TraSS: Efficient Trajectory Similarity Search Based on Key-Value Data Stores

Huajun He[1,2,3], Ruiyuan Li[4], Sijie Ruan[5,6], Tianfu He[2,3], Jie Bao[2,3], Tianrui Li[1], Yu Zheng[1,2,3*]
[1]*Southwest Jiaotong University, Chengdu, China* [2]*JD iCity, JD Technology, Beijing, China* [3]JD Intelligent Cities Research, China [4]*Chongqing University, China* [5]*Xidian University, Xi'an, China* [6]*Nanyang Technological University, Singapore*
hehuajun@my.swjtu.edu.cn;{ruiyuan.li,sijieruan,msyuzheng}@outlook.com;{hetianfu3,baojie3}@jd.com;trli@swjtu.edu.cn

Abstract—**Similarity search has recently become an integral part of many trajectory data analysis tasks. As the number of trajectories increases, we must find similar trajectories among massive trajectories, necessitating a scalable and efficient framework. Typically, massive trajectory data can be managed by key-value data stores. However, existing works with key-value data stores use a coarse representation to store trajectory data. Besides, they do not provide efficient query processing to search similar trajectories. Thus, this paper proposes *TraSS*, an efficient framework for trajectory similarity search in key-value data stores. We propose a novel spatial index, XZ*, which utilizes fine-grained index spaces with irregular shapes and sizes to represent trajectories elaborately. Further, we devise a bijective function from the index spaces of XZ* to continuous integers, which is simple but effective for query processing. To improve the efficiency of similarity search, we employ two steps to prune dissimilar trajectories: (1) global pruning. It leverages the XZ* index to prune index spaces with no trajectories similar to the query trajectory. Our global pruning can only pick out index spaces with similar sizes and shapes to the query trajectory. Compared to the state-of-the-art index, our global pruning reduces I/O overhead up to 66.4% during query processing; (2) local filtering. It filters dissimilar trajectories in a way with low complexity. We use a few representative features extracted from a trajectory by the Douglas-Peucker algorithm to accelerate the local filtering. We implement an open-source toolkit (*TraSS*) on a popular key-value data store. Extensive experiments show that *TraSS* outperforms state-of-the-art solutions.**

Index Terms—**similarity search, trajectory data management, key-value database, trajectory index**

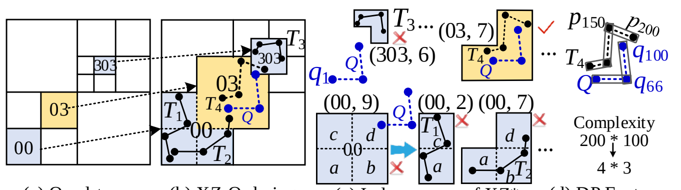
I. Introduction

In modern times, Internet of Things technology enables us to easily capture trajectories of moving objects. As a result, a large number of trajectories have been collected. For example, there are more than 1TB trajectory logs generated by over 60,000 couriers of JingDong each day [1]. T-Drive [2] contains 790 million trajectories collected in Beijing over only three months. Such massive trajectory data calls for efficient and scalable data management. There are many query operations for trajectory data management, e.g., the spatial range query finds trajectories passing a given spatial range, and the similarity query searches all trajectories that are similar to a given trajectory. Among these queries, trajectory similarity search is a vital and fundamental operation. For example, to find the close contacts of a patient with an infectious disease, we would look for trajectories that are similar to the patient's trajectory [3]. Trajectory similarity search is also conducive to carpooling trajectory clustering. Key-value data stores such as HBase [4] are widely used in big data management because they achieve fast write/read throughput and fast lookup on the row key. There are some works [5]–[10] utilize key-value data stores for big trajectory data management. However, they do not provide trajectory similarity search efficiently because (1) they store trajectory with a coarse **representation**, and (2) they lack efficient **query processing** to search similar trajectories.

Representation. A trajectory is composed of multi-dimensional points, but key-value data stores store objects using one-dimensional key-value pairs. Thus, we must allocate an appropriate key to a trajectory. Existing works use spatial indexes, such as R-trees [11]–[16] and XZ-Ordering [17], to represent trajectories in key-value data stores. They distribute massive trajectories into many index spaces. Each index space has a spatial boundary and is equipped with a unique value. Thus, we can use that unique value as a part of the key to store and query data. Dynamic indexes such as the R-tree and its variants are inevitable to adjust index structures when inserting much data [18]. Thus, existing works [19], [20] that use dynamic indexes suffer from maintainability and scalability problems in big trajectory data management. Demonstrated by [8], [9], [21], [22], XZ-Ordering is a better choice to represent trajectories in key-value data stores. It can be regarded as an extension of a quad-tree by doubling the width and height of each sub-space, as indicated by the arrows in Figure 1(a)(b). XZ-Ordering utilizes the smallest index space to represent a trajectory's MBR (minimum bounding box). As shown in Figure 1(b), T_1 is represented by a bigger index space ('00'), whereas T_3 is denoted by a smaller index space ('303'). However, the points of a trajectory may only appear in a small portion of the index space, e.g., T_1 occupies less than half of its index space ("00"). As a result, the index space of XZ-Ordering is still too coarse for a trajectory. Thus, when querying, it is laborious to filter out dissimilar trajectories by pruning index spaces of XZ-Ordering only.

Query processing. Calculating the similarity of two trajectories is time-consuming, as similarity measures tend to be rather complex. Therefore, it is essential to prune as many dissimilar trajectories as possible in advance to avoid incurring unnecessary I/O overhead and calculation during query processing. The process of scanning all trajectories in the regions and calculating their similarities to a query trajectory is time-consuming and irrational. Thus, two steps are frequently adopted to improve query efficiency: (1) **global**

*Yu Zheng is the corresponding author.



(a) Quad-tree. (b) XZ-Ordering. (c) Index spaces of XZ*. (d) DP Features.

Fig. 1. Representation of Trajectories.

pruning avoids visiting irrelevant storage regions. Two similar trajectories are likely to be spatially close to each other. Thus, index spaces that are spatially far from the query trajectory can be pruned. Typically, the index space of most existing spatial indexes is a rectangle encompassing the MBRs of trajectories. Many existing works [8], [10], [21] do not prune index spaces that intersect the MBR of a query trajectory, resulting in a large number of false hits that cause unnecessary I/O overhead. As shown in Figure 1(b), T_1, T_2 and T_3 are false hits because they are far away from Q; (2) **local filtering** rapidly removes dissimilar trajectories in each data region in a low-complexity way. Existing works [8], [9], [19], [20] use the MBR or pivot points (e.g., start and end points) of a trajectory to filter dissimilar trajectories. However, these features cannot meticulously represent the spatial shape of a trajectory.

Our Solution. To address the shortcomings above, we propose $TraSS$, an efficient framework for **Tra**jectory **S**imilarity **S**earch on key-value data stores. (**Representation**). We propose the XZ* index to represent as many similar trajectories as possible within the same index space. Specifically, it divides each enlarged element of XZ-Ordering into four equal sub-quads and uses the combinations of sub-quads as index spaces. A larger index space is used to cover bigger trajectories, while a smaller index space is used to cover smaller trajectories. Additionally, trajectories of similar size can be differentiated by their spatial shapes. As illustrated in Figure 1(c), we divide an enlarged element ("00") of XZ-Ordering into four equal parts *a*, *b*, *c*, *d*, and then use the combinations of them represent trajectories with different shapes, e.g., (00, 2) and (00, 7) represents T_1 and T_2, respectively. Additionally, we devise a bijective function that converts the index spaces to continuous integer values, instead of maintaining an index structure in memory. It consumes less storage overhead than string encoding and improves maintainability and scalability. (**Query processing**). (**Global pruning**). Because our index can precisely describe the size and shape of a trajectory, we would only visit a few index spaces similar to a query trajectory Q, avoiding visiting trajectories covered by index spaces that are dissimilar to Q. Intuitively, index spaces that are too large or too small than Q, and index spaces with similar sizes to Q but are far from Q can be directly pruned. As illustrated in Figure 1(c), part-a of '00' is far away from Q, so that index spaces combined by part-a can be pruned. Additionally, in Figure 1(c), q_1 of Q is far away from the index space of (303, 6), implying that Q is dissimilar to the trajectories covered by (303, 6). In theory, compared with a state-of-the-art solution, we can reduce I/O overhead by up to 83.6% (cf. Section IV-B). As illustrated in Figure 1(c), only the index space of (03, 7) can cover the trajectories that are similar to Q. (**Local filtering**). We extract representative points from a trajectory using the DP (Douglas-Peucker) algorithm [23], which leverages very few points to approximate the spatial shape of a trajectory. Besides, we use bounding boxes to represent points between any two continuous representative points. We can quickly filter out dissimilar trajectories because the numbers of representative points and bounding boxes are much smaller than points in a trajectory. As shown in Figure 1(d), we use four points and three bounding boxes to represent a 200-point trajectory T_4, p_{150} is the third representative point of T_4 and is far away from Q, so we can quickly eliminate T_4. Finally, only a few trajectories must calculate the inevitable and time-consuming similarity after global pruning and local filtering. To sum up, the contributions of this paper are three-fold:

- We propose a novel spatial index, XZ*. It represents a trajectory using a fine-grained index space with different sizes and irregular shapes. Theoretically, it can reduce I/O overhead up to 83.6% compared to XZ-Ordering. We propose a bijective function from index spaces of XZ* to continuous integer values, which retains the spatial characteristics of XZ* and improves the maintainability and scalability, reducing storage overhead on row keys by up to 32% compared to string encoding.
- We devise two efficient query processing algorithms to execute similarity searches based on a given threshold and top-*k* similarity search. Our global pruning strategy carefully generates range scans of index spaces instead of scanning all index spaces intersecting with a query trajectory. Experimentally, it reduces I/O overhead up to 66.4% compared to XZ-Ordering. The local filtering uses the DP features to minimize the overhead associated with calculating the similarity of two trajectories.
- We conduct extensive experiments on two real and five synthetic datasets. Additionally, we create an open-source toolkit called *TraSS* [24] by integrating our framework into HBase, which outperforms other solutions.

