In-Memory Distance Threshold Queries on Moving Object Trajectories

Michael Gowanlock
Department of Information and Computer Sciences and
NASA Astrobiology Institute
University of Hawai‘i, Honolulu, HI, U.S.A.
Email: gowanloc@hawaii.edu

Henri Casanova
Department of Information and Computer Sciences
University of Hawai‘i, Honolulu, HI, U.S.A.
Email: henric@hawaii.edu

***Abstract*—The need to query spatiotemporal databases that store trajectories of moving objects arises in a broad range of application domains. In this work, we focus on in-memory distance threshold queries which return all moving objects that are found within a given distance d of a fixed or moving object over a time interval. We propose algorithms to solve such queries efficiently, using an R-tree index to store trajectory data and two methods for filtering out trajectory segments so as to reduce segment processing time. We evaluate our algorithms on both real-world and synthetic in-memory trajectory datasets. Choosing an efficient trajectory splitting strategy to reduce index resolution increases the efficiency of distance threshold queries. Interestingly, the traditional notion of considering good trajectory splits by minimizing the volume of MBBs so as to reduce index overlap is not well-suited to improve the performance of in-memory distance threshold queries.**

Keywords-spatiotemporal databases; query optimization.

I. Introduction

Moving object databases (MODs) have gained attention as applications in several domains analyze trajectories of moving objects (animals, vehicles, humans, stellar bodies, etc.). Contributing to the motivation for MOD research is the proliferation of mobile devices that provide location information (e.g., GPS tracking). We focus on MODs that store historical trajectories [1], [2], [3], [4], such as the movement patterns of animals over a given period of observation, and that must support queries over subsets, or perhaps the full set, of the trajectory histories. In particular, we focus on two types of *distance threshold queries*:

1) Find all trajectories within a distance d of a given static point over a time interval $[t_0, t_1]$.
2) Find all trajectories within a distance d of a given trajectory over a time interval $[t_0, t_1]$.

An example query of the first type would be to find all animals within a distance d of a water source within a day. An example query of the second type would be to find all police vehicles on patrol within a distance d of a moving stolen vehicle during an afternoon. We investigate efficient distance threshold querying on MODs, making the following contributions:

- We propose algorithms to solve the two types of in-memory distance threshold queries above.
- We make the case for using an R-tree index for storing trajectory line segments.
- Given a set of candidate line segments returned from the R-tree, we propose methods to filter out line segments that are not part of the query result set.
- We propose decreasing index resolution to exploit the trade-off between the amount of index overlap and the number of entries in the index by exploring three trajectory splitting strategies.
- We demonstrate that, for in-memory queries, lower-bounding the index resolution is more important than minimizing the volume of hyperrectangular minimum bounding boxes (MBB), and thus index overlap.
- We evaluate our proposed algorithms using both real-world and synthetic datasets for both 3-D and 4-D trajectory data (i.e., the temporal dimension plus either 2 or 3 spatial dimensions).

This paper is organized as follows. In Section II, we outline related work. Section III defines the distance threshold query. Section IV discusses the indexing method. In Section V we present our algorithms and present an initial performance evaluation in Section VI. Section VII motivates, proposes, and evaluates methods to filter the candidate line segments. Section VIII presents and evaluates methods to split trajectories to reduce index resolution for efficient query processing. Finally, Section IX concludes the paper with a brief summary of findings and perspectives on future work.

II. Related Work

A trajectory is a set of points traversed by an object over time in Euclidean space. In MODs, trajectories are stored as sets of spatiotemporal line segments. The majority of the literature on indexing spatiotemporal data utilizes R-tree data structures [5]. An R-tree indexes spatial and spatiotemporal data using MBBs. Each trajectory segment is contained in one MBB. Leaf nodes in the R-tree store pointers to MBBs and the segments they contain (coordinates, trajectory id). A non-leaf node stores the dimensions of the MBB that contains all the MBBs stored (at the leaf nodes) in the non-leaf node's subtree. Searches traverse the tree to find all (leaf) MBBs that overlap with a query MBB. Variations of the R-tree and other methods have been proposed (TB-trees [6], STR-trees [6], 3DR-trees [7], SETI [8], and TrajStore [9]).

Few works on trajectory similarity searches have studied distance threshold queries [3]. Other types of trajectories, which we do not consider in this work, have been studied (e.g., flocks [10], convoys [4], swarms [11]). The well-studied k Nearest Neighbors (kNN) queries [12], [13], [14], [15] are related to distance threshold queries. A distance threshold query can be seen as a kNN query with an unknown value of k. As a result, previous work on kNN queries (with known k) cannot be applied directly to distance threshold queries (with unknown k).

III. Problem Statement

A. Motivating Example

One motivation application for this work is in the area of astrophysics [16]. The past decade of exoplanet searches implies that the Milky Way, and hence the universe, hosts many rocky, low mass planets that may sustain complex life. However, regions of a galaxy, such as the Milky Way, may be inhospitable due to transient radiation events, such as supernovae explosions or close encounters with flyby stars that can gravitationally perturb planetary systems. Studying habitability thus entails solving the following two types of *distance threshold queries* on the trajectories of (possibly billions of) stars orbiting the Milky Way: (i) Find all stars within a distance d of a supernova explosion, i.e., a non-moving point over a time interval; and (ii) Find the stars, and corresponding time periods, that host a habitable planet and are within a distance d of all other stellar trajectories.

B. Problem Definition

Let D be a database of N trajectories, where each trajectory T_i consists of n_i 4D (3 spatial + 1 temporal) line segments. Each line segment L in D is defined by the following attributes: x_{start}, y_{start}, z_{start}, t_{start}, x_{end}, y_{end}, z_{end}, t_{end}, trajectory id, and segment id. These coordinates for each segment define the segment's MBB (note that the temporal dimension is treated in the same manner as the spatial dimensions). Linear interpolation is used to answer queries that lie between t_{start} and t_{end} of a given line segment.

We consider historical continuous *searches* for trajectories within a distance d of a *query* Q, where Q is a moving object's trajectory, Q_t, or a stationary point, Q_p. More specifically:

- DistTrajSearch_Q$_p$(D,Q_p,Q_{start},Q_{end}, d) searches D to find all trajectories that are withing a distance d of a given query static point Q_p over the query time period [Q_{start},Q_{end}]. The query is continuous, such that the trajectories found may be within the distance threshold d for a subinterval of the query time [Q_{start},Q_{end}]. For example, for a query Q_1 with a query time interval of [0,1], the search may return T_1 between [0.1,0.3] and T_2 between [0.2,0.6].
- DistTrajSearch_Q$_t$(D,Q_t,Q_{start},Q_{end}, d) is similar but searches for trajectories that are within a distance d of a query trajectory Q_t.

DistTrajSearch_Q$_p$ is a simpler case of DistTrajSearch_Q$_t$. We focus on developing an efficient approach for DistTrajSearch_Q$_t$, which can be reused as is for DistTrajSearch_Q$_p$.

