# TrajSpark: A Scalable and Efficient in-memory Management System for Big Trajectory Data

Zhigang Zhang, Cheqing Jin✉, Jiali Mao, Xiaolin Yang, and Aoying Zhou

School of Data Science and Engineering, East China Normal University, China
{zgzhang,jlmao1231,xlyang}@stu.ecnu.edu.cn
{cqjin,ayzhou}@sei.ecnu.edu.cn

**Abstract.** The widespread application of mobile positioning devices has generated big trajectory data. Existing disk-based trajectory management systems cannot provide scalable and low latency query services any more. In view of that, we present TrajSpark, a distributed in-memory system to consistently offer efficient management of trajectory data. TrajSpark introduces a new abstraction called IndexTRDD to manage trajectory segments, and exploits a global and local indexing mechanism to accelerate trajectory queries. Furthermore, to alleviate the essential partitioning overhead, it adopts the time-decay model to monitor the change of data distribution and updates the data-partition structure adaptively. This model avoids repartitioning existing data when new batch of data arrives. Extensive experiments of three types of trajectory queries on both real and synthetic dataset demonstrate that the performance of TrajSpark outperforms state-of-the-art systems.

**Keywords:** big trajectory data; in-memory; low latency query

## 1 Introduction

Recently, with the explosive development of positioning techniques and popular use of intelligent electronic devices, trajectory data of MOs (Moving Objects) has been accumulated rapidly in many applications, such as location-based services (LBS) and geographical information systems (GIS). For example, DiDi [1], the largest one-stop consumer transportation platform in China, now has 1.5 million registered active drivers, and provides services for more than 300 million passengers. The total length of all trajectories generated in this platform reaches around 13 billion kilometers in 2015. Moreover, the volume of trajectory data increases in a surging way. In March 2016, the number of trajectories generated in one day has already exceeded 10 million. It is challenging to provide real-time service over such data. However, as almost all of existing trajectory management systems are disk-oriented (e.g., TrajStore [4], Clost [13], and Elite [18]), they cannot support low latency query services upon big trajectory data.

[1] `http://www.xiaojukeji.com/en/taxi.html`

Recently, in-memory computing systems are a widely used to provide low latency query services. For instance, Spark [2], a distributed in-memory computing system, has been widely used. Spark provides a data abstraction called RDDs (Resilient Distributed Datasets), to maintain a collection of objects that are partitioned across a cluster of machines. Users can manipulate RDDs conveniently through a batch of predefined operations. However, Spark is lack of indexing mechanism upon RDDs and needs to scan the whole dataset for a given query. Recently, some Spark-based system prototypes have been proposed to process big spatial data, including SpatialSpark [19], LocationSpark [14], GeoSpark [20] and Simba [17]. Amongst them, SpatialSpark implements the spatial join query on top of Spark, but it does not index RDDs. GeoSpark provides a new abstraction, called SRDD, to represent spatial objects such as points and polygons. Although it embeds a local index in each SRDD partition, global index is not supported. LocationSpark proposes a solution to solve query skewness. In contrast of them, Simba extends Spark SQL with native support to spatial operations. Meanwhile, it introduces both global and local indexes over RDDs. However, these prototypes view data as a set of spatial points and employ the point-based indexing strategies. Such strategies decrease the trajectory query performance as points of an MO need to be retrieved from different nodes and sorted to form a chronologically ordered sequence [4]. Moreover, they are by nature designed to manage a static dataset and cannot efficiently react to data distribution changes as data increases. To handle a batch of new data, the whole dataset should be repartitioned from scratch, which is quite computation costly.

Inspired by above observations, we design and implement TrajSpark (Trajectory on Spark) system to support low-latency queries over big trajectory data. TrajSpark proposes a new abstraction called IndexTRDD to manage trajectories as a set of trajectory segments. To accelerate query processing, it imports the global and local indexing mechanism which embeds a local hash index in each data partition and builds a global index over these partitions. Furthermore, TrajSpark tracts the change of data distribution by using a time decay model to continuously support efficient management over the daily increasing big trajectory data. Our main contributions can be summarized as follows:

- We first propose TrajSpark to mange the big trajectory data while existing Spark-based systems only support a static big spatial dataset.
- We introduce IndexTRDD, an RDD of trajectory segments, to support efficient data storage and management by incorporating a global and local indexing strategy.
- We monitor the change of data distribution by importing a time decay model which alleviates the repartitioning overhead occurred in existing Spark-based systems and gets a good partition result at the same time.
- We execute three types of trajectory queries on TrajSpark and conduct extensive experiments to evaluate query performance. Experimental results demonstrate the superiority of TrajSpark over other Spark-based systems.

[2] `http://spark.apache.org/`

The rest of the paper is organized as follows. Section 2 reviews related works. We give an overview of TrajSpark in Section 3, and detailed the system in In Section 4 and Section 5. In Section 6, we introduce the implementation of three typical trajectory queries in TrajSpark. Section 7 provides an experimental study of our system. Finally, we give a brief conclusion in Section 8.

## 2 Related Work

We review the work mostly related to our research in this section.

