Distributed Trajectory Similarity Search

Dong Xie, Feifei Li, Jeff M. Phillips
School of Computing, University of Utah
{dongx, lifeifei, jeffp}@cs.utah.edu

ABSTRACT

Mobile and sensing devices have already become ubiquitous. They have made tracking moving objects an easy task. As a result, mobile applications like Uber and many IoT projects have generated massive amounts of trajectory data that can no longer be processed by a single machine efficiently. Among the typical query operations over trajectories, similarity search is a common yet expensive operator in querying trajectory data. It is useful for applications in different domains such as traffic and transportation optimizations, weather forecast and modeling, and sports analytics. It is also a fundamental operator for many important mining operations such as clustering and classification of trajectories. In this paper, we propose a distributed query framework to process trajectory similarity search over a large set of trajectories. We have implemented the proposed framework in Spark, a popular distributed data processing engine, by carefully considering different design choices. Our query framework supports both the Hausdorff distance the Fréchet distance. Extensive experiments have demonstrated the excellent scalability and query efficiency achieved by our design, compared to other methods and design alternatives.

1. INTRODUCTION

Thanks to the explosive adoption and development of mobile and sensing devices, tracking moving objects has already become an easy task. Many applications rely on user's (or an object of interest's) location data to make critical decisions. The growth of IoT (Internet of Things) applications and mobile apps is both explosive and disruptive, and they often collect the locations of moving objects in some fixed period, e.g, an Uber car reports its location every few seconds. As a result, applications from different domains end up collecting massive amounts of location data and locations of the same object over time form a *trajectory*. This leads to massive amount of trajectories that applications must be able to store, process, and analyze efficiently.

For instance, T-Drive [44] contains 790 million trajectories generated by 33,000 taxis in Beijing over only a three-months period, which implies that mobile-based ride-sharing applications like Uber, Lyft, and Didi are likely to generate trajectory datasets orders of magnitude larger than those generated from only 33,000 taxis in just 3 months. The amount of trajectories easily exceeds the storage capacity and the processing capability of a single machine, and thus needs a cluster of (commodity) machines for storage and processing. This naturally brings the important challenge of designing efficient and scalable distributed query processing algorithms and frameworks for large-scale trajectory data.

There are many different, useful query operations over a trajectory data set, e.g., range queries to find all trajectories passing through a spatial or spatio-temporal query range. Among the many different query operations, trajectory similarity search is a fundamental operator that is non-trivial to process. The objective is to find 'similar' trajectories of a query trajectory. This is useful in many mobile and IoT applications; for example, return all taxis that share similar routes to a query trajectory representing a query moving object (another taxis, a user, etc.). Trajectory similarity search is also useful in sports analytics, weather forecast and modeling, and transportation planning and optimization. It is also a building block towards many advanced mining and learning tasks such as clustering and classification.

Many existing studies focused on defining trajectory similarity measures such as *Dynamic Time Warping* (DTW) [41], *Longest Common Sub-Sequence* (LCSS) [37], *Edit Distance on Real Sequence* (EDR) [13], etc. More importantly, to the best of our knowledge, none of the prior studies have investigated how to perform trajectory similarity search in a distributed and parallel setting.

In light of that, we propose a general framework for supporting trajectory similarity search in a distributed environment. Rather than directly indexing all trajectories, we design an *segment-based* distributed index structure. This is coupled with an effective *pruning algorithm* to mark far trajectories only using individual segments. We demonstrate the design and instantiation of the proposed framework using two popular similarity measures for geometric curves and time series data, namely, the discrete segment Hausdorff distance and the discrete segment Fréchet distance [7]. We have also investigate the challenges and design issues associated with realizing our framework in a popular distributed computation engine, Apache Spark.

The real trajectory of a moving object is always a continuous curve in space, but trajectories collected and stored in the database are not, due to the fact that only discrete samples are taken by the sensing devices. For example, a taxi equipped with GPS will report its location every 1 minute. Discrete samples from one moving object form an *ordered sequence of segments*. When the sample rate is high enough, these segments will be able to approximate the real trajectory of a moving object fairly accurately.

That said, we adopt a segment-based representation in this work to represent the trajectory of a moving object. A trajectory T is a sequence of line segments. Formally,

Proceedings of the VLDB Endowment, Vol. 10, No. 11

Definition 1. A ***trajectory*** is a sequence of consecutive line segments, denoted as $T = \langle \ell_1, \ell_2, \cdots, \ell_m \rangle$, where ℓ_i is a line segment in $\mathbb{R}^2$. The end point of ℓ_i is denoted as s_{i+1} and it is the starting point of ℓ_{i+1}.

T represents a trajectory constructed from $(m+1)$ sample points with m line segments, where each sample point s_j is a location of the moving object at the time s_j was sampled, and ℓ_i is the i-th line segment connecting two consecutive sample points s_i and s_{i+1}. The segment-based approach has been shown to be a more accurate approximation to real trajectories, than the classic *point-based approach* in various contexts, including similarity search, outlier detection, clustering, and classification [29, 25, 23, 24]. The later represents a trajectory simply as a sequence of points (using the sampled locations taken for a moving object).

To summarize, the main contributions of our work are:

- We propose the first distributed framework that leverages segment-based partitioning and indexing to answer similarity search queries over large trajectory data. Two established similarity measures, discrete segment Hausdorff and Fréchet distances, are used to demonstrate our framework.
- By realizing our framework and several baselines in Apache Spark, we identify and overcome critical bottlenecks specific to the distributed indexing of trajectories. Our instantiation of the framework includes a carefully designed instance of IndexRDD, the integration of a compressed bitmap into the index's internal nodes, and a dual index to take advantage of two types of indexes with reasonable space overhead.
- We conduct a comprehensive empirical evaluation of the proposed framework using large synthetic and real trajectory data sets to measure its scalability and efficiency. The experimental results clearly demonstrate the superior performance that our framework achieves against possible alternatives.

The rest of the paper is organized as the following. Section 2 formalizes the trajectory similarity search problem and provides two baseline solutions. Section 3 surveys different similarity measures defined in the literature and the query processing methods that they have used. We describe the proposed framework in Section 4 and instantiate it with two similarity measures. Optimizations and detailed design considerations are discussed in Section 5. Section 7 presents the results of our experimental study and Section 8 concludes the paper with remarks on future work.

2. PRELIMINARIES

In this section, we formally define the problem of trajectory similarity search and provide two baseline solutions for a distributed environment. Table 1 lists the frequently used notation in this paper. We focus on the efficiency aspect of processing large-scale trajectory similarity searches. The focus of our work is not to investigate the effectiveness of different similarity search measures for retrieving similar trajectories under different application contexts. Instead, we adopt the well-known Hausdorff and Frechet distances as our similarity measures (over discrete segments in a trajectory) that are widely used for measuring similarity between curves and geometric trajectories.

2.1 Problem formulation

We will only consider two-dimensional points as the locations of a moving object in this paper since it represents the most common application scenarios for trajectories. It is straightforward to extend our solutions to higher fixed dimensions.

Table 1: Frequently used notations.

Notation	Description
$\ell = (s, e)$	A segment from a starting point s and an end point e
$T = \langle \ell_1 \cdots \ell_m \rangle$	A trajectory consist of m segments
$Q = \langle g_1 \cdots g_n \rangle$	Query trajectory consist of n segments
g_i, ℓ_j	i-th (resp. j-th) segment of Q (resp. T)
$\mathcal{T}(\mathcal{T}_Q)$	All trajectories (in partitions intersected by Q)
$\|p - q\|$	L_2 distance between two points p and q
$\vec{d}(p, \ell)$	Distance from point p to segment ℓ: $\min_{q \in \ell} \|p - q\|$
$d(\ell_1, \ell_2)$	Distance between two segments ℓ_1 and ℓ_2: $\max(d(s_1, \ell_2), d(e_1, \ell_2), d(s_2, \ell_1), d(e_2, \ell_1))$
$\text{mindist}(\ell, Q)$	minimum distance from segment ℓ to trajectory Q: $\min_{g_j \in Q} \min_{p \in \ell} d(p, g_j)$
$\text{mindist}(A, Q)$	minimum distance from spatial area A to trajectory Q: $\min_{g_j \in Q} \min_{p \in A} d(p, g_j)$
$D_H(Q, T)$	discrete segment Hausdorff distance between Q and T
$D_F(Q, T)$	discrete segment Fréchet distance between Q and T

Definition 2. Given a query trajectory Q, a set of trajectories $\mathcal{T} = \{T_1, \cdots, T_N\}$, a distance measure D, and an integer k; a ***trajectory similarity search query*** returns the set $S(Q, \mathcal{T}, D, k) \subset \mathcal{T}$ where $|S(Q, \mathcal{T}, D, k)| = k$ and for any $T, T' \in \mathcal{T}$:

$$\text{if} \quad T \in S(Q, \mathcal{T}, D, k), \text{ and } T' \notin S(Q, \mathcal{T}, D, k)$$
$$\text{then} \quad D(Q, T) < D(Q, T').$$

The above definition assumes no ties between distances from two trajectories in $\mathcal{T}$ to a query trajectory. If that is not the case and there is a tie at the k-th position, ranked by trajectories' distances to Q, we break the tie arbitrarily.

