A Universal Large-Scale Trajectory Indexing for Cloud-Based Moving Object Applications

Omar Alqahtani
Department of Computer Science and Engineering
University of Colorado Denver
Denver, USA
e-mail: omar.alqahtani@ucdenver.edu

Tom Altman
Department of Computer Science and Engineering
University of Colorado Denver
Denver, USA
e-mail: tom.altman@ucdenver.edu

***Abstract*—The tremendous upsurge in low-cost geospatial chipsets brings out huge volumes of moving object trajectories, which catalyze a wide range of trajectory-driven applications (e.g., sustainable cities, smart transportation, green routing, intelligent homeland security, etc.). Consequently, there has been an emergence of more divergent queries and increased processing complexity. Instead of developing a query-specific approach for limited applications, we propose a Universal Moving Object Index, a flexible index that is capable of fine-tuning based on the application needs, without any structural modification. Also, we introduce a Light-Weight Hybrid Index for heavily-loaded memory. Besides the ability to support trajectory-driven applications universally, both approaches are designed to be easily adopted by cloud-compatible MapReduce platforms. An extensive empirical study is conducted to validate our approaches and to highlight some critical challenges.**

Keywords–big data; moving objects; distribution algorithms; spatial indexing; Apache Spark.

I. INTRODUCTION

The evolution of Global Positioning System (GPS) with the growth of embedded systems and electronic gadgets generate massive numbers of moving object trajectories. Most of our daily devices (e.g., smartphones, wearable devices, navigation systems, tablets, etc.) are capable of recording our movements. Also, the rise in modern transportation services (e.g., ridesharing, electric-bike renting, car sharing, etc.) has increased the number of these trajectories. Moving object trajectories are used in many applications over a wide range of domains. Transportation services and smart navigation heavily depend on both historical and near-future trajectories, e.g., analyzing a hotspot area and routing based on specific preferences, such as green routing. Historical trajectory is also playing a significant role in planning smart cities by analyzing many trajectory-driven factors related to environmental or economic issues.

Consequently, advanced techniques and large-scale computing platforms have become a necessity to cope with storing and processing vast volumes of big spatial data [1]. MapReduce is an exemplary solution that provides an effective distributed computation framework used by many large-scale data processing platforms, such as Apache Spark [2]. Spark is a general-purpose in-memory computing platform, which is supported by most of the cloud computing systems (e.g., Amazon AWS, Google Cloud Engine, IBM Cloud, Microsoft Azure, Cloudera, etc.).

However, the diversity of applications and the adoption of distributed computation platforms raises new challenges in trajectory processing. One of the natural characteristics of trajectory, and spatial data in general, is spatial skewness, which leads to an imbalance in distribution and computation. Although imbalance distribution can be unfolded by using one of the state-of-the-art indexes, which offer a balanced distribution [3]–[5], the skewness will arise again in the intermediate result of a multistage query. Another skewness form is computation skewness, which occurs because the selective queries are most often related to self-skewed space or trajectory. In most cases, computation skewness affects performance by reducing cluster utilization, i.e., creating hotspots within each cluster. Moreover, a space-splitting distribution works fine for simple space-based queries. However, the communication cost for object-based or sophisticated (multistage) space-based queries represents a performance bottleneck. One of the critical challenges is how to take advantage of the spatial and object localities.

To overcome the aforementioned challenges, we propose a Universal Moving Object index (*UMOi*) for in-memory processing of large-scale historical trajectories. *UMOi* is a universal approach that is capable of leveraging two different thoughts of trajectory partitioning (i.e., space-based and object-based partitioning) to preserve spatial and object localities together. The goal of *UMOi* is to satisfy various query types by providing a variable locality mechanism, which boosts *UMOi*'s flexibility to be suitable for a wide range of applications. This advantage makes it more appealing for cloud platforms. Also, we introduce a Light-Weight Hybrid index (*LWHi*), which focuses on object-locality more than spatial-locality during partitioning. *LWHi* provides an ideal computation distribution and guarantees a full *trajectory-preservation* without losing the advantages of spatial pruning. *LWHi* provides a significant performance in heavily-loaded memory situations, i.e., the main memory is saturated with data, which helps reduce the overall cloud cost. The main contributions of this paper are as follows:

- We introduce *UMOi* that is able to control both spatial and object localities.
- We also introduce *LWHi*, which guarantees a full *trajectory-preservation*.
- We formalize and analyze different queries, including continuous spatial queries, and select a representative query for each query type. Next, we develop efficient algorithms for query processing.
- We evaluate our work by conducting extensive per-

 ISBN: 978-1-61208-687-3

formance experiments comparing varies space-based indexing schemes and reveal the implications of computation skewness, intermediate results skewness, and communication.

The rest of the paper is organized as follows: the related work is highlighted in Section II. The structure of the proposed algorithms is discussed in Section III. The query processing approaches are detailed in Section IV. An extensive experimental study is presented in Section V, and a conclusion in Section VI.

II. Related Work

The development and improvement of computing platforms and the demands of new applications create opportunities to adopt new techniques and methods for managing and processing moving object trajectories. From the computing platforms perspective, we classify the prior work into three groups: centralized systems, parallel databases, and MapReduce-based systems. However, we first review some of the access methods and index structures used in most of the related work.

A. Access Methods

R-tree [3] and its variants (e.g., R*-tree [6], R+-tree [7], etc.) are among the most popular access methods which work in a hierarchical manner to group objects in a minimum bounding rectangle. Other types of indexes focus on space-splitting instead of grouping the objects, such as simple grid, k-d-tree [4] and its variants (e.g., k-d-B-Tree [8] and Quad tree [9]). Many versions of the previous structures were adapted for moving object trajectories, which can be grouped into augmented multi-dimensional indexes and multi-version structure indexes. Augmented multi-dimensional indexes can be built using any of the previous indexes, mostly R-trees, with augmentation on the temporal dimension, e.g., spatiotemporal R-tree and Trajectory-Bundle tree (TB-tree) [5]. Spatiotemporal R-tree keeps segments of a trajectory close to each other, while TB-tree ensures that the leaf node only contains segments belong to the same trajectory, i.e., the whole trajectory can be retrieved by linking those leaf nodes together. On the other hand, multi-version indexes, such as Historical R-tree (HR-tree) [10], use, mostly, R-trees to index each timestamp frame. Then, the resulting R-trees are also indexed by using a 1-d index, such as a B-tree. Unchanged nodes from time frame to time frame do not need to be indexed again. Instead, they will be linked to the next R-tree.