**Centralized Trajectory Management Systems:** There are many centralized systems to manage trajectory data. PIST, an off-line system, supports indexes over points. It first partitions data according to a spatial index, and then supports a temporal index in each partition [2]. SETI segments trajectories into sub-trajectories with the guidance of a spatial index, and groups them into a collection of spatial partitions [3]. It shows that supporting index over trajectory segments is more efficient than indexing trajectory points. TrajStore not only uses an I/O cost model to dynamically segment trajectories, but also uses clustering and compressing techniques to reduce storage overhead. But it only supports range query [4]. These systems can not meet the requirement of big data processing as they adopt the centralized architecture.

**Disk-based Distributed Spatial and Spatio-temporal Data Management Systems:** Recently, some distributed disk-based systems have been proposed to manage spatial data by utilizing the Hadoop[3] framework. SpatialHadoop [5] pushes spatial data inside Hadoop core by adopting a layered design and supports efficient spatial operations by employing a two-level index structure. AQWA [1] is an improved version of SpatialHadoop by proposing a workload-aware partition strategy which divides those frequently accessed regions into more fine-grained subregions. There are also some systems particularly designed for big spatio-temporal/trajectory data management. PRADASE [10] and Clost [13] are directly built on top of Hadoop and accelerate queries through a global spatio-temporal index. MD-HBase [11], RHBase [6] and GeMesa [7] are built on top of distributed key/value stores, and they use space-fill curves [6, 11] and Geohash [7] algorithms to map spatio-temporal points into single-dimension space separately. Different from above works, Elite [18] is built on top of OpenStack[4] for big uncertain trajectorie . Nevertheless, all the above systems are disk-based, and none of them can provide low latency query services.

**Memory-based Spatial Data Management Systems:** SharkDB [16] proposes a column-wise storage format to manage trajectory within main memory. However, it is deployed on a big-memory machine and cannot scale out to the distributed environment. Besides, some distributed in-memory spatial data management systems have been proposed. SpatialSpark [19] and GeoSpark [20] are two systems built on top of Spark. SpatialSpark is specifically designed for spatial join queries. GeoSpark proposes a new RDD called SRDD to support

[3] http://hadoop.apache.org/
[4] http://www.openstack.org/

typical spatial queries. It supports a local index for each partition of SRDD. In comparison of GeoSpark, LocationSpark proposes a solution to solve query skewness by using Bloom Filter [14]. Simba, built on top of Spark SQL, supports multi-dimensional data queries [17]. Moreover, both a query optimizer which leverages indexes and some spatial-aware optimizations are imported in Simba to improve query efficiency. The above systems process spatial data as independent data points, while trajectory data are usually viewed as a collection of time series. Directly using these systems to process trajectories queries may sacrifice the query efficiency. Moreover, these systems cannot scale well when a new batch of data is imported to the system.

## 3 System Overview

The architecture of TrajSpark, as shown in Fig.1, is composed of four layers: 1) *Apache Spark Layer*, where regular operations and fault tolerance mechanisms are supported by Apache Spark, 2) *Trajectory Presentation Layer*, where a new abstraction called IndexTRDD is designed to support indexes over trajectory data. 3) *Assistant Data Layer*, which monitors the change of data distribution and guides the partitioning of forthcoming data. A global index which indexes partitions of IndexTRDD is maintained. 4) *Query Processing Layer*, which processes trajectory queries in an efficient way by utilizing indexes.

- **Apache Spark Layer:** This layer is directly inherited from Apache Spark, and the description of it is omitted in this paper.
- **Trajectory Presentation Layer:** In this layer, the trajectory segments that are spatio-temporally close will be grouped into the same data partition. In each partition, segments belonging to the same MO are depicted in a space-efficient format. A new abstract called IndexTRDD is proposed to organize all those segments, and a rich operations library is provided to manipulate trajectories and IndexTRDD. As the core of TrajSpark, we give a detailed description in Sec. 4.

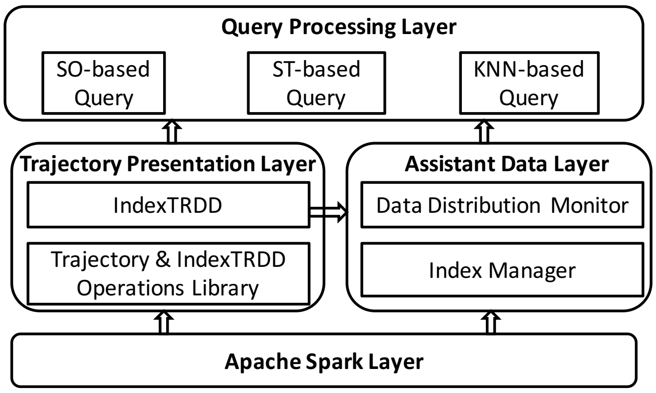


**Fig. 1.** System Overview

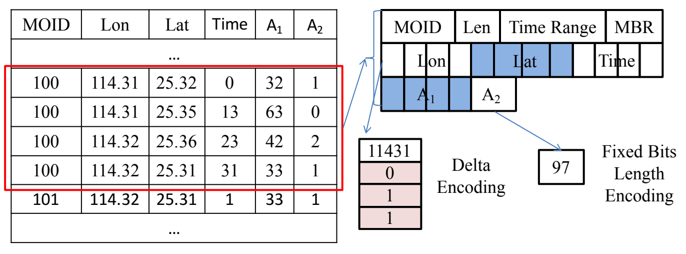
– **Assistant Data Layer:** A few statistics are maintained in this layer to record the change of data distribution. This layer also maintains a global index which indexes all partitions of IndexTRDD. This layer is detailed in Sec. 5.
– **Query Processing Layer:** We introduce the implementation of three typical trajectory quires in this layer (detailed in Sec. 6).

