Road-aware Indexing for Trajectory Range Queries

Yong Wang Kaiyu Li Guoliang Li Nan Tang

Abstract—Answering spatio-temporal range queries (RQs) on trajectory databases, i.e., finding all trajectories that intersect given ranges, is crucial in many real-world applications. Various kinds of indexes have been proposed to accelerate RQs. However, existing indexes typically use Euclidean distance to prune irrelevant regions without concerning the underlying road network information. Nevertheless, as vehicle trajectories are generated on road network edges, the road network could be seen as meta knowledge of trajectories and be used to index and query trajectories. To this end, we propose RP-Tree, a road network-aware partition tree to support efficient RQs. The basic idea is partitioning a road network graph into hierarchical subgraphs and generate a balanced tree structure, where each tree node maintains its associated trajectories. We compactly index the spatio-temporal information of trajectories on the corresponding road network edges. Then, we design efficient search algorithms to support RQs by pruning irrelevant trajectories through subgraph range borders associated with RP-Tree nodes. Last but not least, we scale RP-Tree to very large datasets by devising approximate algorithms with bounded confidence at an interactive speed. Experimental results on three real-world datasets from Porto, Chengdu, and Beijing show that our method outperform baselines by 1 to 2 orders of magnitude.

Index Terms—Trajectory Data, Range Query, Spatio-temporal Data, Approximate Query Processing

1 Introduction

Answering spatio-temporal range queries (RQs) on trajectory databases is fundamental to trajectory data management. An RQ asks for all trajectories that intersect some given spatio-temporal rectangles [53]. For example, a policeman may want to find all the trajectories passing by the crime scene and airport between 4 pm and 6 pm.

Example 1. *Figure 1 shows 4 trajectories $\{t_1, t_2, t_3, t_4\}$, and a road network with 5 nodes $\{v_1, v_2, v_3, v_4, v_5\}$ and their associated edges. Let an RQ be "*finding all trajectories passing by range r at any time*", which are $\{t_1, t_2, t_3\}$.*

Traditional spatio-temporal indexes focus mainly on using geometric properties, *e.g.*, 3DR-tree [34], MV3R-tree [41] and grid-based index [2], [53], [55], without considering the underlying road-networks to prune irrelevant trajectories. More concretely, they typically partition the map into spatial blocks [55], e.g., minimum bounding rectangles (MBRs) or girds. Each block indexes all trajectory segments passing by it, and thus corresponding trajectories for a given spatio-temporal range query can be quickly found by accessing the blocks that overlap with query ranges. Without loss of generality, we show the spatio-temporal query processing on grid-based index through an example.

Example 2. *[Grid-based Index.] If we use a grid-based index to answer the above RQ in Figure 1, it needs to first find the two grids that intersect with the query range,* i.e., *the left grid indexing $\{t_1, t_2, t_3\}$ and the right grid indexing $\{t_2, t_3, t_4\}$. Then we should verify whether each of the trajectory in $\{t_1, t_2, t_3\} \cup \{t_2, t_3, t_4\} = \{t_1, t_2, t_3, t_4\}$ intersects with range r. However, it is rather expensive to verify all these candidates to determine the relevant trajectories, because it needs to examine all the "segments" of trajectories and each trajectory may have hundreds or thousands segments.*

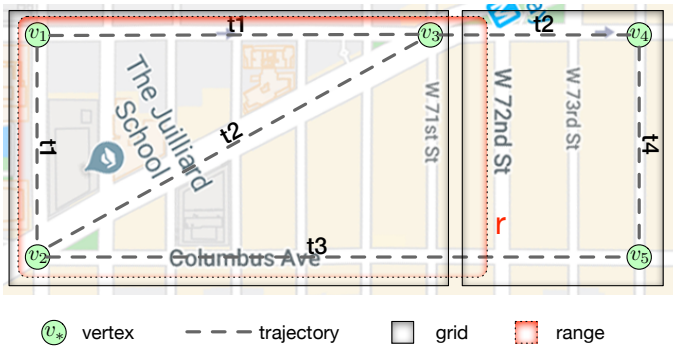

Fig. 1. A road network with 4 trajectories t_1–t_4 and a *red* range r.

Next consider an index leveraging the road network.

Example 3. *[Road-aware Index.] If we can leverage the road network graph to index trajectories, we just need to check the trajectories passing by edges $(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_2, v_3)$ and (v_2, v_5). For example, edge (v_1, v_2) indexes one segment of t_1. We can get the union of all the indexed trajectories of selected road edges, and merge them to get the exact query result $\{t_1, t_2, t_3\}$, without any further refinement.*

The above examples motivate us that utilizing the road network speeds up trajectory range query processing.

Our Methodology. Our essential idea is to build connections between trajectories and road network graph, using a

- *Yong Wang, Kaiyu Li and Guoliang Li are with the Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China. wangy18@mails.tsinghua.edu.cn, ky-li18@tsinghua.org.cn, liguoliang@tsinghua.edu.cn. Kaiyu Li and Guoliang Li are the corresponding authors.*
- *Nan Tang is a senior scientist at Qatar Center for Artificial Intelligence, QCRI, HBKU, Qatar. ntang@hbku.edu.qa.*

sequence of road edges with corresponding timestamps to represent a trajectory and indexing the trajectories based on the road edges. We can then utilize the road-aware index to effectively prune irrelevant trajectories and efficiently support RQs. Moreover, in order to support spatio-temporal range queries that find trajectories within a time range, we need to integrate the time ranges with the road edges.

Based on the above idea, we need to build a tree index over the road network graph to index spatio-temporal trajectory data. To this end, we recursively partition a road network graph into hierarchical subgraphs based on trajectory distribution to compose balance tree structure [22], [56]. Formally, we present RP-Tree, a road network partition tree data structure. RP-Tree makes full use of the underlying road network, which indexes spatio-temporal trajectories passing by road edges based on hierarchical graph partitions and is able to support update for new trajectories. Then, we design efficient RP-Tree-based search algorithms to support both single and multiple range queries by fully exploiting the graph topologies and properties to prune unpromising searching branches. Moreover, to scale RP-Tree to very large datasets, we propose sampling techniques to support confidence bounded approximate RQs.

Contribution. Our primary argument is that a road-aware index significantly benefit the trajectories range query processing, in which we make the following contributions.

(1) *A Novel Index*: We propose RP-Tree to index trajectories by leveraging the underlying road network. We present how to efficiently build and update RP-Tree (Section 3).

(2) *Query Processing*: We design tree-based algorithms with pruning by fully exploiting the graph topologies and properties to support single and multiple range RQs (Section 4).

(3) *Approximate Query Answering:* We support efficient yet error bounded approximate RQs. To the best of our knowledge, we are the first work on supporting this (Section 5).

(4) *Experiments:* We conducted extensive experiments on real-world datasets (taxi trajectories of Beijing, Porto and Chengdu). Experimental results show that our methods outperform baselines by 1 to 2 orders of magnitude (Section 6).

Organization. Section 2 introduces preliminaries. Section 3 describes how to construct and update an RP-Tree. Section 4 discusses efficient algorithms to support exact RQs. Section 5 devises approximate solutions. Section 6 presents experimental findings. Sections 7 discussed related work. Finally, Section 8 closes the paper by concluding remarks.

2 PRELIMINARIES

2.1 Data Model

We summarize the notations used in this paper in Table 1.

Road Networks. A *road network* is modeled as an undirected connected graph $G = (V, E)$, where V is the set of vertices (*i.e.,* road joints) and E is the set of edges (*i.e.,* road segments). Note that we use an undirected graph (see Figure 2) for simplicity, supporting directed graph is straightforward.

Example 4 (Road Network)**.** *Figure 2 shows a road network* $G = (V, E)$*, where* $V = \{v_1, v_2, \ldots, v_{17}\}$ *and* $|E| = 25$ *(e.g.,* $(v_1, v_3) \in E$*).* v_∞ *is a virtual vertex (introduced in Section 4).*

TABLE 1
Table of Notations.

Notation	Definition
$G = (V, E)$	road network
$t = \langle p_0, p_1, \ldots, p_k \rangle$	a map-matched trajectory
$p_i = (v_i, \tau_i)$	tuple of vertex and timestamp
$\mathbf{r}/R$	single/multiple query range(s)
$\hookrightarrow / \nrightarrow$	intersects/do not intersect
$T_\mathbf{r}/T_R$	trajectories intersect range(s) $\mathbf{r}/R$
$tr(e)$	trajectories passing by edge e
$idx(e)$	index that stores $tr(e)$
N	node N in RP-Tree
N_l, N_r	left and right children of N
$G_N = (V_N, E_N)$	graph represented by node N
$\mathbb{L}_N$	linking edges between N_l and N_r
S_r	spatial area of range r
θ_H	minimum level threshold for indexing
η	threshold of max leaf node size
$G_r = (V_r, E_r)$	range graph of spatial range r
$\mathcal{E}(\cdot), \mathcal{D}(\cdot)$	expectation and deviation
δ, c	error bound, confidence

Raw Trajectories. A *raw trajectory* is a sequence of spatio-temporal GPS points, where each point is an ordered triplet of latitude, longitude and timestamp.

Map-matched Trajectories. A raw trajectory can be mapped to a *map-matched trajectory*, *i.e.,* $t = \langle p_0, p_1, \ldots p_k \rangle$ where $p_i = (v_i, \tau_i)$ is a tuple of vertex and timestamp at v_i, $v_i \in V, (v_{i-1}, v_i) \in E$ and $\tau_{i-1} < \tau_i$, $\forall\ 1 \leq i \leq k$. Specifically, the moving object of t is at vertex $v_i \in V$ at time τ_i. Moreover, for each edge $e \in E$, we denote the set of map-matched trajectories that passing $e \in E$ as $tr(e)$. Raw trajectories can be mapped to map-matched trajectories using map-matching algorithms [27], [37].

In practice, a raw trajectory usually does not start from or end at a vertex of the road network, *i.e.,* at some middle point a road edge. To handle this issue, we align the start and end points to vertices for map-matched trajectories and additionally maintain the offsets [6]. In the following context, we suppose trajectories are aligned for simplicity. We index map-matched trajectories and hence use *trajectories* for map-matched trajectories in short in the following context.

Example 5 (Trajectory)**.** *There are 12 map-matched trajectories* $\{t_1, \ldots, t_{12}\}$ *in Figure 2. Specifically,* t_9 *consists of three consecutive edges,* i.e., $t_2 = \langle (v_7, \tau_1), (v_2, \tau_2), (v_1, \tau_3), (v_3, \tau_4) \rangle$*, and* $tr(v_6, v_8) = \{t_3, t_6, t_9\}$*. We only show the map-matched details of* t_2, t_4 *and* t_8 *for simplicity. We suppose* $\tau_i < \tau_{i+1}$ *for* $1 \leq i \leq 10$*. Note that the raw trajectory* w.r.t. t_4 *starts from the "pick up" point.*

2.2 Query Model

Query Ranges. A *query range* (or range, for short) is a spatio-temporal rectangle (*i.e.,* 3-dimensional cube of latitude, longitude and time). Let $\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m\}$ be a set of ranges. r_j (spatial range) is the projection of $\mathbf{r}_j = (r_j, \tau_{j1}, \tau_{j2})$ (spatio-temporal range) on the coordinates of latitude and longitude, *i.e.,* a 2-dimensional rectangle.

