Cloud-based Framework for Spatio-Temporal Trajectory Data Segmentation and Query

Huaqiang Kang, Yan Liu, *Member, IEEE,* Weishan Zhang, *Member, IEEE*

Abstract—Trajectory segmentation is a technique of dividing sequential trajectory into segments. These segments are building blocks to various applications. Hence a system framework is essential to support trajectory segment indexing, storage, and query. When the size of segments is beyond the computing capacity of a single processing node, a distributed solution is proposed. In this paper, we develop a distributed trajectory segmentation framework that includes a greedy-split segmentation method. This framework consists of distributed in-memory processing and a cluster of graph storage respectively. For fast trajectory queries, we design a distributed spatial R-tree index of trajectory segments. Using the indexes, we build queries of segments from in-memory processing and the graph storage. Based on this framework, we define two metrics to measure trajectory similarity and chance of collision. These two metrics are further applied to identify moving groups of trajectories. We quantitatively evaluate the effects of data partition, parallelism, and data size on the system. We identify the bottleneck factors at the data partition stage and validate two mitigation techniques to data skew. The evaluation demonstrates our distributed segmentation method and the system framework scale as the growth of the workload and the size of the parallel cluster.

Index Terms—Trajectory data, segmentation, distributed computing, parallel query.

1 Introduction

With the fast development in micro-electronics, increasing location tracking equipment is facilitated in transportation, personal health, and public security fields. GPS tracking systems have been applied to collecting data of persons, devices, vehicles to discover a person's behavior, track a vehicle's route, and produce diverse applications including point of interests, recommendation, health monitoring safety regulation, and fleet management.

These raw data are usually recorded as spatio-temporal points. For example, an IMU sensor [1] applied to automotive car produces ten records per second, with each record of at least 20 Bytes. For every hour each car on the road, the IMU sensor generates 36,000 records with the size of over 700MB [2]. As the scale of managed cars expands dramatically, the trajectory data become a source of Big Data. Therefore the processing of trajectory data inherits Big Data challenges. One challenge is to develop efficient queries to provide insightful views on the trajectory data. When the data size is beyond the capacity of a single processing node, the trajectory data are partitioned to distributed nodes. Partitioned data are then analyzed in parallel.

There are different methods of partition. A simple partition is dividing the trajectory data evenly to each node. Each node has entire sequences of individual trajectories. In scenarios of trajectory classification and clustering, a trajectory is further partitioned into segments on each node, referred as *Trajectory Segmentation*. These segments are stored and queried for certain analysis. In this paper, we focus on three research questions of distributed parallel processing of trajectory segmentation.

- R1: How to partition and segment trajectories in a distributed framework ?
- R2: What are the key factors of parallelism that affect the scalability of trajectory segmentation and queries ?
- R3: How to evaluate the performance gain in parallel ?

In this paper, we present a parallel trajectory segmentation framework that scales horizontally as the size of the trajectory data, the load of queries and the number of worker nodes grow. This framework consists of a greedy-split algorithm for parallel trajectory partition and workflows for queries of segments for trajectory analysis. This workflow is implemented to investigate the parallelism factors of scalability using two architectures, one is distributed in-memory processing, and the other is NoSQL graph storage. We build spatial indexes and query operations in each architecture.

To realize this workflow, we further define a data model for representing trajectory segments and associated geometric operations. This data model is then implemented using Resilient Distributed Datasets (RDDs) of Apache Spark [3]. Consequently, the in-memory storage and queries of trajectory segments are built on a spatial processing framework, called GeoSpark [4]; and other geometric objects are stored in a NoSQL graph database, called Neo4j. In Neo4j, queries of trajectory segments are database transactions.

To demonstrate the usage of our proposed framework, we define two metrics that are integral to applications such as trajectory clustering analysis. One metrics estimates the similarity of trajectories. We define a threshold of Euclidean distances of trajectories to count any moving objects are within this threshold at a certain period. The other metrics

- *H. Kang and Y. Liu are with the Faculty of Engineering and Computer Science, Concordia University, Montreal, QC Canada, H3G 1M8. E-mail: hu_kan@encs.concordia.ca, yan.liu@concordia.ca*
- *W. Zhang is with School of Computer and Communication Engineering China University of Petroleum, China. E-mail: zhangws@upc.edu.cn*

Manuscript received April 19, 2005; revised August 26, 2015.

detect the collision chance by measuring the intersections of two trajectories.

We evaluate our method by two means, (1) system level performance evaluation and (2) comparison of results from our trajectory clustering workflow with another clustering method [5]. For the system level performance evaluation, we deploy our workflows on the Amazon Elastic MapReduce (EMR) platform. The trajectory data are from Microsoft GeoLife [6] with the size of 1.6GB and 24 million moving object records. Our experiments show the performance on In-memory framework has an improvement speedup ranging from 2 to 4.6 with 8 node cluster or 16 node cluster compared with the single node system. The evaluation on the graph storage framework has a maximum speedup of 17.5 improvement when expanding the cluster size from 1 node to 16. In term of accuracy of clustering results, we reach 87.2% compared to GPFinder [5] as ground truth.

Through this research study, we identify the bottleneck is data skew due to the geographic imbalance of dataset. We adopt a dynamic partitioning method to adjust each partition's load of objects. We further device a data skew mitigation solution by involving other non-geographical property as the secondary key when distributing data.

The main contribution of our paper is four-fold. Our implementation is accessible from Github [7].

1) We design an algorithm of parallel segmentation of trajectories in a distributed system;
2) We develop a system workflow that queries trajectories segments in-memory and in a graph database. We integrate indexing techniques for a fast query;
3) We define metrics that are further utilized applications such as trajectory clustering analysis;
4) We develop a data skew migration solution to balance the workload as both the size of data and the computing cluster grow.

The paper follows this structure: Section 2 introduces the related works of trajectory segmentation, storage and accessing. Section 3 is our framework system architecture. Section 4 presents our segmentation algorithm in the Spark framework. Section 5 demonstrates our trajectory indexing workflow. Section 6 reveals the query workflow. Section 7 explains our two trajectory evaluation metrics. Section 8 is an application based on above two metrics. Section 9 shows our experiment results and Section 10 discusses our data skewness for improvement.

2 Related Works

2.1 Algorithms and Queries

Trajectories need to be simplified by cutting into smaller, less complex primitives. Anagnostopoulos et al. [8] illustrate a method of segmentation that adapts to Nearest-Neighbor search and analyzes the segmentation problem in a global view. Cudre-Mauroux et al. [9] give a solution for the large size of trajectories on disk. They maintain an optimal index and the data are dynamically co-located. To reduce I/O, the system also adapts to queries for optimization. Mokbel et al. [10] show us three major spatio-temporal access methods, namely "Indexing the Past", "Indexing the Current Positions" and "Indexing the Current and Future Positions". The most used one is indexing the past positions, such as STR-tree or RT-tree [11]. Another method commonly used is to index the current positions, such as LUR-tree [12] and Hashing. The third method of indexing is to index the current and future positions, including PMR-quadtree [13] and SV-Model [14]. A new trend of accessing trajectories is using parametric rectangles [14]. They do not enclosure the trajectories directly. Instead, they create the bounding rectangles as a function of time. The moving objects are in the same rectangles for a time instance t.

For motion classification, Fu et al. propose a similarity-based pattern grouping method compared with fuzzy K-means [15]. Giannotti et al. [16] also present two purely temporal trajectory pattern mining approaches. They firstly transform the sequences of points into regions of interest. Then they use origin-destination matrices to find out pre-conceived regions or use a dense based discretization method to find out the popular regions. Panagiotakis et al. [17] introduce a methodology to find out the most representative sub-trajectories. They represent the trajectories based on multiple attributes and then sample them without supervision.

2.2 Spatial Data/ Storage

Most geological data are based on geometry. The most common method is to use traditional RDBMS as column wise store. For the geographical data, there are two ways for storage, raster-based and vector-based [18]. The object is represented as a series of lines connected to form a polygon. The RDBMS can easily store the coordinates of vertices. In a raster-based system, a real world object is formatted by cells and represented as a series of contiguous cells. It is popular to use geometry database to store this geological information. The vector-based system uses one x,y coordinate pair to represent a single point, a line is made up of a series of coordinate pairs. An area further consists of multiple lines to form a polygon. DISASTER [19] is a Portuguese GIS database based on the most popular open source MySQL database engine. It stores floods and landslides for the period of the year 1865 to the year 2010. In [20], they have developed mechanisms to integrate multiple data sources and finally a seamless database is achieved. A data warehouse supporting streaming data is designed [21] by Orlando et al. The data warehouse supports trajectory properties such as average velocity, maximum acceleration as well as aggregation operations.

2.3 Processing Framework

Simba [22] is a new distributed spatial processing framework. Most of its operations are based on native Spark APIs. It extends the query features with the support of SQL statements. This framework does not support permanent data persistence. SpatialHadoop [23] is another distributed processing framework based on Hadoop. Therefore, all the intermediate data are stored in file systems. So the processing speed is I/O bounded. LocationSpark [24] and Magellan [25] are both spatial data processing extensions based on Spark. They provide multiple partition techniques, indexes and multiple query methods such as range query, KNN or spatial join.

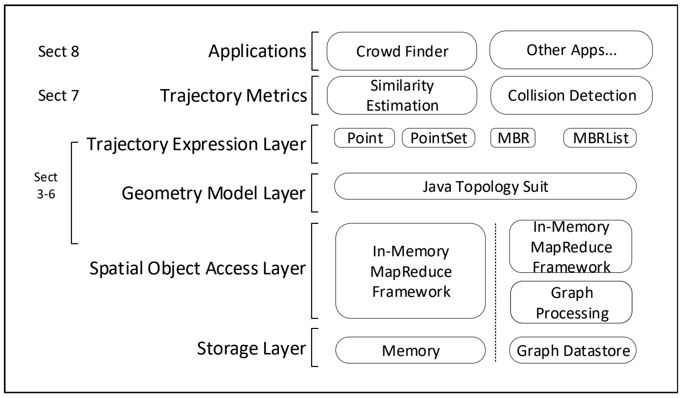


Fig. 1. Overview of Framework Architecture

3 The Framework Architecture

Our framework architecture follows a layered architecture to support metrics calculation, topology modeling, parallel distributive processing, query and storage. We use vector-based expression method to represent our trajectory. The greedy-split algorithm makes a long trajectory is segmented into several parts for further R-tree indexing. We apply Hadoop based processing framework to partition our dataset into multiple parts. The query is fulfilled in a NoSQL database or in-memory processing framework. The architecture is shown in Fig. 1.