## 4 Trajectory Presentation Layer

### 4.1 Trajectory Segment Presentation

Instead of storing trajectories as a time series directly, the raw trajectories generated from data sources are usually stored as GPS logs and each log record corresponds to a trajectory point. The schema of such points can be viewed as a table with the following form: ($MOID$, $Location$, $Time$, $A_1$, $\cdots$, $A_n$), where $MOID$ is the identification of an MO, $Time$ and $Location$ are the temporal and spatial information. The rest attributes vary in different data sources. Although it is simple to represent the trajectories as an RDD of points by directly loading the raw data into Spark, it leads to a high storage overhead due to the limitation of row-stores. Moreover, as analyzed in [4], trajectory segment based technologies can improve the query efficiency more significantly than point based ones .

To covert raw data into trajectory segments, TrajSpark partitions points that are spatio-temporally close into the same partition firstly (detailed in Sec. 4.2). Then, points of the same MO are sorted to form a trajectory segment and the segment is packed into a space-efficient format as shown in Fig. 2(a). In this format, values of the same attribute are stored and compressed continuously. For *numberic* attributes(such as time and location attribute), data are compressed by *delta encoding* [4]. For an *enum* attribute($A_2$ in Fig. 2(b)), fixed bits length encoding is used. For other attributes, such as the *string*, we simply use the *gzip* compression. Besides, we maintain other sketch data such as the $MOID$, $Len$ (length) and $MBR$ (Minimum Bounding Rectangle) of the segment to achieve a quick pruning for queries. Finally, data in each partition are changed to a set of compressed trajectory objects, and the whole dataset is transformed to an RDD of such objects(we call this RDD as TRDD).



| MOID | Lon | Lat | Time | $A_1$ | $A_2$ |
|---|---|---|---|---|---|
| | | ... | | | |
| 100 | 114.31 | 25.32 | 0 | 32 | 1 |
| 100 | 114.31 | 25.35 | 13 | 63 | 0 |
| 100 | 114.32 | 25.36 | 23 | 42 | 2 |
| 100 | 114.32 | 25.31 | 31 | 33 | 1 |
| 101 | 114.32 | 25.31 | 1 | 33 | 1 |
| | | ... | | | |

(a) Raw Trajectory Data (b) Trajectory Storage Format

**Fig. 2.** Trajectory Presentation

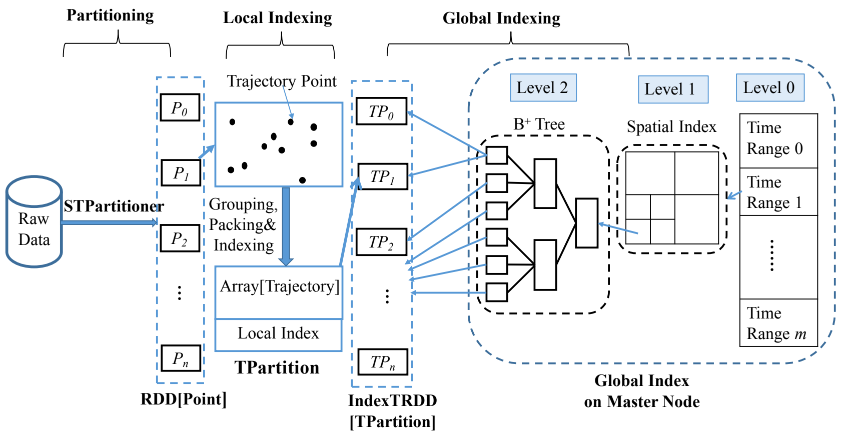


**Fig. 3.** Indexing from Raw Data

## 4.2 Indexing for Trajectory Data

In the above section, we introduce how to transform the GPS logs into TRDD. While TRDD only supports sequential scan for queries which is very expensive as it needs to access the whole dataset. Hence, we need to support indexing strategy to improve the query efficiency and at the same time without changing the core of Spark. To overcome these challenges, we propose IndexTRDD which changes the storage structure of TRDD by embedding a local hash index in each partition, and get $O(1)$ computation to retain the trajectory of a given MO. Furthermore, we build a global index over data partitions of IndexTRDD to prune irrelevant partitions. Fig. 3 details the indexing mechanism which can be divided into three phases: partitioning, local indexing, and global indexing.