Different distance functions can be used to define the distance between two trajectories, $D(Q, T)$, in the above definition. Since we have adopted a *segment-based* representation for trajectories, we define the trajectory distance through the distances of their segments. The distance between two line segments is defined:

Definition 3. Given two line segments $\ell_1 = (s_1, e_1)$ and $\ell_2 = (s_2, e_2)$, we define their distance as

$$d(\ell_1, \ell_2) = \max(\vec{d}(s_1, \ell_2), \vec{d}(e_1, \ell_2), \vec{d}(e_2, \ell_1), \vec{d}(e_2, \ell_1)),$$

where the distance between a point p and a segment ℓ is defined as:

$$\vec{d}(p, \ell) = \min_{q \in \ell} \|p - q\|, \ \| \cdot \| \text{ is the } L_2 \text{ norm.}$$

The above definition for distance between two segments ℓ_1 and ℓ_2 is equivalent to the well-known Hausdorff distance [7] defined between *all points on* ℓ_1 and *all points on* ℓ_2. This is because a line segment ℓ is a convex object, thus the point on ℓ with the maximum distance to another line segment must be one of the two end points of ℓ. Hence, instead of implementing the Hausdorff distance over all points on ℓ_1 and ℓ_2, we can use the simplified expression with their end points. The above distance definition being equivalent to the Hausdorff distance implies that $d(\ell_1, \ell_2)$ is a *metric*, and can thus be used to build an index for segments using any indexing structures that work for a metric space.

There exists a large variety of trajectory similarity measures, depending on how a trajectory is represented (e.g., point-based vs. segment-based) and the objective of the application at hand. Instead of defining yet another similarity measure, we will demonstrate the flexibility of our framework using two classic distance measures between two trajectories (curve objects), namely, the discrete Hausdorff and the discrete Fréchet distances [7] (defined over segments from the two trajectories). They are notably different in that the Hausdorff distance does not account for directional information (so a route from A to B is equivalent to a route from B to

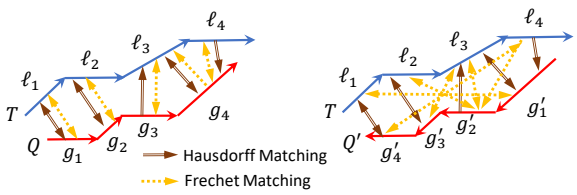


Figure 1: Difference between segment Hausdorff and Fréchet distances: arrows on segments in a trajectory indicate the direction of a trajectory; other dotted arrows indicate a matching from a segment from T (Q) to a segment in Q (T) with respect to the segment Hausdorff (Fréchet) distance.

A if they follow the same paths), whereas the Fréchet distance does incorporate directional information.

Definition 4. For any two trajectories $Q = \langle g_1, g_2, \cdots, g_n \rangle$ and $T = \langle \ell_1, \ell_2, \cdots, \ell_m \rangle$, their ***discrete segment Hausdorff distance*** is defined as:

$$D_H(Q,T) = \max\left\{ \max_{g_i \in Q} \min_{\ell_j \in T} d(g_i, \ell_j), \max_{\ell_j \in T} \min_{g_i \in Q} d(\ell_j, g_i) \right\}.$$

Definition 5. For any two trajectories $Q = \langle g_1, g_2, \cdots, g_n \rangle$ and $T = \langle \ell_1, \ell_2, \cdots, \ell_m \rangle$, a coupling between them is a set L of pairings $\gamma_1, \gamma_2, \ldots, \gamma_h$ where $\gamma_i = (\alpha_i, \beta_i) \in [n] \times [m]$. In particular, $\gamma_1 = (1,1)$ and $\gamma_h = (n,m)$, and given $\gamma_i = (\alpha_i, \beta_i)$, then $\gamma_{i+1} = (\alpha_{i+1}, \beta_{i+1})$ is one of $(\alpha_i + 1, \beta_i)$, $(\alpha_i, \beta_i + 1)$, or $(\alpha_i + 1, \beta_i + 1)$. Then the ***discrete segment Fréchet distance*** between Q and T is defined as:

$$D_F(Q,T) = \min_{L} \max_{\gamma_i \in L} d(g_{\alpha_i}, \ell_{\beta_i}).$$

The discrete segment Fréchet distance is an extension of the well-studied discrete Fréchet distance which measures the distance between points from two curves [7], whereas ours *measures distances between segments*. This is more robust and does not suffer from the sampling rate of the trajectories in the same way that the discrete Fréchet distance does. Both are more selective models of trajectories than the Hausdorff distance in that they enforce that the alignment between trajectories must *restrict to the ordering of the trajectory segments*. Whereas the Hausdorff variant can allow one trajectory to be the reverse of the other, or go back and forth (say if a rider forgot something and had to go back to get it), without increasing their distance. Figure 1 illustrates these two distances and their differences.

Both of these distances *are metrics*, which is important since most applications expect distances between two trajectories to be a metric (which for instance implies, distance from trajectory A to trajectory B should be the same as the distance from trajectory B to trajectory A). This also implies that they can be used towards building effective indexing structures (since they satisfy the triangle inequality). But it is worth mentioning precisely what this means.

In both cases, there is a base metric space $(\mathcal{S}, d)$ where the objects $\mathcal{S}$ are segments, and the distance d is the distance between segments as defined above in Definition 3. Then Hausdorff just defines the trajectory as a set of these objects, mapping identical segments to a single point in $\mathcal{S}$; note that the distance d does not capture the direction of the segment. And the Hausdorff variant is a metric between such sets. Alternatively, the Fréchet variant maintains the ordering between these segment objects, and is a metric between ordered sets, again defined by the furthest distance in the best possible alignment. Hence the Fréchet variant is more discriminative that the Hausdorff one, but on their respective representations, they are both metrics.

2.2 Baseline solutions

As shown in the problem formulation, trajectory similarity search is essentially a top-k query under a customized ordering defined by the query trajectory and the similarity measure. Thus, the classic distributed top-k algorithm [10] is a natural solution for this problem.

Specifically, *distributed top-k* algorithm runs as the following. First, all trajectories in $\mathcal{T}$ are partitioned as whole objects, which necessarily means no trajectory in $\mathcal{T}$ will span two different partitions. Next, in each partition, distances between each $T_i \in \mathcal{T}$ and Q are calculated and a local top-k result is found. Finally, we collect all local top-k results and merge them to get the global top-k result. Even though such baseline solution can solve the problem directly, it involves a full scan through the whole data trajectory set for each incoming query. To make the matter worse, calculating similarity measures between two trajectories is often expensive, which may potentially cause a heavy straggler problem.

Another baseline is to build a distributed R-Tree over the data trajectories. In particular, each trajectory will be treated as an individual object and its location property is identified by the centroid of its minimum bounding box (MBR); much like a Hilbert R-Tree [22]. Such index structures can help us avoid scanning the whole trajectory data for each incoming query. For each query, we can adopt a similar technique as that for kNN queries over points. First, we need to find a threshold such that at least k data trajectories are covered. Then, an index based pruning is conducted, where we calculate the minimum distance between the MBR of a data trajectory to that of Q and see if it falls within the threshold. Finally, we invoke the first baseline solution over all potential candidates to get the final result. In this baseline solution, the distributed index does help us prune some useless trajectories, yet its pruning power is limited.

The *final baseline* is to build a distributed index over all data trajectories in its metric space. Note that the two distance measures defined in Section 2.1 are metric. On a single machine, index structures like vantage point trees (VP-Tree) [19, 36, 42] and M-Trees [14, 8] can be used. However, these have never been studied in a distributed environment.

We can extend this approach to a distributed setting as follows. We first sample a group of trajectories uniformly at random as *pivots* and partition the data by assigning each trajectory to its closest pivot according to the distance measure. Next, we build a local metric tree (VP-Tree or M-Tree) for each partition. During the local indexing procedure, for each partition $\mathcal{T}_i$, we collect its pivot V_i, the cover radius $r_i = \max_{T \in \mathcal{T}_i} D(V_i, T)$, and the partition size at the master node. If we have a pruning bound ε from the query trajectory Q covering at least k data trajectories, we can prune a whole partition $\mathcal{T}_i$ if $D(V_i, Q) - r_i > \varepsilon$.

A trajectory similarity query $S(Q, \mathcal{T}, D, k)$ will be processed as follows. First, we find the nearest pivots to Q such that their represented partitions are sufficient to cover k data trajectories. Next, we find the k nearest neighbors to Q in these partitions to get a pruning bound ε. Then, we use the condition $(D(V_i, Q) - r_i > \varepsilon)$ to build a filter and prune all useless partitions and the partitions we have already checked. Finally, we launch the distributed top-k algorithm over the remaining candidate partitions and merge the results to get the final answer.

According to our experiments, this solution does provide good pruning power, and has the least total number of distance calculations among all baselines. However, it is not scalable, as it takes a very long time to build its index, and the method cannot fully utilize the cluster resources during its querying process. More details are provided in Section 7.

3. RELATED WORKS

As mentioned, previous work on indexing trajectories and performing trajectory similarity search have only considered the case when data sets fit on a single machine. They focused on studying different distance measures such as *Dynamic Time Warping* (DTW) [41, 43], *Longest Common Sub-Sequence* (LCSS) [37], *Edit Distance on Real Sequence* (EDR) [13], *Edit distance with Real Penalty* (ERP) [12], *Edit Distance with Projections* [29], and DISSIM (a dissimilarity measure) [18]. All of the aformentioned distance measures do not satisfy the triangle inequality, and thus are not metrics. As a result, most of the design decisions are in designing new indexing and pruning strategies specific to those measures. Moreover, most operate with the base metric defined on points, and thus have slightly less modeling power.

In contrast, our base metric is defined using *all points* from the two segments. And instead of proposing a new distance metric, we have adopted the classic Hausdorff and Fréchet distances that are widely used in computational geometry to measure distances between geometric objects, especially for curves and trajectories [7, 5, 3, 35, 6, 21, 46, 4, 38]. Both distance measures are *metrics*, hence, are more intuitive in modeling trajectory distances. Furthermore, all previous works are conducted on a single machine environment, and the proposed index structures cannot be adopted trivially in a distributed system.