The first layer is the *Application Layer*. All applications utilize trajectory metrics are defined to exist in this layer.

The second layer is the *Trajectory Metric layer*. In this layer, all trajectory metrics are calculated. We have three dimensions in our system, namely time, latitude and longitude. The topology calculation results are further processed as numeric metrics.

The third layer is the *Trajectory Expression Layer*, where the raw GPS coordinate data are generated into trajectory segments. The raw spatio-temporal data generated from portable devices are loaded in the system in a batch mode. At this layer, each trajectory data are converted to *Point* objects in the data model of Fig. 1. And then the trajectories are further processed as Minimum Boundary Rectangles (MBRs).

The fourth layer is the *Geometry Model Layer*. At this layer, we perform topology calculation. We convert MBR objects to JTS objects. JTS is a topology suit and follows Dimensionally Extended nine-Intersection Model(DE-9IM) [26]. The benefit of converting to JTS objects is that our trajectory metrics relying on geometric calculating operations on MBRs are supported by JTS library, such as spatial predicates, convex hull, and metric calculation referred in Section 8.

The fifth layer is the *Spatial Object Access Layer*. The operations on MBRs are specific to the data processing frameworks. In this paper, we consider two kinds of data processing frameworks, namely a NoSQL graph storage framework and an in-memory processing framework. When MBRs are stored in a graph processing framework (such as Neo4j), a graph query language Cypher is programmed to operate these data. When the MBRs are operated by an in-memory processing framework(such as GeoSpark), we program Spark parallel functions to access MBRs.

TABLE 1
Frequently Used Notations.

Notation	Meanning
Tr_p (resp. Tr_r)	a trajectory p (resp.r)
$mbr_{u,Tr_p,i}$	an MBR in trajectory Tr_p, sequence i, partition u , $K \in N, k \in [1, K]$, K is the maximum segmentation number of one trajectory
$pt_{Tr_p,i}$	a GPS point in trajectory Tr_p, sequence i, $pt \in Tr$
$pt.x$	point longitude
$pt.y$	point latitude
$pt.t$	point timestamp
$R_{c,v}(resp.R_{q,u})$	Candidate MBR relation in partition v (resp. query MBR in partition u)
$Est(Tr_p, Tr_r)$	Trajectory similarity estimation between Tr_p and Tr_r
$Dist(pt_a, pt_b)$	the Euclidean distance between points a and b

The sixth layer is the *storage layer*. In the graph processing system, MBRs are indexed and stored in the directed graph structure. For the in-memory processing framework, MBRs are indexed and stored in a customized tree structure.

4 The Trajectory Segmentation Method

A segmentation method splits one trajectory into maximal K segments. In this paper, segments are the first class entities for indexing, storage, and query. In this segmentation method, we design a data model with components to encapsulate the operations on segments. Then we present a greedy split algorithm to split each trajectory into segments. Since trajectories are independent, this greedy-split method is processed in parallel. The commonly used notations in this paper are listed in Table 1.

4.1 The Data Model

The data model's entities and their relationships are presented in Fig. 2.

Point. Since the GPS trajectories are described as spatio-temporal points, the fundamental component in this data model is *Point*. Compared to existing data models or topology, they mostly support only 2-D attributes of longitude as X and latitude as Y. Out data model support an extra attribute of the timestamp T.

PointSet. A trajectory consists of a cluster of points as time elapses. We define the `PointSet` class to express the sub-trajectories. A `PointSet` can not be empty. We can link these sub-trajectories of `PointSet` to form a longer sub-trajectory by using `addAll()` function or only add one point to extend a current `PointSet`. All these internal points have their sequence order by sorting their timestamps.

We get one position snapshot at a specific time by the function `getPtSnp()`. As an estimation in between two real points, a virtual position is calculated based on the

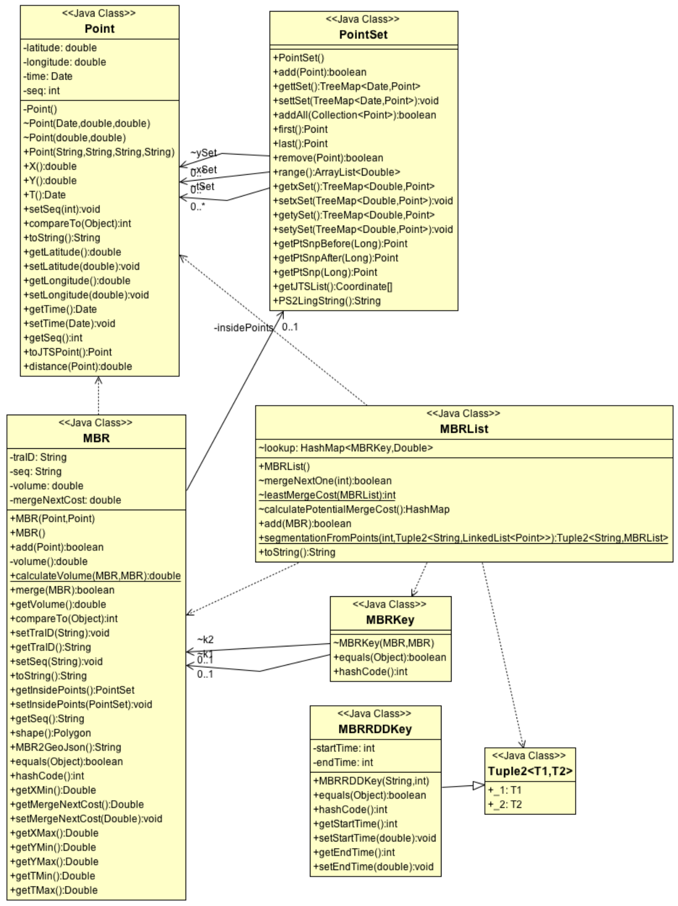


Fig. 2. Data Model of Trajectory Segmentation

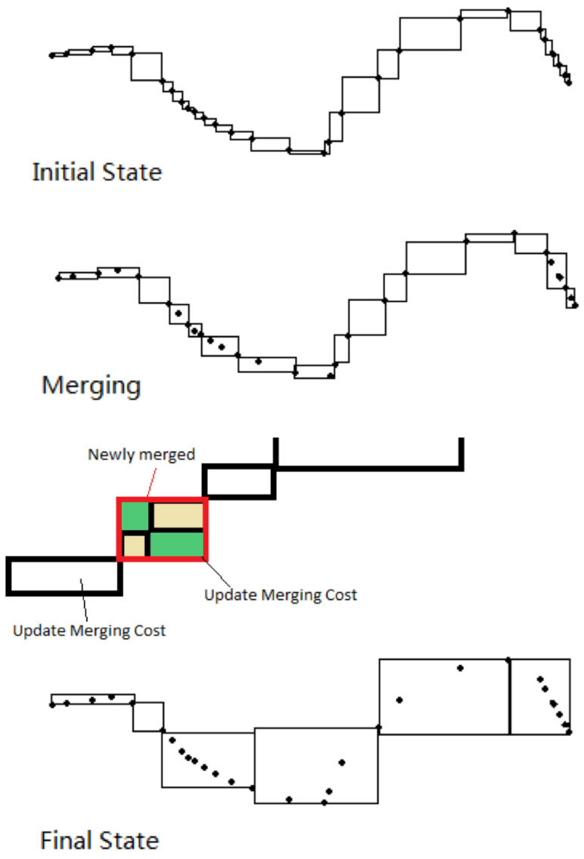


Fig. 3. Main Steps of the Greedy Splitting Process

average velocity between the two adjacent positions with the function above.

Minimum Boundary Rectangle (MBR). The MBR class focuses on the attributes that relate to operations on trajectory segments such as the volume and merging cost in the greedy-split algorithm. the details follow later. Since an MBR covers a sub-trajectory, it is a composition made from this sub-trajectory's `PointSet` and this MBR's four vertexes. We get the MBR vertexes of its sub-trajectory by the `range()` function and get its volume by `volume()` function.

MBRList. The MBRList is a data structure to organize the MBRs in a linked list. The attribute `MBRKey` is used in the `MBRList` for queries.

4.2 The Greedy Split Algorithm

The segmentation process transforms a trajectory expressed by points into a sequence of MBRs. An illustrating process is depicted in Fig. 3. In this process, smaller MBRs are aggregated into larger MBRs. Initially, every two consecutive points in a trajectory sequence resemble as diagonal vertexes of an MBR.

The next step is merging two direct adjacent MBRs. Since an MBR may have its left or right neighbour to merge, the criteria of merging is based on the merging cost. Suppose two consecutive MBRs mbr_a and mbr_b are merged to a new mbr_{ab}, the merging cost is defined as [27] :

$$Cost(mbr_a, mbr_b) = Vol(mbr_{ab}) - Vol(mbr_a) - Vol(mbr_b)$$

Where *Vol* denotes the volume function of the MBR class. The merging action is listed in Algorithm 1.

The merging that leads to a smaller volume is selected. In one round of the greedy splitting algorithm, this merging action repeats till all the MBRs are scanned.

We adopt a greedy-split algorithm [28] to balance the cost and approximation quality. Our implementation uses the data model defined in Section 4.1. The full details of greedy-split are listed in Algorithm 2.

When data points are missing at specific timestamps, the algorithm uses the next available data point to merge MBRs. Therefore, in our implementation of the greedy split algorithm, the size of MBRs, as well as the time span of individual MBRs are both varied.