**Partitioning.** In this phase, TrajSpark loads the raw dataset from disk into memory as an RDD of trajectory points. This RDD needs to be repartitioned according to the following three constraints: 1)*Data Locality.* Trajectory points that are spatio-temporally close to each other should be assigned to the same partition. 2)*Load Balancing.* All partitions should be roughly of the same size. 3)*Partition Size.* Each partition should have a proper size so as to avoid memory overflow. Spark provides two predefined partitioners for one-dimensional keys, including range and hash partitioner. However, they cannot fit well for multi-dimensional data such as trajectory. To address this problem, TrajSpark defines a new partitioner named **STPartitioner** which contains a spatial Quad tree or k-d tree index. The spatial index can be learned from data distribution and ensures that each leaf node contains the same amount of data. STPartitioner uses the boundaries of those leaf nodes to partition points. Then, trajectory points located in the same boundary are grouped together. Finally, due to the constraint of partition size, TrajSpark splits points belonging to the same boundary into a few data partitions according to $MOID$(in default) or time attribute, and makes sure each partition satisfy the above constraints.

**Local indexing.** After the partitioning phase, the dataset is still an RDD of trajectory points. In this step, we transform the above RDD into TRDD by grouping and packing such points firstly. Then, we add a local index at the head of each partition which maps the MOID of each trajectory to its subscripts. We call the combined data structure of index and trajectory array as TPartition. So the whole dataset is transformed into an RDD of TPartitions, where the RDD is **IndexTRDD**. Finally, we collect the ID (each partition of RDD has a unique ID), the spatial and temporal ranges of each data partition (a TPartition object) to construct the global index.

**Global indexing.** The last phase is to build the global index *gIndex* over all partitions. As shown in Fig. 3, *gIndex* is a three-level hybrid index. Data is divided by the level-0 coarse time ranges according to its temporal attribute firstly. Each coarse time range corresponds with a level-1 spatial index which is used by STPartitioner. To index partitions that belong to the same spatial boundary, a level-2 B$^+$-Tree is used. When TrajSpark is initialized with the first batch of data, level-0 index contains only one value (the beginning timestamp of that batch of data), and the level-1 spatial index is the same one used in STPartitioner. The spatial and temporal information collected from all partitions are used to construct the level-2 indexes. Each spatial range in level-1 index corresponds with a level-2 index. TrajSpark keeps *gIndex* in the memory of master node and updates it when new data partitions arrive. Even for a big trajectory dataset, the number of partitions is not very large (shown in Fig. 4(c)). Thus, the global index can be easily fitted in the memory of master node.

## 5 Assistant Data Layer

### 5.1 Data Distribution Monitor

In real applications, new batches of data are appended on an hourly or daily basis [1], and the data distribution changes accordingly. On one hand, a static partitioning strategy results in unbalanced data partitions. On the other hand, if we repartition the whole dataset (required in existing systems [14, 17, 19, 20]) when each batch of data arrives, it leads to an expensive workload. Meanwhile, it is worthless to repartition the old data, because new data are more valuable than those old ones. So, when a new batch of data arrives, TrajSpark tries to only partition this batch of data without touching existing data which differs from the target of AQWA [1] who needs to repartition part of existing data. Moreover, TrajSpark focuses on long term data distribution changes and also tries to alleviate the influence of temporal changes.

In the light of above considerations, TrajSpark adopts the time decay model to depict the change of data distribution by giving the recent data a higher weight. TrajSpark divides the whole spatial area into $m * m$ fine-grained cells and computes the data distribution by counting the number of points in each cell. When a new batch of data arrives, TrajSpark maintains two matrices: $A_{existing}$ and $A_{new}$, which separately record the distribution of data that have been loaded in TrajSpark and the new one. After loading the new batch of data, $A_{existing}$

decays weight by dividing $\gamma$ firstly ($\gamma$ is the decay factor, which gives *older* data lower weights). Then, $A_{new}$ is added to $A_{existing}$ and set to zero. Observe that after a batch of data is appended, $A_{existing}$ changes accordingly. To better depict the change of $A_{existing}$, we use the notation $A^n_{existing}$ to represent the spatial distribution of data after $n$ batches of data are appended.

To depict the adaptivity of our partitioning strategy, we define a new matrix $PA_c$ (Partition with $A$) which is initialized with $A^0_{existing}$, and create the spatial index of STPartitioner from $PA_c$ by partitioning the whole spatial area into subregions with equal number of points. After the $n$-th batch of data is loaded, if the difference between $A^n_{existing}$ and $PA_c$ is larger than a given threshold, it means that the distribution of recently loaded data has greatly changed, TrajSpark updates the value of $PA_c$ with $A^n_{existing}$, and updates STPartitioner using a new spatial index created from the new $PA_c$ to partition the incoming data. In TrajSpark, we use the JSD distance to measure the difference between two data distributions [12] (both the distribution matrices should be normalized before computing). The lazy-update property of time decay model enables TrajSpark to resist abrupt or temporary data distribution changes.

### 5.2 Index Manager

Index manager mainly supports the update and persistence of the *gIndex*. Two cases will lead to the update of *gIndex*. The first is when the STPartitioner updates its spatial index. At this case a new time range will be added to level-0 index, and the spatial index will be added to level-1 as its children. The second case is when all partitions of the new data have been added to IndexTRDD, the information of these partitions will be added to the level-2 index of *gIndex*. The index manger stores *gIndex* in the memory of the master node. Besides, TrajSpark also chooses to persist it into the file system (after its updating) and has the option of loading it back from the disk. This enables TrajSpark to load indexes back to the system even after system failure. It needs to mention that, TrajSpark supports spatio-temporal operations for *gIndex*, such as *intersect*, *overlap* and so on, to find partitions satisfying the query constraints.

