2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00067

REPOSE: Distributed Top-k Trajectory Similarity Search with Local Reference Point Tries

Bolong Zheng[1], Lianggui Weng[1], Xi Zhao[1], Kai Zeng[2], Xiaofang Zhou[3], Christian S. Jensen[4]
[1]Huazhong University of Science and Technology, Wuhan, China
Email: {bolongzheng, liangguiweng, zhaoxi}@hust.edu.cn
[2]Alibaba Group, Hangzhou, China
Email: zengkai.zk@alibaba-inc.com
[3]Hong Kong University of Science and Technology, Hong Kong
Email: zxf@cse.ust.hk
[4]Aalborg University, Aalborg, Denmark
Email: csj@cs.aau.dk

***Abstract*—Trajectory similarity computation is a fundamental component in a variety of real-world applications, such as ridesharing, road planning, and transportation optimization. Recent advances in mobile devices have enabled an unprecedented increase in the amount of available trajectory data such that efficient query processing can no longer be supported by a single machine. As a result, means of performing distributed in-memory trajectory similarity search are called for. However, existing distributed proposals either suffer from computing resource waste or are unable to support the range of similarity measures that are being used. We propose a distributed in-memory management framework called REPOSE for processing top-k trajectory similarity queries on Spark. We develop a reference point trie (RP-Trie) index to organize trajectory data for local search. In addition, we design a novel heterogeneous global partitioning strategy to eliminate load imbalance in distributed settings. We report on extensive experiments with real-world data that offer insight into the performance of the solution, and show that the solution is capable of outperforming the state-of-the-art proposals.**

***Index Terms*—trajectory similarity, top-k query, distributed**

I. Introduction

With the widespread diffusion of GPS devices (such as smart phones), massive amounts of data describing the motion histories of moving objects, known as trajectories, are being generated and managed in order to serve a wide range of applications, e.g., travel time prediction [17], [18], taxis dispatching, and path planning [30], [36]. For example, Didi has released an open dataset that includes 750 million GPS points in Xi'an with a sampling rate of 2–4 seconds over the span of one month.

Top-k trajectory similarity search that finds k trajectories that are most similar to a query trajectory is a basic operation in offline analytics applications. In the context of massive trajectory data, it is non-trivial to enable efficient top-k trajectory similarity search. Many existing studies [3], [4], [11]–[14], [16], [22], [25], [26] focus on optimizing the query processing on a single machine. However, if the data cardinality exceeds the storage or processing capacity of a single machine, these methods do not extend directly to a distributed environment.

Instead, a distributed algorithm is called for that is able to exploit the resources of multiple machines. DFT [28] and DITA [19] are state-of-the-art distributed trajectory similarity search frameworks. They include global partitioning methods that place trajectories with similar properties in the same partition, and they use a global index to prune irrelevant partitions. Then, they merge the results of local searches on the surviving partitions. Finally, they return a top-k result. However, these methods have two shortcomings that limit their use in practice.

(1) Computing resource waste. DITA and DFT aim to guarantee load balancing by means of their global partitioning strategies. In particular, DITA places trajectories with close first and last points in the same partition. DFT places trajectory segments with close centroids in the same partition. However, only surviving partitions are employed on the distributed nodes, while compute nodes with no surviving partitions remain idle and are not utilized. This has an adverse effect on computing resource utilization.

(2) Limited support for similarity measures. We observe that DFT supports Hausdorff, Frechet [1], and DTW [31], but does not support LCSS [25], EDR [8], and ERP [7]. Further, DITA supports Frechet, DTW, EDR, and LCSS, but does not support Hausdorff and ERP. In order to support diverse application scenarios, it is important to be able to accommodate a wide range of similarity measures in a single system.

We propose an efficient distributed in-memory management framework, REPOSE, for top-k trajectory similarity query processing that supports a wide range of similarity measures. To eliminate poor computing resource utilization, REPOSE includes a novel heterogeneous global partition strategy that places similar trajectories in different partitions. Therefore, partitions have similar composition structure with the effect that most partitions and compute nodes are likely to contribute to a query result, which improves the computing resource utilization significantly and enables a load balancing.

In addition, we propose a reference point trie (RP-Trie) index to organize the trajectories in each partition. During index construction, we convert each trajectory into a refer-

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00067

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SUMMARY OF NOTATIONS

Notation	Definition
$\mathcal{D}$	Trajectory dataset
τ	A trajectory
τ^*	A reference trajectory
$\mathsf{D}_\mathsf{H}(\tau_1, \tau_2)$	Hausdorff distance between τ_1 and τ_2
$\mathcal{A}$	A square region that encloses all trajectories
g	The grid
δ	The grid side length
LB_o	One-side lower bound
LB_t	Two-side lower bound
LB_p	Pivot based lower bound
N_p	The number of the pivot trajectories

ence trajectory by adopting Z-order [9] that preserves sketch information. Then, we build an RP-Trie based on this representation, which reduces the space consumption. During query processing, we traverse the local RP-Trie in a best-first manner. We first develop a one-side lower bound on internal nodes for pruning. Moreover, we devise a two-side lower bound on leaf nodes for further improvement. Finally, we design a pivot-based pruning strategy for metrics.

In summary, we make the following contributions:

- We propose a distributed in-memory framework, REPOSE, for the processing of top-k trajectory similarity queries on Spark. The framework supports multiple trajectory similarity measures, including Hausdorff, Frechet, DTW, LCSS, EDR, and ERP distances.
- We discretize trajectories into reference trajectories and develop a reference point trie to organize the reference trajectories. Several optimization techniques are proposed to accelerate query processing.
- We design a novel heterogeneous global partitioning strategy to achieve load balancing and to accelerate query processing by balancing the composition of partitions.
- We compare the performance of REPOSE with the state-of-the-art distributed trajectory similarity search frameworks using real datasets. The experimental results indicate that REPOSE is able to outperform the existing frameworks.

The rest of the paper is organized as follows. Section II states the problem addressed. Section III discusses how to discretize trajectories into reference trajectories and build RP-Trie. Section IV introduces the query processing and several optimization techniques. Section V introduces the global partition strategy. Section VI describes how to extend our algorithm to other similarity measures. Section VII presents the results of our experimental study. Section VIII reviews related work. Finally, Section IX concludes the paper.

II. PROBLEM DEFINITION

We proceed to present the problem definition. Frequently used notation is summarized in Table I.

TABLE II
POINT COORDINATES OF TRAJECTORIES

Trajectory	Point Coordinates
τ_1	$(0.5, 7.5), (2.5, 7.5), (6.5, 7.5), (6.5, 4.5)$
τ_2	$(1.5, 0.5), (2.5, 0.5), (2.5, 4.5), (4.5, 4.5)$
τ_3	$(4.5, 0.5), (7.5, 0.5), (7.5, 2.5), (4.5, 2.5), (4.5, 1.5)$
τ_4	$(0.5, 7.5), (2.5, 7.5), (5.5, 7.5), (5.5, 3.5)$
τ_5	$(1.5, 0.5), (2.5, 0.5), (2.5, 5.5), (0.5, 5.5), (0.5, 2.5)$
τ_q	$(0.5, 6.5), (2.5, 6.5), (4.5, 6.5)$

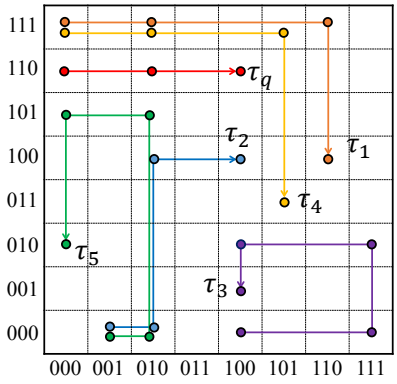


Fig. 1. A Running Example

Definition 1 (Trajectory)**.** *A trajectory τ is a finite, time-ordered sequence $\tau = \langle p_1, p_2, \ldots, p_n \rangle$, where each $p_i \in \tau$ is a sample point with a longitude and a latitude.*

A range of distance functions have been used for quantifying the similarity between trajectories, including the Hausdorff, Frechet, DTW, LCSS, EDR and ERP distances. To ease the presentation, we initially focus on the Hausdorff distance, and extend the coverage to include other distances later.

Definition 2 (Trajectory Distance)**.** *Given two trajectories $\tau_1 = \langle q_1, q_2, \ldots, q_m \rangle$ and $\tau_2 = \langle p_1, p_2, \ldots, p_n \rangle$, the Hausdorff distance between τ_1 and τ_2 is computed as follows.*

$$\mathsf{D}_\mathsf{H}(\tau_1, \tau_2) = \max\{\max_{q_i \in \tau_1} \min_{p_j \in \tau_2} d(q_i, p_j), \max_{p_j \in \tau_2} \min_{q_i \in \tau_1} d(q_i, p_j)\}, \quad (1)$$

where $d(p_i, q_j)$ is the Euclidean distance.