4.3 Parallel and Distributed Implementation

The greedy-split algorithm is independently applied to each trajectory. Thus the segmentation is processed in parallel. When the dataset contains a large number of trajectories that is beyond a single node's capacity, the dataset can be partitioned on a cluster of nodes. Therefore, each partition

Algorithm 1 The Algorithm of Merging MBRs

```
Input: one MBR $mbr_a$ and its consecutive right side MBR $mbr_b$
Output: a new MBR $mbr_{ab}$that covers both $mbr_a$ and $mbr_b$
1: $P_{a'} := P_a \cup P_b$, where $P_a$ and $P_b$ is $mbr_a$ and $mbr_b$'s inside PointSet
2: get $x_{max} := Max\,(P_{a'}.X)$,
   $x_{max} := Max\,(P_{a'}.Y)$ ,
   $x_{min} := Min\,(P_{a'}.X)$,
   $y_{min} := Min\,(P_{a'}.Y)$
3: $pt_{SW} := Point(x_{min}, y_{min})$ ,
   $pt_{SE} := Point(x_{max}, y_{min})$,
   $pt_{NE} := Point(x_{max}, y_{max})$,
   $pt_{NW} := Point(x_{min}, y_{max})$
4: $mbr_{ab} := Polygon(pt_{SW}, pt_{SE}, pt_{NE}, pt_{NW})$,
   $mbr_{ab}$inside pointSet =$P_{a'}$
5: return a new MBR$mbr_{ab}$ covering $mbr_a$ and $mbr_b$
```

Algorithm 2 The Algorithm of Greedy Split

```
Input: $Tr_p = \{pt_1, pt_2, \cdots\}$: a single spatio-temporal trajectory,
    $K$: an integer denoting the final number of segments split into(All subscripts $Tr_p$ are omitted compared to Table. 1)
Output: An MBR list $MBRList = \{mbr_1, mbr_2, \cdots\}$ that covers $Tr$
    Creation of MBR :
1: for each South West Point $pt_{SW} \in Tr$ and its consecutive right side $pt_{NE}$(assuming located at NE direction) do
2:     create new Points
       $pt_{NW} := Point(pt_{SW}.x, pt_{NE}.y)$,
       $pt_{SE} := Point(pt_{NE}.x, pt_{SW}.y)$
3:     create new MBR, with above four points as vertexes
       $m := Polygon(pt_{SW}, pt_{SE}, pt_{NE}, pt_{NW})$
4:     $MBRList.insert\,(m)$
5: end for
6: for each two consecutive MBRs $mbr_l \in MBRList$ and $mbr_r := mbr_l.next()$ do
7:     call merging algorithm to merge $mbr_l$ and $mbr_r$ to a new temporary MBR $mbr_{lr}$
8:     $Cost(l, r) := Volume(mbr_{lr}) - Volume(mbr_r) - Volume(mbr_l)$
9:     $CostQue.put(Cost(l, r))$
10: end for
    Merging loop :
11: while $M.size() > k$ do
12:     $Cost(i, j) := CostQue.min()$
13:     $mbr_i := merge(mbr_i, mbr_j)$
14:     $MBRList.remove(mbr_j)$,
        $CostQue.remove(Cost(i, j))$
15:     merger $mbr_i$ and $mbr_k := mbr_i.next()$
        to get $Cost(i, k)$
16:     CostQue.insert($Cost(i, k)$)
17: end while
18: return an MBRList $M$ covering $Tr$
```

contains a number of trajectories of which each trajectory is segmented in parallel.

To enable parallel processing on segmentation, we develop our greedy-split algorithm using Resilient Distributed Datasets (RDDs) in Apache Spark [3]. RDDs are distributed datasets processed in memories of worker nodes. RDDs are first created by reading the dataset from stable storage such as HDFS into the partitioned collection of records. The initial RDDs are already distributed and partitioned. The partition size and its distribution are inherited from HDFS's block size. The trajectories in the same partition origin from the same block in HDFS host.

Each trajectory is a text-based file in our dataset. The initial RDDs contain the `key` as the identity of that trajectory, and the `value` as this trajectory's `Point` list as `LinkedList<Point>`. The Point object has latitude, longitude and the timestamp in our data model.

The next transformation is using the greedy-split algorithm to group points into MBRs. After this transformation, the point list is replaced by `MBRList`. In the `MBRList`, each MBR is a polygon element that contains the sub-trajectory `Points` in the `PointSet` structure.

Afterward, we use the `flatMap` transformation to flatten RDD's value that is represented as `MBRList` to a sequence of `MBRs`. The new RDD has the compound key called `MBRRDDKey` that consists of two attributes, (1) trajectory identity and (2) the trajectory's MBR sequence in a chronological order acquired from the `MBRList`.

5 Partition and Indexing

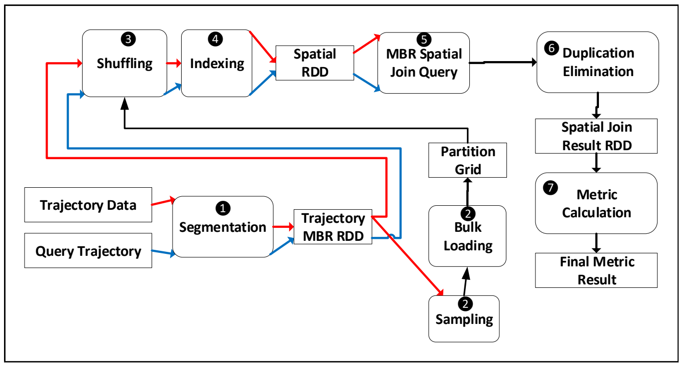
The trajectory is repartitioned and system shuffles all MBRs within a specified geographic boundary to the same partition. We further extend the partition to the temporal boundary that MBRs within a certain time period are partitioned to the same node. Under this spatio-temporal repartitioning, an intersection query to sub-trajectories within the spatial-temporal boundaries occurs within the same partition. This is different from the initial MBR based partition discussed in Section 4.3. In the initial MBR based partition, all MBRs of the same trajectory are located in the same partition, and all the trajectories with similar name prefix are also located in the same partition. Compared to the initial file based partition, the spatio-temporal partition significantly reduces the data shuffling.

The workflow of parallel partition is depicted in Fig. 4. It contains seven stages: (1) segmentation; (2) data sampling and bulk loading; (3) data shuffling; (4) indexing; (5) spatial query; (6) duplication elimination; and (7) metric calculation. Stages execute to fulfill the workflow logic. A stage further contains several tasks with each task performs the transformations on each partition of an RDD in parallel.

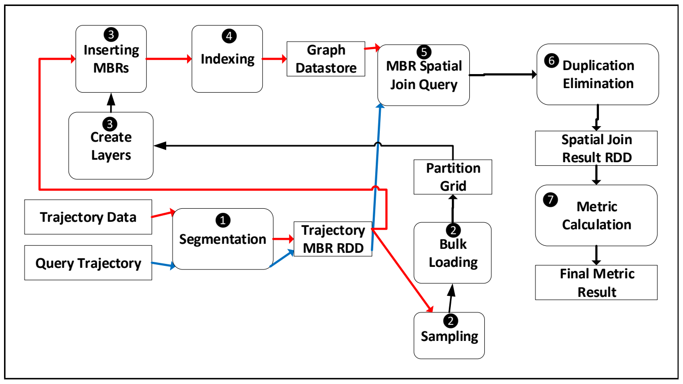
We notate activities of this data flow with numbers to present the techniques involved and the mapping of activities in the distributed cluster deployment (depicted in Fig. 5).

5.1 Partitioning Techniques

The *Spatial Partition* activity in the data flow applies a spatial partition method. The spatial partitioning methods include Equidistant, Hibert, Voronoi, Quad-tree and R-tree [29].



(a) In-memory Processing Workflow



(b) Graph Database Workflow

Fig. 4. Framework Workflows - Both workflows share stages. After segmentation, the in-memory processing framework stores MBRs in cluster node memories; and the graph database framework stores MBRs in the distributed graph database.

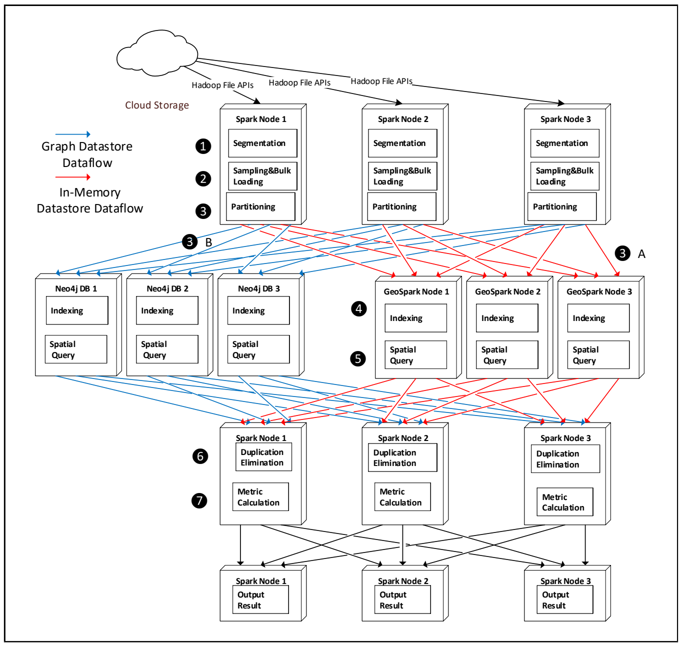


Fig. 5. Dataflow Between Nodes

We develop an R-tree partition to achieve balanced partition of MBRs. Since we aim to put MBRs within the same spatio-temporal boundary to the same partition, the even distribution of MBRs among partitions is achieved through the adjustment of boundary size. The boundary size of a partition is determined by factors as the number of partitions and the number of trajectory MBRs. Assume we have 2000 MBRs after the segmentation process, and sample 1% MBRs to build the R-tree. Then we get 20 MBRs to form the range of boundaries. Assume further that we have 10 partitions, then 20 MBRs' spatio-temporal span is divided into the 10. In addition, we have one more range called overflowed partition for any MBRs that are beyond the spatio-temporal span from the samples. In general, if we target p number of partition, we finally have $p + 1$ ranges.

The construction of R-tree is through inserting an MBR object into the node of the R-tree whose range covers the MBR's spatio-temporal scope. To insert an MBR object, the tree is traversed recursively from the root node. Within a current directory node (non-leaf nodes), all MBRs in the node are examined until it reaches a leaf node. If a leaf node is full, the leaf node must be split before the insertion. Therefore overflow from a leaf node is recursively propagated to nodes in previous levels. When there are more options to split a node, one heuristic rule is to split the node that requires the least enlargement. Eventually, the goal of building an R-tree is to build a balanced tree in which 1) leaf nodes are of the same level of depth; and 2) bounding boxes should not cover too much empty space.

