TMan: A High-Performance Trajectory Data Management System Based on Key-value Stores

Huajun He[1,2,3], Zihang Xu[1], Ruiyuan Li[4], Jie Bao[2,3], Tianrui Li[1*], Yu Zheng[1,2,3*]
[1]*School of Computing and Artificial Intelligence, Southwest Jiaotong University, China*
[2]*JD iCity, JD Technology, Beijing, China* [3] *JD Intelligent Cities Research, China* [4]*Chongqing University, China*
hehuajun@my.swjtu.edu.cn;{xuzihang829,ruiyuan.li,msyuzheng}@outlook.com;baojie3@jd.com;trli@swjtu.edu.cn

***Abstract*—The effective management of trajectory data heavily relies on the utilization of fundamental spatio-temporal queries. The surge in trajectory data, with its dynamic spatio-temporal properties, poses notable management challenges. Existing systems are inadequate in providing fine-grained trajectory representations and efficient architecture for processing queries, leading to significant computational overhead. This paper introduces TMan to address these challenges. First, TMan presents two innovative index structures that precisely capture the spatio-temporal characteristics of trajectory data. Compared to the state-of-the-art indexes, our indexes for temporal range and spatial range queries can reduce the number of retrievals by up to 77% and 83%, respectively. Next, TMan devises concise and effective encoding methods for these indexes. Leveraging these indexes, TMan provides a distributed storage structure and an index caching mechanism for efficiently managing trajectories in key-value data stores. Moreover, TMan introduces a parallel query processing approach incorporating a push-down strategy to enhance the efficiency of fundamental queries. Extensive experimental results demonstrate that TMan's index structures and architecture outperform the baselines.**

I. Introduction

Trajectory data has seen a surge in applications due to the extensive use of GPS and other location-based technologies. For example, more than 1TB of trajectory logs are generated daily by 60,000 JD Logistics couriers [1]. Consequently, the urgent need for efficient systems to manage large-scale trajectory data is evident. Key-value data stores are commonly used for storing vast amounts of data due to their scalability and ability to quickly lookup data within extensive datasets. However, it is a challenging task for trajectory data management. On the one hand, trajectory data has a complex structure, but **lacks fine-grained representation**. On the other hand, the query scenarios required by applications are diversified, so that the system must **efficiently support various query types**.

Fine-grained representation. Trajectory data consists of a sequence of locations with timestamps, forming a temporal range and an irregular spatial shape. These spatio-temporal features play a crucial role in trajectory analysis tasks. Databases usually employ one-dimensional structures for data storage. Hence, it is essential to convert these features into one-dimensional representations using indexing techniques.

Temporal feature. The temporal information of a trajectory refers to a time range that starts from the beginning to the end of the trajectory. Most indexes provided by databases only index a time point rather than a time range. TrajMesa [2] introduces an advanced temporal index called XZT index. It uses the dichotomy to divide a big time period into multiple *Element*s with various resolutions. However, the use of dichotomy leads to a maximum of 1/2 dead region, which hinders the efficiency of queries (Section II-1 provides examples). Additionally, some approaches directly use multiple time periods to represent time ranges. It stores a trajectory multiple times, resulting in significant redundancy. Hence, it requires a more accurate time range index without redundant storage.

Spatial feature. Locations visited by a trajectory contain spatial information, such as the shape of the trajectory. Spatial indexes aim to capture the spatial characteristics of trajectories by encompassing them in index spaces. The index values of index spaces are used to manage trajectories. R-tree and its variants [3]–[8] are commonly used spatial indexes. However, they inevitably suffer considerable overhead in adjusting and maintaining their index structures when managing large data, making them uncomfortable for key-value databases. In recent years, systems [2], [9]–[11] have employed XZ-ordering index to represent trajectories. XZ-ordering extends the principles of a quad-tree to generate enlarged elements with various resolutions. It uses the smallest enlarged element to represent trajectories. However, enlarged elements are rectangles that cannot accurately depict the shape of trajectories. Extended by XZ-Ordering, [12] proposes XZ* index. It divides an enlarged element into four sub-quads and uses non-rectangle index spaces formed by sub-quads to represent trajectories. Nevertheless, XZ* can only coarsely represent trajectory shapes.

Supporting various query types. Applications require various trajectory queries, such as temporal range queries, spatial range queries, and similarity queries. Therefore, it is necessary to devise an efficient storage schema and query processing techniques to accommodate as many queries as possible. Built on HBase, TrajMesa [2] and VRE [11] support multiple trajectory queries. TrajMesa provides diverse spatio-temporal indexes tailored to different query scenarios. It stores a trajectory in multiple index tables, leading to significant data redundancy. VRE introduces a segment-based storage model that sets the commonly used index as the primary table and stores other indexes in secondary index tables. However, it requires segmenting trajectories, which breaks their integrity. As a result, a large amount of reassembly overhead is required to reconstruct a whole trajectory. Thus, there is an urgent

*The research was done when the first author was an intern at JD iCity, JD Technology, China and JD Intelligent Cities Research. Yu Zheng and Tianrui Li are the corresponding authors.

need for a spatial index that can store intact trajectories while mitigating storage redundancy.

Our solution. We propose TMan (A High-Performance Trajectory Data Management System Based on Key-value Stores) to address the limitations mentioned above. TMan proposes two novel index structures and encoding methods to convert the spatio-temporal features of trajectories into one-dimensional values precisely. Moreover, TMan devises a storage scheme, an index cache mechanism, and corresponding query processing techniques to support trajectory queries.

TR index. The time ranges of trajectories vary in length. TMan represents time ranges using time bins consisting of k consecutive time periods, where k can be adjusted by the length of each time range. In contrast to XZT, the time period in TMan is very short, e.g., 30mins. A larger time bin corresponds to longer time ranges. Instead of repeatedly storing a trajectory in any intersecting time period, TMan devises an elegant and efficient encoding formula to avoid redundant storage. It assigns a distinct integer value to each time bin while endeavoring to ensure that adjacent index values correspond to time bins as near as possible. The encoding of our index remains fixed and concise (Equation 1), regardless of the data volume, and the computational overhead is negligible, making it highly suitable for distributed systems.

TShape index. To better represent the spatial features of trajectories, TMan proposes TShape index. Trajectory shapes are irregular, but conventional indexes can only depict the MBRs (minimum bounding rectangle) of trajectories. However, an MBR is too coarse to portray the shape of a trajectory. Thus, TShape index is devised to delineate the irregular shapes of trajectories using index spaces composed of multiple cells with variable shapes. TShape index selects an optimal combination of cells to represent the spatial shape of a trajectory, thereby enabling better differentiation of the spatial characteristics among different trajectories. Generally, if index spaces with high spatial correlation possess more adjacent index values, executing spatial queries can effectively get more related trajectories in a single I/O operation, reducing I/O overhead and improving query efficiency. Thus, TMan proposes a novel encoding approach, proving that this encoding is a traveling salesman problem (TSP, an NP-hard problem). TMan uses the greedy algorithm and genetic algorithm to implement this encoding. Therefore, TShape index can meticulously represent the spatial shapes of trajectories with fine-grained index spaces and store spatially correlated trajectories closer.

Storage. As TMan possesses strong representative capabilities, there is no need to split a trajectory into numerous small segments without practical semantics during storage. In TMan, a trajectory can be entirely stored in a single row, guaranteeing data integrity and reducing reassembly overhead during query processing. Additionally, TMan adopts a design with primary and secondary tables to accommodate diverse queries and reduce storage redundancy. Furthermore, TMan provides an index caching mechanism that substantially reduces the computation time of invalid index values.

The contributions of this paper are summarized as follows:

- We devise a concise and efficient temporal index called TR index. It represents time ranges of trajectories with suitable time bins and provides an encoding method to store them without redundancy. For temporal range queries, TR index can reduce the number of retrievals by up to 77%.
- We propose TShape index to portray the irregular shapes of trajectories using index spaces with variable shapes. Moreover, we prove that the encoding of TShape index is a TSP problem, and we solve it by the greedy algorithm and genetic algorithm. For spatial range queries, TShape index can reduce up to 83% of retrievals than XZ-Ordering.
- We provide a storage schema that stores intact trajectories instead of segments. Furthermore, with the careful design of the index cache mechanism and efficient query processing framework, TMan push-down queries into the storage layer and achieves the best performance than baselines.

The rest of this paper is introduced as follows. Section II reviews related trajectory management systems. Section III displays the overview of TMan. Section IV introduces the design of indexes and storage schema. Next, Section V gives query processing techniques for basic queries. We evaluate our work in Section VI and draw conclusions in Section VII.

II. Related Works

We review related works in 1) temporal indexes; 2) spatial and spatio-temporal indexes; and 3) trajectory systems.

1) Temporal Indexes: Like ST-Hadoop [13], a commonly used approach is partitioning time into adjacent and disjoint time bins with a fixed time period. However, a large dead region would be formed if the time period is too big. Conversely, a small time period could not encompass numerous time ranges, such that a trajectory may be stored in intersecting time periods with a copy, leading to redundant storage overhead and deduplication computation. VRE [11] divides trajectories into segments based on a time duration *d*, and it only uses the start time to index trajectory segments. When performing a time range query $q = [ts, te]$, it is necessary to inspect all segments with a start time located at $[\lfloor \frac{ts}{d} \rfloor * d, te]$, Figure 1(a) gives an example. This design has two limitations. 1) It requires segmenting trajectories, which breaks their integrity. As a result, a large amount of reassembly overhead is required to reconstruct a whole trajectory; 2) When the value of *d* is small, it may need to access a large number of irrelevant trajectory segments. TrajMesa [2] proposed XZT index. Firstly, XZT divides time into time periods with a big fixed period (e.g., two weeks). Next, it divides each time period into elements using the rule of dichotomy. Then, each element is doubled to get an *XElement*. For a time range $T.tr$, XZT index selects the smallest *XElement* that covers $T.tr$ to represent $T.tr$. However, the dead region of XZT index could also be big for a trajectory with a long time range. Figure 1(b) shows an example.