Spatial Query. For spatial query of r_j, a trajectory $t = \langle p_0, p_1, \ldots p_k \rangle$ *intersects* range r_j, denoted as $t \hookrightarrow r_j$, if there exists v_i within r_j or edge (v_{i-1}, v_i) that overlaps with r_j; otherwise $t \nrightarrow r_j$.

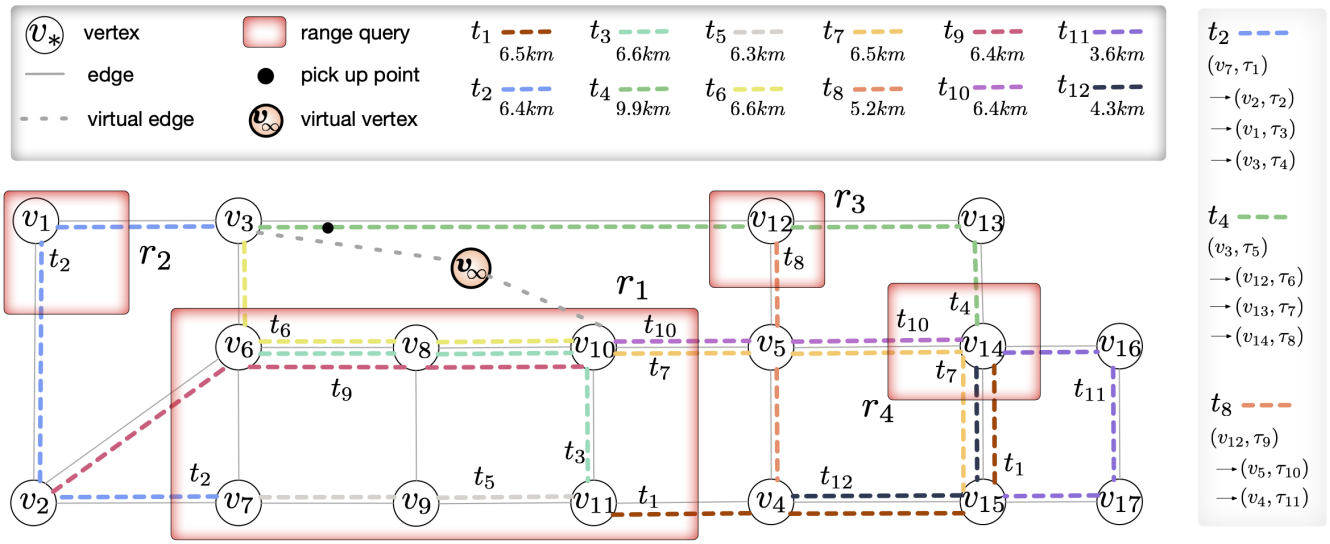

Fig. 2. Examples of Trajectories and RQs on A Road Network.

Q_1 : SELECT t.ID FROM T WHERE t intersects $(r_1, [0, \infty])$;
Q_2 : SELECT t.ID FROM T
WHERE t intersects $\{(r_1, [\tau_1, \tau_3]), (r_2, [\tau_2, \tau_4])\}$;
Q_3 : SELECT SUM(t.len) FROM T WHERE t intersects $(r_4, [0, \infty])$;
Q_4 : SELECT AVG(t.len) FROM T WHERE t intersects $(r_4, [0, \infty])$
CONFIDENCE $\geq 95\%$ ERROR $\leq 5\%$;
Q_5 : SELECT COUNT(t.len) FROM T
WHERE t intersects $(r_1, [\tau_1, \tau_3])$ ERROR $\leq 10\%$;
Q_6 : SELECT SUM(t.len) FROM T WHERE t intersects $(r_1, [\tau_1, \tau_3])$
CONFIDENCE $\geq 90\%$ ERROR $\leq 10\%$.

Fig. 3. Query Examples with SRQL.

Spatial-temporal Query. For spatio-temporal query of $\mathbf{r}_j$, a trajectory $t = \langle p_0, p_1, \dots p_k \rangle$ *intersects* range $\mathbf{r}_j$, denoted as $t \hookrightarrow \mathbf{r}_j$, if there exists p_i s.t. v_i is within r_j and $\tau_i \in [\tau_{j1}, \tau_{j2}]$, or (p_{i-1}, p_i) s.t. (v_{i-1}, v_i) overlaps with r_j during $[\tau_{j1}, \tau_{j2}]$; otherwise $t \nrightarrow \mathbf{r}_j$. Furthermore, a spatial query range r can be considered as a special case of spatio-temporal range $(\mathbf{r}, [0, \infty])$.

Example 6 (Query Range). *In Figure 2, r_1, r_2, r_3 and r_4 are spatial ranges. We have $t_8 \hookrightarrow r_3$ and $t_8 \nrightarrow r_1$. $\mathbf{r}_3 = (r_3, [\tau_4, \tau_7])$ is a spatio-temporal range. We have $t_4 \hookrightarrow \mathbf{r}_3$ and $t_8 \nrightarrow \mathbf{r}_3$.*

Definition 1 (Range Queries (RQs)). *Given a set of trajectories $T = \{t_1, \dots, t_n\}$ and a set of query ranges $R = \{\mathbf{r}_1, \dots \mathbf{r}_m\}$, find the set of trajectories $T_R \subseteq T$ where (1) $\forall t \in T_R$ and $\forall \mathbf{r} \in R$, we have $t \hookrightarrow \mathbf{r}$; (2) $\forall t \in T \setminus T_R$, $\exists \mathbf{r} \in R$ s.t. $t \nrightarrow \mathbf{r}$.*

More concretely, we use RQs to find all trajectories such that each of them intersects *all* of the given query ranges. Note that, an RQ degenerates into a *single range query* (or a *single RQ*) when there is only one query range.

Range Query Language. We propose *spatio-temporal range query language* (SRQL), a SQL-like syntax to specify RQs. We define two types of queries for SRQL, *i.e.,selection* and *aggregation*. Given a set R of query ranges, *selection queries* ask for $T_R \subset T$ that satisfies the RQ, and *aggregation queries* conduct aggregation operation AGG (*e.g.*, AVG, VARIANCE, COUNT and SUM) on trajectories in T_R.

Example 7. *In Figure 3, Q_1 and Q_2 are* selection *queries, and Q_3, Q_4, Q_5 and Q_6 are* aggregation *queries. Q_1 is a single RQ where $R = \{(r_1, [0, \infty])\}$, $T_R = \{t_1, t_2, t_3, t_5, t_6, t_7, t_9, t_{10}\}$ (shown in Figure 2), and Q_2 is an RQ where $R = \{(r_1, [\tau_1, \tau_3]), (r_2, [\tau_2, \tau_4])\}$, $T_R = \{t_2\}$. Q_3 is a single RQ on spatial range r_3 for summation of lengths of trajectories in anytime,* i.e., *$T_R = \{t_1, t_4, t_7, t_{10}, t_{11}, t_{12}\}$, and the query result $\sum_{t_i \in T_R} t_i.len = 37.2km$.*

3 RP-Tree: Road Network Partition Tree

In this section, we present the intuition of how to design a road-aware index to support RQs and propose the structure of RP-Tree. Then we devise a co-partition algorithm that hierarchically partitions the trajectories into sub-groups by considering both spatial shape and road networks.

3.1 The Structure of RP-Tree

Index Desiderata. Intuitively, a good index for supporting RQs on trajectories fulfills the following properties.

(1) *Road-aware.* Utilize the road constraints to do pruning.

(2) *Hierarchical structure.* Build a hierarchical structure which enables efficient search and pruning from coarse-grained higher levels to fined-grained lower levels.

(3) *Balanced structure.* A balanced structure reduces total tree levels and outperforms unbalanced structures for indexing large datasets.

Co-partition. To index trajectories with a road network graph, we partition the road network into a balanced hierarchical structure according to the density of trajectories on each edge. Meanwhile, to support range query of rectangles, we maintain the spatial information. Based on these two criteria, we partition the road network into a hierarchical tree, where each node contains a subgraph (of the road network) and its minimum bounding rectangle (MBR), and an edge between two nodes denotes the parent-child relationship (see Section 3.2). Next we formally define RP-Tree.

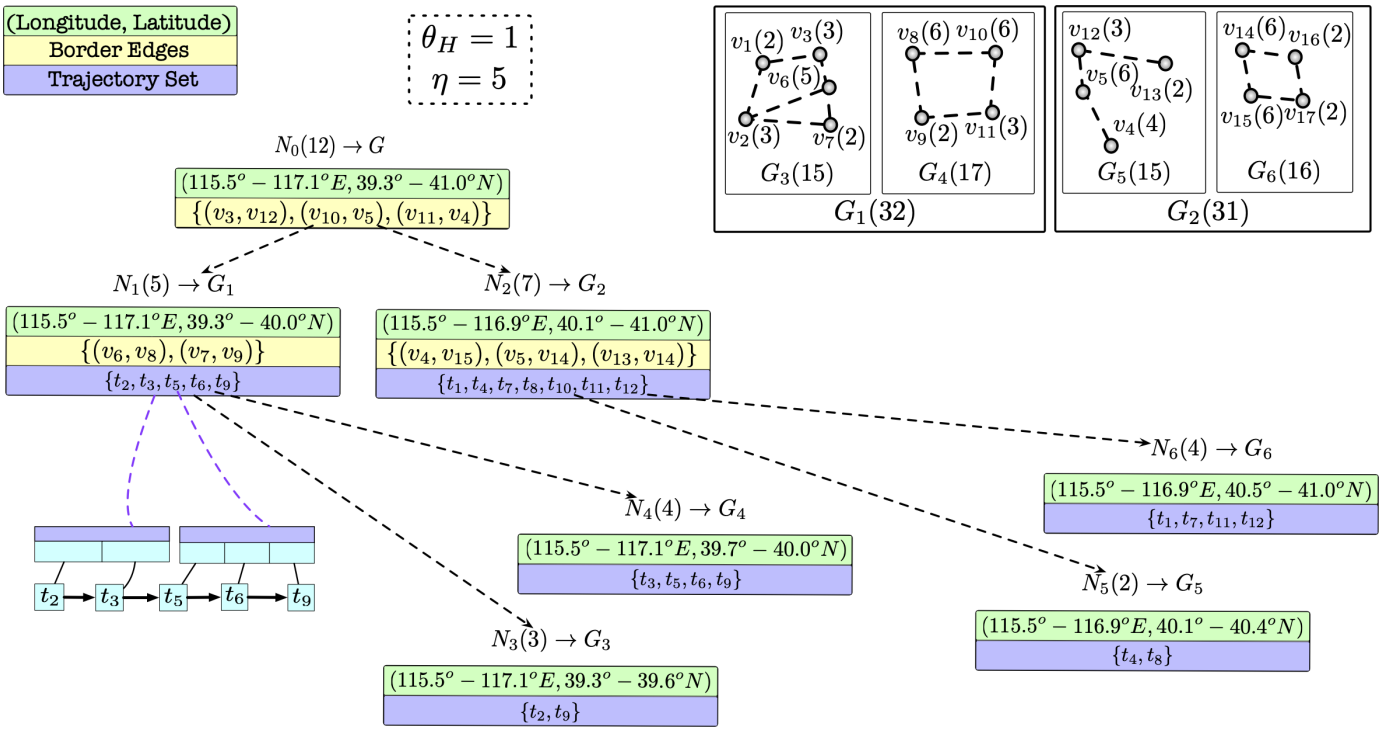

Fig. 4. RP-Tree Partition Example.