In all that follows, we consider *in-memory databases*, meaning that the database fits and is loaded in RAM once and for all. Distance threshold queries are relevant for scientific applications that are typically executed on high-performance computing platforms such as clusters. It is thus possible to partition the database and distribute it over a (possibly large) number of compute nodes so that the application does not require disk accesses. It is straightforward to parallelize distance threshold searches (replicate the query across all nodes, search the MOD independently at each node, and aggregate the obtained results). We leave the topic of parallel searches for future work. Instead we focus on efficient in-memory processing at a single compute node, which is challenging and yet necessary for achieving efficient parallel executions. Furthermore, as explained in Section IV, no criterion can be used to avoid index tree node accesses in distance threshold searches. Therefore, there are no possible I/O optimizations when (part of) the database resides on disk, which is another reason why we focus on the in-memory scenario.

IV. Trajectory Indexing

Given a distance threshold search for some query trajectory over some temporal extent, one considers all relevant query MBBs (those for the query trajectory segments). These query MBBs are *augmented* in all spatial dimensions by the threshold distance d. One then searches for the set of trajectory segment MBBs that overlap with the query MBBs, since these segments may be in the result set. Efficient indexing of the trajectory segment MBBs can thus lower query response time.

The most common approach is to store trajectory segments as MBBs in an index tree [12], [13], [14], [15]. Several index trees have been proposed (TB-tree [6], STR-tree [6], 3DR-tree [7]). Their main objective is to reduce the number of tree nodes visited during index traversals, using various pruning techniques (e.g., the *MINDIST* and *MINMAXDIST* metrics in [17]). While this is sensible for kNN queries, instead for distance threshold queries *there is no criterion for reducing the number of tree nodes that must be traversed*. This is because any MBB in the index that overlaps the query MBB may contain a line segment within the distance threshold, and thus must be returned as part of the candidate set.

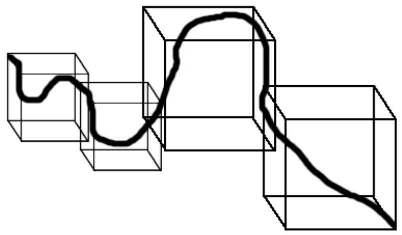


Figure 1. An example trajectory stored in different leaf nodes in a TB-tree.

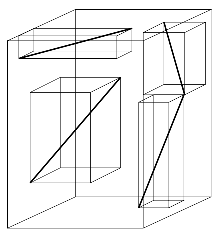


Figure 2. Four line segments belonging to three different trajectories within one leaf node of an R-tree.

Let us consider for instance the popular TB-tree, in which a leaf node stores only contiguous line segments that belong to the same trajectory and leaf nodes that store segments from the same trajectory are chained in a linked list. As a result, the TB-tree has high "temporal discrimination" (this terminology was introduced in [6]). Figure 1 shows a trajectory stored inside four leaf nodes within a TB-tree (each leaf node is shown as a bounding box). The curved and continuous appearance of the trajectory is because multiple line segments are stored together in each leaf node. By contrast, the R-tree simply stores in each leaf node trajectory segments that are spatially and temporally near each other, regardless of the individual trajectories. Figure 2 depicts an example with 4 segments belonging to 3 different trajectories that could be stored in a leaf node of an R-tree. For a distance threshold query, the number of TB-tree leaf nodes processed to perform the search could be arbitrarily high (since segment MBBs from many different trajectories can overlap the query MBB). Therefore, the TB-tree reduces the important R-tree property of overlap reduction; with an R-tree it may be sufficient to process only a few leaf nodes since each leaf node stores spatially close segments from multiple trajectories. For distance threshold queries, high spatial discrimination is likely to be more efficient than high temporal discrimination. Also, results in [6] show that the TB-tree performs better than the R-tree (for kNN queries) especially when the number of indexed entries is low; however, we are interested in large MODs (see Section III-A). We conclude that an R-tree index should be used for efficient distance threshold query processing.

V. Search Algorithm

We propose an algorithm, TrajDistSearch (Figure 3), to search for trajectories that are within a threshold distance of a query trajectory (defined as a set of connected trajectory segments over some temporal extent). All entry MBBs that overlap the query MBB are returned by the R-tree index and are then processed to determine the result set. More specifically, the algorithm takes as input an R-Tree index, T, a query trajectory, Q, and a threshold distance, d. It returns a set of time intervals annotated by trajectory ids, corresponding to the interval of time during which a particular trajectory is within distance d of the query trajectory. After initializing the result set to the empty set (line 2), the algorithm loops over all (augmented) MBBs that correspond to the segments of the query trajectory (line 3). For each such query MBB, the R-Tree index is searched to obtain a set of candidate entry MBBs that overlap the query MBB (line 4). The algorithm then loops over all the candidates (line 5) and does the following. First, given the candidate entry MBB and the query MBB, it computes an entry trajectory segment and a query trajectory segment that span the same time interval (line 6). The algorithm then computes the interval of time during which these two trajectory segments are within a distance d of each other (line 7). This calculation involves computing the coefficients of and solving a degree two polynomial [15]. If this interval is non-empty, then it is annotated with the trajectory id and added to the result set (line 10). The overall result set is returned once all query MBBs have been processed (line 14). Note that for a static point search Q.MBBSet (line 3) would consist of a single (degenerate) MBB with a zero extent in all spatial dimensions and some temporal extent, thus obviating the need for the outer loop. We call this algorithm PointDistSearch.

```
1:  procedure TrajDistSearch (R-Tree T, Query Q, double d)
2:      resultSet ← ∅
3:      for all querySegmentMBB in Q.MBBSet do
4:          CandidateSet ← T.Search(querySegmentMBB, d)
5:          for all candidateMBB in CandidateSet do
6:              (EntrySegment, QuerySegment) ← interpolate(
                        candidateMBB,querySegmentMBB)
7:              timeInterval ← calcTimeInterval(
                        EntrySegment,QuerySegment,d)
8:              if timeInterval ≠ ∅ then
9:                  result_id ← atomic_inc(global_index)
10:                 resultSet[result_id] ← resultsSet ∪ timeInterval
11:             end if
12:         end for
13:     end for
14:     return resultSet
15: end procedure
```

Figure 3. Pseudo-code for the TrajDistSearch algorithm (Section V).