TR index is proposed to improve the precision for representing time ranges and avoid duplicated storage. It uses a time bin consisting of small time periods to represent the time range of a trajectory and design a graceful encoding for the time bin instead of duplicated storage (cf. Section IV-A1).

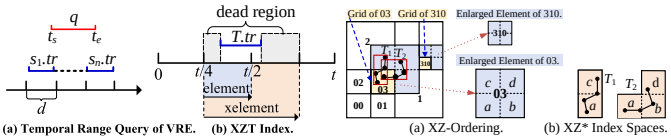


Fig. 1. Time Range Indexes.

Fig. 2. Spatial Indexes.

2) Spatial and Spatio-Temporal Indexes: Dynamic indexes (e.g., R-tree and its variants [3]–[8]) require much overhead to maintain their index structures when managing a large number of trajectories. Hence, these indexes are uncomfortable for key-value databases. XZ-Ordering [14] is widely used to index trajectories in systems based on key-value databases, including GeoMesa [9], TrajMesa [2], [15], JUST [10], and VRE [11]. XZ-Ordering divides the entire spatial region into grids with different resolutions. Next, each grid is doubled as an enlarged element. XZ-ordering uses the smallest enlarged element to represent the MBR of a trajectory. As depicted by the red lines in Figure 2(a), the girds labeled as '03' and '310' are doubled to generate their enlarged elements. Extended by XZ-Ordering, TraSS [12] proposes XZ* index. It divides the enlarged element into four sub-quads and employs the combination of these sub-quads as an index space to represent the shapes of trajectories. As illustrated in Figure 2(b), XZ* index uses the combinations of '*ac*' and '*abd*' to represent trajectories T_1 and T_2, respectively. Nevertheless, these indexes can only coarsely represent trajectory shapes.

[16] proposes a 3D R-tree that treats temporal information as the third dimension. However, this approach is unsuited for indexing trajectories with long time ranges [17]. HR tree [7] and H+R tree [8] divide the time dimension into disjoint time periods and build a spatial index within each period. However, these dynamic indexes suffer from maintainability and scalability problems, making them unsuitable for key-value databases. CSE-tree [18] uses grids to segment trajectories. Within each grid, it builds two B+trees based on trajectory segments' start and end times. Similar to VRE, this approach breaks the integrity of a trajectory and may miss the results of temporal range queries.

We propose *TShape* index to represent irregular shapes of trajectories elaborately (Section IV-A2). In addition, we present *ST* index for spatio-temporal queries (Section IV-A4).

3) Trajectory Management Systems: TrajStore [19] and Torch [20] are single machine-based systems that suffer from scalability problem. MobilityDB [21] is extended on PostgreSQL [22] with PostGIS [23]. It implements the GiST and SP-GiST indexes on R-tree for supporting spatio-temporal queries. ST-Hadoop [13], Summit [24], and PRADASE [25] process massive trajectories based on MapReduce framework, which requires an amount of I/O operations. UlTraMan [26], DITA [27], DFT [28], ST4ML [29], and REPOSE [30] are distributed in-memory systems. They are uneconomical in managing massive trajectory data. Simba [31], GeoSpark [32], and LocationSpark [33] can handle spatial objects but lacks supporting spatio-temporal data. [2], [9]–[12], [15], [34]–[36], [36]–[38] are based on NoSQL. THBase [39], TrajMesa [2], and VRE [11] are built on HBase. However, THBase does not optimize the trajectory storage. TrajMesa stores a trajectory multiple times in different index tables, which causes big storage costs. VRE stores segments instead of intact trajectories, which breaks the integrity of trajectories.

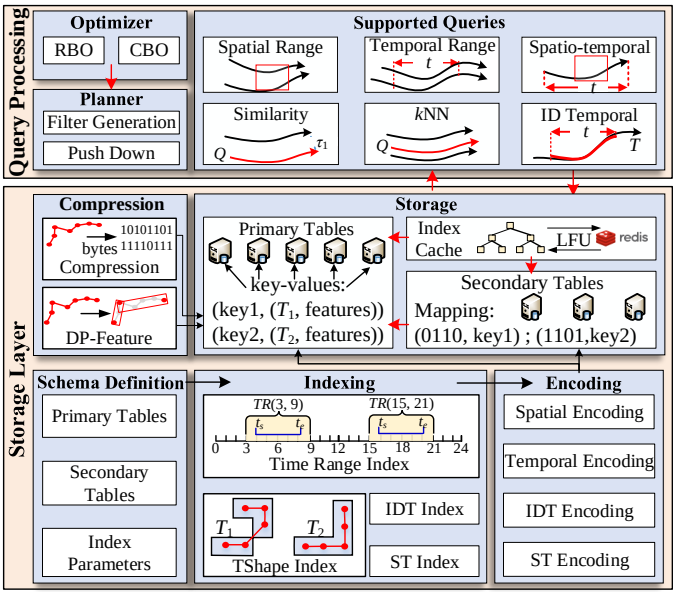


Fig. 3. Overview of TMAN.

Benefiting from the strong representative capabilities of our indexes, TMan can store an intact trajectory in a single row.

III. System Overview

Figure 3 gives the overview of TMan. It contains two core layers: storage layer and query processing layer.

Storage Layer. We employ primary and secondary tables to accommodate diverse queries and minimize redundant storage. Key-value data stores use one-dimensional key-value pairs to store data. The efficiency of trajectory queries heavily depends on precisely converting the spatio-temporal features of trajectories into one-dimensional encoding. To accomplish this, we introduce four indexes to accurately represent trajectory features. Additionally, we aim to retain the spatio-temporal features during encoding. When storage, trajectories are stored in primary tables, while the mappings of other indexes to the primary table are stored in secondary tables. Moreover, trajectories consist of multiple points, and consecutive points often exhibit similar features. Thus, TMan utilizes lossless compression to compress trajectories into bytes. Simultaneously, the dp-feature [12] is used to retain the representative features of trajectories. Finally, TMan incorporates an index cache mechanism to improve query processing efficiency.

Query Processing Layer. TMan provides efficient query processing capabilities to efficiently support various basic queries, such as spatial range query, ID temporal range queries, spatio-temporal range queries, and similarity queries. When a query is received, TMan generates the optimal query planner based on the Rule-Based Optimizer (RBO) and Cost-Based Optimizer (CBO). Next, TMan pushes down filters into relevant table regions and executes the query in parallel.

IV. Storage Layer

To efficiently manage large-scale trajectory data, TMan utilizes key-value data stores to support fundamental queries. However, two challenges need to be addressed.

Firstly, trajectories are multi-dimensional spatio-temporal data, and many queries highly rely on their spatio-temporal features. However, key-value databases store data using one-dimensional key-value pairs. Therefore, it is necessary to develop efficient indexing techniques that accurately capture the spatio-temporal features of trajectories. Moreover, encoding methods need to convert features to one-dimensional values. The primary spatio-temporal features of trajectories include time ranges and spatial shapes. To accurately represent these features, we devise TR index (Section IV-A1) and TShape index (Section IV-A2). Furthermore, we develop concise and efficient encoding methods for these indexes to preserve spatio-temporal properties of trajectories during storage.

Secondly, different query types require different indexes. To avoid redundancy, TMan uses primary and secondary tables with different indexes to manage trajectories. Moreover, we employ an index cache mechanism to enhance query performance and avoid unnecessary computations (Section IV-B).

A. Indexing and Encoding

TMan provides four fundamental indexes to support queries efficiently: 1) TR index, a static temporal range index, assigns integer values to the time ranges of trajectories, enabling efficient temporal range queries (Section IV-A1); 2) TShape index, a novel spatial index, uses non-rectangular index spaces to represent the spatial features of trajectories, avoiding retrieving many irrelevant trajectories during query processing (Section IV-A2); 3) IDT index combines IDs of trajectories and TR indexes to support ID-temporal range queries (Section IV-A3); 4) ST index (Section IV-A4) supports spatio-temporal range queries by merging TR and TShape indexes.

1) TR Index: Temporal range query (*TRQ*) retrieves all trajectories that intersect a given time range, which is crucial in various fields, such as analyzing the movement patterns of objects in a specific time duration. TR index is designed to support *TRQ* efficiently. A trajectory T spans a time range $[t_s, t_e]$ that starts from the time of the start point to that of the end point. TR index aims to: 1) use subtle time bins to represent time ranges of trajectories and 2) encode the time bins to partition trajectories with adjacent time ranges nearly.

Index Structure of *TR* index. The main idea of TR index is to combine small time periods as an index space. Initially, the timeline is divided into adjacent and disjoint time periods with a fixed length (e.g., an hour). Given a time range $T.tr = [t_s, t_e]$, ts and t_e are locate in the i-th and j-th time periods, respectively. Thus, we use the time bin from the i-th time period to the j-th time period to represent $T.tr$. As shown in Figure 4, we use $(j - i + 1)$ time periods as a time bin to represent the time range $T.tr$.

Encoding of *TR* index. Key-value data stores use the form of key-value pair to store data. Thus, we allocate each time bin to a unique integer value. The goal of our encoding is to assign adjacent time ranges to as many near values as possible. Before drilling into the details, we first give some related definitions.

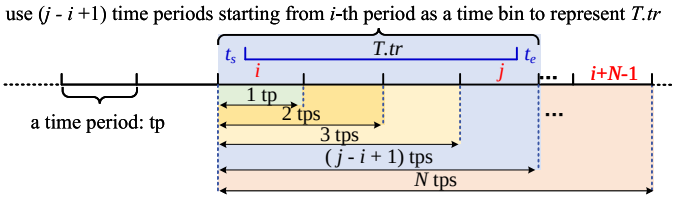


Fig. 4. Examples of TR Index.

