Efficient Multi-range Query Processing on Trajectories

Munkh-Erdene Yadamjav[1], Farhana M. Choudhury[1], Zhifeng Bao[1], and Hanan Samet[2]

[1] School of Science, RMIT University, Melbourne, Australia
[2] Department of Computer Science, University of Maryland, College Park, Maryland 20742

Abstract. With the widespread use of devices with geo-positioning technologies, an unprecedented volume of trajectory data is becoming available. In this paper, we propose and study the problem of multi-range query processing over trajectories, that finds the trajectories that pass through a set of given spatio-temporal ranges. Such queries can facilitate urban planning applications by finding traffic movement flows between different parts of a city at different time intervals. To our best knowledge, this is the first work on answering multi-range queries on trajectories. In particular, we first propose a novel two-level index structure that preserves both the co-location of trajectories, and the co-location of points within trajectories. Next we present an efficient query processing algorithm that employs several pruning techniques at different levels of the index. The results of our extensive experimental studies on two real datasets demonstrate that our approach outperforms the baseline by 1 to 2 orders of magnitude.

Keywords: Spatio-temporal index · Multi-range query · Spatial database

1 Introduction

With the proliferation of GPS enabled devices, the spatio-temporal positions of the moving objects (e.g., cars, public transports, pedestrians, etc.), known as trajectory data, are being generated at an unprecedented rate.

Such information facilitates effective planning of urban area and intelligent infrastructure management. As an example, suppose that an urban planning authority wants to find the trajectories that involve travel from Richmond (a suburb in Melbourne) to the Melbourne city center via Kensington (another suburb in Melbourne) during office start hours, i.e., 7 am to 9.30 am. The result of such queries can facilitate the transportation authority to explore movement flows between different parts of a city at different time intervals and identify traffic congestion to improve road infrastructure and traffic throughput.

In this paper, given a set of spatio-temporal ranges, we address the problem of finding the trajectories that pass through each of the query ranges, where a query range consists of a spatial region (e.g., a rectangle or a circle) and a time interval. In this work, we consider both the spatial-only and spatio-temporal trajectories. We denote this problem as a *Trajectory multi-range query* (*MRQ*). The size of the spatio-temporal query range can be arbitrary and does not have any pre-defined hierarchy [11]. Thus, it is difficult to precompute the intersections of all possible spatial range combinations.

There are several studies on trajectory data indexing [1, 2, 4, 6, 20] and processing different types of queries, such as finding regions of interest [18], popular routes [3, 9, 16], similarity etc. In the literature, the studies on answering *trajectory range queries* to find the trajectories that pass through a single spatial region during a specific time interval [2, 10, 12, 19, 21, 22] are the closest to our work. However, the indexes and approaches for single range queries are not designed to capture the trajectory movements from one region to another, and thus they suffer from several drawbacks to answer a multi-range trajectory query (see Section 3.2 for details).

To overcome the drawbacks of existing studies, we propose a novel two-level index structure to maintain the trajectory movement relation between different regions and time intervals. The key idea of our index structure is to store the close-by trajectories together in the first level of the index, and maintain the close-by points of the trajectories together in the second level of the index. Such structure enables us to retrieve only the necessary trajectories that pass through all the query ranges in the given query time intervals efficiently. We denote this index as the ***Lo****cation preserving two-level* ***T****rajectory* ***I****ndex* (*LoTI*). We propose several pruning steps over this index as our query processing approach.

The contributions of the paper are as follows:

- We propose and study the problem of efficiently processing multi-range queries on trajectories that can be beneficial to many real life applications.
- We propose a novel two-level index structure, *LoTI* (Section 4.1) that supports storing the co-located trajectories together, and at the same time, stores the co-located points of trajectories together. We present several strategies to prune the unnecessary trajectories and points of the trajectories efficiently using this structure (Section 4.2).
- We evaluate our algorithms through an extensive experimental study on real datasets. The results demonstrate both the efficiency of our algorithm by 1 to 2 orders of magnitude over a baseline solution (Section 5).

2 Related Work

In this section, we review the closely related studies, specifically in the area of trajectory indexing and query processing. Since many of the trajectory indexes use variants of the traditional spatial indexes, and there is no index structure that can directly support *MRQ*, we start this section by briefly reviewing the spatial and spatio-temporal indexes that are used in the baseline and our proposed method.

2.1 Conventional Index Structures

The *R-Tree* [5] is a tree structure where the *Minimum Bounding Rectangle* (MBR) of the root covers the region containing all spatial data objects. Every node has a maximum capacity, where the leaf nodes contain pointers to the actual data objects and the non-leaf nodes cover the MBRs of all of its child nodes. Many researchers (e.g., [22]) propose a 3D R-tree where time is treated as the third dimension to index spatio-temporal data.
The *Bucket Quadtree* [17] is an adaptive form of the uniform grid data structure that handles spatial data of an arbitrary distribution. There are a number of quadtree variants,

most of which repeatedly partition the underlying space into four equal-sized regions when (i) the number of objects in the space exceeds a threshold value (e.g., bucket quadtree for points [17] or PMR quadtree [7]), or (ii) when the underlying spanned space is not homogeneous (e.g., the region quadtree [8]).

The *Bucket Octree* [17] is a three-dimensional version of a bucket quadtree that handles three-dimensional data or spatio-temporal data by treating time as the third dimension where the space is partitioned into eight equal-sized cube regions.