The rest of the paper is organized as follows. Section II formalizes the problem of trajectory similarity search. Section III gives the overview of our framework. Section IV introduces the representation of a trajectory, including the design of the index structure and encoding. Section V shows pruning strategies for query processing. Section VI evaluates the performance of our framework. Section VIII discusses the related works. Section IX is the conclusion.

II. Preliminary

A. Definitions

Definition 1. *(Trajectory). A trajectory T is a collection of multidimensional points, defined as $T = (t_1, ..., t_n)$.*

For simplicity, we represent each point as a 2-dimensional tuple, e.g., $t_i = (latitude, longitude)$. It can be easily extended to support multidimensional data. We use T and T_i to represent a trajectory or a trajetory with ID i, respectively. $\mathcal{T} = \{T_1, ..., T_k\}$ denotes a set of trajectories. T^j denotes the prefix of T up to the j-th point. For example, if $T = (t_1, ..., t_{10})$, then $T^3 = (t_1, t_2, t_3)$.

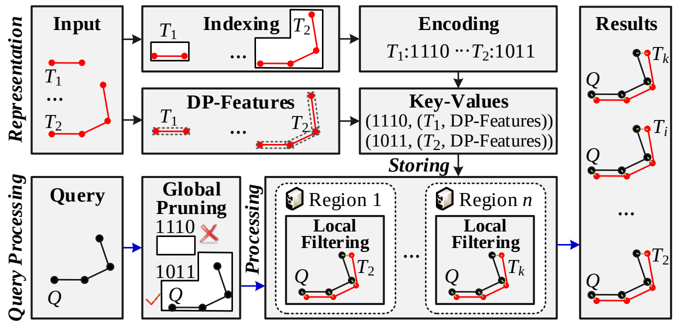


Fig. 2. Overview of TraSS.

Similarity Distance. There exist a variety of trajectory similarity measures. Instead of designing a new similarity measure, we adopt classic measures [25] in our framework, e.g., Discrete Fréchet distance, Hausdorff distance, and DTW. In this paper, we use Discrete Fréchet distance as the default similarity measure. Other similarity measures are discussed in Section VII. Given two trajectories $Q = (q_1, ..., q_n)$ and $T = (t_1, ..., t_m)$, the definition of Discrete Fréchet distance is as follows:

Definition 2. *(Discrete Fréchet).*

$$D_F\left(Q^n, T^m\right) = \begin{cases} \max\limits_{k=1}^{m} d(q_1, t_k) & \text{if } n = 1 \\ \max\limits_{k=1}^{n} d(q_k, t_1) & \text{if } m = 1 \\ \max\{d(q_n, t_m), \min\{D_F(Q^{n-1}, T^m), \\ D_F(Q^n, T^{m-1}), D_F(Q^{n-1}, T^{m-1})\}\} \end{cases},$$

where $Q = Q^n = (q_1, ..., q_n)$, $T = T^m = (t_1, ..., t_m)$, and $d(p_1, p_2)$ is the Euclidean distance of two points.

B. Problem Formulation

We use $f(Q, T)$ to represent the similarity distance of two trajectories Q and T, where $f(Q, T)$ can be Fréchet (default measure), Hausdorff, or DTW. In this paper, we aim to solve two types of the trajectory similarity search problem.

Definition 3. *(Threshold Similarity Search). Given a query trajectory Q and a threshold ε, the threshold similarity search finds trajectories $\mathcal{T}$, such that $\forall\, T \in \mathcal{T}$, we have $f(Q, T) \leq \varepsilon$.*

Definition 4. *(Top-k Similarity Search). Given a query trajectory Q and an integer number k, the top-k trajectory similarity search finds k trajectories $\mathcal{T}$, such that $|\mathcal{T}| = k$, for $\forall\, T_i \in \mathcal{T}$ and $\forall\, T_j \notin \mathcal{T}$, we have $f(Q, T_i) < f(Q, T_j)$.*

III. Overview

Figure 2 gives an overview of our framework. Its core operations are representation and query processing.

Representation. We propose the XZ* index that provides an index space with an irregular shape to represent a trajectory (cf. Section IV-B). Then, we number each index space with a unique integer for conveniently storing and querying trajectories (cf. Sections IV-C and IV-E). Besides, we calculate the representative features of a trajectory for accelerating the query processing (cf. Section IV-D). As shown in Figure 2 (*Representation*), trajectories T_1 and T_2 are indexed by suitable index spaces, equipped with DP-features, and then encoded with elaborate codes (1110 and 1011, respectively) for storing.

Query Processing. We provide efficient query processing to find similar trajectories from the database. As shown in Figure 2 (*Query Processing*), given a query trajectory Q, we first use the global pruning to filter out index spaces that are impossible to contain trajectories similar to the query trajectory (cf. Section V-C). Then, we eliminate trajectories that would not be the final answers in the remaining index spaces through the local filtering (cf. Section V-D) on each data region. Section V-E gives the detailed steps.

IV. Representation

This section explores how to represent a trajectory in a key-value data store. We first introduce the main idea of designing our spatial index for key-value data stores in Section IV-A. Based on the main idea, XZ* index is proposed, which includes two main operations, namely Indexing (Section IV-B) and Encoding (Section IV-C). Then, we extract representative features to approximately represent a trajectory (Section IV-D). Finally, Section IV-E introduces how to store trajectories.

A. Main Idea of XZ* Index

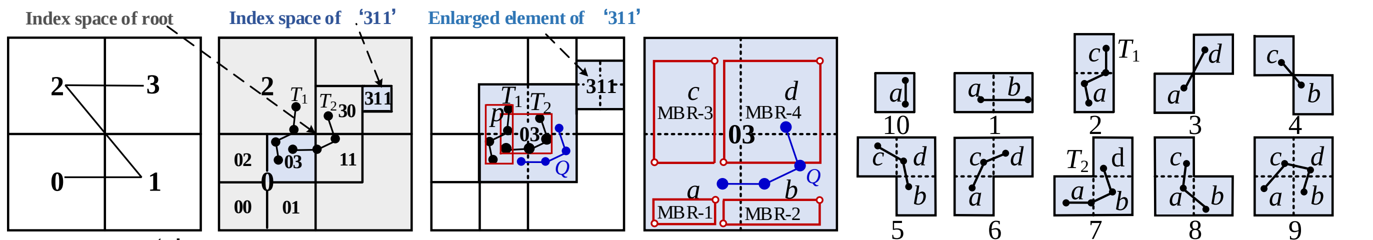
A trajectory is a sequence of multidimensional points, while key-value data stores use one-dimensional key-value pairs to store data. To use a one-dimensional key to represent a trajectory, we should divide the space into multiple partitions and design an elaborate encoding for the spatial partitions. Therefore, we propose XZ* index.

Index structure of XZ* index. The existing spatial index represents a trajectory using the grid cell that covers the MBR of the trajectory. Because MBR is a coarse description of trajectory shape, and the grid cell is also an inaccurate representation of the MBR, trajectories with significant different shapes would be indexed by the same grid cell, which is not conducive to the efficiency of pruning. To this end, we provide a fine-grained index space, which uses an irregular index space to represent the shape of a trajectory meticulously.

Encoding of XZ* index. After using suitable index spaces to represent trajectories, key-value data stores need one-dimensional row keys to store and query them. Thus, we devise a bijective function to map index spaces to integers, which allocates a unique index value to an index space of the XZ*.

B. Indexing

We propose a fine-grained static index, namely XZ*, which can represent the shape of a trajectory. First, we divide the entire space into many sub-spaces with different resolutions using the rule of a quad-tree, then give each sub-space a **quadrant sequence**. After that, inspired by the **enlarged element** of XZ-ordering [17], we double the height and width of each sub-space towards the upper-right corner to cover trajectories with different sizes. Finally, to approximately reflect the shape of a trajectory, we evenly divide the enlarged element into four sub-quads. We use the combination of sub-quads as an index space to represent trajectories and give each combination a



(a) Reversed 'Z'. (b) Quad-tree. (c) Enlarged Element. (d) Enlarged Element(03). (e) Index Spaces (10 appears only at the maximum resolution).

Fig. 3. Index Structures of Existing Indexes and XZ* Index.

position code to represent its related position in an enlarged element, as shown in Figure 3(e). In this way, we can use an irregular index space to represent a trajectory meticulously.

Quadrant Sequence. We adopt the rule of a quad-tree to recursively divide the current index space into four equal quads until the maximum resolution is reached, where the resolution is the number of recursions. As shown in Figure 3(b), '2' is at 1-resolution and '00' is at 2-resolution, respectively. Each split will generate four sub-quads, and we number them from 0 to 3 like the order of a reversed 'Z,' as shown in Figure 3(a). Each sub-space is associated with a quadrant sequence, such as '00', '30' and '311' shown in Figure 3(b). The length of a quadrant sequence is equal to its resolution. A larger resolution of a quadrant sequence means a finer area of index space. As shown in Figure 3(b), '311' represents an index space at 3-resolution, and its area is smaller than the index space of '30'.

Enlarged Element. However, a trajectory may intersect more than one sub-quads and would be completely contained by a big sub-quad, e.g., T_2 in Figure 3(b) appears in '03','11' and '30,' but the smallest sub-quad that can contain T_2 is the 'root'. It is too large for T_2, which may cause the query processing tough to prune it by the space of 'root'. Inspired by XZ-Ordering, we extend each sub-quad to an enlarged element by doubling the height and width of the quad towards the upper-right corner. As shown in Figure 3(c), spaces with light blue color are the enlarged elements of '03' and '311', respectively. We use the MBR of a trajectory to locate the smallest enlarged element that can cover the trajectory. As shown in Figure 3(c), the red rectangles are the MBRs of T_1 and T_2, and the enlarged element of '03' covers them. Thus, T_2 can be represented by the enlarged element of '03' rather than the space of the 'root', which makes the representation of the trajectory's MBR more acceptable.

