$\tilde{O_m}$);
CreatePartitions($t_i - \delta t$,t_i);
GetIntexPoints(I_1, I_2, I_3, I_4);
for each origin O_j in O_1 .. O_m **do**
    for each moving object o_r in $o_r \cup o_s$ **do**
        Compute $\sigma_{t_i,\bar{O}_j}(I_i.o)$
        σ.push($\sigma_{t_i,\bar{O}_j}(I_i.o)$);
for all $\sigma_{t_i,\bar{O}_j}(o_i)$ in σ **do**
    $W_{i,j} = \sqrt{\sigma_{t_i,\bar{O}_j}(o_i)}$
W.sort()
for (i = 1; i¡= W.size; i++) **do**
    Entry $o_x = W$[i].objectID
    if $x \in o_r$ **then**, FindPairsInWindow(o_x,i,W,V,ϵ)
while V not empty **do**
    Entry $< o_{ri}, o_{sj} >= V$.top
    if $o_{ri} \in o_r$ and $o_{sj} \in o_s$ satisfy the criteria **then**
        R.push(o_{ri}, o_{sj})
Return R

Grid No	11	12	13	21	22	23	31	32	33
11	0	0	10	0	0	10	10	10	14
12	0	0	0	0	0	0	10	10	10
13	10	0	0	10	0	0	14	10	10
21	0	0	10	0	0	10	0	0	10
22	0	0	0	0	0	0	0	0	0
23	10	0	0	10	0	0	10	0	0
31	10	10	14	0	0	10	0	0	10
32	10	10	10	0	0	0	0	0	0
33	14	10	10	10	0	0	10	0	0

The indexing space for this example is shown in figure 8. Next to each index point we have the moving object number o_i to which it belongs and the grid cell number $\bar{x}_j$ in format $\langle o_i, \bar{x}_j \rangle$. Consider a CSTJ query with spatial threshold $\epsilon = 3$ and time period $\delta t = 3$, that is introduced in the system on time instance 6. Object o_1 and object o_3 belong to the first set s and object o_2 belongs to the second set r. The partitioning for time interval $(3; 6)$ is shown in figure 8.

We have three indexing points in region 1 (one for each moving object) and three indexing points in region 2. This means that during this time period each object left the grid cell where it was in the beginning of the period and moved into another grid cell. For simplicity we choose a scenario with two origins. Both of them are trajectory approximations with length $\delta t = 3$. Again for simplicity we choose the first one to be

Algorithm 2 FindPairsInWindow

Input: o_x,i,W,V,ϵ

1: $j \leftarrow i$
2: **while** W[j].scores - o_x.scores $< \sqrt{\epsilon^2 \delta t}$ **do**
3: Entry $o_y = W$[j]
4: **if** $o_y \in o_s$ **then**, V.push(o_x, o_y)
5: $j--$
6: $j \leftarrow i$
7: **while** W[j].score - o_x.score $< \sqrt{\epsilon^2 \delta t}$ **do**
8: Entry $o_y = W$[j]
9: **if** $o_y \in o_s$ **then**, V.push(o_x, o_y)
10: $j++$

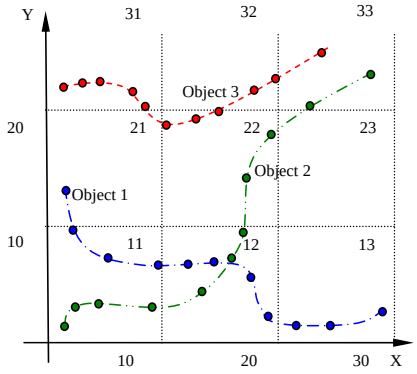

Fig. 7. Moving objects example.

$\bar{O}_1 = 33, 33, 33$ and the second one $\bar{O}_2 = 31, 31, 31$. We can then compute the scores for the first trajectory $w_1(o_1) = \sqrt{(4-(6-3))(14)^2 + (6-4)*(10)^2} = 19,89$ and $w_2(o_1) = \sqrt{(4-(6-3))(10)^2 + (6-4)*(10)^2} = 17,32$. In the same way we compute the scores for the other moving objects $w_1(o_2) = 19,89$, $w_2(o_2) = 17,32$, $w_1(o_3) = 14,14$ and $w_2(o_3) = 0$.