Definition 1. *(**TP_i: Time Period**). TR index divides a timeline into adjacent and disjoint time periods by a fixed length. We use TP_i to represent the i-th time period.*

Definition 2. *(**$TB_{i,j}$: Time Bin**). $TB_{i,j}$ is a time bin starting from the i-th time period to j-th time period. Note that $i \leq j$.*

As the time is infinite, we set the start time of the timeline to that of the UNIX system, i.e., 1970-01-01 00:00:00. The number of time bins that starts from a time period is unlimited, causing the encoding space to be infinite. Thus, we set N as the number of the maximum periods that a time bin covers to limit the encoding space. Therefore, for a time bin $TB_{i,j}$, we have $i \leq j \leq i + N - 1$. In fact, after preprocessing, almost all trajectories do not have a time range longer than 48 hours. Hence, if the time period is 1 hour, N does not need to exceed 48. N can also be configured by users for flexibility. For a temporal range query q, if a time period TP_i intersects with q, all time bins $\mathcal{TB} = \{TB_{i,k} | i \leq k \leq i + N - 1\}$ starting from TP_i intersect with q, thereby $\mathcal{TB}$ are candidates for q. Therefore, we first encode the time bins starting from the same time period TP_i one by one. Next, we encode time bins starting from the next time period TP_{i+1}. Thus, the index value for time bin $TB_{i,j}$ is as follows:

$$TR(TB_{i,j}) = i * N + j - i. \tag{1}$$

Discussion. Although the encoding formula is concise, it ensures the uniqueness of the encoding and guarantees that adjacent time bins are also adjacent.

Lemma 1. *Index values of two adjacent time bins from the same time period are contiguous, i.e.,*

$$TR(TB_{i,j}) + 1 = TR(TB_{i,j+1}).$$

Lemma 2. *Index values of time bins starting from adjacent time periods are contiguous, i.e.,*

$$TR(TB_{i,i+N-1}) + 1 = TR(TB_{i+1,i+1}),$$

and their maximum interval is $2N - 1$, i.e.,

$$TR(TB_{i+1,i+N}) - TR(TB_{i,i}) = 2N - 1.$$

Proof. It is easy to conclude these lemmas by Equation 1. □

2) TShape Index: A trajectory contains a set of consecutive spatio-temporal points, which forms an irregular shape. Thus, we propose TShape index to assign appropriate codes to represent the shapes of trajectories.

(1) Index Structure of TShape Index. Inspired by XZ* index, TMan proposes TShape index. Firstly, a cell is recursively divided into four sub-cells with a higher resolution using the rule of quad-tree. Figure 5(a) illustrates cells at various resolutions. Next, an enlarged element is formed by using $\alpha * \beta$ cells at the same resolution, and the combination of cells is

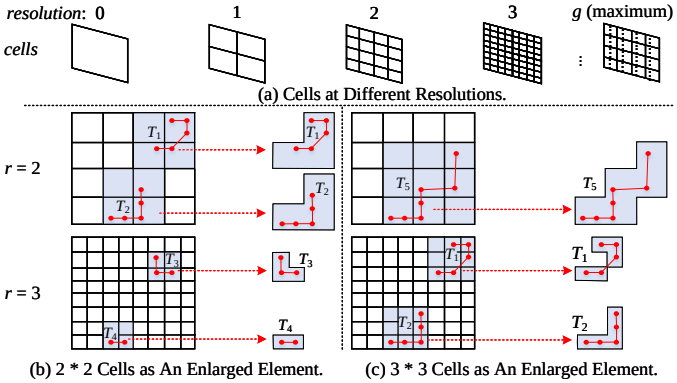


Fig. 5. Examples of TShape Index.

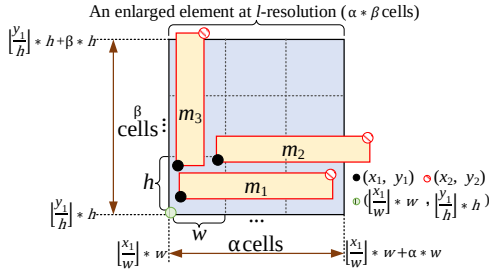
utilized as an index space to depict the shape of a trajectory. Figures 5(b) and (c) show examples. When $\alpha * \beta$ is $2 * 2$, T_1 and T_2 are represented by three cells in the second resolution, while T_4 is represented by two cells in the third resolution. When $\alpha * \beta$ is $3 * 3$, T_1 and T_2 can be finely represented by five cells in the 3rd resolution.

Trajectories vary in size, and cell size changes with resolution, making it inconvenient to represent large trajectories with $\alpha * \beta$ cells at a high resolution and impractical to represent small trajectories with $\alpha * \beta$ cells at a low resolution. Therefore, for a given trajectory T, we first determine the largest resolution that includes an enlarged element covering T. To simplify the process, we normalize the height and width of the spatial range to [0, 1]. An MBR is denoted by its lower left and upper right corners $((x_1, y_1), (x_2, y_2))$. Assuming that the appropriate resolution for an MBR of a trajectory is r.

Lemma 3. $r = l$ *or* $l-1$, $l = \lfloor log_{0.5}(max\{\frac{x_2-x_1}{\alpha}, \frac{y_2-y_1}{\beta}\})\rfloor$.

Lemma 4. *Let* $w = h = 0.5^l$, *if* $\lfloor \frac{x_1}{w} \rfloor * w + \alpha * w \geq x_2$ *and* $\lfloor \frac{y_1}{h} \rfloor * h + \beta * h \geq y_2$, *then* $r = l$, *otherwise,* $r = l - 1$.

Proof. The width and height of an enlarged element at r-resolution are $\alpha * w_1$ and $\beta * h_1$, respectively, where $w_1 = h_1 = 0.5^r$. The width and height of the MBR are $w_2 = x_2 - x_1$ and $h_2 = y_2 - y_1$, respectively. The enlarged element must cover the MBR, so $w_2 \leq \alpha * w_1$ and $h_2 \leq \beta * h_1$. Thus, $r \leq \lfloor log_{0.5}(max\{\frac{x_2-x_1}{\alpha}, \frac{y_2-y_1}{\beta}\})\rfloor = l$. However, there are MBRs whose widths are lower than $\alpha * 0.5^l$, but they cross more than α cells at l-resolution on the x axis. These MBRs must be represented by a lower resolution, specifically $l - 1$. The lower left corner of the MBR is located in the lower left cell of the enlarged element. The lower left corner of the enlarged element is $(\lfloor \frac{x_1}{w} \rfloor * w, \lfloor \frac{y_1}{h} \rfloor * h)$. Thus, if $\lfloor \frac{x_1}{w} \rfloor * w + \alpha * w < x_2$, the MBR crosses more than α cells at l-resolution in the x axis and cannot be represented by any enlarged element at the l resolution. The width of cell at $(l-1)$-resolution is $w_3 = 0.5^{l-1} = 2 * w$, so $w_2 \leq \alpha * w = \frac{\alpha}{2} * w_3$. Thus, the MBR crosses at most $\frac{\alpha}{2} + 1 \leq \alpha$ cells at $(l-1)$-resolution on the x axis, where $\alpha \geq 2$. Analogously, the MBR crosses at most $\frac{\beta}{2} + 1 \leq \beta$ cells on the y axis, thereby the MBR can be covered by an enlarged element at $(l-1)$-resolution. Similarly, if $\lfloor \frac{y_1}{h} \rfloor * h + \alpha * h < y_2$, the MBR must be represented by an enlarged element at $l - 1$ resolution. □



Appropriate resolutions for m_1, m_2, and m_3 are l, l - 1, and l - 1, respectively.

Fig. 6. Appropriate Enlarged Element for an MBR.

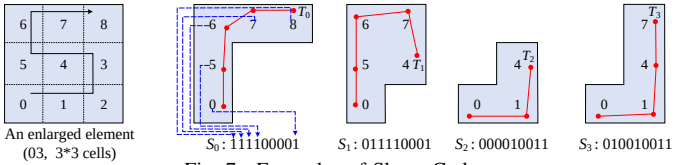


Fig. 7. Examples of Shape Codes.

Figure 6 provides examples. m_1 and m_2 have the same width, but m_2 crosses more than α cells on the x-axis, making it unable to be represented by the enlarged element at l-resolution. On the other hand, m_1 can be represented by the enlarged element at l-resolution. At the same time, m_3 can not be represented by the enlarged element at l resolution.

(2) Encoding of TShape Index. An enlarged element consists of $\alpha * \beta$ cells starting from the lower left cell. Therefore, we represent the enlarged element using the *quadrant sequence* of the lower left cell. Additionally, we use *shape codes* to indicate shapes within an enlarged element.

Quadrant Sequence and Quadrant Code: Each division of the quad-tree generates four sub-cells, and we number them from 0 to 3. Thus, a cell at r-resolution has a quadrant sequence from 1-resolution to r-resolution, denoted as $Q = q_1 q_2 ... q_r$. Figure 2(a) shows examples where a cell at 2-resolution is denoted by '03', and a cell at 3-resolution is denoted by '310'. We adopt the core idea of XZ-Ordering [14] to convert Q into an integer (quadrant code) while preserving its lexicographical order by using a depth-first visiting order. Formulaically,

$$code(Q) = \sum_{i=1}^{r} (q_i * \frac{4^{g-i+1} - 1}{3} + 1) - 1, \qquad (2)$$

where g is the maximum resolution. Figure 8(a) provides examples, e.g., the quadrant codes for enlarged element '03' and '33' are 4 and 20, respectively. d

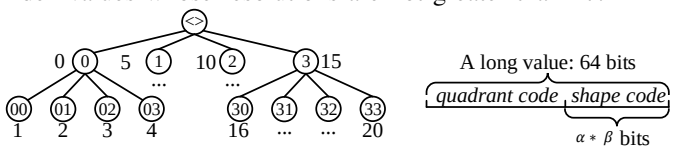
Shape Code: An enlarged element contains $\alpha * \beta$ cells, and we use the combination of cells as an index space to represent trajectory shapes. We use $\alpha * \beta$ bits to indicate whether a cell is used in forming an index space. If a trajectory intersects with a cell, its corresponding bit is 1. Otherwise, the bit is set to 0. Figure 7 gives examples. The enlarged element of '03' contains $3*3$ cells and covers T_0, T_1, T_2, and T_3. Their shapes are represented by shape codes s_0, s_1, s_2, and s_3, respectively.