2.2 Trajectory Index Structures

A number of spatial and spatio-temporal index structures have been proposed over the past decades to index trajectories. These indexes can be broadly categorized as (i) point-based indexes, and (ii) trajectory based indexes.

Point-based indexes. Several approaches propose to index trajectories, and spatio-temporal objects in general where the structure is constructed purely based on the locality of the points. As there are multiple traditional structures to index a set of points based on their locality (e.g., R-trees using MBRs or quadtrees using space partitioning), all the points of trajectories in a dataset are stored using such an index in point-based indexing techniques [6, 10, 19–21]. To maintain the relation between the points and trajectories (i.e., which point belongs to which trajectory), either an auxiliary lookup structure is maintained, or the trajectory identifiers are embedded into the point data.

The studies [6, 20] both use variants of R-tree to store the points of trajectories. The *HR-tree* [10] stores a separate R-Tree for each timestamp to allow queries on the past states of a spatial point database. Spatial objects that do not move between consecutive timestamps can result in a substantial storage overhead. The HR-tree organizes consecutive trees to share the same branches to avoid indexing stationary objects multiple times. However, the entire node is replicated when only one object updates its location. The *Start/End timestamp B-tree (SEB-tree)* [19] partitions the search space into zones and the 'zoning information' of objects is stored in the database. An object requests an update when it enters into a new zone. A separate index is constructed for each zone based on the objects that moved in, moved out, or are currently in that zone. All objects in one zone are indexed by start and end timestamps in the corresponding B-tree. This is similar to the active object data structure used in plane-sweep solutions [14]. The *MV3R-tree* [21] proposes to use a multi-version R-tree (MVR-tree) to index the data and all leaf nodes of the MVR-tree are used to build an auxiliary 3D R-tree. The idea of combining two different structures is to improve the query efficiency by choosing the appropriate structure. The MVR-tree is chosen for timestamp queries while the 3D R-tree is used to perform long time interval queries. A threshold parameter defines which structure is to be chosen to conduct short time interval queries.

In point based trajectory index structures, the points that are close in space are stored close-by, but points of the same trajectory may not be stored close-by. Thus the locality of trajectories are not preserved in such structures.

Trajectory-based indexes. In contrast to the point-based indexes, trajectory-based indexes [2, 12] consider the relevance of points while indexing the data. Points from the same trajectory are more likely to be stored close-by, and in some structures, all the points of the same trajectory are stored together.

The *Scalable and Efficient Trajectory Index (SETI)* [2] assumes that a spatial dimension does not change as frequently as a temporal dimension. SETI partitions the search space into static and non-overlapping cells. An in-memory structure is used to store the last location of each object and to compute a trajectory segment when a new location update comes. Long segments that span more than one spatial cell are split at the cell boundaries. A time interval that covers all segments in a cell is computed and used to create a temporal R-tree of each cell. The *STR-tree* [12] extends R-tree to store line segments of trajectories by preserving trajectory locality partially, where the line segments of the same trajectory are more likely to be stored together. While the insertion criterion of the R-tree is based on the least enlargement of the bounding rectangles, the STR-tree introduces an additional insertion criterion based on the least number of nodes required to store the line segments of the same trajectory at different levels of the tree. The *TB-tree* [12] ensures that a leaf node contains only the line segments that belong to the same trajectory. All the leaf nodes that store a trajectory are linked by forward and backward pointers to reduce the time to retrieve a complete trajectory. In this structure, as the line segments of different trajectories that are close to each other are not stored together, their experimental results show that TB-tree has a higher cost than STR-tree for spatial range queries.

Other index structures. Other index structures, e.g., *TrajTree* [15], *SharkDB* [24] are also proposed to efficiently store trajectories. The idea of TrajTree [15] is to compute edit distances between trajectories by segmentation, which is not the focus of our problem. SharkDB [24] is an in-memory column oriented timestamped trajectory storage. This index can support k nearest neighbor queries and range queries in the spatio-temporal domain, but cannot be directly applied to solve multiple range queries.

2.3 Trajectory Query Processing

There are several studies that answer different types of queries on trajectory data [13, 18, 23, 25]. The queries can be broadly categorized as (i) range queries, (ii) similarity based queries (k nearest neighbor, reverse k nearest neighbor queries, etc.). As the range queries are the closest to our problem, we focus on reviewing the studies on range queries in spatial and spatio-temporal databases, specifically on trajectory data.

In general, given a spatial range as a rectangle window or a circle, a range query on trajectories finds all the trajectories that pass through that range. For spatio-temporal range queries, a time interval is also given where a timestamp of the resulting trajectories in that region needs to fall in that time interval as well. As a range query is one of the most common search problems, all of the above mentioned point based indexing [6, 10, 19–21] and trajectory based indexing [2, 12] studies can answer this query. If the index is point based, the points that are inside the query range are retrieved, and then the auxiliary structure or the embedded trajectory identifiers are obtained to finally get the trajectories as the result of the range query. In trajectory-based indexes, the nodes that do not intersect with the query range are pruned, and then a further pruning is applied based on the intersection of the trajectory line segments with the query range. However, all of these approaches are designed to answer individual range queries. If these approaches are used to answer multiple range queries, many irrelevant trajectories and trajectory points are retrieved (details in Section 3.2).

3 Problem formulation and the baseline

3.1 Problem formulation

As we consider both the spatial-only and spatio-temporal trajectories in this work, we present our problem formulation for both settings.