Objects are sorted and placed on a line and then we use the sliding window algorithm where the size of the window is $2\sqrt{\epsilon^2 \delta t} = 2\sqrt{3^2 3} = 5,19$ (figure 10). We place the midpoint of the window on the first s element in the sorted list o_3 we check if there are elements from the second set inside the window. Since there are none we place the window over the next s element o_1. This time there is an element from the set r inside the window - o_2. Thus we report the pair $\langle o_1, o_2 \rangle$ as a candidate pair. There are no more elements in the sorted list so we exit the sliding window algorithm. At the end of the initial evaluation phase we check every candidate pair ($\langle o_1, o_2 \rangle$ in our example) if it indeed satisfies the query criteria.

5.2 Continuous Query Reevaluation

Once the initial result is formed the evaluation of the query moves to its second phase where the query stays active until it is removed from the monitoring system. In this

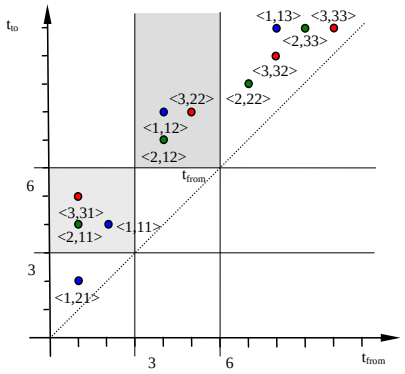

Fig. 8. Partitioning for time interval (3;6).

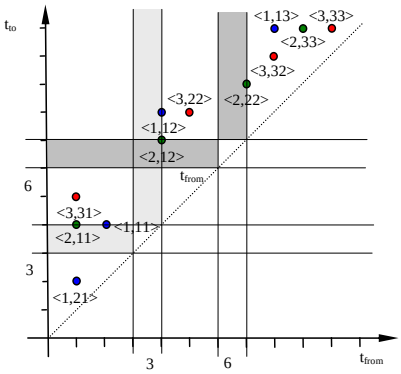

Fig. 9. Time intervals (3;4) and (6;7).

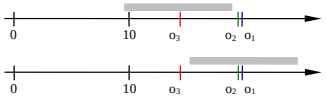

Fig. 10. Sliding window algorithm.

phase we constantly reevaluate the query result and modify it according to the current state of the stream S. To minimize the reevaluation cost we keep the intermediate results produced in every iteration and apply to them the changes which have occurred in the stream. We keep the first $T(o_i).\bar{f}$ and the last grid cell $T(o_i).\bar{l}$ from the trajectory approximations computed in the previous step along with the sum of the squared distances to the origins $\sigma_{t_i,\bar{O}_j}(o_i)$.

Assume that the last query reevaluation was at time instance t_p and the current timestamp is t_n. Using the partitioning in the index space shown on figure 3 we can compute the grid cells which form the trajectory approximation for the time period $(t_p - \delta t; t_n - \delta t)$ and those who form the approximation for $(t_p; t_n)$. These portions in the trajectory approximation sustain the difference between the approximations at time instances t_p and t_n. To do so we issue two spatial queries for regions 1, 2 and 4, using time interval $(t_p - \delta t; t_n - \delta t)$ and also time interval $(t_p; t_n)$ to create the partitioning in the index space as it is shown on figure 6. We are focused on these 3 partitions because they contain information about changes in object location during these periods. If there is no change in a location for the period $(t_p - \delta t; t_n - \delta t)$ (e.g there are no indexing points for this object in regions 1,2 and 4 in the partitioning for this time interval) then the object is inside the first grid cell $T(o_i).\bar{f}$ for the whole time interval $(t_p - \delta t; t_n - \delta t)$. The same for time interval $(t_p; t_n)$. If there is no index point in regions 1,2 and 4 for this time period for some object, then the object is still inside grid cell $T(o_i).\bar{l}$. This way keeping the first and the last grid from the grid approximation from the previous time period we avoid the costly spatial query inside region 3 which has the biggest size of all regions.