Definition 2 (RP-Tree). *Given a road network $G = (V, E)$ and a leaf node size threshold η, an RP-Tree is a balanced binary search tree where each node N represents a corresponding subgraph $G_N = (V_N, E_N)$ (including the spatial MBR of G_N). Each non-leaf node N has a left child node N_l and right child N_r, denoted by G_{N_l} and G_{N_r}, respectively. Each leaf node N maintains the subgraph G_N, where $|V_N| \leq \eta$. For each node N in level no less than threshold θ_H, it indexes trajectories intersecting some road edge(s) in N chronologically with respect to trajectory departure time, denoted as $idx(N)$.*

For each edge $e \in E$, we denote the trajectories passing e by $tr(e)$. To further support the temporal dimension, we maintain the trajectories passing e chronologically with respect to the starting time on e, where chronologically sorted trajectories on e are denoted by $idx(e)$. In this way, we can more conveniently build trajectory index $idx(N)$ for a node N with $\bigcup_{e \in E_N} idx(e)$. Note that we set a minimum level threshold θ_H for indexing to balance the index size of very large road networks. If θ_H is too small, we repeatedly store the trajectories' IDs for large subgraphs in lower levels, while there are rarely range queries with very large spatial ranges. If θ_H is too large, we need more time to answer RQs with big ranges covering some large subgraphs whose trajectories are not indexed. Thus we need to select a proper θ_H to balance storage and query speed.

Example 8. *Figure 4 depicts an RP-Tree example for trajectories and the road network in Figure 2. The root node N_0 maintains the border latitudes and longitudes of G,* i.e., *$(115.5^oE - 117.1^oE, 39.3^oN - 41.0^oN)$ and the border edges linking G_1 and G_2,* i.e., *$\{(v_3, v_{12}), (v_{10}, v_5), (v_{11}, v_4)\}$. Besides, the root node maintains the number of trajectories it has indexed,* i.e., *12. The non-leaf node N_1 represents G_1. Similarly, it maintains the border latitudes and longitudes of G_1, and border edges linking G_3 and G_4. As $\theta_H = 1$, trajectories in G_1 are indexed in chronological order of trajectory starting time,* i.e., *$idx(N_1) = \{t_2, t_3, t_5, t_6, t_9\}$. For leaf node N_3 with $|V_{N_3}| \leq \eta = 5$, the partition stops, it maintains border latitudes and longitudes and trajectories $idx(N_1) = \{t_2, t_9\}$ in G_3.*

3.2 The Construction of RP-Tree

Partition Method. The construction of RP-Tree is a recursive process that can be formulated to the problem below.

A graph $G_N = (V_N, E_N)$ is partitioned into two parts $G_{N_l} = (V_{N_l}, E_{N_l})$ and $G_{N_r} = (V_{N_r}, E_{N_r})$ bounded by rectangles without any overlaps, and they share the common edges (linkings) $\mathbb{L}_N$. It aims to find an edge-balanced partitions, *i.e.*,

$$\min || \sum_{e \in E_{N_l}} tr(e)| - | \sum_{e \in E_{N_r}} tr(e)||$$

For vertex sets V_{N_l} and V_{N_r}, the weight of each vertex v is the summation of all its edges' weights, *i.e.*, $w(v) = \sum_{e \in v.edges} |tr(e)|$ where $v.edges$ denotes the set of all adjacent edges of v. Then, minimizing a vertex-balanced partition problem is to find a partition method such that:

$$\min || \sum_{v \in V_{N_l}} w(v)| - | \sum_{v \in V_{N_r}} w(v)|| \quad (1)$$

We have the following lemma:

Lemma 1. *For graph G_N, edge-balanced partition is equivalent to vertex-balanced partition.*

Proof 1. *Since $w(v) = \sum_{e \in v.edges} |tr(e)|$, we have*

$$\begin{cases} |\sum_{v \in V_{N_l}} w(v)| = \sum_{e \in E_{N_l}} |tr(e)| + \sum_{e \in \mathbb{L}_N} |tr(e)| \\ |\sum_{v \in V_{N_r}} w(v)| = \sum_{e \in E_{N_r}} |tr(e)| + \sum_{e \in \mathbb{L}_N} |tr(e)| \end{cases}$$

Algorithm 1: RPConstruct(G, η, θ_H)

Input: Graph $G = (V, E)$, threshold η and θ_H.
Output: RP-Tree of graph G.

1 $\mathcal{G}$ = `Split`$(G, \eta, \theta_H, 1)$;
2 **return** $\mathcal{G}$;

Function Split(G, η, θ_H, d)

1 // $G = (V, E)$, η, θ_H are thresholds
2 // h is current depth in RP-Tree
3 $N = \varnothing$; // Initialize empty node
4 $N.border$ = $V.border$; // Geographical rectangle range
5 **if** $d \geq \theta_H$ **then**
6 //Index trajectories in N
7 $idx(N)$ = $building\text{-}idx(\bigcup_{e \in E} tr(e))$;
8 **if** $|V| < \eta$ **then**
9 $N.adjacentlist$=(V, E); // Construct leaf node
10 **else**
11 $V.sortby(\arg\max_x(x.range))$, $x \in \{lat, lng\}$;
12 $p =$ $\arg\min_i |\sum_{j\in[0,i]} w(v_j) - \sum_{j\in[i+1,|V|-1]} w(v_j)|$;
13 $N.linking = \bigcup_{i\in[0,p],j\in[p+1,|V|-1]}(v_i, v_j)$;
14 // $G_{N_l} = (V_{N_l}, E_{N_l})$, $(v_i, v_j) \in E_{N_l}$, $i, j \in [0, p]$
15 N_l = `Split`$(G_{N_l}, \eta, \theta_H, h + 1)$;
16 // $G_{N_r} = (V_{N_r}, E_{N_r})$, $(v_i, v_j) \in E_{N_r}$, $i, j \in [p + 1, |V| - 1]$
17 N_r = `Split`$(G_{N_r}, \eta, \theta_H, h + 1)$;
18 **return** N;

Thus

$$\big|\big|\sum_{v\in V_{N_l}} w(v)\big| - \big|\sum_{v\in V_{N_r}} w(v)\big|\big| = \big|\big|\sum_{e\in E_{N_l}} tr(e)\big| - \big|\sum_{e\in E_{N_r}} tr(e)\big|\big|$$

Note that, if the value of Equation 1 is too large, the RP-Tree will finally be unbalanced which can lead to bad performance on supporting RQs. Following the above lemma we can give the upper bound of Equation 1 as follows.

Lemma 2. *For graph G_N, there exists a solution to minimize* vertex-balanced *partition on G_N,* i.e., *optimize* $\min \big|\big|\sum_{v\in V_{N_l}} w(v)\big| - \big|\sum_{v\in V_{N_r}} w(v)\big|\big| \leq max(w(v)),\ v \in V_N$.

Proof 2. *Suppose vertices $V_N = \langle v_1, v_2, \ldots v_x\rangle$ are sorted according to latitude or longitude, there are $x - 1$ candidate positions to split V_N into V_{N_l} and V_{N_r}. There exists $1 \leq i \leq x$ where $\sum_{j=1}^{i} w(v_j) \leq \sum_{j=i+1}^{x} w(v_j)$ and $\sum_{j=1}^{i+1} w(v_j) \geq \sum_{j=i+2}^{x} w(v_j)$. We have either $\sum_{j=i+1}^{x} w(v_j) - \sum_{j=1}^{i} w(v_j) \leq w(v_i) \leq max(w(v))$ or $\sum_{j=1}^{i+1} w(v_j) - \sum_{j=i+2}^{x} w(v_j) \leq w(v_i) \leq max(w(v))$ where $v \in V_N$. Thus, $\min \big|\big|\sum_{v\in V_{N_l}} w(v)\big| - \big|\sum_{v\in V_{N_r}} w(v)\big|\big| \leq w(v_j) \leq max(w(v))$ where $v \in V_N$.* □

Now, the objective of partitioning graph G_N is to minimize Equation 1. To construct more balanced index structure, we present a dynamic partition strategy in Algorithm 1. We recursively partition a graph into two subgraphs G_{N_l} and G_{N_r} based on the partition algorithm, until the number of the vertices in the node is less than threshold η (i.e., a leaf node). During partitioning each graph G_N, we select the longer side (latitude or longitude) as the next partition side and divide the graph in this side with minimal cost according to Equation 1.

Example 9. *Consider Figure 4. As the latitude range is longer than longitude range, we select the vertical direction and split vertices in G into 9 vertices in G_1 and 8 vertices in G_2. The total trajectory weights on G_1 and G_2 are 32 and 31, respectively, where $|32 - 31| = 1$ is the minimal $\big|\big|\sum_{v\in V_{N_l}} w(v)\big| - \big|\sum_{v\in V_{N_r}} w(v)\big|\big|$. Similarly, G_1 is partitioned into G_3 and G_4. G_2 is partitioned into G_5 and G_6. N_1 (with level $\geq \theta_H = 1$) indexes t_2, t_3, t_5, t_6 and t_9 and the linkings between G_3 and G_4,* i.e., *$\{(v_6, v_8), (v_7, v_9)\}$. Each leaf node maintains no more than $\eta = 5$ vertices, e.g., N_3 maintains a sub-graph with 5 vertices.*

Complexity. Each time partitioning node N, there are two key steps: (1) sorting the vertices according to latitude or longitude (line 11 in Function `Split`), and (2) doing binary search on the proper partition position (line 12 in Function `Split`).

[Complexity (1)] For sorting V_N, each sorting complexity is $V_N \log(V_N)$. Suppose $|V| = n$, the worst case of sorting is:

$$C_1 \sum_{i=\eta}^{n} i \ln i \leq C_1\Big(\int_{\eta}^{n} x \ln x dx - \frac{1}{2} n \ln n\Big) \leq C_1\Big(\frac{n^2}{2}\ln n - \frac{n^2}{4}\Big)$$

The best case of sorting is:

$$\sum_{i=0}^{\lfloor log_2 \frac{n}{\eta}\rfloor} \frac{n}{2^i} \log_2 \frac{n}{2^i} \leq C_2 \sum_{i=0}^{\lfloor log_2 \frac{n}{\eta}\rfloor} \frac{n}{4^i} \ln n \leq \frac{4C_2}{3} n \ln n$$

where C_1 and C_2 are two constants w.r.t. used sorting algorithms and threshold η.