## 6 Query Processing Layer

Typical trajectory queries include SO(Single Object)-based query [8, 10, 13], STR (Spatio-Temporal Range)-based [8, 15, 16] query and KNN (K Nearest Neighbor)-based query [8, 11, 16]. In this section, we introduce how TrajSpark efficiently processes these queries by utilizing indexes and the operation libraries.

### 6.1 SO-based Query

An SO-based query retrieves the trajectory of a given MO by receiving two parameters: *moid* and *tRange*, where *moid* denotes the ID of an MO and *tRange* is the temporal constraint. Spark expresses this query as an RDD *filter* action,

**Algorithm 1** SO-based query

```
Input: moid, tRange;
Output: one trajectory;
1: pids = gIndex.intersect(tRange);
2: ts = IndexTRDD.PartitionPruningRDD(pids)
                  .getTraWithID(moid).mapValues(sub(tRange));
3: return ts.reduceByKey(merge).collect();
```

which requires scanning the whole dataset. TrajSpark can achieve better performance by utilizing indexes. It leverages two observations: 1). The level-0 index is sufficient to prune irrelevant partitions, and the level-2 index can find the partitions whose time ranges are intersected with *tRange*. 2). For each partition, the trajectory of *moid* can be filtered quickly according to the local *hash* index.

Based on the above observations, Algorithm 1 introduces the detailed steps. Firstly, TrajSpark traverses the global index to find data partitions whose time ranges are intersected with the *tRange* (line 1). It needs to mention that, the global index *gIndex* is a spatio-temporal index, and the input parameter for *intersect* operation can also contain a spatial constraint. Next, IndexTRDD calls a Spark API— PartitionPruningRDD, to mark required partitions. Then, TrajSpark randomly accesses the trajectory in each partition according to the given *moid* and finds the sub-trajectory located in the *tRange* (line 2). Finally, all sub-trajectories of the given MO are merged into one. Note that TrajSpark provides the *merge* function to merge two trajectory segments of the same MO.

## 6.2 STR-based Query

An STR-based query retrieves trajectories within a spatio-temporal range. It receives two parameters *tRange* and *sRange*. By utilizing the indexes, TrajSpark can also achieve better performance than the *filter* operation of Spark. Algorithm 2 sketches basic steps to process such queries. At first, TrajSpark traverses the global index to filter partitions that are intersected with the given spatio-temporal range (line 1). Then, for each partition, TrajSpark filters candidates whose spatial bounding box and temporal range are intersected with the given spatio-temporal constraint. Furthermore, it finds a sub-trajectory which is bounded by the spatio-temporal constraint for each candidate (line 2).

**Algorithm 2** STR-based query

```
Input: tRange, sRange;
Output: a set of trajectories;
1: pids = gIndex.intersect(tRange, sRange);
2: ts = IndexTRDD.PartitionPruningRDD(pids)
                  .filter(tRange, sRange).mapValues(sub(tRange, SRange));
3: return ts.reduceByKey(merge).collect();
```

**Algorithm 3** KNN-based query

```
Input: $IndexTRDD$, $disM$;
Output: the $k$ most similar trajectories to $tr$;
1: $mbr = tr.MBR$, $tRange = tr.TimeRange$;
2: repeat
3:     $pids$ = gIndex.intersect($mbr$, $tRange$);
4:     ts = IndexTRDD.PartitionPruningRDD($pids$)
                         .filter($mbr$, $tRange$).reduceBykey(merge);
5:     $mbr$.expand($1 + \alpha$);
6: until ($ts$.size $> k$)
7: $candidate$=$ts$.collect();
8: return $candidate$.map($t \rightarrow$(disM($t, tr$),$t$)).sortByKey.top($k$);
```

### 6.3 KNN-based Query

There are many variations of KNN-based query, and we focus on finding top-$k$ trajectories who are most similar to the reference one. This kind of query is very common in trajectory patten analysis, and we represent it with **KNN(**$tr$**,** $disM$**)**. Here, $tr$ refers to the query reference, and $disM$ refers to the distance/similarity metric between trajectories (popular metrics such as Euclidean distance, DTW and LCSS are supported in TrajSpark). The processing procedure is shown in Algorithm 3. TrajSpark gets the MBR and time range of $tr$ firstly (line 1). This is because candidate results are spatio-temporally close to the reference, the using of $mbr$ and $tRange$ facilitates the pruning of candidate. Then, TrajSpark filters candidate partitions using the global index (line 3). After that, sub-trajectories are further pruned and merged into complete trajectories (line 4). These trajectories are the candidates of the final result. However, if the number of these candidates is smaller than $k$, TrajSpark expands the region of $mbr$ (the center of $mbr$ will not change, while the width and length become $1 + \alpha$ ($0 < \alpha < 1$) times) and re-executes the spatio-temporal query until the number of candidates is larger than $k$. Here, the default value of $\alpha$ is set to 0.2. Finally, TrajSpark measures the similarity for those candidate trajectories and selects $k$ smallest ones as the final result.