B. Centralized Systems and Parallel Database

Reference [11] uses centralized architecture to process a top-k query on activity trajectories, where the points of the trajectory represent some events such as tweeting or posting on Facebook. They use a simple grid to partition the space and some auxiliary indexes to process the events and trajectories. Also, [12] implements a parallel spatial-temporal database to manage both network transportation and trajectory and to support spatiotemporal SQL queries. They use a space-based index that partitions the data based on a space-splitting technique. Any trajectory that crosses a partition boundary is going to be split into sub-trajectories, while any sector of the transportation network that crosses a partition boundary is going to be replicated in all of the crossed partitions.

C. MapReduce-Based Contributions

SpatialHadoop [13] is an extension of Hadoop designed to support spatial data (Point, Line, and Polygon) by including global and local spatial indexes to speed up spatial query processing as range query, k-Nearest Neighbors (k-NN), spatial join, and geometry query [14]. MD-HBase [15] is an extension of HBase; a non-relational Key-Value database that runs on top of Hadoop, which only focuses on spatial point types. The main idea of MD-HBase is to use any multidimensional index and then linearize it to a single dimensional index via the Z-order space-filling technique. Hadoop-GIS [16] extends Hive, a warehouse Hadoop-based database, to process spatial data by using a grid-based global index and an on-demand local index. [17] only focuses on processing the nearest neighbor queries by using a Voronoi-based index. However, none of the previous systems support trajectories directly. PRADASE [18] concentrates on processing trajectories, however, it only covers range queries and trajectory-based queries. It partitions space and time by using a multilevel grid hash index as a global index where no segment cross the partition boundary. Another index is used to hash all segments on all the partitions belonging to a single trajectory to speed up the object retrieving query. Nevertheless, all Hadoop-based contributions inherit the continuous disk access drawback.

GeoSpark [19] is implemented on-top of Spark, and it is identical to SpatialHadoop in terms of indexing and querying. LoctionSpark [20] reduces the impacts of query skewness and network communication overhead. It tracks query frequencies to reveal cluster hotspots and cracks them by re-partitioning. Network communication overhead is reduced by embedded bloom filter technique in the global index, which helps avoid unnecessary communication. SpatialLocation [21] is aimed to process the spatial join through the Spark broadcasting technique and grid index. However, the trajectories are not directly supported by the previous contributions. [22] implements trajectory searching query by using an R-tree as a local and global index. Also, [23] processes top-k similarity query (a trajectory-based query) by using a Voronoi-based index for spatial dimension, where each cell is statically indexed on temporal dimension. Any trajectory that crosses a partition boundary is going to be split, and all segments belong to a trajectory are traced with Trajectory Track Table. SharkDB [24] indexes trajectory only based on time frames in a column-oriented architecture to process range query and k-NN.

Generally, most of the prior work focused on static spatial data (Point, Polygon, and Line), which does not fit moving object trajectories. On the other hand, the works focusing on trajectory rely on spatial or temporal distribution, i.e., data distribution depends on partitioning space and time dimensions. Most often, the resulting distribution would partially preserve spatial and object localities.

Locality represents a key performance, and it might affect the whole system attainments. The nature of a trajectory (i.e., as consecutive time-stamped spatial points) creates contradictory domains, which can be seen in spatial locality and object locality. As a result, some of the previous contributions optimize their systems to contain this contradiction by focusing on spatial locality and spatial queries (e.g., range query, k-NN, etc.) with an object-based auxiliary index, or by narrowing it down to a particular operator and building an ad-hoc index for that purpose only. To the best of our knowledge, there is

 ISBN: 978-1-61208-687-3

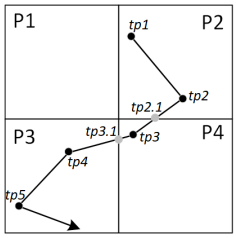


Figure 1. Space-based partitioning.

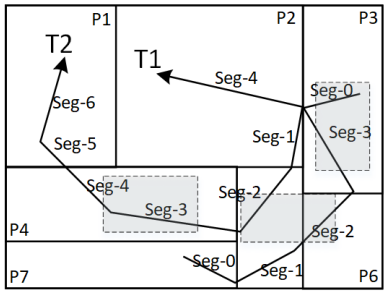


Figure 2. Continuous range query with three shaded regions.

no work that has been conducted on a distributed computation that would balance the losses and gains of both localities and cover a large variety of queries.

III. Trajectory Indexing Overview

In this section, we present our proposed approaches for historical trajectories processing. Before that, we will overview the traditional Space-Based Index (SPI), since it is intensively used by the related works and represents the baseline in our experiments. Based on [25], three phases: Partitioning, Global Index, and Local Index are the most general steps used, but with some differences. In the partitioning phase, a master node partitions the space based on flat or hierarchical indexes. Each partition is assigned to a worker node. When a segment crosses more than one partition, it is divided into several segments as illustrated in Figure 1. As shown, $Seg{<}tp1, tp2{>}$ will be placed in partition 2. However, $Seg{<}tp2, tp3{>}$ needs to be split into two segments where $Seg{<}tp2, tp2.1{>}$ is assigned to partition 2 and $Seg{<}tp2.1, tp3{>}$ is assigned to partition 4.

The global index is also built by the master node based on the partitions that are distributed over the worker nodes. The idea is to speed up query execution by only targeting the partitions that contain the required trajectories instead of searching the whole data. A local index is built and managed by each worker node for each partition to speed up the process of refinements. Similar to the global index, flat or hierarchical indexes might be used for the local index. Some related work, such as [19], use a cost-based model to see if building a local index is worth it.

There are some drawbacks related to space-based partitioning, which increases the performance by pruning the searching space on some types of queries on centralized systems. However, applying the same methodology on a distributed system would result in pruning the searching space, but it would reduce the cluster utilization (i.e., some cluster nodes will be idle), especially when using coarse-grained partitioning. Moreover, since the partitioning is only based on the spatial and temporal dimensions, processing advanced multistage queries (e.g., continuous geo-fencing query) will increase network communication overhead. For example, consider the Continuous Range Query (CRQ) in Figure 2, which we formally define in Section IV-B. It consists of three range queries. The final result should contain all trajectories passing all three ranges. Assuming each cluster node has one partition, $P1$, $P2$, and $P7$ will be idle. The rest will return Trajectory ids (Tids) of any segment crossing any of the three ranges to the master node. The master node needs to process them to find the trajectories pass all the ranges. Processing such a query results in higher communication, or even worse if it exceeds the master memory limit. Another side effect is the reduction in cluster utilization, where some cluster nodes sit idle. This can be improved by using batch queries technique. However, that results in more complicated queries and needs more refinement processing steps to get the final answer, which negatively influences the response time.