Using indexing points I_n from the first spatial query along with the last grid cell $T(o_i).\bar{l}$ we compute the sum of squared distances for the interval $(t_p - \delta t; t_n - \delta t)$

$$\Delta_{neg} = \sum \begin{cases} (t_n - t_p) d(T(o_i).\bar{l}, \bar{O}_j)^2 \text{ if } I_{n1} \cup I_{n2} \cup I_{n3} \in \emptyset; \\ (I_1.t_t - (t_p - \delta t)) d(T(o_i).\bar{l}, \bar{O}_j)^2 \text{ for } I \in I_{n1}; \\ (t_p - I_1.t_f) d(I_1.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_{n2}; \\ (I_1.t_t - I_1.t_f) d(I_1.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_{n4}. \end{cases}$$

This sum of squared distances for $(t_p - \delta t; t_n - \delta t)$ has to be removed from $\sigma_{t_n, \bar{O}_j}(o_i)$ from the previous iteration.

In analogy we compute the sum of squared distances for the period $(t_p; t_n)$ using the indexing points I_p from the second query and the first grid cell $T(o_i).\bar{f}$.

$$\Delta_{pos} = \sum \begin{cases} (t_n - t_p) d(T(o_i).\bar{f}, \bar{O}_j)^2 \text{ if } I_{p1} \cup I_{p2} \cup I_{p3} \in \emptyset; \\ (I_2.t_t - (t_n - \delta t)) d(T(o_i).\bar{f}, \bar{O}_j)^2 \text{ for } I \in I_{p1}; \\ (t_n - I_2.t_f) d(I_2.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_{p2}; \\ (I_2.t_t - I_2.t_f) d(I_2.\bar{g}, \bar{O}_j)^2 \text{ for } I \in I_{p4}. \end{cases}$$

Having $\sigma_{t_p, \bar{O}_j}(o_i)$ from the previous iteration, we can compute the scores for the current time instance

$$\sigma_{t_n, \bar{O}_j}(o_i) = \sigma_{t_p, \bar{O}_j}(o_i) + \Delta_{pos} - \Delta_{neg}$$

$$w_j(o_i) = \bar{\mathcal{D}}_{\delta t, t_n}(\bar{T}(o_i), \bar{O}_j) = \sqrt{\sigma_{t_n, \bar{O}_j}(o_i)}$$

The trajectory scores $w_1, w_2, ...,$ are resorted and processed with the multiple origins sliding window evaluation algorithm to produce the result for time instance t_n. An advantage of this reevaluation schema is that by having a short reevaluation period, the size of regions 1, 2 and 4 in the index partitioning schema will be comparatively small resulting in a limited number of index points accessed during the reevaluation steps.

We will illustrate the reevaluation phase using the same example shown in figure 7. Assume that the query reevaluation is done every time instance and that the current time instance is 7 (e.g one time instance after the initial evaluation). We create the partitioning for time intervals $(3; 4)$ and $(6; 7)$ according to the algorithm. The result is shown on figure 9. There are two indexing points in both regions 2 and 3 (they belong to objects o_1 and o_2) for time interval $(3; 4)$ and one indexing point in the same regions for time interval $(6; 7)$ (generated from object o_2). There are no indexing points for object 3 which means that this object did not change its grid cell during the interval $(3; 4)$ and for this period it was inside grid cell 31 (this is the first grid cell $T(o_3).\bar{f} = 31$ in the trajectory approximation in the previous evaluation step done for time period $(3; 6)$). In analogy object 3 was inside grid cell 22 (which is $T(o_3).\bar{l}$) during the interval $(6; 7)$. We compute $\Delta_{neg}(o_3, \bar{O}_1) = (4-3)10^2 = 100$, $\Delta_{neg}(o_3, \bar{O}_2) = (4-3)0^2 = 0$, $\Delta_{pos}(o_3, \bar{O}_1) = (7-6)0^2 = 0$ and $\Delta_{pos}(o_3, \bar{O}_2) = (7-6)0^2 = 0$. So the updated scores for object 3 are $w_1(o_3) = \sqrt{\sigma_{t_6, \bar{O}_1}(o_3) + \Delta_{pos}(o_3, \bar{O}_1) - \Delta_{neg}(o_3, \bar{O}_1)} = \sqrt{200 - 100} = 10$ and $w_2(o_3) = \sqrt{\sigma_{t_6, \bar{O}_2}(o_3) + \Delta_{pos}(o_3, \bar{O}_2) - \Delta_{neg}(o_3, \bar{O}_2)} = \sqrt{0} = 0$. Similarly, $\Delta_{neg}(o_1, \bar{O}_1) = 196$, $\Delta_{neg}(o_1, \bar{O}_2) = 100$, $\Delta_{pos}(o_1, \bar{O}_2) = 100$, $\Delta_{pos}(o_1, \bar{O}_2) = 100$, $\Delta_{neg}(o_2, \bar{O}_1) = 196$, $\Delta_{neg}(o_2, \bar{O}_2) = 100$, $\Delta_{pos}(o_2, \bar{O}_2) = 100$, $\Delta_{pos}(o_2, \bar{O}_2) = 100$ and the new scores for objects o_1 and o_2 $w_1(o_1) = 17.32$,