[Complexity (2)] For binary searching on V_N, each searching needs at most

$$\sum_{i=1}^{\lceil log_2 |V_N|\rceil} \frac{|V_N|}{2^{i-1}} = (1 - (\frac{1}{2})^{\lceil log_2 |V_N|\rceil})|V_N|$$

The worst case of searching is:

$$\sum_{i=\eta}^{n} (1 - (\frac{1}{2})^{log_2 i}) i \leq \sum_{i=\eta}^{n} i = \frac{(n+\eta)(n-\eta+1)}{2}$$

The best case of searching is:

$$\sum_{i=0}^{\lfloor log_2 \frac{n}{\eta}\rfloor} (1 - (\frac{1}{2})^{log_2 \frac{n}{2^i}}) \frac{n}{2^i} \leq \sum_{i=0}^{\lfloor log_2 \frac{n}{\eta}\rfloor} \frac{n}{2^i} \leq 2n.$$

Combining the above two complexities, the complexity of constructing RP-Tree for $|V| = n$ is $O(n^2 \log n)$.

η is used to determine the size of leaf nodes, the RP-Tree will be higher if η is smaller, thus a proper selection of η is also important for the speed of querying. Besides, the partitioning scheme can be altered and extended to support various constraints [14], [19], [20], such as minimizing cut edges and balancing the size of sub-partitions.

3.3 The Update of RP-Tree

When the trajectory dataset is updated, a batch of new raw trajectories will be map-matched first. Then each map-matched trajectory segment will be indexed on the edges it passes, *i.e.,* each $idx(e)$ of $e \in E$ will be updated. Next the weight of each subgraph G_N will be updated. As a

consequence, RP-Tree may become unbalanced with respect to the number of trajectories on each node. To rebalance RP-Tree when it is highly skewed, we re-partition influenced nodes as follows.

We first collect the set of all unbalanced nodes $\mathbb{N}$ after weights update of subgraphs according to the partition optimization metric. We then filter the least ancestors in $\mathbb{N}$, i.e, we drop N from $\mathbb{N}$ if there exists $N' \in \mathbb{N}$ and N' is the ancestor of N. Next, for each $N \in \mathbb{N}$, we adjust the partition position p of splitting N into sub-nodes according to the partition optimization metric (i.e, line 12 in Function `Split`). Specifically, if current weight of N_l is larger than weight of N_r, we decrease the value of p, otherwise, we increase p until we find a p with balanced partition. We finally repeat the partition recursively until meeting the stop threshold η. In this way, we effectively update RP-Tree for new incoming trajectories with optimized costs by only updating the influenced nodes.

4 Supporting Range Queries

In this section, we first discuss the spatial relations among ranges, edges, vertices and trajectories (Section 4.1). Then we discuss how to use RP-Tree to support single RQ (Section 4.2) and multiple RQs (Section 4.3). For ease of presentation, we mainly focus on presenting the spatial dimensions (longitude and latitude), which can be easily extended to support temporal dimension (longitude, latitude, and time).

4.1 Range Graph of r on G

We first categorize the relation of edge e and *spatial range* r into three types: *inside*, *intersect* and *outside* for better presenting our methods.

Definition 3 (Inside). *For edge $e = (v_i, v_j)$, if both v_i and v_j are within r, e is inside r. All the edges* inside *r are denoted as E_{ri}.*

Definition 4 (Intersect). *For edge $e = (v_i, v_j)$, if v_i is within r and v_j is outside r, or both v_i and v_j are outside r but $e \cap r \neq \varnothing$, e intersects r. All the edges cover r are denoted as E_{rc}.*

Definition 5 (Outside). *For edge $e = (v_i, v_j)$, if $e \notin (E_{ri} \cup E_{rc})$, e is outside r. $E_{ro} = E - E_{ri} - E_{rc}$.*

Note that an edge e inside a range r can be seen as a particular case of that e intersects r. Next, we introduce the concepts of range graphs.

Definition 6 (Range Graph). *A graph $G_r = (V_r, E_r)$ is the* range graph *of a spatial range r, where E_r is the set of all edges (v_i, v_j), when any of the following hold: (i) both v_i and v_j are inside r; (ii) only one of v_i and v_j is inside r; or (iii) both v_i and v_j are outside r but the edge (v_i, v_j) intersects r.*

Definition 7 (Inner Vertices/ Outer Vertices). *A vertex inside r is an inner vertex of r. V_{ri} is the set of all the inner vertices of r. $V_{ro} = V_r - V_{ri}$ is the set of outer vertices of r.*

4.2 Supporting Single RQ

For a single RQ with range r, we use RP-Tree to recursively find E_r. The answer to this RQ is: $T_r = \bigcup_{e \in E_r} tr(e)$.

Algorithm 2: *SRQ*($\mathcal{G}$, r, θ_H): Single Range Query

Input: RP-Tree $\mathcal{G}$, range r.
Output: T_r.

```
1 if G.border ∩ r = ∅ then
2     return ∅
3 else
4     return Search(G, r, h)
```

Function Search(N, r, h)

```
1 // Threshold θ_H is a global variable
2 if r = ∅ then
3     return ∅;
4 else if N.border ⊆ r and h ≥ θ_H then
5     return N.index;
6 else if N is leaf then
7     return ⋃_{e∈E_N∩r} tr(e)
8 else
9     return Search(N_l, N_l.border ∩ r, h + 1) ∪
        Search(N_r, N_r.border ∩ r, h + 1) ∪
        ⋃_{e∈L_N∩r} tr(e);
```

We present the pseudo-code in Algorithm 2. Note that given a node N, if range $N.range \subseteq r$ and $N.height \geq \theta_H$, we do not need to recursively visit descendants of N. Instead, we directly returns all trajectories in N, or all trajectories in its descendants with trajectory index if the level of N is lower than θ_H.

Example 10. *In Figures 2 and 4, consider query Q_1 with range r_1. r_1 intersects linkings of G_2 but does not overlap G_2. Thus, we check linkings (v_{10}, v_5) and (v_{11}, v_4) to get trajectories t_1, t_7 and t_{10}. Then, we consider the overlapped part with G_1, i.e., $r_1 \cap G_1$. Since $G_4.range \subseteq r_1$, we directly get $G_4.index = \{t_3, t_5, t_6, t_9\}$ and get $\{t_2, t_9\}$ by checking linkings of G_3. Finally, we get query result of Q_1: $\{t_3, t_5, t_6, t_9\} \cup \{t_2, t_9\} \cup \{t_1, t_7, t_{10}\} = \{t_1, t_2, t_3, t_5, t_6, t_7, t_9, t_{10}\}$.*

Note that our method does not need any "`refine`" operation for single RQ, because the trajectories have been indexed. Based on this advantage, RP-Tree outperforms existing indexes on single-range RQs.

Discussion. RP-tree can support circle range queries, which is also common in real-world applications. If the query range r is a circle, we use the minimum bounded rectangle which can hold r to approximate r in Algorithm 2. Then we can prune irrelevant trajectories which pass by the rectangle but do not pass by the circle (Line 7 in Function Search.

4.3 Multiple Spatial Range Queries (RQs)

For an RQ with range $R = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m\}$, we reduce it into m single RQs. For each single RQ with spatial range r_j (corresponding to spatio-temporal range $\mathbf{r}_j$), suppose the set of trajectories *intersect* r_j is T_{r_j}. We further compute the intersection between all single-range RQs. Specifically,

$$T_R = \bigcap_{j=1}^{m} T_{r_j} = \bigcap_{j=1}^{m} \bigcup_{e \in E_{r_j}} tr(e) \tag{2}$$

Example 11. *In Figure 2, consider queries Q_2, we can get $T_{r_1} = \{t_1, t_2, t_3, t_5, t_6, t_7, t_9, t_{10}\}$ and $T_{r_2} = \{t_2\}$. Thus we have $T_R = T_{r_1} \cap T_{r_2} = \{t_2\}$.*

Note that computing the union and intersection of many ID sets is computationally expensive. Thus, we propose an efficient method for speeding up the computation process. The basic idea is to prune some unnecessary edges within the ranges. To this end, we formulate the following lemma to prune such edges.

Lemma 3. *For query range r_j, find the set of edges E^*, such that for edge $e \in E^*$, either only one of v_i and v_j is inside r or both v_i and v_j are outside r while edge (v_i, v_j) intersects r. We have $\bigcup_{e \in E^*} tr(e) = \bigcup_{e \in E_{r_j}} tr(e)$.*

Proof 3. *In an RQ with multiple ranges, a trajectory t_k intersects range r_j should pass by one of the edges $(v_i, v_j) \in E^*$; if all of the points of t_k are inside r_j, t_k cannot intersect other ranges.* □

Minimum Multi-Way Cut (M^2C) Algorithm. The time overhead for computing multi-range RQs leveraging multiple times of single-range RQs can be very large when there are large sizes of trajectories queried for single-range RQs. We now introduce how to speed up multi-range RQs based on a minimum multi-way cut (M^2C) algorithm.

Given query ranges $R = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m\}$. Without loss of generality, R is given in chronological order, i.e., the first range is r_1 and the last range is r_m, and we have $R = \langle r_1, \{r_2, r_3, \ldots r_{m-1}\}, r_m \rangle$. Thus the ranges can be categorized into two types. The first type includes r_1 and r_m and the second type includes the rest. Consider the second type, each trajectory t *intersect* a second type range r must enter r via an $e_j \in E_{ro}$ and leave r via another edge $e_k \in E_{ro}$, i.e., both the pick-up and drop-off point of t_i is outside r. We denote the set of trajectories across r as $T_r^a \in T_r$. Then we compute T_r^a for $\{r_2, r_3, \ldots r_{m-1}\}$ (m-2 times) and compute T_r for r_1 and r_m in Equation 2. We now discuss how to effectively compute T_r^a. Intuitively, many arterial roads in a city are crucial for transportation. Most vehicles pass these roads otherwise they cannot reach their destinations. Motivated by this phenomenon, we are now ready to propose a minimum multi-way cut (M^2C) algorithm to prune the number of considered edges for computing T_r^a.

Definition 8 (Representative Graph)**.** *Find representative graph $G_r^* = (V_r^*, E_r^*)$, $E_r^* \subseteq E_r$ where if we eliminate all of the edges of $E_r^* \in E_r$, we can not start from v_j to v_k via a path p, where each point of p is within r, $\forall v_j, v_k \in V_{ri}$.*

If we get E_r^*, we can compute $\bigcup_{e \in E_r^*} tr(e)$ as $\bigcup_{e \in E_r} tr(e)$, because any trajectory t_i could not *intersect* r_j ($j \neq 1, m$) if edges in E_r^* are eliminated. We aim to find G_r^* with the minimum number of edges such that the computation cost can be greatly reducedf. Consider the following two cases:

(1) $\forall e_i, e_j \in E_{rc}$, e_i and e_j are not adjacent.

For this case, we randomly select $|E_{rc}|-1$ edges in E_{rc} as E'_{rc}, then $T_r^a = \bigcup_{e \in E'_{rc}} tr(e)$ based on the following Lemma:

Lemma 4. *The minimum size of E_r^* is $|V_{ro}|-1$, if $\forall e_i, e_j \in E_{rc}$, e_i and e_j are not adjacent.*

Proof 4. *For a connected graph, if we split it into k connected components, we should at least cut k-1 edges [46]. Cutting $|V_{ro}|-1$ edges in E_{ro} can split G_r into $|V_{ro}|$ connected components. Now we find a solution which can cut only $|V_{ro}|-1$ edges to split G_r into $|E_{ro}|$ components with the minimal size $|V_{ro}|-1$.*

(2) $\exists e_i, e_j \in E_{r_o}$ e_i and e_j are adjacent.