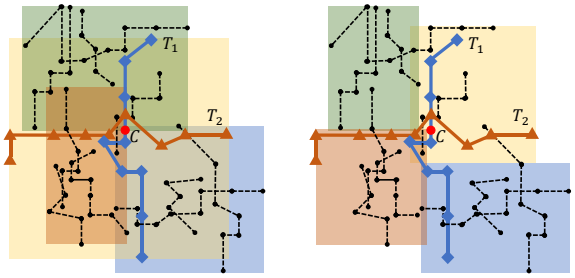
Another line of work is in efficiently designing fast algorithms for calculating distances between a pair of trajectories, for instance under the Hausdorff [21, 6], Fréchet [16], or discrete Fréchet distances [1]. These improvements are complementary to our work, they can improve time to compute each distance which is useful in the last stage of query processing, computing distances between each candidate trajectory and the query trajectory, but do not influence the partitioning or pruning strategies.

Trajectory similarity search is a useful building block towards many other important and useful mining operations, such as clustering and classification. To that end, trajectory clustering and trajectory classification have been extensively studied; see [24, 25] and references therein. Another related problem is trajectory outlier detection [23]. Calibrating trajectories and simplification of trajectories are useful as a data cleaning step towards more effective similarity search results [31, 32, 28, 27]. Indexing and summarizing trajectories have also been well studied [29, 47, 33]. A number of efforts were devoted to building a general trajectory store/system for storing and analyzing large trajectories [34, 39, 15]. These stores do leverage a distributed cluster to store and process trajectories, but they do not support similarity search over trajectories. A complete survey of related work in trajectory queries and mining is beyond the scope of this paper, and we refer interested reader to [47] for a more comprehensive review.

4. FRAMEWORK

Next, we describe the distributed processing framework designed for executing trajectory similarity search. The framework is based on a distributed index structure and a suite of pruning techniques leveraging the distributed index. We will investigate how we partition and build a distributed index structure over a large trajectory dataset, and examine how the distributed query processing procedure utilizes the index under discrete segment Hausdorff distance and discrete segment Fréchet distance.

Notably, a distributed index can have significantly different high-level structure than a non-distributed one. The goal is to minimize computation and communication on a small number of data shuffles. There are two main bottlenecks. First, a top-level grouping into partitions so on a query we can quickly avoid examining most



(a) Partition by trajectory. (b) Partition by segment.

Figure 2: Two very different trajectories with the same centroid, for instance two paths across a city center, one is north-south oriented (T_1) and the other is east-west oriented (T_2). This demonstrates the difficulty of using MBRs and their centroids for an entire trajectory for partitioning and indexing.

of the partitions entirely. Second, an efficient and effective pruning strategy so we only need to brute force compute distances for a small candidate set of trajectories. We still can use traditional indexes (like R-trees) within each examined partition to help with the pruning goal, but unlike the VP-Tree baseline, we are less concerned about creating a single big hierarchy. Instead those two main levels of filtering are most important. This means we can "touch" information about each trajectory within a partition as long as it is done efficiently to very effectively prune these from the next level. These design observations greatly influence the strategy of our proposed approach based on segment indexing.

4.1 Distributed indexing of segments

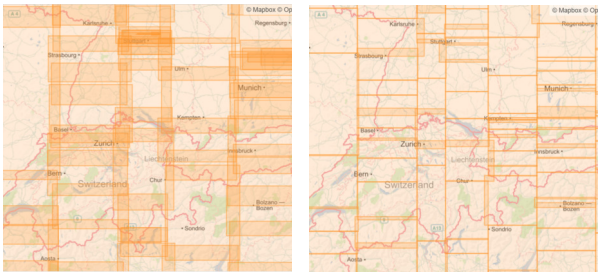
The key to building an effective distributed index is a carefully-designed partitioning strategy to partition the data into blocks, so on a query, most of the blocks can be quickly pruned. Moreover, we need to make sure each partition is roughly of the same size to keep the load balanced. Finally, we need to watch the memory footprint to make sure the index structure will not blow up the heap memory of any executor in the distributed system.

Trajectories, however, are from a complex data type which vary greatly in shape, geometric and spatial span, sample rate, speed, and number of segments. The multi-variant nature makes designing a good partitioning strategy a non-trivial task. For instance, building a MBR (minimum bounding rectangle) for each trajectory, summarizing each one with the trajectory centroid, and building an R-tree like structure on these MBRs is efficient and easy to achieve load balance, and has low memory footprint. But as can be seen in Figure 2 where two very different trajectories have the same centroid, it leads to very limited pruning power.

In contrast, our approach is to partition and index the individual segments. We create an MBR of each segment, represent each segment with its centroid (its midpoint), and then build R-trees on these MBRs using these centroids. This leads to much more efficient and effective partitioning of the data, as can be seen in Figure 3. However, it requires more care in keeping track of the trajectory associated with each segment, and in pruning an entire trajectory

To index a segment ℓ from a trajectory T, in addition to storing its start and end points s and e, we assign a tuple (tid, sid) where tid is the trajectory id of T, and sid indicates ℓ's position within trajectory T. For instance, a segment ℓ assigned with the tuple $(3, 6)$ indicates that ℓ is the 6th segment in trajectory T_3. This tuple also helps reconstruct a trajectory from segments.

All segments are of the same object size, which consists of the coordinates of its two end points and a (tid, sid) tuple. The load balancing problem, during partitioning, is thus naturally reduced to



(a) Whole trajectories. (b) Trajectory segments.

Figure 3: MBRs of whole trajectories and MBRs of their segments, and the resulting partitions from a subset of the same OpenStreetMap GPS trace dataset. The segment-based partitions clearly have far less overlapping regions.

balance the number of segment objects in each partition, rather than worrying about the sum of the object size. Moreover, since in most applications, moving objects have a speed limit, and a relatively high sampling rate to keep track of its locations, segments in a trajectory usually come with a small spatial span, where the spatial span of a geometric object refers to the spatial area covered by the object's MBR. Partitioning by segments dramatically reduces overlapping between MBRs of different partitions. Most importantly, it is often sufficient to prune a trajectory by simply checking the individual relationships between its segments and a query trajectory's segments. A pruning theorem for discrete segment Hausdorff and discrete segment Fréchet distance will be introduced in Section 4.2.

Nonetheless, the segment-based partitioning and indexing approach does introduce overheads. We often need to reconstruct the data trajectory, if it ends up being a candidate, before calculating its similarity measure to the query trajectory. If the segments of a trajectory T are separated and stored into different partitions on different nodes, reconstructing T would involve a data shuffle with non-trivial communication cost. We also need an efficient solution to link pruned segments back to trajectories in order to prune trajectories, which is surprisingly a tricky task. Fortunately, these overheads can be addressed with careful design choices, which we will demonstrate in details in Section 5.

The indexing procedure in our framework consist of three phases: *partitioning*, *local indexing* and *global indexing*. In this section, these stages are described in principle. We will demonstrate the instantiation of a two-level indexing strategy on Apache Spark with more details in Section 5.2.

Partitioning. In this phase, we extract segments from the input trajectories and partition them according to their spatial locality. Any partitioning strategy that offers strong spatial locality and good load balancing can be adopted. We choose to adopt the STR (Sort-Tile-Recursive) partitioning strategy due to its simplicity and proven effectiveness by existing studies [26]. Specifically, our STR partitioning strategy first takes a set of uniform random samples from the input segment collection and then runs the first iteration of Sort-Tile-Recursive algorithm over the *centroids* of sampled segments to determine partition boundaries. Then, each segment ℓ is assigned to the partition whose spatial region contains the centroid of ℓ.

Local Indexing. Within each partition, we build an R-Tree [20] like data structure over the segments as its local index. Figure 4 shows an example of our local index structure. The main difference between our local index structure and classic R-Tree is that each internal node u in the tree contains the complete set of trajectory IDs (denote as the TID set of u) for all segments contained by the subtree rooted at u. The TID sets identify all trajectories that have passed through the spatial region of any node (MBR of the node) in

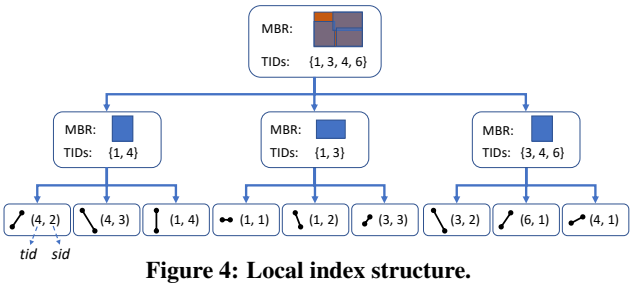


Figure 4: Local index structure.

Algorithm 1: Similarity Search $S(Q, \mathcal{T}, D, k)$

1: Find the closest partitions $P_1, P_2, \cdots, P_m$ to Q that cover at least $c \cdot k$ trajectories.
2: Sample $c \cdot k$ trajectories $T'_1, T'_2, \cdots T'_{c \cdot k}$ uniformly at random without replacement from $P_1, P_2, \cdots, P_m$.
3: Calculate the distances from $T'_1, T'_2, \cdots T'_{c \cdot k}$ to Q; take the k-th smallest result as the pruning bound ε.
4: Use the global index to identify all partitions A whose $\text{mindist}(Q, A) = \min_{g_i \in Q} \min_{p \in A} \vec{d}(p, g_i) > \varepsilon$; union all their TIDs as F_1.
5: For all partitions A whose $\text{mindist}(Q, A) \leq \varepsilon$, find all trajectories which contain a segment ℓ_i such that $\text{mindist}(\ell_i, Q) > \varepsilon$; collect and union TIDs of all such trajectories at the master node as F_2.
6: Find all the segments whose TID is **not in** $F_1 \cup F_2$, reconstruct the whole trajectories for these segments.
7: Calculate the distance from Q to each such trajectory, launch a distributed top-k algorithm to identify the k trajectories $R_1, R_2, \cdots, R_k$ with smallest distances.
8: **return** $R_1, R_2, \cdots, R_k$

the local index. Note that we typically have many more *segments* in a local index subtree than the number of *trajectories* passing through that subtree. Hence, keeping TID sets enables us to prune all trajectories that pass through a node in a local index without traversing its children.