Let T be a set of spatio-temporal trajectories where each $t \in T$ is a sequence of tuples of the form $\langle loc, \tau \rangle$. Here, loc is a point location and τ is a timestamp associated with that location. For spatial-only trajectories, each $t \in T$ is a sequence of locations of the form $t = (loc_1, loc_2, \ldots, loc_m)$ where m is the number of points in the trajectory.

Definition 1. ***Trajectory Range Query***. *Given a trajectory dataset T, a spatial range in space (e.g., a rectangle or a circle) and a time interval, a Trajectory Range Query returns all trajectories that intersect with both the spatial range and the time interval.*

Definition 2. ***Trajectory Multi-Range Query (MRQ)***. *Given a trajectory dataset T, a sequence of spatio-temporal ranges $\{\langle R_1, I_1 \rangle, \langle R_2, I_2 \rangle, \ldots\}$, where R_i is a spatial region and I_i is a time interval, a Trajectory Multi-Range Query returns all trajectories from T that intersect with each region R_i in the corresponding time interval I_i.*

For the spatial-only case, only the spatial ranges are given. An MRQ query can be considered as a sequence of independent range queries. Based on this observation, we present a baseline approach in the following to answer the *MRQ* using existing methods.

3.2 Baseline Approach

As there are multiple studies that can find trajectories intersecting a single spatial region in a given time interval (details in Section 2), we can apply the following steps to find the solution of an *MRQ* in a straightforward way - (1) Find the answer of the single range queries individually using an existing approach. These are the set of candidate trajectories; (2) Compute the intersection of the individual responses to obtain resulting trajectories that intersect each of the ranges. We denote this straightforward method as our *Baseline Approach*.

Example 1. Let, Q_1 and Q_2 be two given query ranges as shown in the shaded region in Figure 1a, where there are seven trajectories $\{T_1, \ldots, T_7\}$ in the dataset. The baseline approach finds $\{p_{41}, p_{71}\}$ for query range Q_1 and $\{p_{42}, p_{62}\}$ for query range Q_2. The candidate trajectories for Q_1 and Q_2 become $\{T_4, T_7\}$ and $\{T_4, T_6\}$, respectively, based on the retrieved points. The intersection of the candidates returns T_4 as the result for the baseline.

Limitations. Unfortunately, the baseline approach is computationally expensive for several reasons:

- Many irrelevant trajectories are retrieved in the first step that intersect at least one range, but not with all of the ranges.
- We need to compute the intersection of the set of candidate trajectories for all query ranges. As the intersection of just two sets takes $O(mn)$ where n and m are the number of trajectories in each set, computing the intersections of all these sets is computationally expensive.

4 Our Proposed Solution

To address the limitations of baseline, the key idea of our approach is to maintain both the co-location of trajectories and the points of the trajectories. We propose a two-level index to maintain such information to facilitate pruning at every level and check only the necessary trajectories. In Section 4.1 we present our index structure, *LoTI*, and in Section 4.2 we describe the multi-range query processing using that index.

4.1 Index structure: LoTI

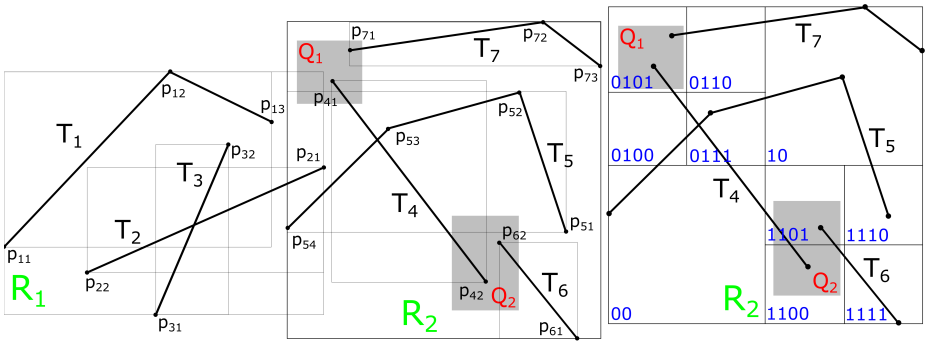
In our proposed index, *LoTI*, the first level is built over the trajectory MBRs to help pruning the trajectories that do not intersect with all query ranges. We use an R-tree as the first level index. The second level is created over the previous index for further pruning. We use a Bucket Quadtree [17] (details in Section 2) for each leaf node of the R-tree as the second level index. We present the details of our proposed index in the following. For ease of demonstration, we show the illustrations for the spatial-only query. For the spatio-temporal query, the R-tree and the Bucket Quadtree are replaced by 3D-Rtree and Bucket Octree, where time is the third dimension.

First level. We compute the MBR of each trajectory, enclosing all of its data points. The computed trajectory MBRs are indexed using an R-tree. Let the fanout of the R-tree be f, i.e., each leaf node contains at most f trajectories. This first-level index groups the trajectories into leaf nodes and helps to prune trajectories based on the MBRs rather than the trajectory points (explained in Section 4.2). Figure 1a shows seven trajectory MBRs indexed by an R-tree, where R_1 and R_2 are two leaf nodes of the R-tree.

