TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System

Ruiyuan Li, *Member, IEEE*, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, *Member, IEEE*, Liang Hong, *Member, IEEE*, and Yu Zheng, *Fellow, IEEE*

Abstract—With the development of positioning technology, a large number of trajectories have been generated, which are very useful for many urban applications. However, it is challenging to manage trajectory data for its spatio-temporal dynamics and high-volume properties. Existing trajectory data management frameworks suffer from efficiency or scalability problem, and support only limited trajectory query types. This paper takes the first attempt to build a holistic distributed NoSQL trajectory query engine, named TrajMesa, based on GeoMesa, an open-source indexing toolkit for spatio-temporal data. TrajMesa can manage a prohibitively large number of trajectories, and support plenty of query types efficiently. Specifically, we first design a novel trajectory storage schema, which reduces the storage size tremendously. We then devise a novel indexing key schema for time ranges, based on which ID (i.e., moving object identifier) temporal query can be supported efficiently. To reduce the amount of retrieved trajectory data for a spatial range query, we propose a position code to indicate the spatial location of trajectories accurately. We also propose a bunch of pruning strategies for similarity query and k-NN query in the NoSQL environment. Extensive experiments are conducted using two real datasets and one synthetic dataset, verifying the powerful query efficiency and scalability of TrajMesa. The results show that TrajMesa is about $100 \sim 1000$ times faster than the state-of-the-art trajectory management frameworks in our experimental settings. TrajMesa is currently deployed in JD company, processing over 1T trajectories of JD Logistics every day.

Index Terms—Trajectory data management, distributed NoSQL storage, spatio-temporal indexing and query processing

1 Introduction

With the proliferation of positioning technology, a large number of trajectories have been generated. For example, DiDi, the largest rider-sharing company in China, produces over 15 billion location points of 70TB data size per day. As shown in Fig. 1, to utilize such huge trajectories, various trajectory queries have been proposed: 1) *ID Temporal Query*, which retrieves the trajectories of a given moving object within a specified time range, is used frequently in package tracking services. For example, users would use this type of query to check the status of their packages, i.e., where they are or when they will arrive. 2) *Spatial Range Query*, which finds the trajectories travelling through a given spatial range, can be used to discover reachable areas [1] (A reachable area is an area that can be reached from a given region within a specified time budget). In this case, we retrieve the trajectories that have passed the query region using spatial range query. The road segments covered by these trajectories form a reachable area. 3) *Similarity Query*, which returns all trajectories similar to a given trajectory, would help people to detect travelling companions [2] and ride sharing [3]. For example, taxi companies can use this type of query to find users with similar trajectories, and recommend ride sharing services to them. And 4) *k-NN (Nearest Neighbour) Query*, which finds k trajectories that are most similar to a given trajectory, can be a building block for trajectory clustering. For example, in the bike lane planning project [4], we first cluster trajectories based on k-NN query, then identify the road segments with many trajectories traversed.

It is desirable for a scalable and unified trajectory query engine to support all of these queries efficiently. Centralized solutions, e.g., TrajStore [5] and Torch [6], are based on a single machine, thus could not cope with such huge trajectories obviously. The distributed frameworks based on MapReduce, e.g., [7], [8], [9], are designed for massive trajectories, but they still face the efficiency problem, as they may incur multiple disk I/Os even for a single job. Most recently, many distributed in-memory trajectory data management frameworks, e.g, [10], [11], [12], [13], [14], [15], have emerged. However, they suffer from three limitations. First, these frameworks load all trajectories into memory.

- *Ruiyuan Li is with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: liruiyuan@cqu.edu.cn.*
- *Huajun He and Rubin Wang are with JD Intelligent Cities Research, Beijing 100101, China, and also with Southwest Jiaotong University, Chengdu 610032, China. E-mail: {hehuajun3, wangrubin3}@jd.com.*
- *Sijie Ruan is with JD Intelligent Cities Research, Beijing 100101, China, and also with Xidian University, Xi'an 710071, China. E-mail: ruansijie@jd.com.*
- *Tianfu He is with JD Intelligent Cities Research, Beijing 100101, China, and also with the Harbin Institute of Technology, Harbin 150001, China. E-mail: hetianfu3@jd.com.*
- *Jie Bao, Junbo Zhang, and Yu Zheng are with JD Intelligent Cities Research, Beijing 100101, China. E-mail: baojie@jd.com, {msjunbozhang, msyuzheng}@outlook.com.*
- *Liang Hong is with Wuhan University, Wuhan 430072, China. E-mail: hong@whu.edu.cn.*

Manuscript received 13 February 2020; revised 10 March 2021; accepted 26 April 2021. Date of publication 13 May 2021; date of current version 7 December 2022.
This work was supported in part by the National Key R&D Program of China under Grant 2019YFB2101801 and in part by the National Natural Science Foundation of China under Grant 61976168.
(Corresponding authors: Ruiyuan Li and Liang Hong.)
Recommended for acceptance by P. Pietzuch.
Digital Object Identifier no. 10.1109/TKDE.2021.3079880

1041-4347 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

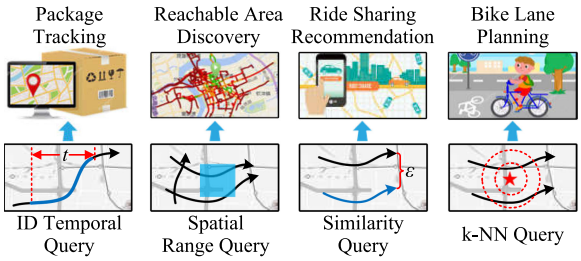


Fig. 1. Motivation for various trajectory queries.

They require high-performance clusters with much memory, hence their scalability is limited. Second, for each query request, they need to scan the big indexes in memory, which hurts query efficiency. Third, all of these frameworks only support limited trajectory query types, therefore cannot support sophisticated urban applications. Distributed NoSQL (Not Only SQL) data stores, such as Bigtable [16] and HBase, are suitable for real-time read/write random access to big data. However, due to lack of secondary indexes, these NoSQL data stores do not natively support spatio-temporal data management. GeoMesa [17] is an open-source tool that manages large-scale spatio-temporal data on the top of distributed NoSQL data stores. It transforms multi-dimensional information into one-dimensional key. However, GeoMesa cannot be applied to manage trajectories directly.

This paper is extended from our previous work [18]. To the best of our knowledge, we are the *first* to build a holistic distributed NoSQL trajectory query engine, *TrajMesa* [19], based on GeoMesa. TrajMesa has three notable characteristics: 1) *Scalability*. TrajMesa is based on distributed NoSQL data stores, thus it requires little for clusters. It can manage massive trajectories with limited cluster resources. 2) *Efficiency*. We carefully design a novel storage schema and a set of indexing techniques, thus it supports various trajectory queries efficiently. It is even $100 \sim 1000$ times faster than the advanced in-memory trajectory data management frameworks in our experimental settings. 3) *Plenty of queries support*. TrajMesa supports various widely used trajectory queries, including but not limited to: ID temporal query, spatial range query, similarity query and k-NN query.[1] The contributions of this paper are summarized as follows:

1) We take the first attempt to build a holistic distributed NoSQL trajectory query engine based on GeoMesa, in which a novel trajectory storage schema is designed. It not only reduces the storage size tremendously, but also supports various very useful trajectory queries efficiently.
2) We devise a novel indexing key schema for time ranges, based on which ID temporal query can be supported efficiently. To reduce the amount of retrieved trajectory data for spatial range query, we innovatively propose a position code to indicate the spatial location of trajectories accurately. We propose multiple pruning strategies for similarity query and k-NN query in the NoSQL environment.
3) Extensive experiments are conducted based on two real datasets and one synthetic dataset, which verifies the powerful query efficiency and scalability of TrajMesa.
4) An online demo system is deployed and publicly available in [19]. At present, TrajMesa is deployed in JD,[2] processing over 1T trajectories of JD Logistics every day.

Outline. We give the preliminaries in Section 2. The framework of TrajMesa is presented in Section 3. In Sections 4 and 5, we detail indexing & storing and query processing techniques of TrajMesa, respectively. We present the evaluation results in Section 6, followed by the related works in Section 7. Finally, we conclude this paper in Section 8.

2 Preliminary

This section gives related definitions, and introduces some knowledge of GeoMesa to help understand our designs.

2.1 Definition

Definition 1. (GPS Point) *A GPS point $p = (lat, lng, t)$ contains a latitude lat, a longitude lng, and a timestamp t.*

Definition 2. (Trajectory) *A trajectory $tr = \{p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n\}$ is a GPS point sequence ordered by timestamps.*

$tr.oid$ is the identifier of the moving object generating tr. $|tr|$ is the number of GPS points in tr. The minimum bounding rectangle (MBR) $tr.mbr$ is the smallest axis-aligned rectangle that contains all locations of GPS points in tr; $tr.p_s$ and $tr.p_e$ are the first and the last GPS point, respectively. Similarly, $tr.t_s$ and $tr.t_e$ are the start and end time, respectively.

Definition 3. (ID Temporal Query) *Given a trajectory dataset $\mathcal{T}$, a moving object identifier oid, a temporal range $R = [t_s, t_e]$, ID temporal query returns all trajectories $tr_i \in \mathcal{T}$, where $tr_i.oid = oid$, and there exists at least one GPS point p_j in tr_i that is generated during R. Formally,*

$$IDT\_query(\mathcal{T}, oid, R) = \{tr_i \in \mathcal{T} \mid tr_i.oid = oid \wedge \exists p_j \in tr_i, t_s \leq p_j.t \leq t_e\}. \quad (1)$$

The constraint "$\exists p_j \in tr_i$, $t_s \leq p_j.t \leq t_e$" can be rewritten as "$tr_i.t_s \leq t_e \wedge tr_i.t_e \geq t_s$". It means if there is at least one GPS point in a trajectory tr generated during $[t_s, t_e]$, then tr should be returned.

Definition 4. (Spatial Range Query) *Given a trajectory dataset $\mathcal{T}$, a spatial range $S = \{lat_{min}, lng_{min}, lat_{max}, lng_{max}\}$, spatial range query returns all trajectories $tr_i \in \mathcal{T}$, where there exists at least one GPS point p_j in tr_i that is located in S. Formally,*

$$SR\_query(\mathcal{T}, S) = \{tr_i \in \mathcal{T} \mid \exists p_j \in tr_i, lat_{min} \leq p_j.lat \leq lat_{max} \wedge lng_{min} \leq p_j.lng \leq lng_{max}\}. \quad (2)$$

Definition 5. (Similarity Query) *Given a trajectory dataset $\mathcal{T}$, a query trajectory q, a distance function f, a threshold ε, similarity query finds all trajectories $tr_i \in \mathcal{T}$, where the distance*

1. TrajMesa also supports other useful trajectory queries, which can be found in Appendix B on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2021.3079880

2. https://en.wikipedia.org/wiki/JD.com

between q and tr_i is not greater than ε. Formally,

$$Sim\_query(\mathcal{T}, q, f, \varepsilon) = \{tr_i \in \mathcal{T} \mid f(q, tr_i) \leq \varepsilon\}. \quad (3)$$

The distance function f measures the similarity of two trajectories. This paper focuses on one of the most widely used trajectory distance functions, i.e., Fréchet distance [20] f_F, which measures the minimum distance of all GPS point pairs between two trajectories meanwhile considers the GPS point order in a trajectory. Other euclidean space distance functions, such as Hausdorff distance [21] and DTW [22], are also supported by TrajMesa (See Appendix A), available in the online supplemental material.