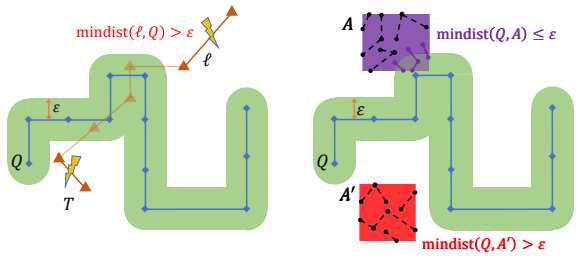
Global Indexing. Lastly, the master node will collect statistics from each partition to build a global index. Specifically, we will collect the partition boundaries and the TID sets of the root nodes from all local indexes. The global index enables us to prune trajectories passing through the spatial region of a specific partition without invoking any task to look into that partition.

4.2 Search procedure

The search procedure consists of three steps: pruning bound selection, index-based pruning, and finalizing the results. The search procedure is outlined in Algorithm 1. We will first discuss how an entire trajectory can be pruned by just examining individual segments given a distance threshold ε for the trajectory distance. Then we will describe how we can generate a good ε, and finalize the search results.

Index-based pruning. The distributed index is used to prune far-away trajectories given a query trajectory Q and a distance threshold ε. Any trajectory T such that $D(T, Q) > \varepsilon$ can be pruned away efficiently. Next, we show how to do this using the distributed segment index with only information on segments, for both discrete segment Hausdorff and discrete segment Fréchet distance.

THEOREM 1. *Given a distance threshold $\varepsilon > 0$ and two trajectories Q and T. If there exist a segment $\ell_i \in T$ such*



(a) Pruning by segment. (b) Pruning by MBR.

Figure 5: Pruning Trajectory T with Bound ε.

that $\text{mindist}(\ell_i, Q) = \min_{g_j \in Q} \min_{p \in \ell_i} \vec{d}(p, g_j) > \varepsilon$, *then we have* $D_H(Q,T) > \varepsilon$ *and* $D_F(Q,T) > \varepsilon$.

PROOF. Discrete segment Hausdorff distance is:

$$D_H(Q,T) = \max\left\{\max_{g_j \in Q}\min_{\ell_i \in T} d(g_j, \ell_i), \max_{\ell_i \in T}\min_{g_j \in Q} d(\ell_i, g_j)\right\}$$
$$\geq \min_{g_j \in Q} d(\ell_i, g_j) \geq \min_{g_j \in Q}\min_{p \in \ell_i} \vec{d}(p, g_j) > \varepsilon.$$

As for discrete segment Fréchet distance, each segment $\ell_i \in T$ will be matched with at least one segment $g_j \in Q$. The discrete distance is then defined by the matching with the smallest maximum gap between all pairs of segments. If there exists $\ell_i \in T$ such that $\text{mindist}(\ell_i, Q) > \varepsilon$, the maximum gap in any possible matching between Q and T must be greater than $\text{mindist}(\ell_i, Q) > \varepsilon$, which further implies $D_F(Q,T) > \varepsilon$. □

Thus to apply the theorem above, we only need to check individual segments of a data trajectory against individual segments of a query trajectory. For instance, as shown in Figure 5(a), given a pruning bound ε and the query trajectory Q, trajectory T can be safely pruned as there are two segments of T whose distances to Q are greater than ε. Any of these two segments of T can individually serve as a witness to prune T in its entirety. Note that we have introduced the concept of *minimum distance* between a segment ℓ and a trajectory Q, denoted as $\text{mindist}(\ell, Q)$. It represents the minimum possible distance from ℓ to any segment of Q.

In our framework, we first invoke a range query in the global index to find all partitions whose minimum distance to Q is greater than ε. According to Theorem 1, we can prune all trajectories passing through these partitions. Note that the minimum distance between a trajectory Q and a partition (which is identified by an MBR A) is defined as:

$$\text{mindist}(Q, A) = \min_{g_i \in Q}\min_{p \in A} \vec{d}(p, g_i).$$

We union the TID sets of all such partitions returned by the global index; which can efficiently be represented with the compression techniques as we will detail in Section 5.3. This provides a collection of trajectories that can be safely pruned. Next, we invoke the same range query now on each local index of the *remaining partitions*. The TID set associated with each node in a local index allows us to avoid traversing down the subtree of a node u when u's MBR is already out of the search range (i.e., $\text{mindist}(Q, A(u)) > \varepsilon$). This enables each local index to return another set of TIDs, representing those trajectories passing through that partition that can be safely pruned.

Next, the framework unions these TID sets at the master node, including the earlier TID set returned by the global index, to derive the final set of trajectory ids that can be safely pruned.

Pruning bound selection. Selecting a good distance bound ε to be used for the above pruning strategy is critical. We will show how to select a relatively tight pruning bound ε to help us prune away most irrelevant trajectories, given a query trajectory Q. To safely prune a data trajectory, the threshold ε must satisfy that at least k data trajectories from $\mathcal{T}$ have distances at most ε from Q.

A key observation is that similar trajectories will pass through similar spatial regions. Hence we use the global index to identify all partitions which Q intersects; let $\mathcal{T}_Q$ be the set of trajectories contained in these partitions. Then from these partitions we generate $c \cdot k$ uniform random samples from the union of their trajectory ids (i.e., $\mathcal{T}_Q$, the union of their TID sets), for some parameter c. We set ε as the k-th smallest distance among those $c \cdot k$ sampled trajectories. If there are fewer than $c \cdot k$ trajectories of interest in these partitions, we invoke a nearest neighbor query on the global index to find enough partitions to cover $c \cdot k$ different trajectories.

The choice of c represents how effectively these partitions can prune trajectories. If c is small (say $c = 3$), and very few additional partitions are required (i.e., partitions with MBR A such that $\text{mindist}(Q, A) > \varepsilon$), then the search is efficient and the pruning power is high. If c is large (say $c = 100$), or many additional partitions must be considered, then the search becomes slow.

To help understand the choice of c, we observe that given $c \cdot k$ samples, the k-th closest distance estimates the $(1/c)$-th quantile of distances among the set $\{D(Q,T) \mid T \in \mathcal{T}_Q\}$, i.e. from Q to trajectories in partitions Q intersects. If this value ε is smaller than the minimum distance to any point in a partition, then we can prune all segments in that partition, hence, pruning away all trajectories passing through that partition. It is reasonable that among partitions which Q does not intersect (i.e., the trajectories, $\mathcal{T} \setminus \mathcal{T}_Q$), most are further than (for instance) the 0.2- or 0.1-quantile of those distances in partitions Q does intersect, making $c = 5$ or $c = 10$ a good choice. Indeed we observe that $c = 5$ is a good enough in Section 7.5. However, what remains is to understand how accurate an estimate of the $(1/c)$-th quantile is; we summarize in the following bound.

LEMMA 1. *Let* $\mathcal{T}'$ *be* $c \cdot k$ *trajectories randomly sampled from* $\mathcal{T}_Q$. *Let* ε *be the* k*-th smallest distance from* $\{D(Q,T) \mid T \in \mathcal{T}'\}$. *We can guarantee with probability at least* $9/10$, *that* ε *is a* γ*-quantile of* $\{D(Q,T) \mid T \in \mathcal{T}_Q\}$, *where*

$$\gamma \leq \frac{1}{c} + \sqrt{\frac{3}{2ck}}.$$

PROOF. The $(1/c)$-quantile is the distance ε^* where with probability $(1/c)$ a distance sampled from $\{D(Q,T) \mid T \in \mathcal{T}_Q\}$ is smaller than ε^*. To show that our pruning bound γ is useful, we need to bound the fraction of these distances smaller than ε^*. We can map each trajectory $T \in \mathcal{T}'$ to a random variable which is 1 if the corresponding distance is $D(T,Q) < \varepsilon^*$ or 0 otherwise. The average of these is well-concentrated around $1/c$, and the fraction more than $1/c$ can be bound with a standard Chernoff-Hoeffding bound as $\sqrt{\frac{1}{2ck}\ln(2/\delta)}$ with probability $1 - \delta$. Setting $\delta = 1/10$ proves our claim. □

For instance, this implies that if $c = 10$ and $k = 20$, then with 0.9 probability, the error $\sqrt{\frac{3}{2ck}} = \sqrt{\frac{3}{2 \cdot 10 \cdot 20}} \leq 0.087$, and thus ε is at most a $\frac{1}{10} + 0.087 = 0.187$-quantile of the distances induced from $\mathcal{T}_Q$.

Finalizing results. Here we reconstruct all trajectories that survive the pruning (i.e., those trajectories whose id is not in the pruning set of trajectory ids constructed above), evenly distribute them to all CPU cores, and calculate their exact similarity measures to Q. Then, we invoke a distributed top-k algorithm to get the final results.

Correctness of the search algorithm. Recall that we will only prune a trajectory T if there exists a segment $\ell_i \in T$ such that $\text{mindist}(\ell_i, Q) > \varepsilon$. By Theorem 1, the similarity measure from such trajectories to Q must be greater than ε. In addition, note that there are at least k trajectories (the k samples retrieved for generating the pruning bound) whose similarity measure is within ε. This implies we will include all top-k results in the comparison in the finalizing stage. Hence, the search algorithm is correct.