VI. Initial Experimental Evaluation

A. Datasets

Our first dataset, *Trucks* [18], is used in other MOD works [13], [14], [15]. It contains 276 trajectories corresponding to 50 trucks that travel in the Athens metropolitan area for 33 days. This is a 3-dimensional dataset (2 spatial + 1 temporal). Our second dataset is a class of 4-dimensional datasets (3 spatial + 1 temporal), *Galaxy*. These datasets contain the trajectories of stars moving in the Milky Way's gravitational field (see Section III-A). The largest *Galaxy* dataset consists of 1,000,000 trajectory segments corresponding to 2,500 trajectories of 400 timesteps each. Distances are expressed in kiloparsecs (kpc). Our third dataset is a class of 4-dimensional synthetic datasets, *Random*, with trajectories generated via random walks. An adjustable parameter, α, is used to control whether the trajectory is a straight line ($\alpha = 0$) or a Brownian motion trajectory ($\alpha = 1$). We vary α in 0.1 increments to produce 11 datasets for datasets containing between ~1,000,000 and ~5,000,000 segments. Trajectories with $\alpha = 0$ spans the largest spatial extent and trajectories with $\alpha = 1$ are the most localized. All trajectories have the same temporal extent but different start times. Other synthetic datasets exist, such as GSTD [19]. We do not use GSTD because it does not allow for 3-dimensional spatial trajectories.

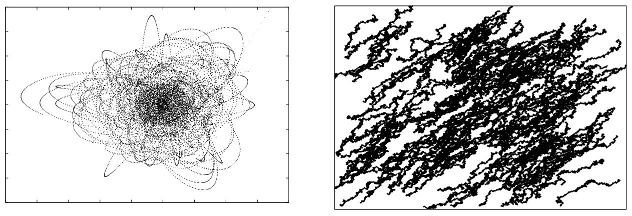

(a) (b)

Figure 4. (a) *Galaxy* dataset: a sample of 30 trajectories, (b) 200 trajectories in the *Random* dataset with $\alpha = 0.8$.

Figure 4 shows a 2-D illustration of the *Galaxy* and *Random* datasets. An illustration of *Trucks* can be found in previous works [13], [14]. Table I summarizes the main characteristics of each dataset. The *Galaxy* and *Random* datasets are publicly available [20].

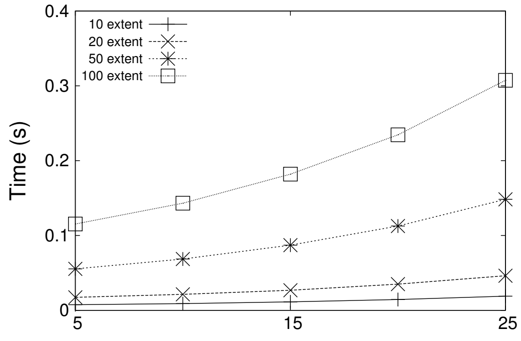
TABLE I. CHARACTERISTICS OF DATASETS

Dataset	Trajec.	Entries
Trucks	276	112152
Galaxy-200k	500	200000
Galaxy-400k	1000	400000
Galaxy-600k	1500	600000
Galaxy-800k	2000	800000
Galaxy-1M	2500	1000000
Random-1M ($\alpha \in \{0, 0.1, \ldots, 1\}$)	2500	997500
Random-2M ($\alpha = 1$)	5000	1995000
Random-3M ($\alpha = 1$)	7500	2992500
Random-4M ($\alpha = 1$)	10000	3990000
Random-5M ($\alpha = 1$)	12500	4987500

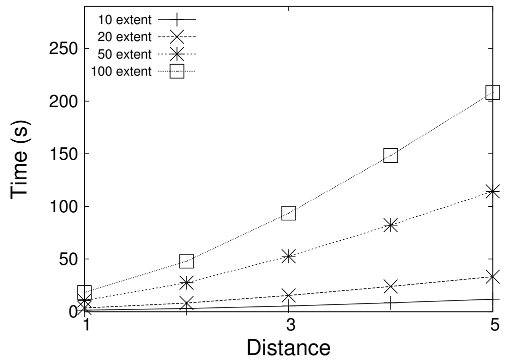
B. Experimental Methodology

We have implemented algorithm TRAJDISTSEARCH in C++, reusing an existing R-Tree implementation based on that initially developed by A. Guttman [5], and the code is available [21]. We execute this implementation on one core of a dedicated Intel Xeon X5660 processor, at 2.8 GHz, with 12 MB L3 cache and sufficient memory to store the entire index. We measure query response time averaged over 3 trials. The variation among the trials is negligible so that error bars in our results are not visible. We ignore the overhead of loading the R-Tree from disk into memory, which can be done once before all query processing.

C. Trajectory Search Performance



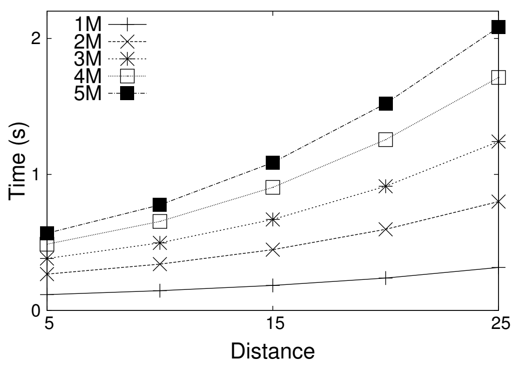
(a) *Random* $\alpha = 1$



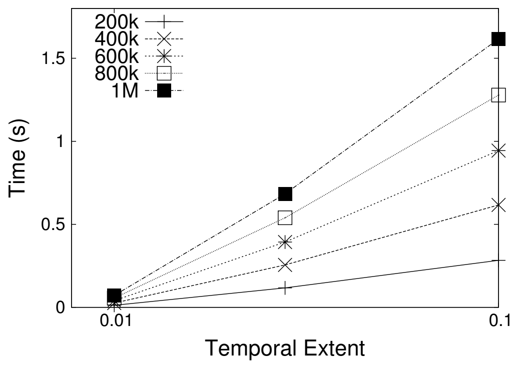
(b) Galaxy-1M

Figure 5. Query response time vs. threshold distance for 10%, 20%, 50% and 100% of the temporal extents of trajectories in S1 using the *Random*-1M $\alpha = 1$ dataset (a) and the *Galaxy*-1M dataset with search S2 (b).

We measure the query response time of TRAJDISTSEARCH for the following sets of trajectory searches:



(a) *Random* $\alpha = 1$



(b) Galaxy

Figure 6. (a) Response time vs. threshold distances for various numbers of segments in the index using search S3. (b) Response time vs. temporal extent for various numbers of segments in the index using search S4.

- S1: *Random*-1M dataset, $\alpha = 1$, 100 randomly selected query trajectories, processed for 10%, 20%, 50% and 100% of their temporal extents, with various query distances;
- S2: Same as S1 but for the *Galaxy*-1M dataset;
- S3: *Random*-1M, 2M, 3M, 4M and 5M datasets, $\alpha = 1$, 100 randomly selected query trajectories, processed for 100% of their temporal extent, with various query distances;
- S4: *Galaxy*-200k, 400k, 600k, 800k, 1M datasets, 100 randomly selected trajectories, processed with for 1%, 5% and 10% of their temporal extents, with a fixed query distance $d = 1$.