(I) If $|V_{r_o}| = 2$, we can reduce it to a Minimum Cutting Edges problem[1] by setting the weight of each $e_i \in E_r$ as 1. In a Minimum Cutting Edges problem, we regard $v_i \in V_{ro}$ as source node and another $v_j \in V_{ro}$ as destination node. Then the solution of Minimum Cutting Edges of G_r, is G_r^*. We can use Ford-Fulkerson[2] algorithm or its successor Edmonds-Karp[3] algorithm to find the minimum cut in time $O(|V_r||E_r|^2)$.

(II) If $|V_{r_o}| > 2$, we have following Lemma.

Lemma 5. *Finding G_r^* with minimum number of edges is NP-hard when $|V_{r_o}| > 2$.*

Proof 5. *Set the weight of $\forall e_i \in E_r$ as 1, it is a classical multi-way cut problem, it has been proved to be NP-hard [17].*

Since finding G_r^* with the minimum number of edges is NP-hard, we devise a heuristic algorithm. There are $|V_{ro}|$ outer vertices. We select one vertex $v_i \in V_{ro}$ as source node, dummy an infinite node V_∞ as destination node and link each $v_j \in V_{ro}$ where $v_i \neq v_j$ to V_∞. We set the weight of edge (V_∞, v_j) to $+\infty$. We eliminate the minimum cut in this graph and then we cannot travel from v_i to any v_j. Iteratively, we can find arteries E_r^* which may not be the optimal. The worst case is that we iterate all the $\frac{|V_{r_o}|(|V_{r_o}|-1)}{2}$ pairs of outer vertices, thus, the time complexity is:

$$t \leq \sum_{j=1}^{|V_{ro}|-1} O(j|E_r|^2) \leq O(\frac{|V_{ro}|(|V_{ro}|-1)}{2}|E_r|^2) = O(|V_{ro}|^2|E_r|^2)$$

Example 12 (M^2C algorithm)**.** *For example, in Figure 2, we first select v_2 as source node, dummy an infinite destination v_∞(the orange vertex) and link it to v_3, v_4 and v_5. The minimum cut is $\{e_2, e_3\}$. Then, v_2 can not reach v_3, v_4 and v_5. We iteratively use such strategy and finally find $E_r^* = \{e_2, e_3, e_{11}, e_{12}, e_{13}\}$.*

5 Supporting Approximate RQs

If the trajectory datasets are too large, the exact algorithm cannot answer an RQ query in seconds. To meet the high-performance requirement, we support *approximate range query* (ARQ) which finds approximate answers within an error bound instantly.

Definition 9 (Approximate RQs (ARQs))**.** *Given a set of trajectories $T = \{t_1, t_2, \ldots, t_n\}$, a target attribute attr, a set of query ranges $R = \{\boldsymbol{r}_1, \ldots \boldsymbol{r}_m\}$ and an error bound δ, it returns an approximate aggregation result $\hat{\texttt{AGG}}(T_R.attr)$ of exact $\texttt{AGG}(T_R.attr)$, such that $\frac{|\hat{\texttt{AGG}}(T_R.attr) - \texttt{AGG}(T_R.attr)|}{|\texttt{AGG}(T_R.attr)|} \leq \delta$ with a confidence no less than c.*

1. https://en.wikipedia.org/wiki/Minimum_cut
2. https://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm
3. https://en.wikipedia.org/wiki/Edmonds-Karp_algorithm

When there is only one spatio-temporal range, we call it a *single approximate aggregation spatio-temporal range query* (or a *single ARQ*). Note that only for aggregation queries, we apply approximate solutions. In other words, RQ can support both *selection* and *aggregation*, while ARQ can only support *aggregation*. For example, in a traffic flow control task, an analyst may need to analyze the density of real-time trajectories to make in-time decision, thus implementing approximate query processing (AQP) techniques on trajectory queries is essential [24]. We devise AQP algorithms by using a small subset of the trajectories to answer real-time RQs based on RP-Tree. Given an error bound δ, our method can determine which level of an RP-Tree is to be accessed to get enough number of trajectories to give a confident enough approximation for an ARQ. We first discuss the challenges of approximate analytical tasks on trajectory data based on the observation from real-world datasets. Then we show how to utilize RP-Tree to support ARQs.

5.1 Using Sampling to Support ARQs

Existing methods [26], [42] implement sampling techniques on online trajectory aggregation, where trajectories and spatial points are supposed to be distributed uniformly. They use minimum bounding rectangles (MBRs) or grids to index the points or trajectory segments. However, in real-world, some road segments are more likely to be crowded (*e.g.,* central business districts) while others are sparse (*e.g.,* suburban districts). To be more straightforward, we plotted the traffic flow density of three cities, Porto, Chengdu and Beijing, which will be used for experimental evaluations. The visualized results are shown in Figure. 5

Intuitively, stratified sampling would outperform random sampling in trajectory analytical tasks because the trajectory distribution is skewed. Hence, as we index trajectories by road segments, it is feasible to implement stratified sampling method on RP-Tree based on trajectory density of road segments. Moreover, RP-Tree uses an edge-balanced partition strategy, thus the partitioned sub-graphs have as close trajectory density as possible.

Suppose trajectory t has an attribute $t.attr$ of interest. We consider four fundamental types of aggregations, AVG, VARIANCE, COUNT and SUM on $t.attr$ which could be used to support other complex trajectory analytical tasks.

5.2 Using RP-tree to Support ARQs

Challenges. Generally speaking, any trajectory analytical aggregation of the whole population can be estimated from a uniform sample. The larger the sample size, the higher the approximation accuracy. However, as many trajectories are repeatedly indexed by passing by edges, random sampling in RP-tree cannot be seen as a uniformly random sampling with replacement, leading to difficulties in the error analysis of approximate queries.

Our method. Our partition method is based on edge-balanced strategy in Section 3, i.e., we give higher weights for more frequently used edges (deeper color parts in Figure 5). Thus, random sampling on RP-Tree can be seen as uniformly sampling with replacement. We propose an approximate estimation algorithm for single ARQs in Algorithm 3. Multiple ARQs can be supported similarly.

Algorithm 3: *Single_ARQ*($\mathcal{G}$, r, $attr$, δ): Approximate Single Range Query.

Input: RP-Tree $\mathcal{G}$, range r, attribute $attr$, error threshold δ.
Output: $\hat{\texttt{AVG}}(T.attr)$, $\hat{\texttt{VARIANCE}}(T.attr)$, $\hat{\texttt{SUM}}(T.attr)$, $\hat{\texttt{COUNT}}(T.attr)$ with confidence interval.

// Inside blocks $\mathcal{B}_I$, outside blocks $\mathcal{B}_O$
$\mathcal{B}_I = \varnothing$, $\mathcal{B}_O = \varnothing$, threshold θ_H;
ApproSearch($\mathcal{G}$, r, θ_H);
S_r =$RandomSample(\mathcal{B}_I, \delta, c)$;
$\hat{\texttt{AVG}}(T_r.attr) = \hat{\mu}(S_r), \hat{\texttt{VARIANCE}}(T_r.attr) = \hat{\sigma}^2(S_r)$;
// Lower bound $\mathbb{LB}$, upper bound $\mathbb{UB}$;
$\mathbb{LB}$=$\bigcup_{e\in\mathcal{B}_I}$ tr(e), $\mathbb{UB}$=$\bigcup_{e\in\mathcal{B}_O}$ tr(e);
for *i in range(Len($\mathcal{B}_I$))* **do**
  if $\frac{|\mathbb{UB}|-|\mathbb{LB}|}{|\mathbb{LB}|} \leq \delta$ **then**
    $\hat{\texttt{COUNT}}(T_r.attr)$=[$|\mathbb{LB}|, |\mathbb{UB}|$];
    break;
  else
    $\mathbb{LB} = \mathbb{LB} \cup (\mathcal{B}_O[i] \cap r)$, $\mathbb{UB} = \mathbb{UB} - (\mathcal{B}_O[i] \cap r)$;
$\hat{\texttt{SUM}}(T_r.attr)$ = $\hat{\texttt{AVG}}(T_r.attr)$ $*\hat{\texttt{COUNT}}(T_r.attr)$;
return $\texttt{AVG}'(T_r.attr), \hat{\texttt{VARIANCE}}(T_r.attr)$, $\hat{\texttt{SUM}}(T_r.attr), \hat{\texttt{COUNT}}(T_r.attr)$.

Function ApproSearch($\mathcal{N}$, r, θ_H)

if $N.border \subseteq r$ *and* $N.height \geq \theta_H$ **then**
  $\mathcal{B}_I$ += N
else if $\mathcal{N}$ *is leaf* **then**
  $\mathcal{B}_O$ += N
else
  ApproSearch(N_l, $N_l.border \cap r$);
  ApproSearch(N_r, $N_r.border \cap r$);
$\mathcal{B}_I$ += $\bigcup_{e\in\mathbb{L}_N\cap r} tr(e)$;

5.2.1 Single ARQs

(i) Unbiased Estimator of AVG and VARIANCE. Suppose we estimate the mean of $t.attr$ for the trajectories meet with query range r, i.e., $\texttt{AVG}(T_r.attr)$. Once we get a sample S_r from the population T_r, we give a closed-form estimation:

$$\hat{\texttt{AVG}}(T_r.attr) = \mathcal{E}[\frac{1}{|S_r|}\sum_{t\in S_r} t.attr] = \hat{\mu}$$

Similarly, the $\texttt{VARIANCE}(T_r.attr)$ can be estimated as the population standard deviation:

$$\hat{\texttt{VARIANCE}}(T_r.attr) = \mathcal{D}(T_r.attr) = \mathcal{E}[(T_r.attr - \hat{\mu})^2] =$$
$$\mathcal{E}[\frac{1}{|S_r|-1}\sum_{t\in S_r}(t.attr - \mu)^2] = \hat{\sigma}^2$$

To compute the confidence interval, we refer to [12], [13]. A simple and effective method is to use Gaussian distribution to give a closed form of confidence interval rather than bootstrap confidence interval [15], [51] (which are too time-consuming for online analytical tasks). According to the central limit theorem, if $|S_r|$ is large enough, we can use Gaussian distribution $N(\hat{\mu}, \hat{\sigma}^2)$ to approximate

(a) Porto *(b) Chengdu* *(c) Beijing*

Fig. 5. Trajectory Spatial Distribution Density Example.