Definition 3 (Trajectory Similarity Search)**.** *Given a set of trajectories $\mathcal{D} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, a query trajectory τ_q, a distance function $\mathsf{Dist}(\cdot)$, and an integer k, the top-k trajectory similarity search problem reports a set R of k trajectories, where $\forall \tau \in R$ and $\forall \tau' \in \mathcal{D} - R$, we have $\mathsf{Dist}(\tau_q, \tau) < \mathsf{Dist}(\tau_q, \tau')$.*

Example 1. *Given a query trajectory τ_q and the dataset $\mathcal{D} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ in Table II, we process a top-2 query. Computing the Hausdorff distance between τ_q and all trajectories in $\mathcal{D}$, we get $\mathsf{D}_\mathsf{H}(\tau_q, \tau_1) = 2.83$, $\mathsf{D}_\mathsf{H}(\tau_q, \tau_2) = 6.08$, $\mathsf{D}_\mathsf{H}(\tau_q, \tau_3) = 6.71$, $\mathsf{D}_\mathsf{H}(\tau_q, \tau_4) = 3.16$, $\mathsf{D}_\mathsf{H}(\tau_q, \tau_5) = 6.08$. Therefore, the top-2 result is $\{\tau_1, \tau_4\}$.*

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

III. Reference Point Trie

Before describing the distributed framework, we proceed to introduce a reference point trie (RP-Trie) index for local search. First, we explain how to convert trajectories into reference trajectories. Second, we show how to build an RP-Trie on the reference trajectories. Finally, we propose an order-independent optimization to improve the performance.

A. Discretizing Trajectories with Z-order

Inspired by signature-based trajectory representation [23], we adopt the Z-order [9] to map trajectories from their native two-dimensional space to a one-dimensional space. Let $\mathcal{A}$ be a square region with side length U that encloses all trajectories. We partition $\mathcal{A}$ by means of a regular $l \times l$ grid with side length δ, where $l = U/\delta$ is a power of 2. Each cell g has a unique z-value and a reference point (the center point of g).

Example 2. *In Fig. 1, we show an 8×8 grid. The z-value of a cell is the alternate combination of its horizontal and vertical coordinates. For instance, the cell whose (binary) horizontal and vertical coordinates are 010 and 101, respectively, has z-value 011001.*

Definition 4 (Reference Trajectory)**.** *We convert a trajectory $\tau = \langle p_1, p_2, \ldots, p_n \rangle$ into a reference trajectory $\tau^* = \langle p_1^*, p_2^*, \ldots, p_n^* \rangle$, where p_i^* is the reference point of the cell g_i that p_i belongs to. Note that τ^* corresponds to a sequence of of z-values $Z = \langle z_1, z_2, \ldots, z_n \rangle$, where z_i is the z-value of g_i.*

Note that the grid granularity affects the fidelity of a reference trajectory. A small δ ensures a high fidelity.

B. Building an RP-Trie Index

We proceed to cover how to build an RP-Trie on a set of reference trajectories, which is similar to building a classical trie index. We thus use the z-value as the value of the trie node and insert all reference trajectories into the trie. The difference is that if one reference trajectory is a prefix of another reference trajectory, we append the character \$ at the end, which guarantees that every reference trajectory ends at a leaf node.

The structure of an RP-Trie is shown in Fig. 2. In internal nodes, the Label attribute is the z-value of the node that corresponds to the coordinates of the reference point, and Ptr is a pointer array pointing to child nodes. In leaf nodes, the Label attribute has the same meaning as internal nodes. Since each reference trajectory ends at a leaf node and represents multiple trajectories, we use Tid to record the trajectory ids.

Pivot trajectory. For similarity metrics such as Hausdorff, Frechet, and ERP, the triangle inequality can be used for pruning. To enable this, we use pivot trajectories to estimate the lower bound distance between a query trajectory and all trajectories in a subtree rooted at a node. Specifically, we select N_p trajectories as global pivot trajectories. For each node, let $\mathcal{T}_{sub}$ be the set of reference trajectories covered by it. We then keep an N_p-dimensional array HR, where HR$[i]$ stores a tuple (min, max) that represents the minimum and maximum

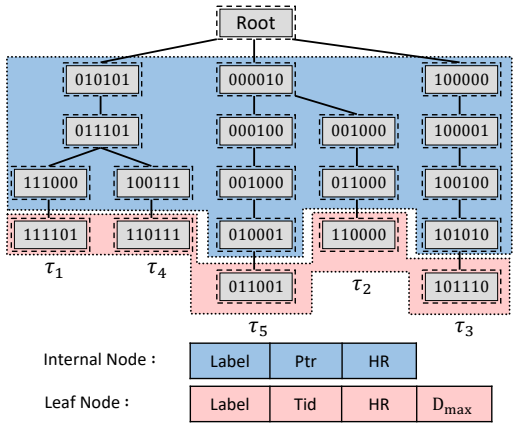


Fig. 2. The Structure of RP-Trie

distances between all reference trajectories in $\mathcal{T}_{sub}$ and the i-th pivot trajectory.

In particular, the selection of pivot trajectories has a substantial effect on the pruning performance. Thus, existing proposals consider how to find optimal pivots [5], [20]. In general, we aim to obtain a pivot set such that any two pivot trajectories in it are as distant as possible. With this in mind, we adopt a practical but effective method [21]. We uniformly and randomly sample m groups of N_p trajectories. In each group, we compute the distances of any two trajectories, and let the sum of all distances be the score of the group. Finally, we choose the N_p trajectories in the group with the largest score as the set of pivot trajectories.

In addition, we store a value D_{max} for each leaf node that is the maximum distance between the node's reference trajectory τ^* and the trajectories in the leaf node.

Cost analysis. Assume that we have N reference trajectories with maximum length L. As the RP-Trie has at most $N \cdot L$ nodes, the space cost is $O(N \cdot L \cdot N_p)$. To construct an RP-Trie, the cost of the computation of the distances between pivots and all trajectories dominates all other costs and takes $O(N \cdot L^2 \cdot N_p)$. Since both L and N_p are small ($N_p = 5$ is used in experiments), both the space and time costs are affordable.

Succinct trie structure. We observe that the upper levels of an RP-Trie consist of few nodes that are accessed frequently, while the lower levels comprise the majority of nodes that are accessed relatively infrequently due to pruning. Inspired by SuRF [37], we introduce a fast search structure for the upper levels of an RP-Trie and a space-efficient structure for the lower levels. Specifically, we optimize the RP-Trie by switching between bitmaps and byte arrays at different layers. For each upper level node, we use two bitmaps B_c and B_l with the same size as the number of grid cells, to separately record the value of a child node and the state of the child node. If a cell is the node's child node, we set the corresponding bit in B_c to 1. If a child node is not a leaf node, we set the corresponding bit in B_l to 1. Then B_c and B_l of all nodes are concatenated in breadth-first order separately such that we

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

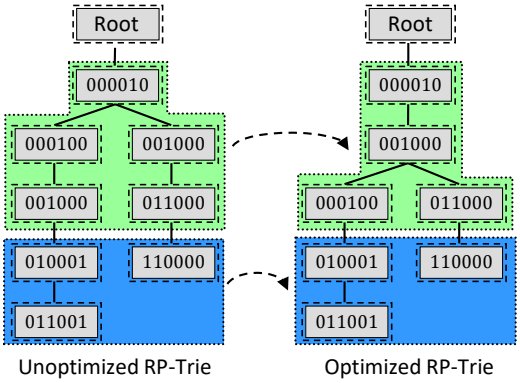


Fig. 3. Optimized Reference Point Trie

can quickly access any upper level node. For each lower level node, we serialize the structure with byte sequences. Given that the trie becomes sparse in these levels, it is space-efficient to use byte sequences.

C. Optimization by Z-value Re-arrangement

Given τ_q and a reference trajectory τ^*, if we interchange the positions of any two points in τ^* to generate a new reference trajectory $\tau^{*'}$, we have $\mathsf{D_H}(\tau_q, \tau^*) = \mathsf{D_H}(\tau_q, \tau^{*'})$. Therefore, we know that, unlike other distance measures, Hausdorff is order independent. Hence, we propose an optimization that reduces the size of an RP-Trie by rearranging the reference trajectories that achieve longer common prefixes. Specifically, we simplify τ^* in two steps: (1) z-value deduplication: We keep only one point when two or more points in τ^* have identical z-value. (2) z-value re-ordering: We re-order the points in τ^*. For example, assume that an RP-Trie is constructed from trajectories τ_2 and τ_5 as shown to the left in Fig. 3. If the z-values 000100 and 001000 of τ_5 are swapped, the total number of nodes decreases, as shown to the right in Fig. 3.

Thus, we aim to build an optimized RP-Trie with fewer nodes by means of z-value re-arrangement. Let Z be the set of z-values of a reference trajectory. Let $\mathcal{Z} = \{Z_1, Z_2, \ldots, Z_N\}$ be the collection of z-values sets of all trajectories. In order to find the optimized RP-Trie structure, we want each level of the trie to have the minimum number of nodes. In other words, we need to find the smallest cell set $G_{opt} = \{g_1, g_2, \ldots, g_r\}$ such that for any Z_i in $\mathcal{Z}$, Z_i contains at least one cell in G_{opt}. This way, we regard $g_1, g_2, \ldots, g_r$ as child nodes of the root and partition $\mathcal{Z}$ into classes $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_r$. Using this method to recursively partition each class $\mathcal{C}_i$, we obtain an optimized RP-Trie.