Suppose there are two trajectories $Q = \langle q_1, q_2, \ldots, q_n \rangle$ and $tr = \langle p_1, p_2, \ldots, p_m \rangle$, Fréchet distance is defined as

$$f_F(Q, tr) = \begin{cases} \max_{1 \leq i \leq n} d(q_i, p_1) & m = 1 \\ \max_{1 \leq j \leq m} d(q_1, p_j) & n = 1 \\ \max\{d(q_n, p_m), \min\{f_F(Q^{n-1}, tr), & \\ \quad f_F(Q, tr^{m-1}), f_F(Q^{n-1}, tr^{m-1})\}\} & others \end{cases}, \quad (4)$$

where $d(q_i, p_j)$ is the euclidean distance between two GPS points q_i and p_j, and $Q^{n-1} = \langle q_1, q_2, \ldots, q_{n-1} \rangle$ and $tr^{m-1} = \langle p_1, p_2, \ldots, p_{m-1} \rangle$ are the sub-trajectories of Q and tr, respectively.

Definition 6. (k-NN Query) *Given a trajectory dataset $\mathcal{T}$, a query position or a query trajectory q, a positive integer k, a distance function f, k-NN query returns a set of trajectories $\mathcal{T}' \subseteq \mathcal{T}$, where $|\mathcal{T}'| = k$, and for each $tr_i \in \mathcal{T}'$, $tr_j \in \mathcal{T} \setminus \mathcal{T}'$, $f(q, tr_i) < f(q, tr_j)$. Formally,*

$$kNN\_query(\mathcal{T}, q, f, k) = \{tr_i \in \mathcal{T}' \mid \mathcal{T}' \subseteq \mathcal{T} \wedge |\mathcal{T}'| = k \wedge \forall tr_j \in \mathcal{T} \setminus \mathcal{T}', f(q, tr_i) < f(q, tr_j)\}. \quad (5)$$

If q is a trajectory, f can be Fréchet distance (or other euclidean space-based distance), and it is called k-NN trajectory query [11], [23]. If q is a point, f can be defined as Equation (6), and it is entitled k-NN point query [10], [23].

$$f_P(q, tr) = \min_{p_j \in tr} d(q, p_j), \quad (6)$$

where $d(q, p_j)$ is the euclidean distance between q and p_j.

2.2 GeoMesa

GeoMesa [17] is an open-source tool, which manages large-scale spatio-temporal data on the top of distributed NoSQL data stores. Its main idea is to transform multi-dimensional data into one-dimensional linear keys using space filling curves [24]. Essentially, GeoMesa stores multiple copies of data into different tables. With a carefully designed key, each table can support several types of queries efficiently. GeoMesa provides various indexing strategies with a common key combination

$$shard + feature(Optional) + id,$$

where "+" represents a concatenation operation, the same with the remaining equations when we refer to key combination; $shard$ is a random number to distribute data across region servers for load balance; $feature$ contains the spatial or spatio-temporal information that are extracted from a record; and id is the identifier of a record assigned by users or generated randomly as UUID (Universally Unique Identifier). Among these three parts, $shard$ and id are consistent in all strategies, but $feature$ is not the same. We introduce some indexing strategies related to this paper here.

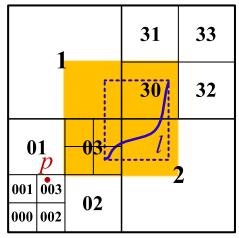


Fig. 2. Z2 and XZ2.

Z2 and XZ2 Indexing Strategies. Z2 and XZ2 are used to support spatial-related queries efficiently. Z2 is for point data, and its $feature$ is $Z(lng, lat)$, where $Z(*, *)$ is the Z-ordering [25] function to project two-dimensional geographical coordinates onto one-dimensional data. As shown in Fig. 2, this function partitions the spatial space into 4 sub-spaces of equal size, which are numbered from 0 to 3. Each sub-space is further partitioned recursively, until a certain resolution is achieved. The obtained number sequence represents the position of a point. For example, in Fig. 2, point p is transformed into "003".

XZ2 is for non-point data (e.g., polygons or lines) based on XZ-ordering [26], an extension of Z-ordering. Its $feature$ is $XZ2(lng_{min}, lat_{min}, lng_{max}, lat_{max})$. It first extracts the MBR of non-point data, whose left bottom corner determines a sub-space r, and the width and height decide a proper resolution. A proper resolution is that, the enlarged sub-space of r (i.e., we fix the left bottom corner of r, and double its width and height) *just* covers the data. As shown in Fig. 2, line l is projected as "03", as the enlarged sub-space of "03" (marked as orange) covers line l, but the enlarged sub-space of "032" cannot cover l.

Attribute Indexing Strategy. To speed up the queries by a given attribute value (e.g., query trajectories according to moving object identifiers), GeoMesa provides an attribute indexing strategy. Here, $feature$ consists of two parts

$$attrVal + 2ndTier,$$

where $attrVal$ is the value of an attribute, followed by a zero byte to mark its end. $2ndTier$ is a secondary index, which can be one of other index keys, e.g., Z2 or XZ2. For more details about the indexing strategies of GeoMesa, please refer to [17], [19], [26], [27].

3 Framework

Fig. 3 gives the framework of our proposed platform, TrajMesa, which consists of three main modules: *Preprocessing*, *Indexing & Storing*, and *Query Processing*.

Preprocessing. This module takes raw GPS points as input, and performs three main tasks: 1) noise filtering, which removes outlier GPS points that may be caused by the poor signal of positioning systems; 2) stay point detection, which identifies the locations where a moving object has stayed for a while within a certain distance threshold; and 3) segmentation,

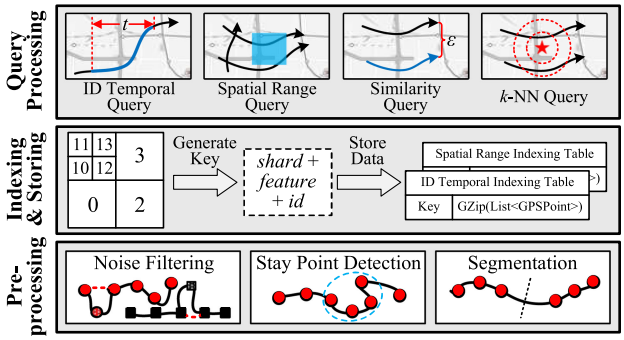


Fig. 3. Framework of TrajMesa.

which breaks a long trajectory into several meaningful short trajectories, such as multipe trips of taxies. Trajectory preprocessing is not only necessary for many urban applications, e.g., [4], [28], but also very important for the selection of the underlying storage schema and index building (see Section 4). As this paper mainly focuses on the indexing and query processing, please refer to our previous work [29] for more details about trajectory preprocesing.

Indexing & Storing. This module builds indexes for the preprocessed trajectories, and stores the trajectory data into the underlying data store of GeoMesa. Specifically, we generate two different keys that combine the spatio-temporal attributes and other necessary information of a trajectory. Each key and trajectory data forms a key-value pair, which is then stored into the data store of GeoMesa. In other words, we store two copies of a trajectory into two tables with different keys (detailed in Section 4).

Query Processing. With the help of built indexes, TrajMesa efficiently supports most useful trajectory queries, including: ID temporal query, spatial range query, similarity query, and k-NN query (detailed in Section 5).

4 Indexing And Storing

TrajMesa indexes and stores cleaned and segmented trajectories. In this section, we first discuss the selection of underlying storage schema, which is vital for index building and query processing. After that, we introduce the index techniques for various trajectory queries. Note that we essentially store two copies of trajectory data with different carefully designed keys in different tables, as the disk storage cost is much cheaper than the computing cost [30].

4.1 Storage Schema Selection

Vertical Storage Schema. One basic idea to store trajectories in a key-value store is that, the trajectory data is stored with each GPS point as one row, as most existing non-relational trajectory management systems did [29], [30], [31], [32]. We call this schema as vertical storage schema (*V-Store*). An example of V-Store is given in Fig. 4a, where the value of each point can be categorized into two parts:

(1) *Spatio-temporal properties*, which consists of the latitude lat, longitude lng, and time t of this GPS point. They are used to build the spatio-temporal indexes.

(2) *Other properties*, which includes the moving object identifier oid that generates this point, the trajectory id tid that this point belongs to, and other property readings.

V-Store regards each GPS point in a trajectory as an independent entity, which leads to several drawbacks. 1) To fetch a trajectory, we need to first retrieve all of its GPS points, then group them by tid, and sort each group by t. This procedure is time-consuming and slows down the query efficiency. 2) V-Store is unfit for trajectory queries, especially for similarity query and k-NN query, as we can hardly know the full information of a trajectory before we acquire all of its GPS points. 3) The number of rows is equal to the number of GPS points, which results in prohibitively numerous key-value entries. More key-value entries need more disk storage space, which triggers more disk I/Os when retrieving the same number of trajectories. This further hurts the query efficiency.

Horizontal Storage Schema. To address the aforementioned issues, we propose a novel horizontal storage schema, i.e., *H-Store*, to store each trajectory in a single row. As shown in Fig. 4b, the value of each entry contains four parts:

(1) *Spatio-temporal properties*, which includes the MBR mbr, the start and end time t_s and t_e, and the start and end positions p_s and p_e of a trajectory.

(2) *GPS point list*. The GPS points in a trajectory are first serialized using Kryo[3] (a fast serializer that transforms data into bytes, which is necessary for the following compression), and then compressed with GZip[4] (a popular compressor that achieves a good balance of compression ratio and efficiency). This not only reduces the storage size tremendously, but also improves the efficiency of storing and querying by reducing disk I/Os.[5]

(3) *Signature*. In most scenarios, a trajectory only locates in a very small part of its MBR. That is to say, the MBR of a trajectory cannot represent its position exactly. To this end, we design a signature, which provides finer-grained information of the trajectory location. As shown in Fig. 4c, the MBR of a trajectory is divided into $\alpha \times \alpha$ disjoint regions (i.e., *signature regions*) with equal size, and each region is numbered. The signature is a binary sequence of $\alpha \times \alpha$ bits. If one or more GPS points of the trajectory are located in a region, the corresponding bit is set to 1, otherwise set to 0. A bigger α means a finer representation, but it requires more storage space and more query complexity. Fig. 4c gives an example of signature with $\alpha = 4$.

(4) *Other properties*. Like V-Store, we store the moving object id oid, trajectory id tid, and other related properties.

Discussion. Most NoSQL data stores limit the maximum storage size for each row by default. As the GPS point list of a segmented trajectory would not be too long (this is one of the reasons that we need the trajectory preprocessing procedure before indexing & storing), it should be fit in a row in most cases. Otherwise, TrajMesa would throw an exception. One can also change or remove the size limitation. For example, the limitation of HBase is set 10M by default, and this limitation is configurable by `hbase.client.keyvalue.maxsize`.

In the following, we will elaborate the design of keys for each indexing table using H-Store in TrajMesa.

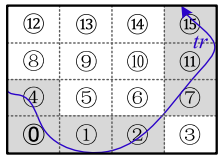
3. https://github.com/EsotericSoftware/kryo
4. https://www.gzip.org/
5. More details about the selection of compression and serialization methods can be found in Appendix C, available in the online supplemental material.