Index Value: We use a long integer as the index value of a shape in an enlarged element. Compared to strings, integers are easier to maintain and more efficient. Additionally, integers occupy less space. For instance, representing an index space of an enlarged element with $4 * 4 = 16$ cells at the 8th resolution would require at least 24 bytes for string encoding, whereas a

long integer only requires 8 bytes. We utilize the last $\alpha*\beta$ bits of a long integer to record the shape code (s) of a trajectory and use the remaining bits to record the quadrant code of the corresponding enlarged element (E). Formulaically,

$$TShape(code(E), s) = (code(E) << (\alpha * \beta)) \mid s, \quad (3)$$

where '<<' is the bitwise left shift operator, '|' is the 'or' operator, as shown in Figure 8(b). The maximum quadrant code is $code(3^g) = \sum_{i=1}^{g} 4^{g-i+1} - 1 = \frac{4^{g+1}-4}{3} - 1 < \frac{4^{g+1}}{2} = 2^{2g+1}$. Thus, if $2g + 1 + \alpha * \beta \leq 64$, 64 bits are sufficient to encode index values, e.g., when $\alpha * \beta = 3 * 3$, we can hold index values whose resolutions are not greater than 27.



(a) Quadrant Codes of Enlarged Element (g=2). (b) Index Value of TShape.

Fig. 8. Index Values of TShape Index.

(3) Optimization of Shape Code. An enlarged element with $\alpha * \beta$ cells can generate $2^{\alpha*\beta}$ of shape codes. A larger $\alpha*\beta$ allows more detailed index values to represent trajectory shapes. However, a large range of index values can lead to two issues. 1) The number of shapes intersecting with a query would be enormous, resulting in extensive computation; 2) The bitmap does not consider the spatial features of trajectory shapes, resulting in scattered index values and causing similar trajectories to be stored in many different data regions. Thus, it leads to many I/O operations during querying. In reality, only a small number of shapes in an enlarged element are used to represent the shape of trajectories. Figure 16(a) illustrates a real case of used shapes in enlarged elements with $5 * 5$ cells. Almost all of the enlarged elements use less than 1000 shapes. Thus, we only need to encode the shapes that are actually used. Similar index spaces are likely to represent similar trajectories. Hence, our encoding should preserve this characteristic. Suppose an enlarged element has M shapes ($\mathcal{S} = \{s_0, s_1, ..., s_{M-1}\}$) that are actually used. The optimization goal is to renumber them from 0 to $M-1$ with the best order. Figure 9 shows an example.

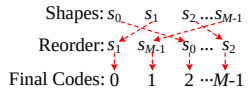


Fig. 9. Optimization of Shape Codes.

Shapes that are similar in space, the closer their shape codes should be. To measure similarity, we use the Jaccard similarity coefficient. The similarity of shapes s_i and s_j is:

$$Sim(s_i, s_j) = \frac{|s_i \cap s_j|}{|s_i \cup s_j|}, \quad (4)$$

where $|s_i \cap s_j|$ is the number of cells that are covered by s_i and s_j, $|s_i \cup s_j|$ is the number of cells that are covered by s_i or s_j. We aim to find a better order ($\mathcal{O} = \langle o_0, o_1, o_2, ..., o_{M-1} \rangle$) that maximizes the cumulative value of similarity between any two adjacent codes, where $o_i \in \mathcal{S}$, and $\forall i \neq j, then\ o_i \neq o_j$.

$$\arg\max_{\mathcal{O}} \sum_{i=0}^{M-2} Sim(o_i, o_{i+1}). \quad (5)$$

This problem can be seen as a variant of the TSP problem (Traveling Salesman Problem, an NP-hard problem), which finds the longest path without returning to the starting point. Given M shapes, we can represent the similarities between these shapes using a complete graph. Each shape corresponds to a node in the graph, and the similarities between shapes correspond to the edges between nodes. The objective is to find the longest path that visits all shapes exactly once. In this paper, we utilize the Greedy Algorithm [40] and Genetic Algorithm [41] to solve the encoding of shapes. Specifically, the greedy algorithm is a heuristic algorithm that builds a path by choosing the furthest unvisited shape for each iteration. The advantage of the greedy algorithm is its simplicity and efficiency. Figure 7 presents four shapes, and their similarities are shown in Figure 10. The cumulative similarity of the order ($\langle s_0, s_1, s_3, s_2 \rangle$) selected by the greedy algorithm is 1.92, while that of the raw order ($\langle s_0, s_1, s_2, s_3 \rangle$) is 1.75.

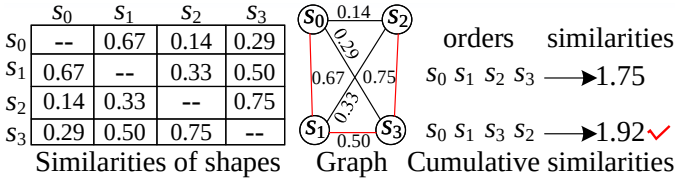


Fig. 10. Cumulative similarities of different orders.

3) IDT Index: We propose IDT index to support ID temporal queries. Given a trajectory T, its start time and end time are in TP_i and TP_j, respectively. The IDT index value is,

$$IDT(T) = T.oid :: TR(TB_{i,j}),$$

where '::' is the concatenation operation, $T.oid$ is the identifier of the object that generates the trajectory T, and $TR(TB_{i,j})$ is the TR index value of the time range of the trajectory.

4) ST Index: ST index supports spatio-temporal range query (STRQ) efficiently. The index value of ST index is

$$ST(T) = TR(TB_{i,j}) :: TShape(code(E), s),$$

where $TR(TB_{i,j})$ is the index value of time range of T, E is quadrant sequence of the smallest enlarged element that covers T and s is the optimize shape code of T.

B. Storage Schema

The goal of designing a storage schema is to manage large-scale trajectory data in key-value databases effectively. TMan provides two kinds of table types: the primary table and the secondary table. Users can create primary tables for query scenarios that require high efficiency. For query scenarios that do not require the best efficiency, it is recommended to use secondary tables, which can reduce storage overhead. Additionally, we adopt an index cache mechanism to minimize the computation of the index. The metadata table stores the parameters of indexes and user configurations.

(1) Primary Table. It stores the trajectory data using a primary index. The key of the primary table is as follows:

$$rowkey = shards :: index\ value :: tid, \quad (6)$$

where $shards$ is a hash number used to avoid the hot-spotting problem. The primary index calculates $index\ value$ of a

Primary Table (*TShape*)					
key	value				
	oid	tid	points	tr	features
0001t1	1	t1	byte[ ]	11	dp-features
0011t2	1	t2	byte[ ]	11	dp-features
0011t3	2	t3	byte[ ]	12	dp-features
0111t4	3	t4	byte[ ]	12	dp-features
1011t5	4	t5	byte[ ]	23	dp-features
1101t6	4	t6	byte[ ]	23	dp-features
...	...	...	...	...	...

SecTable (*IDT*)	
key	value
1-11	[0001t1, 0011t2]
2-12	[0011t3]
...	...

SecTable (*TR*)	
key	value
12	[0011t3, 0111t4]
23	[1011t5, 1011t6]
...	...

Fig. 11. Storage Schema (TShape Index is the primary index).

trajectory to support the primary query efficiently, e.g., if there is a need for numerous spatial range queries, set TShape as the primary index. tid is the unique identifier of a trajectory.

The value of the primary table contains all trajectory information. As depicted in Figure 11, the value of each row consists of oid, tid, $points$, tr and $features$. An object can generate at least one trajectory. oid is the identifier of the object that generates the trajectory, and tid is the identifier of the trajectory. tr denotes the index value of TR index.

$points$: Trajectory data often consists of a large number of location points. It requires significant storage and transmission overhead. Thus, instead of storing the raw points, we compress a trajectory into bytes without losing data quality. First, a trajectory is converted to three arrays: latitude, longitude, and timestamp values. Then, we can use compression methods such as Elf [42], Elf+ [43], VGB [44], simple8b [45], and PFOR [46] to compress these values into bytes, which are stored in the *points* column. Besides, [47] gives more compression methods for integers.

$features$: We utilize the DP-Features proposed in [12] to extract representative points and bounding boxes from a trajectory. [12] proved that DP-Features can greatly reduce the computation overhead for similarity queries. TMan also leverages DP-Features to improve the efficiency of spatial range queries and spatio-temporal queries.

(2) Secondary Table. The secondary tables of TMan only store the relationship between the secondary index values and the primary keys. Figure 11 shows two examples. The key of the IDT table consists of the oid of an object and the TR index value of the time range of a trajectory. The value corresponds to row keys of a primary table that record trajectories generated by the object. At the same time, the key of TR table is the TR index values that are used to represent trajectory time ranges.

(3) Index Cache. Only a few shapes within an enlarged element are used to represent trajectories and are renumbered by way of Section IV-A2(3). Therefore, we must maintain the mapping between final codes and used shapes in an enlarged element. We utilize Redis [48] to store the tuple $\langle enlarged\ element,\ shape,\ final\ code \rangle$. For example, the tuples for s_0, s_1, s_2, and s_3 in Figure 7 and Figure 10 are $\langle$03, 111100001, 0$\rangle$, $\langle$03, 011110001, 1$\rangle$, $\langle$03, 000010011, 3$\rangle$, and $\langle$03, 010010011, 2$\rangle$, respectively. When the database receives a query request, it first looks up the corresponding enlarged elements in the index cache. If an enlarged element is not found in the index cache, the related tuples of the enlarged element are read from Redis into the index cache. Additionally, to ensure query performance and reasonably use the limited cache space, the LFU [49] strategy is employed to remove the least frequently used cache from memory.