A. LWHi Index

LWHi is a light-weight, efficient trajectory index which fuses object-based distribution with spatial index to get the most benefit of them and reduces the impacts of their drawbacks if they were deployed alone. The key performance of *LWHi* is to focus on moving objects instead of spatial properties in the partitioning phase, as illustrated in Figure 3, which forces each trajectory to be located on a single partition. It distributes trajectories based on the Tids. So, $T1$ and $T3$ are settled in partition 1, and $T2$ and $T4$ are settled in partition 2. As a result, *LWHi* ensures full *trajectory-preservation* which results in increasing the moving object's locality while still keeping the advantage of pruning the searching space by employing a local spatial index.

Building LWHi Index: It starts by reading the trajectory dataset as segments of trajectories, where each segment carries its own Segment id (Sid) and Tid. Next, *LWHi* launches a *GroupBy* transformation, which will get all the worker nodes engaged, to group all the segments of one trajectory to reside on one partition. The grouping is based on a hash function on the Tids, which is implemented using a Spark *Partitioner*. The *Partitioner* is responsible for returning the $Partition\_Id$ through a modular hashing when a Tid and the required number of partitions are given. At this point, all the trajectories are distributed among the clusters through the partitions of the Spark RDD. After that, by launching a *MapPartitions* transformation, each worker node will create a local spatial index, using a Sort-Tile-Recursive packed R-tree (*STR-tree*) [26], for its own partitions.

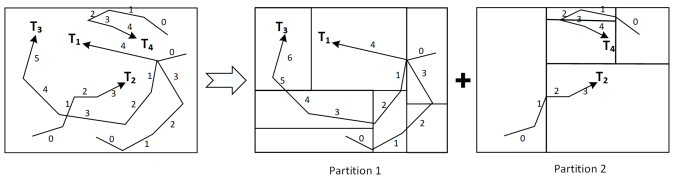


Figure 3. Partitioning and local index of *LWHi*.

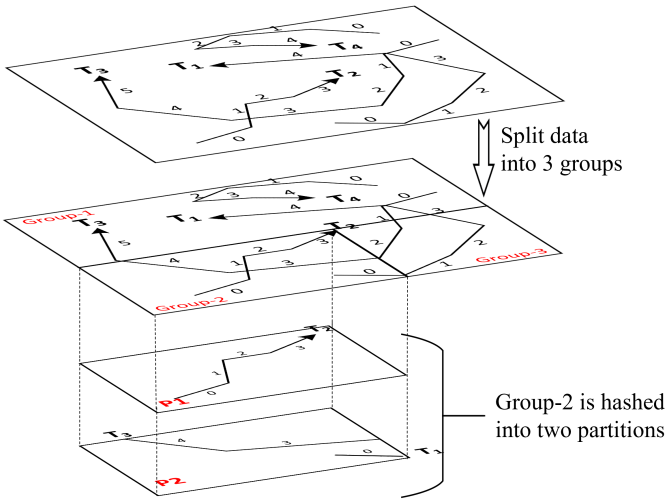


Figure 4. Partitioning phase in *UMOi* with $pd = 2$ and $tpn = 6$.

TABLE I. QUERY TYPES

Category	Query Type	Signature
Space-Based Query	Simple	Trajectory × Spatial → Trajectory
		Trajectory × Spatial → Spatial
	Continuous	Trajectory × Spatial → Trajectory
	Constraint	Trajectory × Spatial → Trajectory
		Trajectory × Spatial → Spatial
Trajectory-Based Query	Similarity	Trajectory × Trajectory → Trajectory
		Trajectory × Trajectory → Boolean
	Aggregation	Trajectory × Trajectory → Trajectory
		Trajectory × Trajectory → Real
	Lookup	Trajectory → Trajectory

B. UMOi Index

The idea of *UMOi* is to have a flexible index which merges space-based and object-based partitioning techniques together. It is capable of balancing both spatial and object localities by providing a locality preservation mechanism, which gives the flexibility to satisfy different applications' demands. The required preservation degree (pd) can be given as an input. The range of pd is $< 1, 2, \cdots, tpn >$, where tpn is the target partitions number (i.e., the required number of partitions to be distributed on the cluster). Consider the trajectory set in Figure 4 with $pd = 2$ and $tpn = 6$. *UMOi* will start by splitting the global space into three spatial groups because $tpn \div pd$ gives the required number of spatial groups. Then, each group will be hashed into two partitions to generate the required six partitions as given. The result is a combination of spatial and object partitioning which cannot provide spatial locality like SPI, or object locality like *LWHi*, but it can instead provide a balance of both localities simultaneously with a pivot that can be adjusted without changing the index structure.

Both SPI and *LWHi* are special cases of *UMOi*. When pd is set to the maximum value (i.e., $pd = tpn$), *UMOi* does not have a space-split step. This is because the required number of spatial groups is $tpn \div pd = 1$, which is the global space itself. So, *UMOi* continues in building the index in the same manner as *LWHi*. On the other hand, when $pd = 1$, *UMOi* needs to split the space into tpn spatial groups. Those groups represent the final partitions, just like SPI. As a result, *UMOi* is capable of universally supporting trajectory-driven applications that depend on space-based or trajectory-based queries by adjusting pd.

Building UMOi Index: the partitioning phase in *UMOi* consists of two steps: space splitting and hashing. Given a trajectory dataset T on space S, *UMOi* starts by generating $ST \subset T$ as a sample set which can fit the master node's memory. Then, on the master node, it builds a binary skeleton tree (*sk-tree*) on ST. This new *sk-tree* is similar to a k-d-B-tree in the way it is constructed, but it is only used to represent the required sub-regions (i.e., the required groups). Even though it would be sufficient to stop building the *sk-tree* after having $tpn - pd$ leaf nodes, *UMOi* continues until we have tpn leaf nodes. We will discuss the reason for this later. After that, *UMOi* merges the regions of pd leaf nodes to form one group. Next, each segment $\in T$ is inserted in the *sk-tree* to be tagged by the correct leaf's code. In case of having a segment that does not fit in a leaf's region, the segment is split into two segments and then reinserted again. Each segment's tag, denoted as Seg_Tag, contains on the leaf's code and its Tid. Finishing partitioning phase, *UMOi* constructs a *Partitioner*, where it is used through *GroupBy* transformation to place each segment to its target partition. *Partitioner* is essential in distribution and query processing. Given a Seg_Tag, *Partitioner* first locates the proper spatial group. Then, using a hashing function on Tid based on pd and a particular group, it returns the required $Partition\_Id$.