Key	Value	
Point Key	Spatio-temporal Properties	Other Properties
p_1	lat_1 lng_1 t_1	oid_1 tid_1 ⋯
p_2	lat_2 lng_2 t_2	oid_2 tid_2 ⋯
⋯	⋯	⋯
p_n	lat_n lng_n t_n	oid_n tid_n ⋯

(a) Vertical Storage Schema

Key	Value			
Traj. Key	Spatio-temporal Properties	GPS Point List	Signature	Other Properties
$traj_1$	mbr_1 t_{s1} t_{e1} p_{s1} p_{e1}	gzip(kryo(List<GPSPoint>))	00010101	oid_1 tid_1 ⋯
$traj_2$	mbr_2 t_{s2} t_{e2} p_{s2} p_{e2}	gzip(kryo(List<GPSPoint>))	10011001	oid_2 tid_2 ⋯
⋯	⋯	⋯	⋯	⋯
$traj_m$	mbr_m t_{sm} t_{em} p_{sm} p_{em}	gzip(kryo(List<GPSPoint>))	10001001	oid_m tid_m ⋯

(b) Horizontal Storage Schema



(c) Trajectory Signature

Fig. 4. Storage schema selection.

4.2 ID Temporal Indexing

Main Idea. To efficiently support ID temporal query, TrajMesa stores a copy of trajectory data with the key designed based on attribute indexing strategy. The main idea is to keep the trajectory data of the same moving object at adjacent time together as much as possible.

Challenges. Recall that the key of attribute indexing strategy is $shard + attrVal + 2ndTier + id$, where $attrVal$ can be set as the moving object id oid, and id is set as the trajectory id tid. To support a temporal paradigm, one may simply replace $2ndTier$ by the start time t_s (or end time t_e) of a trajectory. However, there could be some results missing if we only encode t_s (or t_e). As shown in Fig. 5a, suppose the time range of a query q is $[tq_s, tq_e]$, when $t_s < tq_s$, a qualified trajectory will be overlooked. Note that we retrieve trajectories if only *part* of their GPS points located in the given temporal range. One of the challenges is how to index time ranges using one-dimensional keys.

Solution. Although interval tree [33] allows to efficiently find all intervals that overlap with a given interval, however, it requires to maintain a tree structure, which is not fit for the NoSQL environment. XZ2 indexing strategy [26] projects spatial ranges onto one-dimensional keys to support spatial range queries. Inspired by XZ2, we propose *XZT* (eXtended Z-ordering for Time range) to support time range queries by projecting time ranges onto one-dimensional keys. The main idea of XZT is to find an *ordered and unique* key for each trajectory time span. Before drilling in the details, we first give some related definitions.

Definition 7. (Element) *A time range $E = [t_s, t_e]$ is called an element, whose length is defined as $\Delta t = t_e - t_s$.*

Definition 8. (XElement) *The extended element (XElement) of $E = [t_s, t_e]$ is $XE = [t_s, t_e + \Delta t]$, where $\Delta t = t_e - t_s$. Note that the length of XE is twice of the length of E.*

As there is no limit for the time dimension, we divide the time line into disjoint time period bins (e.g., one day or one year). For each bin, we encode the time ranges whose start time locates in this bin. Specifically, as shown in Fig. 5b, the key of element $E = [t_s, t_e]$ consists of two parts:

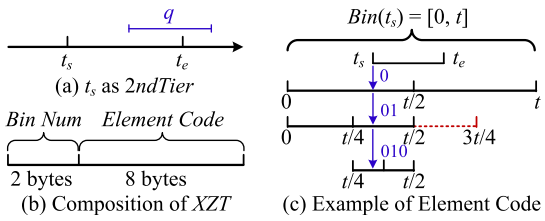


Fig. 5. Techniques of ID temporal indexing.

(1) *Bin Num*. It indicates which time period bin that t_s locates in, defined by Equation (7).

$$Bin(t_s) = \lfloor (t_s - RefTime) \div BinLen \rfloor, \tag{7}$$

where t_s is the start time of E, $RefTime$ is the reference time (e.g., 1970-01-01T00:00:00Z), and $BinLen$ is the number of seconds in a time period bin. We use two bytes to store the *Bin Num*, as it can represent at least $2^{16}/365 \approx 180$ years when $BinLen$ is 1 day, which satisfies most cases.

(2) *Element Code*. It represents the offset of E in its time period bin, denoted by $C(E)$. There are two steps to get the element code: *Sequence Calculation* and *Code Generation*.

• *Sequence Calculation*. This step gets a binary sequence $S = \langle s_0, s_1, \ldots, s_{l-1} \rangle$ in a way similar to binary search. As shown in Fig. 5c, we *recursively* find a line segment $L = [tl_s, tl_e]$ in bin $Bin(t_s) = [tb_s, tb_e]$ to represent $E = [t_s, t_e]$. If t_s locates in the left half part of the search space, we append 0 to S; otherwise, we append 1. This procedure is terminated when at least one of the following conditions is not met.

$$\begin{cases} tl_s \le t_s \wedge tl_e + \Delta t \ge t_e & (I) \\ |S| < g & (II) \end{cases}. \tag{8}$$

Condition (I) guarantees that the XElement of L, i.e., $[tl_s, tl_e + \Delta t]$, *just* fully contains the time range E (i.e., if we further equally split L into two line segments L_1 and L_2, neither of the XElements of L_1 and L_2 can fully contain E). Condition (II) means that the length of S is not greater than a user specified constant g, which avoids an overlong sequence. Algorithm 1 gives the pseudo-code, which is self-explanatory. Note that we may append an extra bit to S, as the last line segment may do not meet condition (I). We should remove it as shown in Line 8-9.

Algorithm 1. Sequence Calculation

Input: Element $E = [t_s, t_e]$ to be indexed, time period bin $Bin(t_s) = [tb_s, tb_e]$, max sequence length g.
Output: Sequence $S = \langle s_0, s_1, \ldots, s_{l-1} \rangle$.

```
1:  S = ⟨⟩; tl_s = tb_s; tl_e = tb_e; t_c = (tl_s + tl_e)/2;
2:  while (tl_s ≤ t_s ∧ tl_e + Δt ≥ t_e) ∧ |S| < g do
3:      if t_s < t_c then
4:          S.append(0); tl_e = t_c;
5:      else
6:          S.append(1); tl_s = t_c;
7:      t_c = (tl_s + tl_e)/2;
    // The last may do not meet condition (I)
8:  if tl_s > t_s ∨ tl_e + Δt < t_e then
9:      S.removeLast();
10: return S;
```

$$shard + oid + \overbrace{BinNum(2\text{ bytes}) + ElementCode(8\text{ bytes})}^{XZT} + tid$$

Fig. 6. Key of ID indexing table.

Example. As shown in Fig. 5c, suppose $Bin(t_s) = [0, t]$, the XElement of line segment $[t/4, t/2]$ (i.e., $[t/4, 3t/4]$) contains $[t_s, t_e]$, but the XElement of $[t/4, 3t/8]$ (i.e., $[t/4, t/2]$) does not. Hence, the sequence of $[t_s, t_e]$ is "01".

• *Code Generation.* This step generates a long integer code from the binary sequence, according to Equation (9). It can be regarded as a process of converting binary to decimal.

$$C(S) = \sum_{i=0}^{l-1} s_i \times (2^{g-i} - 1) + 1. \tag{9}$$

Discussion. XZT requires that, the time range E to be indexed should be within the XElement of a line segment in a time period bin. To this end, XZT could not index a time range whose length is greater than the bin. If so, we cannot find a sub-time range in a bin whose XElement fully contains E, and TrajMesa would throw an exception. Fortunately, as most segmented trajectories would be no longer than one day (this is another reason that we preprocess trajectories), we can easily select an appropriate time period.

Summary. In summary, the key of ID Temporal Indexing Table is shown as Fig. 6.

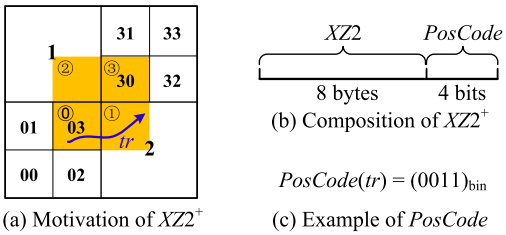
4.3 Spatial Range Indexing

Main Idea. We build Spatial Range Indexing Table for spatial range query, similarity query and k-NN query based on XZ2 indexing strategy, which encodes the spatial information into a linear key. The main idea is to store spatially close trajectories together as much as possible.

Challenges. Recall that the key in XZ2 indexing strategy is $shard + XZ2 + id$, where id can be set as the trajectory id tid. However, $XZ2$ cannot represent the spatial location of trajectories exactly. As shown in Fig. 7a, XZ2 indexing strategy uses 03 to represent the trajectory tr, because the XElement of 03 (marked orange, the concepts are borrowed from Definitions 7 and 8 but with two dimensions in this case) just covers tr. However, the XElement of 03 is too big for tr, as tr only crosses a small portion of it. To this end, it is necessary to design a more accurate encoding method to indicate the spatial location of trajectories.

Solution. This paper proposes $XZ2^+$ to hash out the aforementioned issue. As shown in Fig. 7b, the keys generated by $XZ2^+$ consist of two parts:

(1) $XZ2$. This part is generated based on XZ-ordering [26]. It is a long integer which is converted from the non-point



(a) Motivation of $XZ2^+$ (b) Composition of $XZ2^+$ (c) Example of *PosCode*

Fig. 7. Techniques of $XZ2^+$.

$$shard + \overbrace{XZ2(8\text{ bytes}) + PosCode(4\text{ bits})}^{XZ2^+} + tid$$

Fig. 8. Key of spatial range indexing table.

data. In Fig. 7a, the $XZ2$ is 03 (note that this is a quaternary number).

(2) *PosCode* (position code). We divide the XElement into $\beta \times \beta$ disjoint areas of equal size, and each area is numbered. Then the spatial location of a trajectory tr is indicated by $\beta \times \beta$ bits. If at least one GPS point of tr locates in an area, the corresponding bit is set 1, otherwise set 0. Fig. 7c gives the *PosCode* of tr in Fig. 7a with $\beta = 2$.

Discussion. Note *PosCode* is not the same as the trajectory signature. Trajectory signature is based on the MBR of a trajectory, but *PosCode* is based on the XElement. Here, a bigger β means a more accurate spatial location representation, but it results in more storage space and may damage the query efficiency (see Section 5 for details). In our implementation, we set $\beta = 2$.

Summary. In summary, the key of Spatial Range Indexing Table is shown as Fig. 8.

5 Query Processing

With the carefully designed indexes, TrajMesa efficiently supports various useful queries. Most of these queries follow a common three steps: 1) *query window generation*, which generates multiple query windows by given query paradigms; 2) *query execution*, which executes trajectory queries in parallel; 3) *result refinement*, which removes unsatisfied or duplicated trajectories and returns final results.

5.1 ID Temporal Query

Query Window Generation. Recall that the key of ID Temporal Indexing Table is $shard + oid + BinNum + ElementCode + tid$. Given an ID temporal query q with a time range $[tq_s, tq_e]$ and a moving object id oid, this step is further divided into five substeps:

(1) $shard$ generation, which enumerates all possible values of $shard$.

(2) oid generation, which combines the query moving object id with a zero byte that marks the end of object id. It is obviously unique when an ID temporal query is given.