Algorithm 3 CSTJ - Continuous phase

Input: Query $\mathcal{Q} = \{o_r, o_s, \delta t, \epsilon\}$,$\sigma$,$\tilde{O_1}$...$\tilde{O_m}$,$t_p$,$t_n$
Output: Set of pairs (o_{ri}, o_{sj}) where $T(o_{ri})$ and $T(o_{sj})$ are joined for the last δt time instances

Set $W \leftarrow \emptyset$, $V \leftarrow \emptyset$, $Res \leftarrow \emptyset$
CreatePartitions$(t_p; t_n)$;
GetIntexPoints(I_{p1}, I_{p2}, I_{p4});
CreatePartitions$(t_p - \delta t; t_n - \delta t)$;
GetIntexPoints(I_{n1}, I_{n2}, I_{n4});
for each origin O_j in O_1 .. O_m **do**
    for each moving object o_r in $o_r \cup o_s$ **do**
        Compute $\Delta_{pos,i,j}$
        Compute $\Delta_{neg,i,j}$
        σ.pop($\sigma_{t_p,\bar{O}_j}(I_i.o)$);
        $\sigma_{t_n,\bar{O}_j}(I_i.o) = \sigma_{t_p,\bar{O}_j}(I_i.o) + \Delta_{pos,i,j} - \Delta_{neg,i,j}$
        σ.push($\sigma_{t_n,\bar{O}_j}(I_i.o)$);
for all $\sigma_{t_n,\bar{O}_j}(o_i)$ in σ **do**
    $W_{i,j} = \sqrt{\sigma_{t_n,\bar{O}_j}(o_i)}$
W.sort()
for (i = 1; i¡= W.size; i++) **do**
    Entry $o_x = W$[i].objectID
    if $x \in o_r$ **then**, FindPairsInWindow(o_x,i,W,V,ϵ)
while V not empty **do**
    Entry $< o_{ri}, o_{sj} >= V$.top
    if $o_{ri} \in o_r$ and $o_{sj} \in o_s$ satisfy the criteria **then**
        R.push(o_{ri}, o_{sj})
Return R

$w_2(o_1) = 17.32$, $w_1(o_2) = 17.32$ and $w_2(o_2) = 17.32$. Then the objects are resorted using the new scores and the sliding window algorithm is re-run.

6 Experimental Evaluation

We proceed with the experimental evaluation of the proposed indexing structure and algorithms for continuous evaluation of CSTJ queries.

6.1 Experimental Environment

In our experiments we use synthetic data to test the behavior of the proposed technique and indexing structure under different settings. We generated synthetic datasets of moving object trajectories. The datasets are generated by simulation using the the freeway network of Indiana and Illinois (see figure 11). We use up to 150,000 objects moving in a 2-dimensional spatial universe which is 1,000 miles long in each direction. The object velocities follow a Gaussian distribution with mean 60 mph, and standard deviation 15 mph. We run simulations for 1000 minutes (time-instants). Objects follow random routes on the network traveling through a number of consecutive intersections and report their position every time-instant. In addition, at least 10% of the objects issue a

modification of their movement parameters per time-instant. We choose an R tree with utilization factor 64% as an indexing structure build on the top of the indexing space.