Second level. The MBR of each leaf node of the R-tree is indexed using a Bucket Quadtree by setting a *threshold* for the number of trajectories allowed in a leaf node of the Bucket Quadtree. If the number of trajectories exceeds the threshold, then the Bucket Quadtree node is split into four equal-sized sub-nodes. Each node of the Bucket Quadtree is assigned an identifier corresponding to the binary representation of the path from the root. The identifier known as the *location code* can be represented by 2 bits and 3 bits in a specific tree level for the Bucket Quadtree and Bucket Octree, respectively. Each time we split a region, the bottom-left, top-left, top-right, bottom right sub regions are assigned $00, 01, 10, 11$ binary values respectively as shown in Figure 1b.

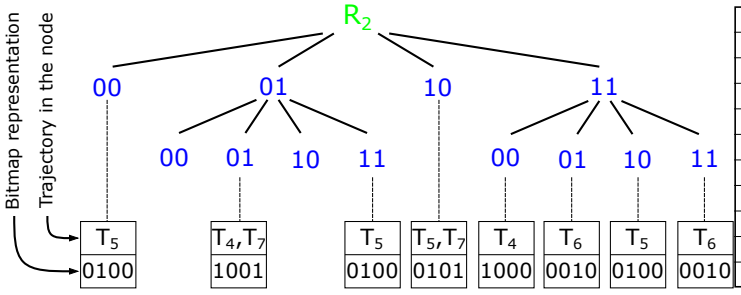
As the size of the Bucket Quadtree increases, and there are multiple quadtrees to be stored, we need much more space to store the MBR information of each node. Thus, we use the *location code tree* based on the Bucket Quadtree to eliminate the use of MBR information. We can easily check whether a node in the location code tree intersects with the query range using its code. Location code trees are much smaller than the original Bucket Quadtree and can be stored in main memory to increase query performance.

Moreover, as we know the number of trajectories in each Bucket Quadtree node, we assign a bit position for each trajectory as shown in Figure 2a. Such a bit representation can facilitate the search of the trajectories intersecting with a query range (explained later in Section 4.2). A lookup table shown in Figure 2b stores a pair (b, t) containing the location code b of the bucket and trajectory t of the trajectory that passes through b. These are the keys to the table, and the values stored in each table entry (b, t) are the points of trajectory t that occur in the bucket b.



(a) R-tree leaf nodes R_1, R_2 (node capacity: 4) (b) Bucket Quadtree split shown for R_2 (bucket capacity: 2)

Fig. 1: Running example for the proposed index



Key pair	Value
<00,T_5>	p_{54}
<0101,T_4>	p_{41}
<0101,T_7>	p_{71}
<0111,T_5>	p_{53}
<10,T_5>	p_{52}
<10,T_7>	p_{72},p_{73}
<1100,T_4>	p_{42}
<1101,T_6>	p_{62}
<1110,T_5>	p_{51}
<1111,T_6>	p_{61}

(a) The tree representation of the bucket decomposition given in Figure 1b (b) Lookup table

Fig. 2: Running example for the proposed index

4.2 Processing Multi-Range Queries

Algorithm 1 contains the pseudocode used to answer a multi-range query using our proposed index. Given a set of query ranges as the multi-range query, we perform the following steps to prune the unnecessary trajectories and answer the query.

1. The first level of the index, i.e., the R-tree of the trajectory MBRs is traversed first. The leaf nodes of the R-tree that intersect with *all of the query ranges* are obtained and kept in a list RN (Lines 1.1-1.2). As a trajectory can be a result if it intersects all of the query ranges, the resulting trajectories are guaranteed to be found in the nodes in RN.
2. For each of the nodes in the list RN we load the corresponding in-memory location code tree (Line 1.3-1.4). These structures at the second level of the index are used to further prune the trajectories that cannot be in the result.
3. We find the nodes of the location code trees that intersect each of the query ranges. For each of these nodes, we obtain the trajectories that pass through the node and set corresponding bits in the $bitSet$ (Lines 1.6-1.8). $bitSet$ contains only trajectories where its contained nodes intersect with all query ranges. Thus, we minimize the number of trajectories that we need to check in a later step.

Algorithm 1: Multi-range query (MRQ) processing

Input: The *LoTI* index over a trajectory dataset D, a set of query ranges Q
Output: a list of trajectories R

```
1.1  Traverse the R-tree of LoTI from the root node for Q
1.2  RN ← The leaf nodes L of the R-tree such that ∀_{q∈Q} intersects(q, L)
1.3  foreach rn ∈ RN do
1.4      lt ← get_LocationCodeTree(rn)
1.5      bitSet ← assign a bit for each trajectory t ∈ rn
1.6      foreach q ∈ Q do
1.7          LN_q ← Get nodes from lt intersecting q
1.8          set bits of trajectories such that ∀_{t∈rn} intersects(q, lt)
1.9      foreach t ∈ bitSet do
1.10         foreach q ∈ Q do
1.11             if contains(q, LN_q) is false then
1.12                 Get trajectory points from the lookup table of LoTI
1.13                 if no point of t in q then
1.14                     Eliminate trajectory t by setting bitSet(t, false)
1.15     if True t ∈ bitSet then
1.16         R.add(t)
1.17 return R
```

4. After the above pruning steps, finally we need to perform a verification step to get the results. Using the trajectory identifier and its corresponding location code tree node identifier, we find the actual points that are stored in the lookup table (described in Section 4.1) and check whether there is a point of that trajectory that actually falls inside the query range. If yes, then that trajectory is a candidate result, obtained from the corresponding query range (Lines 1.11-1.14). If a node of the location code tree is completely contained inside a query range, then all trajectories that pass through that node are considered as candidate trajectories for that query range (Lines 1.11). Otherwise, we check actual points for each trajectory of that node and eliminate them from the candidate set without checking for remaining query ranges if no point intersects the query range. Finally, if a trajectory is a candidate for all of the query ranges, then that trajectory is a result of the multi-range query (Lines 1.15-1.16). We return all such result trajectories.