UMOi continues building the *sk-tree* to the end, i.e., without stopping at $tpn - pp$ leaf nodes, for two reasons. *UMOi* essentially is used as a proof-of-concept and needs to be dynamic for changing pd without reconstructing the *sk-tree*. More importantly, during the experimental process, it is critical to have a consistent space-split among SPI and *UMOi*s, where *UMOi*s refer to different *UMOi* versions based on the pd values. The different statistical readings of the empirical study require consistency to isolate the impact of the dissimilarity in space-splitting, so they only show the influences of different pd values.

Tracking trajectory has been addressed in the literature, such as [18] [23] [27]. *UMOi* follows the same notion by building a Trajectory Tracking Table (TTT) as a hash table. Each entry in TTT consists of a Tid, as a key, and a list of $Partition_Id$s. It is mainly used when there is a need to retrieve a trajectory (i.e., Lookup query). After tagging the segments, *UMOi* can build TTT by launching a *Job* to locally reduce segments on their Tid for each partition and to combine it with the corresponding $Partition\_Id$. Then, the result is globally reduced on Tid to create a list of different $Partition_Id$s for each trajectory.

IV. QUERY PROCESSING

In this section, we first discuss different query types. After that, we explain how *LWHi* and *UMOi* process different queries.

Our focus is on a query that is formed to ask about the interaction or the relationship between a trajectory and a defined place in the space or between a trajectory and other

trajectories. We can classify the query into two categories: space-based query and trajectory-based query, as seen in Table I. For simplicity, we do not include temporal queries at this early stage because we only need to focus on space and object (with their related quires, localities, and partitioning) without increasing complexity by adding time.

Space-based query can be classified into: Simple, Continuous and Constraint query. Simple spatial query emphasizes the interaction or relationship between a trajectory and a defined spatial type such as region, point or line. The best example of a simple spatial query is the range query, which has been discussed in most previous studies. Continuous spatial query represents a sequence of simple queries. It comes as a result of the nature of a trajectory and the demands of modern applications by requiring queries that identify certain trajectories based on their movements. For example, agencies might be interested to know who might have been in three suspected areas which can be directly addressed by a continuous range query, as seen in Figure 2. Another example involving continuous spatial query would be a query that is used to understand traffic flow by showing long-distance commuters, short-distance commuters, and visitors in one query. The third group in space-based query, called constraint query, represents the spatial queries that are constrained by a defined function (e.g., Euclidean distance, Counting, Maximum, etc.), such as k-NN or top-k.

The second category, trajectory-based query, concentrates on the interaction and relationship between and among trajectories. We divide this group, based on the nature of the queries, into three types: Similarity, Aggregation and Lookup queries as seen in Table I. The first query type, similarity query, depends on a well-defined similarity function, which is mostly used in trajectory data mining, as in [28]–[30]. Aggregation query employs aggregation functions to provide statistical information about trajectories (e.g., longest trajectory) or their different properties (e.g., average speed). Lookup query is the simplest, but it is still an essential query that basically retrieves a particular trajectory by giving its Tid.

In our framework, we concentrate on both space-based and trajectory-based queries to reveal the compliance level of the proposed approaches in different scenarios. A representative query is selected from each query type, except similarity and constraint, since they are structurally similar to the other types. The selected queries are as follows: Range Query, Continuous Range Query, Longest Trajectory, and Lookup Query. Next, we discuss each selected query and how it is processed by *UMOi* and *LWHi*.

A. Range Query

Given a range query $RQ =< P_{bl}, P_{ur} >$, where P_{bl} is the bottom left point of the spatial range and P_{ur} is the upper right point, *UMOi* first determines the involved $Partition_Id$s. It traverses the *sk-tree* to find the required groups that spatially overlap with RQ space and then returns all the engaged groups' partitions. Next, a Spark *Job* is initialized that targets only the required partitions (Spark RDD partitions) based on a given hash set of $Partition_Id$s. During the *Job* execution, the engaged worker node traverses a partition's local index tree (*STR-tree*) based on RQ space. SPI follows the same steps, with a minor difference when traversing *sk-tree*. It returns the engaged $Partition_Id$s directly, since it does not have the groups and their hashed partitions concept. Alternatively, *LWHi* starts a Spark *Job* directly on all $Partition_Id$s. Each worker node traverses its partitions' *STR-trees* based on the given RQ space. The result of all the approaches comes as a new RDD, which only contains the segments covered by RQ.

Algorithm 1: *UMOi*: Processing k-CRQ

```
   Input: k-CRQ
   Output: RDD < Tid >
1  Pid[k]{} ← φ
2  for i ← 1 to k do
3      Pid[i] ← SK-Tree.traverse(RQ_i.space)
4  Transformation MapPartitions(k-CRQ, Pid)
5      R[k][ ] ← φ
6      for i ← 1 to k do
7          if i ∈ Pid[i] then
8              R[i] ← STR-Tree.intersect(RQ_i.space)
               /* It returns all Tids intersect
                  with RQ_i space                    */
9      return R as list of 2-tuple < i, Tid >

   /* By now, all participated worker nodes
      finished MapPartitions and returned R
      lists will be collected in
      RDD < RQ_id, Tid >                             */
10 Transformation GroupBy(RDD < RQ_id, Tid >)
11     return Tid as Key

   /* The result is formed as a
      PairRDD < Tid, [RQ_ids] >                      */
12 Transformation Filter(k, PairRDD < Tid, [RQ_ids] >)
       /* Eliminate duplication [RQ_ids]            */
13     Set U ← [RQ_ids]
14     if U.size = k then
15         return True
16     else
17         return False
18 return RDD < Tid >
```

B. Continuous Range Query

When receiving a k-$CRQ =< RQ_1, RQ_2, \cdots, RQ_k >$ on a trajectory set T, the algorithm needs to return any $Traj \in T$ *s.t.* $Traj_{space} \cap RQ_{space}\ \forall\ RQ \in CRQ$. From Algorithm 1, *UMOi* traverses the *sk-tree* for each RQ_i, where $1 \leq i \leq k$, to determine the required $Partition_Id$s. It returns: an overall set and an array of sets. The overall set contains all the $Partition_Id$s required for all RQs of CRQ. It is used when initializing the Spark *Job*. The second returned item, an array of sets, contains the required $Partition_Id$s as an individual set for each RQ_i. It is used to refine unnecessary *STR-tree* traversal, as shown in line 7. In line 4, *UMOi* identifies any trajectory that intersects with any RQ by using a transformation *MapPartitions*, which is running in parallel by the worker nodes on the given RDD partitions, known as map phase. An array of lists is used by each engaged worker node for each RDD partition to collect the Tids intersect with a RQ and the corresponding RQ_id. The results, coming as lists of 2-tuple of RQ_id and Tid, are then reduced in a new RDD, known as the reduce phase. The new RDD, called RDD_{map}, includes any trajectory overlap with at least one RQ_i. Next, we group all the elements in RDD_{map} by the Tids, as shown in

