VRE: A Versatile, Robust, and Economical Trajectory Data System

Hai Lan*‡, Jiong Xie†, Zhifeng Bao*, Feifei Li†, Wei Tian†, Fang Wang†, Sheng Wang†, Ailin Zhang†
*RMIT University, †Alibaba Group
{hai.lan,zhifeng.bao}@rmit.edu.au,{xiejiong.xj,lifeifei,changfeng.tw,solar.wf,sh.wang,zhangailin.zal}@alibaba-inc.com

ABSTRACT

Managing massive trajectory data from various moving objects has always been a demanding task. A desired trajectory data system should be versatile in its supported query types and distance functions, of low storage cost, and be consistently efficient on processing trajectory data of different properties. Unfortunately, none of the existing systems can meet the above three criteria at the same time. To this end, we propose VRE, a **v**ersatile, **r**obust, and **e**conomical trajectory data system.VRE separates the storage from the processing. In the storage layer, we propose a novel segment-based storage model that takes advantage of the strengths of both point-based and trajectory-based storage models. VRE supports these three storage models and ten storage schemas upon them. With the secondary index, VRE reduces the storage cost up to 3x. In the processing layer, we first propose a two-stage processing framework and a push-down strategy to alleviate full trajectory transmission cost. Then, we design a unified pruning strategy for five widely used trajectory distance functions and numerous tailored processing algorithms for five advanced queries. Extensive experiments are conducted to verify the design choice and efficiency of VRE, from which we present some key insights that are crucial to both VRE and future trajectory system's design.

PVLDB Reference Format:
Hai Lan, Jiong Xie, Zhifeng Bao, Feifei Li, Wei Tian, Fang Wang, Sheng Wang, Ailin Zhang. VRE: A Versatile, Robust, and Economical Trajectory Data System. PVLDB, 15(12): 3398-3410, 2022.
doi:10.14778/3554821.3554831

1 INTRODUCTION

At Alibaba Cloud, we have experienced a growing need from various customers to manage large-scale trajectory data generated from different domains, such as vessels, vehicles and airplanes [21, 31, 39, 40]. All these customers have common concerns on management cost in dollars and similar query needs. For example, most customers need to employ (different forms of) similarity join in integrating multiple trajectory datasets to identify near duplicate trajectories; they also need to frequently employ (different forms of) similarity search for numerous purposes, such as trajectory clustering to facilitate traffic monitoring, public transport route design to meet travel demands, and charging station site selection to improve facility coverage. Moreover, these customers need to employ different distance functions to cater for their own scenarios, and trajectory data under different domains usually exhibit very different properties. In addressing various customers' needs, we find that a desired trajectory data system should be *versatile*, *robust* and *economical*. Unfortunately, existing systems [2, 12, 14, 18, 19, 25, 28, 33, 41] do not meet at least one of the following desiderata:

D1: Versatile to Support Various (Advanced) Query Types and Distance Functions. Different users have their own frequently issued queries over trajectories. We summarize eight representative queries (three basic and five advanced) in Sec. 2.1 that a versatile system should support. In the existing systems, Summit [12] and MobilityDB [14] only support the basic queries, TrajMesa [24], DFT [33] and DITA [28] only support a limited number of advanced queries (Table 1). Notably, subtrajectory search and top-k similarity join (the most expensive query) cannot be supported by any system.

Moreover, distance function is an indispensable ingredient of advanced queries such as trajectory similarity search and join. A versatile system should also support all typical trajectory distance functions to cater for various application needs, i.e., DTW [36], EDR [17], Fréchet [10], Hausdorff [27], and LCSS [30]. TrajMesa, DFT, and DITA only support a limited number of them (Table 1). It is not trivial to extend them to support other distance functions since powerful distance-specific pruning bounds have to be proposed first. Detailed justifications are presented in our technical report [8].

D2: Robust to Trajectory Data of Different Properties. Different moving objects generate trajectories with different properties. We profile one trajectory dataset using three metrics: 1) number of points in a trajectory (*NoP*); 2) spatial span of a trajectory (*SpS*); 3) density of a trajectory dataset (*DoT*), i.e., the number of trajectories in a unit area. From our evaluation, we find that the query performance is closely related to these metrics:

1) *Trajectory with a large NoP.* Almost all existing systems assume that trajectories have a small *NoP* (less than 1000) and split long trajectories into several short ones. For instance, when evaluating DITA [28], a state-of-the-art system for trajectory analytics, using the AIS dataset [1], we find its performance unacceptable for two reasons: 1) the default number of pivot points is not enough to prune irrelevant trajectories; 2) when calculating the DTW distance, DITA declares a large two-dimensional array to store the intermediate results, which can easily cause a page fault.

2) *Trajectory with a large SpS.* Suppose users issue a spatial range query to find the trajectories passing through a given area. Point-based storage model (i.e. each point is stored as a single row), such as GeoMesa [2], can prune irrelevant trajectories efficiently but reconstructing the resulting trajectories takes a longer time due to group and sort operations. When storing trajectories with trajectory-based

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554831

‡This work was done when the first author did an internship at Alibaba.

model (i.e. each trajectory is stored as a single row), such as TrajMesa [25], the query efficiency may be heavily limited by many false positive trajectories being returned.

3) *Trajectory with a high DoT.* A typical dataset is Porto [5], upon which the candidates of similarity search in TrajMesa [25] are quite large (sometimes reaching 400,000), while only a few of them contribute to the final results. Transferring these candidates from the storage engine to the client or the processing layer takes much time which is unacceptable in real applications.

D3: Economical to Store Massive Trajectories. In particular, the system should 1) exhibit high scalability in the storage layer, and 2) strike a balance between query efficiency and storage cost – the storage cost should be appropriately close to the cost of storing trajectories in files with compression, and meanwhile the system can answer most (computationally-intensive) advanced queries with a satisfactory latency. When revisiting the state-of-the-art, we find DITA [28] and DFT [33] are built upon Spark and need to load all trajectories into memory, so they cannot work when the trajectory size exceeds the memory resources. Although TrajMesa [24, 25] adopts HBase [3], a distributed storage engine, to store massive trajectories, each index is stored with one data replica, which leads to data redundancy and tremendous storage cost.

Our Contributions. We propose VRE, a **v**ersatile, **r**obust, and **e**conomical disk-based trajectory management system with elaborative designs in both storage layer and processing layer (see Figure 1). VRE can support a wide range of Alibaba's customer applications on trajectories of different properties. A full technical report of this work is available at [8].

Storage Layer (Sec. 4). First, we propose a new storage model, segment-based model, which splits a trajectory into several segments. It has lower cost in both storage and trajectory reconstruction than the point-based model, and a better filtering capability than the trajectory-based model. Second, upon these three storage models, we design various storage schemas and present an indepth analysis on their storage cost (Sec. 4.4) and their impact on the processing time of different query types (Sec. 5.5). Third, similar to existing work [2, 25], four indexes are introduced to directly support the basic queries. To reduce data redundancy, VRE adopts the secondary index, which reduces the storage cost up to 3x in our experiments while having negligible influence on query latency. Last, a well-designed metadata is proposed, which is crucial to query processing.

Processing Layer (Sec. 5). First, to alleviate high data transmission cost of full trajectories, we propose a two-stage framework, with which VRE obtains the metadata and prunes irrelevant trajectories with metadata only, then fetches the full trajectories for verification. As shown in our experiments, it outperforms the one-stage counterpart at least 2x in both basic and advanced queries. Second, we propose a unified and effective pruning method for five commonly-used distance functions and embed it into our two-stage framework for processing advanced queries. A pushdown strategy to push some pruning steps into storage layer is proposed to significantly reduce the metadata transmission cost, especially, in the datasets of high density. VRE pushes some pruning steps into the storage layer in advanced queries. Last, under the above designs, we propose tailored algorithms for each of the advanced queries.

Evaluations and Insights (Sec. 6). We extensively evaluate different design choices of VRE and compare VRE with other systems. Our evaluation is done on real datasets of various properties and the trajectory number is up to almost 100 million. Key findings are summarized in Sec. 6.1 and we believe they are not only crucial to VRE but also shed light on future trajectory systems' design.

2 PRELIMINARIES AND RELATED WORK

2.1 Trajectory Query Types

We study eight representative trajectory queries and start from some spatial concepts. A point $p = \{id, lng, lat, t, a_1, a_2, ..., a_m\}$ contains an object id (*id*), spatial-temporal information *lng* for longitude, (*lat* for latitude, and t for timestamp) and other attributes a_i if any, e.g., the velocity of the object. A trajectory T is a sequence of points sorted by timestamp$\{p_1, p_2, ..., p_n\}$. $|T|$ denotes the number of points in T. Given a trajectory T, $T[i, j]$ is a subtrajectory [32] that denotes the portion of T from the i^{th} point to the j^{th} point.

The queries supported by VRE can be divided into two categories in Table 1: 1) basic queries – ID Temporal Query (*IDTQ*) [25], Spatial Range Query (*SRQ*) [25], and Spatio-Temporal Range Query (*STRQ*) [23]; 2) advanced queries – Threshold-based Trajectory Similarity Search (*Tb-Search*) [28], Top-k Trajectory Similarity Search (*k-Search*) [33], Threshold-based Trajectory Similarity Join (*Tb-Join*) [28], Top-k Trajectory Similarity Join (*k-Join*), and Subtrajectory Similarity Search (*sub-Search*) [32]. Since they are widely used in the literature, readers can refer to [8] for formal definitions.

2.2 Related Systems

In this paper, we focus on raw trajectories. Another line of studies adopts map matching [26, 35] to simplify them to sequences for subsequent query processing [38] on road network. However, trajectories that are not network constrained, such as vessels and airplanes, cannot be handled at all, and map matching itself raises data quality issues [16]. Centralized systems such as TrajStore [18] suffer from scalability, support basic queries but cannot support any advanced queries. Thus, we focus on distributed systems.

Distributed Disk-based Systems. Summit [12] is built on ST-Hadoop [13] to support *STRQ* and kNN point-based queries. MobilityDB [14] is built on PostgreSQL [7] and PostGIS [6]. It introduces two partitioning strategies to distribute trajectories across nodes to achieve spatial-temporal locality and load balance, but can only support *SRQ* and *STRQ* on trajectories due to such a design. TrajMesa [25] is built on GeoMesa and stores the data in HBase [3]. It stores each trajectory as a single row and designs two indexes for *IDTQ* and *SRQ*. It supports *Tb-Search* and *k-Search* on Hausdorff and Fréchet only. Other systems, e.g., Hadoop-GIS [11], Spatial-Hadoop [20], and ST-Hadoop [13], are proposed to handle basic spatial objects, e.g., point, polygon, or linestring.

Distributed In-memory Systems. UlTraMan [19] aims at building an in-memory trajectory data system while it only supports *SRQ* in Table 1. DITA [28], one of the state-of-the-art systems, is built on Spark to support *Tb-Search*, *k-Search*, and *Tb-Join*. It first partitions trajectories with the start and end points and builds a global index. A local Tire-like index is built by the selected pivot points. With the two-level indexes, DITA efficiently prunes dissimilar trajectories. Xie et al. [33] propose DFT to answer *k-Search* under Hausdorff and Fréchet. They build global and local indexes based

Table 1: Comparing VRE With Other Distributed Systems

Work	Basic Query			Advanced Query					Scalability		Dataset Properties		
	IDTQ	*SRQ*	*STRQ*	*Tb-Search*	*sub-Search*	*k-Search*	*Tb-Join*	*k-Join*	*Processing*	*Storage*	*NoP*	*SpS*	*DoT*
Summit [12]	×	✓	✓	×	×	×	×	×	✓	✓	-	-	-
MobilityDB [14]	×	✓	✓	×	×	×	×	×	✓	✓	✓	✓	×
TrajMesa [25]	✓	✓	✓	F/H[1]	×	F/H	×	×	×	✓	×	×	×
DFT [33]	×	×	×	F/H	×	F/H	×	×	✓	×	×	×	×
DITA [28]	×	✓	×	F/D/L/E[2]	×	F/D/L/E	F/D/L/E	×	✓	×	×	×	✓
REPOSE [41]	×	×	×	×	×	F/H/D	×	×	✓	×	-	-	-
UlTraMan [19]	×	✓	×	×	×	×	×	×	✓	×	-	-	-
VRE	✓	✓	✓	F/H/D/L/E	F/H/D/L/E	F/H/D/L/E	F/H/D/L/E	F/H/D/L/E	✓	✓	✓	✓	✓

[1] F, H, D, L, and E refer to Fréchet, Hausdorff, DTW, LCSS, and EDR, which are widely-used distance metrics between two trajectories.
[2] LCSS's definition in DITA [28] is not equivalent to that in the original paper [30]. We give the proof in Appendix F of [8].

on segments of trajectories, and use bitmap and dual indexing to boost the search performance. REPOSE [41] introduces a global partitioning strategy that places similar trajectories in different partitions, but it only supports *k-Search*. It is not trivial to extend it to support other queries, especially the join queries. That is because REPOSE places similar trajectories in different partitions, for a specific partition, many partitions can be joinable with it, which would lead to tremendous processing cost. Other systems, such as Simba [34], GeoSpark [37], the Spark module of GeoMesa [2], and LocationSpark [29], are proposed to handle other spatial objects, i.e., point, polygon or linestring. Linestring can be used to store trajectories while these systems (including the systems in the last category) can support *SRQ* only.

Remark. We give a comprehensive comparison of existing distributed systems for trajectories in Table 1 and highlight three key observations: 1) No existing system can support typical distance functions and the eight representative query types at the same time. Particularly, no systems can support *sub-Search* and *k-Join* that are prohibitively expensive but highly demanding in real world data integration and analysis. 2) As shown in Sec. 1, dataset properties can significantly influence the efficiency of query processing, while only the authors of MobilityDB state they consider *NoP* and *SpS* in [14]. 3) All in-memory systems have to load the whole dataset into the memory, which incurs a large memory consumption, e.g. DFT [33] incurs out-of-memory when processing *k-Search* (see Sec. 6.6). TrajMesa has an inherent data redundancy issue due to one data replica in each index built.