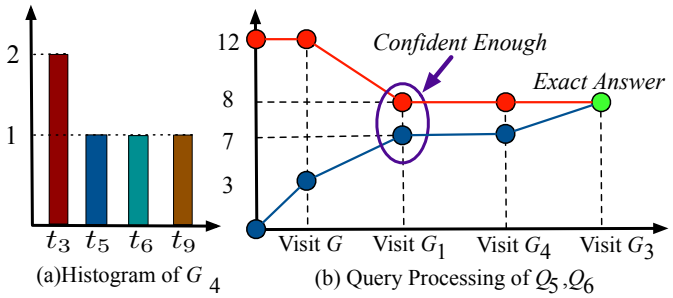

Fig. 6. Approximate Query Example.

the distribution of $T_r.attr$. Since $\hat{\mu}$ and $\hat{\sigma}^2$ are unbiased estimator of AVG and VARIANCE, such stratified method can give theoretical guarantee with even a small sample size [5].

Next, we discuss how to get sample S_r. We present the pseudocode in Algorithm 3. We search the RP-Tree and recursively collect the nodes whose spatial range $N.border \subseteq r$ to get the list $\mathcal{B}_I$. Then we randomly select trajectories from S_r, from which we give answer within an error bound δ with a certain confidence interval. Notice that the difference between sampling with/without replacement are not significant [13] when estimating SUM and VARIANCE. Thus, even the sampled nodes include repeated trajectories, we can regard it as sampling with replacement and the result will still be confident, as we give higher weight for trajectories with more edges in RP-Tree.

As for supporting random sampling, we build histogram for each node N by counting the weight of each trajectory in N. For example, G_4 indexes trajectory $\{t_3, t_5, t_6, t_9\}$. t_3 passes two edges, $e_{v_8 v_{10}}$ and $e_{v_{10} v_{11}}$, thus the weight is 2. When we conduct random sampling from G_4, t_3 has the probability $\frac{2}{1+1+1+2} = 40\%$ to be selected.

(ii) Deterministic Estimator of COUNT. Computing COUNT is always hard because the accuracy relies on the trajectories' distribution. Existing works use a grid-based method or R-tree, suppose query range is r, these methods select a range $r' \subset r$ and compute $\texttt{COUNT}(T_r) = \texttt{COUNT}(T_r') * \frac{area(r)}{area(r')}$ where $area(r)$ denotes the area of range r. The drawbacks are obvious. First, trajectories will repeatedly occur in different regions, i.e., though $t_i \hookrightarrow r'$, it is possible that $t_i \hookrightarrow r - r'$. Second, the trajectories' distribution is not uniformly distributed as we observed. Thus, we cannot make assumption that $|T_r| \propto area(r)$.

A deterministic approximate query processing can give the bound of an aggregation with 100% confidence. Such methods are easy to implement on relational databases, e.g., DAQ [36] which transforms data to bit slices and compute the answer from low bits to high bits. However, such methods cannot be used on existing spatial indexes. To tackle the above limitations, we propose a deterministic estimator for COUNT based on RP-Tree.

Based on Algorithm 3 (lines 6-13), by finding all the blocks in $\mathcal{B}_I$ and $\mathcal{B}_O$, we can tighten the difference between lower bound $\mathbb{LB}$ and upper bound $\mathbb{UB}$ of the selectivity. After the difference is small enough, we will end up the querying processing. For example, for Q_5, in Figure 6 (b), we first initialize $\mathbb{UB} = 12$ and $\mathbb{LB} = 0$ of T_r. After accessing G, we find linkings $(v_{11}, v_4), (v_{10}, v_5)$ and (v_6, v_3) and corresponding trajectories $\{t_1, t_7, t_{10}\}$, and G_1 indexes $\{t_2, t_3, t_5, t_6, t_9\}$. Thus, the result is updated to $\mathbb{UB} = 8$ and $\mathbb{LB} = 3$. After accessing G_1, we know t_3, t_5, t_6 and t_9 indexed by (v_7, v_9) and (v_6, v_8). We only do not know whether t_2 *intersects* r, where t_2 in G_3. However, we are satisfied with the counting number interval $[7, 8]$ and stop the querying without accessing G_3 and G_4 for corresponding details. Here, our method can always give an approximation with 100% confidence of COUNT aggregation. Thus, the approximation can give an answer with error less than 15%, and with 100% confidence.

(iii) Hybrid Estimator of SUM. Once the COUNT and AVG have been given, it is easy to give an hybrid estimator of SUM by combining unbiased statistical estimator and deterministic estimator. $\texttt{SÛM}(T_r.attr) = \texttt{AV̂G}(T_r.attr) * \texttt{COÛNT}(T_r.attr)$. The confidence is the product of the statistical confidence and the deterministic estimator's confidence (e.g., 95% * 100%). Besides, we give an answer interval of $[\texttt{AV̂G}(T_r.attr).\text{lower} * \mathbb{LB}, \texttt{AV̂G}(T_r.attr).\text{upper} * \mathbb{UB}]$. For example, for Q_6, by using answer from Q_4 and Q_5, we can guarantee that the answer is within $[44.1, 52.0]$ with a confidence of 98%.

5.2.2 ARQs

We briefly discuss how to support RQs with multiple ranges in RP-Tree. For query with $R = \{r_1, r_2, \ldots, r_m\}$, we conduct single RQ for m times and compute the results. As the computation of intersection of m sets will reduce the selectivity of query result, it will be harder to get enough samples than single RQs. Nevertheless, we can cache the intermediate results during computing RQ with multiple ranges and dynamically update one result of single RQ for r_i. Based on cached intermediate results, we do not need to compute the results again, which speeds up the process.

5.3 Interactive Trajectory Query Processing

In interactive analytical tasks, some users may change the query conditions during query execution. To meet users' demands, interactive approximate trajectory [24] range query processing is needed. We discuss the scenarios of how to use RP-Tree to support two types of interactive patterns, i.e., stop the query when satisfied with the current results and add query ranges during the query execution.

(1) Satisfy with The Current Answer. If the searching algorithm accesses more blocks, we then get a more reliable result. By continuously searching, we will find more trajectories, and the confidence will be higher. When the analyst is satisfied with the current answer, the user can stop the query processing and use the current approximate answer to support further analyses. Specifically, at time t_k, the user has gotten the aggregation answer $\hat{\texttt{AGG}}$ and can stop the query processing. Otherwise, we will give a better estimation of the result at t_{k+1} based on the result at t_k, i.e., the AQP answer on RP-Tree is incrementally updated. For approximating `AVG` and `VARIANCE`, we give higher accuracy by sampling more tuples (line 4 in Algorithm 3). For approximating `COUNT`, we give higher accuracy by dynamically reducing δ (line 9 in Algorithm 3). For approximating `SUM`, we can alternately update the answer of `AVG` and `COUNT`.

(2) Add Query Range in RQ. During the query processing, the analyst can add new ranges for more specific queries. When a set of new query range R' is added, the query process can be regarded as two steps. First, it finds a sample of $T_{R'}$, trajectories that intersect R' and computes an approximate answer based on the current answer. Then it converts the current query range R to $R \cup R'$ and continues the new query in an interactive pattern.

6 Experimental Evaluation

We will introduce baselines in Section 6.1 and experimental settings in Section 6.2. We will then present RP-Tree evaluation results on supporting RQ(s) in Section 6.3 and ARQ in Section 6.4, respectively. The evaluation of indexing performance will be discussed in Section 6.5.

6.1 Baselines

We categorized baselines into two types, *point-based* indexes and *trajectory-based* indexes. Our method is *road-based* index. We also built a simple *road-based* index as a baseline to validate the performance boost brought by RP-Tree and corresponding querying algorithms.

Point-Based Indexes. Point-based indexes stored the spatial points in raw trajectory into blocks based on locality. The trajectory identifiers were embedded into the spatial points. Here, we mainly considered R-Tree [11] and grid-based indexes [55] denoted as `R-Tree` and `Grid` in experiments.

Trajectory-Based Indexes. Contrasting with point-based indexes, trajectory-based methods maintained the relevance of points in a raw trajectory. For exact RQs, we considered four baselines. `SETI` [2]: It partitioned the space into cells without any overlaps. A raw trajectory was stored as segments which were split by boundaries of different cells. Each segment was a sequence of points. `STR-Tree` [34]: Similar to `SETI`, `STR-Tree` indexed the space into many MBRs and stored the trajectory segments in leaf nodes. `TB-Tree` [34]: Based on `STR-Tree`, `TB-Tree` ensured that segments of the same trajectory would be stored in the same leaf node. Leaf nodes indexing the same trajectories would be linked by two-way pointers. `LoTI` [47]: A two-level index which stored trajectory segments in MBRs in the first level, and indexed the trajectories in each leaf-node MBR as a Bucket Quadtree [38] in the second level. To support ARQ, we implemented `RIS` [26] on different partition strategies. Using `RIS` on grids and `R-Tree` were denoted by `RIS-G` and `RIS-R` in experimental results respectively.

Road-Based Index. In order to evaluate how much the structure of RP-Tree index boost the query performance, we also build a road-based index as baseline. This method simply uses map-matched trajectories to answer queries and use grids to index the trajectory segments without using graph partition for indexing and pruning algorithms in Section 4. We denote this map-matched index as `MMI`.

6.2 Setting

Datasets. The experiment was conducted on three real-world trajectory datasets. Porto[4]: taxi trajectories of Porto, Portugal from Kaggle, Chengdu[5]: taxi trajectories of Chengdu, China from Didi open data and Beijing[6]: taxi trajectories of Beijing, China [39]. The road networks of Porto and Chengdu were downloaded from OpenStreetMap[7], and the road network of Beijing was as used in [39]. The details of the datasets were shown in Table 2.

Queries. We set the query range size as $1km - 5km$ according to the default measuring scale in Baidu Map[8]. For single range queries (RQ or ARQ), we selected 40 ranges at downtown and 10 ranges near suburb. For multiple range queries (RQs or ARQs), the number of ranges was selected as $m \in \{1, 2, 3, 4, 5\}$. We randomly generated 50 combinations of m ranges from above 50 ranges.

We varied three features: (1) Range Size. We set the range as a rectangle with the length of one side from 1 kilometre to 5 kilometre. (2) Range number. The number of query ranges from 1 to 5 as above setting. (3) Sampling rate. In all the ARQ(s) methods, trajectories with different IDs were stored in different blocks (leaf MBRs, grids or RP-Tree leaf nodes), the sampling rate was the ratio of accessed blocks.

Metrics. For exact queries, we mainly focused on query time (in seconds). For approximate queries, we compared different methods in two aspects. (1) Query time (in seconds) (2) Query answer accuracy (the rate of aggregated trajectories that are real answers). For indexing performance, we compared different methods in two aspects. (1) Index building time (in minutes). (2) Index storage (in GB).

Data Preprocessing. To better support our experiment, we removed spatial outliers from a trajectory by following T-Drive [50] and GeoLife [54]. Also, we followed ST-Matching [27], a hidden markov based model, to map

4. https://www.kaggle.com/crailtap/taxi-trajectory/home
5. https://outreach.didichuxing.com/research/opendata/
6. http://more.datatang.com/en
7. https://www.openstreetmap.org/
8. https://map.baidu.com

TABLE 2
Three Datasets Description.