5. INSTANTIATION OF THE FRAMEWORK

It is possible to instantiate our framework in different distributed systems. Instantiation on different systems introduce slightly different challenges, but the underpinning principles are similar. Thus, we choose to demonstrate such principles using one of the most popular distributed computation engines, Apache Spark [45], for this purpose. In particular, we discuss the techniques we have developed to achieve distributed indexing in Spark, to support compact representation of the trajectory IDs on each node (the TID set) in global and local indexes, and to build an auxiliary structure to avoid reconstructing a full trajectory in finalizing the query results.

5.1 Apache Spark overview

Apache Spark[45] is a general-purpose distributed computing engine for in-memory big data analytics. It provides a data abstraction called Resilient Distributed Dataset (RDD), which is a distributed collection of objects partitioned across a cluster. User can manipulate RDDs through functional programming APIs (e.g. `map`, `filter`, `reduce`). RDDs are fault-tolerant since Spark can recover lost data using linage graphs by rerunning operations to rebuild missing partitions. RDDs can also be cached in memory or made persist on disk explicitly to support data reusing and iteration. Moreover, RDDs are evaluated *lazily*: each RDD actually represents a "logical plan" to compute a dataset, which consists of one or more "transformations" on the original input RDD, rather than the physical, materialized data itself. Spark will wait until certain output operations (known as "actions"), such as `collect`, to launch a computation. This allows the engine to execute pipelining operations, as a result Spark never needs to materialize intermediate results.

5.2 Building an index over RDDs

The RDD abstraction is an abstraction originally designed for sequential scan, thus random access through this abstraction is expensive as it may simply fall back to a full scan over the data collection. An extra complexity is that we do not want to alter the Spark core or the RDD abstraction, in order to support easy migration to future Spark releases. To overcome these challenges, we adopt the `IndexRDD` abstraction introduced in our previous work Simba [40] to fit the two-levelindexing strategy.

IndexRDD. To build an index over an RDD, we pack all objects within an RDD partition into an array, which gives each record a unique subscript as its index. This structure makes random access inside a RDD partition an efficient operation with $O(1)$ cost. To achieve this, we define the `IPartition` data structure as below:

```
case class IPartition[T](Data: Array[T], I: Index)
```

`Index` is an abstract class that represents the local index for data in this partition, and can be instantiated with our local index structure presented in Section 4.1. `IndexRDD` is simply defined as an RDD of `IPartition`:

```
type IndexRDD[T] = RDD[IPartition[T]]
```

Objects in an RDD are partitioned by a *partitioner*, and then packed into a set of `IPartition` objects, which contains a *local index* over records in that partition. Furthermore, each `IPartition` object

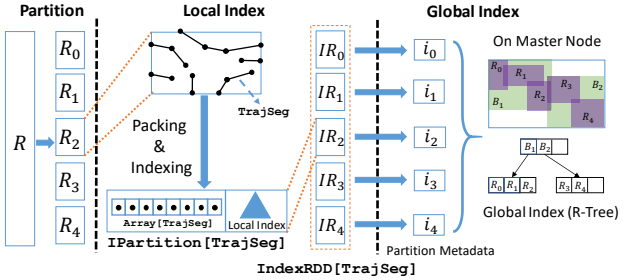


Figure 6: Distributed segment index structure in Spark.

emits its meta information, including the boundary of the partition and the TID set of the root node from the local index of this partition, to construct the *global index* at the master node. By the construction of `IndexRDD`, RDD elements and local indexes are *naturally fault tolerant because of the RDD abstraction of Spark*. The global index is kept in the heap memory of the driver program on the master node with no fault tolerance guarantee. Nevertheless, global indexes can be lazily reconstructed by the statistics collected from persisted RDD when required.

More specifically, to instantiate our framework, elements in an RDD are objects defined as below:

```
case class TrajSeg(seg: LineSegment, meta: TrajMeta)
```

In this definition, `seg` is a line segment and `meta` contains its trajectory ID and segment ID (representing the tuple (tid, sid) as discussed in Section 4.1). The index structure and its construction is summarized in Figure 6. We first partition all trajectory segments using the STR partitioning strategy. Spark allows users to define their own partitioning strategies through an abstract interface called `Partitioner`. In our framework, we instantiate the STR partitioning strategy in `STRPartitioner` to generate partition boundaries and specify how segments (of all trajectories) map to a partition. Then, local indexes are built inside `IPartition` objects following the design described in Section 4.1, in parallel on multiple, different nodes (cores). Finally, the driver program collects statistics from each partition, including the partition's MBR and the TID set of the root node of its local index, to build a global index at the master node.

5.3 Compressed bitmap

As a key component of our index design, a TID set is attached to each internal node in the tree. Recall that the number of segments in a branch of a local index can be far more than the number of trajectories that pass through the same region represented by the MBR of that branch. As a result, storing a TID set significantly boosts the performance of our pruning algorithm as it can retrieve the IDs of all trajectories which pass through an internal tree node to be pruned without traversing the entire subtree. However, this will also increase the memory footprint of the tree significantly, which hurts the scalability of our framework in an in-memory instantiation like on Spark. As we move up in the tree, the union of the TID sets from lower levels lead to a large set of ids to remember, thus, saving all trajectory IDs in an uncompressed data structure causes a substantial impact on memory consumption.

One solution is to use a Bloom filter to encode each TID set on an internal node. A Bloom filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It hashes each item with h independent hash functions to a single bitmap with m bits. However, it allows false positives and other drawbacks and limitations we expound upon next.

First, note that there are two directions to apply the pruning bound introduced in Section 4.2. On one hand, a trajectory T could

qualify as a candidate of the top-k set if there exists at least one segment $\ell_i \in T$ such that $d(\ell_i, Q) \leq \varepsilon$. On the other hand, T can be safely pruned if there exists at least one segment $\ell_i \in T$ such that $d(\ell_i, Q) > \varepsilon$. The latter offers better pruning power since it gets rid of a lot of false positives. However, if we use a Bloom filter $b(u)$ to encode the TID set TID(u) of a node u in the index, u is pruned when $\text{mindist}(u, Q) > \varepsilon$, but $b(u)$ may return false positives when we test if a particular tid belongs to TID(u).

Recall that in the finalizing step, we need to find all candidate trajectory ids that are *not* in the set of TIDs being pruned away during the search procedure. The fact that $b(u)$ may return false positives means that we cannot find all such candidate trajectory ids, i.e., a trajectory close to the query trajectory Q can be missed.

Second, as trajectory ids are integer values from a fixed domain, say from $[N]$ (there are N trajectories in $\mathcal{T}$). Storing the trajectory ids in a TID set may often end up using more space than using just a length N bitmap (where 1 at the i-th position indicates that T_i is part of this TID set, and 0 indicates otherwise). Using a bitmap structure will not introduce any false positives, thus we can fully leverage the power of our pruning bound. However, for a large value of N, storing a size-N bitmap at each node of every local index is still prohibitively expensive.

To solve this problem, we adopt a roaring bitmap [11], which is a concise, compressed bitmap. In other words, we represent each TID set as a size-N bitmap as explained above, but compress it using a roaring bitmap. It builds a hybrid data structure combining three container types (arrays, bitmaps and runs) into a two-level tree. Elements in roaring bitmaps are separated into chunks by their most significant bits, and then each chunk is organized as an uncompressed bitmap for *dense* chunks or an array container for *sparse* chunks. Roaring bitmap has been tested against competitive implementations of other popular formats (such as Concise, WAH, EWAH), and is typically two orders of magnitude faster. In addition, roaring bitmap often offers significantly better compression ratio than other approaches. In our case, the size-N bitmaps for most nodes in local indexes are sparse due to the strong spatial locality achieved within each partition by our partitioning strategy. Hence, roaring bitmaps can encode and replace these sparse size-N bitmaps very effectively with high compression ratio, leading to significantly smaller memory footprint. Finally, roaring bitmap supports set operations like union, which is needed when building the bitmap of a parent node from the bitmaps of its children nodes.

5.4 Dual indexing

A major overhead in our framework is that we need to regroup all segments of a candidate trajectory before its similarity distance to the query trajectory Q can be calculated. Since our framework relies on segment-based partitioning and indexing, segments of the same trajectory may end up in different partitions on different nodes in a cluster. The step introduces a data shuffling stage that may involve notable communication overhead. If we were to partition trajectories using the trajectory-based approach, all segments of a trajectory are guaranteed to locate within a single partition. But as illustrated earlier, trajectory-based partitioning and indexing is ineffective in terms of pruning.

Given these observations, we design an auxiliary structure in addition to the segment-based partitioning and indexing, in an approach called dual indexing. It combines the pruning power of segment-based indexing and the benefit of avoiding the potentially expensive retrieval of all segments to reconstruct a trajectory.

In particular, we will keep two copies of the indexed dataset. In the first copy, data trajectories are partitioned by their segments, and indexed by the distributed index structure as described in Section 4. This copy is used for pruning. In the second copy, each trajectory is viewed as a single object during partitioning, and they are partitioned by the centroid of each trajectory's MBR. However, we do *not* build any local indexes over the partitions from the second copy. The master node still collects the partition boundaries to build a global index for the second copy.