## 7 Experiments

### 7.1 Experimental Setup

We evaluate the performance of TrajSpark in this section. All experiments are conducted on a 12-node clustering running Spark 1.5.2 over Ubuntu 12.0.4. Each node is equipped with an 8 cores Intel E5335 2.0GHz processor and 16GB memory. The Spark cluster is deployed in standalone mode.

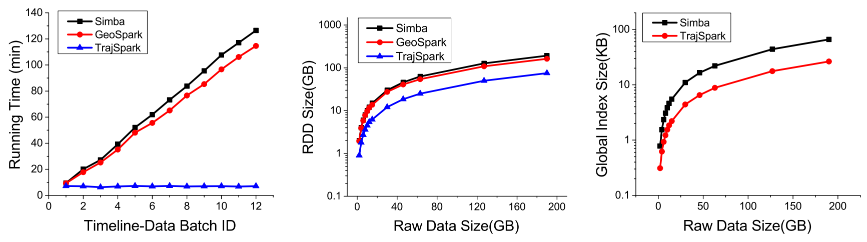
Two trajectory datasets of Beijing taxis [9], including a real dataset and a synthetic one, are used to evaluate the performance. The real one, gathered by 13,007 taxis in 3 months (from October to December in 2013), has 2.5 billion records and comprises about 190 GB. Each record contains the following attributes: taxi ID, time, longitude, latitude, speed and many other descriptive information. To better show the scalability of TrajSpark, we generate the synthetic dataset by extending the real one. In the synthetic one, every taxi reports its location every five seconds when it is taken by passengers, and the number of records is 18 billion comprising about 1.4TB. It needs to mention that the former dataset can be completely loaded into the distributed memory, while the storage overhead of the latter one far exceeds the memory capability of our cluster. Thus, only partial of the synthetic data can be loaded in memory.

We compute the MBR of the spatial range of Beijing and split the rectangle into $1,000 * 1,000$ cells where each cell covers an area of nearly 180m * 180m. We compare the performance of TrajSpark with GeoSpark and Simba in terms of query latency and scalability. The latency is represented by the average running time of a few queries, and the scalability is evaluated when different amount of data is loaded into those systems.

## 7.2 Performance of Data Appending

Firstly, we study the performance of data appending when batches of data are loaded into those systems. Fig. 4(a) gives the running time when batches of real dataset are appended (each batch comprises about 32G). In GeoSpark and Simba, the time cost of appending a batch of data increases linearly as the volume of existing data increases, because they should repartition both existing and the new batch of data. While TrajSpark requires less time and the loading time keeps steady with the increase of data volume. This is because TrajSpark only needs to partition the new batch of data and also can reach balanced data partitions. So, in real big data applications where the volume of data grows rapidly, TrajSpark outperforms GeoSpark and Simba significantly.

Next, we investigate the storage overhead of different systems using the real dataset and show the results with the RDD size of loaded data in Fig. 4(b). TrajSpark has the lowest storage cost due to data compression, while Simba and GeoSpark consume more storage space (about 2-3X), because they should store both the original data and the index tree in each partition. Simba requires more space consumption than GeoSpark since the temporal dimension is also used to index data. We also evaluate the size of global index in these systems. Global



(a) Time cost (b) RDD storage cost (c) Size of global index

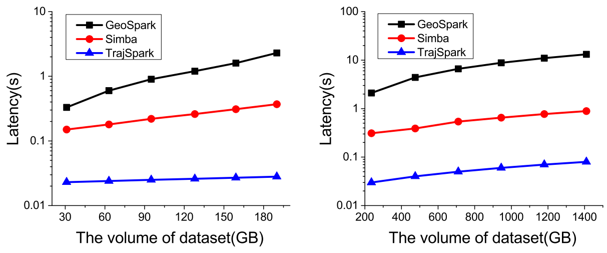
**Fig. 4.** Time and Storage cost for appending data

index is not supported in GeoSpark, so we only show the results of TrajSpark and Simba. Fig. 4(c) indicates that both TrajSpark and Simba have small global index storage overhead (in order of KB). The size of global index in TrajSpark is so small that it can be easily fitted into the memory of the master node. Moreover, there are fewer partitions in TrajSpark due to data compression, so the size of global index in TrajSpark is only about 1/3 that of Simba.

## 7.3 Query Performance

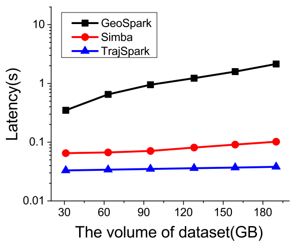
We first examine the efficiency and scalability of SO-based query. The query latency is represented by the average query time of 100 queries which retrieve the whole history of the given MOs. We increase the volume of dataset by loading the daily generated data. Fig. 5 demonstrates that TrajSpark is an order of magnitude faster than Simba, and nearly two orders of magnitude faster than GeoSpark, because GeoSpark needs to scan the whole dataset. Although Simba can prune irrelevant partitions using the global index, it needs to traverse all the content of the selected partitions. In contrast, TrajSpark not only utilizes the global spatio-temporal index to prune partitions but also uses the local hash index to support random access to trajectories. Note that these systems perform better on the real dataset than the synthetic one. This is mainly because the real one can be completely loaded in memory, while only a small part of the synthetic one can be loaded. So queries on the latter dataset require extra I/O cost. Nevertheless, these systems still performs well on the synthetic dataset due to the following reasons: (i) We persist data at the storage level of "MEM_AND_DISK_SER", so hot data can be cached in memory, (ii) By using the global indexes, a huge amount unnecessary I/O costs can be avoided.