We use s to stand for the quadrant sequence of the smallest enlarged element. Because the expansion direction of an enlarged element is towards the upper-right corner, the smallest enlarged element covering the MBR is always expanded from the sub-quad at the lower-left corner of the MBR. Thus, s is a prefix of the maximum quadrant sequence generated by the lower-left corner of the MBR. The length of s is determined by the height and width of the MBR. It can be calculated by Lemmas 1 and 2. Without losing generality, we normalize the entire space range to an interval of 0-1.

Lemma 1. *The most appropriate quadrant sequence s of an MBR $((x_1, y_1), (x_2, y_2))$ has a length of $|s| = l$ or $l + 1$, where, $l = \lfloor log_{0.5}(max\{x_2 - x_1, y_2 - y_1\}) \rfloor$.*

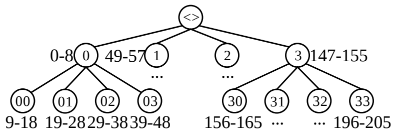
Lemma 2. *Let $w = 0.5^{l+1}$, if $\lfloor \frac{x_1}{w} \rfloor * w + 2 \cdot w \geq x_2$ and $\lfloor \frac{y_1}{w} \rfloor * w + 2 \cdot w \geq y_2$, then $|s| = l$, else $|s| = l + 1$.*

Proof. Proofs of Lemmas 1 and 2 can refer to [17], [24]. □

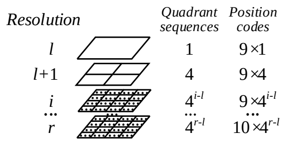
Position Code. Similar trajectories usually have similar shapes. However, the enlarged element covering the MBR of a trajectory is still too large, which is not friendly to describe the shape of a trajectory, so it cannot prune many invalid trajectories during the query processing. As shown in Figure 3(c), Q is a query trajectory that is contained by the enlarged element of '03', so all trajectories indexed by '03' must be extracted from the database and compared with Q. However, T_1 is dissimilar to Q because T_1 has a point p at the left-upper corner of '03' that is far away from Q. Thus, we propose a concept of position code that uses an irregular index space to reflect the shape of a trajectory. Then, we generate meticulous position codes to avoid extracting trajectories with dissimilar shapes. We evenly divide each enlarged element into four sub-quads, marked as a, b, c, d, respectively, as shown in Figure 3(d). We use the combination of sub-quads to represent trajectories. Each combination corresponds to an index space, as shown in Figure 3(e). Note that any index space of an enlarged element at a resolution lower than the maximum resolution must contain at least two sub-quads, because if an index space contains only one sub-quad, it would be represented by an enlarged element at a larger resolution.

An enlarged element represents the MBRs whose lower-left corners must locate in quad-a. Thus, according to the positions of their upper right corners, there are only four kinds of MBRs in an enlarged element. As shown in Figure 3(d), the upper right corners of the MBR-1, MBR-2, MBR-3, and MBR-4, are located in the quads of a, b, c, d, respectively. We enumerate all kinds of trajectories in each MBR, obtaining ten combinations of sub-quads. We give each combination a position code, as shown in Figure 3(e), where trajectories in '10', '1', '2', and from '3' to '9' are equipped with the MBR-1, MBR-2, MBR-3, and MBR-4, respectively. Note that the index space of '10' contains only quad-a. It occurs in the maximum resolution.

Discussion. In a nutshell, a trajectory would be represented by the form of (*quadrant sequence*, *position code*). For example, T_1 and T_2 are represented by ('03', 2) and ('03', 7), respectively. It helps query processing quickly prune trajectories with dissimilar shapes. Because similar trajectories usually have similar spatial shapes, if a quad is far away from a query trajectory, then the index spaces containing this quad can be pruned (the reason is discussed in Lemma 10). Given a query trajectory Q, assuming that quad-c is far away from Q, such that we do not need to extract trajectories indexed with position codes 2, 4, 5, 6, 8, 9. Assuming that trajectories are evenly distributed in ten index spaces, we can reduce I/O overhead by 60% ($\frac{6}{10}$). Analogously, for quad-a, quad-b, and



(a) Encoding (r =2).



(b) Index Spaces Prefixed with s.

Fig. 4. Encoding of The XZ* Index.

quad-d, we can reduce I/O overhead by 80%, 60%, 50%, respectively. Moreover, if two or three quads are far away from Q, we can eliminate more index spaces. For example, if quad-b and quad-c are both away from Q, except for position codes 10 and 3, we can discard other index spaces of the enlarged element so that we reduce I/O overhead by 80%. Analogously, for quad-ab, ac, ad, bd, cd, abc, abd, acd, bcd, we can reduce I/O overhead by 100%, 100%, 90%, 80%, 80%, 100%, 100%, 100%, 90%, respectively. On average, we reduce I/O overhead by 83.6%. In Section VI-D gives an experimental result.

C. Encoding

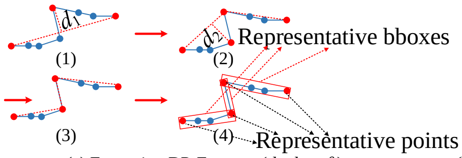
Key-value data stores use the form of key-value pair to store data. We devise a bijective function to generate the key to support storage and query better. The advantages of our encoding are: (1) the index space of a trajectory can be calculated by a mathematical formula without maintaining the index structure in the memory. Thus, we can reduce much maintenance cost to manage trajectories; (2) the storage overhead is less than the string encoding. For example, for an index space at 16-resolution, the string encoding requires 16 bytes for a quadrant sequence and one byte for a position code, while our encoding only requires an integer of 8 bytes so that we can reduce storage overhead on a key by about 53% ($\frac{17-8}{17}$); (3) using the simple concatenation will make the encoding discontinuous, which will increase the number of key range searches when querying and reduce the performance.

Main idea. The smaller the resolution, the larger the index space, and the lower similarity between two index spaces at smaller resolutions. Thus, we use a deep-first strategy to design our encoding, ensuring that the longer the same prefix of two-quadrant sequences, the closer their converted numbers are. The algorithm begins at the root node and numbers index spaces in depth-first visiting order. As described in Section IV-B, every enlarged element at i-resolution ($i < r$, r is the maximum resolution) has nine index spaces, and at r-resolution it has ten index spaces, respectively. As shown in Figure 4(a), the maximum resolution is 2. We first number index spaces at '0' from 0 to 8 and number index spaces at '00' from 9 to 18. Finally, we number '33' from 196 to 205.

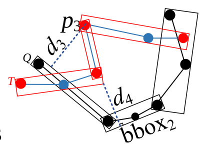
Numbering. In order to number all index spaces of XZ*, we must guarantee that the interval between the converting values of any two index spaces (s_1, p_1) and (s_2, p_2) can exactly accommodate all index spaces between (s_1, p_1) and (s_2, p_2). Lemmas 3 and 4 calculate the total number of the index spaces under a quadrant sequence s with length l, where $s = (q_1 ... q_l)$.

Lemma 3. *The total number of quadrant sequences at resolution i ($0 < l \leq i \leq r$) prefixed with s is*

$$N_{qs}(i, l) = 4^{i-l}, \tag{1}$$



(a) Extracting DP-Features ($d_1, d_2 > \theta$).



(b) Filtering by DP-Features.

Fig. 5. Example of DP Features.

where $l = |s|$ is the length of s and implies its resolution.

Lemma 4. *The number of index spaces prefixed with s is*

$$N_{is}(l) = 10 * N_{qs}(r, l) + \sum_{i=l}^{r-1} 9 * N_{qs}(i, l) = 13 \times 4^{r-l} - 3, \tag{2}$$

where r is the maximum resolution. Notably, $N_{is}(l)$ can also represent the number of index spaces prefixed with any quadrant sequence with the length of l.

Proof. There are 4^l quadrant sequences at l-resolution, and 4^i quadrant sequences at i-resolution. Therefore, there are 4^{i-l} quadrant sequences with length i prefixed with a same quadrant sequence with length l, as shown in Figure 4(b). Thus, the total of index spaces prefixed with s is Equation(2). □

Given a quadrant sequence $s = \langle q_1 ... q_i ... q_l \rangle$ and a position code p of a trajectory, we can calculate its index value by Definition 5. It accumulates each quadrant number q_i ($i < l$) in s multiplies $N_{is}(i)$ and plus its corresponding index spaces (i.e., $\sum_{i=1}^{l}(q_i * N_{is}(i) + 9)$), then plus $q_l * N_{is}(l) + p - 1$ to obtain the final index value. Thus, our encoding is as follows:

Definition 5. *(Index value). The index value $V(s, p)$ of a quadrant sequence s and a position code p is*

$$V(s, p) = \sum_{i=1}^{l-1} [q_i * N_{is}(i) + 9] + q_l * N_{is}(l) + p - 1. \tag{3}$$

As shown in Figure 3(c) and Figure 3(e), index values of T_1 and T_2 are V('03', 2) = 40 and V('03', 7) = 45, respectively.

V is a bijection function because each index space has only one index value, and an index value has only one index space. Moreover, the lexicographical order of quadrant sequences and position codes corresponds to the less-equal order of index values, so V retains the spatial characteristics of XZ*.

D. DP Features

Trajectories in the real world may contain many points, so the computation of similarity is expensive. However, many consecutive points may be very close to each other. Thus, we use the Douglas-Peucker (DP) algorithm [23] to reduce the size of trajectories without affecting the accuracy. DP is an algorithm with low complexity and can be effortless to implement. It finds representative points of a trajectory. The distance from any points between the line of two successive representative points is lower than a predefined distance θ. As shown in Figure 5(a), after three iterations, we obtain four representative points (red points in Figure 5(a)(4)) to represent the trajectory. Besides, we further extract the bounding box (abbreviated as bbox, it is not necessarily parallel to the coordinate axis) to cover points among any two successive representative points, as the red bboxes shown in Figure 5(a)(4).

E. Storing

We use an irregular index space to represent a trajectory and assign an index value for the index space. Thus, we can use the index value as the spatial key for storing and querying trajectories. The key is combined as follows,

$$rowkey = shards + index\ value + tid,$$

where '+' is the concatenation operation; *shards* is a hash number to decentralize trajectories, which can avoid hot-spotting problem; *index value* represents the spatial information of the trajectory, which is calculated by Equation (3); tid is the identifier of a trajectory.