Each leaf node of the R-tree has even numbers of MBR objects contained. Therefore the boundary range of each leaf node is dynamically changed to make the balanced distribution of MBR objects. Since the range of each leaf node represents a geographic area within a period time, the adjustment of the range scale of a leaf node eventually modify the geographic area given a time covered by the partition, thus the density distribution of MBRs on each partition. If the objects in an area are scattered, this leaf node contains more extensive range than average; if the objects in an area are dense, this leaf node contains a smaller range than average.

We develop the R-tree spatial partition in-memory. The details of the workflow steps are presented below.

Stage 2: Data sampling and bulk loading In this stage, we create a geological partitioning strategy. This is a spatial grid based on R-tree rectangles.

Data sampling. We randomly sample 1% of the whole MBRs for establishing range boundaries of the R-tree. This sampling method avoids building a global index thus helps to reduce the computation cost.

Bulk loading. We build the R-tree using the Sort-Tile-Recursive (STR) algorithm [11] to split overflowed nodes. The STR algorithm estimates the number of leaves required as $l = \lceil samplesize/p \rceil$, except the last overflowed partition that represents the range beyond the boundaries of samples. Eventually, the R-tree has $l + 1$ leaves.

Each leaf represents one geological boundary. The generated R-tree is stored as a `SpatialRDD` for further query usage. `SpatialRDD` is an abstract class that stores geometry distributively with index support. It also allows users to accomplish multiple spatial operations such as `Distance Join`, `Range Query` and `KNN Query`.

Stage 3: Partition assignment and distribution. When an MBR inserted has an intersection with any leaf node boundary in the R-tree, the MBR is assigned to the partition number that the leaf node belongs to. The partition number becomes the key to the MBR's RDD. We further develop a *replication* method to handle the cases that boundaries may have overlapping or one MBR is large enough to across multiple partitions. With this *replication* method, the MBR across multiple boundaries is assigned to multiple partitions. To make the query result consistent, duplicated copies are removed after a query. The assignment algorithm is shown in Algorithm 3.

Algorithm 3 Algorithm for R-tree Partition Assignment

```
Input: $mbr_{Tr_c}$: one trajectory $Tr_c$ segmented Candidate MBR and
    partitionList: the partition grid generated from R-tree partition method
Output: a partition ID indicating which partition it belongs to
 1: containFlag := false
    Iterate Each Partition :
 2: for each $i^{th}$ partition from partitionList do
 3:   if $partition_i$ covers $mbr_{Tr_c}$ then
 4:     partitionID := i
        /*Contain only check*/
 5:     containFlag := true
 6:   else if $Partition_i$ intersects $mbr_{Tr_c}$ then
 7:     partitionID := i
 8:   end if
 9: end for
10: if containFlag is false then
11:   partitionID = overflow
12: end if
13: return partitionID
```

5.2 Data Shuffling

Data shuffling step enables MBRs are distributed into multiple nodes based on partitioning rules. All MBRs within the same geographical partition should be located on the same worker node. We apply the `partitionBy()` function in Spark to locate all MBRs (in RDDs) with the same key to the same partition. This incurs data shuffling between Spark nodes.

5.3 Data Persistence

Data persistence provides the possibility for MBRs to be distributed into multiple graph databases. The persistent method of MBRs migrates the MBRs in Spark RDDs to the NoSQL database, Neo4j. To support our spatial data model, we deploy the Neo4j-spatial extension [30] that contains map layers. A Neo4j map layer is similar to the Spark partition. One map layer exists in only one data node and is not distributable. One data node has several map layers. Each layer is independent. Similar to data shuffling on GeoSpark, MBRs with the same key are inserted into the same map layer of Neo4j. To keep the term consistent, we refer a map layer as `partition` too.

Create partitions on Neo4j. We deploy a total number of s Neo4j data nodes. Each node runs independently with distinct data partitions. Therefore we deploy each node in the standalone mode rather than clustering nor in master/worker mode. The number of map layers (or partitions) is $p + 1$. Then we assign $p + 1$ map layers (or partitions) to s nodes using the binning method. To identify the node destination, $NodeNumber = MBRKey\%s$ and $LayerNumber = MBRKey$. We build a router to route each MBR to a designated map layer (or partition).

Inserting MBRs to Neo4j partitions. We traverse all the MBRs in each partition and execute the insertion operation in parallel. If MBRs are assigned to a partition on its local Neo4j node, the data are shuffled between map layers (partitions) on the same physical node. When MBRs are assigned to a remote Neo4j node, a remote procedure call of Neo4j is executed, which incurs data shuffling across the network.

5.4 Local R-tree Indexing

Stage 4: Indexing Inside each partition, local R-tree indexes are built for fast retrieving. The implementation of R-tree indexing differs from in-memory processing and the graph storage system.

In-memory R-tree indexing. In each partition, each MBR is traversed to create an entry in a local R-tree index. Each partition's local R-tree is represented in `SpatialRDD` that is also distributed.

Graph storage R-tree indexing. The R-tree indexes are built simultaneously when inserting MBRs into the map layers. The key of each MBR is a compound key including `Partition ID` and `Time Slot ID`. The `Time Slot ID` is used as an extra attribute to balancing the data distribution over time. We divide one day into multiple time periods and number each time slot. More details are presented in Section 10. Based on this compound key, we hash the key and map the MBR object to a map layer. The visualization of an R-tree structure in Neo4j is shown in Fig. 6.

6 The Query Workflow

Above steps generate spatio-temporal data indexes and store the data partitions in a cluster of nodes. We further develop queries on trajectories and output MBRs that meet certain predicates. The queries compute metrics regarding MBRs' attributes. Given a query trajectory, the objective is to find out other trajectories having a similar route or having any intersection with the given one. A metric evaluating the degree of similarity is also defined below. The overall workflow is shown in Fig. 7.

Preprocessing Query Trajectory. The query trajectory needs to be presented in the same format of the trajectory datasets that are already indexed and stored. This step involves the same techniques of trajectory segmentation, partitioning and indexing as discussed in previous sections. Queries from the in-memory processing system and from the graph storage system have separate techniques.

6.1 The Parallel Intersection Join

When a query is executed on partitioned segments of trajectories, a property is required to ensure the intersection join consistent as if the intersection join is performed sequentially.

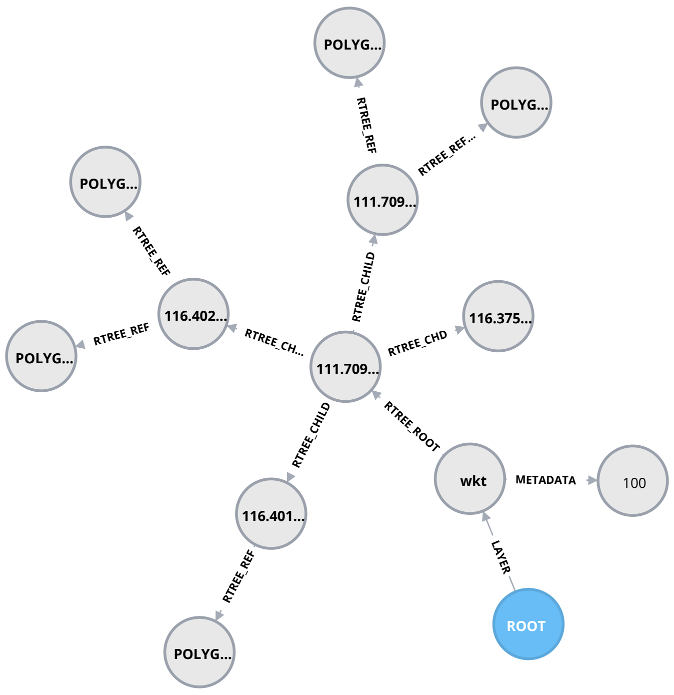


Fig. 6. The Neo4j Local R-tree Visualization. (A root node is called `spatial_root` in the blue colour. The root node links to each `Layer Node` using a `Layer` relationship. For a specific `Layer Node`, another node records the max node for each R-tree layer. The geometries are linked by the `RTREE_METADATA` relation. The top level of the R-tree is a Boundary Box (BBox) covering all the geometries. The BBox is accessed following the `RTREE_ROOT` relation. The non-top level BBoxes are connected by `RTREE_CHILD` relations. At the leaf level, MBRs are serialized in the WKT format and stored as one property.)

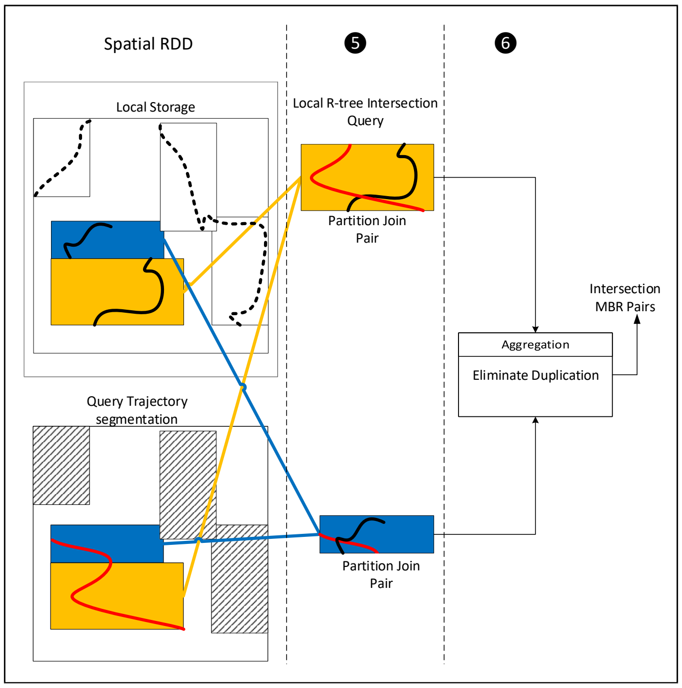


Fig. 7. The Trajectory Query Workflow in Two Parallel Partitions. (A query trajectory is shown in red color (In the query spatial RDD) that is covered by MBRs distributed in two partitions, in yellow color and in blue color respectively. The candidate MBRs (illustrated by spatial RDD local storage) in blue color and yellow color on two distributed partitions.