Subsequently, we examine the impact of data size for the STR-based query. Since the spatial area is a critical parameter for the query result due to the unbalanced data distribution, we randomly select 100 areas as the spatial constraint for our queries, and each of these areas contains 20*20 cells. These queries only select trajectories generated in the last week. Fig. 6(a) and (b) show the performance of these algorithms. We can see that TrajSpark and Simba behave steady, while the query latency of GeoSpark increases linearly. Without a global index, GeoSpark needs to scan the whole dataset. While TrajSpark and Simba utilize the global index to prune data partitions, and the number of data partitions to be scanned does not vary significantly since the query range has not

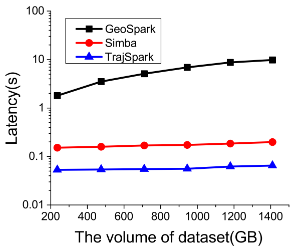


(a) Real (b) Synthetic

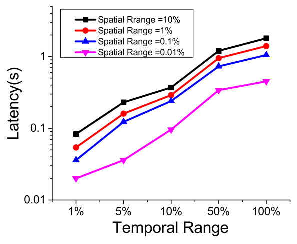
**Fig. 5.** Performance of SO-based query



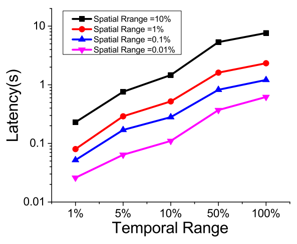
(a) Real



(b) Synthetic



(c) Real



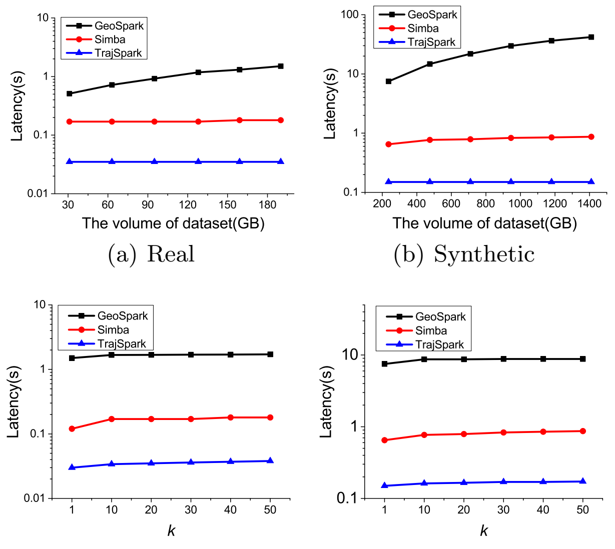
(d) Synthetic

**Fig. 6.** Performance of STR-based query

changed greatly. Moreover, both of the latter two systems use a local index to prune trajectories in each partition. Consequently, TrajSpark and Simba are about an order of magnitude faster than GeoSpark. Moreover, TrajSpark is 3-5 times faster than Simba, because it prunes candidates through the MBR and time range of the trajectory. So it can find the result in $O(\log_n)$ ($n$ is the length of a segment) time as the segment is ordered. Differently, Simba needs to sort points of the MO to restore the original segment which costs $O(n \log_n)$.

Furthermore, we report the performance of our system on STR-based queries under various spatio-temporal ranges. The spatial constraints are 10%, 1%, and 0.1% of the entire region. The temporal constraints are 100%, 50%, 10%, 5% and 1% of the 3 months. As shown in Fig. 6(c) and (d), a large spatial or temporal range usually leads to a longer query latency. But the performance is not essentially linear to the query range, because the number of partitions to be scanned and the amount of data to be accessed in each partition do not grow in a linear way. For example, when the spatial-range is set to 0.01%, and the temporal range increases from 1% to 100%, the query latency grows 30 times. Similarly, when the temporal range is set to 1%, and the spatial range increases from 0.01% to 10%, the query latency grows about 9 times.

Finally, we evaluate the performance of top-$k$ similar trajectory query by using the Euclidean distance as the similarity metric. In this experiment, the trajectories of ten taxis from the same day are selected as the query reference. Fig. 7(a) and (b) shows the scalability of TrajSpark when different amount of data is loaded into the system and the value of $k$ is set to 10. TrajSpark is two orders of magnitude faster than GeoSpark, and runs about 4-6x faster than Simba. That is because TrajSpark and Simba prune data partitions with the global index, while GeoSpark has no global index and needs to access all data



(a) Real (b) Synthetic

(c) Real (d) Synthetic

**Fig. 7.** Performance of KNN-based query

partitions. In comparison of Simba, TrajSpark does not need to sort the points of each trajectory. This result is similar to that of STR-based query because the core of this query is an iterative spatio-temporal query. Furthermore, we evaluate the impact of the parameter $k$ by varying it from 1 to 50. Fig. 7(c) and (d) show that the performance of these systems are not really affected by $k$. This is due to the reason that when $k = 1$, data partitions which contain the most similar result have already contain enough candidates for larger values of $k$.