TABLE I
SCHEMA OF TRAJECTORY TABLE

rowkey	**value**				
	tid	**points**	**dp-points**	**dp-mbrs**	...
01110t1	t1	MultiPoint(...)	List⟨Integer⟩	MultiPolygon(...)	..
11011t2	t2	MultiPoint(...)	List⟨Integer⟩	MultiPolygon(...)	...

Table I gives the schema of a trajectory table, where tid is the identifier and $points$ stores the points of a trajectory. Notably, to efficiently execute the trajectory similarity search, we also need columns to store some valuable features, e.g., $dp\text{-}points$ records the indexes of DP-points in the raw points and $dp\text{-}mbrs$ stores the MBRs of DP-features. Note that most key-value stores have an automatic partitioning strategy, so this paper does not take time to design a partitioning strategy.

V. QUERYING PROCESSING

In this section, we describe how trajectory similarity queries are processed. Specifically, we first give our main idea to avoid redundant searches in Section V-A. Based on the main idea, two pruning strategies are proposed, namely *Global Pruning* (Section V-C) and *Local Filtering* (Section V-D). Furthermore, in Section V-E, we will show how the pruning strategies improve the two typical trajectory similarity queries, i.e., *Threshold Similarity Search*, and *Top-k Similarity Search*.

A. Main Idea

Trajectories are stored by the step of Section IV-E, and searched by the row keys. Each row key is equipped with an index value that represents an index space covering the shape of a trajectory. Query processing eliminates index spaces that are dissimilar to the query trajectory and avoids calculating complicated similarities with unnecessary trajectories.

In Section V-B, we propose Lemma 5, which is fundamental for other lemmas of Section V-C, Section V-D and Section VII

In Section V-C, we design the global pruning strategy to delicately filter out index spaces containing trajectories that are dissimilar to a query trajectory. An index space is represented by an enlarged element and a position code. Section V-C1 gives Lemmas to prune out unnecessary enlarged elements, and Section V-C2 introduces Lemmas to filter out impossible position codes of remaining enlarged elements.

After executing the global pruning, we extract trajectories of the remaining index spaces in each region. In Section V-D, to alleviate the computation of complicated similarity, we propose local filtering based on representative features of trajectories to filter out dissimilar trajectories efficiently.

Based on global pruning and local filtering, we implement two frequently-used similarity searches in Section V-E. The performance of our Lemmas increases step by step, but the computation also goes up. Thus, we execute Lemmas from simple to complex to reduce the computation.

B. Basic Lemma and Definition

We give fundamental Lemma and Definitions, which are very useful for global pruning and local filtering.

Lemma 5. *If $\exists t \in T_1$, such that $d(t, T_2) > \varepsilon$, then we have $f(T_1, T_2) > \varepsilon$ and T_1 is dissimilar to T_2, where $d(t, T_2) = \min_{q \in T_2} d(t, q)$.*

Proof. $f(T_1, T_2) = D_F(T_1, T_2)$. As described in Definition 2, it is easy to know that $D_F(T_1, T_2) \geq d(t, q)$, where q of T_2 is the matching point of t, so that we have $d(t, T_2) \leq d(t, q) \leq D_F(T_1, T_2)$. Accordingly, if $d(t, T_2) > \varepsilon$, then $f(T_1, T_2) = D_F(T_1, T_2) > \varepsilon$, i.e., T_1 is dissimilar to T_2. $\square$

Definition 6. *($SEE(MBR)$). It represents the smallest enlarged element that covers the MBR.*

Definition 7. *($Ext(MBR, \varepsilon)$). It represents the space that extends the MBR by ε.*

For example, as shown in Figure 6(a), the red rectangle is an extended MBR of a trajectory by extending ε.

C. Global Pruning

Global pruning aims to prune irrelevant index spaces. An index space is represented by an enlarged element and a position code. Thus, we first use Lemmas 6-9 to filter irrelevant enlarged elements out, then we use Lemmas 10-11 to select candidate position codes of the remaining enlarged elements.

1) Candidate Enlarged Elements: The trajectory has a spatial shape. The shapes of similar trajectories are usually similar to that of the query trajectory. Thus, extracting trajectories in too large or too small index spaces is unnecessary. Definitions 8 and 9 determine the resolutions at which an enlarged element can cover similar trajectories.

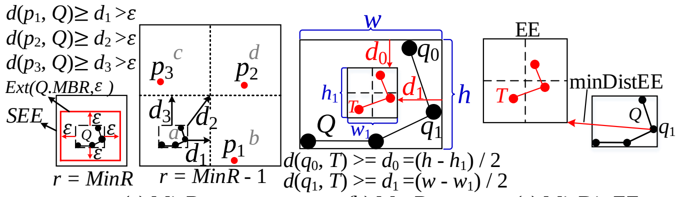
Calculated by the way of Section IV-B, we can obtain the smallest enlarged element that covers $Ext(Q.MBR, \varepsilon)$ of a query trajectory Q, i.e., $SEE(Ext(Q.MBR, \varepsilon))$.

Definition 8. *(MinR). $MinR$ is the resolution of $SEE(Ext(Q.MBR, \varepsilon))$.*

As shown in Figure 6(a), SEE is the smallest enlarged element that covers $Ext(Q.MBR, \varepsilon)$.

Lemma 6. *Any trajectory contained in an enlarged element with a resolution that is lower than MinR cannot be similar to the query trajectory Q.*

Proof. If we divide the enlarged element at a lower resolution than $MinR$ into four quads, $Ext(Q.MBR, \varepsilon)$ occurs only in one quad. As shown in Figure 3(e), a trajectory T contained in an enlarged element with a resolution lower than the maximum resolution occupies at least two quads. Thus, there is always a point p of T appears in a quad that disjoints $Ext(Q.MBR, \varepsilon)$,



(a) MinR. (b) MaxR. (c) MinDistEE.

Fig. 6. Pruning Enlarged Elements.

so that $d(p, Q) > \varepsilon$. Based on Lemma 5, $f(T, Q) \geq d(p, Q) > \varepsilon$, such that T is dissimilar to Q. □

As shown in Figure 6(a), when the resolution is $MinR-1$, $Ext(Q.MBR, \varepsilon)$ only be covered by quad-a. Points in quads b, c and d are far away from Q, so that trajectories have points in quad-a, b, or c are dissimilar to Q.

Assume that the width and height of an enlarged element are w_1 and h_1, respectively. The height and width of the MBR of a query trajectory are h and w. The maximum distance between the enlarged element and the MBR is the smallest when the enlarged element is located in the center of the MBR, as shown in Figure 6(b). We use $d_0 = (h - h_1)/2$ and $d_1 = (w - w_1)/2$ to represent the distance of height and width, respectively.

Definition 9. *(MaxR). $MaxR$ is a resolution, which satisfies that d_0 and d_1 of an enlarged element at $MaxR$-resolution are both lower than or equal to ε while d_0 or d_1 of any enlarged element at $(MaxR+1)$-resolution is greater than ε.*

Lemma 7. *$\forall$ trajectory T that is at a bigger resolution than $MaxR$ cannot be similar to the query trajectory.*

Proof. There is at least one point q of the query trajectory must be attached to the edges of its corresponding MBR. The maximum distance of the points on the edges to the trajectory covered in the enlarged element is inevitably greater than or equal to d_0 or d_1. Thus, if d_0 or d_1 is greater than ε, then $d(q, T) > \varepsilon$. Based on Lemma 5, the query trajectory cannot be similar to the trajectories of enlarged elements at resolutions that are bigger than $MaxR$. □

As shown in Figure 6(b), q_0 and q_1 are attached to the MBR of Q, such that $f(Q, T) \geq d(q_0, T) \geq d_0$ and $f(Q, T) \geq d(q_1, T) \geq d_1$. If d_0 or $d_1 > \varepsilon$, then Q is dissimilar to T.

The number of enlarged elements with resolutions between $MinR$ and $MaxR$ is still large. We further prune enlarged elements by Lemmas 8 and 9 that filter enlarged elements by their distances to the query trajectory.

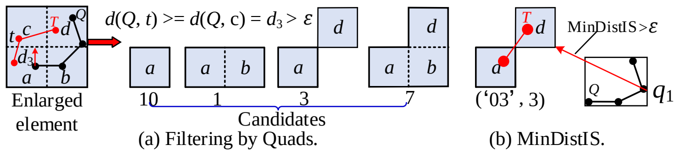
Lemma 8. *If enlarged elements do not intersect with $Ext(Q.MBR, \varepsilon)$, then they have no trajectories that are similar to Q because the distance of any point in their index spaces to the query trajectory is greater than ε.*

Definition 10. *(minDistEE). It is the minimum possible distance between a trajectory Q and an enlarged element (EE).*

$$minDistEE(Q, EE) = \max_{p \in Q.MBR \wedge Q} d(p, EE),$$

where p is point at the edge of the MBR and $d(p, EE)$ denotes the distance bewteen p and the enlarged element.

Each edge of the MBR of a trajectory contains at least one point. Thus, $minDistEE$ is the largest minimum distance



(a) Filtering by Quads. (b) MinDistIS.

Fig. 7. Pruning Index Spaces.

between the points attached to the MBR and the enlarged element. Figure 6(c) gives an example of the *minDistEE*.

Lemma 9. *If $minDistEE(Q, EE) > \varepsilon$, then $\forall T \in EE$ is dissimilar to Q, where EE is an enlarged element.*

Proof. $\forall T \in EE$, $\forall t \in T$, we have $minDistEE(Q, EE) \leq d(t, Q) \leq f(Q, T)$. Thus, if $minDistEE(Q, EE) > \varepsilon$, then $f(Q, T) > \varepsilon$, Q is dissimilar to T. □

As shown in Figure 6(c), $minDistEE(Q, EE) = d(q_1, EE)$. If $\varepsilon < minDistEE(Q, EE)$, then$d(q_1, T) \leq f(Q, T)$, i.e., Q is dissimilar to T.