Using the second copy and its global index, once the search from the first copy has returned a set of candidate trajectory ids (after removing any trajectory ids from its pruning process), denoted as $C(Q)$, we can first find all partitions that may contain any candidate trajectories by using the pruning bound established in Theorem 1. More specifically, we can calculate the distance $\text{mindist}(g_i, A_j)$ between each segment g_i from the query trajectory Q and the MBR A_j for the j-th partition on the second copy, and use these distances to perform pruning. Note that A_j's are available from the global index of the second copy. $C(Q)$ must be from the remaining partitions after pruning using the global index of the second copy, and we find the entire trajectory for each trajectory id from $C(Q)$ by probing into those partitions in parallel.

More importantly, with dual indexing, the local indexes in the first copy *does not* require to store an array of `TrajSeg` any more as included in Figure 6, since we no longer need the (tid, sid) pairs to reconstruct a trajectory from its segments. Hence, dual indexing is to *separate* the `IndexRDD` as presented in Section 5.2 *into two RDDs* with different partitioning strategies and local structures.

Furthermore, for trajectories in the second copy, since we *do not need* to maintain any local indexes on any partition, each partition is simply a hash table where the key is a trajectory id tid and the value is a trajectory T_{tid}. *All our framework needs* from the second copy is to retrieve candidate trajectories in the final step from different partitions using their trajectory ids. Thus, we can *compress* each trajectory in a partition in the second copy to reduce the memory footprint. This adds a small overhead during index construction, and while decompressing a trajectory during query processing. But for query processing, we only need to decompress those candidate trajectories, which is a very small set and is done in parallel.

As a result, dual indexing actually *does not* lead to notable memory storage overhead compared to using only one copy with the original `IndexRDD` structure. The overall memory footprint of dual indexing is *much less* than two factors of using just the distributed segment based indexing alone.

6. EXTENSION TO OTHER METRICS

Our framework will work for other metrics based on $(\mathcal{S}, d)$, such as average of aligned distances [41, 43, 9], averages of all pairs of distances [30], bipartite matchings [2], or partial matchings [17]. Also our framework structurally will work for different base metrics, such as using $(\mathbb{R}^2, L_2)$, the standard Euclidean distance defined over the segment end points s_i. But this model captures less of the connected nature of these trajectories.

As an example, we can use similar pruning techniques, as illustrated in Section 4.2, for Dynamic Time Warping (DTW) with a slightly modified pruning bound. Common forms of DTW matches pair of segment end points (with base metric $(\mathbb{R}^2, L_2)$) and sums these distances along the trajectory. Thus, if there is an endpoint s_i in trajectory T such that $\text{mindist}(s_i, Q) > \varepsilon$, we can still safely prune T since the cost of the best potential match for one end point s_i in T has already exceeded the pruning threshold and all other matchings, which only add more to the cost. Similarly, for *average* aligned distance of segments, if there is a segment $\ell_i \in T$ such that $\text{mindist}(\ell_i, Q) > \varepsilon$, we will have $D(Q, T) > \frac{\varepsilon}{|Q|+|T|}$.

Even though the pruning bounds above are looser than Theorem 1 for discrete Hausdorff and Fréchet distances, we will still quickly

prune most trajectories which are far from Q. Furthermore, if we have more assumptions on spatial spans or segment counts for data trajectories (e.g., aggregating effects from many segments in an index subtree), a tighter pruning bound can be derived.

Metrics based on a max distance (e.g., Hausdorff or Fréchet) work more naturally, and probably more effectively, in our framework, yet the pruning bounds do directly extend to average- or sum-based measures (e.g., DTW). However, as these bounds may not prune as effectively, the best way to handle such measures may be to base a filter on the several segments located within a single partition, as opposed to using just one — or other solutions more tailored to individual similarity measures. We leave these details for future work.

7. EXPERIMENTS

7.1 Setup

Cluster setup. All experiments were conducted on a cluster with 1 master node and 16 client nodes. The master node has two 4-core Intel Xeon E5-2609 @ 2.40GHz processors and 20GB main memory reserved for Spark's driver program. Ten client nodes have a 4-core Intel Core i7-3820 @ 3.60GHz processor, and the other six nodes have a 6-core Intel Xeon CPU E5-1650 v3 @ 3.5GHz processor. All 16 processors on client nodes are configured with hyper-threading. Each node in the cluster is connected to a Gigabit Ethernet switch runs Ubuntu 14.04.4 with Hadoop 2.6.4 and Spark 2.1.0. We reserved 32GB DRAM on each client node (in total 512GB) for running our Spark jobs.

Datasets. We have used the following real-world and synthetic datasets in our experiments.

OSM-FULL: It contains all publicly available GPS traces (of various objects) uploaded in the first 7.5 years of the OpenStreeMap project. We filtered out data trajectories that are clearly outliers, including those extremely long trajectories (bouncing across the world) or too short (stuck at a single point). OSM-FULL contains about 1.4 million data trajectories with a total of roughly 1.06 billion sample points (hence, roughly 1.06 billion segments), and is 50.7 GB.

OSM-DE: This dataset is a subset of OSM-FULL in Germany. It is of the highest data density among all regions in OSM-FULL. OSM-DE contains about 370k data trajectories with roughly 360 million sample points (hence, roughly 360 million segments), and is 16.7 GB.

GEN-TRAJ: To test the scalability of various methods, we also generate a large set of trajectories with the following procedure: we took the entire road network of the United States, randomly generate a large amount of shortest path queries, and take the returned shortest paths as the data trajectories. To better simulate real trajectories on a road network, we randomly generate shortest path queries such that the distance of a returned shortest path follows a Gaussian distribution, whose mean and standard deviation are both set to 40km. The largest synthetic data set, GEN-TRAJ, has 10 million trajectories (roughly 2.35 billion segments) and is 102.9GB. We take a random sample of various size from this data set, ranging from 1 million to 10 million data trajectories, to carry out the scalability test.

Methods tested. We included the following base line solutions as outlined in Section 2.2:

Brute Force: The general distributed top-k algorithm which will calculate the distance between the query trajectory and each data trajectory in distributed and parallel fashion.

Traj Index: A distributed R-Tree is built according to the centroid of each data trajectory's minimum bounding rectangle (MBR). The search procedure is similar to our proposed solution. Specifically, we sample $c \cdot k$ trajectories from all partitions the query trajectory has passed through, to elect a pruning bound ε. Then, we will prune a trajectory T (whose minimum bounding box is A) if $d(Q, A) > \varepsilon$, using the distributed R-tree. Finally, we calculate the exact similarity measures between Q and all remaining trajectories to get the final top-k results.

VP-Tree: A distributed vantage point tree (VP-tree) [19, 36, 42] is built and its search procedure involves two groups of local kNN queries over a limited number of partitions; see our discussion in Section 2.2.

M-Tree: A distributed M-Tree [14, 8] is built while its search procedure follows similar strategy to that of VP-Tree.

Centralized: We also implemented the centralized version of Traj Index, VP-Tree, and M-Tree running on a single machine. The index construction time of these centralized approaches are much more expensive than their distributed versions, as well as their query latencies, due to the lack of parallelism. Furthermore, these solutions are non-scalable when data size starts to grow. Hence, we have omitted Centralized when reporting our experimental results.

DFT, our method: For our proposed framework, we implemented several different variants. They differ in the data structure (either a bloom filter or a roaring bitmap as discussed in Section 5.3) used for representing the TID set associated with an internal node in a local index, and the choice of whether to use dual indexing or not (Section 5.4). We refer to our method as the DFT (Distributed Framework for Trajectory similarity search). The variants tested include: DFT–BF+DI, DFT–RB w/o DI, and DFT–RB+DI, which stands for DFT using bloom filter with dual indexing, DFT using roaring bitmap without dual indexing, and DFT using roaring bitmap with dual indexing, respectively.

The **default** is DFT–RB+DI, and when the context is clear, we will simply use DFT to represent this variant.

Evaluation metrics. We focus on evaluating the following metrics in our experiments:

Query Latency: end to end execution time for a query.

Selectivity: let $C_{\mathbb{M}}(Q)$ be the set of candidate trajectories that a method $\mathbb{M}$ has computed exact similarity distances for, with respect to a query trajectory Q. Selectivity of the method $\mathbb{M}$ on Q is defined as $C_{\mathbb{M}}(Q)/|\mathcal{T}|$.

Index Size: total memory footprint of an indexing structure (including all local indexes and the global index).

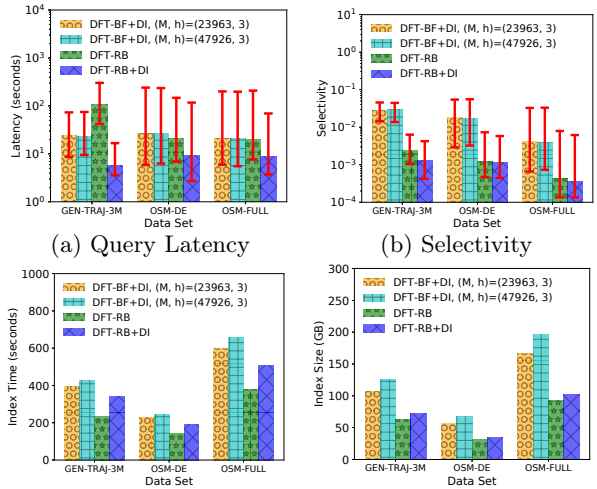
Index Time: time required to build all indexes used in a method before it can start serving the first query.

For all query-processing experiments, we executed 100 queries, where each query trajectory is a trajectory randomly sampled from the data set, to evaluate query latency and selectivity. Since the cost of different queries may vary dramatically, we report latency and selectivity using both the *5%-95% interval* and the *median* from these 100 queries.

Default parameters. By default, we set $k = 10$ and use the discrete segment Hausdorff distance as the similarity measure. For experiments on GEN-TRAJ, the default number of data trajectories is 3 million, which is 30.9GB. For all variants of our framework and the Traj Index baseline, the default value for c is 5. In all experiments except for the study of scalability against cluster size, we make use of all machines in the cluster.