(3) $BinNum$ generation, which finds a list of time period bins whose XElement are overlapped with $[tq_s, tq_e]$. We calculate the bins where tq_s and tq_e locate according to Equation (7). Suppose $b_m = Bin(tq_s)$ and $b_n = Bin(tq_e)$, then the bins b_i, $m - 1 \leq i \leq n$, are selected. Note that bin b_{m-1} is also qualified, because there could be trajectories whose time spans are overlapped with $[tq_s, tq_e]$, e.g., the trajectory tr shown in Fig. 9. The trajectories in other bins would not be qualified.

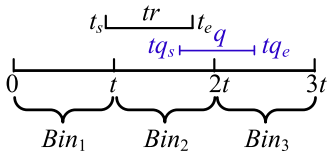
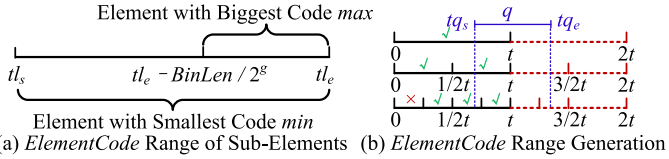


Fig. 9. Bin selection.



(a) *ElementCode* Range of Sub-Elements (b) *ElementCode* Range Generation

Fig. 10. Code range of sub-elements.

(4) *ElementCode* generation. Essentially, we find an element $[tl_s, tl_e]$ to represent the time range $[t_s, t_e]$ of a trajectory during the process of indexing. In this step, for each qualified element $[tl_s, tl_e]$ in a time bin, we calculate the element code range $[min, max]$ of all its sub-elements, where $min = C([tl_s, tl_e])$ and $max = C([tl_e - BinLen/2^g, tl_e])$, as shown in Fig. 10a. Here, g is the max sequence length, and $BinLen/2^g$ is the time span of an element with a sequence length of g. This step is to find all elements in a time bin that overlap with the query time range $[tq_s, tq_e]$.

Algorithm 2 presents the pseudo-code of *ElementCode* generation, which consists of three main parts:

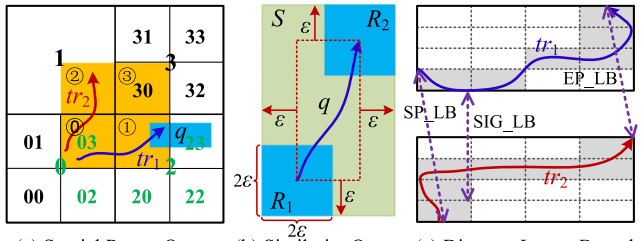
• *Initialization* (Line 1). R records the qualified element code ranges, and l records the sequence length of the elements to be checked (it also represents check level). We use a first-in-first-out queue que to help the check process in a breath-first order. *Flag* represents the end of a level.

Algorithm 2. *ElementCode* Generation in a Bin

Input: Query time range $[tq_s, tq_e]$, $Bin(tq_s) = [tb_s, tb_e]$.
Output: A list of *ElementCode* ranges $\mathcal{R}$.

1: $\mathcal{R} = \emptyset$; $l = 0$; $que.push([tb_s, tb_e])$; $que.push(Flag)$;
2: **while** $l < g \wedge que \neq \emptyset$ **do**
3: $cur = que.pop()$;
4: **if** $cur = Flag$ **then**
5: $l = l + 1$; $que.push(Flag)$; **continue**;
6: $[tl_s, tl_e] = cur$; $\Delta t = tl_e - tl_s$;
// `XElement of` cur `is contained`
7: **if** $tl_s \geq tq_s \wedge tl_e + \Delta t \leq tq_e$ **then**
8: $\mathcal{R}.add(CodeRange(tl_s, tl_e, false))$;
// `XElement of` cur `is overlapped`
9: **else if** $tl_s \leq tq_e \wedge tl_e + \Delta t \geq tq_s$ **then**
10: $\mathcal{R}.add(CodeRange(tl_s, tl_e, true))$;
11: $tl_c = (tl_s + tl_e)/2$;
12: $que.push([tl_s, tl_c])$; $que.push([tl_c, tl_e])$;
// `Processing remaining elements in` que
13: **while** $que \neq \emptyset$ **do**
14: $cur = que.pop()$;
15: **if** $cur \neq Flag$ **then**
16: $[tl_s, tl_e] = cur$;
17: $\mathcal{R}.add(CodeRange(tl_s, tl_e, false))$;
18: **return** $\mathcal{R}$;

• *Recursive Check* (Line 2-12). For each element cur in que, we check the relation of its XElement $Xcur$ with $[tq_s, tq_e]$. There are three cases: 1) If $Xcur$ is contained in $[tq_s, tq_e]$, we get a code range that represents all sub-elements of cur, and add it to $\mathcal{R}$ (Line 7-8); 2) If $Xcur$ is overlapped with $[tq_s, tq_e]$, we add the code that exactly stands for cur to $\mathcal{R}$. Besides, we add the two children of cur to que for further test (Line 9-12); 3) If $Xcur$ does not intersect with $[tq_s, tq_e]$, we do nothing. When the maximum resolution is reached or que is empty, the recursive check process is terminated.



(a) Spatial Range Query (b) Similarity Query (c) Distance Lower Bound

Fig. 11. Spatial range and similarity queries.

• *Remaining Check* (Line 13-17). We process the remaining elements in que if it is not empty (Line 13-17).

Function *CodeRange* (Algorithm 3) returns the code range of $[tl_s, tl_e]$. $part$ indicates whether it returns a code range of all sub-elements or that of the exact $[tl_s, tl_e]$.

Algorithm 3. Function $CodeRange(tl_s, tl_e, part)$

1: $min = C([tl_s, tl_e])$;
2: **if** $part = true$ **then**
3: $max = min$;
4: **else**
5: $max = C([tl_e - BinLen/2^g, tl_e])$;
6: **return** $[min, max]$;

Example. Fig. 10b gives an example of code range generation, where $Bin(tq_s) = [0, t]$ and $g = 2$. The qualified elements (whose XElement is contained in or overlapped with the query time range) are checked.

(5) Query window combination. We combine $shard$, oid, $BinNum$, and code ranges into query windows, whose number is $N^{shard} \times \sum_{i=0}^{N^{bin}-1} N_i^{range}$. N^{shard} and N^{bin} are the number of $shard$ values and qualified bins, respectively, and N_i^{range} is the number of code ranges in bin b_i.

Query Execution. TrajMesa triggers `SCAN` operations over the underlying data store in parallel, where each query window is transformed into an execution.

Result Refinement. Due to the limitation of max sequence length, there could be unqualified trajectories retrieved. Consequently, when all `SCAN` operations are finished, we collect the results and remove the unsatisfied trajectories.

5.2 Spatial Range Query

Query Window Generation. Spatial range query is based on Spatial Range Indexing table, whose keys follow a pattern of $shard + XZ2 + PosCode + tid$. The query window generation step is further divided into four substeps:

(1) $shard$ generation, the same with ID temporal query.

(2) Spatial key range generation. This step generates a list of key ranges by the given spatial range query q, which is similar to Algorithm 2 but extended to two dimensions [26]. As shown in Fig. 11a, we recursively check the XElement Xe of an element e in a top-down fashion. If Xe is contained in q, we generate a key range that includes all elements in e. If Xe overlapped with q, we generate a key range that exactly represents e, and further check the four children of e until the max resolution is reached. If Xe does not intersect with q, we do nothing. All qualified elements (whose

XElement is contained in or overlapped with q) in Fig. 11a are marked green.

(3) *PosCode* generation. As shown in Fig. 11a, tr_2 is not overlapped with q, thus it should not be retrieved. *PosCode* is proposed to address this issue. As mentioned in Section 4.3, we divide an XElement into $\beta \times \beta$ disjoint areas, which can be represented by a sequence of $\beta \times \beta$ bits B. For each XElement of a qualified element e, if q intersects with the ith area, then $B[i] = 1$, otherwise $B[i] = 0$. A *PosCode* of element e must satisfy Equation (10)

$$PosCode(e)\&B \neq 0, \tag{10}$$

where $\&$ means bitwise AND operation. For each qualified e, we generate all of its satisfied *PosCode*s, whose number is between $2^{\beta\times\beta-1}$ (if q intersects with only one area) and $2^{\beta\times\beta} - 1$ (if q intersects with all areas). To this end, β should not be set too large ($\beta = 2$ in our implementation).

Example. As shown in Fig. 11a, q intersects with the second area of the XElement of 03, thus $B = 0010$. We enumerate all numbers from 0000 to 1111, finding the following 8 *PosCode*s that satisfy Equation (10): $0010, 0011, 0110, 0111, 1010, 1011, 1110$, and 1111.

(4) Query window combination. It combines *shard*, key ranges, and *PosCode*s into query windows, whose number is $N^{shard} \times \sum_{i=0}^{N^e-1} N_i^{PosCode}$, where N^{shard} and N^e are the number of *shard* values and qualified elements, respectively, and $N_i^{PosCode}$ is the number of *PosCode*s of e_i.

Query Execution. Similar to ID temporal query, but we use the Spatial Range Indexing table in this case.

Result Refinement. Similar to ID temporal query, but we refine trajectories by the given spatial range.

5.3 Similarity Query

Similarity query follows a filtering-refinement framework. In filtering step, we get a candidate trajectory set based on spatial range queries. In refinement step, we check the real similarity, and get the final result.

Trajectory Filtering. Similarity query regards spatial range query as a building block. As shown in Fig. 11b, given a query trajectory q with a distance threshold ε (we would transform ε from km to coordinate degree), we get two spatial ranges $R_1 = \{lat_1 - \varepsilon, lng_1 - \varepsilon, lat_1 + \varepsilon, lng_1 + \varepsilon\}$ and $R_2 = \{lat_n - \varepsilon, lng_n - \varepsilon, lat_n + \varepsilon, lng_n + \varepsilon\}$, where (lat_1, lng_1) and (lat_n, lng_n) are the start and end points of q, respectively. All similar trajectories should be contained in the spatial range query result $\mathcal{T}' = SR\_query(\mathcal{T}, R_1) \cap SR\_query\ (\mathcal{T}, R_2)$ in terms of Fréchet distance f_F.

Lemma 1. *All similar trajectories of q are in $\mathcal{T}'$ in terms of Fréchet distance f_F.*

Proof. We first prove that all similar trajectories of q are contained in $\mathcal{T}_1 = SR\_query(\mathcal{T}, R_1)$. Suppose $\exists tr$, $f_F(tr, q) \leq \varepsilon$, but $tr \notin \mathcal{T}_1$. As $tr \notin \mathcal{T}_1$, all points of tr are out of R_1. Consequently, $f_F(tr, q) \geq \min_{p_i \in tr} d(p_i, q.p_s) > \varepsilon$, which conflicts with our hypothesis.

Similarly, we can prove that all similar trajectories of q are contained in $\mathcal{T}_2 = SR\_query(\mathcal{T}, R_2)$. To this end, all similar trajectories of q are contained in $\mathcal{T}' = \mathcal{T}_1 \cap \mathcal{T}_2$. □

Result Refinement. After retrieving a candidate trajectory set $\mathcal{T}'$ using spatial range query, it requires to refine it by checking whether $f(tr, q) \leq \varepsilon$ for all $tr \in \mathcal{T}'$. However, the complexity of f_F is $\mathcal{O}(|tr| \times |q|)$, which is time-consuming. Therefore, this paper proposes three types of pruning strategies, all of which can be calculated in $\mathcal{O}(1)$.