## 8 Conclusion

To process the massively increasing trajectory data and support near real-time query services, this paper proposes a distributed in-memory system called TrajSpark. This system is built on top of Spark, and proposes IndexTRDD structure that incorporating a global and local indexing mechanism. Additionally, TrajSpark utilizes the time-decaying model to monitor the change of data distribution and enables the data-partition structure to adapt to data changes. We validate the storage overhead, data loading and query latency of TrajSpark by experiments on both real and synthetic datasets. Experimental results show that TrajSpark outperforms existing systems in terms of scalability and efficiency. For future work, we plan to support more complicated operations by utilizing TrajSpark.

**Acknowledgement.** This paper is supported by the National Key Research and Development Program of China (2016YFB1000905), NSFC (61370101, 61532021, U1501252, U1401256 and 61402180), Shanghai Knowledge Service Platform Project (No. ZF1213).

## References

1. Aly, A.M., Mahmood, A.R., Hassan, M.S., Aref, W.G., Ouzzani, M., Elmeleegy, H., Qadah, T.: AQWA: adaptive query-workload-aware partitioning of big spatial data. PVLDB 8(13), 2062–2073 (2015)
2. Botea, V., Mallett, D., Nascimento, M.A., Sander, J.: PIST: an efficient and practical indexing technique for historical spatio-temporal point data. GeoInformatica 12(2), 143–168 (2008)
3. Chakka, V.P., Everspaugh, A.C., Patel, J.M.: Indexing large trajectory data sets with seti. vol. 1001, p. 12. Citeseer (2003)
4. Cudré-Mauroux, P., Wu, E., Madden, S.: Trajstore: An adaptive storage system for very large trajectory data sets. In: ICDE. pp. 109–120 (2010)
5. Eldawy, A., Mokbel, M.F.: Spatialhadoop: A mapreduce framework for spatial data. In: ICDE. pp. 1352–1363 (2015)
6. Huang, S., Wang, B., Zhu, J., Wang, G., Yu, G.: R-hbase: A multi-dimensional indexing framework for cloud computing environment. In: ICDM. pp. 569–574 (2014)
7. Hughes, J.N., Annex, A., Eichelberger, C.N., Fox, A., Hulbert, A., Ronquest, M.: Geomesa: a distributed architecture for spatio-temporal fusion. In: SPIE Defense+ Security. pp. 94730F–94730F (2015)
8. Lange, R., Dürr, F., Rothermel, K.: Scalable processing of trajectory-based queries in space-partitioned moving objects databases. In: SIGSPATIAL. p. 31 (2008)
9. Liu, H., Jin, C., Zhou, A.: Popular route planning with travel cost estimation. In: DASFAA. pp. 403–418 (2016)
10. Ma, Q., Yang, B., Qian, W., Zhou, A.: Query processing of massive trajectory data based on mapreduce. In: CIKM. pp. 9–16 (2009)
11. Nishimura, S., Das, S., Agrawal, D., El Abbadi, A.: MD-hbase: design and implementation of an elastic data infrastructure for cloud-scale location services. DPD 31(2), 289–319 (2013)
12. Österreicher, F., Vajda, I.: A new class of metric divergences on probability spaces and its applicability in statistics. AISM 55(3), 639–653 (2003)
13. Tan, H., Luo, W., Ni, L.M.: Clost: a hadoop-based storage system for big spatio-temporal data analytics. In: CIKM. pp. 2139–2143 (2012)
14. Tang, M., Yu, Y., Malluhi, Q.M., Ouzzani, M., Aref, W.G.: Locationspark: a distributed in-memory data management system for big spatial data. PVLDB 9(13), 1565–1568 (2016)
15. Tzoumas, K., Yiu, M.L., Jensen, C.S.: Oceanst: a distributed analytic system for large-scale spatiotemporal mobile broadband data. PVLDB 7(1561-1564) (2014)
16. Wang, H., Zheng, K., Zhou, X., Sadiq, S.W.: Sharkdb: An in-memory storage system for massive trajectory data. In: SIGMOD. pp. 1099–1104 (2015)
17. Xie, D., Li, F., Yao, B., Li, G., Zhou, L., Guo, M.: Simba: Efficient in-memory spatial analytics. In: SIGMOD. pp. 1071–1085 (2016)
18. Xie, X., Mei, B., Chen, J., Du, X., Jensen, C.S.: Elite: an elastic infrastructure for big spatiotemporal trajectories. VLDB J. 25(4), 473–493 (2016)
19. You, S., Zhang, J., Gruenwald, L.: Large-scale spatial join query processing in cloud. In: ICDE Workshops. pp. 34–41 (2015)
20. Yu, J., Wu, J., Sarwat, M.: Geospark: a cluster computing framework for processing large-scale spatial data. In: SIGSPATIAL. pp. 70:1–70:4 (2015)