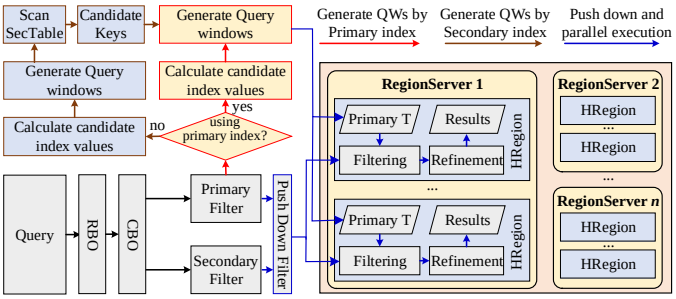


Fig. 12. Flow of Query Processing.

(4) Metadata Table. TMan utilizes a metadata table to store information about the indexes used to build primary and secondary tables. It also records the parameters of the indexes, such as α and β for building *TShape* index.

C. Update

TMan offers an update operation for flexible management of trajectory data. By utilizing TShpae index from IV-A2, we can compute the quadrant codes and shape codes for given trajectories. Subsequently, we group trajectories by their quadrant codes. Within each group, we assign optimized shape codes to the trajectories. To enhance the encoding of shape codes, we have implemented an optimization mechanism and stored the mapping of shape codes and optimized codes in an index cache. We also employ a buffer shape cache to handle shape codes that have not been previously optimized. Fisrtly, we store new trajectories with shape codes that have been encountered by accessing the index cache and buffer shape cache. Subsequently, for trajectories whose shape codes are not found in either the index cache or buffer shape cache, we store these shape codes in the buffer shape cache and store the trajectories using their raw shape codes. Once the number of shape codes in the buffer shape cache exceeds a maximum threshold, we trigger a re-encoding process for all shape codes in both index cache and buffer shape cache. Concurrently, we extract data with outdated index codes, delete them, and store them using the updated index codes. Next, we update index cache and clear buffer shape cache for the next iteration. This approach helps to efficiently manage the encoding and storage of shape codes while minimizing computational overhead.

V. Query Processing

A. Main Idea

Figure 12 presents the process of query processing. With carefully designed indexes and storage schema, TMAN can support various fundamental queries efficiently. Users may not set all indexes as a primary table for economic considerations. Thus, TMAN provides an RBO (Rule-based Optimization) to determine the optimal table for a given query. TMAN supports six types of queries, including temporal range queries, ID-temporal queries, spatial range query queries, spatio-temporal range queries, threshold similarity queries, and top-k similarity queries. Given a query, TMAN first judges its query type. A complicated query can use indexes $\mathcal{I} = \{I_1, I_2, ..., I_n\}$ to accelerate efficiency, the priority is as follows:

$$IDT > primary\ indexes > secondary\ indexes.$$

IDT index is the highest priority because the number of trajectories generated by the same object is typically much smaller than other queries. $primary\ indexes$ has higher priority than $secondary\ indexes$ because the $secondary\ indexes$ need to query the relevant keys from the secondary table first and then obtain the trajectories from the primary table, resulting in sub-optimal performance. Besides, if a query contains two kinds of query types, TMAN may change the query planner by calculating the cost of different execution strategies based on CBO (Cost-based Optimization).

Once the appropriate index is selected to accelerate a query, it is executed using three steps: 1) calculating candidate index values (Sections from V-B to V-F); 2) generating query windows (V-F)(1); and 3) pushing down filters (V-G(2)).

B. Temporal Range Query

TMAN utilizes the *TR* index values to represent time ranges of trajectories. The time bin of an index value is greater than or equal to the time ranges indexed with that value. Given a time range $tr = [ts, te]$, suppose $tr.ts$ is in i-time period TP_i and $tr.te$ is in TP_j. *TRQ* first finds all time bins $\mathcal{TB}$ that satisfy Equation 7, then refines trajectories by comparing trajectory time ranges with tr in the step of pushing down.

$$\forall TB_{k,p} \in \mathcal{TB}, \exists TP_m \in TB_{k,p} \Rightarrow TP_m \cap TB_{i,j} \neq \emptyset. \quad (7)$$

Lemma 5. *$\forall\ TB_{k,p}$ intersecting with $TB_{i,j}$, k and p satisfy:*
1) $i - N + 1 \leq k \leq j$;
2) if $i - N + 1 \leq k < i$, then $i \leq p \leq k + N - 1$;
3) if $i \leq k \leq j$, then $k \leq p \leq k + N - 1$.

Proof. $TB_{k,p}$ cannot exceed N periods. Thus, the start time period of $TB_{k,p}$ must be greater than or equal to TP_{i-N+1}, otherwise $TB_{k,p} \cap TB_{i,j} = \emptyset$. If $k > j$, time bins starting from TP_k cannot intersect with $TB_{i,j}$. Thus, $i - N + 1 \leq k \leq j$.

When $i - N + 1 \leq k < i$, if $p < i$, then $TP_p < TP_i$, so that $TB_{k,p}$ cannot intersect with tr. If $i \leq p$, at least one time period of $TB_{k,p}$ within $TB_{i,j}$, i.e., $TB_{k,p} \cap TB_{i,j} \neq \emptyset$. Besides, $TB_{k,p}$ starts from TP_k, and $|TB_{k,p}| \leq N$, so that $TP_p \leq TP_{k+N-1}$. Thus, $i \leq p \leq k + N - 1$.

When $i \leq k \leq j$, we have $TB_{k,p} \cap TB_{i,j} \neq \emptyset$. For $TB_{k,p}$, the end time period must be among time periods from TP_k to TP_{k+N-1}, so that $k \leq p \leq k + N - 1$. □

Figure 13 shows examples. TMAN calculates corresponding index values of tr based on Lemma 5. The index values of adjacent time bins are contiguous based on Lemma 1 and 2. Thus, when $i - N + 1 \leq k < i$, and $i \leq p \leq k + N - 1$, candidate index values of time bins starting from TP_k are in a closed interval $[TR(TB_{k,i}),\ TR(TB_{k,k+N-1})]$. Additionally, candidate index values of time bins starting from time periods $\{TP_k | i \leq k \leq j\}$ form a closed interval:

$$[TR(TB_{i,i}),\ TR(TB_{j,j+N-1})] = \bigcup_{i \leq k \leq j} [TR(TB_{k,k}),\ TR(TB_{k,k+N-1})].$$

Hence, only N closed intervals can be candidates. Algorithm 1 gives a detailed algorithm to calculate index values.

Discussion. Let Q represent the length of the query range, D represent the size of the dataset, and T represent the number of time periods covering the dataset. Assuming that N represents the maximum time period of the trajectories and the start times of the trajectories are evenly distributed among T time periods. Each period has N time bins, and trajectories are evenly distributed. Thus, each time bin has $D/(T * N)$ of trajectories. A TR query, as described in Algorithm 1, requires retrieving approximately $\frac{N*(N-1)}{2} + Q * N$ of time bins, such that $\frac{N-1+2Q}{2T} * D$ of data must be retrieved by the query. Assuming a time period is 30 minutes, the time range T of D is 1488 (equivalent to one month, $31 * 24 * 2 = 1488$), the maximum time range N of trajectories is 8 time periods, and a query range Q of 2 time periods. By substituting these parameters into the formula, we can calculate that the amount of data required for the TR query is approximately $0.0037D$. It is worth noting that the actual situation may be influenced by factors such as the particular data distribution.

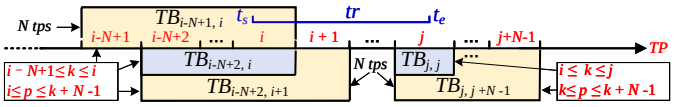


Fig. 13. Examples of TRQ.

Algorithm 1: TR Index Values of tr: *TR_values*(*tr*)

Input: A time range: $tr = [ts, te]$.
Output: Index values: $values$.
$i = TP(tr.ts)$; $j = TP(tr.te)$; //Time periods of ts and te;
$k = i - N + 1$;
for $k < i$ **do**
  $values.add([TR(TB_{k,i}),\ TR(TB_{k,k+N-1})])$;
  $k = k + 1$;
$values.add([TR(TB_{i,i}),\ TR(TB_{j,j+N-1})])$;
return $values$;

C. Spatial Range Query

TMAN uses TShape index to represent the shapes of trajectories. The index space of the TShape index is composed of certain cells in an enlarged element, forming an irregular shape. Figure 5 gives some examples. Given a spatial range SR, the spatial range query SRQ aims to calculate all index values whose corresponding shapes intersect with SR. We first use a Breadth-First Search to check the spatial relationship between enlarged elements and the query range SR, and pick out related shapes. There are three spatial relationships:
(1) ***contains***: If a query range SR covers an enlarged element $E_i = q_1 q_2 ... q_r$, the enlarged element E_j prefixed with E_i is also covered by SR. We use $E_j \subseteq E_i$ to represent E_j prefixed with E_i, e.g., $E_i = 31$ and $E_j = 310$. Formally,

$$E_i \subseteq SR, \forall E_j \subseteq E_i \Rightarrow E_j \subseteq SR. \quad (8)$$

Proof. Quadrant sequences of enlarged elements are encoded by a Depth-First strategy. We use the quadrant sequence of the left lower cell to represent an enlarged element. Thus, the left lower cell of E_i covers that of E_j. In addition, the size of a single cell in E_j is smaller than that of E_i. Therefore, E_i covers E_j. Thus, if SR covers E_i, then $E_j \subseteq SR$. □

Assuming the resolution of E_i is r, the maximum resolution is g, so the number of enlarged elements prefixed with E_i is

$$EN(E_i) = \sum_{i=l}^{g} 4^{i-r}.$$

Based on Equation 2, the quadrant codes of all enlarged elements prefixed with the same quadrant sequence are consecutive. Thus, quadrant codes of these enlarged elements are in a left-closed and right-open interval:

$$[code(E_i), code(E_i) + EN(E_i)).$$

All index spaces of these enlarged elements are picked as candidates. Based on Equation 3, the corresponding index values of these index spaces are in the interval:

$$[TShape(code(E_i), 0), TShape(code(E_i) + EN(E_i), 0)).$$

(2) ***intersects****:* If SR intersects with an enlarged element, we need to verify whether the shapes of this enlarged element intersect with SR. A shape s of an enlarged element can be considered to a candidate only if it intersects with SR.
(3) ***disjoint****:* These enlarged elements can be directly ignored.