(1) *MBR Pruning*. Given a query trajectory q with a threshold ε, where $q.mbr = \{lat_{min}, lng_{min}, lat_{max}, lng_{max}\}$, we get a spatial range $S = \{lat_{min} - \varepsilon, lng_{min} - \varepsilon, lat_{max} + \varepsilon, lng_{max} + \varepsilon\}$. The MBRs of all similar trajectories should be fully contained in S according to Lemma 2.

Lemma 2. *If the MBR of tr is not fully contained in S, i.e., there is at least one GPS point p of tr falling outside S, then tr would not be similar to q in terms of Fréchet distance f_F.*

Proof. Suppose $\exists tr$, $f_F(tr, q) \leq \varepsilon$, but $\exists p \in tr$, p is out of S. As p is out of S, $f_F(tr, q) \geq \min_{q_i \in q} d(p, q_i) > \varepsilon$, i.e., tr is not similar to q, which conflicts with our hypothesis. □

(2) *SEP_LB*. We propose a lower bound based on the start and end points of two trajectories.

$$SEP\_LB_{f_F}(q, tr) = \max\{d(q.p_s, tr.p_s), d(q.p_e, tr.p_e)\}. \tag{11}$$

Lemma 3. *If $SEP\_LB_{f_F}(q, tr) > \varepsilon$, then $f_F(q, tr) > \varepsilon$.*

Proof. $f_F(q, tr) \geq \max\{d(q.p_s, tr.p_s),\ d(q.p_e, tr.p_e)\} = SEP\_LB_{f_F}(q, tr) > \varepsilon$. □

(3) *SIG_LB*. As introduced in Section 4.1, trajectory signatures indicate the finer information of trajectory locations, based on which a signature lower bound is proposed.

$$SIG\_LB_{f_F}(q, tr) = \max\{\max_{r_q \in Sig(q)} \min_{r_{tr} \in Sig(tr)} d(r_q, r_{tr}), \max_{r_{tr} \in Sig(tr)} \min_{r_q \in Sig(q)} d(r_{tr}, r_q)\}, \tag{12}$$

where $r_q \in Sig(q)$ is a signature region of trajectory q, $d(r_q, r_{tr})$ is the region distance between r_q and r_{tr}, which is calculated by Equation (13).

$$d(r_q, r_{tr}) = \min_{p_i \in r_q, p_j \in r_{tr}} d(p_i, p_j). \tag{13}$$

We also define the distance between a point p and a region r as the minimum distance between p and any point $p' \in r$.

$$d(p, r) = \min_{p' \in r} d(p, p'). \tag{14}$$

Lemma 4. *If $SIG\_LB_{f_F}(q, tr) > \varepsilon$, then $f_F(q, tr) > \varepsilon$.*

Proof. For points $q_i \in q$ and $p_j \in tr$, they must locate in signature regions r_q^i and r_{tr}^j, respectively. So we have

$$f_F(q, tr) \geq \max\left\{\max_{q_i \in q} \min_{p_j \in tr} d(q_i, p_j), \max_{p_j \in tr} \min_{q_i \in q} d(p_j, q_i)\right\} \geq$$
$$\max\left\{\max_{q_i \in q} \min_{r_{tr} \in Sig(tr)} d(q_i, r_{tr}), \max_{p_j \in tr} \min_{r_q \in Sig(q)} d(p_j, r_q)\right\} \geq$$
$$\max\left\{\max_{r_q \in Sig(q)} \min_{r_{tr} \in Sig(tr)} d(r_q, r_{tr}), \max_{r_{tr} \in Sig(tr)} \min_{r_q \in Sig(q)} d(r_{tr}, r_q)\right\}$$
$$= SIG\_LB_{f_F}(q, tr) > \varepsilon.$$ □

Algorithm 4. k-NN Point Query

Input: Query point q, result count k, max resolution g.
Output: A set of trajectories $\mathcal{T}_{knn}$.

1: Initial a priority queue cdq with a max size k, whose elements tr are ordered by $f_P(q, tr)$ of Equ. (6);
2: Initial a priority queue req with whole spatial region, whose elements r are ordered by $d(q, r)$ of Equ. (14);
3: $d_{max} = 0$; $checked = \emptyset$;
4: **while** *req is not empty* **do**
5: $r = req.pop()$;
6: **if** $cdq.size() = k \wedge d(q, r) > d_{max}$ **then**
7: **break**; `// Pruning I: Region Pruning`
8: **if** *the resolution of* $r < g$ **then**
9: Add the four children of r to req; **continue** ;
10: $\mathcal{T}_{SR} = SR\_query(\mathcal{T}, r)$;
11: **foreach** $tr \in \mathcal{T}_{SR}$ **do**
12: **if** $tr.tid$ *in checked* **then**
13: **continue**;
14: $checked = checked \cup \{tr.id\}$;
15: **if** $cdq.size() = k \wedge LB_{f_P}(q, tr) > d_{max}$ **then**
16: **continue**; `// Pruning II: LB Pruning`
17: Add tr to cdq; $d_{max} = f_P(q, cdq.last())$;
18: **return** cdq as $\mathcal{T}_{knn}$;

The calculation time complexity of SIG_LB is $\mathcal{O}(\alpha^2)$, but $\alpha \ll |q|$ and $\alpha \ll |tr|$. We set $\alpha = 4$ in our implementation.

If one of above lower bounds is greater than ε, we can safely stop the calculation of trajectory distance, which accelerates similarity queries tremendously.

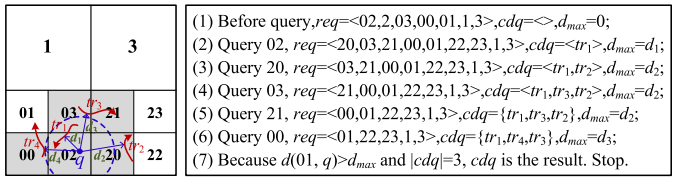
5.4 k-NN Query

k-NN query is categorized into k-NN point query [10] and k-NN trajectory query [11]. We first elaborate k-NN point query in TrajMesa, then extend it to k-NN trajectory query.

k-NN Point Query. The main idea of k-NN point query is to iteratively expand the query spatial range in an inner-outer fashion, until the k most similar trajectories are found. We propose two pruning strategies to stop the expansion process as early as possible. Algorithm 4 presents k-NN point query, which contains two steps:

(1) *Initialization* (Line 1-3). Here, cdq is a priority queue that stores candidate trajectories; req is another priority queue to record the regions to be queried; d_{max} is the currently maximum distance between q and the trajectories in cdq; and $checked$ records the trajectory IDs that we have already checked, which avoids redundant computation.

(2) *Expansion* (Line 4-17). This step pops a region r from req. If there are k trajectories in cdq and the distance between q and r is greater than d_{max}, the query process is terminated (Lemma 5, denoted as *Region Pruning*). If the resolution of r is smaller than g, we add its children to req and continue to check next region. Otherwise, we trigger a spatial range query by r. For each trajectory $tr \in \mathcal{T}_{SR}$, before calculating its real distance to q (which is time-consuming), we first check if it has been checked in an "inner" iteration. If the answer is "yes", we simply omit it. Otherwise, we add it to $checked$, and use a lower bound pruning strategy (Lemmas 6 and 7, denoted as *LB Pruning*). If tr is satisfied with all lower bounds, we add it to cdq, and update d_{max}.



(a) 3-NN query (b) Critical Steps of 3-NN query processing

Fig. 12. Example of k-NN Query ($k = 3$ and $g = 2$).

Lemma 5. *If $d(q, r) > d_{max}$, then $f_P(q, tr) > d_{max}$, where tr is any trajectory queried by r but not retrieved before.*

Proof. If tr has already been retrieved as it crosses an "inner" region, it should have been checked. If tr has not been retrieved, we have: $f_P(q, tr) = \min_{p_j \in tr} d(q, p_j) \geq \min_{p_j \in tr.mbr} d(q, p_j) \geq \min_{p_j \in r} d(q, p_j) = d(q, r) > d_{max}$. □

Lemma 6. *If $MBR\_LB_{f_P}(q, tr) = d(q, tr.mbr) > d_{max}$, then $f_P(q, tr) > d_{max}$.*

Proof. $f_P(q, tr) = \min_{p_j \in tr} d(q, p_j) \geq \min_{p_j \in tr.mbr} d(q, p_j) = d(q, tr.mbr) > d_{max}$. □

Lemma 7. *If $SIG\_LB_{f_P}(q, tr) = \min_{r_{tr} \in Sig(tr)} d(q, r_{tr}) > d_{max}$, then $f_P(q, tr) > d_{max}$.*

Proof. $f_P(q, tr) = \min_{p_j \in tr} d(q, p_j) \geq \min_{p_j \in tr.mbr} d(q, p_j) \geq \min_{r_{tr} \in Sig(tr)} d(q, r_{tr}) = SIG\_LB_{f_P}(q, tr) > d_{max}$. □

Example. Fig. 12a gives an example of k-NN point query, where the critical steps are shown in Fig. 12b.

k-NN Trajectory Query. The procedure of k-NN trajectory query is similar to that of k-NN point query, but with the following three differences.

(1) The elements of cdq and req are ordered by $f_F(q, tr)$ and Equation (15), respectively.

$$Region\_LB_{f_F}(q, r) = \max_{q_i \in q} d(q_i, r). \tag{15}$$

(2) We use Equation (15) to perform Region Pruning, whose correctness can be guaranteed by Lemma 8.

Lemma 8. *If $Region\_LB_{f_F}(q, r) > d_{max}$, then $f_F(q, tr) > d_{max}$, where tr is a trajectory queried by r but not retrieved before.*

Proof. If tr has already been retrieved, it should have been checked. If tr has not been retrieved, we have: $f_F(q, tr) \geq \max\{\max_{q_i \in q} \min_{p_j \in tr} d(q_i, p_j), \max_{p_j \in tr} \min_{q_i \in q} d(p_j, q_i)\} \geq \max\{\max_{q_i \in q} \min_{p_j \in tr.mbr} d(q_i, tr.mbr), \max_{p_j \in tr.mbr} \min_{q_i \in q} d(tr.mbr, q_i)\} \geq \max\{\max_{q_i \in q} \min_{p_j \in r} d(q_i, p_j), \max_{p_j \in r} \min_{q_i \in q} d(p_j, q_i)\} \geq \max_{q_i \in q} \min_{p_j \in r} d(q_i, p_j) = \max_{q_i \in q} d(q_i, r) = Region\_LB(q, r) > d_{max}$. □

(3) We use the lower bounds proposed in Section 5.3 to perform LB Pruning for k-NN trajectory query.