3 SYSTEM ARCHITECTURE

The overall architecture of VRE is presented in Figure 1, which mainly consists of two layers: storage layer and processing layer.

Storage Layer is introduced in the following two aspects.

Storage Model. We propose a new storage model, i.e. segment-based model, to store trajectory data. It splits a full trajectory into several segments and stores each segment as one row. Compared with point-based model, it cuts down the storage cost and the complexity of trajectory reconstruction. Compared with trajectory-based model, it fetches fewer false positive results when processing *SRQ* and *STRQ*. More details will be discussed in Section 4.1.

Storage Engine. In VRE, we expect the underlying storage engine to meet four desiderata – 1) *Scalability:* it can store massive trajectories and scales out easily. 2) *Native Secondary Index:* with the secondary index, VRE only needs to store one data replica to save storage cost. 3) *Cold-Hot Data Separation:* with this feature, outdated trajectories can be moved to a slower and cheaper storage to further reduce the storage cost. 4) *Local Processing:* with local processing capability in storage engine, VRE can push down some processing steps into storage layer to reduce data transfer cost. Here, we adopt a customized HBase, which meets the desiderata, but the choice of storage engine is orthogonal to VRE's design.

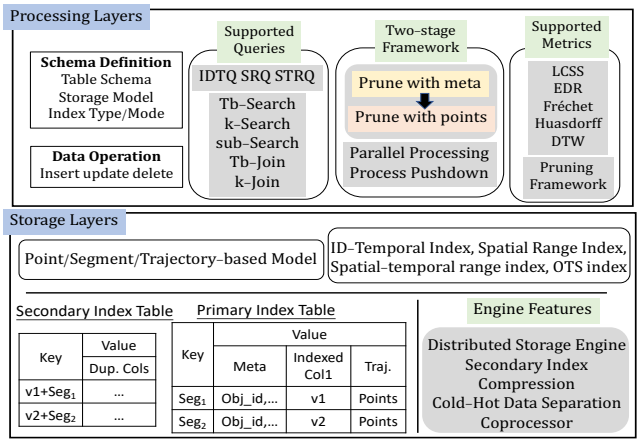


Figure 1: Architecture of VRE

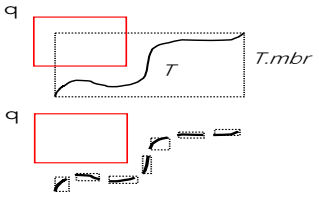
Processing Layer consists of three modules: schema definition, data operations, and query processing. Before loading data into VRE, users can define the storage schema, including the attributes of points in the trajectory, the primary indexes to be built, the storage model, and whether to build the secondary indexes. This layer supports all the basic and advanced queries as outlined in Section 2.1. We will elaborate on this layer in Section 5.

4 STORAGE LAYER

Apart from the existing point-based model and trajectory-based model, we first propose a new storage model for the trajectory, namely segment-based model. Then, we elaborate a range of physical storage schemas supported in VRE upon these three storage models. Next, we present the indexing strategies in VRE. Last, we discuss the storage cost of each storage schema.

4.1 Segment-based Model for Trajectory

Point-based model stores each point as one row [2, 6] but has two drawbacks: 1) high trajectory reconstruction cost when fetching a trajectory; 2) high storage cost due to numerous rows and inability



(a) Segment-based Model

Key	Value				
Rowkey	Metadata	Point List	Indexed Column	Indexed Column	Indexed Column
key_1(SR)	$meta_1$	$List_1$<Point>	key_1(IDT)	key_1(ST)	key_1(OTS)
key_2(SR)	$meta_2$	$List_2$<Point>	key_2(IDT)	key_2(ST)	key_2(OTS)
...	...	...	...	...	...
key_n(SR)	$meta_n$	$List_n$<Point>	key_n(IDT)	key_n(ST)	key_n(OTS)

Example Secondary Index Table to support IDT query

Key	Value
Rowkey	Duplicated Column
key_1(IDT) + key_1(SR)	...
key_2(IDT) + key_2(SR)	...
...	...
key_n(IDT) + key_n(SR)	...

(b) Segment-based Storage Schema. *SR* index is the primary index. *IDT*, *ST*, and *OTS* indexes are the secondary indexes. key(SR) is the key in *SR* index.

Figure 2: Storage Layer Design

to use compression algorithms. Trajectory-based model [25] stores each trajectory as one row. It natively overcomes the two drawbacks earlier, but more false positive trajectories may be returned in this model. For example, in the bottom of Figure 2(a), given an *SRQ* q, since q intersects with the MBR (Minimum Bounding Rectangle) of trajectory T, it needs to fetch T to verify the results. In contrast, if we split the trajectory into several segments, T can be filtered directly due to no intersection. Motivated by the above, we propose a segment-based model that splits a full trajectory into several segments[1] (Figure 2(a)):

Definition 4.1 (Segment). Given a trajectory T, it can be split into n segments $\mathcal{S}_T = \{S_1, S_2, ..., S_n\}$ where S_i is a segment with $|S_i|$ points and sorted with the serial number l among segments.

The splitter adopted in segment-based model comes with four splitting strategies: 1) By Duration: we sequentially check two consecutive points p_i and p_{i+1} with a given duration d. If $\lfloor p_{i+1}.t/d \rfloor \neq \lfloor p_i.t/d \rfloor$, we split the trajectory T from p_{i+1}. 2) By Distance: if the distance between p_i and p_{i+1} exceeds a given distance, we split T from p_{i+1}. 3) By Sliding Rectangle: for a point p in T, we generate an MBR $(p.lng - \theta, p.lat - \theta, p.lng + \theta, p.lat + \theta)$ where θ can be set by users. Then we visit the subsequent points and if there exists a point q not falling in this MBR, we split T from q. We repeat the above for remaining points. 4) By Point Number: we split T if the number of points in T exceeds a given threshold.

4.2 Physical Storage Schemas

All the above three storage models are supported in VRE. Since trajectory-based model and point-based model are a variant of segment-based model each, we present the storage schema upon segment-based model only.

The storage engine is a key-value store. Same as GeoMesa [2], the key is generated by the supported indexing strategies (Sec. 4.3). The value part mainly consists of three fields:

1) *Metadata*: it mainly consists of spatio-temporal properties of each segment, which helps VRE filter irrelevant trajectories without fetching the full trajectory, and thus reduces the data transmission cost from the storage layer. The properties in *Metadata* include:

- Moving object *id*, which generates the trajectory, its start time, the start and end time of a segment, the first and last point of a segment, the MBR of a segment, the serial number l of a segment in a full trajectory, and the type of a segment S.

$$\text{type}(S) = \begin{cases} 0/1/2, & \text{the start/internal/end segment in trajectory} \\ 3, & \text{a full trajectory (not split)} \end{cases}$$

- *sig*: it is a bit array of size $m \times n$ to represent the position of a segment S in its MBR. The MBR of S is divided into $m \times n$ disjoint regions; each region corresponds to a bit. If a point $p_i \in S$ is contained in region r, we set the corresponding bit of r as 1.

2) *Point List*: The points in a trajectory are first serialized by the processing layer when loaded, and then compressed with ZSTD [9].

3) *Columns for Secondary Index*: When deploying the secondary indexes, the keys of corresponding indexes should be added to the base table as additional columns. E.g., the last three columns in the left part of Figure 2(b) are additional columns for the secondary indexes. In this way, only one replica data needs to be stored.

4.3 Indexing

VRE relies on four indexes to efficiently process basic queries: 1) *IDT* index supports the ID-Temporal queries; 2) *SR* index supports the Spatial Range queries; 3) *ST* index supports the Spatio-Temporal Range queries; 4) *OTS* index helps fetch all segments and reconstruct the full trajectories in segment-based model. Note that we treat $(id + T[0].t)$ as trajectory ID tid, and $(id + T[i].t)$ as segment ID sid. Since they are not our main technical contribution, we only present *SR* and *OTS* indexes and refer the rest to Appendix C of [8].

SR Index. In VRE, based on the XZ2 indexing scheme [15], the key in *SR* index is defined as $shard + XZ2(MBR) + sid$ where $shard$[2] is used for load balancing, $XZ2(MBR)$ is the XZ2 codes of one segment's MBR, and sid is the segment id.

OTS Index. The key is defined as $shard + tid + l$ where tid is trajectory id, and l is the serial number of a segment in a full trajectory. With the *OTS* index, VRE fetches all segments of a trajectory with tid first, and then sorts these segments by their serial numbers to generate the full trajectory.

Secondary Index. With the secondary, only one replica data needs to be stored, otherwise, we need to store one replica data for each index, e.g., TrajMesa [25] and GeoMesa [2]. VRE can adopt any aforementioned index as the primary index. In our implementation, we choose *SR* index as the primary and the others as the secondary for two reasons: 1) *SR* index directly supports the spatial range query, which is a cornerstone in processing advanced queries. 2) It enables some pruning strategies of advanced queries to be pushed down into storage engine to reduce the query latency (see Sec. 6.4.1).

4.4 Discussion on Different Storage Schemas

Here, we use a notation VRE_{xyz} to denote different storage schemas. 'x' indicates which storage model we use; it can be P, T, and S, indicating point-based, trajectory-based, or segment-based model. 'y' means whether to use secondary index; it can be S or X, indicating with and without secondary index. 'z' means whether metadata is

[1] The segment in VRE is different with DFT [33] whose segment is the line between two consecutive points, i.e., if a trajectory contains n points, then it has $(n-1)$ segments.

[2] In our implementation, we use one byte with four different values.

used; it can be M or X, indicating with or without the *Metadata* . There are two other values: M_1 indicates that the metadata includes segments' metadata; M_2 indicates the metadata is separated from trajectories. For example, VRE_{TXM} tells that the storage model is trajectory-based, without secondary index, and with the metadata. All storage schemas that use secondary index (VRE_{*S*}) are identical to their counterpart that does not use secondary index.

Different storage schemas are shown in Table 2 where key_t, key_p, and key_s denote the *key* in trajectory-based, point-based, and segment-based model, respectively. Similarly, $meta_x$ (where x can be t, p, or s) represents the *Metadata* in the corresponding model. Last, $meta_{\{s\}}$ denotes a set which includes *Metadata* of all segments from the same trajectory. Compared with VRE_{TXM}, VRE_{TXM_1} includes more *Metadata* so that the number of false positive trajectories returned can be reduced. Compared with VRE_{SXM}, VRE_{TXM_2} stores each full trajectory as one single row, and hence trajectory reconstruction can be avoided.

Table 2: Choices of Storage Schema.

Name	Schema	System
VRE_{PXM}	$key_p + meta_p + point$	GeoMesa[2], VRE
VRE_{TXM}	$key_t + meta_t + traj$	TrajMesa[25], VRE
VRE_{TXM_1}	$key_t + meta_{\{s\}} + traj$	VRE
VRE_{SXM}	$key_s + meta_s + seg$	VRE
VRE_{TXM_2}	$key_t + traj; key_s + meta_s$	VRE
VRE_{*S*}	See above	VRE

A comparison of the physical storage cost is as below. 1) For the first five schemas in Table 2, we have the following: $\text{VRE}_{TXM} < \text{VRE}_{TXM_1} < \text{VRE}_{SXM} < \text{VRE}_{TXM_2} < \text{VRE}_{PXM}$. $\text{VRE}_{TXM} < \text{VRE}_{TXM_1}$ because there is less metadata in VRE_{TXM}. The reason behind $\text{VRE}_{TXM_1} < \text{VRE}_{SXM} < \text{VRE}_{TXM_2} < \text{VRE}_{PXM}$ is that there are fewer keys in the previous one. 2) The corresponding storage schemas with secondary index of the first five have the same order for similar reasons. 3) With secondary index, the storage cost will be reduced dramatically due to less data redundancy. These analyses will be verified in Sec. 6.3.

5 PROCESSING LAYER

In this section, we present how to process five advanced queries. First, we give an overview of two acceleration techniques for all advanced queries. Then, we drill down to the trajectory similarity search query and introduce our pruning strategies. Then, we present our algorithms to process trajectory similarity join and subtrajectory similarity search. For basic queries *IDTQ*, *SRQ* and *STRQ*, they can be directly supported by the indexes introduced in Sec. 4.3. Readers can refer to Appendix D in [8] for details.

5.1 An Overview of Acceleration Techniques

Two-stage Framework. In VRE we propose a two-stage framework to process advanced queries. It prunes unsatisfied candidate trajectories by only using their metadata at stage 1, and then fetches the rest full trajectories for verification at stage 2. Since much fewer full trajectories are fetched after pruning, the data transfer cost, which is the major bottleneck of query processing, will be tremendously reduced and result verification itself can be expedited.

Pushdown Strategy. 1) In *Tb-Search*, it may need to fetch numerous candidates that come along with many spatial range queries issued, leading to large data transfer cost. Thus, VRE tries to push appropriate pruning strategies into the storage engine. 2) In each iteration of *k-Search*, VRE may fetch many checked trajectories during expansion of the search, particularly in the dataset of high density. Thus, apart from pushing down the pruning strategies, VRE also pushes the step of *checking whether a trajectory has been visited before* down to the storage layer. 3) Similarly, the above two can be applied to *Tb-Join* and *k-Join* which need to repeatedly invoke the respective search operator.

5.2 Trajectory Similarity Search

There are two types of trajectory similarity search: Top-k Search (*k-Search*) and Threshold-based Search (*Tb-Search*). Here, we present how to process them based on storage schemas VRE_{S*M} and VRE_{T*M_2} and then provide core modifications in the algorithms to cater for the rest storage schemas, i.e., VRE_{T*M} and VRE_{T*M_1}.