The procedure for generating index values of SRQ is presented in Algorithm 2. We use the breadth-first strategy to pick out related index values. Initially, we add cells at the 1st resolution into the FIFO (First in, first out) queue $remaining$ (Lines 1 to 3) and add a '$LevelTerminator$' that marks all enlarged elements at the same resolution are checked. Then, the candidate index values are generated recursively. In line 6, we poll the top cell c in the $remaining$ queue. If the current enlarged element $E = c.enlarged\_element$ is contained in SR, we add all index values whose corresponding enlarged elements are prefixed with E to the final $value$, as shown in lines 13-16. Otherwise, if SR intersects with E, we obtain the actually used shapes of E from the index cache (cf. Section IV-B(C)). For each shape intersecting with SR, we add its corresponding index value to the final $value$, as shown in lines 18-22. In lines 23-25, we add four children of the current cell into $remaining$ for the next level of checking. If the current c is '$LevelTerminator$' and $remaining$ is not empty, we continue to check index space at the next resolution and add '$LevelTerminator$' to the end of $remaining$, as shown in lines 7-10. If the $remaining$ is empty, we finish the search.

Algorithm 2: TShape Values: *TShape_values*(*SR*)

Input: A spatial range: SR.
Output: Index values: $values$.

```
1  l = 1;
2  for cell ∈ Root.children do
3      remaining.add(cell);
4  remaining.add(LevelTerminator);
5  while remaining ≠ ∅ do
6      c = remaining.poll;
7      if c = LevelTerminator then
8          if remaining ≠ ∅ then
9              l = l + 1;
10             remaining.add(LevelTerminator);
11     else
12         E = c.enlarged_element;
13         if SR contains E then
14             min = TShape(code(E), 0);
15             max = TShape(code(E) + EN(E), 0));
16             value.add([min, max));
17         else if SR intersects E then
18             for shape in Index_cache(E) do
19                 if shape intersects SR then
20                     v = TShape(code(E), shape.code);
21                     value.add([v, v]);
22             if l < g then
23                 foreach subElement ∈ c.children do
24                     remaining.add(subElement);
25 return values;
```

D. ID Temporal Query

Recall the *IDT* index (cf. Section IV-A3), we first calculate the TR index values of the given time range (cf. Section V-B). Then, we combine the given ID and the TR index values to generate query windows and execute the query in parallel.

E. Spatio-temporal Range Query

Recall the *ST* index (cf. Section IV-A4), we need to calculate the range of TR index values for the given time range tr and calculate the range of $TShape$ index values for the given spatial range SR (cf. Sections V-B and V-C). Then, we combine the TR index values and $TShape$ index values to generate query windows and execute the query in parallel.

F. Similarity Queries

Similarity queries [50] are crucial for various trajectory data analysis tasks. When $\alpha = 2$ and $\beta = 2$ and we do not use the index cache, the TShape index is similar to an XZ* index (proposed in $TraSS$ [12]). Additionally, we observe that the query processing of the similarity searches designed by [12] can utilize the TShape index to achieve the same global pruning and local filter proposed in [12] with a slight modification. Thus, TMAN incorporates the capabilities of $TraSS$ to support threshold similarity and top-k similarity queries. Please refer to [12] for more details.

G. Generate Query Windows and Push Down

(1) Generate Query Window. As described in Section IV-B, trajectories are stored in the primary table. Sections from V-B to V-F provide algorithms to calculate candidate index values for fundamental queries. We generate query windows to extract related trajectories based on the candidate index values. Since trajectories are only stored in the primary table, if the primary filter of a given query needs to use the primary index, we combine $shards$ and candidate index values to generate query windows. Otherwise, we scan the secondary index table to obtain candidate keys and generate query windows by these keys. Figure 12 shows the process.
(2) Push Down and Parallel Execution. Typically, key-value databases store data in distributed storage systems. Filter conditions can be pushed down to the storage layer, enabling the storage system only to return data that satisfies the conditions rather than all candidate data. By pushing down queries to the storage layer, we minimize the overhead of data transmission and enhance the efficiency of queries. TMAN supports three basic filters: temporal filter, spatial filter, and similarity filter. It also allows combining multiple filters into a filter chain to implement complex filtering logic. This filter chain is pushed down to relevant data regions and executed in parallel. The push-down operation reduces unnecessary data transmission, while parallel execution takes advantage of the distributed architecture of the storage system to efficiently process queries. By leveraging push-down and parallel execution techniques, TMan improves the efficiency and scalability to handle large-scale trajectory data.

VI. Evaluation

We evaluate the performance of temporal range queries, spatial range queries, spatial-temporal range queries, ID temporal queries, threshold and top-k similarity searches.

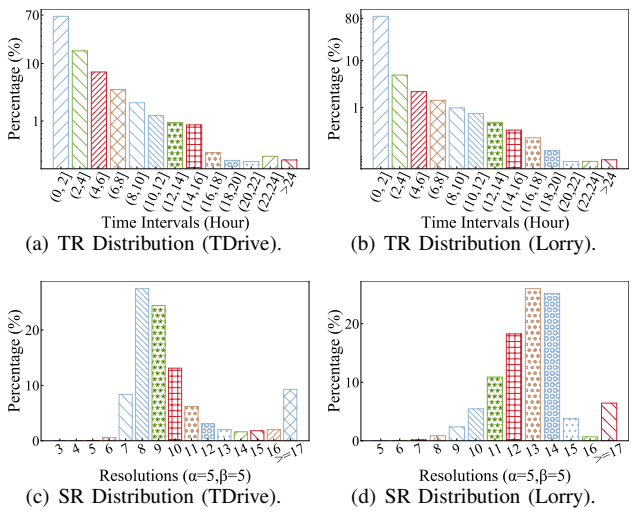
Baselines. We evaluate our work with other state-of-art works, i.e., *TMan* (our work), *TrajMesa* [2], *ST-Hadoop* [13], *TraSS* [12], *DITA* [27], *DFT* [28], and *REPOSE* [30]. Among the systems, *TMan*, *TrajMesa*, and *ST-Hadoop* support temporal range, spatial range, and spatio-temporal range queries. *TMan* and *TrajMesa* also support ID temporal query. We compare similarity queries with *TraSS*, *TrajMesa*, *DITA*, *DFT*, and *REPOSE*. Besides, we retrofit *TrajMesa* by adopting the storage schema and push-down strategies of *TMan*. Note that spatio-temporal indexes adopted by *VRE* [11] are similar to *TrajMesa*, and the code of *VRE* is not released. We have shown the advantages of our system compared to *VRE* in Section II.

Datasets. We evaluate the efficiency of *TMan* using three datasets: (1) **TDrive** [51], which contains 318,744 taxi trajectories of Beijing during a week; (2) **Lorry**, consisting of 2,643,450 lorry trajectories of Guangzhou, China, generated by lorry drivers from 2014-03-01 to 2014-03-31; (3) **Synthetic**. To evaluate the scalability, we offset the time range and spatial location of the original data to generate 10x Lorry data. Section VI-A shows the distribution of TDrive and Lorry.

Setting. We randomly generate 100 query windows within the spatio-temporal range of TDrive and Lorry, respectively, and consider the 50th percentile of the query results as the final result. In Section VI-A, we evaluate the performance of our proposed indexes. We vary the time range from 5 minutes to 24 hours to evaluate the efficiency of *TRQ* in Section VI-B. Next, in Section VI-C, we vary the spatial range from 100m*100m to 2500m * 2500m to evaluate *SRQ*. The performance of the *STRQ* and *IDT* are evaluated in Section VI-D. We analyze similarity queries in Section VI-E. Furthermore, we use synthetic datasets to evaluate the scalability and tail latency in Section VI-F. Our experiments are carried out on five nodes (each with an 8-core CPU, 1T disk, and 32GB memory).

A. Effect of Indexes

1) Distribution of Datasets: Figures 14(a) and (b) illustrate the distribution of time ranges in TDrive and Lorry datasets. In TDrive dataset, approximately 66% of trajectories have time ranges of less than 2 hours, and over 99% of trajectories are less than 18 hours. In Lorry dataset, about 88% of trajectories are less than 2 hours, and 99% of trajectories are less than 14 hours. The spatial distribution of trajectories is evaluated with $\alpha = 5$ and $\beta = 5$. The spatial boundaries of TDrive and Lorry are (110, 35, 125, 45) and (70, 0, 140, 55), respectively. Figures 14(c)(d) display the percentage of trajectories at different resolutions. Figure 14(c) reveals that most TDrive trajectories are concentrated within enlarged elements with resolutions ranging from 7 to 10. That is because drivers in Beijing typically transport passengers between 2.7km and 65km. In Figure 14(d), less than 1% of trajectories in Lorry exhibit big spatial ranges due to transporting goods to other



(a) TR Distribution (TDrive). (b) TR Distribution (Lorry).
(c) SR Distribution (TDrive). (d) SR Distribution (Lorry).

Fig. 14. TR and SR Distribution of Datasets.

cities. Furthermore, most trajectories of Lorry are distributed in resolutions from 9 (about 76km) to 14 (about 2km).