7.2 Effectiveness of design choices in DFT

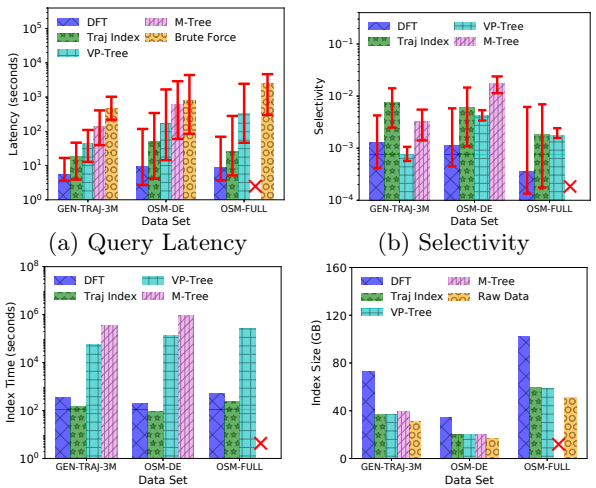
We first investigated the effectiveness of different design choices in DFT. We did not consider using raw bitmaps since it has caused heap memory overflow and crashed even on our smallest dataset.



(a) Query Latency (b) Selectivity

(c) Index Time (d) Index Size

Figure 7: Effectiveness of design choices for DFT.



(a) Query Latency (b) Selectivity

(c) Index Time (d) Index Size

Figure 8: Comparison against baseline solutions.

The (M, h) values for DFT−BF+DI in Figure 7 indicate number of bits (M) and the number of hash functions (h) used in a bloom filter. We used $(23963, 3)$ and $(47926, 3)$, which are the optimal configurations for achieving a false positive rate of 0.1 on $5,000$ and $10,000$ distinct items inserted into a bloom filter, respectively.

As shown in Figure 7(a) and 7(b), DFT−RB+DI has led to the best latency and selectivity among all variants. Specifically, it provides 1.5x-3.5x smaller query latency and around an order of magnitude better selectivity compared to DFT−BF+DI. Even though they share a similar index structure as described in Section 4.1, using bloom filters will introduce false positives and invalidate the more effective pruning strategy mentioned in Section 5.3.

Figures 7(c) and 7(d) show that using roaring bitmaps also achieves better index construction time and smaller index size than using bloom filters. Note that roaring bitmap is a flexible data structure and works extremely well on sparse bitmaps. It will use a very small space at internal nodes of the local indexes, for which their TID sets are sparse.

Dual indexing does not provide much benefit on improving selectivity as shown in Figure 7(b), but help reduce query latency significantly, due to the savings resulted from avoiding the step of regrouping segments back to a trajectory in the final calculation step for all candidate trajectories. In particular, as shown in Figure 7(a), dual indexing helps lower the query latency of DFT−RB by 2 to 15 times. Dual indexing works especially well in cases where most data trajectories span across multiple partitions, which may introduce non-trivial shuffling cost during trajectory reconstruction.

As for the index construction cost, as shown in Figures 7(c) and 7(d), dual indexing does introduce overhead, in terms of both construction time and index size, but this overhead is small and is only a factor of 1.3 to 1.5 compared to without dual indexing.

In all remaining experiments, we will use DFT−RB+DI as the default of our framework, which is simply dubbed DFT.

7.3 Comparison against baseline solutions

Figure 8 compares our solution against baseline solutions using three data sets: GEN-TRAJ (3 million trajectories), OSM-DE, and OSM-FULL. We omit the results for M-Tree on OSM-FULL as it took more than two weeks to build the M-Tree index on this data set, and its query performance is worse on the other data sets than the similar but faster VP-Tree. Clearly, DFT has achieved the smallest query latency, which is 2x-5x faster than Traj Index, 6x-25x faster than VP-Tree, 25x-68x faster than M-Tree and 65x-186x faster than Brute Force on average. DFT also shows more stable query latency than Traj Index (having smaller 5%-95% intervals).

In terms of query selectivity, DFT is constantly better than Traj Index by a factor from 5 to 7. VP-Tree shows the smallest variance in its query selectivity, but is having a much worse selectivity than DFT on both OSM-DE and OSM-FULL datasets. The median selectivity of VP-Tree is slightly better than DFT on the GEN-TRAJ-3 million data set. Nevertheless, VP-Tree still has much worse query latency than DFT even when its query selectivity is slightly better, as its search procedure incurs linear cost. M-Tree has worse selectivity than VP-Tree as it is designed as a dynamic structure and has higher fanout.

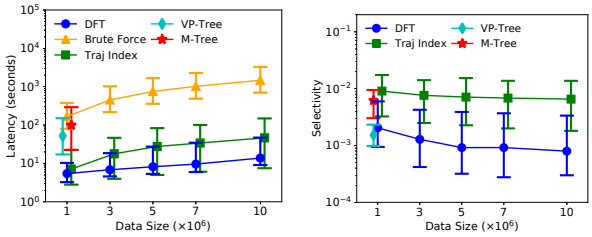
As to the indexing cost, VP-Tree, M-Tree and Traj Index introduce a small overhead on memory consumption compared to the size of the raw data, while our solution is more expensive due to dual indexing. Nevertheless, the index size of DFT is about twice of the raw data size. Note that the index size *DOES ALREADY INCLUDE* the size of the raw data (since they are essentially data items at the leaf levels, which means the size of DFT's index is only 1x of the raw data sizes.

Traj Index shows the fastest indexing time, while our solution is roughly 2x slower than Traj Index. In contrast, it is very expensive to build the VP-Tree and M-Tree. Their construction time are too expensive to be practically useful: for GEN-TRAJ-3M, it takes 15 hours to build VP-Tree and 101 hours (which is more than 4 days) to build M-Tree. This is because both VP-Tree and M-Tree will invoke a large number of similarity measure calculations (which is extremely expensive) during index building. M-Tree performs much worse as it is built dynamically with node split procedures; we expect variants that bulk load to perform similar to VP-Tree.

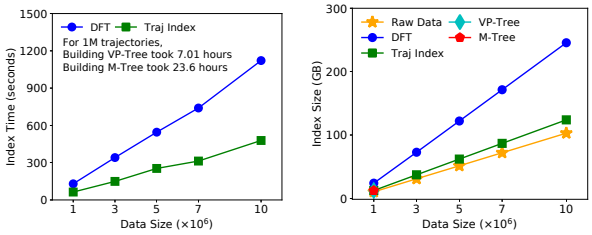
7.4 Scalability

Next, we study the scalability of different solutions with respect to the size of $\mathcal{T}$, using the GEN-TRAJ data set. Figure 9(a) shows that the query latency of all methods grows linearly to data size, while DFT always has the best performance, which is almost 1 order of magnitude faster than Traj Index, and close to or more than 2 orders magnitude faster than VP-Tree, M-Tree, and Brute Force respectively.

For selectivity, all methods show a similar trend and become more selective as the data size grows. The selectivity of our solution is always close to or more than one magnitude better than those of Traj Index and M-Tree. VP-Tree shows slightly better se-

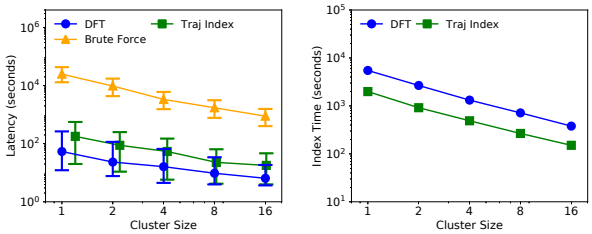


(a) Query Latency (b) Selectivity



(c) Index Time (d) Index Size

Figure 9: Scalability with respect to data size.



(a) Query Latency (b) Index Time

Figure 10: Scalability with respect to cluster size.

lectivity than DFT, but its overall query latency is still much worse due to the reasons explained in last sub-section.

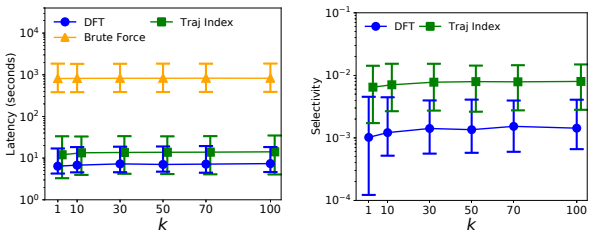
Note that we have ignored most results of VP-Tree and M-Tree from this experiment, due to the fact that they become extremely expensive to build as data size grows. For example, it already takes more than 7 hours to build VP-Tree and more than 23 hours to build M-Tree even with just 1 million trajectories, while DFT and Traj Index only take less than 100 seconds.

Figure 9(c) shows that the indexing time of all methods grows linearly to the data size. The index construction time of our solution is around 2x slower than that of Traj Index. And, VP-Tree and M-Tree take much more time to build than other solutions, and have the fastest growth rate in construction time as data size increases. Figure 9(d) shows that VP-Tree, M-Tree and Traj Index only introduce a small storage overhead compared to raw data size, while our solution is about 2x-2.3x of raw data size (including the raw data size; so the index size is only 1x-1.3x of raw data size).

In addition to scalability against data size, we also conduct experiments to study query and index performance against cluster size. Note that we omit results for index size and query selectivity as they are not influenced by the cluster size. As shown in Figure 10, DFT, Traj Index, and baseline all demonstrate roughly linear scalability against the cluster size while DFT achieves the smallest query latency. For index time, both DFT and Traj Index scale linearly to the cluster size and the difference between them for index construction time is around 2x.