However, it is difficult to find the set G_{opt} for $\mathcal{Z}$. Before explaining how to solve this problem, we introduce a well known NP-hard problem: the hitting set problem.

Definition 5 ($HS(\mathcal{Z}, b)$)**.** *Given a collection of sets $\mathcal{Z} = \{Z_1, Z_2, \ldots, Z_N\}$ and a budget b, the hitting set $HS(\mathcal{Z}, b)$ returns a set G that satisfies for any Z_i in $\mathcal{Z}$,*

$$|G| \leq b \wedge G \cap Z_i \neq \emptyset$$

If no such set G exists, $HS(\mathcal{Z}, b)$ returns an empty set.

Set G is called a hitting set of $\mathcal{Z}$. Usually, a collection of sets $\mathcal{Z}$ has more than one hitting set. It is easy to find a hitting set for it when disregarding the budget b. However, it is costly to obtain a hitting set within a budget b.

Theorem 1. *Finding an optimized RP-Trie is NP-hard.*

Proof. To find an optimized RP-Trie for our reference trajectory collection $\mathcal{Z}$, we find the smallest cell set G_{opt} with size $|G_{opt}| = r$ such that for any Z_i in $\mathcal{Z}$, Z_i contains at least a cell in G_{opt}. Obviously, G_{opt} is the hitting set of $\mathcal{Z}$ with the minimum size. So, the result of $HS(\mathcal{Z}, r)$ is G_{opt}, and the result of $HS(\mathcal{Z}, r-1)$ is an empty set. Therefore, finding G_{opt} is reduced to solving $HS(\mathcal{Z}, r)$ and $HS(\mathcal{Z}, r-1)$. This completes the proof. □

We employ a greedy algorithm to solve this problem. We make the most frequent z-value z_1 in $\mathcal{Z}$ the first child node of the root. Then we put all Z_i containing z_1 into the subtree of z_1 and remove them from $\mathcal{Z}$. Next, we make the current most frequent z-value z_2 in $\mathcal{Z}$ the second child node and place all Z_i containing z_2 in its subtree and remove them from $\mathcal{Z}$. Repeating the above process until $\mathcal{Z}$ is empty, we obtain the division structure of the current level of the trie. We recursively repeat the process for the remaining levels to build an optimized RP-Trie. Assume that we have N reference trajectories and M cells. The computation cost of the greedy algorithm is $O(N \cdot M^2)$, where M is usually a small constant.

IV. Query Processing

We proceed to present the top-k query processing as well as several query optimization techniques. For ease of presentation, we mainly discuss the details of the Hausdorff distance. Extensions to support other distance measures, such as Frechet and DTW, can be found in Section VI. First, we introduce our basic algorithm to perform a top-k query, where the nodes in the RP-Trie are traversed in a best-first manner. We develop a pruning condition called one-side lower bound to filter out unqualified internal nodes. We provide three optimization techniques. The first technique reduces the computing cost on a node by using intermediate computation results. The second technique provides a tight pruning condition called two-side lower bound for pruning on leaf nodes. The third technique utilizes the metric property for pivot based pruning.

A. Search Procedure

To answer a top-k query, the nodes in the RP-Trie are traversed in ascending order of a lower bound, which is called one-side lower bound LB_o used to prune unqualified internal nodes. Let d_k be the k-th smallest found result. An internal node is pruned when its LB_o is greater than d_k.

We briefly introduce the procedure as follows:

1) We construct a priority queue E to keep the nodes based on the ascending order of their lower bounds. We initially insert the root node into E. The $minHeap$ stores the trajectories with distance to τ_q no larger than d_k.

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

2) The head element t is popped from E. If the lower bound of t is smaller than d_k, we go to Step 3). Otherwise, we stop the procedure and return $minHeap$.
3) If node t is a leaf node, we compute the distances between τ_q and the trajectories recorded by Tid. Then, we update d_k and $minHeap$; Otherwise, we compute the LB_o for each child node of t and insert it into E.
4) Repeating Steps 2) and 3) when E is not empty. Otherwise, we stop the procedure and return $minHeap$.

We introduce how to compute the one-side lower bound LB_o. For simplicity, for a sub reference trajectory that starts from the root node and ends at an internal node, we also call it a reference trajectory. Next, we give a formal definition of the one-side lower bound.

Definition 6 (One-Side Lower Bound)**.** *Given the query trajectory $\tau_q = \langle q_1, q_2, \ldots, q_m\rangle$ and a node whose reference trajectory is $\tau^* = \langle p_1^*, p_2^*, \ldots, p_n^*\rangle$, one-side lower bound of the node is defined as follows,*

$$LB_o(\tau_q, \tau^*) = \max\{\max_{p_j^* \in \tau^*} \min_{q_i \in \tau_q} d(q_i, p_j^*) - \frac{\sqrt{2}\delta}{2}, 0\} \quad (2)$$

Two following lemmas ensure that the returned results by above procedure are the correct top-k query results of τ_q.

Lemma 1. *The LB_o of a leaf node is smaller than the minimum distance between τ_q and any trajectory stored in the node.*

Proof. Given $\tau_q = \langle q_1, q_2, \ldots, q_m\rangle$, a node's reference trajectory $\tau^* = \langle p_1^*, p_2^*, \ldots, p_n^*\rangle$ and any trajectory $\tau = \langle p_1, p_2, \ldots, p_{n'}\rangle$ stored in the node, we prove that $\mathsf{D_H}(\tau_q, \tau) \geq LB_o(\tau_q, \tau^*)$. Assume p_i falls into the cell represented by p_k^* (we call it $p_i \in p_k^*$), we have $d(q_j, p_i) \geq \max\{d(q_j, p_k^*) - \sqrt{2}\delta/2, 0\}$ according to the triangle inequality. Then we have

$$\begin{aligned}
\mathsf{D_H}(\tau_q, \tau) =& \max\{\max_{q_j \in \tau_q} \min_{p_i \in \tau} d(p_i, q_j), \max_{p_i \in \tau} \min_{q_j \in \tau_q} d(p_i, q_j)\} \\
\geq& \max_{p_i \in \tau} \min_{q_j \in \tau_q} d(p_i, q_j) \\
\geq& \max_{p_i \in \tau, p_i \in p_k^*} \min_{q_j \in \tau_q} \max\{d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\} \\
=& \max_{p_k^* \in \tau^*} \min_{q_j \in \tau_q} \max\{d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\} \\
=& \max\{\max_{p_k^* \in \tau^*} \min_{q_j \in \tau_q} d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\} \\
=& LB_o(\tau_q, \tau^*)
\end{aligned}$$

□

Lemma 1 indicates that if the node is a leaf node, LB_o is the lower bound distance between τ_q and the trajectories recorded by Tid. Thus, the trajectories recorded by the node can be safely pruned when $LB_o \geq d_k$. However, if the node is not a leaf node, Lemma 1 does not work, so it comes to the following Lemma 2. Lemma 2 shows that LB_o of a node is larger than that of its parent node. If LB_o is larger than d_k, no top-k results exist in the subtree of current node, and we can safely prune it.

Lemma 2. *For an internal node, the LB_o between its child node and τ_q is greater than that between the node and τ_q.*

Proof. Let $\tau_p^* = \langle p_1^*, \cdots, p_{n-1}^*\rangle$ and $\tau^* = \langle p_1^*, \cdots, p_n^*\rangle$ be the reference trajectories of a node and its child node, respectively. Given τ_q, we prove that $LB_o(\tau_q, \tau^*) \geq LB_o(\tau_q, \tau_p^*)$.

$$\begin{aligned}
LB_o(\tau_q, \tau^*) =& \max_{p_k^* \in \tau^*} \min_{q_j \in \tau_q} \max\{d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\} \\
=& \max\{\max_{p_k^* \in \tau_p^*} \min_{q_j \in \tau_q} \max\{d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\}, \\
& \min_{q_j \in \tau_q} \max\{d(q_j, p_n^*) - \frac{\sqrt{2}\delta}{2}, 0\}\} \\
\geq& \max_{p_k^* \in \tau_p^*} \min_{q_j \in \tau_q} \max\{d(q_j, p_k^*) - \frac{\sqrt{2}\delta}{2}, 0\} \\
=& LB_o(\tau_q, \tau_p^*)
\end{aligned}$$

□

Note that Lemmas 1 and 2 also hold for Frechet and DTW with a modification regarding the computation of LB_o. The details can be found in Section VI.