We define $R_c = \bigcup_{v=1}^{p+1} R_{c,v}$ is the relation of candidate MBRs, consisting of $p+1$ partitions, and $R_q = \bigcup_{u=1}^{p+1} R_{q,u}$ is the relation of query MBRs. The *intersection join* is defined as:

$$\begin{aligned} &R_{q,mbr_{Tr_q},j} \bowtie R_c = \\ &\{mbr_{Tr_c,i} | \exists mbr_{Tr_c,i} \in R_c \wedge mbr_{Tr_q,j} \in R_q \\ &\wedge Intersects_{xy}(mbr_{Tr_c,i}, mbr_{Tr_q,j}) = true\}. \end{aligned} \quad (1)$$

Where the `Intersects(a,b)` predicate is defined in Dimensionally Extended nine-Intersection Model (DE-9IM) [26]. `Intersects(a,b)` returns true when geometries a and b have at least one point in common.

This means for each MBR mbr_{Tr_q}, j in partition u within the query range, it may exist an MBR belonging to Tr_q index i, mbr_{Tr_c}, i ($0 \leq i \leq k$) in partition $v_1, v_2 \cdots v_n$ that overlap with mbr_{Tr_q}, j. That is $R_{q,mbr_{Tr_q},j} \bowtie R_c = mbr_{Tr_c,i}$.To ensure consistency of the intersection join on partitions of MBRs, we first define an *intersection closure* as $R_{q,mbr_{Tr_q},j} \cup R_{c,mbr_{Tr_c},i}$ that covers the overlapping MBRs, where

$$R_{c,mbr_{Tr_c},i} = \Pi_{mbr_{Tr_c},i}(\bigcup_{v=v_1}^{v_n} R_{c,v}) \subseteq \bigcup_{v=v_1}^{v_n} R_{c,v}, \quad (2)$$

$$R_{q,mbr_{Tr_q},j} = \Pi_{mbr_{Tr_q},j}(R_{q,u}) \subseteq R_{q,u}. \quad (3)$$

Because of the *replication* strategy, we ensure partition $u = v_1 = v_2 = \cdots = v_n$. Hence, $R_{q,mbr_{Tr_q},j} \cup R_{c,mbr_{Tr_c},i} \subseteq R_{q,u} \cup R_{c,u}$. Consequently, $R_{q,mbr_{Tr_q},j} \bowtie R_c$ is performed on the super partition closure set of $R_{q,u} \cup R_{c,u}$. To generate the super parition clouser set of $R_{q,u} \cup R_{c,u}$, a *reduce* transformation aggregates MBRs from separate partitions in **Stage 5 spatial query**. An example is illustrated in Fig. 7. The two blue partitions form a closure for intersection join and two yellow partitions form another closure for intersection join. Therefore the query is aggregated by two queries. As a result, the intersection join is consistent with the sequential and centralized processing whereby the overlapping MBRs are within the same partition.

6.2 Spatial Query

Stage 5: Spatial query. The parallel spatial query is handled as map-reduce calls to return MBRs. At the map stage, the local R-tree index is retrieved for the range queries. The query MBR interacts with MBRs in each partition within the query range. Then each partition produces candidate MBRs. The algorithm is listed in Algorithm 4.

At the reduce stage, candidate MBRs are aggregated to produce the complete set of candidate MBRs. The algorithm is listed in Algorithm 5.

Our framework handles query implementation separately for the in-memory processing system and the graph storage system. For the in-memory processing system, we utilize the local R-tree on each partition to find out the intersected MBRs with the query trajectory's MBR. This involves the join operation that is converted to multiple rounds of range queries. Finally, each query MBR is associated with

Algorithm 4 Algorithm for Join Query in Map Procedure Call

```
Input: MBR relation in candidate partition k $R_{c,k}$ stored in R-tree structure RTreeIndex and query MBRs queryMBRList segmented from $Tr_q$ in query partition $R_{q,k}$
Output: tupleList: a list of tuples tupleList in which the query MBR as key and intersected MBRs as values
1: for each $mbr_{Tr_q,j}$ in $R_{q,k}$ do
2:   candidateMBRList = RTreeIndex.query($mbr_{Tr_q,j}$) /*Using Index for query*/
3:   for each $mbr_{Tr_c,i}$ in queryResult do
4:     if $mbr_{Tr_c,i}$.intersects($mbr_{Tr_q,j}$) then
5:       candidateMBRSet.add($mbr_{Tr_c,i}$)
         /*Using index can not ensure all results are correct, intersection judgment once more*/
6:     end if
7:   end for
8:   tupleList.add(Tuple($mbr_{Tr_q,j}$,candidateMBRSet))
9: end for
10: return tupleList
```

Algorithm 5 Algorithm for Join Query in Reduce Procedure Call

```
Input: $<< mbr_{Tr_q,j} >$ , candidateMBRs $[mbr_{a,Tr_c,m}, mbr_{b,Tr_d,n} \ldots] >$ collecting from map stage
Output: a list of tuples as the spatial join result
1: for each $mbr$ in candidateMBRs do
2:   MBRList.add($mbr$)
3: end for
4: MBRList.deleteDuplicateGeometry()
5: return $< mbr_{Tr_q,j}$, MBRList>
```

a `JavaRDD< QueryMBR,HashSet<MBR>>` object whereby the `HashSet<MBR>` contains candidate MBRs within the query range. For the graph storage system, the intersection join calls the `intersect` procedure to return candidate MBRs.

The duplicated MBRs replicated to multiple partitions result in duplicated intersection MBRs. This stage groups these duplications and removes them. Only distinct intersection MBRs remain to the final stage of trajectory metrics calculation.

7 Trajectory Metrics

We define two basic metrics that are computed using the framework and workflows developed above. These two metrics provide measurements to applications such as clustering analysis of trajectory data. One metric estimates the trajectory similarity.The other metric is used to detect the collision of trajectories.

7.1 Trajectory Similarity Estimation

We measure the similarity of trajectories by computing the volume of overlapping. Since trajectories are segmented into MBRs, the overlapping of trajectories is assessed by intersecting MBRs. The volume is calculated by three-dimensional volume size.

For a trajectory Tr_p, the i_{th} MBR is notated as $mbr_{Tr_p,i}$. It has six attributes represented as $\{t_l, t_h, x_l, x_h, y_l, y_h\}$. t_l and t_h are the starting and ending time of this MBR; x_l and x_h are the MBR's lowest longitude and the highest longitude; y_l and y_h are the lowest latitude and the highest latitude.

Given two trajectories Tr_p and Tr_r, we define the partial `intersection` operation on the time dimension to get the intersection volume from Tr_p's i_{th} MBR and Tr_p's j_{th} MBR:

$$Intersection_t(mbr_{Tr_p,i}, mbr_{Tr_r,j}) = \bigcap_t^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j}).$$

where (p) denotes the partial intersection on the time dimension. We have the following property:

$$mbr_{Tr_p,i}.t_l \le mbr_{Tr_r,j}.t_l \le mbr_{Tr_p,i}.t_h;$$

or

$$mbr_{Tr_p,i}.t_l \le mbr_{Tr_r,j}.t_h \le mbr_{Tr_p,i}.t_h.$$

Next, the norm for the time dimension is defined as

$$getLength(Intersection_t(mbr_{Tr_p,i}, mbr_{Tr_r,j})) = ||\bigcap_t^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j})||.$$

Likewise, we define the intersection on longitude as $Intersection_x$ and on latitude as $Intersection_y$.

Then the intersection in an area is defined on two dimensions of longitude and latitude as

$$Intersection_{xy}(mbr_{Tr_p,i}, mbr_{Tr_r,j})) = \bigcap_{x,y}^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j}).$$

$$getArea(Intersection_{xy}mbr_{Tr_p,i}, mbr_{Tr_r,j})) = ||\bigcap_{x,y}^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j})||. \tag{4}$$

Now we define the intersection volume as

$$\begin{aligned} & getVolume(Intersection_{xyt}(mbr_{Tr_p,i}, mbr_{Tr_r,j}))) \\ =& getArea(Intersection_{xy}(mbr_{Tr_p,i}, mbr_{Tr_r,j}))) \\ & \times getLength(Intersection_t(mbr_{Tr_p,i}, mbr_{Tr_r,j}))) \\ =& ||\bigcap_V^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j}||_V \\ =& ||\bigcap_{x,y}^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j})|| \\ & \times ||\bigcap_t^{(p)}(mbr_{Tr_p,i}, mbr_{Tr_r,j})||. \end{aligned} \tag{5}$$

where V denotes volume in three dimensions.

For any trajectory Tr_p and a query trajectory Tr_r, we have the `Intersection Volume` calculated as:

$$Volume(Tr_p, Tr_r) = \sum_{i=1}^{n}\sum_{j=1}^{m} ||mbr_{Tr_r,j} \bigcap_V m_{Tr_p,i}||_V.$$

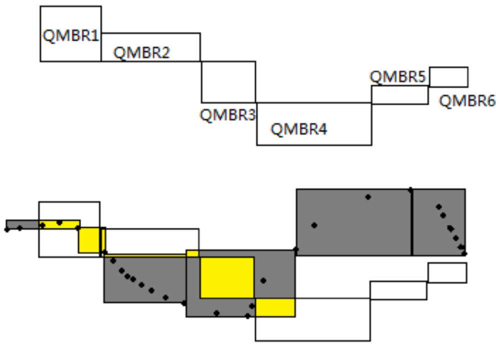


Fig. 8. Similarity Estimation in Two Dimensions

where m is the number of MBRs for Tr_r and n is the number of MBRs for Tr_p.

Finally, we get the similarity estimation between Tr_p and Tr_r as $Est(Tr_p, Tr_r)$:

$$Est(Tr_p, Tr_r) = \frac{1}{length(Tr_p)} \times \sum_{i=1}^{n}\sum_{j=1}^{m} \frac{||mbr_{Tr_r,j} \bigcap_V mbr_{Tr_p,i}||_V}{||mbr_{Tr_p,j}||_V} ||mbr_{Tr_r,j}||, \tag{6}$$

where $length(Tr_p)$ is the trajectory distance of this moving object, m is the number of MBRs for Tr_r and n is the number of MBRs for Tr_p. An example is shown in Fig. 8 whereby the yellow area is the similarity estimation value of these two trajectories. The algorithm is shown in Algorithm 6.