Figures 5 (a) and 5 (b) plot response time vs. query distance for S1 and S2 above. The response time increases slightly superlinearly with the query distance and with the temporal extents. In other words, the R-tree search performance degrades gracefully as the search is more extensive. Figures 6 (a) and (b) show response time vs. query distance for S3 and S4 above. The response time increases slightly superlinearly as the query distance increases for S3, and roughly linearly with the temporal extent increases for S4. Both these figures show results for various dataset sizes. An important observation is that the response time degrades gracefully as the datasets increase in size. More interestingly, note that for a fixed temporal extent and a fixed query distance, a larger dataset means a higher trajectory density, and thus a higher degree of overlap in the R-tree index. In spite of this increasing overlap,

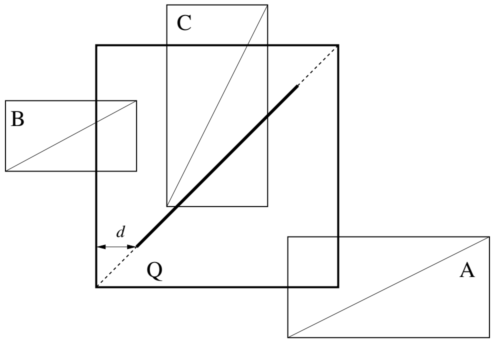


Figure 7. Three example entry MBBs and their overlap with a query MBB.

the R-tree still delivers good performance. We obtained similar results for PointDistSearch, which are omitted here.

VII. Trajectory Segment Filtering

The results in the previous section illustrate that TrajDistSearch maintains roughly consistent performance behavior over a range of query configurations (temporal extents, threshold distances, index sizes). In this and the next section, we explore approaches to reduce response time.

At each iteration our algorithm computes the moving distance between two line segments (line 7 in Figure 3). One can bypass this computation by "filtering out" those line segments for which it is straightforward (i.e., computationally cheap) to determine that they cannot possibly lie within distance d of the query. This filtering is applied to the segments once they have been returned by the index, and is thus independent of the indexing method.

Figure 7 shows an example with a query MBB, Q, and three overlapping MBBs, A, B, and C, that have been returned from the index search. The query distance d is indicated in the (augmented) query box so that the query trajectory segment is shorter than the box's diagonal. A contains a segment that is outside Q and should thus be filtered out. The line segment in B crosses the query box boundary but is never within distance d of the query segment and should be filtered out. C contains a line segment that is within a distance d of the query segment, and should thus not be filtered out. For this segment a moving distance computation must be performed (Figure 3, line 7) to determine whether there is an interval of time in which the two trajectories are indeed within a distance d of each other. The fact that candidate segments are returned that should in fact be ignored is inherent to the use of MBBs: a segment occupies an infinitesimal portion of its MBB's space. This issue is germane to MODs that store trajectories using MBBs.

In practice, depending on the dataset and the search, the number of line segments that should be filtered out can be large. Figure 8 shows the number of candidate segments returned by the index search and the number of segments that are within the query distance vs. α, for the *Random*-1M dataset, with 100 randomly selected query trajectories processed for 100% of their temporal extent. The fraction of candidate segments that are within the query distance is below 16.5% at $\alpha = 1$. In this particular example, an ideal filtering method would filter out more than 80% of the line segments.

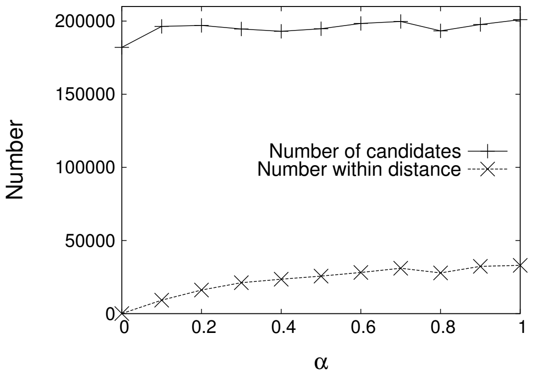


Figure 8. Total number of moving distance calculations vs. the number that are actually within a distance of 15 in the *Random*-1M datasets.

A. Two Segment Filtering Methods

After the query and entry line segments are interpolated so that they have the same temporal extent (Figure 3, line 6), various criteria may remove the candidate segment from consideration. We consider two filtering methods beyond the simple no filtering approach:

Method 1 – No filtering.
Method 2 – After the interpolation, check whether the candidate segment still lies within the query MBB. This check only requires floating point comparisons between spatial coordinates of the segment endpoints and the query MBB corners, and would occur between lines 6 and 7 in Figure 3. Method 2 would filter out A in Figure 7.
Method 3 – Considering only 2 spatial dimensions, say x and y, for a given query segment MBB compute the slope and the y-intercept of the line that contains the query segment. This computation requires only a few floating point operations and would occur in between lines 3 and 4 in Figure 3, i.e., in the outer loop. Then, before line 7, check if the endpoints of the candidate segment both lie more than a distance d above or below the query trajectory line. In this case, the candidate segment can be filtered out. This check requires only a few floating point operations involving segment endpoint coordinates and the computed slope and y-intercept of the query line. Method 3 would filter out both A and B in Figure 7.

Other computational geometry methods could be used for filtering, but these methods must be sufficiently fast (i.e., low floating point operation counts) if any benefit over Method 1 is to be achieved.

B. Filtering Performance

We have implemented the filtering methods in the previous section in TrajDistSearch and in this section we measure response times ignoring the R-tree search, i.e., focusing only on the filtering and the moving distance computation. We use the following distance threshold searches:

- S5: From the *Trucks* dataset, 10 trajectories are processed for 100% of their temporal extent.
- S6: From the *Galaxy*-1M dataset, 100 trajectories are processed for 100% of their temporal extent.
- S7: From the *Random*-1M datasets, 100 trajectories are processed for 100% of their temporal extent, with a fixed query distance $d = 15$.

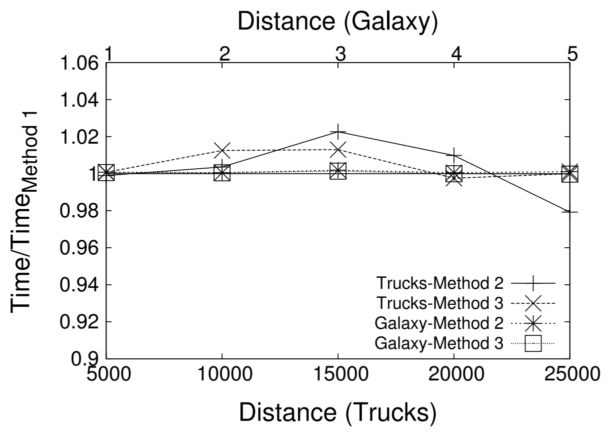


Figure 9. Performance improvement ratio of filtering methods for real datasets with S5 and S6, vs. query distance.

Figure 9 plots the relative improvement (i.e., ratio of response times) of using Method 2 and Method 3 over using Method 1 vs. the threshold distance for S5 and S6 above for the Galaxy and Trucks datasets. Data points below the $y = 1$ line indicate that filtering is beneficial. We see that filtering is almost never beneficial and can in fact marginally increase response time. Similar results are obtained for the Random dataset regardless of the α value.