B. Pruning using Two-Side Lower Bound

We know that one-side lower bound is designed for pruning the subtree of internal nodes. For each internal node, we only have the prefix reference trajectories from the root node to this internal node. In contrast, for a leaf node, we have complete reference trajectories of the trajectories stored in this node. Therefore, a tight lower bound, called two-side lower bound, is provided to further improve the query efficiency. Based on the reference trajectory, the two-side lower bound is computed as follows:

Definition 7 (Two-Side Lower Bound)**.** *Given τ_q and a leaf node whose reference trajectory is τ^*, the two-side lower bound LB_t is computed as follows,*

$$LB_t(\tau_q, \tau^*) = \max\{\mathsf{D_H}(\tau_q, \tau^*) - \mathsf{D}_{max}, 0\} \quad (3)$$

where D_{max} is the maximum distance from all trajectories in the current node to the reference trajectory.

According to the triangle inequality, it is easy to prove that LB_t is the lower bound distance of all the trajectories stored in current node. In other word, LB_t is smaller than the distance between τ_q and any trajectories stored in the leaf node. The focuses of LB_o and LB_t are different. LB_o is able to prune all nodes of a subtree, but its value is loose compared with LB_t. While LB_t is designed for pruning trajectories in a leaf node, its value is close to exact distance. Next, we introduce how to reduce the computation overheads of both LB_o and LB_t by using the intermediate computation results.

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: CompLB(τ_q, p^*, r, c_{max})

Input: The query τ_q, the newly added reference point p^*, an array r with size $|\tau_q|$ and c_{max}
Output: LB_o, LB_t, the updated r and c_{max}

$i \leftarrow 1$; $r_{max} \leftarrow 0$; $c \leftarrow +\infty$;
while $i \leq |\tau_q|$ **do**
  $dist = d(\tau_q[i], p^*)$;
  $r[i] = \min\{dist, r[i]\}$;
  $c = \min\{dist, c\}$;
  $r_{max} = \max\{r[i], r_{max}\}$;
  $i \leftarrow i + 1$
$c_{max} \leftarrow \max\{c_{max}, c\}$;
$LB_o = \max\{c_{max} - \frac{\sqrt{2}\delta}{2}, 0\}$;
$LB_t = \max\{\max\{r_{max}, c_{max}\} - \mathsf{D}_{max}, 0\}$;
return LB_o, LB_t, r, c_{max};

C. Optimization using Intermediate Results

To access each node on the RP-Trie, the computational overheads of LB_o and LB_t are both $O(mn)$, where m and n are the lengths of the query trajectory and the reference trajectory, respectively. We propose an optimization that uses the intermediate computation results of the parent node to reduce the overhead to $O(m)$.

Before we describe the optimization, we first review the steps of computing Hausdorff. We assume that the lengths of τ_q and the reference trajectory τ^* are both 2. As shown in Fig. 4, q_1 and q_2 are points in the query trajectory, while p_1^* and p_2^* are points in the reference trajectory. Let r_i be the minimum value in row i, e.g., the minimum of $d(q_i, p_1^*)$ and $d(q_i, p_2^*)$. Let c_j be the minimum value in column j, e.g., the minimum of $d(q_1, p_j^*)$ and $d(q_2, p_j^*)$. Let c_{max} be the maximum value among all c_j. According to the definition of Hausdorff,

$$LB_o(\tau_q, \tau^*) = \max\{c_{max} - \frac{\sqrt{2}\delta}{2}, 0\},$$

and

$$LB_t(\tau_q, \tau^*) = \max\{\max\{\max\{r_i\}, c_{max}\} - \mathsf{D}_{max}, 0\}.$$

After accessing p_2^*, we record r_1, r_2 and c_{max} as intermediate results. For a new reference trajectory $\tau_{new}^* = \langle p_1^*, p_2^*, p_3^* \rangle$, both $LB_t(\tau_q, \tau_{new}^*)$ and $LB_o(\tau_q, \tau_{new}^*)$ can be quickly computed by $d(q_1, p_3^*)$, $d(q_2, p_3^*)$ and intermediate results, since we only have to update the values of c_{max} and all r_i, where

$$\begin{aligned} r_1' &= \min\{r_1, d(q_1, p_3^*)\} \\ r_2' &= \min\{r_2, d(q_2, p_3^*)\} \\ c_{max}' &= \max\{c_{max}, \min\{d(q_1, p_3^*), d(q_2, p_3^*)\}\}. \end{aligned} \tag{4}$$

Similarly, r_1', r_2' and c_{max}' are the intermediate results after accessing p_3^* that can be used for the subsequent computation.

Therefore, the computational overheads of LB_o and LB_t for each node are $O(m)$. Algorithm 1 describes the process that we compute LB_o and LB_t in detail.

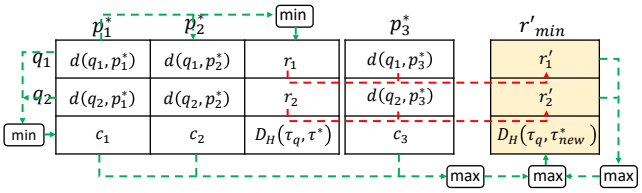


Fig. 4. Distance Matrix

D. Pivot based Pruning

As the Hausdorff distance is a metric, the pivot-based pruning strategy [6] is used to further improve the query efficiency. Therefore, we propose a pruning strategy by using a pivot based lower bound LB_p.

During preprocessing, we select N_p trajectories as pivot trajectories. For each node in RP-Trie, we store a distance array HR. During the query processing, we first compute the distances d_{qp} between τ_q and all pivot trajectories. The time complexity is $O(N_p \cdot m \cdot n)$, where m and n are the lengths of τ_q and pivot trajectory respectively.

We compute the pivot based lower bound LB_p as follows,

$$LB_p = \max\{|d_{qp}[i] - \mathsf{HR}[i].max - \frac{\sqrt{2}}{2}\delta|\}. \tag{5}$$

Since N_p is small, the time cost of computing LB_p is small.

V. Distributed Framework

We proceed to cover the distributed framework. First, we discuss the drawbacks of existing global partitioning strategies. Then, we introduce our partitioning strategy that aims to ensure better computing resource utilization during query processing. Finally, we discuss how to build a local RP-Trie based on Spark RDDs.

A. Drawbacks of Existing Global Partitioning Strategies

A straightforward partitioning method is to randomly divide trajectories into different partitions such that similar numbers of trajectories are kept in each partition. However, this does not guarantee load balancing because the query times in different partitions can be quite different. Existing proposals, such as DITA, DFT, and DISAX [29], employ a homogeneous partitioning method that groups similar trajectories into the same partition. Then, they use a global index to prune irrelevant partitions without trajectories that are similar to the query trajectory. However, these methods have limited performance for two main reasons:

- **Computing resource waste**. After global pruning, only compute nodes with surviving partitions compute local results, while other nodes are idle and do not contribute.
- **Less effective local pruning**. If the trajectories in each partition are similar, the pruning in local search is less effective since the MBRs of the local index may have large overlaps.

The homogeneous partitioning strategy targets batch search (e.g., high concurrency), where different partitions response

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

to different query trajectories. However, not all partitions of trajectories are involved in the batch query processing if the queries are skewed. For example, ride-hailing companies tend to issue a batch of analysis queries in hot regions to increase profits. Therefore, some computing resource waste cannot be avoided.

B. Our Global Partitioning Strategy

An effective global partitioning strategy should ensure load balancing among partitions and should accelerate local-search part of query processing. Unlike existing strategies, we propose a heterogeneous partitioning strategy that places similar trajectories in different partitions based on the following observations:

- As mentioned, the pruning in local search is less effective in existing methods. Inspired by the principle of maximum entropy, a high degree of data discrimination in a partition is likely to enable better pruning.
- The composition structure of each partition is similar, since each partition contains trajectories from different groups of similar trajectories. It is possible that most partitions contain query results. Therefore, the query times of most partitions are similar, which enables load-balancing.

Specifically, we use the simple but effective clustering algorithm SOM-TC [10] to partition trajectories. We encode each trajectory τ as a reference trajectory τ^* using geohash. If $\tau_1^* = \tau_2^*$, we group τ_1 and τ_2 into a cluster. Note that we vary the space granularity of geohash to control the number of clusters. At first, we set the space granularity to a small value such that a cluster contains only one trajectory. Then we enlarge the space granularity gradually and group trajectories into larger clusters. This process stops when we reach about N/N_G clusters, where N is the dataset cardinality and N_G is the number of partitions. We sort the trajectories based on their cluster id and trajectory id. Finally, we assign the trajectories from the sorted sequence to partitions in round-robin fashion.

C. Building Local Index based on RDD

Spark Core [34] is based on the concept of abstract Resilient Distributed Dataset (RDD), which supports two types of operations: transformation and action operations. However, RDDs are designed for sequential processing, and random accesses through RDDs are expensive since such accesses may simply turn into full scans over the data. In our method, random access is inevitable. We solve this challenge by exploiting an existing method [28], that is, we package the data and index into a new class structure and use the class as the type of RDD.