5.2.1 k-Search. *k-Search* iteratively expands the spatial range, until the k most similar trajectories are found. It consists of two main steps, as shown in Algorithm 1:

1) **Initialization** (Lines 1-3). cdq is a priority queue that stores candidate trajectories sorted by the distance between candidate and the query Q, $mbrq$ is a priority queue to record the MBRs that need to be queried, and d_{max} is the current maximum distance between Q and the trajectories in cdq. Moreover, there are two parameters set as per the properties of dataset: the max resolution g, and the $segNum$. If the number of collected segments of a trajectory is larger than $segNum$, we fetch the full trajectory.

2) **Expansion** (Lines 4-18). After popping an MBR r from $mbrq$, if there are k trajectories in cdq and the distance[3] between Q and r exceeds d_{max}, the processing terminates. If the resolution of r is smaller than g, we add its four children nodes to $mbrq$ and continue to check the next MBR in $mbrq$. Otherwise, we trigger a spatial range query by r and obtain the candidates, denoted by C_s. Note that we only fetch the trajectories' metadata rather than the full trajectories (Line 10). Then, it comes with two substeps: i) **Metadata-based Pruning** (Line 12-14). For each segment $S \in C_s$, we first add S into C_t and check whether its corresponding trajectory can be pruned with the current collected metadata by our tailored pruning strategies in Sec. 5.2.3. ii) **Candidate Verification** (Lines 15-18). We check whether the number of segments of each tid in C_t is equal to $segNum$ or all segments' metadata have been fetched. If yes, we fetch the full trajectory T with tid by the OTS index and calculate the exact distance[4] between Q and T. If the distance is less than d_{max}, we add T to cdq, update d_{max}, and go to next iteration. A running example is in Appendix E.2 of [8].

Optimizations for Trajectory-based model. Recall Sec. 4.4 the trajectory-based model corresponds to two storage schemas VRE_{T*M} and VRE_{T*M_1}. For them, we propose three optimizations to boost the efficiency of *k-Search*.

1) *Parallel Processing.* Different from storage schemas on segment-based model, we fetch all metadata of one trajectory at the same time (Line 9) in VRE_{T*M} and VRE_{T*M_1}. Before the expansion step, we send the candidates to n workers, where n can be set by users. Then, each worker enters into the expansion step. Last, the main worker collects results from the n workers and updates the global

[3]The distance here is the lower bound of the unvisited trajectories.
[4]When the distance function is LCSS, we use $-f(Q, tr)$ as the distance.

Algorithm 1: Top-k Search (*k-Search*)

Input: query trajectory Q, trajectory dataset $\mathcal{T}$, trajectory distance function $f(\cdot,\cdot)$, a positive integer k
Output: A set of trajectories $\mathcal{T}_{knn}$

```
Initialize a priority queue cdq and mbrq;
Initialize a HashMap C_t = (tid, List<segment>); d_max = 0;
while mbrq is not empty do
    r = mbrq.dequeue();
    if cdq.size = k ∧ f_{r→t}(Q, r) ≥ d_max then
        break;
    if the resolution of r < g then
        Add the four children nodes of r to mbrq; continue;
    C_s = SRQ_meta(r, T);
    foreach S ∈ C_s do
        C_t(S.tid).add(S);
        if cdq.size = k then
            LB_SES Pruning (II); LB_Pivots Pruning (III);
            LB_PartialSim Pruning (IV); LB_SIG Pruning (IV);
        if C_t(S.tid).size = seg# or isFull(C_t(S.tid)) then
            T = trajConstructor(OTS(tid, T));
            if f(Q, T) ≤ d_max then
                cdq ← T; d_max = f(Q, cdq.last());
return cdq as T_knn;
```

top-k candidates.

2) *Sorting Before Pruning.* If trajectories in the final top-k set are found earlier, the pruning bound d_{max} will converge more quickly. Based on this idea, we sort the candidates in each worker by the distance of start and end points of Q w.r.t. one candidate.

3) *Local Bound Synchronization.* One worker may have a tighter local pruning bound d_{max} than other workers. The tighter local bound can be broadcast to other workers to reduce the overall processing cost. Here, we design a lock-free algorithm to synchronize the local bounds from all workers during query processing.

5.2.2 Tb-Search. *Tb-Search* is essentially a one-iteration version of *k-Search*, where d_{max} is the threshold τ in *Tb-Search*. We use the spatial range $R_0 = (lng_{min} - \tau, lat_{min} - \tau, lng_{max} + \tau, lat_{max} + \tau)$[5], where $(lng_{min}, lat_{min}, lng_{max}, lat_{max})$ is the MBR of Q, to get all candidate segments. Then, we group the candidates by trajectory id and send the results to n workers. Each worker does metadata-based pruning and candidate verification locally. Last, the main worker collects and returns the results of n workers. In VRE_{T*M} and VRE_{T*M_1} storage schemas, the group operation is removed. The pseudocode for *Tb-Search* is in Appendix E.1 of [8].

5.2.3 Pruning Strategies. Here, we use *Tb-Search* as an example to illustrate our pruning strategies. As aforementioned *Tb-Search* is actually a one-iteration version of *k-Search*, so such strategies apply to *k-Search* as well. Given a query $Q = \{q_1, q_2, ..., q_n\}$ with a distance threshold τ, after getting the candidates only with metadata, we group these segments by their tid. Each group is formed as $G = \{S_1, S_2, ..., S_{|G|}\}$, where $S_i = \{p_1, p_2, ..., p_{|S_i|}\}$ denotes a segment. Based on the well-designed metadata in Sec. 4.2, we propose five pruning strategies, and the lower bounds specific to each distance metric are in Table 3.

Completeness. For each candidate group G, we check whether a *complete* trajectory can be recovered from all the segments in G. Note that we do not check the completeness in EDR and LCSS since they allow partial points out of R_0 according to their definition.

1) **LB_SES.** It represents the lower bound by considering the start and end segments. If the lower bound is larger than τ, we can prune the group G.

2) **LB_PartialSim.** This pruning strategy gives lower bound based on the collected segments' metadata, i.e., partial segments.

• *Hausdorff and Fréchet.* For each segment $S_i \in G$, we calculate the Euclidean distance between the start point of S_i and any point in Q, and we do the same operation on the end point of S_i. Then, the maximum value will be selected as the distance for segment S_i. Finally, we select the maximum value among all the segments in G as the lower bound of candidate G.

• *DTW, EDR, and LCSS.* (i) For each segment $S_i \in G$, we first calculate the distance between the mbr of S_i and any point $q_j \in Q$. (ii) For DTW, we use the minimum value times the number of points in S_i as the distance for one segment S_i. For EDR and LCSS, if the minimum value is larger than the matching threshold in their definition, we use the number of points in S_i as the distance for one segment S_i. (iii) Finally, we sum the distances for all the segments in G as the lower bound of candidate G. If the calculated lower bound is larger than the given threshold τ, we remove G.

3) **LB_Pivots.** Inspired by the idea of pivot points in DITA [28], we design this strategy. Given a trajectory T, we generate the set of pivot points P by adding the start and end points of T, and any point $t \in T$ which is on the border of the MBR of T into P.

Following the above selection strategy, we first generate the pivot points set $P = \{q_1, q_2, ..., q_m\}$ for query Q. For each pivot point $q_j \in P$, we calculate the minimum distance between q_j and the mbr of any segment $S_i \in G$. Then, similar to the *LB_PartialSim* strategy, we select the maximum value among all pivot points as the lower bound for Hausdorff and Fréchet, while we sum the value of all pivot points as the lower bound of *LB_Pivots* strategy for DTW, EDR, and LCSS.

4) **LB_SIG.** As shown in Sec. 4.2, the signature of a query Q is formed as $sig_Q = (b_1, b_2, ..., b_{m \times n})$, where b_i denotes whether there is at least one point in the corresponding region r_i. For a candidate G, we collect the disjoint regions with $b_i = 1$ into a set sig_G. For each region $r_i \in sig_Q$, we calculate the minimum distance between r_i and any region $r_j \in sig_G$. Then, we get the maximum value among all minimum distances of regions $r_i \in sig_Q$ as the lower bound for Hausdorff and Fréchet. As for DTW, we sum the value, which is the product of the minimum distance of $r_i \in sig_Q$ and the number of points contained in r_i, as the lower bound. For EDR and LCSS, we sum the number of point from the r_i where the minimum distance is larger than matching threshold as the lower bound.

5.3 Trajectory Similarity Join

Tb-Join. Its process is divided into two steps: 1) *Partitioning*, which aims to put the trajectories that are likely similar into the same partition. 2) *Searching*, which runs *Tb-Search* for each partition.

[5] When the distance function is LCSS [30] or EDR [17], we use the matching threshold in their definition instead of τ to generate R_0.

Table 3: Pruning Strategies for Five Widely-adopted Trajectory Distance Measures

Metric	Completeness	LB_SES	LB_PartialSim	LB_Pivots	LB_SIG
Hausdorff	✓	0	$\max_{S_i \in G} \max_{q_j \in Q} \{f_{p\to p}(q_j, t_1), f_{p\to p}(q_j, t_{\lvert S_i \rvert})\}$	$\max_{q_j \in P} \min_{S_i \in G} \{f_{p\to r}(q_j, mbr_{S_i})\}$	$\max_{r_i \in sig_Q} \min_{r_j \in sig_G} \{f_{r\to r}(r_i, r_j)\}$[3]
Fréchet	✓	$\max\{f_{p\to p}(q_1, t_1)$,[1] $f_{p\to p}(q_{\lvert Q \rvert}, t_{\lvert S_{\lvert G \rvert} \rvert})\}$	$\max_{S_i \in G} \max_{q_j \in Q} \{f_{p\to p}(q_j, t_1), f_{p\to p}(q_j, t_{\lvert S_i \rvert})\}$	$\max_{q_j \in P} \min_{S_i \in G} \{f_{p\to r}(q_j, mbr_{S_i})\}$	$\max_{r_i \in sig_Q} \min_{r_j \in sig_G} \{f_{r\to r}(r_i, r_j)\}$
DTW	✓	$\max\{f_{p\to p}(q_1, t_1), f_{p\to p}(q_{\lvert Q \rvert}, t_{\lvert S_{\lvert G \rvert} \rvert})\}$	$\sum_{S_i \in G} \lvert S_i \rvert \times \min_{q_j \in Q} \{f_{p\to r}(q_j, mbr_{S_i})\}$[2]	$\sum_{q_j \in P} \min_{S_i \in G} \{f_{p\to r}(q_j, mbr_{S_i})\}$	$\sum_{r_i \in sig_Q} \lvert r_i \rvert \times \min_{r_j \in sig_G} \{f_{r\to r}(r_i, r_j)\}$
EDR	✗	$\max\{f_{p\to p}(q_1, t_1), f_{p\to p}(q_{\lvert Q \rvert}, t_{\lvert S_{\lvert G \rvert} \rvert})\}$	$\sum_{S_i \in G} \lvert S_i \rvert \times \min_{q_j \in Q} \{f_{p\to r}(q_j, mbr_{S_i})\}$	$\sum_{q_j \in P} \min_{S_i \in G} \{f_{p\to r}(q_j, mbr_{S_i})\}$	$\sum_{r_i \in sig_Q} \lvert r_i \rvert \times \min_{r_j \in sig_G} \{f_{r\to r}(r_i, r_j)\}$
LCSS	✗	$\max\{f_{p\to p}(q_1, t_1), f_{p\to p}(q_{\lvert Q \rvert}, t_{\lvert S_{\lvert G \rvert} \rvert})\}$	-	-	-

[1] $f_{p\to p}(p, q)$ denotes the Euclidean distance between a point p and a point q. In EDR and LCSS, it denotes the discrete distance defined by themselves.
[2] $f_{p\to r}(p, r) = \min_{p' \in r} f_{p\to p}(p, p')$ denotes the distance between a point p and a region r.
[3] $f_{r\to r}(r_1, r_2) = \min_{p_i \in r_1, p_j \in r_2} f_{p\to p}(p_i, p_j)$ denotes the region distance between r_1 and r_2.

1) **Partitioning.** First, we use the Sort-Tile-Recursive (STR) partitioning method [22] to partition all trajectories in $\mathcal{T}_1$ with their start points into N_s disjoint partition. Then, in each partition, we further use STR to partition the trajectories in each partition with the end points of these trajectories into N_e partitions. Finally, there are $N_s \times N_e$ partitions. In this way, similar trajectories are more likely to be in the same partition and each partition has roughly the same number of trajectories, even for highly skewed data.
2) **Searching.** For each partition, we trigger a *Tb-Search*$(par_i, \mathcal{T}_2, f, \tau)$, but with four differences: 1) R_0: we first merge the MBRs of all trajectories in one partition and extend it with the given threshold τ to get R_0. 2) R_1: which is an MBR that covers all start points of all trajectories in one partition and is extended with the given threshold τ. 3) R_2: which is an MBR that covers all end points of all trajectories in one partition and is extended with the given threshold τ. and 4) after getting the candidate trajectories according to R_0, R_1, R_2, VRE traverses all trajectories in one partition to search similar trajectories in $\mathcal{T}_2$. In each partition, we build a buffer to store the full trajectories fetched from $\mathcal{T}_2$, to avoid repeatedly fetching the same trajectories from $\mathcal{T}_2$. Note that when answering *Tb-Join*, we send $N_s \times N_e$ partitions to n workers.
k-Join. Similar to *Tb-Join*, after VRE partitions all trajectories in $\mathcal{T}_1$, it triggers a *k-Search*$(par_i, \mathcal{T}_2, f, k)$ for each partition. In each partition, VRE traverses all trajectories and also builds a buffer.