Algorithm 6 Intersection MBR volume calculation

Input: MBRRDD: an RDD containing MBRs
Output: a tuple that the two intersected MBRs as key and their intersected volume as value

```
1:  <<QueryMBR>,[<IntersectedMBRs>]>
    queryResult := MBRRDD.spacejoin(MBRRDD);
2:  for each $< K, Y >$ pair in queryResult do
3:    for each $mbr$ in $Y$ list [$< IntersectedMBRs >$] do
4:      intersectedShape := QueryMBR.intersection($mbr$)
5:      volume2D := intersectedShape.getArea();
6:      intersectedTimePeriod :=
        QueryMBR.getTimeInterval().overlap(
        $mbr$.getTimeInterval());
7:      volume3D := volume2D*intersectedTimePeriod;
8:      return <<QueryMBR,mbr>,<volume3D>>
9:    end for
10: end for
```

7.2 Collision Detection Metric

The collision detection metric is defined as the boolean value to check if two moving objects have overlapping under a certain time span.

We first select the collision detection candidates from MBR pairs whose *similarity estimation* > 0. Therefore the collision detection depends on the metrics of similarity estimation. We give a margin to expand the range of MBRs so that the *similarity estimation* > 0. Usually, the margin is set slightly larger than half of the threshold.

To test the condition of collision detection, we sample location points on sub-trajectory in an MBR. Certainly, the more points sampled, the more precise the result. We have three steps to calculate this metric. First, we find out the timestamps of the checkpoints. A checkpoint is a trajectory position point at a certain timestamp. Secondly, we calculate the interpolation between two real data points as checkpoints. Finally, we examine the distance between a series of checkpoint pairs.

Checkpoint timestamp selection. For any time span as a result of the operation

$$Intersection_t(mbr_{Tr_p,Tr_r}),$$

we define

$$T_{min} = Min\{Intersection_t(mbr_{Tr_p,Tr_r})\}$$

and

$$T_{max} = Max\{Intersection_t(mbr_{Tr_p,Tr_r})\}$$

.

We have a parameter L as an input reflecting how many checkpoints are required to examine. To get the timestamp of a series of checkpoints:

$$T_{ckp}[l] = \frac{l \times (T_{max} - T_{min})}{L} + T_{min}, \quad 0 \le l < L, l \in N. \tag{7}$$

Checkpoint coordinate calculation. Since not all data points are recorded at T_{ckp}, we use a liner interpolation method to estimate the checkpoint position. To find out the index h and $h+1$ of nearest data points to $pt_{Tr_p}(T_{ckp})$:

$$indexSet = \{h| \exists h, getTime(pt_{Tr_p,h}) < T_{ckp}[l] < getTime(pt_{Tr_p,h+1})\}. \tag{8}$$

We find the coordinate x and y for Tr_p or Tr_q at $T_{ckp}[l]$ timestamp. For x coordinate of Tr_p at time instant $T_{ckp}[l]$:

$$x_{Tr_p,T_{chk}[l]} = pt_{Tr_p,h}.x + Dist(pt_{Tr_p,h}, pt_{Tr_p,h+1}.x * \frac{T_{ckp}[l] - getTime(pt_{Tr_p,h})}{getTime(pt_{Tr_p,h+1}) - getTime(pt_{Tr_p,h})}; \tag{9}$$

For y coordinate of Tr_p at time instant $T_{ckp}[l]$:

$$y_{Tr_p,T_{chk}[l]} = pt_{Tr_p,h}.y + Dist(pt_{Tr_p,h}, pt_{Tr_p,h+1}).y * \frac{T_{ckp}[l] - getTime(pt_{Tr_p,h})}{getTime(pt_{Tr_p,h+1}) - getTime(pt_{Tr_p,h})}. \tag{10}$$

Where the `Dist()` function is the Euclidean distance between two points.

Collision detection Based on the point position, measure the Euclidean distance between two trajectories at timestamp $T_{chk}[l]$ to decide if there is any collision.

We define collision detection to be true as a condition that:

$$\exists \quad l \in d, \; Dist((x_{Tr_p,T_{chk[l]}}, y_{Tr_p,T_{chk[l]}}), (x_{Tr_r,T_{chk[l]}}, y_{Tr_r,T_{chk[l]}})) < threshold.$$

The algorithm is listed in Algorithm 7. In our system, we set L as a constant value of 3. When two MBRs collide, we record the MBR identity and the collision timestamp.

Algorithm 7 Algorithm for Trajectory Collision Detection

```
Input: L: number of checkpoints, threshold: distance considered two trajectories are collided, $Tr_p, Tr_r$ : two trajectories
Output: a boolean value indicating if $Tr_p$ collides with $Tr_r$
 1: collision = false
 2: for each $i^{th}$ MBR $mbr_i$ in Trajectory $p$ do
 3:   for each $j^{th}$ MBR $mbr_j$ in Trajectory $r$ do
 4:     if similarity estimation between $mbr_i$ and $mbr_j$ = 0 then
 5:       return false
 6:     end if
 7:     for each checkpoint sequence $l$ from 0 to d-1 do
 8:       calculate $T_{ckp}[l]$ using formular (7)
 9:     end for
10:     for each $T_{ckp}[l]$ in $T_{ckp}$ do
11:       find index $h_p$ and $h_r$ that satisfy the function (8)
12:       calculate points $P_a = (x_{Tr_p,T_{ckp}[l]}, y_{Tr_p,T_{ckp}[l]})$ as well as $P_b = (x_{Tr_r,T_{ckp}[l]}, y_{Tr_r,T_{ckp}[l]})$
13:       if $Dist(P_a, P_b) < threshold$ then
14:         collision = true
15:       end if
16:     end for
17:   end for
18: end for
19: return collision
```

8 An Evaluation Application

We develop an application by finding out moving crowds applying above similarity estimation and collision detection metrics. A crowd is a set of reachable objects that have collisions in a particular time span [31]. We use graph theory model to explain the crowd query procedure.

This model is expressed as a graph G consisting of the edges $E(G)$ and vertices $V(G)$. An MBR is a vertex in the graph G, expressed as $u \in V(G)$ or $v \in V(G)$. A collision event is an undirected edge connecting every two vertices with the numerical value of collision timestamp, marked as $(u, v) \in E(G)$. We call this associated value as weight, marked as $D(u, v)$. Between these MBRs, a crowd is a set of MBRs in which MBRs are maximally connected to each other by the edges.

By searching for the connected components using the Breadth First Search (BFS) or Depth-first search (DFS), we derive the crowds. The graph creating and search procedures are listed in Algorithm 8. The result expressed in graph theory is shown in Fig.9.

8.1 Test Dataset

We apply our crowd mining solution on open data from Microsoft GeoLife project [6], which is a GPS trajectory dataset generated by 182 users. The trajectory length varies from a few minutes to several days, mostly distributed in Beijing urban area. The whole data size is of 1.6GB, including 17621 trajectories with a distance of 1,300,000 km and a total of 50,000 hours. We use slices of datasets from the size of 100MB to the full size of 1.6GB for the varied size of the data input. Since the trajectories are sparsely distributed in five-year range, we ignore the date attribute but keep the time attribute to make the data denser as if the events happen in one day. We randomly select 3330 trajectories to find out crowds.

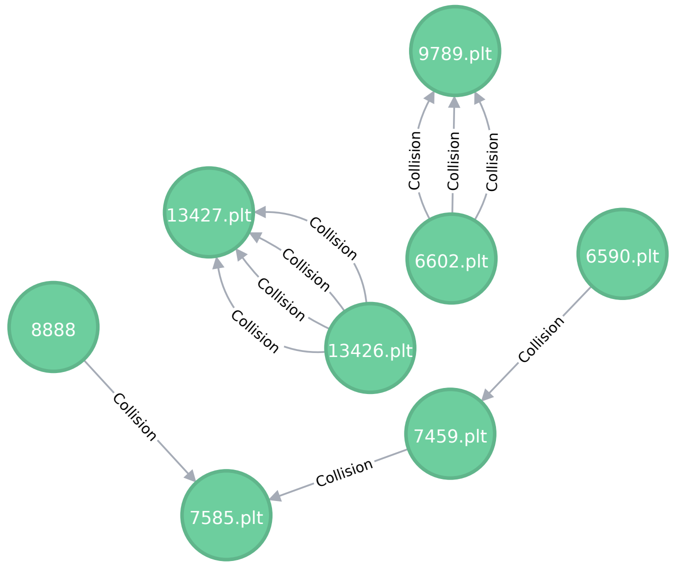


Fig. 9. The Connected Components(Crowds) in Graph Database (We use trajectory nodes instead of the MBR nodes for a better illustration.)

Algorithm 8 Algorithm for Crowds Search

```
Input: $[< mbr_{Tr_q,j}, mbr_{Tr_c,i} >]$ : a list of MBR pairs that have collisions with each other.
Ensure: Set crowds: each element in the set is a connected component representing a crowd.
 1: for each MBR pair $< mbr_{Tr_c,i}, mbr_{Tr_q,j} >$ do
 2:   create vertex $u_i$, vertex $v_j$.
 3:   create edge $(u_i, v_j)$
 4: end for
 5: Set crowds = new Set()
 6: for each edge$(u, v) \in E(G)$ do
 7:   Breadth-First or Depth-First Search starting from $u$ to find out a connected component $G'$
 8:   crowds.add($G'$)
 9: end for
10: return crowds
```

8.2 Existing Gathering Implementation

A *gathering* is defined by Zheng et al. [32] as a pattern occurring at a certain area or location in a certain time period indicating a non-trivial event. They use the DBSCAN [33] algorithm to discover the crowd on snapshots, then detect the gathering patters. Finding crowds is the intermediate step to detect the gathering patterns.

The gathering crowd is the intermediate result when detecting the gathering patterns. We compare the crowds produced by our workflow and ones produced by the gathering Spark implementation, referred as `GPFinder` [5]

TABLE 2
Confusion Matrix for Gathering Detection Result,a total of 3330 Trajectories

		GPFinder	
		Positive	Negative
Our Predicted Results	Positive	2484	256
	Negative	421	169

TABLE 3
Confusion Matrix for Positive Detected 2740 Trajectories as Input (There is no trajectory we detected as negative as input for GPFinder.)