It turns out that our methods filter only a small fraction of the line segments. For instance, for search S7 Method 2, resp. Method 3, filters out between 2.5% and 12%, resp. between 3.2% and 15.9%, of the line segments. Therefore, for most candidate segments the time spent doing filtering is pure overhead. Furthermore, filtering requires only a few floating point operations but also several if-then-else statements. The resulting branch instructions slow down executions (due to pipeline stalls) when compared to straight line code. We conclude that, at least for the datasets and searches we have used, our filtering methods are not effective.

One may envision developing better filtering methods to achieve (part of the) filtering potential seen in Figure 8. We profiled the execution of TRAJDISTSEARCH for searches S5, S6, and S7, with no filtering, and accounting both for the R-tree search and the distance computation. We found that the time spent searching the R-tree accounts for at least 97% of the overall response time. As a result, filtering can only lead to marginal performance improvements for the datasets and queries in our experiments. For other datasets and queries, however, the fraction of time spent computing distances could be larger. Nevertheless, given the results in this section, in all that follows we do not perform any filtering.

VIII. INDEX RESOLUTION

According to the cost model in [22], index performance depends on the number of nodes in the index, but also on the volume and surface area of the MBBs. One extreme is to store an entire trajectory in a single MBB as defined by the spatial and temporal properties of the trajectory; however, this leads to a lot of "wasted MBB space." Representing the object using multiple MBBs decreases the amount of empty space by storing the object in a series of consecutive multi-segment MBBs. The other extreme is to store each trajectory line segment in its own MBB, as done so far in this paper and in previous work on kNN queries [12], [13], [14], [15]. In this scenario, the volume occupied by the trajectory in the index is minimized, with the trade-off that the number of entries in the index will be maximized.

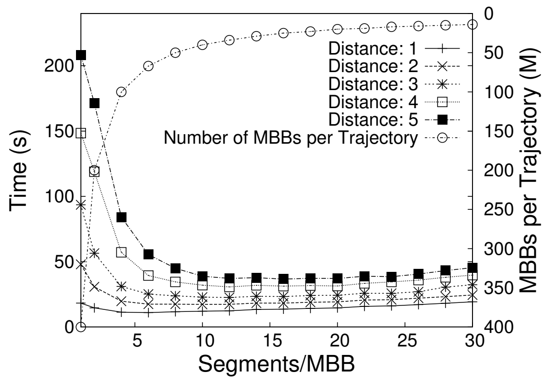
Assigning a fraction of a trajectory to a single MBB, as a series of line segments, increases overlap in the index, as the resulting MBB is larger in comparison to minimizing the volume of the MBBs by describing each individual trajectory line segment by its own MBB. As a result, an index search can return a portion of a trajectory that does not overlap the query, leading to increased overhead when processing the candidate set of line segments returned by the index. However, the number of entries in the index is reduced, thereby reducing tree traversal time. To explore the tradeoff between number of nodes in the index, the amount of wasted volume required by a trajectory, index overlap, and the overhead of processing candidate trajectory segments, in this section we evaluate three strategies for splitting trajectories into a series of consecutive MBBs, implemented as an array of references to trajectory segments (leading to one extra indirection when compared to assigning a single segment per MBB). We evaluate performance experimentally by splitting the trajectories, and then creating their associated indexes, where the configuration with the lowest query response time is highlighted. We leave analytical performance models of trajectory splitting methods for future work.

A. *Static Temporal Splitting*

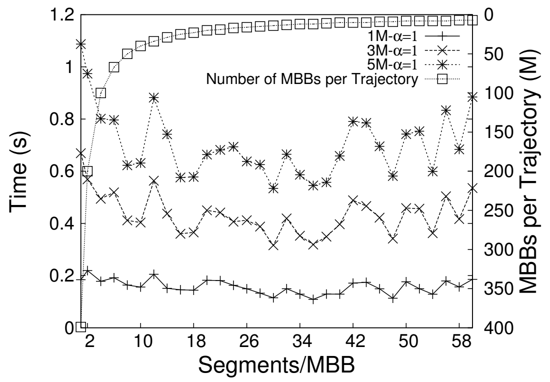
Assuming it is desirable to ensure that trajectory segments are stored contiguously, we propose a simple method. Given a trajectory of n line segments, we split the trajectory by assigning r contiguous line segments per MBB, where r is a constant. Therefore, the number of MBBs, M that represent a single trajectory is $M = \lceil \frac{n}{r} \rceil$. By storing segments contiguously, this strategy leads to high temporal locality of reference, which may be important for cache reuse in our in-memory database, in addition to the benefits of the high spatial discrimination of the R-tree (see Section IV).

Figure 10 plots response time vs. r for the S6 (*Galaxy* dataset) and S7 (*Random* dataset) searches defined in Section VII-B. For S6, 5 different query distances are used, while for S7 the query distance is fixed as 15 but results are shown for various dataset sizes for $\alpha = 1$. The right y-axis shows the number of MBBs used per trajectory. The data points at $r = 1$ correspond to the original implementation (rather than the implementation with $r = 1$, which would include one unnecessary indirection).

The best value for r depends on the dataset and the search. For instance, in the *Galaxy*-1M dataset (S6) using 12 segments per MBB leads to the best performance (or $M = 34$). We note that picking a r value in a large neighborhood around this best value would lead to only marginally higher query response times. In general, using a small value of r can lead to high response times, especially for $r = 1$ (or $M = 400$). For instance, for S6 with a query distance of 5, the response time with $r = 1$ is above 208 s while it is just above 37 s with $r = 12$. With $r = 1$ the index is large and thus time-consuming to search. A very large r value does not lead to the



(a) *Galaxy*-1M



(b) *Random* datasets

Figure 10. Static Temporal Splitting: Response time vs. r for (a) S6 for the *Galaxy*-1M dataset for various query distances; and (b) S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15. The number of MBBs per trajectory, M, is shown on the right vertical axis.

lowest response time since in this case many of the segments returned from the R-tree search are not query matches. Finally, results in Figure 10 (a) show that the advantage of assigning multiple trajectory segments per MBB increases as the query distance increases. For instance, for a distance of 2 using $r = 12$ decreases the response time by a factor 2.76 when compared to using $r = 1$, while this factor is 5.6 for a distance of 5. Note that the difference in response times between Figure 10 (a) and (b) are largely due to more queries within d in *Galaxy* in comparison to *Random* for the query distances selected.

B. Static Spatial Splitting

Another strategy consists in ordering the line segments belonging to a trajectory spatially, i.e., by sorting the line segments of a trajectory by the x, y, and z values of the segment's origin. We then assign r segments per trajectory into each MBB, as in the previous method. With such spatial grouping, the line segments are no longer guaranteed to be temporally contiguous in their MBBs, but reduced index overlap may be achieved. Figure 11 plots response time vs. r for the S7 (*Random* dataset) searches. We see that there is no advantage to assigning multiple trajectory segments to an MBB over assigning a single line segment to a MBB ($r = 1$ in the plot). When comparing with results in Figure 10 (b) we find that spatial splitting leads to query response times higher by several factors than that of temporal splitting.