Example 2. We use the same example of query ranges as Example 1 to illustrate our algorithm's pruning power. Here, let all trajectories be indexed into two leaf nodes, R_1 and R_2. An MBR-based first level filter returns a candidate node R_2 as it intersects all query ranges. The corresponding location code tree for the candidate node R_2 is shown in Figure 2a. The node 0101 intersects query range Q_1 while the nodes 1100, 1101 intersect query range Q_2. The bitset for the node 0101 is $\{1001\}$. The merged bitset of the nodes 1100, 1101 is 1010. The logical *AND* operation on the bitsets returns the value $(1001 \cap 1010 = 1000)$ where the position of the trajectory T_4 is set to $true$. In this example, no nodes are contained in query ranges. In the final step, we check points $\{p_{41}, p_{42}\}$ of the trajectory T_4 from the lookup table on the disk and return as a result.

5 Experimental evaluation

In this section, we compare our proposed algorithm with the baseline approach (presented in Section 3.2) through an extensive experimental evaluation using real datasets.

5.1 Experimental Settings

All algorithms are implemented in JAVA. Experiments were run on a 24 core Intel Xeon E5-2630 2.3 GHz using 256GB RAM, and 1TB 6G SAS 7.2K rpm SFF (2.5-inch) SC Midline disk drives running Red Hat Enterprise Linux Server release 7.2.

Datasets. All experiments were conducted using two real datasets, (i) T-drive dataset and (ii) Beijing dataset.

The T-drive dataset[3] contains 10,357 raw taxi trajectories in Beijing collected for a week in Feb 2008. The Beijing dataset contains 28,162 raw trajectories in Beijing collected for a month in March 2009. Each trajectory is a sequence of GPS locations (latitude and longitude) and the corresponding timestamp. As the locations recorded by a GPS device may contain some noisy data, and the start-end of a taxi trip is not explicitly specified in these datasets, we perform the following cleaning steps - (i) If the location of a taxi does not change for some specific time duration (i.e., the locations in a sequence of consecutive timestamps are very close to each other), it implies that the taxi is likely to be standby to pick up a passenger, and we split the raw trajectory into two separate trajectories at such points. (ii) We have calculated the speed of a moving taxi at each location from its consecutive locations and their timestamps. In some cases we found the speed to be unrealistically high, which is likely to be caused by a wrong GPS location record or a time reset. We split the taxi trajectory into two at such points. We chose the suitable splitting parameters based on the properties of the dataset. (iii) As we want to explore the trajectory movements at different time intervals of the day, each trajectory also spans at most one day.

The cleaned T-drive dataset contains 933,518 trajectories with a total of 12,806,605 points, while the cleaned Beijing dataset contains 608,603 trajectories with a total of 176,470,199 points. Table 1 shows the detailed properties of the cleaned datasets.

Query Generation. We generate the query ranges based on the density distributions of the datasets. Specifically, we divide the dataset area into multiple grid cells, and find the cells with at least a threshold number of trajectories (1000, in our experiments). We choose grid cells randomly and use the center positions for our query ranges, where the number varies from $2-4$. We take the timestamps of a trajectory in the dataset that passes through 2-dimensional query ranges to generate 3-dimensional query ranges. The ranges in a multi-range query do not overlap with each other. We generate such sets

[3] https://protect-au.mimecast.com/s/08IJCE8kAGuOQ1l9Twb4SR?domain=microsoft.com

Statistics Property	T-drive	Beijing
Total no. of points	12,806,605	176,470,199
Total no. of trajectories	933,518	608,603
Avg trajectory length	8.1km	195.4km
Avg points per trajectory	13	289

Table 1: Dataset statistics

Parameter Description	Values
Number of ranges	**2**, 3, 4
Search region (km)	1, 2, **4**, 8, 16
Search time interval (hour)	0.5, **1**, 2, 4, 8
Quad(Oc)tree node capacity	5 trajectory

Table 2: Experimental parameters

Dataset	2D				3D			
	Baseline		LoTI		Baseline		LoTI	
	Size(GB)	*Time(min)*	*Size(GB)*	*Time(min)*	*Size(GB)*	*Time(min)*	*Size(GB)*	*Time(min)*
T-drive	0.9	19.4	0.8	0.7	1.3	49.4	1.6	2.4
Beijing	12.4	424	14.2	32.4	17.2	605	12.6	16.2

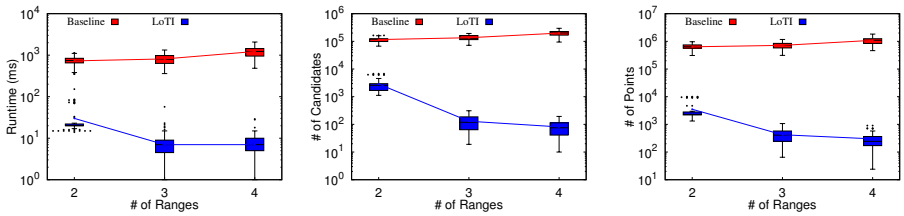
Table 3: 2D and 3D index sizes and construction times for T-drive and Beijing datasets

of 100 multi-range queries and report the performance. The experiments are conducted on both the spatial-only and spatio-temporal trajectories. We use the same set of queries for both setups, where the temporal information is removed in the spatial-only case.