2) Performance of TR Index: *TR* index represents a time range by m time periods. We set the time period of TR index as 10 minutes, 30 minutes, 1, 2, 4, 6, and 8 hours, denoted as *TR*-10M, *TR*-30M, *TR*-1H, *TR*-2H, *TR*-4H, *TR*-6H, and *TR*-8H, respectively. Besides, we compare *TR* index with XZT. Table I presents the query time and the number of candidates varying with query windows. Our observations are: 1) as the query window increases, more time is required to visit more trajectories; 2) *TR* index outperforms *XZT* index. For instance, when the query window is 24 hours, *TR*-1H outperforms *XZT* by about 3x; 3) Shorter periods result in fewer trajectories being accessed. However, a shorter time period increases the encoding space of index values, which negatively affects data locality. Thus, although *TR*-1H retrieves more data than *TR*-10M, the data storage of *TR*-1H is more centralized than *TR*-10M, so sometimes *TR*-1H outperforms *TR*-10M.

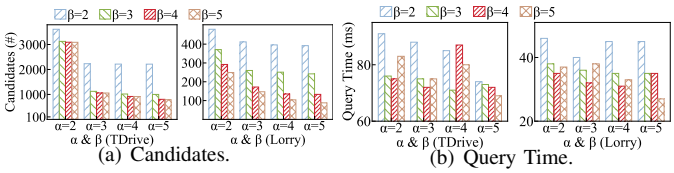
TABLE I
Performance of Temporal Indexes (Lorry)

Indexes	Query Time of Different Time Windows (ms)							Candidates of Different Time Windows (#)						
	5m	10m	30m	1h	6h	12h	24h	5m	10m	30m	1h	6h	12h	24h
XZT	273	276	345	543	1116	2058	4029	24020	25510	29974	35739	108883	202772	412558
TR-10M	**133**	204	222	252	519	894	1560	**3692**	**3829**	**4959**	**7299**	**23420**	**45086**	**90965**
TR-30M	162	**170**	**171**	**186**	546	876	1486	5148	5242	6307	8317	24308	4586	92139
TR-1H	159	180	198	204	**480**	**837**	**1342**	6982	7114	8980	10099	25935	47703	93921
TR-2H	165	172	174	195	498	891	1498	10595	10595	10595	10697	29548	51363	97261
TR-4H	207	213	213	234	531	903	1590	16911	16911	16911	16911	33989	59525	104700
TR-6H	288	279	291	291	639	999	1659	23669	23669	23669	23669	46527	67104	111423
TR-8H	333	318	312	342	636	1047	1752	29589	29589	29589	29589	37724	69831	117707

3) Performance of TShape Index: We evaluate the efficiency of *TShape* index by executing spatial range queries (1.5km * 1.5km). The index space of *TShape* index is composed of a maximum of $\alpha * \beta$ cells. We devise encoding methods tailored for index spaces to store trajectories.

Effect of α and β: We vary $\alpha * \beta$ from $2 * 2$ to $5 * 5$ to observe the effect of $\alpha * \beta$. Figure 15(a) shows that the number of visited candidates decreases as β increases while keeping α fixed. This phenomenon occurs because a larger $\alpha * \beta$ allows for the representation of finer shapes so that more trajectories can be filtered. However, a larger $\alpha * \beta$ generates more index spaces, leading to more scattered data storage. Consequently, query processing requires more time to

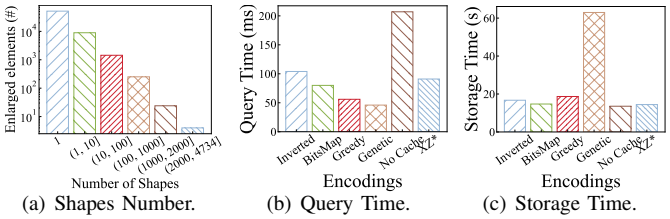
calculate the intersecting index spaces with the query range. Thus, as shown in Figures 15(a)(b), we observe that while the ability to represent shapes of $3 * 3$ is slightly worse than that of $3 * 4$, $3 * 3$ exhibits faster query times.



(a) Candidates. (b) Query Time.

Fig. 15. Varying α and β.

Effect of Encoding: An enlarged element formed by $\alpha * \beta$ cells has the potentiality to generate $2^{\alpha*\beta}$ shapes. However, in real-world datasets, only a small fraction of these shapes are used to represent trajectories. Figure 16(a) illustrates that an enlarged element only utilizes less than 4734 shapes to represent trajectories, and most enlarged elements contain fewer than 10 shapes. Therefore, we employ an index cache mechanism to maintain the shape codes that are actually used in each enlarged element. Additionally, we explore different encoding methods to optimize the representation of shapes. Figure 16(b) demonstrates the consequence of not utilizing an index cache. In this case, significant computation time is wasted searching for $2^{\alpha*\beta}$ shapes that may intersect the query conditions. Among the encoding methods, genetic encoding outperforms the others because it assigns adjacent index values to proximity shapes, reducing disk I/O operations during querying. Moreover, we apply the index cache and push-down strategies to *XZ** index. Experimental results indicate that *TShape* index performs better than *XZ** index. Nonetheless, as depicted in Figure 16(c), compared to greedy encoding and bitMap encoding, genetic encoding requires more time to determine the optimal order of shape codes, resulting in more storage time than others. Besides, instead of indexing a trajectory using a code, we use the inverted list of intersecting cells to store each trajectory, which requires more storage cost and brings more I/O cost. Moreover, it needs time to remove duplicates. Thus, it is slower than *TShape* index.

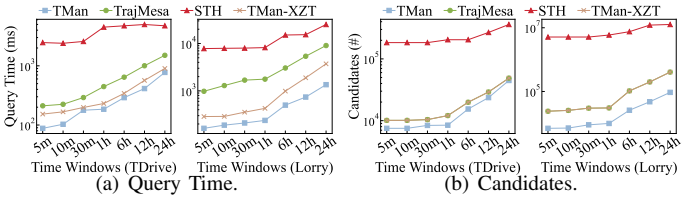


(a) Shapes Number. (b) Query Time. (c) Storage Time.

Fig. 16. Performance of *TShape* Index (TDrive, $\alpha * \beta = 5 * 5$).

B. Temporal Range Queries

We evaluate the temporal range queries. Figure 17(a) displays the query time, and Figure 17(b) shows the number of candidates. For *TrajMesa* and *TMan*, candidates are visited trajectories, whereas for *STH*, candidates are visited points since trajectories are split into points and stored in HDFS. Figure 17(a) indicates that *TMan* achieves the best performance. *TrajMesa* utilizes XZT to index the time ranges of trajectories, while *TMan*-XZT uses the same XZT index in our framework. *TMan*-XZT outperforms *TrajMesa*, due to its ability to push down the query condition. *TR* index has an exquisite index structure and encoding method, which reduces the number of visited candidates in TDrive and Lorry by 30% and 77% than *TrajMesa*, respectively. Furthermore, Figure 17(b) shows that *TMan* using *TR* index accesses the fewest number trajectories, even by one or two orders of magnitude than *STH*.

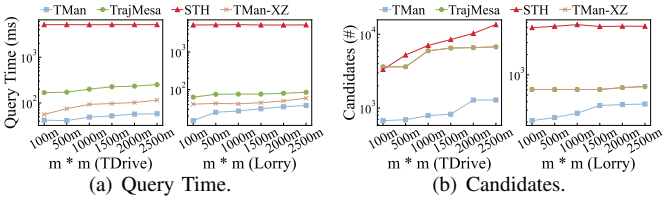


(a) Query Time. (b) Candidates.

Fig. 17. The Performance of Temporal Range Query.

C. Spatial Range Queries

We vary spatial windows from 100m*100m to 2500m * 2500m to evaluate spatial range queries. Figure 18 shows that as the spatial window size grows, all systems require longer query time due to accessing more candidates. *TMan* is faster than *TrajMesa* and *STH* by pruning more irrelevant trajectories. *STH* needs to build a Map-Reduce task on HDFS, leading to multiple I/Os that slow down its query performance. *TMan-XZ* adopts the spatial index of *TrajMesa*. The results indicate: 1) with the index cache and push-down strategies, our work is superior to *TrajMesa*; 2) *TMan* with *TShape* surpasses *TMan-XZ*, demonstrating *TShape* index is fine than XZ-Ordering. Averagely, compared to XZ-Ordering, the *TShape* index reduces the number of visited candidates on TDrive and Lorry by 83% and 52%, respectively.



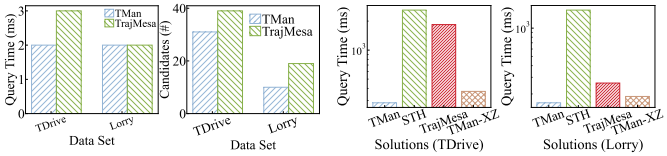
(a) Query Time. (b) Candidates.

Fig. 18. Performance of SRQ.

D. IDT and STR Queries

IDT Query: We conduct experiments for IDT query, as shown in Figure 19(a). Among our baselines, only *TrajMesa* supports IDT query. Typically, the number of trajectories that are generated by a moving object is very small. Figure 19(a) shows that 50% of the moving objects do not generate more than 40 trajectories over 12 hours. Therefore, the IDT queries in TrajMesa and TMan are very fast, as shown in Figure 19(a).
Spatio-temporal Query. We evaluate $STRQ$ experiments on TDrive and Lorry, by randomly combining spatial and temporal ranges of Sections VI-B and VI-C. Figure 19(b) shows that: *TMan* and *TMan*-XZ outperform *TrajMesa* and *STH* by up to 6-10 times. *TrajMesa* generates time periods that intersect with the given time range, uses XZ+ index to get spatial index values, and combines them into query windows. However, TrajMesa has a time period (e.g., a week) that requires checking many irrelevant trajectories for a short time range. *STH* partitions trajectories by time slice and builds the spatial index in each partition. *STH* starts a MapReduce job

to filter trajectories in partitions that intersect with the given time range, incurring expensive job startup and disk I/O costs.