5.4 Subtrajectory Similarity Search

VRE also supports searching for the most similar subtrajectory [32], whose processing can be divided into three steps (Algo. 2):
1) *Fetching candidates* (Lines 1-2). We fetch the trajectories whose MBRs intersect with Q's MBR (MBR_Q) as the candidates.
2) *Status matrix generation* (Lines 3-16). For each candidate $T = (t_0, t_1, ..., t_{m-1})$, we calculate a matrix dt to store the minimum distance between candidate subtrajectories and $Q = (q_0, q_1, ..., q_{n-1})$, as defined by Eq. (1). Meanwhile, we use a matrix *pos* to record the position of the start point for candidate subtrajectories, as defined by Eq. (2). Note that in Eq. (1), the distance function $f(\cdot, \cdot)$ can be one of the Fréchet, DTW, EDR.
3) *Subtrajectory generation* (Line 17). According to the values in dt and *pos*, we get the positions of start and end points for each candidate subtrajectory in their full trajectory. Note that we can calculate the most similar subtrajectory in each candidate in parallel.

$$\mathrm{dt}(T, Q) = \begin{cases} f(t_i, q_0) & \text{if } n = 1 \\ \sum_{j=0}^{n-1} f(t_0, q_j) & \text{if } m = 1 \\ f(t_m, q_n) + prev & \text{otherwise} \end{cases} \quad (1)$$
$$prev = \min(\mathrm{dt}(T^{m-1}, Q^{n-1}), \mathrm{dt}(T^{m-1}, Q), \mathrm{dt}(T, Q^{n-1}))$$

Algorithm 2: Subtrajectory Similarity Search (*sub-Search*)

Input: query trajectory Q, trajectory dataset $\mathcal{T}$, trajectory distance function $f(\cdot, \cdot)$
Output: A set of subtrajectories $\mathcal{T}_{subsim}$
1 Candidates $C_s = SR_{meta}(MBR_Q, \mathcal{T})$;
2 Get the trajectory id set C_{tid} from C_s;
3 **foreach** $tid \in C_{tid}$ **do**
4 $T = trajConstructor(OTS(tid, \mathcal{T}))$;
5 Compute the matrix $dist[\lvert T \rvert, \lvert Q \rvert]$ as defined by Eq. (1);
6 Compute the matrix $pos[\lvert T \rvert, \lvert Q \rvert]$ as defined by Eq. (2);
7 Initialize (d, s, e) as $(+\infty, 0, 0)$ respectively;
8 **for** $i \leftarrow$ *1 until* $\lvert T \rvert$ **do**
9 $d_{min} = dist[i-1].last$; $t_s = pos[i-1].last$; $t_e = i - 1$;
10 **if** $d_{min} \le d$ **then**
11 $flag = true$;
12 **for** $j \leftarrow$ *0 until* $\lvert Q \rvert$ **do**
13 **if** $dist[i][j] < d_{min} \wedge pos[i][j] \le t_e$ **then**
14 $flag = false$;
15 **if** *flag* **then**
16 $(d, s, e) \leftarrow (d_{min}, t_s, t_e)$;
17 $\mathcal{T}_{subsim} \leftarrow T[s, e]$;

$$\mathrm{pos}(T, Q) = \begin{cases} i & \text{if } n = 1 \\ 0 & \text{if } m = 1 \\ \mathrm{pos}(T^{m-1}, Q^{n-1}) & \text{else if } prev = \mathrm{dt}(T^{m-1}, Q^{n-1}) \\ \mathrm{pos}(T^{m-1}, Q) & \text{else if } prev = \mathrm{dt}(T^{m-1}, Q) \\ \mathrm{pos}(T, Q^{n-1}) & \text{else if } prev = \mathrm{dt}(T, Q^{n-1}) \end{cases} \quad (2)$$

Other Distances. 1) For *Hausdorff*, we design a divide-and-conquer algorithm to find the most similar subtrajectory, whose complexity is $O(\lvert T \rvert \lvert Q \rvert \log min\{\lvert T \rvert, \lvert Q \rvert\})$. 2) For *LCSS*, we adopt the *Exact-S* [32]. The pseudocode is in Appendix E.3 of [8].

5.5 Impact of Storage Schemas on Efficiency

Last, we discuss how five choices of storage schema (proposed in Section 4.4) impacts the query processing efficiency. Empirical impacts will be explored in experiments (see Sec. 6.5).
Basic Queries. First, if we only need the trajectory *id* or partial points as the returned results, the efficiency of these schemas from high to low is $\mathsf{VRE}_{P**} > \mathsf{VRE}_{S*M} > \mathsf{VRE}_{T**}$. Second, if we need the entire trajectories that meet the conditions of the queries, the order of efficiency is $\mathsf{VRE}_{S*M} > \mathsf{VRE}_{T**} > \mathsf{VRE}_{P**}$.
Advanced Queries. The overall order of efficiency is $\mathsf{VRE}_{T**} > \mathsf{VRE}_{S*M} \gg \mathsf{VRE}_{P**}$. Point-based model cannot achieve any competitive performance in advanced queries due to its excessive cost

in reconstructing full trajectories. Compared to VRE_{S*M}, VRE_{T**} avoids trajectory reconstruction process and easily runs in parallel.

6 EXPERIMENTS

We conduct a comprehensive evaluation of VRE as well as existing systems, aiming to seek answers to the following questions:
Q1: (i) How much storage is taken under different storage schemas and (ii) whether the storage cost of the metadata is negligible?
Q2: How well our proposed optimization rules behave over datasets of different properties?
Q3: Given the optimization rules, which storage schema is the optimal choice for a specific query type over a type of dataset?
Q4: Compared with state-of-the-art (SOTA) systems, how efficient is VRE over datasets of different properties?

We start from presenting the insights gained through our evaluation and lessons learned from our system design (Sec. 6.1). Then, we give the experimental setup in Sec. 6.2 and evaluation details for **Q1**-**Q4** from Sec. 6.3 to Sec. 6.6 where the first three questions dig into the design guidance of VRE.

6.1 Key Findings

6.1.1 Insights.
I1: Our experiments affirm the storage cost analysis of different schemas in Sec. 4.4, regardless of whether the secondary index is employed (**Q1(i)**). The metadata takes only 4% of the total storage cost but makes query processing 2x faster when it is combined with our two-stage framework, and we believe this is acceptable (**Q1(ii)**).
I2: 1) Secondary index influences the queries that have large candidate size due to the non-negligible random access. In our two-stage framework, we can optimize it with quite small extra storage (See Sec. 6.4.1). 2) Combined with the metadata, the two-stage framework is 2x faster than the one-stage, because it fetches fewer full trajectories and results in a reduced data transfer cost. 3) Our pushdown strategy works on the datasets of high density, e.g., Porto (2x faster for *Tb-Search*). For a dataset of low density, e.g., AIS, it only slightly worsens the efficiency. When applying the pushdown strategy, there is a trade-off between the increasing overhead on storage layer and the reduced network cost (See Sec. 6.4.3). (**Q2**)
I3: No single storage schema can outperform others in all cases (**Q3**). The query type, the type of query result, i.e., full trajectories, trajectory ids only, or subtrajectoies, and the dataset property can influence the choice of storage schema, each of which will be elaborated in our design guidance (in Sec. 6.1.2). Moreover, VRE performs consistently well on datasets with different properties while SOTA is not. Taking *k-Search* for example, DITA outperforms DFT on Porto while it is beat by DFT on AIS (see Table 14). The main reason is that query types and dataset properties are not incorporated in their system design, while a versatile system should do.
I4: As a result of our tailored pruning strategies, processing algorithms and the proposed optimizations, VRE shows better or competitive performance as compared to SOTA in processing all advanced queries (that are also supported by SOTA), not to mention VRE uses much fewer cores (32 vs. 256) to achieve the above (**Q4**) and VRE supports two more advanced queries (*k-Join* and *sub-Search*) that cannot be supported by any existing system. Taking *Tb-Search* as an example, out of a total of 75 settings, VRE outperforms SOTA on 70 settings[6] (see Table 13).

6.1.2 Our System Design Guidance.
G1: Trajectories With Small *NoP* and Small *SpS*. Most existing datasets are from urban applications and belong to this category. 1) When the query type is one of the advanced queries, we recommend VRE_{TSM} and choose *SR* index as the primary index. If the dataset is of high density, e.g., Porto, the pushdown strategy should be applied. 2) For basic query types, if the result type is trajectory *id* or a set of points, we recommend VRE_{TSM} due to its low storage cost. If users care more on the latency, VRE_{PXM} can be used. If the full trajectory is returned, we recommend VRE_{TSM}. The choice of the primary index depends on the query type. Overall, we recommend *SR* index as the primary index. This is because *IDTQ* usually has a small candidate size and *STRQ* can be answered by *SRQ* with another filter on the time range.
G2: Trajectories With Large *NoP* and Large *SpS*. Vessel trajectories fall in this category, e.g., AIS [1]. 1) For advanced queries, we recommend VRE_{TSM} and VRE_{TSM_1} as they can do parallel processing and do not need trajectory reconstruction. Furthermore, which storage schema to use depends on the pruning effectiveness of the metadata. Similar to **G1**, *SR* index should also be the primary index and the pushdown strategy should be applied to the dataset of high density. 2) For basic queries, we recommend VRE_{SSM} and *SR* index as the primary index, which strikes a good balance between query performance and storage cost.
G3: Other Cases. In reality, there are very few datasets with small *NoP* and large *SpS* or with large *NoP* and small *SpS*. We recommend **G1** for the former case and **G2** for the latter case.

Table 4: Statistics of Datasets

	AIS	Porto	Beijing	OSM
# Trajectories	42,446	1,645,908	11,114,613	96,648,669
Size (GB)	5.34	1.94	10.4	201
Avg # Points (NoP)	2,678.9	50.0	22.2	49.8
Avg Spatial Span (SpS)	(2.0508,1.4909)	(0.0322,0.0221)	(0.1,0.374)	(0.016,0.03)
Density of Trajs (DoT)	2,310	410,326	827,359	1,491

6.2 Experimental Setup

Datasets. Our experiments are conducted on four representative real-world trajectory datasets, AIS [1], Porto [5], Beijing [28], and OSM [4]. The data statistics are shown in Table 4. Density is the average number of trajectories in a unit area. We randomly issue 20 spatial range queries with 0.3 on AIS and 0.003[7] on Porto and Beijing respectively, and calculate the average number of returned trajectories as the density. As shown in Sec. 1, the properties of trajectory dataset have a significant impact on query efficiency. We classify the first three datasets into two types: 1) datasets with large *SpS*, large *NoP*, and low *DoT*, e.g., AIS; 2) datasets with small *SpS*, small *NoP*, and high *DoT*, e.g., Porto and Beijing. These are two extreme cases. Other datasets from reality will be similar to one of them or in between, e.g., OSM. Notably, almost all datasets in existing studies [25, 28, 33] are from urban applications and share similar properties to Porto and Beijing.

[6] One setting refers to a choice of distance metric and a dataset, and note that VRE supports all advanced query types and widely-used distance metrics.
[7] 0.3 and 0.003 are roughly 33.3 kilometers and 333 meters respectively. The trajectories in OSM are from many countries, and we calculate its density by dividing the cardinality by 360*180.

Table 5: Parameters (Default value is highlighted)

Parameter	Value
Time Window	12h, **1d**, 1w, 2w, 1m
Spatial Window	0.001, 0.002, **0.003**, 0.004, 0.005
Threshold τ	0.001, 0.002, **0.003**, 0.004, 0.005
k	1, 2, 5, **10**, 20
Data Size (%)	25, 50, **100**, 200, 400
# of Cores	1, 2, 4, 8, 16, **32**

Parameter Setting. Table 5 shows the key parameters used. When we vary a parameter, other parameters are set to default value. We set spatial window size and threshold τ of AIS and OSM to be 100x and 10x larger than those in the rest datasets, respectively. For *Tb-Search*, *k-Search*, and *sub-Search*, query trajectories are randomly sampled from the dataset.

Machines. VRE runs on a single machine with 32-core CPU and 32GB Memory, and a customized HBase cluster with two nodes. Each node is connected to a Gigabit Ethernet switch and runs CentOS 7 with Hadoop 2.10.0.

Table 6: Storage Cost (MB) and Insertion Time (s)

	Disk	*TSM*	*SSM*	*SXM*	*SSX*	TSM_1	TSM_2	*TXM*	*PXX*
Cost	795	1167	1508.9	5535	1440.9	1291.6	1566.1	3455	7362
Ratio	1.0	1.5	1.9	6.7	1.8	1.62	2.0	4.34	9.6
Insert	-	83.8	99.1	214	94.7	81.8	112.4	183.8	624.5

6.3 Storage Cost of Different Schemas

We compare the storage cost of five storage schemas introduced in Sec. 4.4. Note that VRE_{TXM} and VRE_{PXX} have similar storage costs to TrajMesa and GeoMesa respectively. Since there is no need to split short trajectories, we conduct experiments on AIS which contains long trajectories. Disk means storing trajectories in a file with a *zip* compression. Ratio means the ratio of the storage cost of one schema w.r.t that of Disk. Insert means the data insertion time.

From Table 6, we have the following observations. 1) By comparing VRE_{TSM} and VRE_{TXM} (TrajMesa), the storage cost is reduced by 3x when using the secondary index, because there is only one replica data in VRE_{TSM}. A similar result can be found in VRE_{SSM} and VRE_{SXM}. 2) By comparing VRE_{SSM} and VRE_{SSX}, the storage cost of the metadata is 4% of the total storage cost. We will later show that with the metadata on our two-stage framework, the basic queries and advanced queries can be processed 2x faster. We believe this is acceptable in real applications. 3) With the secondary index, the ratio of the segment-based model (VRE_{SSM}) and trajectory-based model (VRE_{TSM}) are less than 2 while VRE_{PXX} (GeoMesa) is 9.6 and VRE_{HXM} (TrajMesa) is 4.34. 4) Insertion time is proportional to the storage cost.