Dataset	Cardinality	Size	AvgLen(#Points)	#Drivers	Time	Data Source	Road Network Size
Porto	1,710,670	1.94G	48.7	442	One Year	Kaggle	$\|V\| = 12751, \|E\| = 28411$
Chengdu	15,316,372	110G	37.4	15378	One Months	Didi Open Data	$\|V\| = 52198, \|E\| = 159625$
Beijing	11,114,613	79GB	22.2	8562	Six Months	Official Institute	$\|V\| = 25906, \|E\| = 58831$

trajectories onto road network. We used Valhalla[9], an open source map-matching tool for OpenStreetMap data. The data preprocessing time would be accounted into offline indexing time.

Implementation. The experiment was performed on a server using an Intel(R) Xeon(R) E5-2630 v4 CPU with 10 cores, 128G RAM running Ubuntu 16.04.5 LTS, implemented in C++. Before conducting the queries, indexes were resident in main memory. We conducted all the experiments for 10 times and computed the average results.

6.3 Exact Range Query

6.3.1 Varying Range Size for Single Range query

We observed that when we linearly varied the range size, the increasing of query time was not apparent, thus we varied the range size as $(1km)^2, (2km)^2 \dots (5km)^2$ spatial rectangles. The results were shown in the first column in Figure 7. We had the following observations.

(1) With the increasing of the query range size, the query time increased because all the methods need time for searching more trajectories in the indexes.

(2) RP-Tree outperformed all existing work due to road-network-based method did not need to recheck the candidate trajectories during the query processing and could be further optimized for query processing.

(3) RP-Tree was 10x faster than trajectory-based methods while 100x faster than point-based methods, it was because point-based methods needed more time to check each of the candidate points but trajectory-based methods would stoped refining a candidate when it was determined to be contained in the query answer.

(4) Query time of RP-Tree increased more slowly than existing methods. Because in most cases, RP-Tree only needed to compute outer edges in E^* which was linearly increased as the size of query range polynomially increased. But the query time of other methods increased with the same scale of increasing query range.

(5) RP-Tree was 2-8x faster than MMI, because performance boost came not only from map matching but also the index structure of RP-Tree and the judiciously designed query-answering algorithms, which increased the retrieving speed. The graph partition structure could help prune irrelevant trajectories during the query process.

Other Observations. For single RQ, we also compared RP-Tree with existing methods on varying different scalability and trajectory density (figures were omitted). We had the following observations.

(1) RP-Tree was 10x faster than other methods in query time when the dataset size is large. As with large data, they took much time on "refine" operation, but RP-Tree used pruning which pruned considered redundant edges and reduced the computation. The larger the data scale, the better RP-Tree's performance than other methods. As it took more time computing intersection and union of integer sets, whose time cost was sub-linear to data scale changes.

(2) RP-Tree outperformed existing methods for both dense and sparse trajectory densities. The query time of other methods increased sharply for datasets of dense trajectories because they needed to process more trajectories to get the answer, i.e., their computation complexities were subject to the trajectory density for each MBR or gird.

Summary. RP-Tree was 10x-100x faster than existing methods and was not sensitive to query range size, dataset size, and trajectory density.

6.3.2 Varying Range Number for Multiple Range Query

For RQs with multiple ranges, we varied the number of query ranges from 1 to 5 and the results were shown in Figure 7. We had the following observations.

(1) With the increasing of range number, query time of all methods increased because we computed more number of single RQ when the number of query ranges was larger.

(2) With the increasing of range number, the query time of RP-Tree increased more slowly because we computed less intersections when the number of query ranges was larger.

(3) The linearly increased query time of RP-Tree was hardly affected by the number of query ranges. Because the size of intermediate results was small enough thus was hardly affected by the number of ranges. By using the pruning algorithm, RP-Tree outperformed existing methods.

6.4 Approximate Range Query

Varying Range Number. As single ARQ was simple, we evaluated ARQ with multiple query ranges. We compared RP-Tree, `RIS-G`, `RIS-R` by varying the query ranges from 1 to 5 and the sampling rate was 30%. We computed both the accuracy and query time.

The query time of all methods were shown in the third column in Figure 7. We observed that RP-Tree used less query time than `RIS-G` and `RIS-R` as many trajectory IDs could be directly obtained in the non-leaf nodes of RP-Tree. Thus, query time on ARQs for RP-Tree was still sub-linearly increasing but was linearly increasing for other methods.

The accuracy of all methods for supporting ARQ were shown in the fourth column in Figure 7. We found that even we used just a small fraction of the whole dataset, we could get high accuracy when the query range number was small. However, when the query range number was large, the selectivity of an RQ with multiple ranges would be smaller, thus the accuracy would be smaller. RP-Tree outperformed `RIS-G` and `RIS-R` on accuracy even with higher query speed.

9. https://github.com/valhalla/valhalla

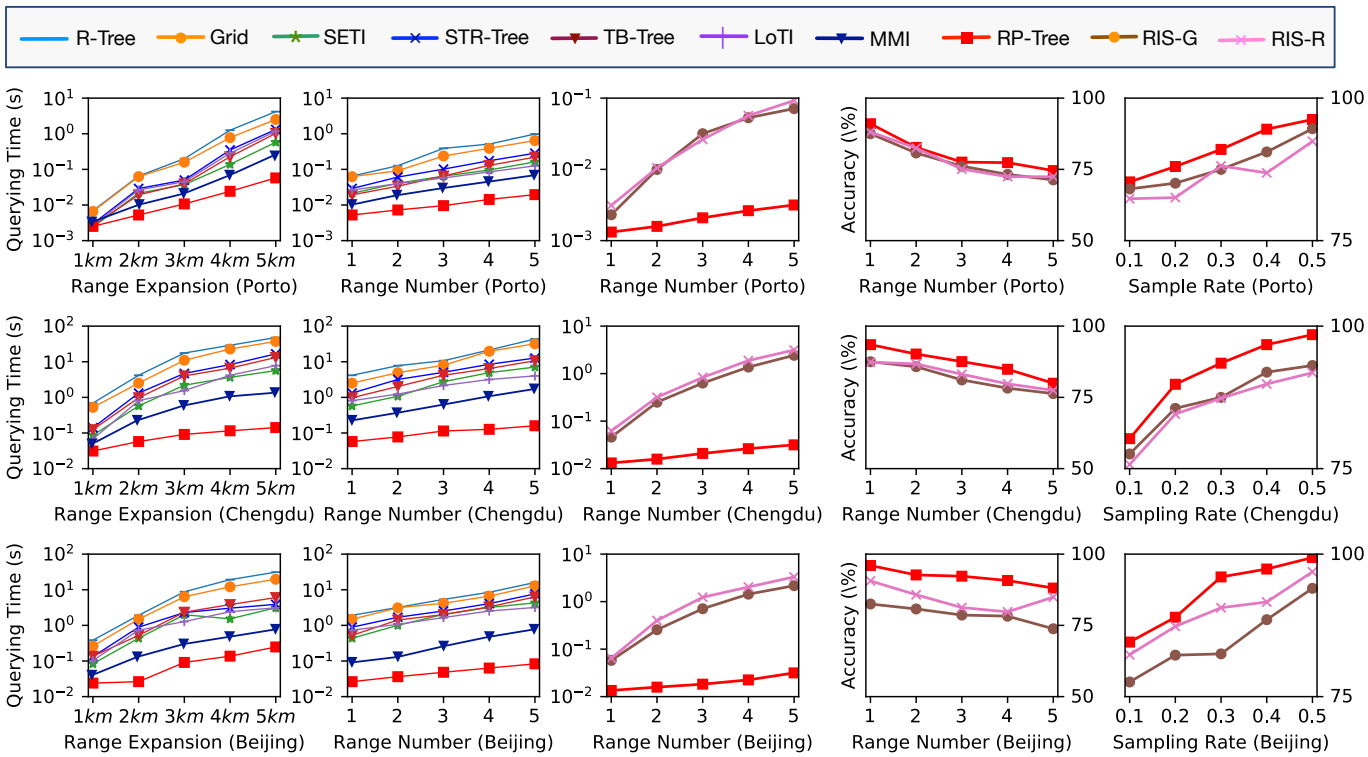

Fig. 7. RQ and ARQ Performance.

Varying Sampling Rate. The accuracy of all methods for supporting ARQs were shown in the fifth column in Figure 7. We had the following observations.

(1) RP-Tree had accuracy of more than 90% in most cases and when the sampling rate was more than 30%, the accuracy of RP-Tree approached 100%. Because the data in our proposed method was uniformly distributed, thus random sampling performed well.

(2) `RIS-G` and `RIS-R` performed badly in accuracy due to unreasonable assumption on trajectory distribution.

(3) `RIS` performed worse in `Porto` dataset than `Beijing` and `Chengdu` because the city of `Porto` was small and the distribution of trajectories was skewed but the city of `Beijing` and `Chengdu` were more uniform in trajectory distribution (see Figure 5).

Other Observations. When supporting ARQ with multiple ranges, `RIS-G` and `RIS-R` performed bad for supporting queries with more than one range with accuracy less than 60% in `Porto` and less than 70% in `Beijing` and `Chengdu`. Because the error would be multiplied when each of the single RQ of a multiple RQ had error when we used a uniform random sampling in terms of longitude and latitude. We studied the effect of error threshold δ. The experimental results were shown in Figure 9. As δ was highly related to the sampling rate in approximate trajectory range queries, the accuracy and query time decreased as δ increased.

6.5 Index Evaluation

We evaluated the offline indexing time and index space in Figure 8. As existing methods were mainly based on R-tree and grid-based index, thus we only considered R-Tree, grid-based index and newly proposed LoTI here.

Indexing Space. We had the following observations for indexing space.

(1) RP-Tree used less than one fifth storage than `R-Tree`, `Grid` and LoTI for all datasets. Because all the trajectories could be represented by a sequence of road segment IDs when using `RP-Tree` and the road segments were static.

(2) LoTI used more storage than R-tree and grid-based index, because it used a two-level index which maintained close trajectory segments.

Indexing Time. We had the following observations.

(1) The indexing time of `RP-Tree` was less than `R-Tree` and `Grid` because `R-Tree` and `Grid` needed to split one trajectory into segments and stored them in MBRs or grids but `RP-Tree` only needed to index map-matched road segments. Besides, `R-Tree` needed more time to maintain a dynamic index. (2) RP-Tree needed a little more indexing time than `LoTI` as it needed data preprocessing.

7 Related Work

7.1 Spatial and Spatio-temporal RQs

As illustrated in Section 6.1, existing methods of indexing and retrieving trajectory data mainly use `R-Tree` [1], [11] and `Grid` [2], [4], [7], [8], [35], [44], [49] or their variants [9], [16], [23], [29], [30], [33], [37], we categorize these methods as *point-based* and *trajectory-based* indexes and our method is a *road-based* index.