		GPFinder	
		Positive	Negative
Our Predicted Results	Positive	2671	69
	Negative	No Input	No Input

in the following analytic. We set our $collision\ threshold =$ gathering finder application's $threshold$ for further tests.

8.3 Small Size Trajectory Analytics

Evaluating the accuracy of the whole dataset is not quantifiable since the data are not labeled for ground truth. The analysis is non-supervised. To solve this problem, we manually label 13 trajectories and use them as ground truth data for evaluation. We set our collision detection threshold and margin to 10 meters. The K value (max segmentation number per trajectory) is set to 20. We compare our gathering crowds with [5]'s result.

We visualize the crowd trajectories for manual evaluation. From Fig. 10, we find that the A, B and C crowd pairs occurred at a bus station in front of a university campus and the G pair occurred at a subway station. We notice that D, E, and F gathering pairs have the same common trajectory whose MBRs are extremely large. One reason is the poor data quality that some coordinates of this trajectory location recording may be lost.

8.4 Medium Size Trajectory Analytics

We also select 400MB data, which include 3330 trajectories for crowd finding. The collision detection threshold is set to 5 meters. We find 2740 trajectories are positive, which means they form crowds; while the rest 590 trajectories are isolated. We also use similar parameter settings with `GPFinder` algorithm to find out the gathering crowds with the same dataset. The confusion matrix is listed in Table 2.

We notice that in our system, the sensitivity remains 86%, but the specificity is only 49%. We suspect that it is the 590 isolated trajectories that interfered `GPFinder` to find out crowds.

We extract the positive gathering trajectories from our application, which is 2740 from Table 2. We put these 2740 trajectories as input to rerun both `GPFinder` and our application. Our application remains the result that all 2740 are still positive. From Table 2 and Table 3 we observe that `GPFinder` positive number rises a bit from 2484 to 2671.

8.5 Trajectory Transforming to MBR Visualization

We create a heat map showing the distribution of our MBRs with the dataset of 400MB. Similarly, we also plot

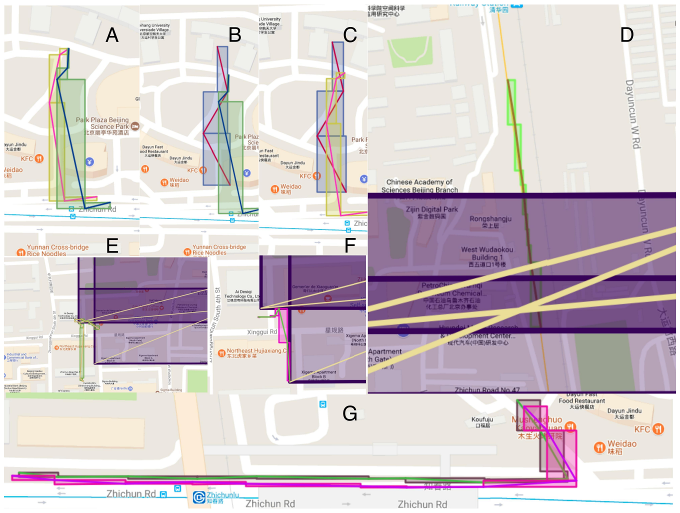


Fig. 10. Positive Crowd Pair Trajectories Snapshot(Due to lacking GPS records, the purple trajectory shows a pathological interpolation.)

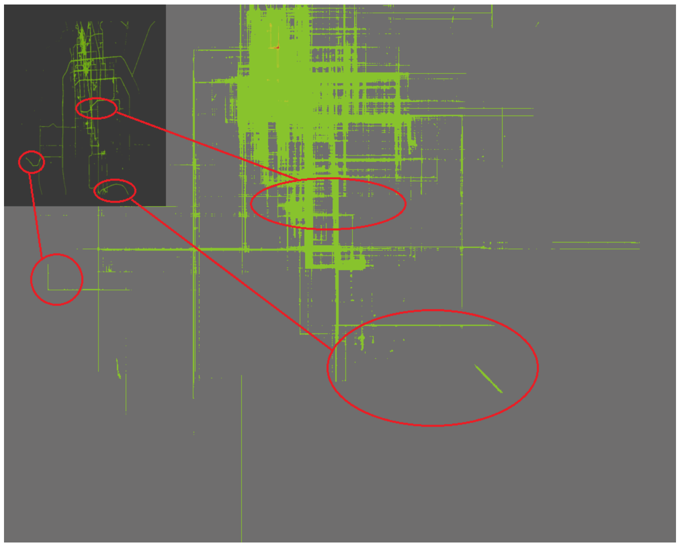


Fig. 11. The Heat Map of Trajectories and MBRs (The picture on the top left is the trajectory heat map, the picture at the center is the MBR heat map.)

the heat map of these trajectories. Here we visualize and compare the skeleton of segmented trajectory MBRs and the original trajectories. The aim of visualizing the MBRs and trajectories is to find out how the trajectory layout changes after we convert the point based trajectory into rectangle based MBRs. The heatmap can easily observe the distributing of trajectory density. We notice the sketch of the trajectory heatmap in the top left corner is highly similar to that of MBRs in the center of Fig. 11. We also highlight the three different parts where the roads are not horizontal or vertical.

9 System Performance Evaluation

In this section we aim to evaluate the system performance and scalability under experiments by varying

1) The cluster size;
2) The input data size;

3) The partition number;
4) The segmentation number.

The input data size vary the workload size. The other three parameters affect the distribution of the workload. We apply the same set of metrics to both the in-memory framework and the graph storage framework to compare the effects of the system architecture. To further identify the performance bottleneck, we decompose the latency by stages.

9.1 The Experiment Setup

Our computing nodes are set up on the Amazon EMR platform. All nodes are R4.2xlarge instances. Each R4.2 xlarge instance has 8 cores with 61 GB memory. We have one extra R4.2xlarge instance as the master node.

The workflow computes the *collision detection* metrics which in turn computes the *similarity estimation* metrics. The tasks for computing each metric execute spatial join with the MBR intersection predicate passing through seven stages shown in Figure Fig. 4. In the collection detection metric, the threshold is set to 5 meters, and the margin is fixed to half of the threshold value.

We refer to common measurements to compare two system architectures of in-memory processing system and the graph storage system.

Latency measures the end-to-end elapsed time.

Throughput measures how many bytes of data a system processes in a given amount of time. It is defined as $Throughput = DataSize/Latency$.

Speedup is defined by:

$$S = \frac{T_s}{T_p}$$

Where the T_s is the single node runtime latency and T_p is the multi-node cluster runtime latency.

Latency decomposition shows the execution time portion of significant tasks to observe the most time consuming steps of a workflow.

Shuffle Read/Write rate indicates to what extent that data are serialized to and from remote nodes. Reducing this rate helps reduce the I/O cost.

$$Shuffle\ Read\ Rate = \frac{Shuffle\ Input\ Data\ Size}{Input\ Data\ Size}.$$

$$Shuffle\ Write\ Rate = \frac{Shuffle\ Output\ Data\ Size}{Input\ Data\ Size}.$$

where `Input Data Size` is the size of data ingesting to a stage; `Shuffle Write Data Size` is the sum of serialized data from all nodes before transmitting in this stage; and `Shuffle Read Data Size` is the sum of serialized data from all nodes after transmitting at the next stage.

9.2 Evaluation Results

9.2.1 Cluster and Partition Size Efffect

The baseline latency is measured with one node. The algorithm is still running in parallel on 8 cores. The factors affecting the speedup metric include network communication, data locality and level of parallelism when the number of nodes increases.

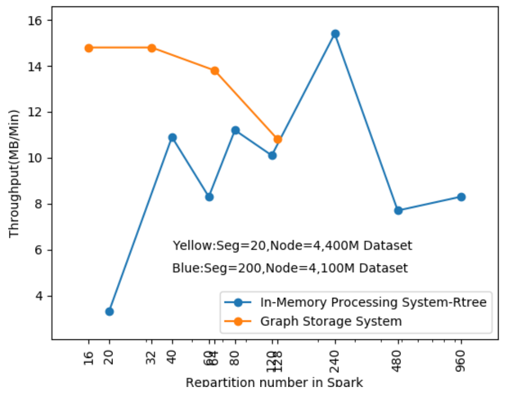


Fig. 12. Throughput Under Different Repartition Numbers

The network communication bandwidth between AWS EMR R4 nodes is 10 GBps. Large amount of data exchange occurs in the reduce stage and the shuffling stage. Serializing the objects is an effective way to reduce the I/O volume. A further discussion is presented in Section 9.2.4.

Data locality refers to how close the data to the processing code. Based on configuration, data and the processing code may reside on different level of locality, such as on the same JVM, on the same node, in the same rack or on different nodes within the network domain. In the experiment, we keep the settings as default to let Spark itself to decide the locality level to minimize the data transfer.

The level of parallelism is reflected by one factor as the number of partitions. The smaller the partition, the more partitions to be scheduled. In Fig. 12, we observe that more partitions do not lead to a higher level of parallelism. When the number of partitions is high, the number of objects across multiple partitions to be aggregated also increases. Hence, the system duplicates these objects for each partition before the local join operation. Meanwhile, at the reduce stage, the system has extra cost of removing duplicated objects.

The speedup under the fixed data size of 400MB is plotted in Fig.13. We automatically set the $partition\ \ number = \frac{total\ \ MBR\ \ number}{300}$ if not explicitly mentioned to minimize the data skew. We observe the in-memory processing system has limited improvement when increasing the cluster size. One reason is that there are more sequential stages in indexing and query than in the graph-based system. We further discuss in Section 10 the data skew effect on the speedup.

The speedup based on the graph storage system has a superlinear benefit. This is due to a much shorter query time when sharding the spatial data into multiple individual databases. The spatial join time complexity is $O(log_M \frac{(n)}{p})$ where M is the capacity per R-tree node and p is the partition number. In the graph storage system, a single spatial join query uses less time after sharding.