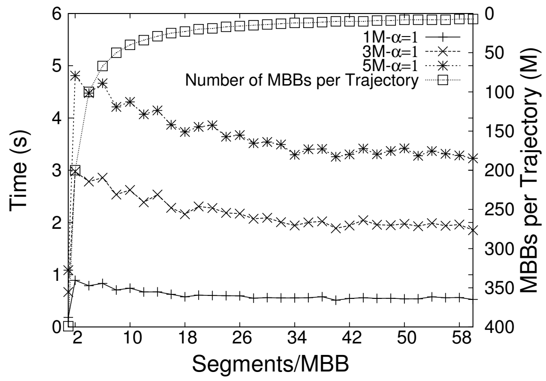


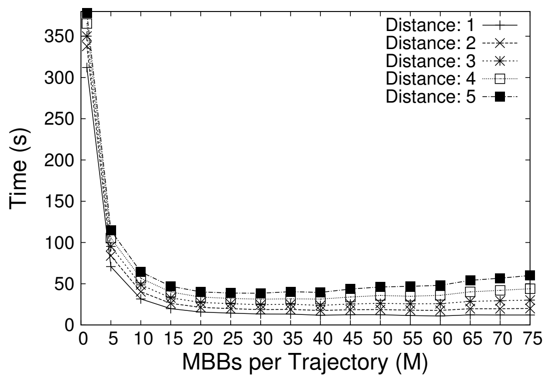
Figure 11. Static Spatial Splitting: Response time vs. r using S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15. The number of MBBs per trajectory, M, for each data point is shown on the rightmost vertical axis.

C. Splitting to Reduce Trajectory Volume

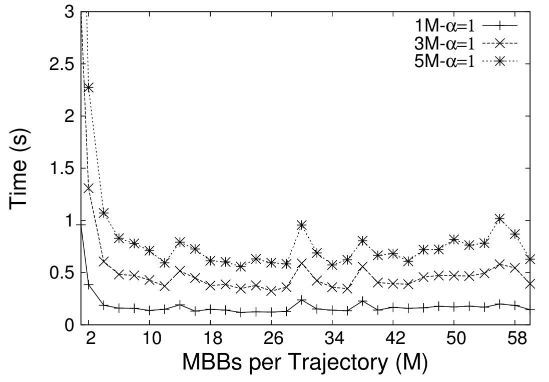
The encouraging results in Section VIII-A suggest that using an appropriate trajectory splitting strategy can lead to performance gains primarily by exploiting the trade-off between the number of entries in the index and the amount of wasted space that leads to higher index overlap. More sophisticated methods can be used. In particular, we implement the heuristic algorithm *MergeSplit* in [23], which is shown to produce a splitting close to optimal in terms of wasted space. *MergeSplit* takes as input a trajectory, T, as a series of l line segments, and a constant number of MBBs, M. As output, the algorithm creates a set of M MBBs that encapsulate the l segments of T. The pseudocode of *MergeSplit* is as follows:

1) For $0 \leq i < l$ calculate the volume of the merger of the MBBs that define l_i and l_{i+1} and store the resulting series of MBBs and their volumes.
2) To obtain M MBBs, repeat $(l-1)-(M-1)$ times and merge consecutive MBBs that produce the smallest volume increase at each step. After the first iteration, there will be $l-2$ initial MBBs describing line segments, and one MBB that is the merger of two line segment MBBs.

Figure 12 shows response time vs. M for S6 (*Galaxy* dataset) and S7 (*Random* dataset). Compared to static temporal splitting, which has a constant number of segments, r per MBB, *MergeSplit* has a variable number of segments per MBB. From the figure, we observe that for the *Galaxy*-1M dataset (S6), $M = 30$ leads to the best performance. Comparing *MergeSplit* to the static temporal splitting (Figures 10 and 12 (a)), the best performance for the S6 (*Galaxy* dataset) is achieved by the static temporal splitting. For S7, the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets, *MergeSplit* is only marginally better than the static temporal splitting (Figures 10 and 12 (b)). This is surprising, given that the total hypervolume of the entries in the index for a given M across both splitting strategies is higher for the simple static temporal splitting, as it makes no attempt to minimize volume. Therefore, the trade-off between the number of entries and overlap in the index cannot fully explain the performance of these trajectory splitting strategies for distance threshold queries.

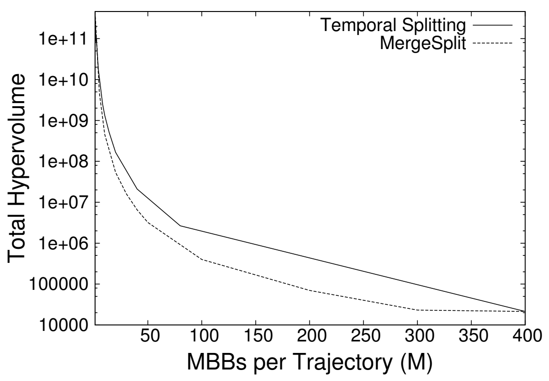


(a) *Galaxy*-1M

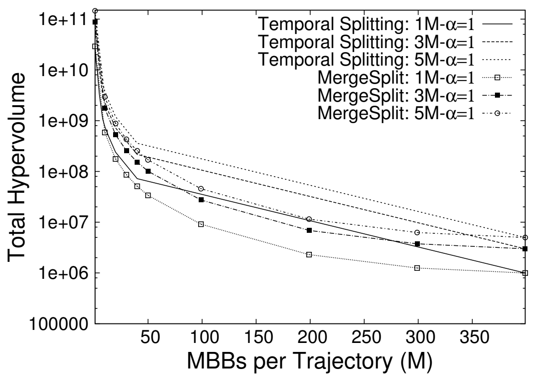


(b) *Random* datasets

Figure 12. Greedy Trajectory Splitting: Response time vs. M for (a) S6 for the *Galaxy*-1M dataset for various query distances; and (b) S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15.



(a) *Galaxy*-1M



(b) *Random*-1M

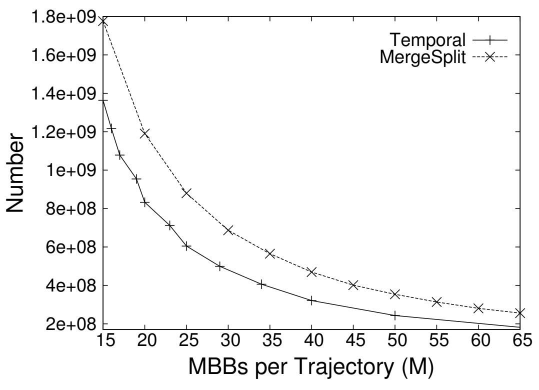
Figure 13. Total hypervolume vs. M for the static temporal splitting strategy and *MergeSplit*. (a)for the *Galaxy*-1M dataset (S6); and (b) for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets (S7).