Table 7: Impact of Secondary Index

	IDTQ (Porto)	*STRQ* (Porto)	*IDTQ* (AIS)	*STRQ* (AIS)
PK (ms)	33.16	91.32	11.75	6.91
SK (ms)	33.56	218	9.75	7.32

6.4 Effectiveness of Proposed Optimizations

In this section, we analyze the impact of our proposed optimizations: the secondary index mechanism (Sec. 6.4.1), the two-stage framework (Sec. 6.4.2), and the pushdown strategy (Sec. 6.4.3).

Settings. We propose *SR* index, *ST* index, and *IDT* index to answer *SRQ*, *STRQ*, and *IDTQ* directly (Sec. 4.3). *SRQ* is always employed in processing advanced queries (Sec. 5.2). To obtain optimal performance in processing advanced queries, we adopt *SR* index as the primary index. The algorithms for *Tb-Join* and *k-Join* are built on the algorithms for *Tb-Search* and *k-Search* respectively, so the impact of the proposed optimizations on the join query is similar to that of *Tb-Search* and *k-Search*, and hence we omit their results. We also exclude *sub-Search* due to its different processing algorithm.

6.4.1 Impact of Secondary Index. Although the secondary index mechanism reduces the storage cost significantly, it is worth discussing whether the efficiency degradation caused by extra disk access is acceptable. With *SR* index as the primary index, we evaluate the impact of the secondary index on *IDTQ* and *STRQ*.

From the results in Table 7, we find that the extra random access caused by the secondary index is only non-negligible in processing *STRQ* over Porto. We dig into our queries on Porto and observe that the candidate size in *IDTQ* is quite small (only about 100), while in *STRQ* it is more than 1000. The time used in the random access then becomes the major overhead.

If users care about query latency when facing a larger candidate size, we can add the metadata as one attached column in the secondary index, and hence we can obtain similar query latency to the primary index. The storage cost is still much smaller than one data replica in one index.

Table 8: Efficiency of Two-Stage Framework

	SRQ (Porto)	*Tb-Search* (Porto)	*SRQ* (AIS)	*Tb-Search* (AIS)
one-stage (s)	14.94	out-of-memory	2.24	2.64
two-stage (s)	6.37	1.86	0.47	0.22

6.4.2 Efficiency of Two-Stage Framework. In this section, we verify the efficiency of the two-stage framework and the effectiveness of the metadata. Under the two-stage framework, VRE prunes unsatisfied trajectories with metadata first to avoid fetching numerous full trajectories while it incurs another round of access.

Table 8 shows the performance of *SRQ* and *Tb-Search* on two different frameworks. In all cases, the latency of two-stage framework is lower. Due to high density of Porto, the candidate size is quite large. For example, the candidate size of one *SRQ* is about 441,205 while only 192,854 remains after pruning. If fetching the full trajectories from storage layer, it incurs a large network cost and memory consumption, which may easily cause out-of-memory (OOM) errors, e.g., *Tb-Search* on Porto. TrajMesa adopts the one-stage framework, which obviously is not efficient for trajectory processing from the above analysis. These results also confirm the effectiveness of metadata in pruning irrelevant trajectories.

6.4.3 Impact of Pushdown. We verify how our pushdown technique affects the overall efficiency in *Tb-Search* and *k-Search* where VRE can push some pruning strategies down to the storage layer. From Table 9, we first observe that pushing down our pruning strategies helps improve the performance of *Tb-Search* on Porto up to 2x while on AIS it slightly worsens the performance. To analyze the reasons, we select one query trajectory from Porto and AIS respectively and break down their performance into four parts: the time used to fetch candidates' metadata (**tc**), the time used for pruning (**tp**), the time used for fetching the full trajectories (**tf**), and the time used for calculating the distance (**td**). The results are shown in Table 10. Table 10 also includes the candidate size (**cs**), results size (**rs**) and total time (**total**). If the time is less than 10ms, we set it as 0. We can see that in all cases, **tc** is the main overhead, about 83.3%-93.2%. On Porto, **tc** without pushdown is 2x

slower because a large number of candidates without pushdown incur more network cost. We also should notice that **cs** without pushdown is 2333.2x slower than its pushdown counterpart. That means we cannot ignore the time used in evoking coprocessor and doing simple pruning in the storage layer. If the candidate size is small, the reduced data transfer cost on network can be offset by the extra cost in evoking coprocessor and pruning. This explains why *Tb-Search* has a lower latency without pushdown on AIS.

Remark. When the system pushes down the pruning strategies to the storage layer, it must be aware of the trade-off between the reduced cost on transferring and the increased processing cost in storage layer. For both *Tb-Search* and *k-Search*, we can do pushdown for trajectory datasets of high density. A better way is to decide whether or not to perform pushdown for every query on the fly by checking the density of surrounding trajectories. In contrast, TrajMesa [25], a state-of-the-art disk-based system, simply never pushes down any pruning strategies into the storage layer. From the analysis above, for high density trajectories, the performance of TrajMesa is rather limited.

Table 9: Efficiency of Pushdown Optimization

	Porto	Porto (w/o push)	AIS	AIS (w/o push)
Tb-Search (s)	0.81	1.86	0.45	0.41
k-Search (s)	3.36	3.58	3.48	2.41

Table 10: Time Breakdown of *Tb-Search* (Pushdown)

dataset	total (s)	rs	cs	tc (s)	tp (s)	tf (s)	td (s)
Porto	0.89	23	474	0.83	0.0	0.0	0.0
Porto (w/o pushdown)	2.17	23	1,105,945	1.82	0.0	0.0	0.0
AIS	0.1	3	3	0.09	0.0	0.0	0.0
AIS (w/o pushdown)	0.06	3	491	0.05	0.0	0.0	0.0

6.5 Query Efficiency w.r.t. Storage Schemas

In this section, we explore the impact of different storage schemas (Sec. 4.4) on the query processing time.

Settings. Similar to Sec. 6.3, we conduct experiments on AIS. Due to space limit, we only report the results on *Tb-Search* and *k-Search*. Results on all basic queries are in our technical report [8]. With the same reasons as in Sec. 6.4, we exclude the join queries and *sub-Search*. We also omit the point-based model which cannot achieve any competitive performance in all advanced queries due to its excessive cost of reconstructing the full trajectories.

Table 11: Time Breakdown of *Tb-Search* (Storage Schemas)

schema	total (ms)	tc (ms)	tp (ms)	tf (ms)	td (ms)
VRE_{TSM}	202	86	7	5	98
VRE_{TSM_1}	233	122	7	3	95
VRE_{SSM}	397	134	152	10	96
VRE_{TSM_2}	390	130	166	5	95

6.5.1 Tb-Search. Similar to Sec. 6.4.3, we break down *Tb-Search* into four parts. Table 11 shows a time breakdown of the best-performing query on VRE_{TSM}. We have the following observations. 1) In VRE_{TSM} and VRE_{TSM_1}, Metadata fetching and distance calculation are the main overhead about 91%-93% of the total time. In VRE_{SSM} and VRE_{TSM_2}, Pruning is a bottleneck because some pruning strategies in these two schemas cannot run in parallel, resulting in longer time. 2) Compared to VRE_{TSM_1}, VRE_{TSM} takes less time in fetching metadata. Recall the schema in Table 2, although the same number of entries are returned, the size of metadata in VRE_{TSM_1} is much larger. 3) More metadata leads to better pruning power, which can reduce the number of full trajectory fetching. In our sampled queries, although the number of fetched trajectories in VRE_{TSM_1}, VRE_{SSM}, and VRE_{TSM_2} is smaller than that in VRE_{TSM}, the time difference is marginal due to low trajectory density in the AIS dataset.

6.5.2 k-Search. *k-Search* works in an iterative manner and stops when the remaining trajectories can be pruned with the estimated bound (Algo. 1). A smaller number of iterations and shorter time in each iteration lead to a better performance. Again, we break down the time in the i^{th} iteration, $\mathbf{tt}_i$, into four parts, $\mathbf{tc}_i$, $\mathbf{tp}_i$, $\mathbf{tf}_i$, and $\mathbf{td}_i$. Similar trajectories are spatially close and hence the first iteration takes longer time compared to subsequent iterations. Thus, we only report the average time of the i^{th}-iteration, where $i > 1$.

Table 12 shows a time breakdown of the best-performing query on VRE_{TSM_1}. We find: 1) All storage schemas spend most of the time on the first iteration. Fetching the full trajectories and the actual distance calculation are two major overheads in the first iteration, because the average point count of each trajectory is extremely large in AIS. 2) The metadata in VRE_{TSM_1} (also in VRE_{TSM_2} and VRE_{SSM}) has more information about the original trajectory and makes our pruning strategies more effective than the metadata in VRE_{TSM}, thus leading to a lower **tf** and **td**. 3) More metadata in VRE_{TSM_1} also leads to higher **tc** and **tp** than VRE_{TSM}.

Table 12: Time Breakdown of *k-Search* (Storage Schemas)

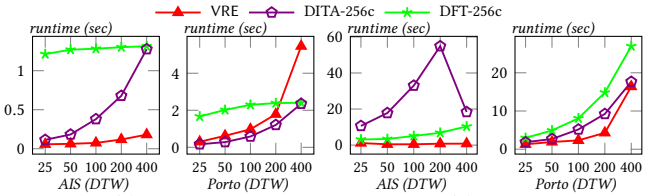
schema	iter	total (ms)	tt_1	tc_1	tp_1	tf_1	td_1	avg	post-pro
VRE_{TSM}	2	7726.0	7482	107	91	901	6472	157	0
VRE_{TSM_1}	2	4362	3990	127	269	483	2673	294	0
VRE_{SSM}	2	4537	3637	297	523	241	2463	759	88
VRE_{TSM_2}	2	4459	3574	305	197	227	2504	753	76

6.6 Comparison with Existing Systems

Here, we focus on the efficiency and scalability for advanced queries, and the results for basic queries are at [8].

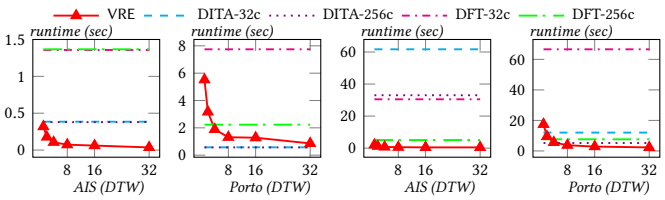
Competitors. We compare with two SOTA systems, DITA [28] and DFT [33], under their supported queries (see Table 1). Both are Spark-based in-memory systems while our VRE is disk-based. DITA supports *Tb-Search*, *k-Search*, and *Tb-Join* on Fréchet, DTW, EDR and a variant of LCSS. DFT supports *k-Search* on Fréchet and Hausdorff. We extend it to support DTW and *Tb-Search* on these three metrics. It is worth highlighting that the pruning designs in DITA and DFT prevent them from being extended to support any other metrics or queries. There are different reasons we do not compare other systems in Table 1: as discussed in Sec. 2.2, MobilityDB [14], UltraMan [19], and Summit [12] only support basic queries. REPOSE [41] only supports *k-Search* and the code is not released. As for TrajMesa [25], the code is not released and we have shown the advantages of our system in different optimizations compared to TrajMesa in Sec. 6.4. We choose VRE_{TSM_1} for AIS without pushdown and VRE_{TSM} with pushdown strategy for Porto, OSM, and Beijing based on our findings in Sec. 6.1. The results on Beijing are in our technical report [8].

Settings. VRE currently is designed to run on a single node and we use 32 cores by default. DITA and DFT run on a cluster with 32 nodes, each with an 8-core CPU and 32GB Memory (i.e., 256 cores by default). Note that although such a comparison is unfair



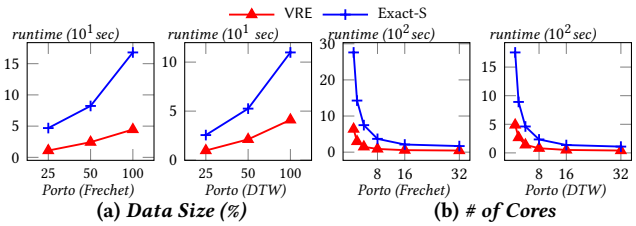
(a) *Tb-Search* (b) *k-Search*

Figure 3: Scalability for Data Size



(a) *Tb-Search* (b) *k-Search*

Figure 4: Scalability for Cores



(a) *Data Size (%)* (b) *# of Cores*

Figure 5: Scalability of *sub-Search*

to VRE, VRE is as competitive as DITA and DFT, which affirms the efficiency and economicality of VRE.

6.6.1 Tb-Search. We evaluate *Tb-Search* on AIS, Porto, and Beijing.

Varying the Threshold. From Table 13, we make the following observations. 1) Overall, except for EDR on Porto, VRE achieves much better or competitive performance with fewer resources. 2) With the increase of threshold, all methods take more time on their supported similarity metrics, since a larger threshold leads to more results. 3) In all datasets, VRE beats DFT by at least 3x. VRE directly issues *SRQ* to fetch the candidates and prune irrelevant trajectories with tailored pruning strategies in parallel while DFT first requires the index to get the bitmap of filtered trajectories, collects the bitmap at the master node, then searches the data with the collected bitmap to verify the similarity. Thus, it creates a barrier between indexing and verification. 4) In all datasets, DITA outperforms DFT, which is the same as the result reported in [28]. 5) All existing trajectory analytics systems, such as DFT and DITA, have poor performance to handle trajectories with the large point count and wide spatial range. On AIS, VRE significantly outperforms DFT and DITA by at least 5x and 2x respectively. 6) On Porto, DITA slightly outperforms VRE. This is because all trajectories in DITA have been loaded into the main memory while VRE needs to fetch them from the storage layer on the fly, which is the main overhead as shown in the analysis in Sec. 6.4.3. 7) On OSM, VRE beats DITA by at least 100x. We analyze the resources usage of DITA and find that DITA consumes a lot of memory, which becomes a bottleneck.