2) Candidate Position Codes: The enlarged element is still too large to represent a trajectory. Thus, we proposed a concept of the position code in Section IV-B, which uses a combination of four sub-quads as an index space, as shown in Figure 3(e). In this part, we will introduce the pruning strategies to filter out uncorrelated index spaces of candidate enlarged elements , which can further narrow the search range of query processing.

Lemma 10. *Assuming that, sq is a sub-quad, which is used to form an index space. If its distance to the query trajectory $d(sq, Q) > \varepsilon$, then the index space can be pruned.*

Proof. $\forall T$ covered by an index space containing sq has at least one point t in sq, such that $d(t, Q) \geq d(sq, Q)$. Thus, if $d(sq, Q) > \varepsilon$, then $f(T, Q) \geq d(sq, Q) > \varepsilon$, so this index space can be pruned. □

In Figure 7(a), the distance from quad-c to Q is $d(c, Q)$. $t \in c$ and $d(t, Q) \geq d(c, Q) > \varepsilon$, so T is dissimilar to Q.

After pruning index spaces containing quads that are far away from Q, we further eliminate remaining index spaces (*IS*) by their distances to Q, i.e., $minDistIS$.

Definition 11. *(minDistIS).*

$$minDistIS(Q, IS) = \max_{p \in Q.MBR \wedge Q} d(q, IS).$$

Lemma 11. *If $minDistIS(Q, IS) > \varepsilon$, then the index space of IS can be pruned.*

Proof. $\forall T \in IS$, $\forall t \in T$, we have $minDistIS(Q, IS) \leq d(t, Q) \leq f(Q, T)$. Thus, if $minDistIS(Q, IS) > \varepsilon$, then $f(Q, T) > \varepsilon$, i.e., Q is dissimilar to any trajectory in IS. □

Figure 7(b) shows an example. $f(Q, T) \geq d(q_1, (\text{'03'}, 3)) = minDistIS(Q, (\text{'03'}, 3)) > \varepsilon$, so T is dissimilar to Q.

D. Local Filtering

After filtering out unnecessary index spaces by the Lemmas of Section V-C, we extract trajectories indexed by the remaining index spaces in each region of the database. Local filtering helps to eliminate dissimilar trajectories in each region with minimal computational cost.

Algorithm 1: Global Pruning: *GP*(Q, ε)

```
Input: A query trajectory: $Q$, a threshold $\varepsilon$.
Output: Index values: $values$.
1  $remaining = new\ \ FIFO\_queue()$; $value = \emptyset$;
2  $remaining.insert(RootElements.children)$;
3  while $remaining \neq \emptyset$ do
4      $e = remaining.pop$;
5      if MinR $\leq$ e.resolution // Lemma 6 then
6          if $minDistEE(Q, e) \leq \varepsilon$ //Lemma 9 then
7              foreach $ind\_s \in e.index\_space$ do
                   // Lemma 10 and Lemma 11
8                  if $\forall\ sq \in ind\_s$ and $d(sq, Q) \leq \varepsilon$ then
9                      if $minDistIS(Q, ind\_s) \leq \varepsilon$ then
10                         $values.add(ind\_s.index\_value)$;
11     if e.resolution < MaxR //Lemma 7 then
12         foreach $child \in e.children$ do
13             if child intersects $Ext(Q.MB, \varepsilon)$) then
14                 $remaining.insert(child)$; //Lemma 8
15 return $values$;
```

Lemma 12. *If Q is similar to T, the distance of the start point of Q and T must be less than or equal to ε, and the distance of end point of Q and T must be less than or equal to ε.*

Proof. Based on Definition 2, $f(Q,T) = D_F(Q^n, T^m) \geq d(q_1, t_1)$ and $D_F(Q^n, T^m) \geq d(q_n, t_m)$. If T is similar to Q, then $f(Q,T) \leq \varepsilon$, $d(q_1, t_1)$ and $d(q_n, t_m)$ must be $\leq \varepsilon$. □

It is time-consuming to calculate the similarity of two trajectories because all points of Q and T must be visited. Thus, we use DP features to represent a trajectory, which efficiently helps query processing filter dissimilar trajectories.

Note again that we calculate the DP features of a trajectory before storing, so we do not need to calculate DP features of extracted trajectories again in the query processing.

We use $T.P = (p_1, ..., p_k)$ to denote the representative points of a trajectory T, and $T.B = (bbox_1, ..., bbox_{k-1})$ to represent the representative bboxes of T, as shown in Figure 5.

Lemma 13. *If $\exists p \in T_1.P$, such that $d(p, T_2.B) > \varepsilon$, then we have $f(T_1, T_2) > \varepsilon$.*

Proof. Because $d(p, T_2) \geq d(p, T_2.B)$, if $d(p, T_2.B) > \varepsilon$, then $d(p, T_2) > \varepsilon$. Based on Lemma 5, $f(T_1, T_2) > \varepsilon$. □

As shown in Figure 5(b), the distance from p_3 of $T.P$ to $Q.B$ is greater than ε, so that T is dissimilar to Q.

Each $bbox$ has four edges, the distance between an $bbox$ and $T.B$ of a trajectory is $d(bbox, T.B) = \max_{edge \in bbox} d(edge, T.B)$.

Lemma 14. *If $\exists bbox \in T_1.B$, such that $d(bbox, T_2.B) > \varepsilon$, then we have $f(T_1, T_2) > \varepsilon$.*

Proof. For $\forall bbox \in T_1.B$, assuming that $q \in edge_i$, q is a point of T_1 and $d(edge_i, T_2.B)$ is the maximum distance than other edges of the $bbox$, we have $d(q, T_2) \geq d(edge_i, T_2) \geq d(bbox, T_2) \geq d(bbox, T_2.B)$. Thus, if $d(bbox, T_2.B) > \varepsilon$, then $d(q, T_2) > \varepsilon$, so $f(T_1, T_2) > \varepsilon$. □

As shown in Figure 5(b), the distance from the $bbox_2$ of $Q.B$ to $T.B$ is greater than ε, so that Q is dissimilar to T.

E. Processing

Based on the lemmas and definitions, we implement two similarity searches: *Threshold Similarity Search* and *Top-k Similarity Search*. We search from the root of the XZ* index and filter irrelevant index spaces by the global pruning, then execute local filtering for each extracted trajectory. Note again that the performance of our Lemmas increases step by step, but the computation also goes up. Thus, we execute lemmas of pruning strategies from simple to complex. Besides, for *Top-k Similarity Search*, we use a best-first (BF) algorithm to find similar trajectories from near to far. Additionally, the complexity of query is constant-level, because the query is transformed as a set of key ranges using the fixed spatial partition and encoding method, then retrieves the corresponding contents from a KV store. As a result, the algorithm complexity does not change with the increase of the number of trajectories.

Algorithm 2: Local Filtering: *LF*(Q, ε)

```
Input: Query trajectory: $Q$, threshold $\varepsilon$.
1  Function filter (T)
2  if $d(T.start\_point, Q.start\_point) \leq \varepsilon$ //Lemma 12 then
3      if $d(T.end\_point, Q.end\_point) \leq \varepsilon$ then
4          if DP features of T and Q do not satisfy
5          Lemmas from 13 to 14 and $f(Q,T) \leq \varepsilon$ then
6              return false;
7  return true;
8  End Function
```

Algorithm 3: Threshold similarity search: *sim*(Q, ε)

```
Input: A query trajectory: $Q$, a threshold $\varepsilon$.
Output: Trajectories: $results$.
1  $index\_values = GP(Q, \varepsilon)$; $scanner = new\ Scanner()$;
2  $scanner.addScanRange(index\_values)$;
3  $scanner.addFilter(LF(Q, \varepsilon))$;
4  return $scanner.execute().results$;
```

Threshold Similarity Search. Given a query trajectory Q and a threshold ε, we first generate candidate index values based on global pruning, as shown in Algorithm 1. The algorithm starts at the root, the resolution of each successive child gets larger. Lines 5 and 11 guarantee that only the enlarged elements at resolutions between $maxR$ and $minR$ would be candidate enlarged elements. In lines 6 and 13, based on Lemmas 8 and 9, we prune dissimilar enlarged elements. In lines 7-10, we prune irrelevant index spaces by Lemmas 10 and 11. Second, we extract trajectories indexed with the candidate index values on each region and filter out dissimilar trajectories by executing local filtering (Algorithm 2). After that, we refine the remaining trajectories by calculating their similarities with Q. Algorithm 3 gives processing of *Threshold Similarity Search*, where line 1 generates candidate index values and creates a scanner for extracting trajectories on the database, line 3 adds local filtering to the scanner, and line 4 visits trajectories and executes local filtering in each region.

Top-*k* Similarity Search. According to the definition of similarity methods, e.g., Fre̒chet and Hausdorff methods, the more similar of two trajectories, the smaller the distance. Therefore, as shown in Algorithm 4, our *top-k trajectory similarity search* uses a Best-First algorithm to find similar trajectories from near to far. $minDistEE(Q, e)$ is the distance between a trajectory Q and an enlarged element e (Definition 10), $minDistIS(Q, i)$ is the distance between a trajectory Q and an index space i (Definition 11). *EQ* and *IQ* are the priority queues for enlarged elements and index spaces. We always pop the nearest enlarged element from the priority queue of EQ (Line 5) and add index spaces into IQ. If the $minDistEE$ of remaining enlarged elements is greater than the $minDistIS$ of IQ, in Line 14, we search trajectories that are indexed by

Algorithm 4: Top-k similarity search: *kNN*(Q, k)

```
   Input: A query trajectory: $Q$, an integer $k$.
   Output: $k$ priority trajectories: $results$.
1  $\varepsilon = MAX\_VALUE$; $result = new\ priority\_queue(k)$;
2  $EQ = \emptyset$; $IQ = \emptyset$;
3  $EQ.push(root, minDistEE(Q, root))$;
4  while $EQ \neq \emptyset$ do
5      $e \leftarrow EQ.pop()$;
6      if $IQ \neq \emptyset$ and $e.minDistEE > IQ.minDistIS$ then
7          while $IQ \neq \emptyset$ do
8              if $IQ.minDistIS > e.minDistEE$ then
9                  . break; //go to Line 16;
10             $i \leftarrow IQ.pop()$;
11             if $i.minDistIS > \varepsilon$ then
12                 return $results$;
13             if !$filter(i.indexSpace, \varepsilon)$//Lemmas 10-11 then
14                 $results.add(search(i.indexValue(), LF(Q, \varepsilon), k))$;
                   if $results.size() == k$ then
15                     $\varepsilon = results.maxDistance$;
16     if !$filter(e, \varepsilon)$ //Lemmas 6-9 then
17         $IQ.add(e.indexSpaces, minDistIS(Q, e.indexSpaces))$
           foreach $child \in e.children$ do
18             $EQ.add(child, minDistEE(Q, child))$;
19 return $results$;
```