 ISBN: 978-1-61208-687-3

Algorithm 2: *LWHi*: Processing k-CRQ

```
   Input: k-CRQ
   Output: RDD < Tid >
1  Transformation MapPartitions(k-CRQ)
2      L[ ] ← φ
3      R[k]{ } ← φ
4      for i ← 1 to k do
5          R[i] ← STR-Tree.intersect(RQ_i.space)
           /* It returns all Tids intersect
              with RQ_i space                    */
6      foreach Tid ∈ R[1] do
7          Flag ← True
8          for i ← 2 to k do
9              if Tid ∉ R[i] then
10                 Flag ← False
11                 Break
12         if Flag = True then
13             L ← push(Tid)
14     return L
   /* The result is formed as a RDD < Tid >
      from returned Ls                          */
15 return RDD < Tid >
```

Algorithm 3: *UMOi*: Processing LTQ

```
   Input: LTQ
   Output: Tid_LT
1  Transformation MapPartitions(LTQ)
2      D{ , } ← φ                  /* Dictionary */
3      forall Tid ∈ Parition do
4          Tid_Length ← Compute.Length(Tid)
5          if D contains Tid then
6              D ← push(Tid, OldValue + Tid_length)
7          else
8              D ← push(Tid, Tid_length)
9      return D as a list of 2-tuple
   /* Result is pairRDD < Tid, Tid_length >   */
10 Define Sum(Tid_length1, Tid_length2)
11     return Tid_length1 + Tid_length2
12 Transformation Aggregate(pairRDD < Tid, Tid_length >)
13     apply(Sum) /* Apply Sum function on
                     elements have the same key Tid */
14
15 return Max( pairRDD < Tid, Tid_length > )
```

Algorithm 4: *LWHi*: Processing LTQ

```
   Input: LTQ
   Output: Tid_LT
1  Transformation MapPartitions(LTQ)
2      D{ , } ← φ                  /* Dictionary */
3      forall Tid ∈ Parition do
4          Tid_Length ← Compute.Length(Tid)
5          if D contains Tid then
6              D ← push(Tid, OldValue + Tid_length)
7          else
8              D ← push(Tid, Tid_length)
9      return D as a list of 2-tuple
   /* Result is pairRDD < Tid, Tid_length >   */
10 return Max( pairRDD < Tid, Tid_length > )
```

line 10. After that, it filters any Tid that does not intersect with all $RQ \in CRQ$. SPI, again, follows the same steps exactly except in the *sk-tree* as mentioned before.

Algorithm 2 shows how *LWHi* processes k-CRQ. The master node runs a transformation *MapPartitions* directly on all the worker nodes for all partitions. Through *MapPartitons*, each worker node traverses its local *STR-tree*, for every $RQ \in CRQ$. The result is an array of hash sets that contains overlapped Tids, as shown in line 5. It uses a hash set to eliminate duplication among trajectories of a particular RQ_i and to speed up the searching in the next step. Then, to find all Tids that overlap with k-CRQ, each Tid from the first set (i.e., $Tids \cap RQ_1.space$) is checked for whether it belongs to the other sets. If it does not belong to at least one set, the process on this Tid is stopped and cannot be included in the final result. The returned lists are then formed in an RDD.

In *LWHi*, all intermediate processing is carried out in parallel by the worker nodes locally, without the need to process them globally by launching another *Job* and causing a costly *shuffle*. This is because *LWHi* guarantees full trajectory preservation. On the other hand, *UMOi* and SPI need to conduct a global refinement on the intermediate results (i.e., *GroupBy* and *Filter* in Algorithm 1, lines 10 and 12) which affects the overall performance in many respects, such as communication, cluster utilization, and GC scan. The communication between nodes is obviously going to increase, especially during *GroupBy*. Also, the distribution of RDD partitions after executing *GroupBy* transformation is skewed because of the keys' original places (Tids), which were collected based on k-CRQ. The degree varies based on the number of involved partitions in solving k-CRQ, so SPI would have the worst case. The intermediate result skewness affects any further computations (e.g., *Filter*, *Count*, etc.) and, therefore, reduces the cluster utilization. In some cases, the skewness with the previous consecutive computation on certain worker nodes could cause cumulative stress and a GC's full scan at the end. However, the overall influence fluctuates based on the different pd values, which reflect the trajectory's preservation degree.

C. Longest Trajectory Query

This query belongs to the aggregation query type, which depends on a well-defined aggregation function. Given a longest trajectory query LTQ on T, it needs a Tid s.t. $Tid_{length} \geq \forall\ Tid_{length} \in T$. Algorithm 3 shows how *UMOi* processes LTQ. It first executes a local reduction, lines 1–8, and then global reduction by using *Aggregation* transformation, line 11. *Aggregation* transformation reduces the elements of pairRDD on their Tid keys by using an aggregation function, as in line 9, and causes a data shuffle. The element with the maximum value is then returned as the longest trajectory. The same steps are used in SPI. *LWHi*, as shown in Algorithm 4, does not require a global reduction since the whole trajectory resides in one partition.

Aggregation transformation actually does a local aggregation on partitions and then a global aggregation on the results. However, in our case, we need to compute the length of different sub-trajectories, and our partitions are built in a way that is hard for a passed function to deal with. So, we implement the first part in both algorithms to compute the

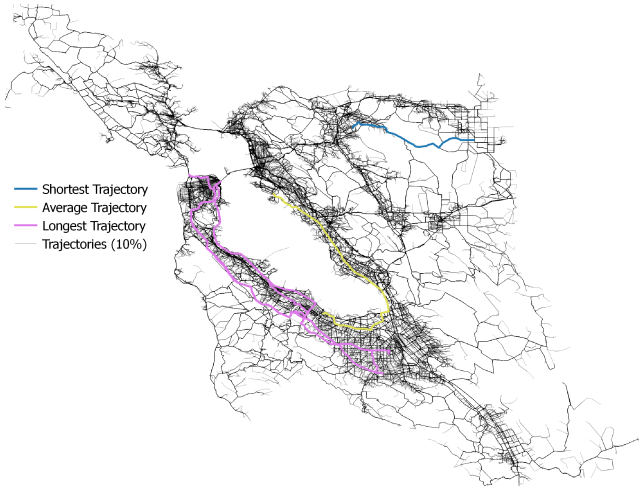


Figure 5. Trajectory dataset.

length of the trajectory and process the local aggregation at the same time.