Scale-up. Figure 3(a) illustrates the processing time on different scales of AIS and Porto. With the increase of data size, all methods incur more time. On AIS, VRE outperforms both DFT and DITA, while VRE is slightly slower than DITA on Porto.

Scale-out. Figure 4(a) illustrates the processing time w.r.t. different core sizes on AIS and Porto. We only vary the number of cores for VRE. 1) As the number of cores increases, VRE achieves more performance gains since more workers run in parallel. 2) On AIS, VRE with only 1 core beats DITA-256c and DFT-256c. VRE with 4 cores obtains a lower latency than DFT-256c on Porto and is as competitive as DITA-256c on Porto. To summarize, VRE adopts fewer resources in processing layer to achieve similar (even better) performance, as compared to the existing Spark-based systems. It verifies that VRE is more economical.

6.6.2 k-Search. We evaluate *k-Search* on AIS, Porto, and Beijing.

Varying the k. From Table 14, we make the following observations. 1) Except for *k-Search* on AIS under EDR, VRE outperforms DFT and DITA by up to an order of magnitude on all datasets, queries, and distance metrics. DFT and DITA first estimate an upper bound of threshold to fetch at least k trajectories, and then transform *k-Search* to *Tb-Search*. The estimated threshold may not be tight enough, incurring more trajectories to be accessed. In contrast, VRE adopts an expanding manner and stops the exploration when the remaining trajectories could be pruned with the updated bound, which is quite efficient. 2) In all datasets, with the increase of k, all methods take more time since a larger k leads to more trajectories being checked. DFT and DITA cannot handle *k-Search* on OSM due to their large memory consumption. 3) On AIS, DFT beats DITA, as the large span and point count per trajectory on AIS limit the pruning effectiveness of Trie-like index in DITA. Moreover, compared with DFT, DITA needs more time to estimate the threshold by its Trie-like index while DFT adopts a sampling strategy to obtain the threshold.

Scale-up. Figure 3(b) illustrates the processing time on different scales of AIS and Porto. With the increase of dataset size, all methods spend more time and the gap between VRE and DITA, DFT tends to be larger since large data size introduces more computation and transmission costs. VRE achieves a better scalability with the tailored storage schema, pruning strategies, and optimizations. Note that DITA in 400% has lower latency than it in 200%. This is because with more trajectories, some trajectories can find the top-k trajectories with a smaller bound.

Scale-out. Figure 4(b) illustrates the processing time on different core sizes. 1) With the increase of cores, VRE performs better since more workers run in parallel. 2) On AIS, VRE obtains a lower latency than DITA-256c and competitive performance with DFT-256 only with 1 core. On Porto, VRE beats DITA-256c and DFT-256c with 4 cores and 8 cores respectively. Similarly, VRE adopts fewer resources in processing layer to achieve competitive performance with the existing Spark-based systems. This further verifies that VRE is more economical.

6.6.3 Tb-subSearch. Following the experimental setting in the most recent work [32], we evaluate our proposed algorithms for *sub-Search* on Porto. We adopt *Exact-S* [32] as the baseline method. Other methods in [32], e.g., *RLS-Skip*, are approximate algorithms, which cannot always find the most similar one. Figure 5 illustrates the performance of *sub-Search* with varying data size and core size.

Scale-up. From Figure 5(a), we observe that with the increase of data size, the gap between *Exact-S* and VRE tends to be bigger.

Table 13: Runtime (s) of *Tb-Search*

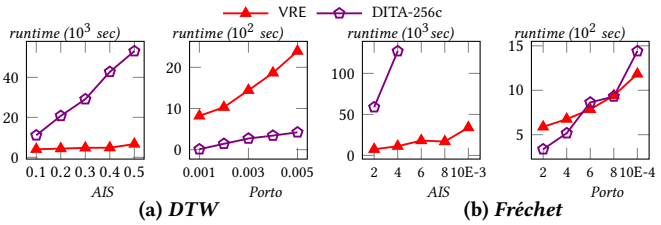
Distance Function	System	*AIS*					*Porto*					*OSM*				
		0.1	0.2	0.3	0.4	0.5	0.001	0.002	0.003	0.004	0.005	0.01	0.02	0.03	0.04	0.05
Hausdorff	DFT	1.37	1.40	1.38	1.43	1.54	2.48	2.39	2.37	2.66	3.31	-	-	-	-	-
	VRE	**0.19**	**0.25**	**0.38**	**0.55**	**0.64**	**0.73**	**0.76**	**0.80**	**0.87**	**0.96**	**0.11**	**0.12**	**0.15**	**0.18**	**0.21**
Fréchet	DFT	1.32	1.37	1.40	1.36	2.38	2.43	2.34	2.28	2.45	2.96	-	-	-	-	-
	DITA	0.60	0.82	0.98	1.11	1.26	**0.63**	**0.62**	**0.68**	**0.72**	**0.81**	44.69	45.85	46.64	46.60	46.91
	VRE	**0.24**	**0.26**	**0.41**	**0.56**	**0.67**	0.72	0.75	0.76	0.80	0.81	**0.09**	**0.11**	**0.15**	**0.17**	**0.19**
DTW	DFT	1.29	1.39	1.38	1.33	2.09	2.34	2.29	2.23	2.37	2.90	-	-	-	-	-
	DITA	0.35	0.36	0.38	0.37	0.43	**0.57**	**0.57**	**0.58**	**0.59**	**0.63**	49.40	42.71	43.74	43.78	43.10
	VRE	**0.20**	**0.17**	**0.17**	**0.18**	**0.18**	0.72	0.75	0.77	0.79	0.80	**0.10**	**0.09**	**0.09**	**0.09**	**0.01**
EDR		10	20	40	80	200	1	2	3	4	5	1	2	3	4	5
	DITA	1.76	1.71	1.77	1.80	1.96	**0.66**	**0.71**	**1.20**	**1.28**	**1.20**	131.72	390.00	-	-	-
	VRE	**0.31**	**0.18**	**0.18**	**0.17**	**0.18**	3.39	3.26	3.30	3.34	3.34	**0.09**	**0.08**	**0.08**	**0.08**	**0.08**
LCSS		0.80	0.85	0.90	0.95	1.0	0.80	0.85	0.90	0.95	1.0	0.80	0.85	0.90	0.95	1.0
	VRE	**0.23**	**0.19**	**0.18**	**0.17**	**0.07**	**4.86**	**4.50**	**4.43**	**4.41**	**3.34**	**0.11**	**0.09**	**0.09**	**0.09**	**0.09**

-: DFT or DITA crashed since it consumes too much memory on big datasets.

Table 14: Runtime (s) of *k-Search*

Distance Function	System	*AIS*					*Porto*					*OSM*				
		1	2	5	10	20	1	2	5	10	20	1	2	5	10	20
Hausdorff	DFT	6.68	2.86	3.42	3.81	3.83	16.32	15.16	15.55	16.65	15.80	-	-	-	-	-
	VRE	**0.89**	**1.42**	**2.09**	**2.76**	**3.42**	**1.75**	**1.55**	**1.62**	**1.67**	**1.77**	**0.10**	**0.13**	**0.15**	**0.27**	**0.55**
Fréchet	DFT	2.02	2.36	2.81	2.89	**2.94**	11.04	12.87	11.60	12.06	12.08	-	-	-	-	-
	DITA	2.34	3.02	4.12	6.84	11.26	2.69	3.17	2.82	2.99	3.27	-	-	-	-	-
	VRE	**0.52**	**0.73**	**1.30**	**2.01**	3.31	**1.40**	**1.29**	**1.31**	**1.32**	**1.34**	**0.09**	**0.14**	**0.16**	**0.29**	**0.60**
DTW	DFT	3.84	4.20	5.02	5.05	5.72	7.68	7.57	7.68	7.62	7.71	-	-	-	-	-
	DITA	3.61	32.45	32.84	33.01	38.67	4.98	4.93	4.72	5.18	5.70	-	-	-	-	-
	VRE	**1.15**	**1.27**	**1.81**	**2.86**	**4.02**	**2.07**	**2.07**	**2.17**	**2.28**	**2.52**	**0.10**	**0.13**	**0.16**	**0.22**	**0.50**
EDR	DITA	**3.17**	**4.88**	**4.98**	**5.47**	**6.29**	14.75	15.99	15.56	16.43	16.46	-	-	-	-	-
	VRE	4.36	5.50	6.04	8.27	9.28	**1.86**	**3.82**	**4.15**	**4.47**	**4.77**	**0.16**	**0.18**	**0.19**	**0.22**	**0.25**
LCSS	VRE	**1.09**	**1.46**	**1.49**	**1.57**	**1.58**	**7.42**	**7.30**	**7.34**	**7.35**	**7.35**	**0.28**	**0.27**	**0.27**	**0.27**	**0.29**

-: DFT or DITA crashed since it consumes too much memory on big datasets.



(a) *DTW* **(b) *Fréchet***

Figure 6: Efficiency of the *Tb-Join*

For example, when we increase the data size from 50% to 100%, *Exact-S*' time increases from 52.53 seconds to 109.8 seconds, while VRE increases from 21.2 seconds to 41 seconds. VRE achieves a better scalability with the tailored algorithms.

Scale-out. From Figure 5(b), we have the following observations. 1) When the core size equals 1, *Exact-S* is about 4x slower than VRE, which is similar to *RLS-Skip* and *RLS* [32]. However, these learning-based methods cannot always find the most similar one for the issued queries and need lots of time to train the model. 2) With the increase of core size, all methods gain the performance improvement. The gap between *Exact-S* and VRE tends to be smaller.

6.6.4 Join Query. Figure 6 illustrates the results of *Tb-Join*. We have the following observations. 1) The time used in VRE and DITA is increased with a larger threshold as more pairs are generated. 2) VRE outperforms DITA on AIS and is as competitive as DITA on Porto under Fréchet. 3) DITA outperforms VRE by about 8x on Porto under DTW. Note that we use 32 cores for VRE and 256 cores for DITA. Considering the parallel processing we have designed, we believe that VRE is as competitive as DITA. 4) The average time of each trajectory to find their joinable trajectories is about 14 ms, which is 13x faster than *Tb-Search* on AIS. It shows the effectiveness of our proposed algorithm. A similar result is also in Porto.

Due to the high complexity of *k-Join*, we conduct an experiment on 25% Porto with $k = 2$ and the distance function is Fréchet. VRE spends 4, 300s on this query. The average time of each trajectory is about 11ms. Due to space limit, more results are in Appendix G of [8].

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new trajectory management system, VRE. The core part of VRE has deployed in Ganos, the spatio-temporal data engine in Alibaba. To our best knowledge, VRE is the first system that supports all basic and advanced query types and distance functions while incurring much lower storage cost for trajectories. We also present our insights through the study and guidelines for optimal storage schema selection in VRE. These insights and guidelines give the lessons for future trajectory systems' design. The main direction of our future work is threefold: (1) enable VRE with the ability to handle the queries across multiple nodes, which is crucial for analytical queries, e.g., join; (2) introduce a cost-aware optimizer to choose the best execution strategy for every incoming query instead of one strategy for one specific dataset; (3) build a benchmark for trajectory data system.

REFERENCES

[1] July, 2022. AIS Dataset. https://marinecadastre.gov/ais/.
[2] July, 2022. GeoMesa. https://www.geomesa.org/.
[3] July, 2022. HBase. https://hbase.apache.org/.
[4] July, 2022. OSM Trace. https://www.openstreetmap.org/traces.
[5] July, 2022. Porto Dataset. https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data.
[6] July, 2022. PostGIS. https://postgis.net/.
[7] July, 2022. PostgreSQL Database. http://www.postgresql.org/.
[8] July, 2022. VRE's technical report. https://github.com/hailanwhu/vre_artifacts.
[9] July, 2022. Zstandard. https://facebook.github.io/zstd/.
[10] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. 2013. Computing the Discrete Fréchet Distance in Subquadratic Time. In *SODA*. 156–167.
[11] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB* 6, 11 (2013), 1009–1020.
[12] Louai Alarabi. 2018. Summit: a scalable system for massive trajectory data management. In *SIGSPATIAL*. 612–613.
[13] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. 2018. ST-Hadoop: a MapReduce framework for spatio-temporal data. *GeoInformatica* 22, 4 (2018), 785–813.
[14] Mohamed S. Bakli, Mahmoud Attia Sakr, and Esteban Zimányi. 2019. Distributed moving object data management in MobilityDB. In *BigSpatial@SIGSPATIAL*. 1:1–1:10.
[15] Christian Böhm, Gerald Klump, and Hans-Peter Kriegel. 1999. XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In *SSD (Lecture Notes in Computer Science)*, Vol. 1651. 75–90.
[16] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. 2005. On Map-Matching Vehicle Tracking Data. In *VLDB*. 853–864.
[17] Lei Chen, M. Tamer Özsu, and Vincent Oria. 2005. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*. 491–502.
[18] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2010. TrajStore: An adaptive storage system for very large trajectory data sets. In *ICDE*. 109–120.
[19] Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. 2018. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *PVLDB* 11, 7 (2018), 787–799.
[20] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*. 1352–1363.
[21] Chenjuan Guo, Bin Yang, Jilin Hu, and Christian S. Jensen. 2018. Learning to Route with Sparse Trajectory Sets. In *ICDE*. 1073–1084.
[22] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*. 497–506.
[23] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. JUST: JD Urban Spatio-Temporal Data Engine. In *ICDE*. 1558–1569.
[24] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System. *TKDE* (2021), 1–1.
[25] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data. In *ICDE*. 2002–2005.
[26] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. 2009. Map-matching for low-sampling-rate GPS trajectories. In *SIGSPATIAL*. 352–361.
[27] Sarana Nutanong, Edwin H. Jacox, and Hanan Samet. 2011. An Incremental Hausdorff Distance Calculation Algorithm. *PVLDB* 4, 8 (2011), 506–517.
[28] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed In-Memory Trajectory Analytics. In *SIGMOD*. 725–740.
[29] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9, 13 (2016), 1565–1568.
[30] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. 2002. Discovering Similar Multidimensional Trajectories. In *ICDE*. 673–684.
[31] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, and Gao Cong. 2021. A Survey on Trajectory Data Management, Analytics, and Learning. *ACM Comput. Surv.* 54, 2 (2021), 39:1–39:36.
[32] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. 2020. Efficient and Effective Similar Subtrajectory Search with Deep Reinforcement Learning. *PVLDB* 13, 11 (2020), 2312–2325.
[33] Dong Xie, Feifei Li, and Jeff M. Phillips. 2017. Distributed Trajectory Similarity Search. *PVLDB* 10, 11 (2017), 1478–1489.
[34] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*. 1071–1085.
[35] Can Yang and Gyözö Gidófalvi. 2018. Fast map matching, an algorithm integrating hidden Markov model with precomputation. *International Journal of Geographical Information Science* 32, 3 (2018), 547–570.
[36] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. 1998. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE*. 201–208.
[37] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial data management in apache spark: the GeoSpark perspective and beyond. *GeoInformatica* 23, 1 (2019), 37–78.
[38] Haitao Yuan and Guoliang Li. 2019. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *ICDE*. 1262–1273.
[39] Haitao Yuan, Guoliang Li, Zhifeng Bao, and Ling Feng. 2020. Effective Travel Time Estimation: When Historical Trajectories over Road Networks Matter. In *SIGMOD*. 2135–2149.
[40] Ping Zhang, Zhifeng Bao, Yuchen Li, Guoliang Li, Yipeng Zhang, and Zhiyong Peng. 2018. Trajectory-driven Influential Billboard Placement. In *KDD*. 2748–2757.
[41] Bolong Zheng, Lianggui Weng, Xi Zhao, Kai Zeng, Xiaofang Zhou, and Christian S. Jensen. 2021. REPOSE: Distributed Top-k Trajectory Similarity Search with Local Reference Point Tries. *CoRR* abs/2101.08929 (2021).