(a) Performance of IDTQ. (b) Performance of STRQ.

Fig. 19. The performance of IDT and STR Queries.

E. Similarity Queries

Threshold Similarity Query. We conduct experiments on Frechet, DTW, and Hausdorff similarity queries. Figure 20 shows the results. We have the following observations: (1) *TMan* is superior to other methods, surpassing them by up to an order of magnitude. It is because *TMan* employs a sophisticated index structure that is more meticulous than the indexes used in *TraSS*, *DFT*, and *DITA*; (2) Despite adopting the pruning strategies of *TraSS*, *TMan* surpasses *TraSS* in performance. This is because the TShape index can generate more detailed shapes than XZ*, and the index cache mechanism reduces the computation of unnecessary shapes.

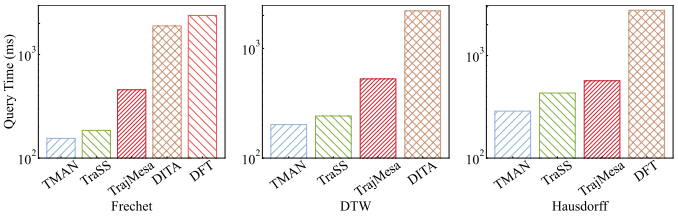


Fig. 20. Threshold Similarity Query (Lorry, $\theta = 0.015$).

Top-*k* Query. We conduct tok-k similarity search experiments. Figure 21 shows that *TMan* exhibits the best performance. In *DFT*, $c * k$ trajectories are selected from each intersecting partition to obtain a threshold. However, trajectories are always equipped with big MBRs, resulting in *DFT* getting a larger threshold that intersects numerous partitions. *REPOSE* applies RP-Trie (reference point trie) index to perform trajectory filtering. When the dataset has a large spatial span, *REPOSE* must build a large structure, which does not benefit from pruning. *DITA* needs to build a large index for Lorry dataset, which caused much time to search for candidates. Benefiting from TShape index and index cache mechanism, *TMan* can find top-k similar trajectories faster than *TraSS*.

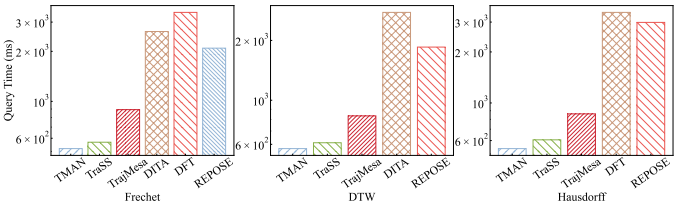
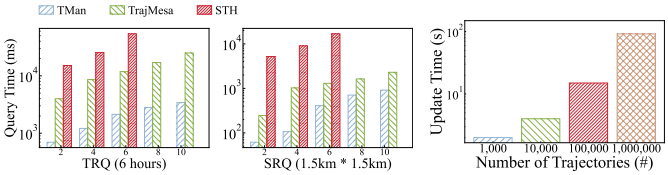


Fig. 21. Top-k Query (Lorry, $k = 50$).

F. Scalability

Data Size. This paper mainly introduces TRQ and SRQ, and we replicate Lorry data i times to evaluate the scalability of TRQ and SRQ. Figure 22(a) displays the results. The query time grows with data size because more trajectories must be processed. *STH* needs to build a large index structure and load much data from disks. Thus, *STH* performs worse than others, and *STH* encounters the out-of-memory issue when the data size is greater than Lorry-6. *TMan* is better than *TrajMesa*, and its advantage becomes more significant as the data grows.

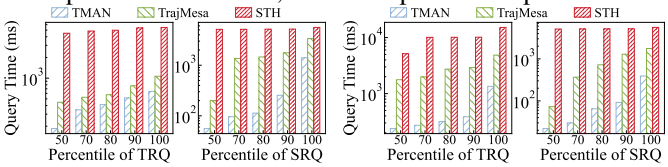
Update. As shown in Figure 22(b), we batch-insert new trajectories into an existing table to evaluate the performance of the update, which ensures the flexibility of TMan.



(a) Data Size (Lorry-x). (b) Update Time.

Fig. 22. Scalability and Update.

Tail Latency. The tail latency is presented by considering the 50th, 70th, 80th, 90th, and 100th percentiles of the query results. As shown in Figures 23(a)(b), query times vary a lot as the percentile increases, *TMan* keeps the best performance.



(a) Tail Latency (TDrive). (b) Tail Latency (Lorry).

Fig. 23. Tail Latency.

VII. Conclusions

This paper proposes TMan, a high-performance trajectory data management system based on key-value data stores. We subtly capture both temporal and spatial features of trajectories using the TR and the TShape indexes. TR Index devises a concise encoding for time ranges. TShape index utilizes non-rectangular index spaces to describe the complex shapes of trajectories. We devise an optimal encoding approach to index trajectory shapes. Based on the proposed indexes, we give a novel storage schema. To support various queries, we design an index cache mechanism, which enhances query performance by caching frequently accessed index and query results. Additionally, we develop an efficient query processing mechanism that takes advantage of the optimized indexes to provide fast and accurate query results. Especially for TRQ and SRQ, compared to the baseline, TMan can reduce false hits by an average of 77% and 83%, respectively. TMan is a part of the JD Urban Spatio-Temporal Data Platform and achieves significant results in real-world applications. Interesting future work includes 1) handling more query types; and 2) exploring alternative learnable methods to encode the shapes of the TShape index.

VIII. Acknowledgements

This work was supported by the National Key R&D Program of China(2019YFB2103201), the Beijing Science and Technology Project(Z211100004121008), the National Natural Science Foundation of China (72242106, 62172034, 62202070), and China Postdoctoral Science Foundation (2022M720567).

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] H. Lan, J. Xie, Z. Bao, F. Li, W. Tian, F. Wang, S. Wang, and A. Zhang, "Vre: a versatile, robust, and economical trajectory data system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3398–3410, 2022.

[12] H. He, R. Li, S. Ruan, T. He, J. Bao, T. Li, and Y. Zheng, "Trass: Efficient trajectory similarity search based on key-value data stores," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2306–2318.

[13] 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.

[14] 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.

[15] 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.

[16] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*. IEEE, 1996, pp. 441–448.

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

[18] L. Wang, Y. Zheng, X. Xie, and W.-Y. Ma, "A flexible spatio-temporal indexing scheme for large-scale gps track retrieval," in *The Ninth International Conference on Mobile Data Management (mdm 2008)*. IEEE, 2008, pp. 1–8.

[19] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 2010, pp. 109–120.

[20] 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.

[21] E. Zimányi, M. Sakr, and A. Lesuisse, "Mobilitydb: A mobility database based on postgresql and postgis," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 4, pp. 1–42, 2020.

[22] "Postgresql database," http://www.postgresql.org/, July 2023.

[23] "Postgis," http://postgis.net/, July 2023.

[24] 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.

[25] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on mapreduce," in *Proceedings of the first international workshop on Cloud data management*, 2009, pp. 9–16.

[26] 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.

[27] 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.

[28] 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.

[29] K. Liu, P. Tong, M. Li, Y. Wu, and J. Huang, "St4ml: Machine learning oriented spatio-temporal data processing at scale," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–28, 2023.

[30] 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.

[31] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 1071–1085.

[32] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: the geospark perspective and beyond," *GeoInformatica*, vol. 23, pp. 37–78, 2019.

[33] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 2016.

[34] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *2011 IEEE 12th International Conference on Mobile Data Management*, vol. 1. IEEE, 2011, pp. 7–16.

[35] J. K. Nidzwetzki and R. H. Güting, "Bboxdb-a scalable data store for multi-dimensional big data," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, pp. 1867–1870.

[36] N. Du, J. Zhan, M. Zhao, D. Xiao, and Y. Xie, "Spatio-temporal data index model of moving objects on fixed networks using hbase," in *2015 IEEE International Conference on Computational Intelligence & Communication Technology*. IEEE, 2015, pp. 247–251.

[37] X. Tang, B. Han, and H. Chen, "A hybrid index for multi-dimensional query in hbase," in *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*. IEEE, 2016, pp. 332–336.

[38] 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.

[39] 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.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[41] S. Mirjalili and S. Mirjalili, "Genetic algorithm," *Evolutionary Algorithms and Neural Networks: Theory and Applications*, pp. 43–55, 2019.

[42] R. Li, Z. Li, Y. Wu, C. Chen, and Y. Zheng, "Elf: Erasing-based lossless floating-point compression," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1763–1776, 2023.

[43] R. Li, Z. Li, Y. Wu, C. Chen, S. Guo, M. Zhang, and Y. Zheng, "Erasing-based lossless compression method for streaming floating-point time series," *arXiv preprint arXiv:2306.16053*, 2023.

[44] J. Dean, "Challenges in building large-scale information retrieval systems," in *Keynote of the 2nd ACM International Conference on Web Search and Data Mining (WSDM)*, vol. 10, no. 1498759.1498761, 2009.

[45] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Software: Practice and Experience*, vol. 40, no. 2, pp. 131–147, 2010.
[46] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 401–410.
[47] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 993–1008.
[48] T. Macedo and F. Oliveira, *Redis cookbook: Practical techniques for fast data manipulation*. " O'Reilly Media, Inc.", 2011.
[49] S. Maffeis, "Cache management algorithms for flexible filesystems," *SIGMETRICS Perform. Eval. Rev.*, vol. 21, no. 2, p. 16–25, dec 1993.
[50] K. Toohey and M. Duckham, "Trajectory similarity measures," *Sigspatial Special*, vol. 7, no. 1, pp. 43–50, 2015.
[51] 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.