D. Lookup Query

UMOi uses TTT to identify the required partitions. Then, it uses a *MapPartitions* transformation to retrieve segments of the given Tid. The same procedure is followed in SPI. In *LWHi*, identifying the required partition is fairly simple. It uses the same hash function used by the *Partitioner* to find the required $Partition\_Id$. Then, similar to *UMOi*, it retrieves the segments of the given Tid from the required partition.

V. Experimental Study

In this section, we discuss the evaluation of our proposed approaches *LWHi* and *UMOi* and compare them with SPI. We present an assessment for trajectory skewness and its impact on spatial and object localities. From the performance perspective, we conduct extensive experiments to evaluate different query types.

A. Experiment Setting

Our implementation uses Apache Spark 2.2.0 with Java 1.8. We adopt Java *ParallelOldGC* as a garbage collector and *Kryo* for serialization. The experiments are conducted on AWS EMR 5.9.0. We use six m3.xlarge instances, where each instance provides 4 vCPU (Intel Xeon E5-2670) and 15 GB of RAM with high network performance. From Spark perspective, the master node (driver) is using one instance with 8 threads (cores) and 10.22 GB. The worker nodes (executors) are using 5 instances, each of which is using 6 threads (cores) and 10.22 GB. Thus, the total worker threads are 30, and they are distributed over 5 instances.

From the data side, we use the well-known moving object generator [31]. The trajectory set consists of more than 119 million trajectory segments (140,000 trajectories) over San Francisco, as seen in Figure 5. Since we are only focusing on in-memory computation, the data is always cached to the main memory during experiments.

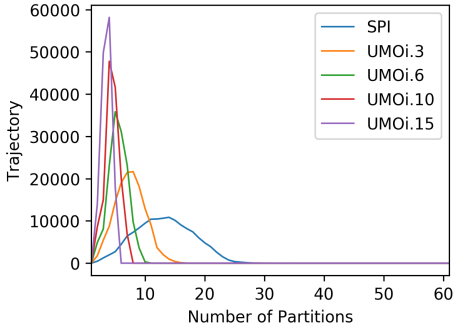


Figure 6. Trajectories settling frequency.

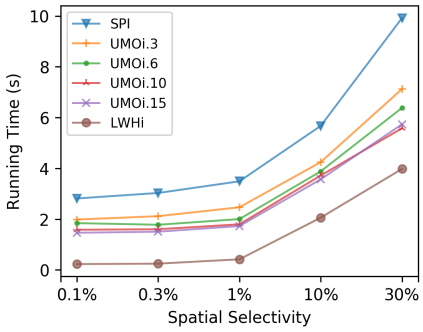
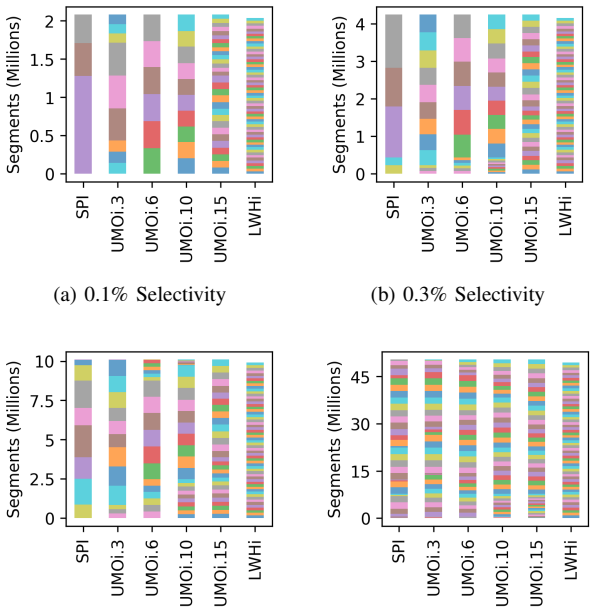


Figure 7. Running time for range query.



(a) 0.1% Selectivity (b) 0.3% Selectivity

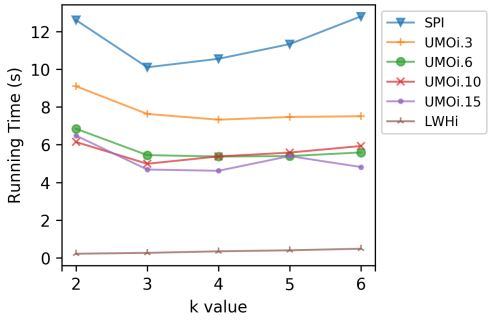
(c) 1% Selectivity (d) 10% Selectivity

Figure 8. Engaged partitions for a particular range query with different spatial selectivity.

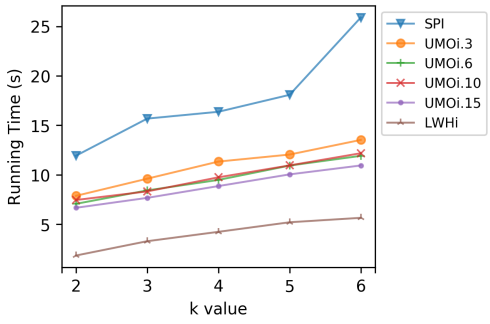
B. Skewness

We are more interested in analyzing the effects of eradicating the skewness rather than the skewness itself. Figure 6 shows trajectory occupancy on 60 partitions, i.e., the frequency

 ISBN: 978-1-61208-687-3



(a) 0.3% Spatial Selectivity



(b) 10% Spatial Selectivity

Figure 9. Running time for continuous range query.

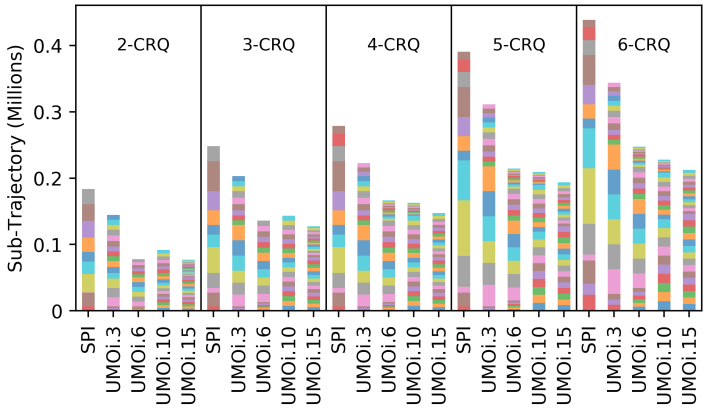
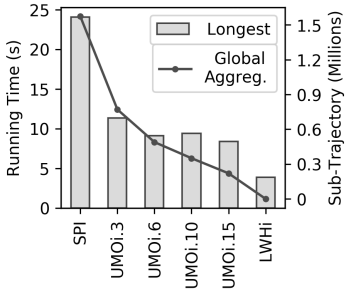
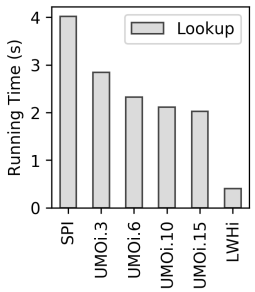


Figure 10. Engaged partitions for a particular CRQ with different k values.