6 Experiments

6.1 Datasets & Settings

Datasets. We use three datasets to evaluate the performance of TrajMesa: 1) *T-Drive* [34], which includes taxi trajectories of Beijing, China from 2008-02-02 to 2008-02-08; 2) *Lorry*, which contains JD logistic lorry trajectories of Guangzhou, China from 2014-03-01 to 2014-03-31; and 3) *Synthetic*, which

TABLE 1
Statistics of Datasets

Attributes	#Points	#Objects	Size	Rate	#Traj.	Max Span
T-Drive	17,662,984	10,366	752MB	177s	314,086	35.5h
Lorry	886,593,200	48,813	136GB	20s	7,280,994	43.5h
Synthetic	8,865,932,000	488,130	1360GB	20s	72,809,940	43.5h

TABLE 2
Parameter Settings

Parameters	Settings
Data Size (%)	20, 40, 60, 80, **100**
Time Range	1h, **6h**, 1d, 1w, 1m, 2m, 3m
Spatial Range (km^2)	1×1, 2×2, **3×3**, 4×4, 5×5
k	**50**, 100, 150, 200, 250
ε (km)	1, 2, **3**, 4, 5
Time Period	1 week, **1 month**, 1 year

TABLE 3
Softwares in the Experiments

Software	Version	Software	Version	Software	Version
Hadoop	2.7.6	GeoMesa	2.3.0	JDK	1.8
Spark	2.3.3	HBase	1.4.9	Scala	2.11

is generated by copying Lorry dataset up to 1T to test the scalability of TrajMesa. Their statistics are shown in Table 1. Note that the max time span of trajectories in the two real datasets are $35.5h$ and $43.5h$, respectively, both of which are greater than 1 day. To this end, we cannot use 1 day as the time period bin in ID Temporal Indexing table.

Settings. Table 2 summarizes the parameters, where the default values are in bold. The max sequence length g is set as 16 (about $1km \times 1km$), as this is a common resolution of spatial range query in many urban applications. Table 3 gives the softwares and their versions. We use Spark to preprocess trajectories, and HBase as the underlying NoSQL data store of GeoMesa. To eliminate the effect of HBase cache,[6] we randomly select 100 different query parameters, perform each query only once, and take the median of all queries as the final results. All experiments are conducted on a cluster of 5 nodes, with each node equipped with Centos 7.4, 8-core CPU, 32 GB RAM, and 1T disk.

Baselines. We compare TrajMesa (i.e., *TM*) with six baselines, as shown in Table 4, where their supported queries are marked. Among these baselines, STH [7] (i.e., ST-Hadoop) is an advanced disk-based trajectory management system, and Dita [12] and DFT [11] are state-of-the-art in-memory trajectory management platforms (we obtain the source code from their authors, and run these systems in our experimental environment. Other systems, e.g., [10], [35], are not compared as we cannot get their source codes). TM_V, TM_{nlb}, and TM_{nps} are the variants of TM.

• **TM**$_V$ adopts V-Store as the underlying storage schema. Note that we retrieve trajectories if PART of their GPS points locate in a given spatial/temporal range. As a result, TM_V does not directly support the queries proposed in this paper. To achieve ID temporal query and spatial range query, we first get the trajectory IDs in a given temporal/spatial range, then retrieve all GPS points for each qualified trajectory ID. In this storage schema, we also store two copies of data with carefully designed spatio-temporal keys.

• **TM**$_{nlb}$ does not apply lower bound pruning techniques to spatial range query and k-NN query.

• **TM**$_{nps}$ does not adopt $PosCode$ in Spatial Range Indexing table.

In this paper, we only present f_F-based results for similarity query and k-NN query, because both Dita and DFT support it, as well as the limitation of pages. The experimental results of parameter tuning (e.g., α, β) can be seen in Appendix C, available in the online supplemental material. More results can be found in our technical reports [19].

TABLE 4
Supported Queries of Comparing Methods

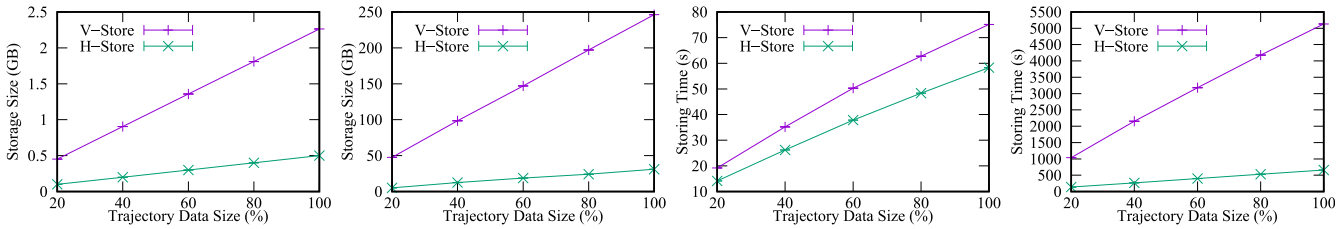
Queries	STH	Dita	DFT	TM_V	TM_{nlb}	TM_{nps}
IDT	×	×	×	√*	√	√
SR	√	√	×	√*	√	√
Sim_{f_F}	×	√	×	×	√	√
$k\text{-NN}_{f_F}$	×	√	√	×	√	√
$k\text{-NN}_{f_P}$	×	×	×	×	√	√

6.2 Performance of Storage

Figs. 13a and 13b compare the storage sizes of different storage schemas when varying the trajectory data size. Note that the storage size includes both ID temporal indexing table and spatial range indexing table. There are two observations: 1) the storage size gets linear growth with an increasing of trajectory data size for both H-Store and V-Store, because it needs more storage space when the data gets larger; 2) V-Store takes up about 5 times of storage space than H-Store for both datasets, because H-Store stores the GPS points in a trajectory together, and can compress the GPS point data more easily. It is also interesting to see that the storage size of H-Store is even much smaller than the raw data size, although we store two copies of data. This owes to the compression mechanism of H-Store.

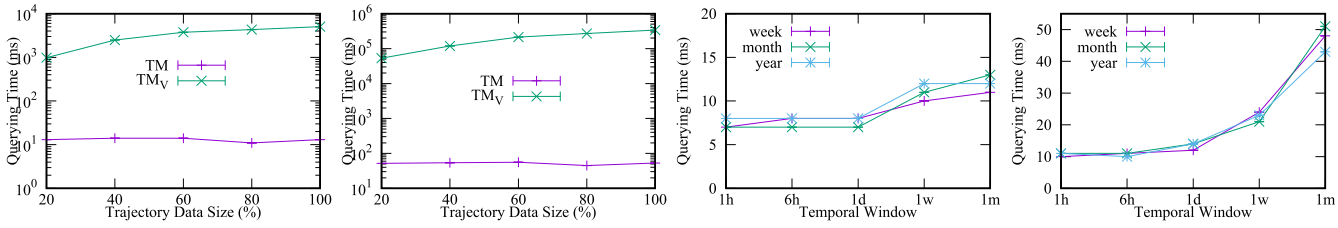
Figs. 13c and 13d show the storing time of V-Store and H-Store with different data sizes. It shows that with more data, the storing time of both storage schemas increases, as we need to process more data. V-Store takes much more time than H-Store. There are two main reasons. First, V-Store stores much more key-value entries than H-Store. More key-value entries means more operations over the underlying NoSQL data store. Second, the storage data size of H-

6. HBase caches results in memory to expedite same queries.



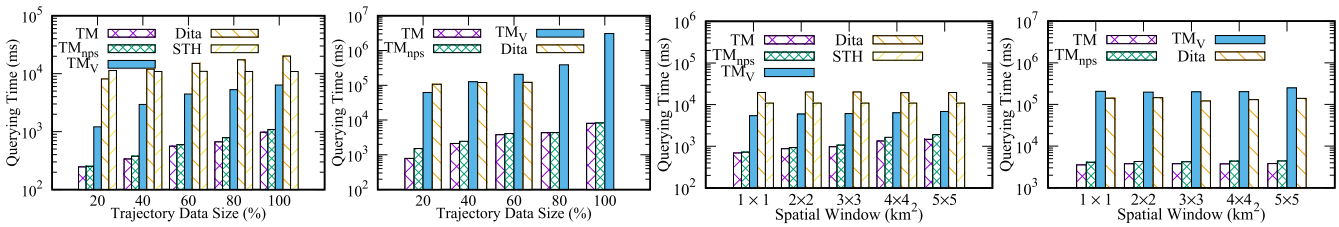
(a) Storage Size (T-Drive). (b) Storage Size (Lorry). (c) Storing Time (T-Drive). (d) Storing Time (Lorry).

Fig. 13. Performance of storage.



(a) Data Size (T-Drive). (b) Data Size (Lorry). (c) Time Window (T-Drive). (d) Time Window (Lorry).

Fig. 14. Performance of ID temporal query.



(a) Data Size (T-Drive). (b) Data Size (Lorry). (c) Spatial Window (T-Drive). (d) Spatial Window (60% of Lorry).

Fig. 15. Performance of spatial range query.

Store is much smaller than V-Store. For the same dataset, smaller storage size triggers less disk I/Os.

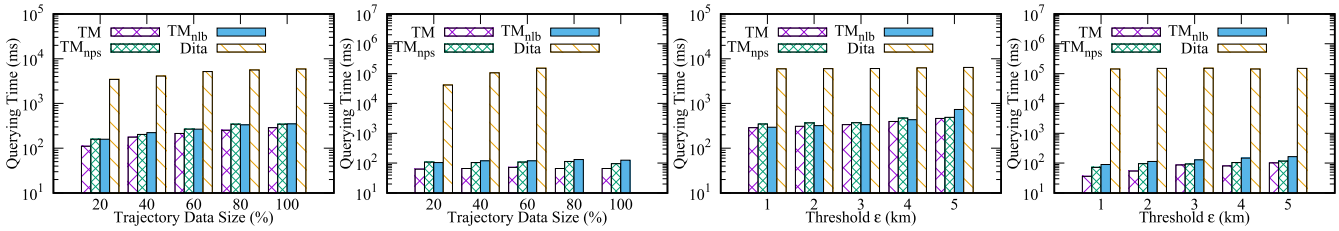
6.3 Performance of ID Temporal Query

Different Data Sizes. We compare the ID temporal query time of TM and TM$_V$ with different data sizes (TM$_{nlb}$ and TM$_{nps}$ are not tested, because they use the same ID indexing table with TM). As shown in Figs. 14a and 14b, TM is much faster than TM$_V$. Because for the same data size, the storage space of TM is much smaller than that of TM$_V$, which leads to less disk I/Os. We can also observe that the ID temporal query time of TM is not affected by the data size, because TM directly locates the trajectories of a given moving object, no matter how big the dataset is. The query time of Lorry is a little more than that of T-Drive, because the sampling rate of Lorry is higher, and it returns more GPS points in a given time window.

Different Time Windows. Figs. 14c and 14d present the ID temporal query time with different time windows. It is observed that, 1) for all time periods, the query time increases when a longer time window is given, as more trajectory data is returned. As T-Drive data only contains one week of data, if given one month of time window, the query time does not increase. 2) The query time over Lorry is more than that over T-Drive, because the sampling rate of Lorry is higher than that of T-Drive. When given the same time window, Lorry returns more GPS points. 3) The selection of time period has little to do with ID temporal query in our experimental settings. Because we set moving object IDs as the prefix of keys, which prunes most unnecessary trajectory data. Another reason could be the powerful parallel ability of underlying data store in TrajMesa. Although a shorter time period may result in more query windows, TrajMesa triggers `SCAN` operations over the underlying data store in parallel. To this end, we suggest choose a longer time period bin in XZT, as XZT with a shorter time period bin cannot index longer trajectories.