Evaluation and Parameterization. We study the performance of both the baseline and our approach by varying several parameters as shown in Table 2, where the values in bold represent the default values. The fanout of the R-tree in baseline and the R-tree of *LoTI* are set to 100. For all experiments, a single parameter is varied while keeping the rest at the default values. We study the impact of each parameter by running 100 queries and report the values of: (i) the query execution time, (ii) the number of candidates considered, and (iii) the number of points checked. We also report the costs of constructing the indexes in Table 3.

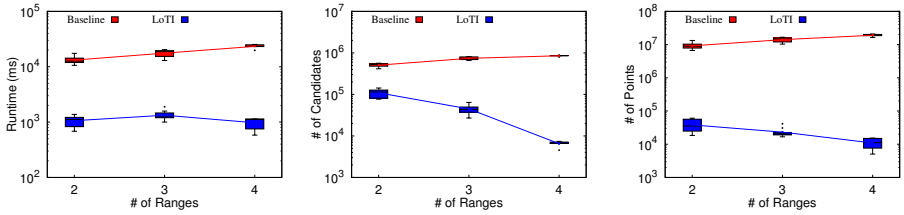
5.2 Performance Evaluation on Spatial-Only Data

The performance for multiple runs is shown in boxplots, where the bounding box shows the first and third quartiles; the whiskers show the range, up to 1.5 times of the interquartile range; and the outliers beyond this value are shown as separate points. The average values are shown as connecting lines.



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 3: Varying the number of query ranges in 2D space (T-drive dataset)

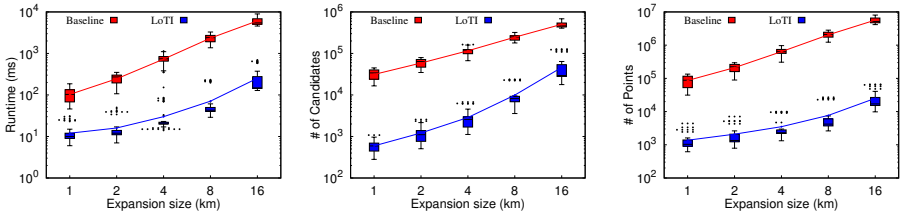


(a) Query performance (b) Candidate generation (c) Points checked

Fig. 4: Varying the number of query ranges in 2D space (Beijing dataset)

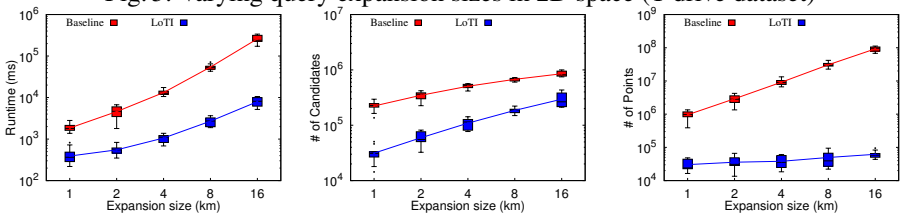
Effect of Varying the Number of Ranges. We conducted experiments by varying three different numbers of ranges and evaluated the performance as shown in Figures 3 and 4 for the T-drive and Beijing datasets, respectively. Our algorithm, *LoTI* is up to 2 orders of magnitude faster than the baseline and checks up to 3 orders of magnitude

fewer candidates and points in spatial-only space. As more points fall into a higher number of ranges, the costs of the baseline increases. In contrast, the benefit of our approach is more pronounced for a higher number of ranges as the trajectories that do not pass through all of the ranges can be quickly identified and pruned from further consideration with our two-level index. As shown with the ranges of the boxplots, our approach significantly outperforms the baseline for all of the 100 sets of queries. Table 4 shows the breakdown of runtime that different steps of our approach take. As shown in the table, the MBR filtering using the first level of *LoTI* is the quickest filter, the second level takes the most time as trajectories are pruned further, and the lookup table takes moderate time to finally obtain results.



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 5: Varying query expansion sizes in 2D space (T-drive dataset)



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 6: Varying query expansion sizes in 2D space (Beijing dataset)

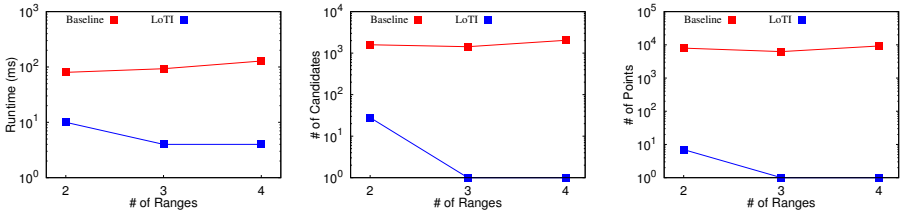
Filter	**# of Ranges**			**Expansion (km)**				
	2	**3**	**4**	**1**	**2**	**4**	**8**	**16**
	Runtime (ms)							
MBR	4	4	3	4	4	4	5	5
Quadtree	912	1177	928	272	414	912	2518	7696
Lookup table	143	148	41	112	130	143	176	261

Table 4: Time to run each filtering on 2-dimensional LoTI index on Beijing dataset

Effect of Varying the Query Range Size. Figures 5 and 6 show the experimental result for different sizes of the query range expansions for the T-drive and Beijing datasets, respectively. As more trajectories are likely to fall inside larger query ranges, thus the cost of both methods increase with the increase of the query range size. In all cases, our method outperforms the baseline by 1 order of magnitude in runtime by efficient pruning, and the gaps in the performance do not vary much.