Specially, the abstract class Partitioner is provided by Spark for data partitioning, and Spark allows users to define their own partitioning strategies through inheritance from the abstract class. We use Partitioner to implement the strategy presented in Section V-B. Then, we package all trajectories within an RDD partition and RP-Trie index into a class structure called RpTraj that is defined as follow.

```
case class RpTraj (trajectory: Array, Index: RP-Trie)
```

We change the type of RDD to RpTraj.

```
type RpTrieRDD = RDD [RpTraj]
```

The RpTrieRDD is an RDD structure. The transformation operation mapPartitions is provided by Spark, and we use RpTrieRDD.mapPartitions to manipulate each partition. In particular, in each partition, we first use Index.build to construct RP-Trie index. Then we use Index.query to execute the top-k query. Finally, the master collects the results from each partition by collect and determines the global top-k result.

VI. Other Similarity measures

We proceed to extend our algorithm to Frechet and DTW. Due to the different properties of similarity measures, the index structures are distinguished with that for Hausdorff.

A. Extension on Frechet

The only difference with Hausdorff is that Frechet is sensitive to the order of points. So the RP-Trie is used without the trie optimization. As Frechet is also a metric, we still use pivot trajectories to accelerate the query processing. Given $\tau_q = \{q_1, q_2, \cdots, q_m\}$ and $\tau = \{p_1, p_2, \cdots, p_n\}$, the Frechet distance between τ_q and τ is computed as follows,

$$\mathsf{D_F}(\tau_q, \tau) = \begin{cases} \max_{j=1}^{n} d(q_1, p_j) & m = 1 \\ \max_{i=1}^{m} d(q_i, p_1) & n = 1 \\ \max\{d(q_m, p_n), \min\{\mathsf{D_F}(\tau_q^{m-1}, \tau^{n-1}), \\ \mathsf{D_F}(\tau_q^{m-1}, \tau), \mathsf{D_F}(\tau_q, \tau^{n-1})\}\} & \text{otherwise} \end{cases} \tag{6}$$

where τ^{m-1} is the prefix of τ by removing the last point.

We modify the computation of LB_o^{F} and LB_t^{F} of Frechet,

$$LB_o^{\mathsf{F}}(\tau_q, \tau^*) = \max\{c_{min} - \frac{\sqrt{2}\delta}{2}, 0\} \tag{7}$$

$$LB_t^{\mathsf{F}}(\tau_q, \tau^*) = \max\{f_{m,n} - \frac{\sqrt{2}\delta}{2}, 0\} \tag{8}$$

where $f_{i,j} = \mathsf{D_F}(\tau_q^i, \tau^{*j})$ is the element of the i-th row and the j-th column of the distance matrix, and c_{min} is the minimum value of the newly added column, e.g., $c_{min} = \min\{f_{1,4}, f_{2,4}, f_{3,4}\}$, as shown in Fig. 5. Fortunately, we are still able to quickly compute LB_o^{F} and LB_t^{F} through the intermediate results. Given the current last column values $\{f_{1,3}, f_{2,3}, f_{3,3}\}$, the values in the new column are computed as follows,

$$f_{i,4} = \max\{d(q_i, p_4^*), \min\{f_{i-1,3}, f_{i-1,4}, f_{i,3}\}\}. \tag{9}$$

In addition, as Frechet is a metric, we can still use the pivot-based lower bound LB_p^{F} for pruning.

Lemma 3. *LB_o^{F}, LB_t^{F} and LB_p^{F} have the same properties as those for Hausdorff.*

1) The LB_o^{F}, LB_t^{F} and LB_p^{F} of a leaf node are all smaller than the minimum Frechet distance between τ_q and any trajectory stored in the node.

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

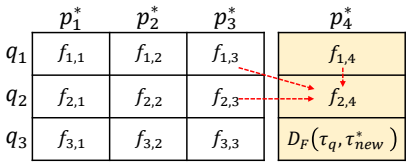


Fig. 5. Distance Matrix of Frechet

2) For an internal node, the LB_o^F between its child node and τ_q is larger than that between the node and τ_q.

The first property indicates that LB_o^{F}, LB_t^{F} and LB_p^{F} are correct lower bounds. The second property allows us to prune the sub-tree of an internal node with LB_o^{F}.

Proof. For property 1, since $LB_t^{\mathsf{F}}(\tau_q, \tau^*) \geq LB_o^{\mathsf{F}}(\tau_q, \tau^*)$, we only need to prove that LB_t^{F} and LB_p^{F} are correct lower bounds. According to Eq. 8 and $f_{m,n} = \mathsf{D_F}(\tau_q, \tau^*)$, we have

$$LB_t^{\mathsf{F}}(\tau_q, \tau^*) = \max\{\mathsf{D_F}(\tau_q, \tau^*) - \frac{\sqrt{2}\delta}{2}, 0\}.$$

For a trajectory τ with a reference trajectory τ^*, we have $\mathsf{D_F}(\tau, \tau^*) \leq \frac{\sqrt{2}\delta}{2}$. Therefore, due to the triangle inequality, we have $LB_t^{\mathsf{F}}(\tau_q, \tau^*) \leq \mathsf{D_F}(\tau_q, \tau)$. Similarly, it is easy to have that $LB_p^{\mathsf{F}}(\tau_q, \tau^*) \leq \mathsf{D_F}(\tau_q, \tau)$.

For property 2, we prove that $LB_o^{\mathsf{F}}(\tau_q, \tau^{*j}) \geq LB_o^{\mathsf{F}}(\tau_q, \tau^{*j-1})$, $2 \leq j \leq n$. According to Eq. 8,

$$\begin{aligned} LB_o^{\mathsf{F}}(\tau_q, \tau^{*j}) &= \max\{\min f_{i,j} - \frac{\sqrt{2}\delta}{2}, 0\} \\ LB_o^{\mathsf{F}}(\tau_q, \tau^{*j-1}) &= \max\{\min f_{i,j-1} - \frac{\sqrt{2}\delta}{2}, 0\} \end{aligned} \tag{10}$$

Assume that $h = \arg\min_i f_{i,j-1}$, we have

$$\begin{aligned} f_{1,j} &= \max\left\{d\left(q_1, p_j^*\right), f_{1,j-1}\right\} \\ &\geq f_{1,j-1} \geq f_{h,j-1} \\ f_{2,j} &= \max\left\{d\left(q_1, p_j^*\right), \min\left\{f_{1,j-1}, f_{1,j}, f_{2,j-1}\right\}\right\} \\ &\geq \min\left\{f_{1,j-1}, f_{1,j}, f_{2,j-1}\right\} \geq f_{h,j-1} \end{aligned} \tag{11}$$

Therefore, we have $f_{i,j} \geq f_{h,j-1}, 1 \leq i \leq m$, and $\min f_{i,j} \geq f_{h,j-1}$. Finally, $LB_o^{\mathsf{F}}(\tau_q, \tau^{*j}) \geq LB_o^{\mathsf{F}}(\tau_q, \tau^{*j-1})$. □

B. Extension on DTW

Since DTW is not a metric and is sensitive to the order of trajectory points, thus only the basic RP-Trie structure is used. The DTW between τ_q and τ is computed as follows,

$$\mathsf{D_{DTW}}(\tau_q, \tau) = \begin{cases} \sum_{j=1}^{n} d\left(q_1, p_j\right) & m = 1 \\ \sum_{i=1}^{m} d\left(q_i, p_1\right) & n = 1 \\ d\left(q_m, p_n\right) + \min\left\{\mathsf{D_{DTW}}\left(\tau_q^{m-1}, \tau^{n-1}\right), \right. & \\ \left. \mathsf{D_{DTW}}\left(\tau_q^{m-1}, \tau\right), \mathsf{D_{DTW}}\left(\tau_q, \tau^{n-1}\right)\right\} & \text{otherwise} \end{cases} \tag{12}$$

The computations of LB_o^{DTW} and LB_t^{DTW} are similar to those of Frechet, which are computed as follows.

$$LB_o^{\mathsf{DTW}}(\tau_q, \tau^*) = c_{min} \tag{13}$$

$$LB_t^{\mathsf{DTW}}(\tau_q, \tau^*) = f_{i,n} \tag{14}$$

where $f_{i,j} = \mathsf{D_{DTW}}\left(\tau_q^i, \tau^{*j}\right)$ is the element of the i-th row and the j-th column of the distance matrix, and c_{min} is the minimum value of the newly added column, i.e., $c_{min} = \min_i\{f_{i,j}\}$ at the j-th column. We update $f_{i,j}$ as follows.

$$f_{i,j} = d'(q_i, p_j^*) + \min\left\{f_{i-1,j-1}, f_{i-1,j}, f_{i,j-1}\right\}. \tag{15}$$

Note that $d'(q_i, p_j^*)$ is the minimum distance between q_i and the cell that p_j^* belongs to. We use it instead of $d(q_i, p_j^*)$ since the triangle inequality does not apply to DTW.

Lemma 4. *LB_o^{DTW}, LB_t^{DTW} have the same properties as those for Hausdorff.*

1) The LB_o^{DTW}, LB_t^{DTW} of a leaf node are smaller than the minimum DTW distance between τ_q and any trajectory stored in the node.