6.4 Performance of Spatial Range Query

Different Data Sizes. Figs. 15a and 15b exhibit the spatial range query time varying with different data sizes (TM$_{nlb}$ is not presented, as it is the same with TM). From the figures, we can observe that: 1) for both datasets, if we increase the data size, it requires more time for all methods, as more data incurs more disk I/Os; 2) TM is faster than TM$_{nps}$ for the same data size, as TM scans less data thanks to the proposed *PosCode*; 3) TM$_V$ is slower than TM, because the storage space of TM$_V$ is much larger than that of TM, which triggers more disk I/Os; 4) Dita builds huge indexes in memory. When the Lorry data size is greater than 60%, Dita fails for an out-of-memory exception; 5) TM is faster than Dita by two orders of magnitude sometimes, because TM locates related data directly, but Dita scans big indexes; 6) STH is slower than TM, because it triggers multiple disk



(a) Data Size (T-Drive). (b) Data Size (Lorry). (c) Threshold (T-Drive). (d) Threshold (60% of Lorry).

Fig. 16. Performance of similarity query.

read/writes even for a single request. Although STH seems faster than Dita for spatial range queries, it takes unbearable time to build the indexes. For example, STH takes up to 61 minutes to build the spatial indexes even for T-Drive data in our experiments. To this end, we do not compare STH for the big Lorry data.

Different Spatial Windows. Figs. 15c and 15d show that with a bigger spatial window, all methods need more time, as more data is read. TM is superior to TM_{nps}, because *PosCode* avoids retrieving invalid trajectories. TM is much faster than TM_V, due to the smaller storage space offered by H-Store. TM is faster than Dita and STH for all given spatial ranges, because TM locates directly the candidate data (with no index in memory), but Dita needs to scan huge indexes in memory (its space complexity is $\mathcal{O}(N_G^2 + N_L^{K+2} + |\mathcal{T}|)$, which is proportional to the number of trajectories), and STH incurs multiple disk I/Os even for a single request.

6.5 Performance of Similarity Query

Different Data Sizes. As depicted in Figs. 16a and 16b, for all methods, the similarity query time increases with a bigger data size, because with more data, it would return more trajectories. TM is faster than TM_{nps}, as similarity query calls spatial range query, where position code improves efficiency. TM_{nlb} is slower than TM, as without lower bound pruning, it needs to calculate the similarity of all candidate trajectories, which is time-consuming. Dita is much slower than TM, because Dita builds big indexes in memory. For each query, Dita would scan the huge indexes, which is costly. On the contrary, TM directly generates the query windows, and triggers `SCAN` operations over the underlying data store in parallel. The scalability of Dita is limited. When the Lorry data size is more than 60%, Dita throws an out-of-memory exception. However, TM works well even for full Lorry data size, which proves the powerful scalability of TM.

Different ε. Figs. 16c and 16d show that with a bigger threshold ε, the similarity query time increases slightly for all methods, because with a bigger ε, more trajectories will be returned. TM and its variants are faster than Dita, even by three orders of magnitude, which further proves the powerful efficiency of TrajMesa. Note here that we only use 60% of Lorry data in Fig. 16c, as Dita does not support larger data in our experimental environment.

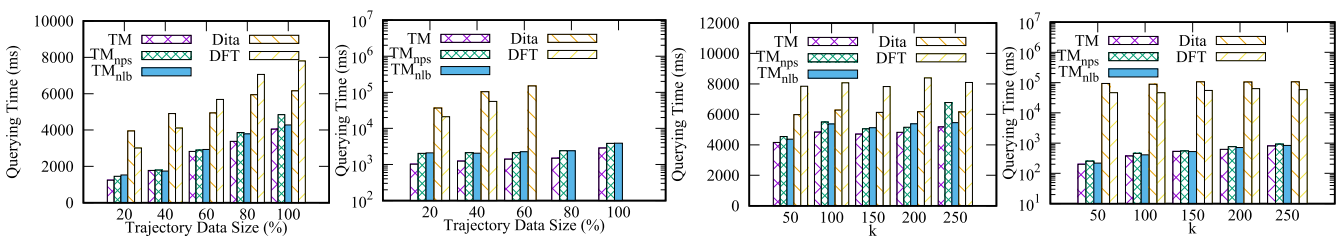
6.6 Performance of k-NN Query

Different Data Sizes. Figs. 17a and 17b show that, with an increasing of data size, the query time of all methods increases, because it extracts more data from the disk in every expansion. TM takes less time than TM_{nps}, which benefits from the position code. TM_{nlb} is slower than TM, because it needs calculate the similarity of all candidates, which is time-consuming. Both Dita and DFT take more time than TM, as they need to scan huge indexes in memory for each request. Dita and DFT could not cope with the situation when the data size of Lorry is more than 40% and 60% respectively, as they need to build memory-consuming indexes, but TM can easily handle it, which proves the powerful scalability of TM.

Different k. Figs. 17c and 17d present the k-NN query time with different k values. Note that we only use 40% of Lorry data, as Dita cannot handle more data for the bottleneck of memory. It shows that for both datasets, with a bigger k, all methods need more time, as they would check more trajectory data. Given a value of k, TM takes less time than TM_{nps} and TM_{nlb}, verifying the effectiveness of position code and pruning rules. TM outperforms Dita and DFT, even by two orders of magnitude for a larger dataset of Lorry. It is because TM directly generates a bunch of query windows, and performs these query windows in parallel. But Dita and DFT need to scan the huge indexes.

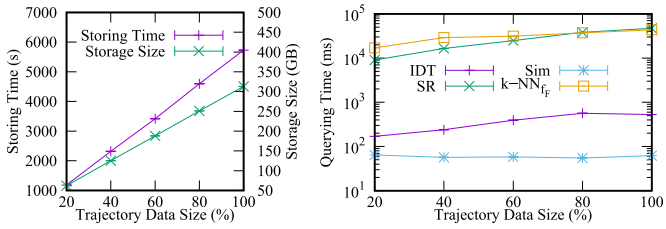
6.7 Scalability of TrajMesa

To further verify the scalability of TrajMesa, we conduct a set of experiments using the synthetic dataset, whose size is over 1T. As shown in Fig. 18a, when the data size gets



(a) Data Size (T-Drive). (b) Data Size (Lorry). (c) k (T-Drive). (d) k (40% of Lorry).

Fig. 17. Performance of k-NN query.



(a) Data Size (Synthetic). (b) Data Size (Synthetic).

Fig. 18. Scalability of TrajMesa.

bigger from 20% to 100%, both storing time and storage size grow linearly, because more trajectories need to be processed. Storing about 1T data only needs about 1.5 hours and 313 GB disk space, which is due to the novel underlying storage schema and compression mechanism.

Fig. 18b shows that the time of all queries increases with a bigger data size, as more trajectories are qualified, and it triggers more disk I/Os and more transmission bandwidth. It is interesting to see that the query time of similarity query is less than that of spatial range query, although we perform two spatial range queries underlying each similarity query. The reason could be that similarity query prunes most trajectories, and there is much less data returned. The transmission bandwidth acts as a bottleneck.

7 Related Works

In this section, we summarize the related works from three aspects: spatial and spatio-temporal indexes, NoSQL for spatio-temporal data, and trajectory data management.

Spatial and Spatio-Temporal Indexes. Traditional relational database management systems, e.g., Oracle Spatial or PostGIS, adopt R-tree [36], k-d tree [37], quad tree [38], or their variants to index spatial data. To index spatio-temporal data, [39] proposes a unified spatio-temporal indexing schema, i.e, 3D R-tree, which regards the temporal information as a third dimension. However, 3D R-tree is not suitable for trajectories with a long period [23]. HR-tree [40] and H +R-tree [41] break the temporal dimension into disjoint time intervals, and build an individual spatial index for each time period. CSE-tree [42], conversely, first partitions spatial data into grids, and for each grid, it builds a B+tree. These systems are mainly centralized implementations. They suffer from scalability problem, and could not manage massive trajectories effectively.

NoSQL for Spatio-Temporal Data. Key-value data stores, like Bigtable [16] and its open-source counterparts, e.g., Cassandra [43] and HBase, have proven to scale to millions of updates, and provide high-scalability, high-availability and fault-tolerant data management. These key-value data stores, however, do not natively support multi-attribute access, which results in full scan of the entire data for spatio-temporal queries. There emerge many works [17], [44], [45], [46], [47], [48] to support multi-dimensional data access over key-value data stores. For example, MD-HBase [44] encodes spatial data using Z-Ordering method, and builds two index structures, i.e., k-d tree and quad tree, among these codes over HBase. BBoxDB [45] proposes a two-level index structure over NoSQL data stores, where the global index uses a k-d tree to indicate which node stores the data, and the local index employs an R-tree in each node to find the partition of each data item. GeoMesa [17] provides a toolkit to transform multi-dimensional data into key ranges, thus enables NoSQL data stores to manage spatio-temporal data. However, these frameworks are not designed for trajectory data, thus cannot be applied to manage trajectories directly.

Trajectory Data Management. To manage massive trajectories, many trajectory management systems have emerged, which can be divided into three main categories: 1) *Single Machine-based Systems* [5], [6]. For example, TrajStore [5] maintains an adaptive grid-based index on the data, and dynamically co-locates and compresses spatially and temporally adjacent data on disk, thus it can retrieve all data in a particular spatio-temporal region efficiently. Torch [6] proposes a unified index, i.e., LEVI with compression, and an efficient query processing technique to support various trajectory queries. However, single machine-based trajectory management systems suffer from scalability problem, and cannot manage big trajectories effectively. 2) *Distributed In-Memory Systems* [10], [11], [12], [13], [14]. For example, Dita [12] identifies representative points as pivots in trajectories, and designs a trie-like index structure based on the pivots to prune dissimilar trajectories efficiently. UlTraMan [10] integrates chronicle map with Spark to relieve the significant pressure on JVM GC, and implement an abstraction TrajDataset for random data access. DFT [11] partitions trajectories based on their segments instead of MBRs, thus can reduce the overlaps of regions and empower the ability of pruning. Most of these in-memory frameworks are based on Spark, and need to load all trajectories into memory to build indexes, hence they require high-performance clusters with much memory, and cannot scale to very large trajectory data. Besides, for each request, they need to scan huge indexes, which is costly. 3) *Distributed Disk-based Systems* [7], [8], [9], [29], [30]. For example, [9] proposes PMI and OII to deal with spatio-temporal range queries of trajectory data based on MapReduce. [8] proposes Summit based on [7] to process massive trajectories using Hadoop. [30] proposes a cloud-based trajectory data management framework, but they adopt V-Store as the underlying storage schema, and do not optimize for similarity query and k-NN query. THBase [35] proposes a coprocessor-based scheme for big trajectory data management based on HBase. It exploits a hybrid local secondary index structure to accelerate spatio-temporal queries. However, THBase is not optimized in the underlying trajectory storage, which hinders its efficiency.

8 Conclusion

This paper proposes TrajMesa, which manages big trajectory data efficiently with plenty of queries support. We carefully design a novel storage schema that reduces the storage size tremendously. We devise a novel method to index time ranges, and a position code to indicate trajectory locations accurately. A bunch of pruning rules are proposed to accelerate similarity query and k-NN query in NoSQL environments. Experiments using three datasets verify the powerful efficiency and scalability of TrajMesa, showing that TrajMesa is **100**$\sim$**1000** times faster than the state-of-the-art trajectory data frameworks in our experimental settings.

ACKNOWLEDGMENTS

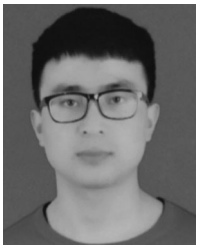
We would like to thank our developers for their contributions, with special thanks to Yuan Sui, Wei Wu, Junwen Liu, Haowen Zhu, Jian Hu, and Peng Wang. Suggestions and comments from anonymous reviewers greatly improve this paper.