(a) Longest Query



(b) Lookup Query

Figure 11. Running time for longest and lookup queries.

of required partitions to hold a particular trajectory, and also reflects the trajectory preservation for each method. *UMOi.x* means *UMOi* with $pd = x$. We do not include *LWHi*, since it only shows that all trajectories need one partition. The big impact is on SPI, where most trajectories need from 5 to 20 partitions. With increasing pd in *UMOi*, the required partitions numbers decrease which means more trajectory preservation. However, the decrease slows down after *UMOi.3*. The influences of trajectory segmentation on each query type are discussed further in the next section.

C. Construction of the Indexes

The average time to construct *UMOi* is 223.6 seconds, while it only takes 75.1 seconds for *LWHi* and 205 seconds for SPI. It is expected that *UMOi* takes longer since it needs to conduct space-based and object-based distributions. However, *UMOi* merges both distributions into one Spark *Job*, just like SPI, and this is why the difference is not significant.

D. Performance Evaluation

We test *LWHi* and *UMOi* and compare them with SPI on the following quires: Range Query, Continuous Range Query, Longest Trajectory (aggregation query), and Lookup Query. We set tpn to 60 and pd to $3, 6, 10$, and 15.

Starting with range query, Figure 7 shows the average running time of 100 random range queries. The running times of all methods increase with larger spatial selectivity. *LWHi* is better than all other algorithms, while *UMOi.3* outperforms SPI by a factor 1.4x on average. *UMOi.6* and *UMOi.10* outperform SPI only by a factor of 1.6x and 1.7x, respectively. Figure 8 shows the required partitions for only one range query, where each involved partition is uniquely colored, and the size of the colored partition reflects the amount of processed segments. The best case is when we have a significant number of engaged partitions, each of which participates equally while processing the same amount of segments. One of the performance factors is the GC's full-scan, which might be triggered when a big chunk of processed segments is settled on one node. Another observation is that increasing pd value does not necessarily mean more involved partitions, as we see in Figure 8a and 8b, where *UMOi.3* has more involved partitions than *UMOi.6*.

For continuous range query, we run 100 random queries with $k = 2, 3, 4, 5$, and 6, and we use two spatial selectivity (0.3% and 10%) as seen in Figure 9. In general, *LWHi* shows a significant speedup, especially with small spatial selectivity, and that is so because it only needs one Spark *Job* to execute the query locally, without any communication overhead. With small selectivity, *LWHi* outperforms SPI by a factor of 33x on average, and by a factor of 4.3x with 10% selectivity. *UMOi*s (*UMOi.3*, *UMOi.6*, etc.) gain a speedup range from 1.5x to 2.1x. Figure 10 gives an important glance at many factors during a particular k-CRQ's execution with different k values. The length of the bars represents the sub-trajectories after the first Spark *Job*, which is responsible for the local computation. They also reveal the remaining amount of global execution. If we take only one k-CRQ, we can see the difference in communication needed by the *GroupBy Job*. In 2-CRQ and 3-CRQ, *UMOi.6* needs less global reduction than *UMOi.10* because of the query location. The second *Job* performance

 ISBN: 978-1-61208-687-3

depends on the location, size, and number of the partitions resulting from the first *Job*, which are uniquely colored in Figure 10. In this case, *UMOi*s have better situations than SPI which leads to more parallelism with a reasonable partition size. More importantly, with fewer resulting partitions, a cluster tends to constrain computation on fewer nodes, which affects cluster utilization and causes a GC's full-scan.

Figure 11a shows the average running time to find the longest trajectory. *LWHi* outperforms SPI by a factor 6.2x. The speedup factors for *UMOi*s range from 2.1x to 2.9x compared to SPI. It also shows the amount of sub-trajectories after the local aggregation that need to be processed globally. It reveals the tremendous difference between SPI and *UMOi.3* in trajectory preservation and how it slows down with higher pd, which reflects in the performance of each index. Also, Figure 11b shows the average running time of lookup query. *LWHi* gives the highest speedup by a factor of 10x, since it does not need a secondary index, such as TTT, and it only has to process one partition.

All the experimental results show that *LWHi* outperforms *UMOi*. However, *UMOi* is more useful in some cases when spatial locality is required. For example, consider applications that require special datasets such that the trajectories are mixed with static spatial data (e.g., buildings, road-network, etc.). With *LWHi*, all the partitions share the same global space. So, all the static spatial data need to be copied to all the partitions of *LWHi*, which results in full redundancy. SPI depends on space-based partitioning which is also suitable for static spatial data, similar to [12], and will have the lowest redundancy. With *UMOi*, the redundancy depends on the value of pd. So, the application user will be able to control the trade-off between performance and redundancy.

E. Limitations

Even though *UMOi* and *LWHi* show a significant performance improvement over traditional techniques, both have some limitations. The first challenge is how to select the optimal pd for a particular application. The optimal value for a pd depends on the nature of the application's queries, the characteristics of the trajectories, and the cluster settings. *UMOi* could be extended to contain a small simulation engine to give the best value for a pd based on a sample from the queries and trajectories. However, that will reflect on the construction time which is already higher than SPI and *LWHi*.

Moreover, both *UMOi* and *LWHi* are designed for in-memory usage. However, Spark also supports partially in-memory computation, which is useful when the data exceeds the main memory limits (i.e., usually 30% of the data reside on the disk). In this case, *LWHi* will always suffer from disk I/O for space-based queries, while it depends on the location of the engaged partitions for SPI and *UMOi*.

VI. Conclusion and Outlook

The huge volumes of moving object trajectories catalyze more trajectory-driven applications with more space-based and trajectory-based queries. As a result, cloud computing platforms are the typical solution to cope with the large-scale data and applications' demands. Spark has been adopted by most of the cloud platforms, and it offers an in-memory distributed computation platform. However, the large-scale trajectories and the adoption of a distributed platform raise the following challenges: communication cost, computation skewness, intermediate results skewness, and GC scan.