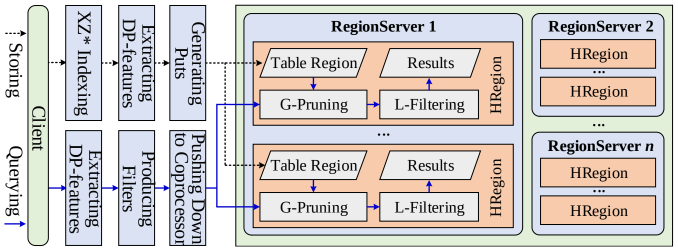


Fig. 8. Instantiation of *TraSS*.

the optimal index value (Lines 8-13) of IQ from the database and execute the local filtering $LF(Q, \varepsilon)$ to eliminate dissimilar trajectories. If the number of extracted trajectories reaches k, we update the current maximum similarity distance ε from the *results*, and use ε to further prune enlarged elements and index spaces by Lemmas 6-11. If the $minDistIS$ of IQ is greater than $EQ.minDistEE$ (Lines 8-9), we continue to search EQ (Lines 16-18) for obtaining more near index spaces. If the $minDistIS > \varepsilon$ (Lines 11-12), we finish the *top-k similarity search*, because the similarity distance of trajectories in the next iteration would always be greater than ε, and they cannot be similar to the query trajectory.

VI. Evaluation

INSTANTIATION. As shown in Figure 8, we implement $TraSS$ based on HBase, a popular key-value store. Firstly, trajectories are indexed using appropriate index spaces. Then, we use Douglas-Peucker (DP) algorithm to pre-calculate DP features of trajectories. After that, we convert trajectories to $puts$ and insert all $puts$ into HBase table regions. When executing the similarity search, we first calculate the DP features of the query trajectory. Then, we push down global pruning (*G-Pruning*) and local filtering (*L-Filtering*) into the coprocessor of HBase. Finally, we return the query results to the client. For more details about $TraSS$, please refer to [24].

Baselines. We evaluate our work with other state-of-art works, i.e., $TraSS$ (our work), DFT [19] (VLDB 2017), $DITA$ [20] (SIGMOD 2018), $JUST$ [9] (ICDE 2020), and *REPOSE* [26] (ICDE 2021, which only support top-k similarty search).

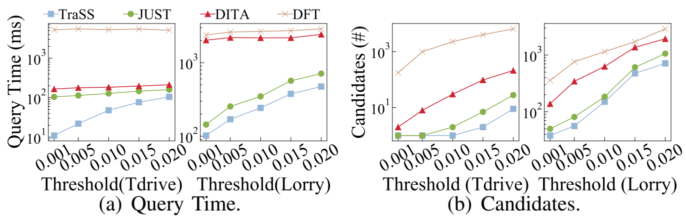


Fig. 9. Efficiency of Threshold Similarity Search.

Datasets. We use two real and five synthetic trajectory datasets to evaluate the efficiency of *TraSS*: (1) **TDrive** [2], which contains 321,387 taxi trajectories of Beijing over two weeks, China (752MB); (2) **Lorry**, which contains 4,394,397 JD logistic lorry trajectories of Guangzhou, China (136GB); and (3) **Synthetic**, to verify the scalability of $TraSS$, we use five synthetic datasets generated by copying i times of the Lorry dataset. Figure 12 shows the distribution of TDrive and Lorry.

Setting. We randomly pick 400 query trajectories from each dataset, respectively, and take the median processing time as the final results. We vary the threshold ε from 0.001 to 0.02 to evaluate the efficiency of *threshold similarity search* in Section VI-A, and vary the size of k from 50 to 250 to evaluate the efficiency of *top-k similarity search* in Section VI-B. In Section VI-C, we evaluate the effect of our pruning strategies. In Section VI-D, we will evaluate the performance of the XZ* index. Next, in Section VI-E, we use five synthetic datasets to evaluate the scalability. The entire index space of the XZ* index covers the earth. The default maximum resolution is 16, and the predefined distance for DP features is 0.01 (mostly used in existing works). All experiments are conducted on a cluster with five nodes, and each node is equipped with an 8-core CPU, 32GB RAM, and 1T disk.

A. Threshold Similarity Search

We evaluate the performance of *threshold similarity search* by comparing the query time and the candidates after pruning. Figure 9(a) shows the query time and Figure 9(b) displays the number of candidates (remaining trajectories after executing the pruning strategies, not the final answer) of different solutions. We have the following observations: (1) with the threshold increase, it takes more query time as a large threshold leads to more visited trajectories; (2) our solution outperforms other solutions, even by one or two orders of magnitude. As shown in Figure 9(a), when $\varepsilon = 0.001$ on T-Drive, *DFT* takes 5,241 milliseconds and *DITA* takes 166 milliseconds while $TraSS$ takes 11 milliseconds. This is because *TraSS* proposes an efficient index structure, which is much better than the R-tree used in *DFT* and a trie-like index used in *DITA*. Based on XZ* index, the query processing of $TraSS$ can efficiently prune dissimilar trajectories, and the scale of candidates after pruning is much smaller than other solutions, as shown in Figure 9(b). DFT uses the index to obtain a bitmap of candidate trajectories, collects the bitmap at the master node, and then extracts data by bitmap to verify the similarity, causing much shuffle and computation. $DITA$ uses *MBR coverage filtering* to filter the search range. However, a trajectory may appear in a small area of its representative MBR. Thus, *MBR coverage filtering* prunes fewer trajectories

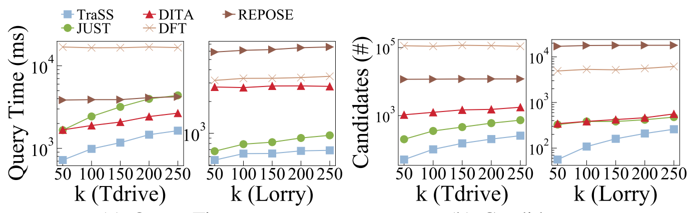

(a) Query Time. (b) Candidates.

Fig. 10. Efficiency of Top-k Trajectory Similarity Search.

than XZ*; (3) Although $JUST$ applies many pruning strategies to avoid much computation, which is more efficient than DFT and $DITA$. However, the XZ-Ordering used in $JUST$ is not finer than XZ*, so $JUST$ needs to scan more trajectories than $TraSS$. Moreover, all pruning strategies of $TraSS$ are pushed down to the coprocessor of HBase while $JUST$ does not. Thus, $TraSS$ is better than $JUST$.

B. Tok-k Similarity Search

As presented in Figure 10, we conduct *top-k trajectory similarity search* experiments to study the effect of the number k. It can observe in Figure 10(a) that the performance of our solution is the best compared with others. As shown in Figure 10(b), DFT and $REPOSE$ retain many candidate trajectories, resulting in they are much slower than our solution. The reason is that DFT randomly selects $c*k$ (the default value of c is 5) trajectories from the partitions that intersect the query trajectory. Then it obtains a threshold to verify trajectories covered by the threshold. Finally, DFT takes top-k trajectories from the candidates. However, most trajectories in the T-Drive dataset have a large MBR, which causes many partitions intersecting the query trajectory. Thus, DFT always gets a large threshold that covers massive candidates. Especially, $REPOSE$ builds the RP-Trie (reference point trie) index on pivot trajectories, and the selection of pivot trajectories has a substantial effect on the pruning performance. However, the spatial span of the lorry dataset covers china, resulting in the RP-Trie needing to build a large structure, which has greatly affected its pruning performance. Besides, $DITA$ performs better than DFT on the T-Drive dataset, but it is not good on the Lorry dataset because $DITA$ builds a large index for the large Lorry dataset, and each node of the index contains many trajectories, which causes much time to search candidates. Both $DITA$ and $JUST$ are slower than $TraSS$. The reason is that they visit more trajectories to obtain candidates and need more time to calculate the final answer from the remaining candidates than $TraSS$. Benefit from XZ* index, $TraSS$ can generate fine-grained scan ranges to cover candidates, which helps $TraSS$ to obtain candidates by visiting only a few trajectories, so $TraSS$ spends less time to execute top-k similarity search.

C. Effect of Pruning Strategies.

We evaluate the effect of pruning strategies by three metrics: pruning time; retrieved trajectories, which reflects the filtration capacity of global pruning; precision, which is the ratio of final answers to candidate trajectories. As shown in Figure 11(a), *TraSS* takes the lowest pruning time than others. Because (1) global pruning consumes very little to generate index values for retrieval trajectories, which does not require much calculation; (2) the local filtering must check trajectories retrieved from the database. After global pruning, *TraSS* retrieves fewer trajectories than others, as shown in Figure 11(b). Thus, $TraSS$ spends less time in local filtering. Besides, as displayed in Figure 11(c), benefiting from the filtration capacity of local filtering, $TraSS$ obtains fewer false hits than other solutions.

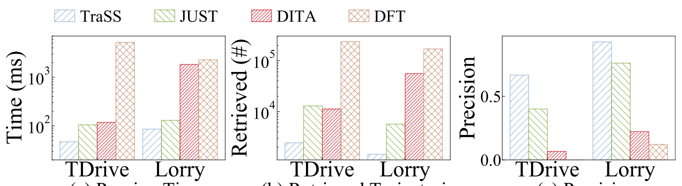

(a) Pruning Time. (b) Retrieved Trajectories. (c) Precision.

Fig. 11. Pruning Strategies of Different Solutions ($\varepsilon = 0.01$).