2) For an internal node, the LB_o^{DTW} between its child node and τ_q is larger than that between the node and τ_q.

The proof is similar to that of Lemma 3.

The basic RP-Trie and the heterogeneous partitioning strategy are applicable to other distance measures. For metrics, such as ERP, the RP-Trie with the trie optimization and the pivot-based pruning can be employed in a similar way as for Frechet. For non-metrics, such as LCSS and EDR, the basic RP-Trie can be used similarly as for DTW.

VII. EXPERIMENTS

A. Experimental Setup

All experiments are conducted on a cluster with 1 master node and 16 worker nodes. Each node has a 4-core Intel Xeon (Cascade Lake) Platinum 8269CY @ 2.50GHz processor and 32GB main memory and runs Ubuntu 18.04 LTS with Hadoop 2.6.0 and Spark 2.2.0.

Datasets. We conduct experiments on 3 types of datasets.

1) Small scale and small spatial span: San Francisco (SF)[1], Porto[2], Rome[3], T-drive [33].
2) Large scale and small spatial span: Chengdu and Xi'an[4].
3) Large scale and large spatial span: OSM[5].

The dataset statistics are shown in Table III. In the preprocessing stage, we remove the trajectories with length smaller than 10, and we split the trajectories with length larger than 1,000 into multiple trajectories. We uniformly and randomly select 100 trajectories as the query set.

Competing algorithms. We compare the performance of REPOSE with three baseline algorithms:

[1]http://sigspatial2017.sigspatial.org/giscup2017/home
[2]https://www.kaggle.com/c/pkdd-15-predict-taxiservice-trajectory-i
[3]http://crawdad.org/roma/taxi/20140717
[4]https://gaia.didichuxing.com
[5]https://www.openstreetmap.org

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

TABLE III
STATISTICS OF DATASETS

Datasets	Cardinality	AvgLen	Spatial span	Size (GB)
T-drive	356,228	22.6	(1.89°, 1.17°)	0.16
SF	343,696	27.5	(0.54°, 0.76°)	0.19
Rome	99,473	152.4	(1.21°, 0.86°)	0.28
Porto	1,613,284	48.9	(11.7°, 14.2°)	1.24
Xi'an	6,645,727	230.1	(0.09°, 0.08°)	26.8
Chengdu	11,327,466	188.9	(0.09°, 0.07°)	37.7
OSM	4,464,399	596.3	(360°, 180°)	50.8

1) **DFT**: A segment-based distributed top-k query algorithm that supports Hausdorff, Frechet, and DTW. Specifically, we choose the variant DFT-RB+DI that has the best query performance and the largest space overhead.
2) **DITA**: An in-memory distributed range query algorithm. In order to support top-k query, DITA first estimates a threshold and then executes a range query to find candidate trajectories. Finally, DITA finds the k most similar trajectories. DITA does not support Hausdorff.
3) **LS**: Brute-force linear search (LS) that computes the distances between the query and all trajectories in each partition and merges the results.

Parameter settings. For DFT, we set the partition pruning parameter $C = 5$. For DITA, we set $N_L = 32$ and the pivot size is set to 4. The pivot selection strategy is the neighbor distance strategy. For REPOSE, due to the different spatial spans and data densities of the datasets, we choose different values of δ. For the SF, Porto, and Roma, we set $\delta = 0.05$. For T-drive, we set $\delta = 0.15$. For OSM, we set $\delta = 1.0$. Next, for the Chengdu dataset, we set $\delta = 0.01$ for Hausdorff, and $\delta = 0.02$ for Frechet and DTW. Finally, for the Xi'an dataset, we set $\delta = 0.01$ for Hausdorff, and $\delta = 0.03$ for Frechet and DTW. We choose $N_p = 5$ pivot trajectories.

For queries, we use $k = 100$ as the default. Since we have 16 worker nodes, each of which has a 4-core processor, we set the default number of partitions to 64 (each core processes a partition).

Performance metrics. We study three performance metrics: (1) query time (QT), (2) index size (IS), and (3) index construction time (IT). We repeat each query 20 times and report the average values. The index construction time of REPOSE includes the time for converting trajectories to reference trajectories, clustering the trajectories, and building the trie.

B. Performance Evaluation

We compare the algorithms on all datasets and three similarity measures (Hausdorff, Frechet, DTW). A performance overview is shown in Table IV. Due to the space limitation, we only report the performances on the T-drive, Xi'an, and OSM datasets and for the Hausdorff and Frechet distances in the remaining experiments.

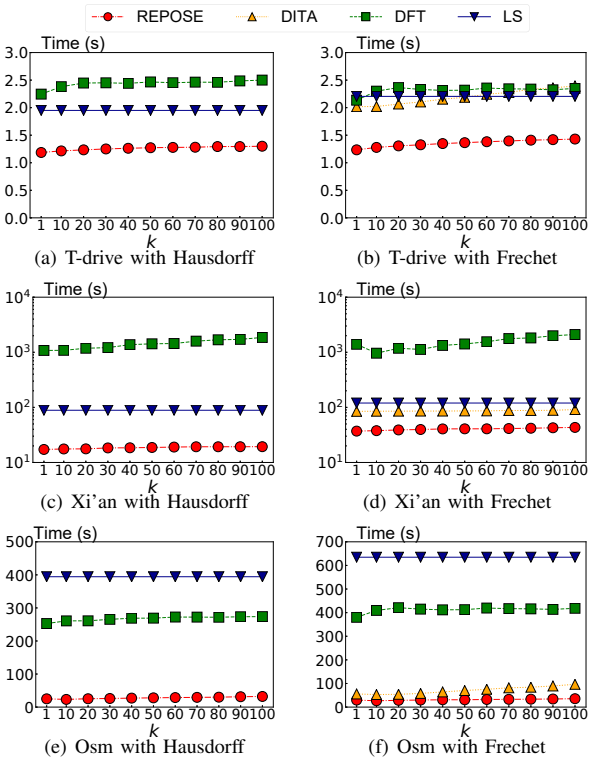
Performance overview. We make the following observations from Table IV: (1) In terms of query time and index construction time, REPOSE significantly outperforms the baseline methods. REPOSE also maintains a smaller index than DFT and DITA in most cases. REPOSE has a slightly larger index than DITA on Chengdu, Xi'an, and OSM for the Frechet and DTW distances. This occurs because DITA compresses all trajectories into fixed-length representative trajectories. Nevertheless, REPOSE can capture trajectory features adaptively and is able to improve the query efficiency. (2) The index size of DFT is about 4 times those of REPOSE and DITA on all datasets. For example on Xi'an, DFT requires 142GB, while REPOSE and DITA only require 32GB. The reason is that DFT needs to regroup line segments into trajectories when computing distances and needs a dual index that takes up extra space. (3) For the first two types of datasets, it is interesting to see that the query time of LS is smaller than those of DITA and DFT. The reason is that the datasets are small, so the pruning capabilities are not utilized fully. In addition, some index-specific operations add computational overhead. For large datasets with small spatial span, the data density is high, which renders pruning more challenging.

TABLE IV
PERFORMANCE OVERVIEW

Metric	Distance	Algorithm	SF	Porto	Rome	T-drive	Xi'an	Chengdu	OSM
QT (s)	Hausdorff	**REPOSE**	**1.51**	**1.84**	**4.51**	**1.29**	**19.14**	**30.55**	**32.16**
		DITA	/	/	/	/	/	/	/
		DFT	4.03	32.91	14.99	2.50	1857.88	2462.54	273.68
		LS	2.46	1.97	4.64	1.95	88.68	119.11	393.36
	Frechet	**REPOSE**	**1.73**	**1.88**	**6.87**	**1.28**	**43.27**	**55.39**	**35.61**
		DITA	2.67	1.92	15.30	2.39	91.31	62.87	94.12
		DFT	3.89	32.03	14.99	2.33	2100.80	2284.49	418.40
		LS	2.88	2.41	7.32	2.20	119.89	156.85	634.78
	DTW	**REPOSE**	**1.98**	**2.23**	**6.99**	**1.48**	**180.09**	**189.52**	**499.52**
		DITA	3.15	2.32	11.06	2.78	186.36	199.98	509.53
		DFT	3.88	32.05	14.96	2.36	2472.35	2300.35	525.38
		LS	2.37	2.35	7.04	1.92	202.46	225.92	608.89
IS (GB)	Hausdorff	**REPOSE**	**0.19**	**1.68**	**0.28**	**0.16**	**28.01**	**39.40**	**47.92**
		DITA	/	/	/	/	/	/	/
		DFT	1.03	6.10	1.39	0.85	142.30	213.71	235.06
		LS	/	/	/	/	/	/	/
	Frechet	**REPOSE**	**0.22**	**1.71**	**0.31**	**0.17**	32.13	56.26	60.39
		DITA	0.23	1.81	0.32	0.21	**31.43**	**44.52**	**58.60**
		DFT	1.03	6.10	1.39	0.85	142.30	213.71	235.06
		LS	/	/	/	/	/	/	/
	DTW	**REPOSE**	**0.22**	**1.71**	**0.31**	**0.17**	32.13	56.26	60.39
		DITA	0.23	1.81	0.32	0.21	**31.43**	**44.52**	**58.60**
		DFT	1.03	6.10	1.39	0.85	142.30	213.71	235.06
		LS	/	/	/	/	/	/	/
IT (s)	Hausdorff	**REPOSE**	**3.96**	**5.16**	**4.40**	**3.47**	**75.41**	**119.08**	**100.96**
		DITA	/	/	/	/	/	/	/
		DFT	17.28	75.80	20.11	15.32	1407.29	2071.53	3389.59
		LS	/	/	/	/	/	/	/
	Frechet	**REPOSE**	**5.27**	**5.19**	**5.34**	**4.30**	**97.61**	**148.63**	**148.98**
		DITA	20.24	20.42	18.21	21.54	235.02	323.38	349.31
		DFT	17.28	75.80	20.11	15.32	1407.29	2071.53	3389.59
		LS	/	/	/	/	/	/	/
	DTW	**REPOSE**	**5.32**	**5.01**	**5.52**	**4.10**	**99.71**	**152.33**	**149.08**
		DITA	17.99	21.15	17.94	18.28	240.75	326.72	363.95
		DFT	19.98	75.81	23.71	14.12	1417.10	2061.31	3379.60
		LS	/	/	/	/	/	/	/