REFERENCES

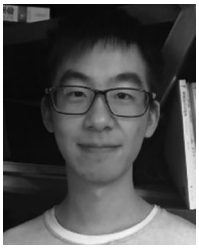
[1] R. Li *et al.*, "Discovering real-time reachable area using trajectory connections," in *Proc. 25th Int. Conf. Database Syst. Adv. Appl.*, 2020, pp. 36–53.
[2] L.-A. Tang *et al.*, "On discovery of traveling companions from streaming trajectories," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 186–197.
[3] S. Ma, Y. Zheng, and O. Wolfson, "Real-time city-scale taxi ridesharing," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1782–1795, Jul. 2015.
[4] T. He *et al.*, "Interactive bike lane planning using sharing bikes' trajectories," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 8, pp. 1529–1542, Aug. 2020.
[5] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 109–120.
[6] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin, "Torch: A search engine for trajectory data," in *Proc. 41st Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2018, pp. 535–544.
[7] L. Alarabi, "ST-Hadoop: A MapReduce framework for big spatio-temporal data," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 40–42.
[8] L. Alarabi, "Summit: A scalable system for massive trajectory data management," *SIGSPATIAL Special*, vol. 10, no. 3, pp. 2–3, 2019.
[9] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on MapReduce," in *Proc. 1st Int. Workshop Cloud Data Manage.*, 2009, pp. 9–16.
[10] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "UlTraMan: A unified platform for big trajectory data management and analytics," *Proc. VLDB Endowment*, vol. 11, no. 7, pp. 787–799, 2018.
[11] D. Xie, F. Li, and J. M. Phillips, "Distributed trajectory similarity search," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1478–1489, 2017.
[12] Z. Shang, G. Li, and Z. Bao, "DITA: Distributed in-memory trajectory analytics," in *Proc. ACM Int. Conf. Manage. Data*, 2018, pp. 725–740.
[13] H. Yuan and G. Li, "Distributed in-memory trajectory similarity search and join on road network," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 1262–1273.
[14] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou, "Trajspark: A scalable and efficient in-memory management system for big trajectory data," in *Proc. Joint Conf. Web Big Data Asia-Pacific Web Web-Age Inf. Manage.*, 2017, pp. 11–26.
[15] Z. Fang, L. Chen, Y. Gao, L. Pan, and C. S. Jensen, "Dragoon: A hybrid and efficient big trajectory management system for offline and online analytics," *VLDB J.*, vol. 30, pp. 1–24, 2021.
[16] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, ACM New York, NY, USA, vol. 26, no. 2, pp. 1–26, 2008.
[17] Geomesa, 2021. [Online]. Available: https://www.geomesa.org/
[18] R. Li *et al.*, "TrajMesa: A distributed NoSQL storage engine for big trajectory data," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 2002–2005.
[19] TrajMesa, 2021. [Online]. Available: http://trajmesa.urban-computing.com/
[20] H. Alt and M. Godau, "Computing the fréchet distance between two polygonal curves," *Int. J. Comput. Geometry Appl.*, vol. 5, no. 01n02, pp. 75–91, 1995.
[21] S. Nutanong, E. H. Jacox, and H. Sagan, "An incremental hausdorff distance calculation algorithm," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 506–517, 2011.
[22] B.-K. Yi, H. V. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *Proc. 14th Int. Conf. Data Eng.*, 1998, pp. 201–208.
[23] Y. Zheng, "Trajectory data mining: An overview," *ACM Trans. Intell. Syst. Technol.*, vol. 6, no. 3, 2015, Art. no. 29.
[24] H. Sagan, *Space-Filling Curves*. Berlin, Germany: Springer Science & Business Media, 2012.
[25] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *Proc. ACM Int. Conf. Manage. Data*, 1984, pp. 181–190.
[26] C. Böhm, G. Klump, and H.-P. Kriegel, "XZ-ordering: A space-filling curve for objects with spatial extension," in *Proc. 6th Int. Symp. Adv. Spatial Databases*, 1999, pp. 75–90.
[27] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "GeoMesa: A distributed architecture for spatio-temporal fusion," in *Proc. Geospatial Informat. Fusion Motion Video Analytics V*, 2015, vol. 9473, Art. no. 94730F.
[28] S. Ruan *et al.*, "Learning to generate maps from trajectories," in *Proc. 34th AAAI Conf. Artif. Intell.*, 2020, pp. 890–897.
[29] S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng, "CloudTP: A cloud-based flexible trajectory preprocessing framework," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1601–1604.
[30] J. Bao, R. Li, X. Yi, and Y. Zheng, "Managing massive trajectories on the cloud," in *Proc. 24th ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2016, Art. no. 41.
[31] R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *Proc. 25th ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2017, Art. no. 96.
[32] B. Aydin, V. Akkineni, and R. A. Angryk, "Modeling and indexing spatiotemporal trajectory data in non-relational databases," in *Managing Big Data in Cloud Computing Environments*. Hershey, PA, USA: IGI Global, 2016, pp. 133–162.
[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.
[34] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2011, pp. 316–324.
[35] J. Qin, L. Ma, and J. Niu, "THBase: A coprocessor-based scheme for big trajectory data management," *Future Internet*, vol. 11, no. 1, 2019, Art. no. 10.
[36] A. Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching*, vol. 14. New York, NY, USA: ACM, 1984.
[37] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications*, vol. 18, no. 9, pp. 509–517, 1975.
[38] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.
[39] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proc. 3rd IEEE Int. Conf. Multimedia Comput. Syst.*, 1996, pp. 441–448.
[40] Y. Tao and D. Papadias, "Efficient historical R-trees," in *Proc. 13th Int. Conf. Sci. Statist. Database Manage.*, 2001, pp. 223–232.
[41] Y. Tao and Papadias, "MV3R-Tree: A spatio-temporal access method for timestamp and interval queries," in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, vol. 1, pp. 431–440.
[42] L. Wang, Y. Zheng, X. Xie, and W.-Y. Ma, "A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval," in *Proc. 9th Int. Conf. Mobile Data Manage.*, 2008, pp. 1–8.
[43] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[44] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi , "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," in *Proc. IEEE 12th Int. Conf. Mobile Data Manage.*, 2011, vol. 1, pp. 7–16.
[45] J. K. Nidzwetzki and R. H. Güting, "BBoxDB–A scalable data store for multi-dimensional big data," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 1867–1870.
[46] 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 *Proc. IEEE Int. Conf. Comput. Intell. Commun. Technol.*, 2015, pp. 247–251.
[47] Y.-T. Hsu, Y.-C. Pan, L.-Y. Wei, W.-C. Peng, and W.-C. Lee, "Key formulation schemes for spatial index in cloud data managements," in *Proc. IEEE 13th Int. Conf. Mobile Data Manage.*, 2012, pp. 21–26.
[48] X. Tang, B. Han, and H. Chen, "A hybrid index for multi-dimensional query in HBase," in *Proc. 4th Int. Conf. Cloud Comput. Intell. Syst.*, 2016, pp. 332–336.

Ruiyuan Li (Member, IEEE) received the BE and MS degrees from Wuhan University, China, in 2013 and 2016, respectively, and the PhD degree from Xidian University, China, in 2020. He was the head of Spatio-Temporal Data Group, JD iCity and a researcher with JD Intelligent Cities Research, leading the research and development of JUST (JD Urban Spatio-Temporal data engine). Before joining JD, he has interned with Urban Computing Group, Microsoft Research Asia from 2014 to 2017. His research interests include spatio-temporal data management and urban computing.

Huajun He received the BE degree in 2018 from Southwest Jiaotong University, where he is currently working toward the PhD degree with the School of Computer Science and Technology. He is currently a research intern with JD Intelligent Cities Research and JD iCity, under the supervision of Prof. Yu Zheng and Dr. Jie Bao. His research interests include urban computing, database, spatio-temporal data mining, and distributed systems.

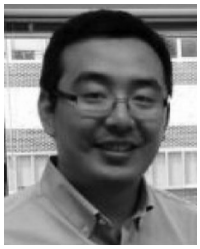


Runbin Wang received the BE degree in 2018 from Southwest Jiaotong University, where he is currently working toward the master's degree with the School of Computer Science and Technology. He is currently a research intern with JD Intelligent Cities Research and JD iCity, under the supervision of Prof. Yu Zheng and Dr. Jie Bao. His research interests include urban computing, spatio-temporal data mining, and distributed systems.

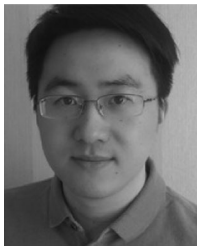


Sijie Ruan received the BE degree in 2017 from Xidian University, where he is currently working toward the PhD degree with the School of Computer Science and Technology. From 2016 to 2017, he was an intern with MSR Asia. He is currently a research intern with JD Intelligent Cities Research and JD iCity, under the supervision of Prof. Yu Zheng and Dr. Jie Bao. His research interests include urban computing, spatio-temporal data mining, and distributed systems.

Tianfu He recieved the BE degree in 2016 from the Harbin Institute of Technology, where he is currently working toward the PhD degree with the School of Computer Science. His current research interests include urban computing, spatio-temporal data management and data mining, especially trajectory data mining.



Jie Bao received the PhD degree in computer science from the University of Minnesota, Twin Cities, in 2014. From 2014 to 2017, he was a researcher with Urban Computing Group, MSR Asia. He currently leads Data Platform Division, JD iCity. His research interests include spatio-temporal data management or mining, urban computing, and location-based services.



Junbo Zhang (Member, IEEE) is currently a senior researcher of JD Intelligent Cities Research. He is leading the Urban AI Product Department of JD iCity, JD Technology, and also AI Lab of JD Intelligent Cities Research. He has authored or coauthored more than 50 research papers, such as, AI Journal, IEEE TKDE, KDD, AAAI, IJCAI, WWW, ACL, and UbiComp, in refereed journals and conferences. His research interests include spatio-temporal data mining and AI, urban computing, deep learning, and federated learning. He is an associate editor for the *ACM Transactions on Intelligent Systems and Technology*. He was the recipient of ACM Chengdu Doctoral Dissertation Award in 2016, Chinese Association for Artificial Intelligence (CAAI) Excellent Doctoral Dissertation Nomination Award in 2016, Si Shi Yang Hua Medal of SWJTU in 2012, and Outstanding PhD Graduate of Sichuan Province in 2013. He is a senior member of China Computer Federation, a member of ACM.

Liang Hong (Member, IEEE) received the BS and PhD degrees in computer science from the Huazhong University of Science and Technology in 2003 and 2009, respectively. He is currently a professor with the School of Information Management, Wuhan University. His research interests include knowledge graph, spatio-temporal data management, and social networks.

Yu Zheng (Fellow, IEEE) is currently a vice president and chief data scientist with JD.COM, passionate about using big data and AI technology to tackle urban challenges. He also leads the JD iCity, as the president and is the director of JD Intelligent Cities Research. Before joining JD, he was a senior research manager with Microsoft Research. He is also a chair professor with Shanghai Jiao Tong University and an adjunct professor with the Hong Kong University of Science and Technology. His research interests include big data analytics, spatio-temporal data mining, machine learning, and artificial intelligence.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.