D. Effect of XZ* Index

Distribution. The XZ* index uses index spaces of different resolutions to represent trajectories. As shown in Figure 12(a), most trajectories are distributed at resolutions from 10 to 16 because the driving ranges of many trajectories are from 0.5km (corresponding resolution is about 16) to 78km (corresponding resolution is about 10). There is a peak at 19-resolution because many taxis stay at interest places to wait for customers, resulting in their trajectories only having points with the same latitude and longitude so that they are always covered by the maximum resolution. As shown in Figure 12(b), we use position codes to subtly represent the trajectory, which improves the granularity of the index space.

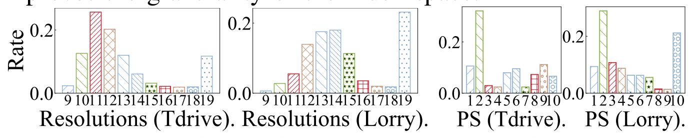
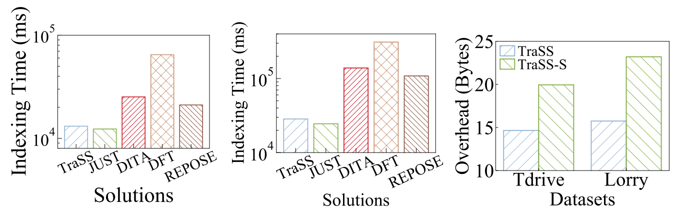

(a) Rate of Resolutions. (b) Rate of Position Codes.

Fig. 12. Distribution of Trajectories.

Overhead. Figure 13 presents the indexing time of different solutions. $TraSS$ and $JUST$ both adopt the static index structure, which does not spend time adjusting its structure. In contrast, DFT, $DITA$ and $REPOSE$ use dynamic index structures, which takes much time to adapt to the dataset by adjusting the index structure. Thus, the indexing time of DFT, $DITA$ and $REPOSE$ is higher than $TraSS$ and $JUST$, especially in the large dataset, i.e., Lorry. Figure 13(c) shows the average storage overhead of a rowkey. $TraSS$ uses the integer encoding, *TraSS-S* uses the string encoding to generate keys, which requires more overhead than integers. Our encoding is more space-efficient than the string encoding. Especially, for Tdrive and Lorry, our encoding reduces 32% and 27% of overhead on rowkeys, respectively.

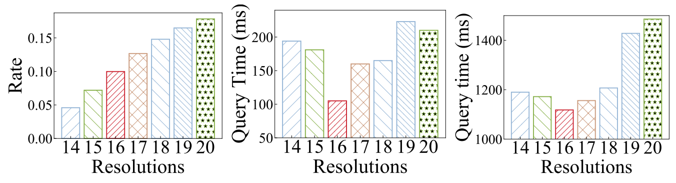
Varying of Resolutions. In this section, we evaluate the performance of different maximum resolutions. We use selectivity to describe the representative of the XZ* index. Selectivity is the ratio of index values to that of the row keys. It represents the degree of difference of the data in the index column. As shown in Figure 14(a) and Figure 15(a), the selectivity of 14-resolution is lower than others, which causes many trajectories



(a) Indexing (Tdrive). (b) Indexing (Lorry). (c) Storage of Rowkey.

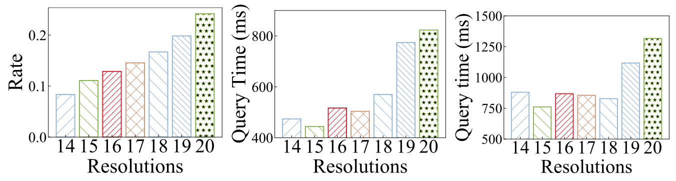
Fig. 13. Overhead of Different Solutions.

to be selected during the query processing. However, if the maximum resolution is large, the query processing requires more time to generate the query ranges because the higher the selectivity, the more scattered the index values. In addition, when executing similarity queries, the greater the resolution, the fewer number of trajectories obtained in each iteration, resulting in the number of iterations increased, thereby increasing the query time, as shown in Figure 14(b)(c) and Figure 15(b)(c).



(a) Selectivity. (b) Threshold Search. (c) Top-k Search.

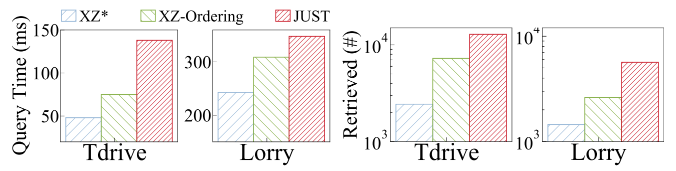
Fig. 14. Query Time of Different Resolutions (Tdrive).



(a) Selectivity. (b) Threshold Search. (c) Top-k Search.

Fig. 15. Query Time of Different Resolutions (Lorry).

Comparing with XZ-Ordering. We also integrate the global pruning and local filtering into XZ-Ordering. As shown in Figure 16(a), XZ* index outperforms XZ-Ordering. Besides, XZ-Ordering equipped with our pruning strategies has a better performance than that of JUST. As shown in Figure 16(b), XZ* index can avoid retrieving many invalid trajectories than XZ-Ordering, so it outperforms the XZ-Ordering. Actually, on Tdrive and Lorry dataset, we reduce 66.4% and 44.7% of retrieved trajectories than XZ-Ordering, respectively.

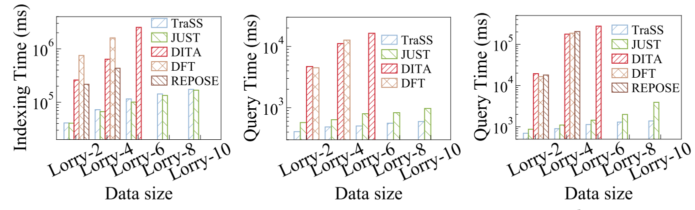


(a) Query Time. (b) Retrieved trajectories.

Fig. 16. Comparing with the XZ-Ordering index ($\varepsilon = 0.01$).

E. Scalability

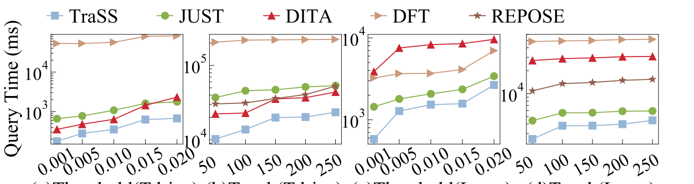
Data size. To evaluate the scalability of $TraSS$, we use five synthetic datasets generated by copying i times of the Lorry dataset. As shown in Figure 17(a), the indexing time linearly grows because we must index more trajectories. As both *DITA*, *DFT* and *REPOSE* rely heavily on memory, they are unable to support a large amount of data in our experimental environment. As shown in Figure 17(b)(c), as the increase of data size, the query time of threshold and top-k similarity search is growing because we must visit more trajectories. Excitingly, the query time of $TraSS$ is still small and the advantage of $TraSS$ becomes more obvious.



(a) Indexing Time. (b) Threshold Search. (c) Top-k Search.

Fig. 17. Effect of The Data Size.

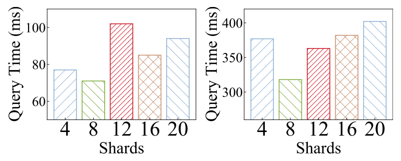
Tail Latency. We take the 99th percentile of the query result to show the tail latency. As shwon in Figure 18, the tail latency of $TraSS$ is lower than others.



(a)Threshold(Tdrive). (b)Top-k(Tdrive). (c)Threshold(Lorry). (d)Top-k(Lorry).

Fig. 18. Tail Latency

Shards. In Section IV-E, we introduced the design of schema for managing massive trajectories in HBase, where the *shards* is a hash number that can avoid data skew problem. As shown in Figure 19, we can observe that $shards = 8$ performs better than others because if the $shards$ is too small, many similar trajectories may be stored in the same region, which results in the data skew problem. In contrast, if the $shards$ is too large, the similar trajectories may be scattered in many regions, resulting in a high communication cost. The $shards = 8$ is an agreeable value because we only have five nodes.



(a) Shards on Tdrive. (b) Shards on Lorry.

Fig. 19. Effect of Shards ($\varepsilon = 0.015$).

VII. OTHER MEASURES

We extend our solution to Haudorff and DTW. We first give definitions of Haudorff and DTW, then discuss how pruning strategies proposed in V can be used in Haudorff and DTW.

A. Extension on Hausdorff

Given two trajectories $Q = (q_1, ..., q_n)$ and $T = (t_1, ..., t_m)$, the definition of Hausdorff is as follows:

Definition 12. *(Hausdorff).*

$$D_H(Q,T) = \max\{\max_{i=1}^{n} d(q_i,T),\ \max_{j=1}^{m} d(t_j,Q)\},$$

where $d(q_i,T) = \min_{t\in T} d(q_i,t)$ *and* $d(t_j,Q) = \min_{q\in Q} d(t_j,q)$.

Pruning strategies. The Hausdorff distance also satisfies Lemma 5, i.e., if $\exists q \in Q$, such that $d(q,T) > \varepsilon$, then we have $D_H(Q,T) > \varepsilon$.

Proof. As described in Definition 12, it is easy to know that $D_H(Q,T) \geq \max_{i=1}^{n} d(q_i,T) \geq d(q,T)$, q is a point of Q. Thus, if $d(q,T) > \varepsilon$, then $D_H(Q,T) > \varepsilon$. □

Except for Lemma 12, all other Lemmas proposed in Sections V-C and V-D are based on Lemma 5. Therefore, except for Lemma 12 is banned in Algorithm 2, the query processing of Hausdorff is the same as that of the Fréchet.

B. Extension on DTW

Definition 13. *(DTW). The definition of DTW is as follows:*

$$D_D(Q^n, T^m) = \begin{cases} \sum_{i=1}^{m} d(q_1, t_i) & \text{if } n = 1 \\ \sum_{i=1}^{n} d(q_i, t_1) & \text{if } m = 1 \\ d(q_n, t_m) + \min\{D_D(Q^{n-1}, T^m), \\ D_D(Q^n, T^{m-1}), D_D(Q^{n-1}, T^{m-1})\} \end{cases},$$

Pruning strategies. The DTW distance satisfies Lemma 5 and Lemma 12. Therefore, both Lemmas proposed in Sections V-C and V-D can be directly used in DTW.