For spatiotemporal range queries, existing methods mainly use two variants of `R-Tree`: `Augmented R-Tree` and `Multi-version R-Tree`.`Augmented R-Tree` regards time as the third dimension along with latitude and longitude, then it builds a 3D-R-Tree to index all of the trajectory segments [34]. `Multi-version R-Tree` partitions the time dimension into many time slots and builds `R-Tree`

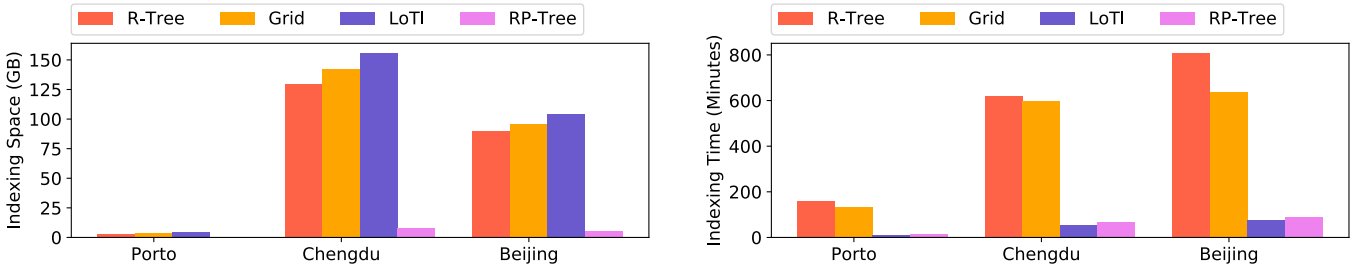

Fig. 8. RP-Tree Indexing Performance.

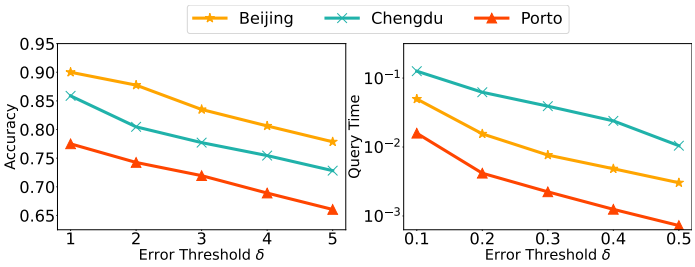

Fig. 9. Varying Error Threshold δ.

for latitude and longitude in each time slot such as HR-Tree [32], its successors HR+-Tree [40] and MV3R-Tree [41]. For spatio-temporal range queries, Grid-based methods build `R-Tree` for starting time and end time of trajectory segments in each grid [43]. Similar to existing methods, RP-Tree builds `R-Tree` for starting time of trajectory segments in each road segment.

There are many other systems which can support RQ(s), for example, SharkDB [52] is a column-oriented system for trajectory analyses but can not directly support RQs with multiple ranges. There are also abundant works on trajectory data management and mining [18], [45]. For example, trajectories can be used to refine traffic flow and model time-dependent road network [45], traffic data forecasting [10], destination prediction [48] and speed prediction [28].

7.2 AQP for Spatio-temporal and Trajectory Databases

Few studies focus on using AQP techniques to support spatial, spatio-temporal and trajectory *online analytical processing (OLAP)*. Wang et al. [42] propose to use AQP algorithms to support online spatial queries. However, it can not straightforwardly support range query on trajectories. RIS [26] uses naive random sampling on three-dimension R-Tree to support trajectory range queries, which performs bad on real trajectory applications due to unreasonable assumption (uniform distribution) on the distribution of trajectories. Instead, trajectories near city business districts will be denser than suburb, *i.e.*, the trajectory distribution is skewed in practice. Thus, based on stratified online sampling and deterministic approximation, RP-Tree outperforms existing arts. For more AQP related techniques on these topics, readers may refer to [3], [21], [24], [25], [31].

8 Conclusion

In this paper, we studied solving spatio-temporal range query on trajectory database and proposed a novel index called RP-Tree by partitioning road network and supporting trajectory range query in seconds. We presented how to construct an RP-Tree with the best performance offline and designed pruning algorithms to optimize the process of single range query and multiple range query based on RP-Tree online. Besides, we devised approximate query processing algorithms for interactive online aggregation based on `RP-Tree`. Extensive experiments verified the effectiveness of RP-Tree and showed that RP-Tree outperformed state-of-the-art algorithms for supporting online range query processing on real-world trajectory datasets.

Acknowledgments. This paper was supported by NSF of China (61925205, 62232009, 62072261), Huawei, and TAL education.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.
[2] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.
[3] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 511–519, 2017.
[4] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266, 2010.
[5] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.
[6] H. Crosby, T. Damoulas, and S. A. Jarvis. Embedding road networks and travel time into distance metrics for urban modelling. *International Journal of Geographical Information Science*, 33(3):512–536, 2019.
[7] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.
[8] S. Dai, Y. Yu, H. Fan, and J. Dong. Spatio-temporal representation learning with social tie for personalized POI recommendation. *Data Sci. Eng.*, 7(1):44–56, 2022.
[9] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. SEAL: spatio-textual similarity search. *Proc. VLDB Endow.*, 5(9):824–835, 2012.
[10] S. Guo, Y. Lin, S. Li, Z. Chen, and H. Wan. Deep spatial–temporal 3d convolutional neural networks for traffic data forecasting. *IEEE Transactions on Intelligent Transportation Systems*, 20(10):3913–3926, 2019.
[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[12] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, pages 51–63, 1997.
[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.
[14] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. *SC*, 95(28):1–14, 1995.
[15] T. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. In *The American Statistician*, volume 69, pages 371–386, 2015.

[16] H. Hu, G. Li, Z. Bao, J. Feng, Y. Wu, Z. Gong, and Y. Xu. Top-k spatio-textual similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(2):551–565, 2016.
[17] T. C. Hu and R. D. Young. *Integer programming and network flows*. Addison-Wesley Pub. Co., 1969.
[18] S. Huang, Y. Wang, T. Zhao, and G. Li. A learning-based method for computing shortest path distances on road networks. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 360–371. IEEE, 2021.
[19] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 29–29. IEEE, 1995.
[20] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 28–28. IEEE, 1998.
[21] T. Kraska. Approximate query processing for interactive data science. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, page 525, 2017.
[22] D. LaSalle and G. Karypis. A parallel hill-climbing refinement algorithm for graph partitioning. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*, pages 236–241, 2016.
[23] G. Li, J. Feng, and J. Xu. DESKS: direction-aware spatial keyword search. In *ICDE*, pages 474–485, 2012.
[24] K. Li and G. Li. Approximate query processing: What is new and where to go? *Data Science and Engineering*, Sep 2018.
[25] K. Li, Y. Zhang, G. Li, W. Tao, and Y. Yan. Bounded approximate query processing. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2018.
[26] Y. Li, C. Chow, K. Deng, M. Yuan, J. Zeng, J. Zhang, Q. Yang, and Z. Zhang. Sampling big trajectory data. In *CIKM*, pages 941–950, 2015.
[27] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate GPS trajectories. In *ACM-GIS*, pages 352–361, 2009.
[28] X. Ma, H. Zhong, Y. Li, J. Ma, Z. Cui, and Y. Wang. Forecasting transportation network speed using deep capsule networks with nested lstm models. *IEEE Transactions on Intelligent Transportation Systems*, 22(8):4813–4824, 2020.
[29] A. R. Mahmood, S. Punni, and W. G. Aref. Spatio-temporal access methods: a survey (2010 - 2017). *GeoInformatica*, 23(1):1–36, 2019.
[30] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.
[31] B. Mozafari. Approximate query engines: Commercial challenges and research opportunities. In *SIGMOD*, pages 521–524, 2017.
[32] M. A. Nascimento and J. R. O. Silva. Towards historical r-trees. In *Proceedings of the 1998 ACM symposium on Applied Computing, SAC'98, Atlanta, GA, USA, February 27 - March 1, 1998*, pages 235–240, 1998.
[33] L. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, 2010.
[34] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *VLDB*, pages 395–406, 2000.
[35] I. S. Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial. PARINET: A tunable access method for in-network trajectories. In *ICDE*, pages 177–188, 2010.
[36] N. Potti and J. M. Patel. DAQ: A new paradigm for approximate query processing. *PVLDB*, 8(9):898–909, 2015.
[37] S. Ranu, D. P, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, pages 999–1010, 2015.
[38] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
[39] Z. Shang, G. Li, and Z. Bao. DITA: distributed in-memory trajectory analytics. In *SIGMOD*, pages 725–740, 2018.
[40] Y. Tao and D. Papadias. Efficient historical r-trees. In *SSDBM*, pages 223–232, 2001.
[41] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.
[42] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. *PVLDB*, 9(3):84–95, 2015.
[43] L. Wang, Y. Zheng, X. Xie, and W. Ma. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. In *MDM*, pages 1–8, 2008.
[44] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, M. Sanderson, and X. Qin. Answering top-k exemplar trajectory queries. In *ICDE*, pages 597–608, 2017.
[45] Y. Wang, G. Li, and N. Tang. Querying shortest paths on time dependent road networks. *Proceedings of the VLDB Endowment*, 12(11):1249–1261, 2019.
[46] D. B. West. *Introduction to Graph Theory, 2/E*. 2004.
[47] M. Yadamjav, F. M. Choudhury, Z. Bao, and H. Samet. Efficient multi-range query processing on trajectories. In *ER*, pages 269–285, 2018.
[48] Z. Yang, H. Sun, J. Huang, Z. Sun, H. Xiong, S. Qiao, Z. Guan, and X. Jia. An efficient destination prediction approach based on future trajectory prediction and transition matrix optimization. *IEEE transactions on knowledge and data engineering*, 32(2):203–217, 2018.
[49] H. Yuan and G. Li. A survey of traffic prediction: from spatio-temporal data to intelligent transportation. *Data Sci. Eng.*, 6(1):63–85, 2021.
[50] J. Yuan, Y. Zheng, X. Xie, and G. Sun. T-drive: Enhancing driving directions with taxi drivers' intelligence. *IEEE Trans. Knowl. Data Eng.*, 25(1):220–232, 2013.
[51] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 277–288, 2014.
[52] B. Zheng, H. Wang, K. Zheng, H. Su, K. Liu, and S. Shang. Sharkdb: an in-memory column-oriented storage for trajectory analysis. *World Wide Web*, 21(2):455–485, 2018.
[53] Y. Zheng. Trajectory data mining: An overview. *ACM TIST*, 6(3):29:1–29:41, 2015.
[54] Y. Zheng, X. Xie, and W. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.
[55] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.
[56] R. Zhong, G. Li, K. Tan, and L. Zhou. G-tree: an efficient index for KNN search on road networks. In *CIKM*, pages 39–48, 2013.

Yong Wang is currently a Ph. D. student in the Department of Computer Science, Tsinghua University, Beijing China. His research interesting includes spatio-temporal data, differentially private database and robust machine learning

Kaiyu Li received the bachelor's degree from the Department of Computer Science and Technology, Harbin Institute of Technology, China. He received his Ph.D. degree in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include approximate query processing, data integration and crowdsourcing.

Guoliang Li is a professor in the Department of Computer Science, Tsinghua University, Beijing, China. He received his Ph.D degree in Computer Science from Tsinghua University, Beijing, China in 2009. His research interests mainly include data cleaning and integration, spatial databases and crowdsourcing.

Nan Tang is a senior scientist at Qatar Center for Artificial Intelligence, QCRI, HBKU, Qatar. His research interests center around data preparation. He got his PhD. degree from The Chinese University of Hong Kong, China. Prior to joining QCRI, he was a Research Fellow at Laboratory for Foundations of Computer Science at the University of Edinburgh, Edinburgh, UK.