Varying the value of k. Fig. 6 shows the performances of all algorithms when varying the value of k in $\{1, 10, 20, \ldots, 100\}$. We notice that with the increase of k, the changing trends differ across the algorithms. For example, the query time of DFT is unstable since it finds $C * k$ trajectories

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.



(a) T-drive with Hausdorff (b) T-drive with Frechet (c) Xi'an with Hausdorff (d) Xi'an with Frechet (e) Osm with Hausdorff (f) Osm with Frechet

Fig. 6. Performance when Varying k

TABLE V
QUERY TIME OF δ ON D_H (HAUSDORFF) AND D_F (FRECHET)

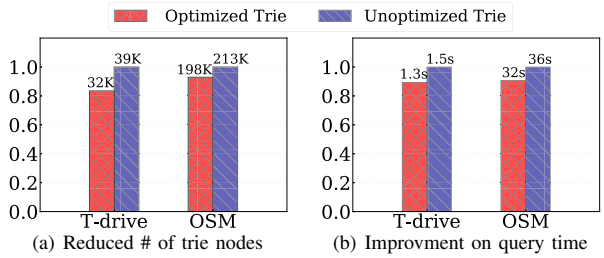
T-drive			Xi'an			OSM		
Value	QT (s)		Value	QT (s)		Value	QT (s)	
	D_H	D_F		D_H	D_F		D_H	D_F
0.01	5.04	4.39	0.005	28.52	68.60	0.1	50.58	50.99
0.05	4.06	3.19	**0.010**	**19.14**	61.31	0.5	36.70	39.52
0.10	2.47	2.55	0.015	21.88	54.11	**1.0**	**32.16**	**35.61**
0.15	**1.29**	**1.28**	0.020	23.98	48.46	1.5	35.66	37.71
0.20	2.26	1.41	0.025	29.81	51.76	2.0	37.10	38.46
0.25	2.50	1.50	**0.030**	32.80	**43.27**	2.5	39.23	41.55
0.30	3.04	2.60	0.035	35.10	56.34	3.0	43.39	45.54

at random from the dataset and uses the k-th smallest distance as the threshold. The query time depends on the quality of the threshold. The query time of DITA increases when k increases. DITA repeatedly reduces its threshold by half until the number of candidate trajectories is less than $C * k$. The k-th smallest distance is used to perform a range search. LS computes the distance between the query trajectory and all trajectories, so the value of k has little effect. REPOSE uses the current best k-th result as a pruning threshold, so the query time also increases when k increases. REPOSE achieves the best performance for all k and offers a considerable improvement.

Parameter chosen on δ. We study the effect of δ on REPOSE. With the change of δ, there are descending and ascending trends on the query time. The reason for the descending trend is that reference trajectories become longer quickly when δ decreases. Although a long reference trajectory can improve the pruning, the benefits may become small after exceeding a certain length, since time overheads for computing LB_o and LB_t are introduced. The ascending trend occurs because the number of cells decreases when δ increases. Then a reference trajectory cannot approximate a trajectory well, which results in a poor pruning and larger query times. In summary, the value of δ affects the query time significantly, and it is important to select an appropriate δ value. Table V shows that the effect of δ is relatively consistent on the same dataset. Therefore, we can reuse the setting of δ from one measure to another when the query efficiency meets the application requirements.

TABLE VI
QUERY TIME OF N_p ON D_H (HAUSDORFF) AND D_F (FRECHET)

T-drive			Xi'an			OSM		
Value	QT (s)		Value	QT (s)		Value	QT (s)	
	D_H	D_F		D_H	D_F		D_H	D_F
1	1.45	1.49	1	22.20	49.14	1	35.09	39.05
3	**1.21**	1.34	3	21.10	46.89	3	34.36	37.85
5	1.29	**1.28**	**5**	**19.14**	43.27	**5**	**32.16**	**35.61**
7	1.30	1.51	**7**	22.99	**41.41**	7	32.33	35.85
9	1.40	2.02	9	23.38	48.20	9	39.43	36.14
11	1.55	2.43	11	24.07	49.91	11	38.86	40.02



(a) Reduced # of trie nodes (b) Improvment on query time

Fig. 7. Improvement by Optimized Trie

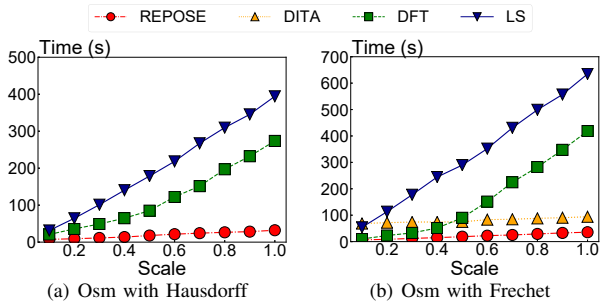
Parameter chosen on N_p. Table VI shows the effect of N_p on the algorithms. Similar to δ, there are descending and ascending trends on query time. As N_p increases beyond a boundary value, the improvement on pruning is small and additional computational overhead is introduced.

Improvement by optimized trie. Fig. 7 shows the improvements by using the optimized trie on the query time and the reduced number of trie nodes. For T-drive dataset, the number of trie nodes is reduced by about 20% compared to that of using unoptimized trie. The query time is decreased by about 12%. For OSM dataset, both of the number of trie nodes and the query time are reduced by about 8%.

Effect of dataset cardinality. From Fig. 8, we observe that the query time of all algorithms grows linearly. REPOSE has the best performance due to a better partitioning scheme and a powerful pruning.

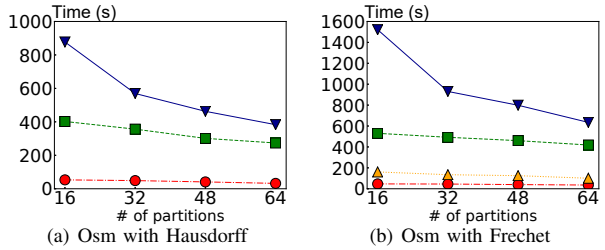
Effect of the number of partitions. Fig. 9 shows that

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.



(a) Osm with Hausdorff (b) Osm with Frechet

Fig. 8. Effect of Dataset Cardinality



(a) Osm with Hausdorff (b) Osm with Frechet

Fig. 9. Effect of the Number of Partitions

with the increase of the number of partitions from 16 to 64, all algorithms gain performance improvements. We have 64 partitions by default, where each core processes a partition. REPOSE achieves a higher performance gain (not including LS) because it implements a better partitioning scheme by equalizing the workload of each partition. LS has the highest performance gain. This is because it suffers from a severe data skew issue when the number of partitions is small. When we increase the number of partitions, the performance is significantly improved.

Effect of partitioning strategies. We deploy three different partitioning strategies, heterogeneous, homogeneous, and random, with the RP-Trie as a local index to analyze the effects of the partitioning strategies. In the homogeneous strategy, we place similar trajectories in a cluster in the same partition. In the random strategy, we place trajectories into partitions at random. The results are shown in Table VII. The heterogeneous partitioning strategy achieves the best performance, and the homogeneous strategy has the worst performance, which can be explained as follows: (1) The local pruning of the homogeneous strategy is less effective since the distances of the trajectories in a partition to the query trajectory are very similar. (2) In the heterogeneous strategy, a high degree of data discrimination in a partition is able to improve the local pruning. In addition, since the composition of each partition is similar, the query times of all partitions are similar, which enables load-balancing.