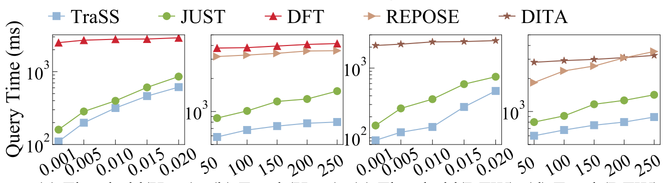
Proof. There are three cases of the length of Q and T:
1) If $n = 1$, we have $D_D(Q,T) \geq d(q_1,t_i) \geq d(q_1,T)$;
2) If $m = 1$, we have $D_D(Q,T) \geq d(q_i,t_1) = d(q_i,T)$;
3) If $n, m > 1$, we have $D_D(Q,T) \geq d(q_i,t_j) \geq d(q_i,T)$, where q_i is a point of Q, and $t_j \in T$ is a matching point of q.

Thus, $\forall q \in Q$, we have that $D_D(Q,T) \geq d(q,T)$. Therefore, if $\exists q \in Q$, such that $d(q,T) > \varepsilon$, then we have $D_D(Q,T) > \varepsilon$, i.e., D_D satisfies Lemma 5. Besides, $D_D(Q,T) \geq d(q_1,t_1)$ and $D_D(Q,T) \geq d(q_n,t_m)$, so that $D_D(Q,T)$ also satisfies Lemma 12. □

The query processing of DTW is the same as that of Fréchet.

C. Efficiency.

DITA does not support the Hausdorff distance, *DFT* does not support the DTW, and *REPOSE* supports only top-k similar search. Figure 20 shows the efficiency of our framework. We can intuitively observe that in the Hausdorff and DTW metrics $TraSS$ is more outstanding than others.



(a) Threshold(Haus). (b) Top-k(Haus). (c) Threshold(DTW). (d) Top-k(DTW).

Fig. 20. Extension on Other Similarity Measures.

VIII. Related Work

A. Trajectory Similarity Search

Trajectory data management systems with trajectory similarity search can be divided into three categories:

Standalone-based Search. For example, Elias et al. [27] propose an efficient method using an R-tree-like index to support top-*k* trajectory similarity search. Ta et al. [28] propose a new bi-directional mapping similarity (BDS) to address the trajectory sample points not aligned problem. Torch [29] designs an inverted index LEVI to support road network trajectory top-*k* trajectory similarity search. Due to the limited resources of a single machine, these standalone solutions can hardly process prohibitively large trajectory data.

Hadoop/Spark-based Search. Built on ST-Hadoop [30], [31], Summit [32] provides various trajectory queries, including top-*k* trajectory similarity search. As Hadoop visits disks multiple times, it may face an efficiency problem. To this end, a lot of Spark-based trajectory data management systems have emerged [19], [20], [26], [33], [34]. DFT [19] implements a distributed query framework to process top-*k* trajectory similarity search. DITA [20] designs a trie-like index in Spark on pivot points to support trajectory similarity join and threshold similarity query efficiently. UlTraMan [33] extends Spark by seamlessly integrating a key-value store Chronicle Map, which enables top-*k* trajectory similarity search. For removing noise points, DISON [34] proposes a distributed in-memory trajectory similarity search and join framework on the road network. Aimed at reducing computing resource waste, REPOSE [26] proposes a framework for processing top-*k* trajectory similarity search on Spark. These Spark-based systems build huge indexes and load all trajectory data in memory. Thus their scalability is limited.

NoSQL and Key-value based Search. NoSQL-based trajectory data management systems store trajectories in key-value data stores [5]–[9]. They turn spatio-temporal information of a trajectory into a one-dimensional key. Hence they are easy to scale up. THBase [5] presents a coprocessor-based scheme for big trajectory data management in HBase and can support top-*k* trajectory similarity search efficiently. TrajMesa [8], [35] and JUST [9] build XZ2 indexes based on GeoMesa [21] to support trajectory similarity query and top-*k* trajectory similarity query. However, most NoSQL-based systems build indexes based on trajectory MBRs, resulting in they would scan many unnecessary data when answering trajectory queries.

IX. Conclusion

This paper proposes $TraSS$, an efficient trajectory similarity search framework in the key-value data store. We utilize XZ* index to represent a trajectory in a key-value data store elaborately and provide an efficient encoding for XZ* to store and query trajectories in key-value data stores, reducing the row key overhead up to 32%. We design efficient query processing, which can execute the trajectory similarity searches without calculating the similarity of the query trajectory to many unnecessary trajectories, reducing the I/O overhead by up to 66.4%. Extensive experiments show that $TraSS$ outperforms state-of-the-art distributed solutions. $TraSS$ supports many trajectory similarity measures such as Fréchet, Hausdorff, and DTW. Besides, XZ* index supports spatial range query [24]. Interesting future work includes how to support other metrics and other spatial queries.

X. Acknowledgment

This paper was supported by the National Key R&D Program of China (2019YFB2103201).

REFERENCES

[1] S. Ruan, C. Long, J. Bao, C. Li, Z. Yu, R. Li, Y. Liang, T. He, and Y. Zheng, "Learning to generate maps from trajectories," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 890–897.

[2] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 316–324.

[3] H. He, R. Li, R. Wang, J. Bao, Y. Zheng, and T. Li, "Efficient suspected infected crowds detection based on spatio-temporal trajectories," *arXiv preprint arXiv:2004.06653*, 2020.

[4] Apache, "Hbase," https://hbase.apache.org/, 2021.

[5] J. Qin, L. Ma, and J. Niu, "Thbase: A coprocessor-based scheme for big trajectory data management," *Future Internet*, vol. 11, no. 1, p. 10, 2019.

[6] J. Bao, R. Li, X. Yi, and Y. Zheng, "Managing massive trajectories on the cloud," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016, pp. 1–10.

[7] R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2017, pp. 1–4.

[8] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 2002–2005.

[9] R. Li, H. He, R. Wang, Y. Huang, J. Liu, S. Ruan, T. He, J. Bao, and Y. Zheng, "Just: Jd urban spatio-temporal data engine," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1558–1569.

[10] H. He, R. Li, J. Bao, T. Li, and Y. Zheng, "Just-traj: A distributed and holistic trajectory data management system," in *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, 2021, pp. 403–406.

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[12] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.

[13] D. Pfoser, C. S. Jensen, Y. Theodoridis *et al.*, "Novel approaches to the indexing of moving object trajectories." in *VLDB*, 2000, pp. 395–406.

[14] X. Xu, J. Han, and W. Lu, "Rt-tree: An improved r-tree indexing structure for temporal spatial databases [c]," in *The International Symposium on Spatial Data Handling (SDH). Zurich*, 1990, pp. 1040–1049.

[15] Y. Tao and D. Papadias, "Efficient historical r-trees," in *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*. IEEE, 2001, pp. 223–232.

[16] Y. Tao, D. Papadias, and J. Sun, "The tpr*-tree: An optimized spatio-temporal access method for predictive queries," in *Proceedings 2003 VLDB conference*. Elsevier, 2003, pp. 790–801.

[17] C. BÖxhm, G. Klump, and H.-P. Kriegel, "Xz-ordering: A space-filling curve for objects with spatial extension," in *International Symposium on Spatial Databases*. Springer, 1999, pp. 75–90.

[18] J. Yu and M. Sarwat, "Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 385–396, 2016.

[19] D. Xie, F. Li, and J. M. Phillips, "Distributed trajectory similarity search," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1478–1489, 2017.

[20] Z. Shang, G. Li, and Z. Bao, "Dita: Distributed in-memory trajectory analytics," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 725–740.

[21] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "Geomesa: a distributed architecture for spatio-temporal fusion," in *Geospatial informatics, fusion, and motion video analytics V*, vol. 9473. International Society for Optics and Photonics, 2015, p. 94730F.

[22] R. Fecher and M. A. Whitby, "Optimizing spatiotemporal analysis using multidimensional indexing with geowave," in *Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings*, vol. 17, no. 1, 2017, p. 12.

[23] Y. Zheng, "Trajectory data mining: an overview," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, pp. 1–41, 2015.

[24] H. He, "TraSS," https://github.com/huajunge/k-sim, 2021.

[25] K. Toohey and M. Duckham, "Trajectory similarity measures," *Sigspatial Special*, vol. 7, no. 1, pp. 43–50, 2015.

[26] B. Zheng, L. Weng, X. Zhao, K. Zeng, X. Zhou, and C. S. Jensen, "Repose: Distributed top-k trajectory similarity search with local reference point tries," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 708–719.

[27] E. Frentzos, K. Gratsias, and Y. Theodoridis, "Index-based most similar trajectory search," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 816–825.

[28] N. Ta, G. Li, Y. Xie, C. Li, S. Hao, and J. Feng, "Signature-based trajectory similarity join," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 870–883, 2017.

[29] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin, "Torch: A search engine for trajectory data," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2018, pp. 535–544.

[30] L. Alarabi, M. F. Mokbel, and M. Musleh, "St-hadoop: A mapreduce framework for spatio-temporal data," *GeoInformatica*, vol. 22, no. 4, pp. 785–813, 2018.

[31] L. Alarabi and M. F. Mokbel, "A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1961–1964, 2017.

[32] L. Alarabi, "Summit: a scalable system for massive trajectory data management," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2018, pp. 612–613.

[33] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "Ultraman: a unified platform for big trajectory data management and analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 7, pp. 787–799, 2018.

[34] H. Yuan and G. Li, "Distributed in-memory trajectory similarity search and join on road network," in *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE, 2019, pp. 1262–1273.

[35] R. Li, H. He, R. Wang, S. Ruan, T. He, J. Bao, J. Zhang, L. Hong, and Y. Zheng, "Trajmesa: A distributed nosql-based trajectory data management system," *IEEE Transactions on Knowledge and Data Engineering*, 2021.