A QUERY DEFINITION

Definition A.1 (ID Temporal Query (IDTQ)). Given a trajectory set $\mathcal{T}$, an object id id, a temporal range $R_t = [t_1, t_2]$, *IDTQ* returns the trajectories in $\mathcal{T}$ whose object id is id and each of which has at least one point in R_t.

Definition A.2 (Spatial Range Query (SRQ)). Given a trajectory set $\mathcal{T}$, and a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$, *SRQ* returns the trajectories in $\mathcal{T}$ each of which has at least one point in R_s.

Definition A.3 (Spatio-Temporal Range Query (STRQ)). Given a trajectory set $\mathcal{T}$, a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$, and a temporal range $R_t = [t_1, t_2]$, *STRQ* returns the trajectories in $\mathcal{T}$ each of which has at least one point with the spatial position in R_s and the timestamp in R_t.

Definition A.4 (Threshold-based Trajectory Similarity Search (Tb-Search)). Given a query trajectory Q, a trajectory set $\mathcal{T}$, a distance function f (e.g., DTW), and a threshold τ, *Tb-Search* returns the trajectories in $\mathcal{T}$ whose distance to Q under f is less than τ.

Definition A.5 (Top-k Trajectory Similarity Search (k-Search)). Given a query trajectory Q, a distance function f, and an integer k, *k-Search* returns the set K of k trajectories, K ($K \subset \mathcal{T}$), whose distances to Q are less than the other trajectories in $\mathcal{T} \setminus K$ to Q.

Currently, in VRE, f can be DTW [36], EDR [17], Fréchet [10], Hausdorff [27], and LCSS [30]

Definition A.6 (Subtrajectory Similarity Search (sub-Search)). Given a query trajectory Q, a trajectory set $\mathcal{T}$ and a distance function f, *sub-Search* returns the subtrajectories each of which is the most similar to Q under f in its corresponding full trajectory.

Definition A.7 (Threshold-based Trajectory Similarity Join (Tb-Join)). Given two trajectory sets $\mathcal{T}_1$ and $\mathcal{T}_2$, a distance function f, and a threshold τ, *Tb-Join* returns all trajectory pairs $(T_i, T_j) \in \mathcal{T}_1 \times \mathcal{T}_2$ whose distance between T_i and T_j is less than τ.

Definition A.8 (Top-k Trajectory Similarity Join (k-Join)). Given two trajectory sets $\mathcal{T}_1$ and $\mathcal{T}_2$, a distance function f, and an integer k, *k-Join* returns all trajectory pairs (T_i, T_j) where T_i is from $\mathcal{T}_1$ and T_j is one of the k-nearest neighbors of T_i in $\mathcal{T}_2$.

B QUERY DEFINITION

Definition B.1 (ID Temporal Query (IDTQ)). Given a trajectory set $\mathcal{T}$, an object id id, a temporal range $R_t = [t_1, t_2]$, *IDTQ* returns the trajectories in $\mathcal{T}$ whose object id is id and each of which has at least one point in R_t.

Definition B.2 (Spatial Range Query (SRQ)). Given a trajectory set $\mathcal{T}$, and a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$, *SRQ* returns the trajectories in $\mathcal{T}$ each of which has at least one point in R_s.

Definition B.3 (Spatio-Temporal Range Query (STRQ)). Given a trajectory set $\mathcal{T}$, a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$, and a temporal range $R_t = [t_1, t_2]$, *STRQ* returns the trajectories in $\mathcal{T}$ each of which has at least one point with the spatial position in R_s and the timestamp in R_t.

Definition B.4 (Threshold-based Trajectory Similarity Search (Tb-Search)). Given a query trajectory Q, a trajectory set $\mathcal{T}$, a distance function f (e.g., DTW), and a threshold τ, *Tb-Search* returns the trajectories in $\mathcal{T}$ whose distance to Q under f is less than τ.

Definition B.5 (Top-k Trajectory Similarity Search (k-Search)). Given a query trajectory Q, a distance function f, and an integer k, *k-Search* returns the set K of k trajectories, K ($K \subset \mathcal{T}$), whose distances to Q are less than the other trajectories in $\mathcal{T} \setminus K$ to Q.

Currently, in VRE, f can be DTW [36], EDR [17], Fréchet [10], Hausdorff [27], and LCSS [30]

Definition B.6 (Subtrajectory Similarity Search (sub-Search)). Given a query trajectory Q, a trajectory set $\mathcal{T}$ and a distance function f, *sub-Search* returns the subtrajectories each of which is the most similar to Q under f in its corresponding full trajectory.

Definition B.7 (Threshold-based Trajectory Similarity Join (Tb-Join)). Given two trajectory sets $\mathcal{T}_1$ and $\mathcal{T}_2$, a distance function f, and a threshold τ, *Tb-Join* returns all trajectory pairs $(T_i, T_j) \in \mathcal{T}_1 \times \mathcal{T}_2$ whose distance between T_i and T_j is less than τ.

Definition B.8 (Top-k Trajectory Similarity Join (k-Join)). Given two trajectory sets $\mathcal{T}_1$ and $\mathcal{T}_2$, a distance function f, and an integer k, *k-Join* returns all trajectory pairs (T_i, T_j) where T_i is from $\mathcal{T}_1$ and T_j is one of the k-nearest neighbors of T_i in $\mathcal{T}_2$.

C OTHER INDEXING STRATEGIES

IDT Index. The key is defined as *shard* + *sid*, where *shard* is a random number to distribute data across region servers for load balance to make the data evenly distributed on each node of the customized HBase and *sid* is the segment id.

ST Index. Inspired by XZ2T [23] indexing scheme, we propose *ST* index for *STRQ*. The key is $shard + Bin_1(T[i].t) + Bin_2(T[i].t) + XZ2(mbr_l)$ + *sid*. We design a novel two-level bin to represent the time dimension in *ST*. The timeline is divided into multiple disjoint time period bins to represent a range for temporal dimension. For each time bin, we collect the segments whose start time is in it. The first-level bin indicates which time period bin that a segment's start time $T[i].t$ belongs to. We define this as $Bin_1(T[i].t) = \left\lfloor \frac{T[i].t - RefTime}{BinLen_1} \right\rfloor$, where $RefTime$ is the reference time 1970-01-01T00:00:00, and $BinLen_1$ is the number of seconds in the first-level bin (it is 86, 400 in our implementation). In practice, we find that many false positive trajectories are returned if *ST* index only has the first level bin. Thus, we add the second-level bin to represent the time dimension in a finer-grained way. We define this as $Bin_2(T[i].t) = \left\lfloor \frac{T[i].t - Bin_1(T[i].t) \times BinLen_1}{BinLen_2} \right\rfloor$, where $BinLen_2$ is the number of seconds in second level bin (it is 600 in our implementation). Similar to *SR* index, we append the XZ2 code and *sid*.

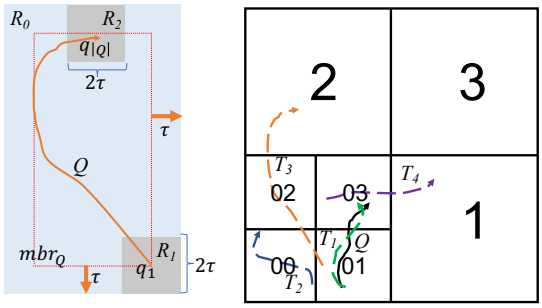
D BASIC QUERY PROCESSING

Spatial Range Query. Given an *SRQ* q with a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$, based on the index of *SR*, the query window generation contains the following steps: i) *shard* generation: it is the same with *IDTQ*. ii) *spatial key range* generation: this step generates a list of key ranges by computing the $XZ2(R_s)$. To avoid the high time cost of this step, we set a parameter to limit the maximal number of ranges generated. Finally, we combine the *shard* and the key ranges to generate query windows. VRE triggers SCAN operations over the *SR* index in parallel, where each query

Algorithm 3: Threshold-based Search

Input: Query trajectory Q, trajectory dataset $\mathcal{T}$, trajectory distance function $f(\cdot,\cdot)$, distance threshold τ
Output: A set of trajectories $\mathcal{T}_{sim}$

Compute R_0, R_1, R_2 as shown in Figure 8(a);
Candidates $C_s = SRQ_{meta}(R_0, \mathcal{T})$;
$C_s = C_s - \{s \in C_s | mbr_s$ is not fully contained in $R_0\}$;
Group the segments in C_s by its tid, generate a HashMap $C_t = (tid, List\langle segment\rangle)$;
foreach $(tid, G) \in C_t$ **do**
  Sort the segments in G by serial number l;
  Completeness Pruning (I); **LB_SES** Pruning (II) ;
  LB_PartialSim Pruning (IV); **LB_SIG** Pruning (V);
  $T = trajConstructor(OTS(tid, \mathcal{T}))$;
  Sample pivot points from T; **LB_Pivots** Pruning (IV);
  if $f(Q, T) \le \tau$ **then**
    $\mathcal{T}_{sim} \leftarrow T$;



(a) Tb-Search **(b) k-Search**

Figure 8: Example of Trajectory Similarity Search

window is transformed into an execution. Here, we only guarantee that the MBR of the returned segments or trajectories intersect with R_s. Under the two-stage framework, the metadata is returned only. VRE first uses the metadata to prune the irrelevant trajectories or segments. Then, VRE fetches the original trajectories or segments with OTS. Note that the $IDTQ$ and $STRQ$ follows the same processing. Last, a verification process is needed to check whether there are points in R_s for each fetched trajectory.

ID Temporal Query. Given an $IDTQ$ q with a time range $[tq_s, tq_e]$ and a moving object id, based on the index of IDT, the query window generation contains the following steps: i) *shard* generation: this step enumerates all possible values of *shard*. ii) *temporal key range* generation: since IDT uses start time of segments directly, the given $[tq_s, tq_e]$ needs to be expanded to $[\left\lfloor \frac{tq_s}{maxDuration} \right\rfloor \times maxDuration, tq_e]$ where $maxDuration$ is the same as that in splitting strategies. Thus, the situations of T_1 and T_2 can be included in result set as shown in Figure 7. Finally, we combine the *shard*, the given id and the temporal key ranges to generate query windows. VRE triggers SCAN operations over the IDT index in parallel. Due to the expansion of time range, unqualified trajectories could be retrieved, such as T_0 in Figure 7. Thus, VRE needs to verify the returned trajectories.

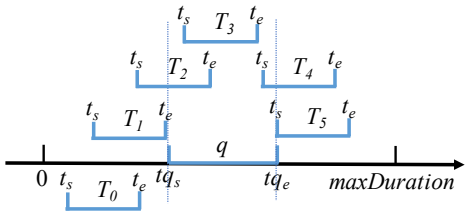


Figure 7: The illustration of *IDTQ* Query Window

Spatio-temporal Range Query. Given an $STRQ$ q with a spatial range $R_s = [lng_{min}, lat_{min}, lng_{max}, lat_{max}]$ and a time range $[tq_s, tq_e]$, based on the index of ST, the query window generation contains the following steps: i) *shard* generation: it is the same with $IDTQ$. ii) $BinNum$ generation: this step finds a list of time period bins that are overlapped with the given time range $[tq_s, tq_e]$. Recall the ST index, we need to calculate the range of Bin_1 and Bin_2 by the given time range. iii) *spatial key range* generation: it is the same with SRQ. Finally, we combine the *shard*, the Bin_1, the Bin_2 and the spatial key ranges to generate query windows. VRE triggers SCAN operations over the IDT index in parallel. A verification process is needed to check both spatial condition and temporal condition.

E ADVANCED QUERY

E.1 Threshold-based Search Pseudocode

Algorithm 3 gives the pseudocode for *Tb-Search*.