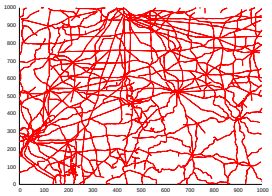

Fig. 11. The map used in the simulations.

The average number of time/location tuples $\langle t_i, l_i \rangle$ per index point I is 11 (it is for the selected speed of 60 mph - later in this section we will present experimental results for objects with average speed less than 60 mph). The maximal relative temporal constraint in the queries is set to $\delta t = 40$ minutes. For the maximal time period $\delta t = 40$ minutes, the R tree build over the indexing space has the properties described in the table below.

Property	Value
Tree height	5
Number of nodes	39900
Leaf capacity	20
Index capacity	20

To test the proposed techniques we use two measures: The average number of index node accesses and the average number of data pages per query that need to be retrieved from storage for verification of the result, assuming that one random access is needed for this operation. We also measure the number of trajectory pairs which satisfy the query (e.g the size of the result). We evaluate the query performance in both the initialization and continuous reevaluation phases. Query reevaluation is performed every 2 minutes.

6.2 Experimental Results

Varying the Dataset Size In the first group of experiments we measure the performance against different data set sizes. We use 4 different datasets with sizes varying from 25,000 up to 150,000 moving objects. The spatial threshold is set to $\epsilon = 30$ miles and the time period δt is set to 20 minutes. The results for the index nodes access, data page access and the number of pairs are shown in figures 12, 13 and 14. As the dataset size increases, the numbers of data pages and index nodes accessed are also growing. As depicted in figure 12, the number of data pages accessed in the initialization and the continuous phases are similar due to the similar number of candidate pairs generated in both phases. The index node access however (figure 13) differs substantially in the two phases. In the continuous phase due to the incremental approach the number of

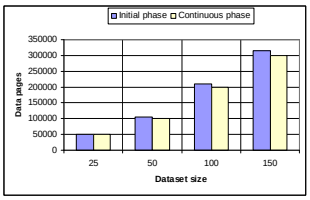

Fig. 12. Size: Data pages.

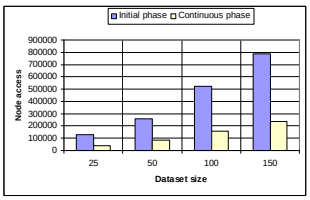

Fig. 13. Size: Index nodes.

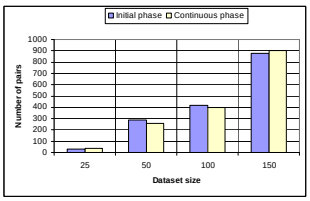

Fig. 14. Size: Result set size.

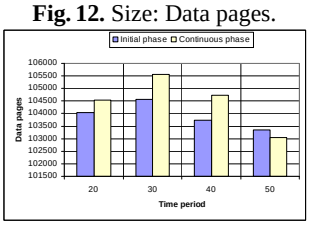

Fig. 15. Time: Data pages.

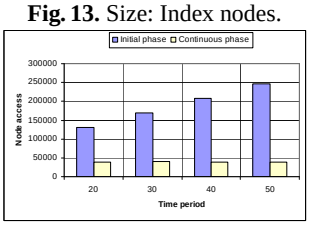

Fig. 16. Time: Index nodes.

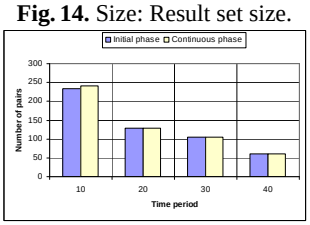

Fig. 17. Time: Result set size.

the accessed index nodes is much smaller than in the initial phase. In the continuous phase we do not access the indexing points in region 3 which has the largest size of all 4 regions. Though the distribution of the points in the indexing space is not uniform this region has the biggest number of index points from all four regions in the index space partitioning. As expected, the number of trajectory pairs which satisfy the query is growing with the increase of the dataset size (figure 14).