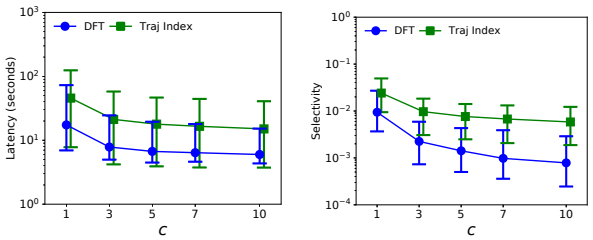
7.5 Impacts of k, c, and query size

Figures 11(a) and 11(b) show that k does not have a significant impact on query latency and selectivity for all methods. This is because, when k only increases moderately, distance calculation for the final candidate set $C(Q)$ dominates the query processing



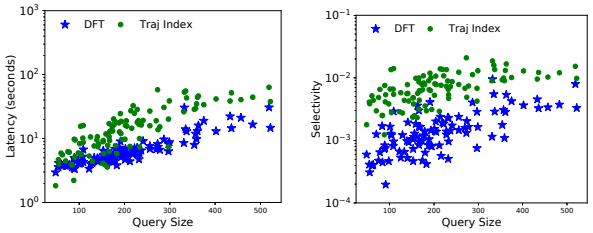
(a) Query Latency (b) Selectivity

Figure 11: Impact of k.



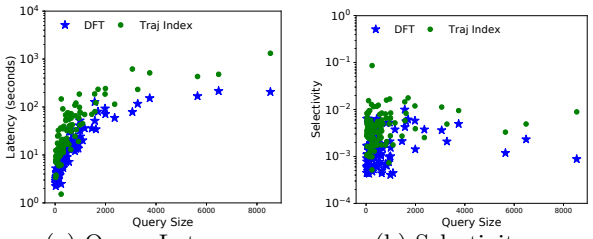
(a) Query Latency (b) Selectivity

Figure 12: Impact of c.



(a) Query Latency (b) Selectivity

Figure 13: Influence of query size on GEN_TRAJ_3M.



(a) Query Latency (b) Selectivity

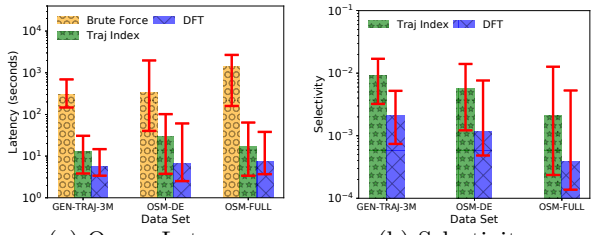
Figure 14: Influence of query size on OSM_DE.

time, and a moderate increase in k value doesn't have change the size of $C(Q)$ significantly.

Figure 12 shows how the value of c does have an impact to the query performance of Traj Index and DFT. As c grows, both solutions have achieved smaller query latency and better selectivity. But note that after $c \geq 5$, query latency of both solutions stop seeing any significant reduction.

Number of segments in a query trajectory, namely query size, is an important factor that will influence both query latency and selectivity. Intuitively, it is more expensive to calculate similarity measures when the query trajectory has more segments. Furthermore, longer trajectories are likely to pass through larger spatial regions, which may limit the effectiveness of partition based pruning.

Figures 13 and 14 demonstrate how query size impacts the query latency and selectivity of both Traj Index and DFT on GEN-TRAJ and OSM-DE datasets. Clearly, a larger query trajectory leads to longer query latency and worse query selectivity, with the exception on OSM-DE, where large queries may actually have a better selectivity. This is due to the fact that OSM-DE contains trajectories from a small area (Germany vs the world), so more segments in a query trajectory may lead to better pruning power using our pruning bound in Theorem 1. Nevertheless, they still lead to more



(a) Query Latency (b) Selectivity

Figure 15: Performance on discrete segment Fréchet distance.

expensive query processing time, as the distance calculation for the final candidate set becomes more expensive with more segments in the query trajectories.

In all cases, DFT has significantly outperformed Traj Index (note that the y-axis is in log scale).

7.6 Discrete Segment Fréchet Distance

Finally, we show how our solution works on the discrete segment Fréchet distance. Note that since our framework uses the same index structures as those for Hausdorff distance, the indexing time and index size remain the same as demonstrated in Section 7.3. As VP-Tree takes too long to build (more than 12 hours), we omitted its results, which follows a similar trend as that in Figure 8. Figure 15 shows that DFT achieves the best query latency and selectivity.

8. CONCLUSION

We present a generic and scalable framework for processing distributed similarity search on a large set of trajectories. Our approach utilizes a segment-based approach with a number of optimizations. We illustrate how to support the classic Hausdorff and Fréchet distances using our framework, and investigate challenges in instantiating our framework in a distributed system. Interesting future work includes how to extend our framework to support other trajectory similarity metrics than the discrete segment-based similarity measures used in this work, deal with updates, process sub-trajectory similarity search, and search for a tighter pruning bound. Another interesting extension is to generalize our framework for non-metric measures using the ideas of embedding.

9. ACKNOWLEDGEMENT

We appreciate the comments from the annoymous reviewers. Authors thank the support from NSF grants 1200792, 1251019, 1350888, 1443046, and 1619287. Feifei Li was also supported in part by NSFC grant 61428204 and a Huawei gift award.

10. REFERENCES

[1] P. K. Agarwal, R. Ben Avraham, H. Kaplan, and M. Sharir. Computing the discrete frechet distance in subquadratic time. *Siam Journal of Computing*, 43:429–449, 2014.

[2] P. K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. In *STOC*, 2014.

[3] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *JCG Appl.*, 5:75–91, 1995.

[4] H. Alt, C. Knauer, and C. Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 2004.

[5] H. Alt and L. Scharf. Computing the hausdorff distance between curved objects. *JCG Appl.*, 18(4):307–320, 2008.

[6] Y.-B. Bai, J.-H. Yong, C.-Y. Liu, X.-M. Liu, and Y. Meng. Polyline approach for approximating Hausdorff distance between planar free-form curves. *CAD*, 43:687–698, 2011.

[7] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 2008.

[8] T. Bozkaya and Z. M. Özsoyoglu. Indexing large metric spaces for similarity search queries. *TODS*, 24(3):361–404, 1999.

[9] S. Cabello, P. Giannopoulos, C. Knauer, and G. Rote. Matching point sets with respect to the Earth mover's distance. *Computational Geometry: Theory and Applications*, 39:118–133, 2008.

[10] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[11] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Softw., Pract. Exper.*, 2016.

[12] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[13] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[15] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[16] M. de Berg, A. F. Cook IV, and J. Gudmundsson. Fast frechet queries. In *Symposium on Algorithms and Computation*, 2011.

[17] A. Driemel and S. Har-Peled. Jaywalking your dog – computing the Fréchet distance with shortcuts. *Siam Journal of Computing*, 42:1830–1866, 2013.

[18] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.

[19] A. W. Fu, P. M. Chan, Y. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDBJ*, 2000.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[21] J. Hangouët. Computation of the Hausdorff distance betwene plane vector polylines. In *Proceedings AutoCarto 12*, 1995.

[22] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, 1994.

[23] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149, 2008.

[24] J.-G. Lee, J. Han, X. Li, and H. Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. In *VLDB*, 2008.

[25] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[26] S. T. Leutenegger, M. Lopez, J. Edgington, et al. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.

[27] C. Long, R. C. Wong, and H. V. Jagadish. Direction-preserving trajectory simplification. *PVLDB*, 6(10):949–960, 2013.

[28] C. Long, R. C. Wong, and H. V. Jagadish. Trajectory simplification: On minimizing the direction-based error. *PVLDB*, 8(1):49–60, 2014.

[29] S. Ranu, D. P, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, 2015.

[30] B. K. Sriperumbudur, K. Fukumizu, and G. R. G. Lanckriet. Universality, characteristic kernels and RKHS embedding of measures. *Journal of Machine Learning Research*, 12:2389–2410, 2011.

[31] H. Su, K. Zheng, J. Huang, H. Wang, and X. Zhou. Calibrating trajectory data for spatio-temporal similarity analysis. *VLDBJ*, 24(1):93–116, 2015.

[32] H. Su, K. Zheng, H. Wang, J. Huang, and X. Zhou. Calibrating trajectory data for similarity-based analysis. In *SIGMOD*, 2013.

[33] H. Su, K. Zheng, K. Zeng, J. Huang, S. W. Sadiq, N. J. Yuan, and X. Zhou. Making sense of trajectory data: A partition-and-summarization approach. In *ICDE*, pages 963–974, 2015.

[34] H. Su, K. Zheng, K. Zheng, J. Huang, and X. Zhou. Stmaker - A system to make sense of trajectory data. *PVLDB*, 7(13):1701–1704, 2014.

[35] C. F. Torres and R. Trujillo-Rasua. The fréchet/manhattan distance and the trajectory anonymisation problem. In *DBSec*, pages 19–34, 2016.

[36] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[37] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, 2002.

[38] H. Wang, H. Su, K. Zheng, S. W. Sadiq, and X. Zhou. An effectiveness study on trajectory similarity measures. In *ADC*, 2013.

[39] H. Wang, K. Zheng, X. Zhou, and S. W. Sadiq. Sharkdb: An in-memory storage system for massive trajectory data. In *SIGMOD*, pages 1099–1104, 2015.

[40] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.

[41] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, 1998.

[42] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, 1993.

[43] R. Ying, J. Pan, K. Fox, and P. K. Agarwal. A simple efficient approximation algorithm for dynamic time warping. In *SIGSPATIAL*, 2016.

[44] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *SIGSPATIAL*, 2010.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[46] Z. Zhang, K. Huang, and T. Tan. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. In *ICPR*, 2006.

[47] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.