E.2 Example of *k-Search*

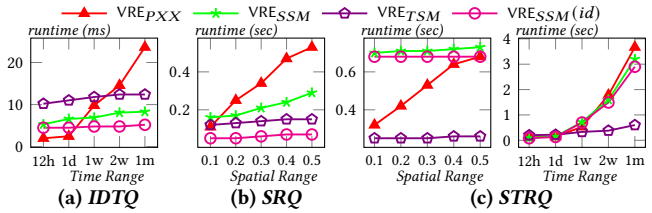
Example E.1. Figure 8(b) gives an example of *k-Search* with $k = 1, g = 2$ and $segNum = 3$: 1) Pop MBR <01>, trigger a spatial range query, and return a segment of T_3 and three segments of T_1. 2) Apply pruning strategies in Sec. 5.2.3 based on the metadata of these segments. 3) Fetch the full trajectory T_1, calculate the similarity by the given distance function, and add T_1 to cdq. 4) Check if the distance between the next MBR (i.e. <03>) and Q is larger than d_{max}. 5) Trigger a spatial range query on MBR <03> and return three segments of T_4 and two segments of T_1. 6) Since T_1 is already in cdq, we only need to process T_4. Fetch full trajectory T_4, calculate its similarity with Q, and update cdq and d_{max}. 7) The distance between the remaining of MBRs <00,02,1,2,3> and Q is larger than d_{max}, so the query processing is terminated.

E.3 Pseudocode Of Subtrajectory Similarity Search On Hausdorff

Algorithm 4 gives the pseudocode for *sub-Search* on Hausdoff. Here, we design a divide-and-conquer algorithm. We break down the problem into three sub-problems (Line 5-7). When the subtrajectory size is 2, we adopt the naive distance method to calculate the distance (Line 1, 2). In $SubSimH_{cross}$, we first extend the subtrajectory $T[mid, mid+1]$ in the left part and stop extend operation when new point will lead a larger distance between Q and current subtrajectory. Then, we extend the subtrajectory $T[ns, mid+1]$ in the right part, where ns is the stop position in last step.

F LCSS PROOF

Given two trajectories T and Q, an integer $\sigma \ge 0$ and a matching threefold $\epsilon \ge 0$, the definitions of LCSS in DITA [28] and the



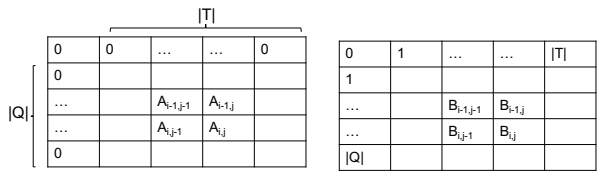
(a) *IDTQ* **(b) *SRQ*** **(c) *STRQ***

Figure 10: Runing Time of Basic Query

Algorithm 4: Subtrajectory Similarity Search (*SubSimH*)

Input: Query trajectory Q, candidate trajectory T, trajectory distance function $f(\cdot,\cdot)$, *start* offset of T and *end* offset of T
Output: Most similar subtrajectory $T[i,j]$ to Q

if $end - start = 1$ **then**
  return $f(T[start, end], Q)$
else
  mid=$(start + end)/2$;
  $(ls, le, ld) = SubSimH(Q, T, start, mid)$;
  $(rs, re, rd) = SubSimH(Q, T, mid + 1, end)$;
  $(cs, ce, cd) = SubSimH_{cross}(Q, T, start, mid, end)$;
  if $ld \le rd$ *and* $ld \le cd$ **then**
    return (ls, le, ld)
  if $rd \le ld$ *and* $rd \le cd$ **then**
    return (rs, re, rd)
  return (cs, ce, cd)



(a) $LCSS^o(T, Q)$ **(b)** $LCSS^d(T, Q)$

Figure 9: Distance under LCSS

original paper [30] are in Eq.(3) and Eq.(4) respectively.

$$\mathrm{LCSS}^{\mathrm{d}}_{\sigma,\epsilon}(T,Q) = \begin{cases} |Q| & \text{if } |T| = 0 \\ |T| & \text{if } |Q| = 0 \\ \mathrm{LCSS}^{\mathrm{d}}_{\sigma,\epsilon}(T^{|T|-1}, Q^{|Q|-1}) & \text{if } ||T| - |Q|| \le \sigma \& dist(T_{|T|}, Q_{|Q|}) \le \epsilon \\ 1 + min(\mathrm{LCSS}^{\mathrm{d}}_{\sigma,\epsilon}(T^{|T|-1}, Q), & \\ \mathrm{LCSS}^{\mathrm{d}}_{\sigma,\epsilon}\ T, Q^{|Q|-1})) & \text{otherwise} \end{cases} \tag{3}$$

$$\mathrm{LCSS}^{\mathrm{o}}_{\sigma,\epsilon}(T,Q) = \begin{cases} 0 & \text{if } |T| = 0 \\ 0 & \text{if } |Q| = 0 \\ 1 + \mathrm{LCSS}^{\mathrm{o}}_{\sigma,\epsilon}(T^{|T|-1}, Q^{|Q|-1}) & \text{if } ||T| - |Q|| \le \sigma \& dist(t_{|T|}, q_{|Q|}) \le \epsilon \\ max(\mathrm{LCSS}^{\mathrm{o}}_{\sigma,\epsilon}(T^{|T|-1}, Q), & \\ \mathrm{LCSS}^{\mathrm{o}}_{\sigma,\epsilon}\ T, Q^{|Q|-1})) & \text{otherwise} \end{cases} \tag{4}$$

Here, we first use mathematical induction prove that

$$2LCSS^o_{\sigma,\epsilon}(T,Q) + LCSS^d_{\sigma,\epsilon}(T,Q) = |T| + |Q| \tag{5}$$

Proof. In figure. 9, $A_{i,j}$ denotes the similarity of $Q[1,i]$ and $T[1,j]$ under the original LCSS definition while $B_{i,j}$ denotes that under the LCSS definition in DITA.

Base case. When $|Q| = 0$ or $|T| = 0$, i.e., in the first column or row in Figure 9, $2A_{i,j} + B_{i,j} = i + j$, where $i = 0$ or $j = 0$.

Inductive step. Suppose Eq.(5) still holds between $A_{m-1,n}$ and $B_{m-1,n}$, $A_{m,n-1}$ and $B_{m,n-1}$, $A_{m-1,n-1}$ and $B_{m-1,n-1}$.

Case 1: When $i = m$ and $j = n$, if Q_m and T_n match, i.e., in third case, we have $A_{m,n} = 1 + A_{m-1,n-1}$ and $B_{m,n} = B_{m-1,n-1}$. $2A_{m,n} + B_{m,n} = 2 + 2A_{m-1,n-1} + B_{m-1,n-1} = m + n$.

Case 2: If Q_m and T_n do not match and $A_{m-1,n} > A_{m,n-1}$, we have $B_{m-1,n} < B_{m,n-1}$. $2A_{m,n} + B_{m,n} = 2A_{m-1,n} + 1 + B_{m-1,n} = m + n$. Similar result is shown in $A_{m-1,n} \le A_{m,n-1}$.

Since both the base case and the inductive step have been proved as true, by mathematical induction Eq.(5) holds.

Given a query trajectory Q, threshold θ, and a trajectory set $\mathcal{T}$, *Tb-Search* finds the trajectory T that $LCSS^o_{\sigma,\epsilon}(T,Q) \ge \theta, T \in \mathcal{T}$. According to Eq.(5), it is equivalently to find the trajectory T that $LCSS^d_{\sigma,\epsilon}(T,Q) \le (|Q|+|T|-2\theta)/2, T \in \mathcal{T}$. Thus, for candidates with different lengths, they have different thresholds under the distance $LCSS^o_{\sigma,\epsilon}(T,Q)$. However, in the pruning framework of DITA, it does not differentiate the trajectories with different lengths.

Moreover, suppose we set the threshold in $LCSS^o$ with θ^o and the threshold in $LCSS^d$ with θ^d. Here exists a $|T'|$ which makes Eq.(5) satisfied. For trajectories in $\mathcal{T}$ whose length is larger than $|T'|$, they should have a larger threshold under $LCSS^d$, i.e., larger than θ^d, according to Eq.(5). Under the current DITA framework, they may be pruned (false negative).

G ADDITIONAL EXPERIMENTS

G.1 Basic Queries.

To evaluate the performance of basic queries, we compare VRE$_{PXX}$, VRE$_{TSM}$, VRE$_{SSM}$ on AIS. For segment-based storage schema, we set the values of splitting duration, splitting distance, sliding rectangle, and splitting point number (Sec. 4.1) with 24 hours, 5 km, 1 km, and 600, respectively. We do not compare UltraMan [19] and TrajMesa [25] due to the lack of open-sourced code. Moreover, we have shown the advancement of our proposed two-stage framework compared to one-stage framework (the framework used in TrajMesa) in Sec. 6.4.2. Results are shown in Figure 10. Note that VRE$_{PXX}$ only returns the satisfied points and trajectory id can be obtained by the returned points. VRE$_{SSM}(id)$ returns the trajectory ids, which pass the queries. From Figure 10, we have the following observations. 1) If the result type is full trajectory, VRE$_{TSM}$ beats VRE$_{SSM}$ in *SRQ* and *STRQ* while it is slower than VRE$_{SSM}$ in *IDTQ*. This is because the trajectory reconstructing in VRE$_{SSM}$ cannot be ignored when the candidate size is large. However, the pruning power of VRE$_{SSM}$ is better than VRE$_{TSM}$, and therefore when candidate size is small, VRE$_{SSM}$ outperforms VRE$_{TSM}$, e.g., *IDTQ* and *SRQ* with the small-time ranges. 2) If the result type is trajectory id, VRE$_{PXX}$ is better or competitive to VRE$_{TSM}$ and VRE$_{SSM}$ in small query ranges (spatial range or time range). As the query range becomes larger, more rows are scanned in VRE$_{PXX}$ and the scanning time becomes the main overhead.

G.2 Experiments on Beijing datasets

Table 15 and Table 16 show the runtime time *Tb-Search* and *k-Search* on Beijing datasets. In all cases, VRE beats DFT and DITA at least 2x. In *Tb-Search*, VRE only needs to fetch the partial of trajectories and uses efficient pruning framework. In *k-Search*, DFT and DITA first estimate an upper bound of threshold to fetch at least k trajectories, and then transform *k-Search* to *Tb-Search*. The estimated threshold may not be tight enough, which incurs more

Table 15: Runtime (s) of *Tb-Search*

Distance Function	System	*Beijing*				
		0.001	0.002	0.003	0.004	0.005
Hausdorff	DFT	-	-	-	-	-
	VRE	**0.25**	**0.26**	**0.31**	**0.35**	**0.39**
Fréchet	DFT	-	-	-	-	-
	DITA	1.84	1.78	1.85	1.82	1.90
	VRE	**0.25**	**0.27**	**0.30**	**0.31**	**0.36**
DTW	DFT	-	-	-	-	-
	DITA	1.78	1.74	1.80	1.76	1.78
	VRE	**0.23**	**0.23**	**0.25**	**0.26**	**0.28**
		1	2	3	4	5
EDR	DITA	1.96	1.99	2.52	2.45	2.39
	VRE	**0.81**	**0.66**	**0.67**	**0.69**	**0.75**
		0.80	0.85	0.90	0.95	1.0
LCSS	VRE	**1.73**	**1.67**	**1.68**	**1.67**	**1.68**

-: DFT crashed since it consumes too much memory on big datasets.

Table 16: Runtime (s) of *k-Search*

Distance Function	System	*Beijing*				
		1	2	5	10	20
Hausdorff	DFT	-	-	-	-	-
	VRE	**0.57**	**0.54**	**0.54**	**0.56**	**0.59**
Fréchet	DFT	-	-	-	-	-
	DITA	6.11	6.32	6.14	6.41	6.73
	VRE	**0.52**	**0.50**	**0.48**	**0.47**	**0.54**
DTW	DFT	-	-	-	-	-
	DITA	6.41	6.39	6.42	6.53	6.89
	VRE	**0.59**	**0.55**	**0.58**	**0.60**	**0.64**
EDR	DITA	8.97	10.55	9.15	10.30	9.89
	VRE	**1.11**	**1.60**	**1.70**	**1.89**	**2.02**
LCSS	VRE	**2.39**	**2.28**	**2.25**	**2.72**	**2.87**

-: DFT crashed since it consumes too much memory on big datasets.

trajectories to be accessed. In contrast, VRE adopts an expanding manner and stops the exploration when the remaining trajectories could be pruned with the updated bound, which is quite efficient. It is interesting that the runtime (*Tb-Search* and *k-Search*) of VRE on OSM dataset is smaller than that on Beijing while the dataset size of OSM is about 10x of Beijing. This is because the average size of candidates for the queries on OSM is much smaller than that on Beijing. Thus, VRE uses less time to transfer the candidates from the storage layer to the processing layer, and to do the pruning processing.

G.3 Top-k Join

Table 17 shows the running time of *k-Join* on 25 %Porto and distance function is Fréchet.

Table 17: Running time(s) of *k-Join*

1	2	3	4	5	10
1974.3	4300.5	5600.7	8201	9886.5	13073.5

H DISCUSSION ON DISTANCE FUNCTION SUPPORT

Here, we give the analyses on the problems when extending the existing systems to support other distance functions. DFT models the trajectories into the segments (two points in one segment) and check if the overlap exists between the segments from the dataset and the extended region of query trajectory (Figure 5 in [33]). If any segment of a trajectory does not intersect with the query, it will be pruned. The similar idea also can be applied on DTW while it is not satisfied in LCSS and EDR. This is because partial points can be out of the extended region in LCSS and EDR. Moreover, the pruning bound of DTW is not tight because DFT does not consider the accumulated distance from all segments of the trajectory. If we extend DITA to support Hausdoff, DITA cannot prune any trajectories until it accesses all nodes in the Tire-like index structure and obtains the minimum distance between the nodes and the query. Due to the high cost in traversing the tree in DITA, the efficiency of this way is limited.