Varying the Spatial threshold ϵ In the next set of experiments we test the behavior of the algorithm for increasing query threshold ϵ (while using a fixed time-interval $\delta t = 20$ minutes). The intuition behind this set of experiments is that by increasing the spatial threshold ϵ the query becomes more relaxed and thus more expensive for evaluation. We use four different values for ϵ varying from 10 to 40 miles. The results are shown in figures 15 16 and 17. As expected with the increase of the threshold we have moderate increase in the number of candidate pairs and the number of result pairs (figure 17). Due to the increased number of reported candidate pairs the data pages accessed (figure 15) also increase since each candidate pair has to be tested using the raw trajectory data. The number of index node accessed however remains constant since the trajectory score computation does not depend on the threshold ϵ (figure 16).

Varying the Time period δt In the next group of experiments we tested the behavior of the proposed algorithm for different values of the time period δt varying from 20 up to 50 minutes. We use dataset containing 50,000 moving objects. The spatial threshold is set to $\epsilon = 20$ miles. As it can be depicted from the plots, the number of index nodes accessed during the initialization phase (figure 19) is increasing proportionally to the increase of the time period δt. This proportional increase is due to the fact that larger time periods δt create larger regions 1 and 2 in the space indexing partitioning resulting

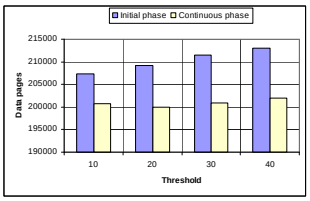

Fig. 18. ϵ: Data pages.

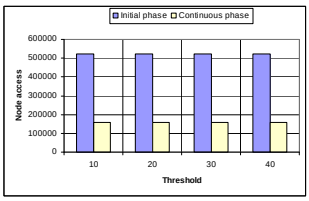

Fig. 19. ϵ: Index nodes.

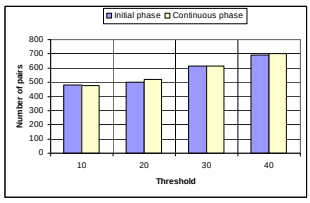

Fig. 20. ϵ: Result set size.

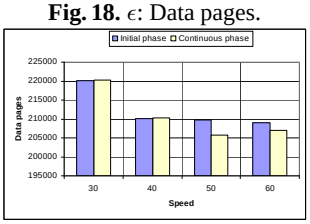

Fig. 21. Speed: Data pages.

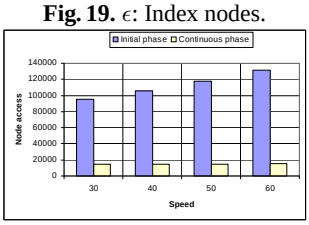

Fig. 22. Speed: Index nodes.

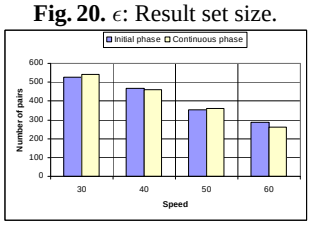

Fig. 23. Speed: Result set size.

in larger number of indexing nodes accessed for these two regions. From the plot in figure 18 it can be seen that after time period $\delta t = 30$ minutes the number of raw data I/Os decrease. This is due to the fact that by increasing the length of the query it becomes more restrictive. We have fewer candidates generated and therefore fewer raw data accesses.

The decreased number of candidate pairs results also in a smaller number of pairs in the result set as it can be seen on figure 20.