Therefore, our goal is to develop a large-scale historical trajectory index to support in-memory processing for both space-based and trajectory-based query types. Also, it needs to overcome all the previous challenges. As a result, we propose *UMOi* as a universal index that is capable of representing different partitioning techniques (i.e., space-based partitioning and object-based partitioning). It provides a flexible preservation degree (pd) parameter to control both spatial and object localities making it suitable to accommodate divergent trajectory-driven applications. With the lowest pd value, *UMOi* will act just like the traditional space-based index. However, with the highest pd value, it will act as our second index, *LWHi*. We distinguish *LWHi* as a standalone index because it guarantees a full *trajectory-preservation*, which allows optimizations to take place on both index construction and query processing.

We also conduct extensive experiments to validate our approaches. The results show a significant performance improvement (on both space-based and trajectory-based query types) compared to space-based indexing. The significant speedup is a result of reducing the communication cost and increasing the cluster utilization. Also, we present an analysis for heavy-loaded memory to show the far-reaching implications, such as GC scan and intermediate results skewness.

For future work, several optimizations and extensions could be considered. Both approaches could be extended to include partially in-memory data processing, which is also provided by Spark. Also, the space-splitting could be enhanced to maximize trajectory preservation by adopting an object-aware spatial partitioning. Finally, it is important to consider nested queries, i.e., queries that consist of different query types, and to analyze how they would benefit from different preservation degrees.

References

[1] D. Abadi et al., "The beckman report on database research," Commun. ACM, vol. 59, no. 2, Jan. 2016, pp. 92–99. [Online]. Available: http://doi.acm.org/10.1145/2845915

[2] https://spark.apache.org/, [retrieved: January, 2019].

[3] A. Guttman, "R-trees: A dynamic index structure for spatial searching," SIGMOD Rec., vol. 14, no. 2, Jun. 1984, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/971697.602266

[4] J. L. Bentley, "Multidimensional binary search trees in database applications," IEEE Transactions on Software Engineering, vol. SE-5, no. 4, July 1979, pp. 333–340.

[5] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in Proceedings of 26th International Conference on Very Large Data Bases, ser. VLDB 2000, Sep. 2000, pp. 395–406.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," SIGMOD Rec., vol. 19, no. 2, May 1990, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/93605.98741

[7] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects," in Proceedings of the 13th International Conference on Very Large Data Bases, ser. VLDB '87, 1987, pp. 507–518.

[8] J. T. Robinson, "The k-d-b-tree: A search structure for large multidimensional dynamic indexes," in Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '81. ACM, 1981, pp. 10–18. [Online]. Available: http://doi.acm.org/10.1145/582318.582321

[9] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," Acta Informatica, vol. 4, no. 1, Mar. 1974, pp. 1–9.

 ISBN: 978-1-61208-687-3

[10] M. A. Nascimento and J. R. O. Silva, "Towards historical R-trees," in Proceedings of the 1998 ACM Symposium on Applied Computing, ser. SAC '98. ACM, 1998, pp. 235–240. [Online]. Available: http://doi.acm.org/10.1145/330560.330692

[11] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang, "Towards efficient search for activity trajectories," in 2013 IEEE 29th International Conference on Data Engineering (ICDE), April 2013, pp. 230–241.

[12] Z. Ding, B. Yang, Y. Chi, and L. Guo, "Enabling smart transportation systems: A parallel spatio-temporal database approach," IEEE Transactions on Computers, vol. 65, no. 5, May 2016, pp. 1377–1391.

[13] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in 2015 IEEE 31st International Conference on Data Engineering, April 2015, pp. 1352–1363.

[14] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan, "CG_hadoop: Computational geometry in mapreduce," in Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ser. SIGSPATIAL'13. ACM, 2013, pp. 294–303. [Online]. Available: http://doi.acm.org/10.1145/2525314.2525349

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

[16] A. Aji et al., "Hadoop gis: A high performance spatial data warehousing system over mapreduce," Proc. VLDB Endow., vol. 6, no. 11, Aug. 2013, pp. 1009–1020. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536227

[17] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based geospatial query processing with mapreduce," in 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Nov. 2010, pp. 9–16.

[18] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on mapreduce," in Proceedings of the First International Workshop on Cloud Data Management, ser. CloudDB '09. ACM, 2009, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1651263.1651266

[19] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, ser. SIGSPATIAL '15. ACM, 2015, pp. 70:1–70:4. [Online]. Available: http://doi.acm.org/10.1145/2820783.2820860

[20] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," Proc. VLDB Endow., vol. 9, no. 13, Sep. 2016, pp. 1565–1568. [Online]. Available: https://doi.org/10.14778/3007263.3007310

[21] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in 2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW), April 2015, pp. 34–41. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICDEW.2015.7129541

[22] H. Wang and A. Belhassena, "Parallel trajectory search based on distributed index," Information Sciences, vol. 388-389, 2017, pp. 62 – 83. [Online]. Available: https://doi.org/10.1016/j.ins.2017.01.016

[23] D. A. Peixoto and N. Q. V. Hung, "Scalable and fast top-k most similar trajectories search using mapreduce in-memory," in Databases Theory and Applications. Springer International Publishing, 2016, pp. 228–241.

[24] H. Wang et al., "Sharkdb: An in-memory column-oriented trajectory storage," in Proceedings of the 23rd ACM International Conference on Information and Knowledge Management, ser. CIKM '14. ACM, 2014, pp. 1409–1418. [Online]. Available: http://doi.acm.org/10.1145/2661829.2661878

[25] A. Eldawy and M. F. Mokbel, "The era of big spatial data: A survey," Information and Media Technologies, vol. 10, no. 2, 2015, pp. 305–316.

[26] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in Proceedings 13th International Conference on Data Engineering, April 1997, pp. 497–506.

[27] F. Chang et al., "Bigtable: A distributed storage system for structured data," ACM Trans. Comput. Syst., vol. 26, no. 2, Jun. 2008, pp. 4:1–4:26. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[28] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie, "Searching trajectories by locations: An efficiency study," in Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '10. ACM, 2010, pp. 255–266. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807197

[29] B.-K. Yi, H. V. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in Proceedings 14th International Conference on Data Engineering, Feb. 1998, pp. 201–208.

[30] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity search in sequence databases," in International conference on foundations of data organization and algorithms. Springer, 1993, pp. 69–84.

[31] T. Brinkhoff, "A framework for generating network-based moving objects," GeoInformatica, vol. 6, no. 2, June 2002, pp. 153–180. [Online]. Available: https://doi.org/10.1023/A:1015231126594

 ISBN: 978-1-61208-687-3