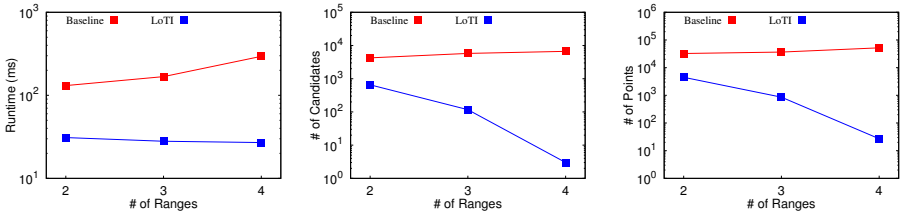
5.3 Performance Evaluation on Spatio-Temporal data

As more trajectories are likely to be pruned by the temporal constraints, fewer trajectories and points are required to be checked for both methods than the spatial-only



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 7: Varying the number of query ranges in 3D space (T-drive dataset)

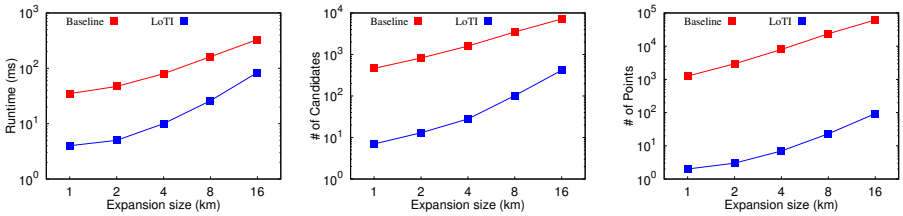


(a) Query performance (b) Candidate generation (c) Points checked

Fig. 8: Varying the number of query ranges in 3D space (Beijing dataset)

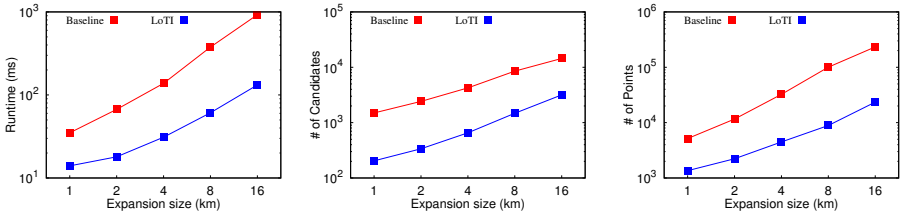
counterpart. As the number of candidates is very small for our approach in some experiments, a boxplot in log-scale is difficult to comprehend. Thus we present the average performance using line plots.

Effect of Varying the Number of Ranges. Figures 7 and 8 show the performances for varying the number of spatial-temporal query ranges, where the default time interval is set to one hour. Our approach, *LoTI* is up to 1 order of magnitude faster than the baseline and checks 2 to 3 orders of magnitude fewer candidates and points. The performance trends are similar to the spatial-only experiments. Trajectories in the Beijing dataset have more points in average than T-drive. Thus, we find more query results as the likelihood to find trajectories increase due to the point density.



(a) Query performance (b) Candidate generation (c) Points checked

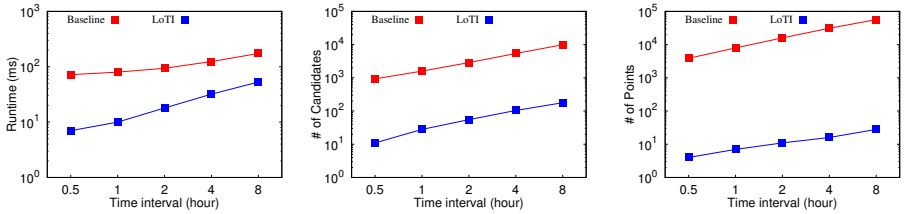
Fig. 9: Varying query expansion sizes in 3D space (T-drive dataset)



(a) Query performance (b) Candidate generation (c) Points checked

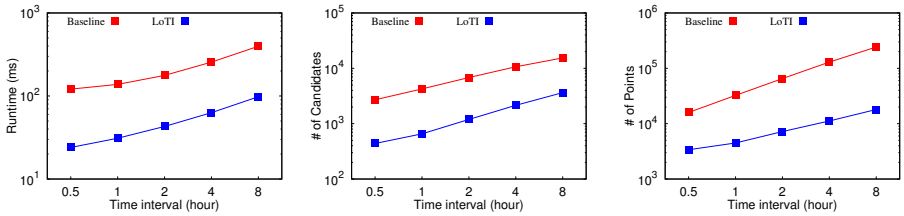
Fig. 10: Varying query expansion sizes in 3D space (Beijing dataset)

Effect of Varying the Query Range Size. Figures 9 and 10 show the experimental result for different sizes of the query range expansions, where the time interval is set to its default value. As more points need to be checked for larger ranges, the number of candidates for both methods increases which lead to more runtime. As the additional dimension can prune some more trajectories, the costs of both methods are less than their spatial-only counterparts.