Comparison with DITA and DFT using heterogeneous partitioning. To further explore the benefits of the heterogeneous partitioning strategy, we apply it to DITA and DFT to examine the effects on performance. We denote these two methods as Heter-DITA and Heter-DFT. The results are shown in Tables VIII and IX. Since DITA does not support Hausdorff, we examine its performance on DTW instead. The results show that both Heter-DITA and Heter-DFT outperform the original DITA and DFT, which offers evidence that the heterogeneous partitioning strategy is superior.

TABLE VII
EFFECT OF PARTITIONING STRATEGY

Distance	Partitioning	T-drive (s)	Xi'an (s)	OSM (s)
Hausdorff	Heterogeneous	**1.29**	**19.14**	**32.16**
	Homogeneous	1.64	35.93	344.74
	Random	1.51	21.38	35.12
Frechet	Heterogeneous	**1.28**	**43.27**	**35.61**
	Homogeneous	1.52	109.42	240.32
	Random	1.45	47.94	37.68

TABLE VIII
COMPARISON WITH DITA USING HETEROGENEOUS PARTITIONING

Distance	Algorithm	T-drive (s)	Xi'an (s)	OSM (s)
DTW	REPOSE	**1.48**	**180.09**	**499.52**
	Heter-DITA	2.74	184.97	503.59
	DITA	2.78	186.36	509.53
Frechet	REPOSE	**1.28**	**43.27**	**35.61**
	Heter-DITA	2.26	84.94	87.56
	DITA	2.39	91.31	94.12

VIII. RELATED WORK

Trajectory similarity search is a fundamental operation in many applications [3], [4], [11]–[14], [16], [25], [26], [28]. Based on the index structures, we classify the existing proposals into three categories:

(1) Space partitioning tree based approach. Traditional space partitioning trees [13], [16], [26], [27] (e.g., R-tree) construct MBRs by dividing trajectories into sub-trajectories and prune non-candidate trajectories based on the MBR intersections. However, large MBR areas limit the pruning capabilities. For metric spaces, classical metric indexes, such as the VP-tree and the M-tree [15], [24], [32], are used for Hausdorff and Frechet. However, these methods perform poorly in practice since they abandon the features of the trajectory and use only the metric characteristics to index the trajectories.

(2) Grid hashing based approach. The grid hashing based approaches [2], [12] use the grid sequence that the trajectories traverse to represent the trajectories, which captures the trajectory characteristics and makes it possible to quickly find candidate trajectories. Although these methods fail to return exact results, they demonstrate good performances in approximate trajectory similarity search.

(3) Distributed index based approach. SmartTrace$^+$ [35] is a real-time distributed trajectory similarity search framework for smartphones that employs a similar pruning and refinement paradigm of REPOSE. DFT [28] uses the R-tree to index trajectory line segments and prunes by using the distances between a query trajectory and the MBRs of line segments. Unfortunately, DFT needs to reconstruct the line segments

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.

TABLE IX
COMPARISON WITH DFT USING HETEROGENEOUS PARTITIONING

Distance	Algorithm	T-drive (s)	Xi'an (s)	OSM (s)
Hausdorff	REPOSE	**1.29**	**19.14**	**32.16**
	Heter-DFT	2.32	1645.59	260.77
	DFT	2.50	1857.88	273.68
Frechet	REPOSE	**1.28**	**43.27**	**35.61**
	Heter-DFT	2.21	1855.01	380.47
	DFT	2.33	2100.80	418.40

into trajectories when computing similarity, which incurs high space overhead. DITA [19] selects pivot points to represent a trajectory and stores close pivot points in the same MBR. Then, DITA utilizes the trie to index all MBRs and prunes by using the distances between the query trajectory and the MBRs. However, DITA fails to retain the features of original trajectories due to its pivot point selection strategy. Moreover, the query processing is inefficient for some distance metrics.

IX. CONCLUSION AND FUTURE WORK

We present a distributed in-memory framework, REPOSE, for processing top-k trajectory similarity queries on Spark. We develop a heterogeneous global partitioning strategy that places similar trajectories in different partitions in order to achieve load balancing and to utilize all compute nodes during query processing. We design an effective index structure, RP-Trie, for local search that utilizes two optimizations, namely a succinct trie structure and z-value re-arrangement. To further accelerate the query processing, we provide several pruning techniques. The experimental study uses 7 widely used datasets and offers evidence that REPOSE outperforms its competitors in terms of query time, index size and index construction time. Compared to the state-of-the-art proposals for Hausdorff, REPOSE improves the query time by about 60%, the index size by about 80%, and the index construction time by about 85%.

ACKNOWLEDGMENTS

This research is supported in part by the NSFC (Grants No. 61902134, 62011530437), the Hubei Natural Science Foundation (Grant No. 2020CFB871), and the Fundamental Research Funds for the Central Universities (HUST: Grants No. 2019kfyXKJC021, 2019kfyXJJS091).

REFERENCES

[1] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *IJCGA*, 5:75–91, 1995.
[2] M. S. Astefanoaei, P. Cesaretti, P. Katsikouli, M. Goswami, and R. Sarkar. Multi-resolution sketches and locality sensitive hashing for fast trajectory processing. In *SIGSPATIAL/GIS*, pages 279–288, 2018.
[3] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras. Efficient trajectory joins using symbolic representations. In *MDM*, pages 86–93, 2005.
[4] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *GIS*, pages 182–191, 2005.
[5] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognit. Lett.*, 24(14):2357–2366, 2003.
[6] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang. Pivot-based metric indexing. *PVLDB*, 10(10):1058–1069, 2017.
[7] L. Chen and R. T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.
[8] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.
[9] H. K. Dai and H. Su. On the locality properties of space-filling curves. In *ISAAC*, volume 2906, pages 385–394, 2003.
[10] P. Dewan, R. K. Ganti, and M. Srivatsa. SOM-TC: self-organizing map for hierarchical trajectory clustering. In *ICDCS*, pages 1042–1052, 2017.
[11] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.
[12] A. Driemel and F. Silvestri. Locality-sensitive hashing of curves. In *SoCG*, LIPIcs, pages 37:1–37:16, 2017.
[13] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.
[14] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.
[15] A. W. Fu, P. M. Chan, Y. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDBJ*, 9(2):154–173, 2000.
[16] E. J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, pages 406–417, 2002.
[17] D. Pfoser, S. Brakatsoulas, P. Brosch, M. Umlauft, N. Tryfona, and G. Tsironis. Dynamic travel time provision for road networks. In *GIS*, page 68, 2008.
[18] D. Pfoser, N. Tryfona, and A. Voisard. Dynamic travel time maps - enabling efficient navigation. In *SSDBM*, pages 369–378, 2006.
[19] Z. Shang, G. Li, and Z. Bao. DITA: distributed in-memory trajectory analytics. In *SIGMOD*, pages 725–740, 2018.
[20] M. B. Shapiro. The choice of reference points in best-match file searching. *Commun. ACM*, 20(5):339–343, 1977.
[21] T. Skopal, J. Pokorný, and V. Snásel. Pm-tree: Pivoting metric tree for similarity search in multimedia databases. In *ADBIS*, 2004.
[22] H. Su, S. Liu, B. Zheng, X. Zhou, and K. Zheng. A survey of trajectory distance measures and performance evaluation. *VLDBJ*, 29(1):3–32, 2020.
[23] N. Ta, G. Li, Y. Xie, C. Li, S. Hao, and J. Feng. Signature-based trajectory similarity join. *TKDE*, 29(4):870–883, 2017.
[24] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.
[25] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.
[26] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. J. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *KDD*, pages 216–225, 2003.
[27] Y. Wang, Y. Zheng, and Y. Xue. Travel time estimation of a path using sparse trajectories. In *KDD*, pages 25–34, 2014.
[28] D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.
[29] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Massively distributed time series indexing and querying. *TKDE*, 32(1):108–120, 2020.
[30] K. Yamamoto, K. Uesugi, and T. Watanabe. Adaptive routing of multiple taxis by mutual exchange of pathways. *IJKESDP*, 2(1):57–69, 2010.
[31] B. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.
[32] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.
[33] J. Yuan, Y. Zheng, X. Xie, and G. Sun. T-drive: Enhancing driving directions with taxi drivers' intelligence. *TKDE*, 25(1):220–232, 2013.
[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.
[35] D. Zeinalipour-Yazti, C. Laoudias, C. Costa, M. Vlachos, M. I. Andreou, and D. Gunopulos. Crowdsourced trace similarity with smartphones. *TKDE*, 25(6):1240–1253, 2013.
[36] D. Zhang, N. Li, Z. Zhou, C. Chen, L. Sun, and S. Li. ibat: detecting anomalous taxi trajectories from GPS traces. In *UbiComp*, pages 99–108, 2011.
[37] H. Zhang, H. Lim, V. Leis, D. G. Andersen, K. Keeton, and A. Pavlo. Succinct range filters. *SIGMOD*, 48(1):78–85, 2019.

Authorized licensed use limited to: University of Electronic Science and Tech of China. Downloaded on October 13,2026 at 12:27:27 UTC from IEEE Xplore. Restrictions apply.