Varying the average speed of the moving objects All previous experiments were performed using datasets where the average speed is set to 60 mph which is reasonable for a highway traffic. In the last set of experiments we study how the speed of the moving objects affects our algorithm. The intuition here is that by having a slowly moving objects in the system, it will take more time for the object to reach the boundaries of a cell and move to another one. The number of time/location tuples $\langle t_i, l_i \rangle$ per index point I_i is increased and the total number of index points in the indexing space is decreased. We run this set of experiments with four datasets of 100,000 moving objects, where the average speed varies from 30mph to 60mph. The spatial threshold in the query is set to $\epsilon = 30$ miles and the time interval δt is 20 time instances. As expected, the decrease of the average speed in the dataset results in a decrease of the number of indexing nodes accessed (figure 22). The indexing space becomes less dense with the decrease of the average speed. For the same time interval and the same number of moving objects the number of time/location tuples per index point in the 30 mph dataset is 70% from the one in the 60 mph dataset. The number of the trajectory pairs in the result set (figure 23) and the number of data pages I/Os (figure 21) however increase with the decrease of the average speed. This is because in a pair of slow objects, it takes more time for one of them to move on distance ϵ from the second one. So if a pair of slowly moving objects, satisfies the join criteria at one time instance it is more likely to satisfy it in

the next time instance. This results in a bigger number of candidate pairs and therefore increased number of raw data I/Os as it can be depicted in figures 21 and 23.

7 Conclusions

We presented an algorithm and an index structure for efficiently evaluating continuous trajectory join queries. Our technique uses compact trajectory representations to build a very small index structure which evaluates approximate answers utilizing a specialized lower bounding distance function. Then, a post filtering step uses only a small fraction of the actual trajectory data before the correct query results can be produced. As future work we plan to extend our techniques for more complex streaming queries with temporal constraints.

References

1. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. of Very Large Data Bases (VLDB)*, pages 570–581, 1998.
2. Petko Bakalov, Marios Hadjieleftheriou, Eamonn Keogh, and Vassilis J. Tsotras. Efficient trajectory joins using symbolic representations. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 86–93, 2005.
3. Petko Bakalov, Marios Hadjieleftheriou, and Vassilis J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 182–191, 2005.
4. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM Management of Data (SIGMOD)*, pages 220–231, 1990.
5. T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. of ACM Management of Data (SIGMOD)*, pages 237–246, 1993.
6. C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 247–252, 1989.
7. B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proc. of Extending Database Technology (EDBT)*, pages 67–87, 2004.
8. H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 336–344, 1991.
9. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.
10. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 237–248, 1998.
11. C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of of moving objects. In *Proc. of Very Large Data Bases (VLDB)*, 2004.
12. N. Koudas, B. C. Ooi, K-L. Tan, and R. Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *Proc. of Very Large Data Bases (VLDB)*, 2004.
13. N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. of ACM Management of Data (SIGMOD)*, pages 324–335, 1997.
14. D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 113–120, 2002.

15. M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *Proc. of Very Large Data Bases (VLDB)*, 2003.
16. B. Lin and J. Su. On bulk loading tpr-tree. In *Proc. of the International Conference on Mobile Data Management (MDM)*, 2004.
17. D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 59–66, 2005.
18. M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. of ACM Management of Data (SIGMOD)*, pages 209–220, 1994.
19. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM Management of Data (SIGMOD)*, pages 247–258, 1996.
20. N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Transactions on Database Systems (TODS)*, 26(4):424–475, 2001.
21. N. Mamoulis and D. Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(1):211–231, 2003.
22. M. F. Mokbel and W. G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 155–163, 2005.
23. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatiotemporal databases. In *Proc. of ACM Management of Data (SIGMOD)*, 2004.
24. K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proc. of ACM Management of Data (SIGMOD)*, pages 634–645, 2005.
25. A. Papadopoulos, P. Rigaux, and M. Scholl. A performance evaluation of spatial join processing strategies. pages 286–307, 1999.
26. J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: an efficient index for predicted trajectories. In *Proc. of ACM Management of Data (SIGMOD)*, pages 635–646, 2004.
27. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. of ACM Management of Data (SIGMOD)*, pages 259–270, 1996.
28. S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput.*, 51(10):1124–1140, 2002.
29. S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 463–472, 2002.
30. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Record*, 29(2):331–342, 2000.
31. J. Shan, D. Zhang, and B. Salzberg. On spatial-range closest-pair query. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 252–269, 2003.
32. Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. of Very Large Data Bases (VLDB)*, pages 790–801, 2003.
33. X. Xiong, M.. Mokbel, and W. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 643–654, 2005.
34. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 631–642, 2005.
35. D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 121–132, 2002.