D. Discussion

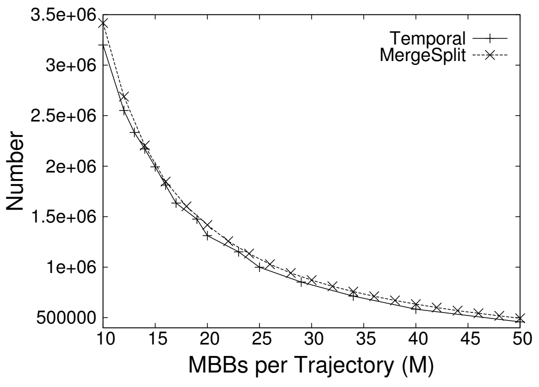
A good trade-off between the number of entries in the index and the amount of index overlap can be achieved by selecting an appropriate trajectory splitting strategy. However, comparing the results of the simple temporal splitting strategy (Section VIII-A) and *MergeSplit* (Section VIII-C), we find that volume minimization did not significantly improve performance for S7, and led to worse performance for S6. In Figure 13, we plot the total hypervolume vs. M for the *Galaxy*-1M (S6) and the *Random*-1M, 3M, and 5M $\alpha = 1$ (S7) datasets. $M = 1$ refers to placing an entire trajectory in a single MBB, and the maximum value of M refers to placing each individual line segment of a trajectory in its own MBB. For the static temporal splitting strategy, $M = 34$ leads to the best performance for the *Galaxy*-1M dataset (S6), whereas this value is $M = 30$ for *MergeSplit*. The total hypervolume of the MBBs in units of kpc^{3}Gyr for the static temporal grouping strategy at $M = 34$ is 3.6×10^7, whereas for *MergeSplit* at $M = 30$, it is 1.62×10^7, or the MBBs require 55% less volume. Due to the greater volume occupied by the MBBs, index overlap is much higher for the static temporal splitting strategy. Figure 14 (a) plots the number of overlapping line segments vs. M for S6 with $d = 5$. From the figure, we observe that independently of M, *MergeSplit* returns a greater number of candidate line segments to process than the simple temporal splitting strategy. *MergeSplit* attempts to minimize volume; however, if an MBB contains a significant fraction of the line segments of a given trajectory, then all of these segments are returned as candidates. The simple temporal grouping strategy has an upper bound (r) on the number of segments returned per overlapping MBB and thus can return fewer candidate segments for a query, despite occupying more volume in the index. For in-memory distance threshold queries, there is a trade-off between a trajectory splitting strategy that has an upper bound on the number of line segments per MBB, and index overlap, characterized by the volume occupied by the MBBs in the index. This is in sharp contrast to other works that focus on efficient indexing of spatiotemporal objects in traditional out-of-core implementations where the index resides partially in-memory and on disk, and therefore volume reduction to minimize index overlap is necessary to minimize disk accesses (e.g., [23]).

E. Performance Considerations for In-memory and Out-of-Core Implementations

The focus of this work is on in-memory distance threshold queries; however, most of the literature on MODs assume out-of-core implementations, where the number of node accesses are used as a metric to estimate I/O activity. Figure 15 shows the number of node accesses vs. M for both of the static temporal splitting strategy and *MergeSplit*. We find that for the *Galaxy*-1M dataset (S6) with $d = 5$, there are a comparable number of node accesses for both trajectory splitting methods. However, for S7 (*Random*-1M), on average, trajectory splitting with *MergeSplit* requires fewer node accesses and may perform significantly better than the simple temporal splitting

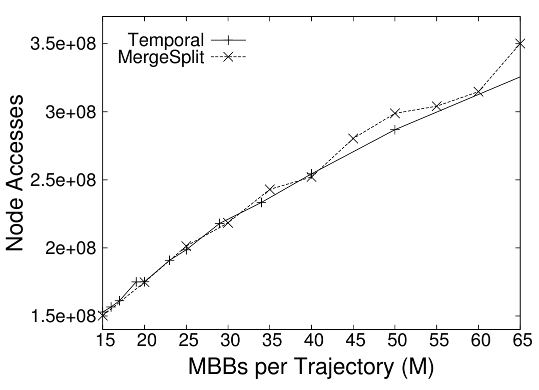


(a) *Galaxy*-1M

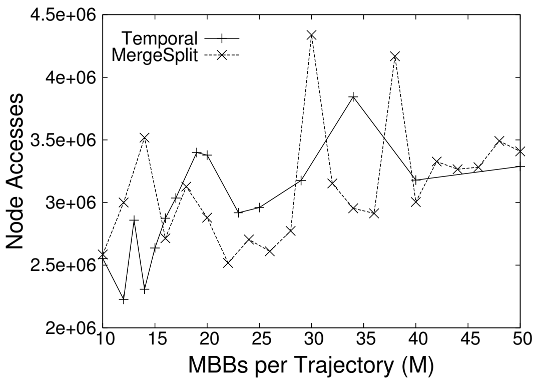


(b) *Random*-1M

Figure 14. Total number of overlapping segments vs. M for the static temporal splitting strategy and *MergeSplit*. (a) S6 for the *Galaxy*-1M dataset with $d = 5$; and (b) S7 for the *Random* $\alpha = 1$ dataset with $d = 15$.



(a) *Galaxy*-1M



(b) *Random*-1M

Figure 15. Node Accesses vs. M for the static temporal splitting strategy and *MergeSplit*. (a) S6 for the *Galaxy*-1M dataset with $d = 5$; and (b) S7 for the *Random* $\alpha = 1$ dataset with $d = 15$.

strategy in an out-of-core implementation. For example, in Figure 15 (b) some values of M have a significantly higher number of node accesses, such as values around 14, 30, 38, due to the idiosyncrasies of the data, and resulting index overlap. However, as we demonstrated in Section VIII-D, distance threshold queries in the context of in-memory databases also benefit from reducing the number of candidate line segments returned, and this is not entirely volume contingent. Therefore, methods that consider volume reduction, such as the *Merge-Split* algorithm of [23], or other works that consider volume reduction in the context of query sizes, such as [24], may not be entirely applicable to distance threshold queries.

A single metric cannot capture the trade-offs between the number of entries in the index, volume reduction, index overlap, and the number of candidate line segments returned (germane to distance threshold queries). However, for *Galaxy*-1M (S6), a value of $M = 34$ and $M = 30$ lead to the best query response time for the temporal splitting strategy and *MergeSplit*, respectively (Figures 10 (a) and 12 (a)). Figure 16 shows the number of L1 cache misses vs. M for S6 with $d = 5$. The number of cache misses was measured using PAPI [25]. The best values of M in terms of query response time for both of the trajectory splitting strategies ($M = 34$ and $M = 30$) roughly correspond to a value of M that minimizes cache misses. Thus, cache misses appear to be a good indicator of relative query performance under different indexing methods. Future work for in-memory distance threshold queries should focus on improved cache reuse through temporal locality of

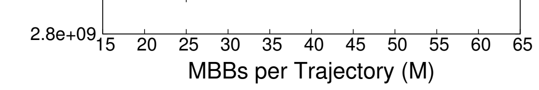


Figure 16. L1 cache misses vs. M for the static temporal splitting strategy and *MergeSplit* for the *Galaxy*-1M dataset (S6) with $d = 5$.

reference (which is in part obtained by storing segments contiguously within a single MBB).