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 11: Varying query time intervals in 3D space (T-drive dataset)



(a) Query performance (b) Candidate generation (c) Points checked

Fig. 12: Varying query time intervals in 3D space (Beijing dataset)

Effect of Varying the Query Time Interval. Figures 11 and 12 show the performance of the methods for varying different query time intervals, ranging from 30 minutes to 8 hours. As more trajectories travel between query ranges for a larger time interval, the costs increase for both methods by having to check more candidates and points. In all cases, our method outperforms the baseline.

6 Conclusion

In this paper, we presented and studied the problem of multi-range query processing on trajectories. We proposed a novel two-level index, *LoTI* that preserves the co-location relationship between both trajectories and the points of the trajectories. We described a query processing approach using an index that prunes unnecessary trajectories. We conducted extensive experimental evaluations using two real datasets by varying different settings of parameters. Our approach significantly outperformed the baseline approach for all of the parameter settings, by one to two orders of magnitude. Long temporal overlaps between query ranges are likely to contain trajectories visiting query ranges with different orders. We consider such a scenario in the modelling of the index and algorithm as future work.

Acknowledgement. This work was partially supported by the National Science Foundation under Grant IIS-13-20791, ARC DP170102726, DP180102050, NSFC 61728204 and 91646204. Zhifeng Bao is a recipient of Google Faculty Award.

References

1. Cai, Z., Ren, F., Chen, J., Ding, Z.: Vector-based trajectory storage and query for intelligent transport system. IEEE Transactions on Intelligent Transportation Systems pp. 1–12 (2017)
2. Chakka, V.P., Everspaugh, A., Patel, J.M.: Indexing large trajectory data sets with SETI. In: CIDR (2003)
3. Chen, Z., Shen, H.T., Zhou, X.: Discovering popular routes from trajectories. In: ICDE. pp. 900–911 (2011)
4. Cudré-Mauroux, P., Wu, E., Madden, S.: Trajstore: An adaptive storage system for very large trajectory data sets. In: ICDE. pp. 109–120 (2010)
5. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD. pp. 47–57 (1984)
6. Han, Y., Chang, L., Zhang, W., Lin, X., Wang, L.: Efficiently retrieving top-k trajectories by locations via traveling time. In: ADC. pp. 122–134 (2014)
7. Hjaltason, G.R., Samet, H.: Speeding up construction of pmr quadtree-based spatial indexes. The VLDB Journal **11**(2), 109–137 (2002)
8. Klinger, A.: Patterns and search statistics. In: Optimizing methods in statistics, pp. 303–337. Elsevier (1971)
9. Li, X., Han, J., Lee, J., Gonzalez, H.: Traffic density-based discovery of hot routes in road networks. In: SSTD. pp. 441–459 (2007)
10. Nascimento, M.A., Silva, J.R.O.: Towards historical r-trees. In: Proceedings of the 1998 ACM symposium on Applied Computing. pp. 235–240 (1998)
11. Papadias, D., Tao, Y., Kalnis, P., Zhang, J.: Indexing spatio-temporal data warehouses. In: ICDE. pp. 166–175 (2002)
12. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches to the indexing of moving object trajectories. In: VLDB. pp. 395–406 (2000)
13. Popa, I.S., Zeitouni, K., Oria, V., Barth, D., Vial, S.: Indexing in-network trajectory flows. The VLDB Journal **20**(5), 643–669 (2011)
14. Preparata, F.P., Shamos, M.I.: Computational Geometry - An Introduction. Springer (1985)
15. Ranu, S., P, D., Telang, A.D., Deshpande, P., Raghavan, S.: Indexing and matching trajectories under inconsistent sampling rates. In: ICDE. pp. 999–1010 (2015)
16. Sacharidis, D., Patroumpas, K., Terrovitis, M., Kantere, V., Potamias, M., Mouratidis, K., Sellis, T.K.: On-line discovery of hot motion paths. In: EDBT. pp. 392–403 (2008)
17. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann (2006)
18. Shang, S., Chen, L., Jensen, C.S., Wen, J., Kalnis, P.: Searching trajectories by regions of interest. TKDE **29**(7), 1549–1562 (2017)
19. Song, Z., Roussopoulos, N.: Seb-tree: An approach to index continuously moving objects. In: MDM. pp. 340–344 (2003)
20. Tang, L.A., Zheng, Y., Xie, X., Yuan, J., Yu, X., Han, J.: Retrieving k-nearest neighboring trajectories by a set of point locations. In: SSTD. pp. 223–241 (2011)
21. Tao, Y., Papadias, D.: Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In: VLDB. pp. 431–440 (2001)
22. Theodoridis, Y., Vazirgiannis, M., Sellis, T.K.: Spatio-temporal indexing for large multimedia applications. In: ICMCS. pp. 441–448 (1996)
23. Wang, H., Zimmermann, R.: Processing of continuous location-based range queries on moving objects in road networks. TKDE **23**(7), 1065–1078 (2011)
24. Wang, H., Zheng, K., Xu, J., Zheng, B., Zhou, X., Sadiq, S.W.: Sharkdb: An in-memory column-oriented trajectory storage. In: CIKM. pp. 1409–1418 (2014)
25. Wang, S., Bao, Z., Culpepper, J.S., Sellis, T., Cong, G.: Reverse k nearest neighbor search over trajectories. TKDE **30**(4), 757–771 (2018)