IX. Conclusion

In-memory distance threshold queries for trajectory and point queries on moving object trajectories are significantly different from the well-studied kNN searches [12], [13], [14], [15]. We made a case for using an R-tree index to store trajectory segments, and found it to perform robustly for two real world datasets and a synthetic dataset. We focused on 4-D datasets (3 spatial + 1 temporal) while other works only consider 3-D datasets [12], [13], [14], [15].

We found the popular "search and refine" strategy to be ineffective for distance threshold searches since many segments returned by the index search must be excluded from the result set. We have proposed computationally inexpensive solutions to filter out candidate segments, but found that they have poor selectivity. A more promising direction for reducing query response time is to reduce the time spent traversing the tree index. We demonstrated that efficiently splitting trajectories is beneficial because the penalty for the increased index overlap is offset by the reduction in number of index entries. We find that for in-memory distance threshold queries, the number of line segments returned per overlapping MBB has an impact on performance, where attempts to reduce the volume of the MBBs that store a trajectory may be at cross-purposes with returning a limited number of candidate segments per overlapping MBB. Therefore, trajectory splitting methods that focus on volume reduction are not necessarily preferable to a simple and bounded grouping of line segments in MBBs.

A future direction is to explore trajectory splitting methods that achieve volume reduction while bounding the number of MBBs used per trajectory. Another direction is to investigate non-MBB-based data structures to index line segments, such as that in [26]. Finally, we plan to develop algorithms for parallel processing of in-memory distance threshold queries both for shared- and distributed-memory executions.

One may wonder whether the idea of assigning multiple segments to an MBB is generally applicable, and in particular for kNN searches on trajectories [12], [13], [14], [15]. The kNN literature focuses on pruning strategies and associated metrics that require a high resolution index, thus implying storing a single trajectory segment in an MBB. Furthermore, kNN query processing algorithms maintain a list of nearest neighbors over a time interval, which would lead to greater overhead if multiple segments were stored per MBB. Therefore, the approach of grouping line segments together in a single MBB may be ineffective for kNN queries. An interesting problem is to reconcile the differences between both types of queries in terms of index resolution.

Acknowledgments

This paper has benefited from the insightful comments of Lipyeow Lim. This material is based upon work supported by the National Aeronautics and Space Administration through the NASA Astrobiology Institute under Cooperative Agreement No. NNA08DA77A issued through the Office of Space Science, and by NSF Award CNS-0855245.

References

[1] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "A data model and data structures for moving objects databases," in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 2000, pp. 319–330.

[2] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, "A foundation for representing and querying moving objects," ACM Trans. Database Syst., vol. 25, no. 1, 2000, pp. 1–42.

[3] S. Arumugam and C. Jermaine, "Closest-point-of-approach join for moving object histories," in Proc. of the 22nd Intl. Conf. on Data Engineering, 2006, pp. 86–95.

[4] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," Proc. VLDB Endow., vol. 1, no. 1, Aug. 2008, pp. 1068–1080.

[5] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1984, pp. 47–57.

[6] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Proc. for Moving Object Trajectories," in Proc. of the 26th Intl. Conf. on Very Large Data Bases, 2000, pp. 395–406.

[7] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," in Proc. of the Intl. Conf. on Multimedia Computing and Systems, 1996, pp. 441–448.

[8] V. P. Chakka, A. Everspaugh, and J. M. Patel, "Indexing large trajectory data sets with SETI," in Proc. of Conference on Innovative Data Systems Research, 2003, pp. 164–175.

[9] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets," in Proc. of the 26th Intl. Conf. on Data Engineering, 2010, pp. 109–120.

[10] M. R. Vieira, P. Bakalov, and V. J. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in Proc. of the 17th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems, 2009, pp. 286–295.

[11] Z. Li, M. Ji, J.-G. Lee, L.-A. Tang, Y. Yu, J. Han, and R. Kays, "Movemine: Mining moving object databases," in Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data, 2010, pp. 1203–1206.

[12] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Nearest neighbor search on moving object trajectories," in Proc. of the 9th Intl. Conf. on Advances in Spatial and Temporal Databases, 2005, pp. 328–345.

[13] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Algorithms for Nearest Neighbor Search on Moving Object Trajectories," Geoinformatica, vol. 11, no. 2, 2007, pp. 159–193.

[14] Y.-J. Gao, C. Li, G.-C. Chen, L. Chen, X.-T. Jiang, and C. Chen, "Efficient k-Nearest-Neighbor Search Algorithms for Historical Moving Object Trajectories," J. Comput. Sci. Technol., vol. 22, no. 2, 2007, pp. 232–244.

[15] R. H. Güting, T. Behr, and J. Xu, "Efficient k-nearest neighbor search on moving object trajectories," The VLDB Journal, vol. 19, no. 5, 2010, pp. 687–714.

[16] M. G. Gowanlock, D. R. Patton, and S. M. McConnell, "A Model of Habitability Within the Milky Way Galaxy," Astrobiology, vol. 11, 2011, pp. 855–873.

[17] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1995, pp. 71–79.

[18] http://www.chorochronos.org/, accessed 5-February-2014.

[19] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento, "On the Generation of Spatiotemporal Datasets," in Proc. of the 6th Intl. Symp. on Advances in Spatial Databases, 1999, pp. 147–164.

[20] http://navet.ics.hawaii.edu/%7Emike/datasets/DBKDA2014/datasets.zip, accessed 12-February-2014.

[21] http://www.superliminal.com/sources/sources.htm, accessed 5-February-2014.

[22] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer, "Towards an analysis of range query performance in spatial data structures," in Proc. of the 12th Symp. on Principles of Database Sys., 1993, pp. 214–221.

[23] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, "Efficient indexing of spatiotemporal objects," in Proc. of the 8th Intl. Conf. on Extending Database Technology: Advances in Database Technology, 2002, pp. 251–268.

[24] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, "A trajectory splitting model for efficient spatio-temporal indexing," in Proc. of the 31st Intl. Conf. on Very Large Data Bases, 2005, pp. 934–945.

[25] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in Proc. of the Department of Defense HPCMP Users Group Conf., 1999, pp. 7–10.

[26] E. Bertino, B. Catania, and B. Shidlovsky, "Towards Optimal Indexing for Segment Databases," in Proc. of the 6th Intl. Conf. on Advances in Database Technology, 1998, pp. 39–53.