The throughputs under different cluster sizes are compared as depicted in Fig.14 and Fig.15 for the in-memory

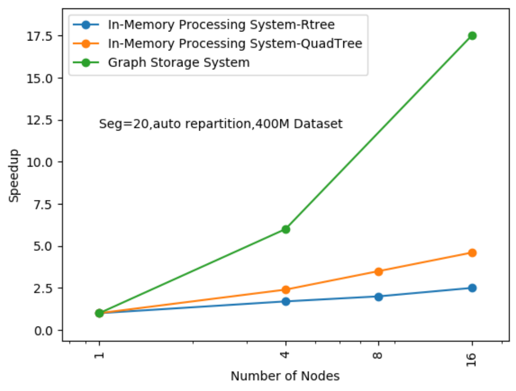


Fig. 13. Speedup Under Different Cluster Size

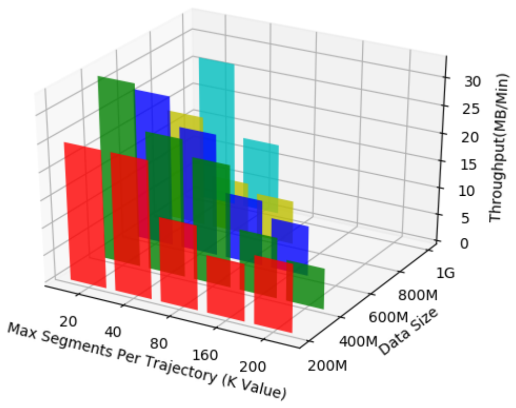


Fig. 15. 16-node clustering of the in-memory processing system

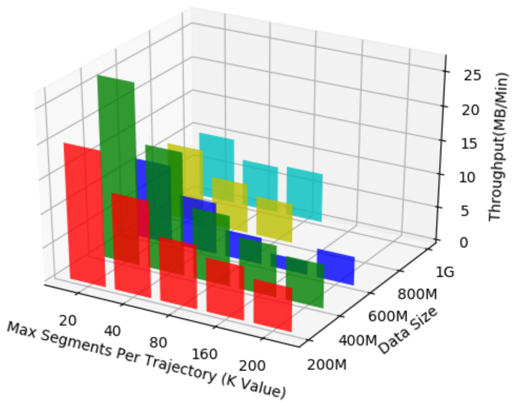


Fig. 14. 8-node clustering of the in-memory processing system

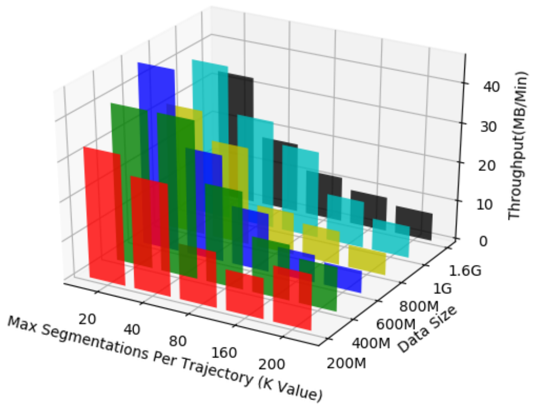


Fig. 16. 16-node clustering of the graph storage system

processing system. We notice some of the throughputs are missing because we failed these tasks for not enough memory reason. Doubling the cluster worker nodes from 8 nodes to 16 nodes improves throughput approximately 194% on average. For the graph storage system, we set K to 200 and use 16 worker nodes. Other configuration remains the same as the in-memory processing system.

9.2.2 Data Size Effect

The throughput trend is displayed in Fig. 17 when the data size increases. We set the K value (the maximum number of segmentation per trajectory) to 20. We observe the throughput of the in-memory processing system is as low as 50% when the dataset is larger than 1GB compared to 800MB dataset. Further scrutinizing the profiling logs, we observe the out-of-memory events due to garbage collection actions of Spark. More garbage collection occurs as the data input size increases. There are two factors causing this. One is the partition skewness. Another reason is the geometric data, especially the R-tree data structures in JVM are organized loosely. They occupy more than ten times of its original data size in memory. Organizing the R-tree structure efficiently in JVM is beyond the scope of this paper. For the graph data storage system, the throughput drops 37.5% when the data size increases from 1000MB to 1800MB.

9.2.3 Segmentation Effect

We increase the value of K (the maximum number of segmentation per trajectory) from 20 to 200. This increases the total number of MBRs. As shown in Fig. 18, the throughput drops by 29.6% and 21.9% on average when double the K value for the in-memory processing system and the graph storage system respectively. The degradation of throughput is caused by two factors. The first factor is the increased processing objects to parse or to shuffle when splitting a trajectory into more segments. The second factor is increasing the query stage execution time due to the R-tree capacity in Section 9.2.1.

9.2.4 Latency Decomposition

To further investigate the bottleneck, we observe the latency decomposition for the in-memory processing system shown in Fig. 19. It shows that R-tree indexing and join

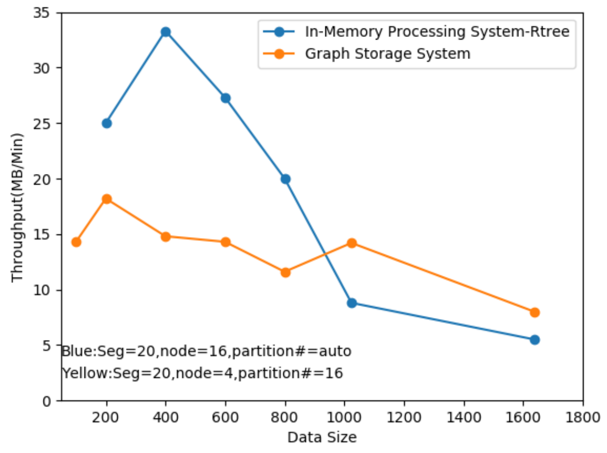


Fig. 17. Throughput Under Different Data Size

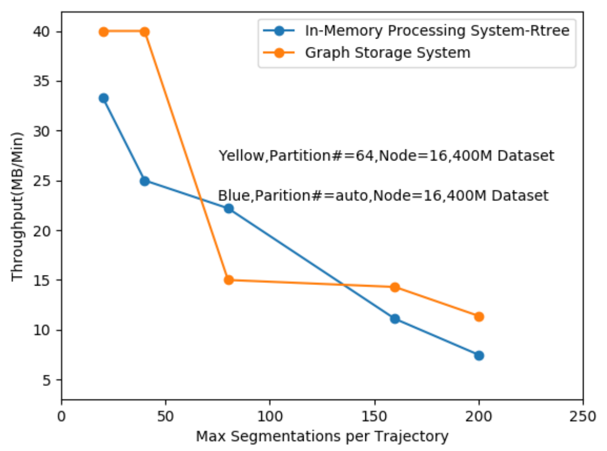


Fig. 18. Throughput Under Different Segments per Trajectory

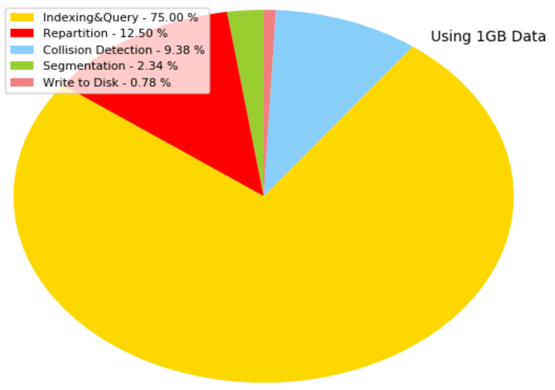


Fig. 19. In-memory Processing Framework Latency Decomposition

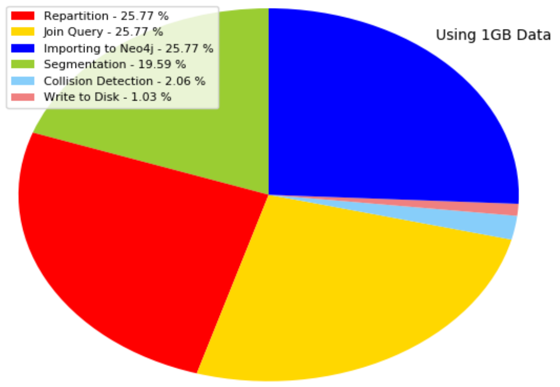


Fig. 20. Graph Database Based Framework Latency Decomposition

query accounts for 75% of the execution time, which is 1.5 hours. We cannot distinguish the R-tree building and query time due to the lazy loading strategy in each partition. After examining the execution log, we observe the garbage collection time taking over 17% of this stage. This is an indication of insufficient memory in the cluster. Following join query stage, the next most time consuming stages are repartition stage(Stage 3) and collision detection stage(Stage 7). The trajectory segmentation(Stage 1 and 2) only takes the proportion of 2.3 percent, which is 3 minutes in the 16-worker-node cluster.

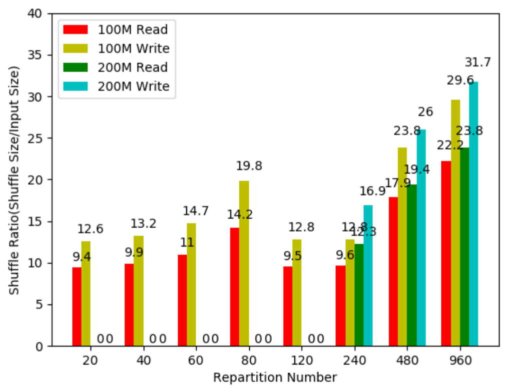
Since R-tree indexing and join query operations are at the stage 4 and stage 5 of the workflow, we further measure the *Shuffle Read Rate* and *Shuffle Write Rate* under varied partition numbers as shown in Fig. 21. When the number of partition grows from 480 to 960, the *Shuffle Read Rate* increases for 24.4% and the *Shuffle Write Rate* increases for 21.1% percent for 200M dataset. The observation indicates to what extent the data shuffling overhead effects to the latency of the indexing and joint query stages when increasing the partition numbers.

The latency decomposition of the graph storage system is shown in Fig. 20. In contrast to the in-memory processing system, no significant bottleneck stage occupies more than a quarter of the time. This indicates the graph storage system is efficient in scaling the workload.

10 Data Skew Analysis

Data skew is a phenomenon of non-uniform distribution of key values and tuples. The published analyses of joins in the presence of data skew indicate data skew curtail scalability [34] [35] [36]. Our above experiments indicate the data skew effects due to the partition and the indexing stages of workflows.

Both workflows of the in-memory processing system and the graph storage system in Fig. 4 have the partition assignment stage before data shuffling. Due to the R-tree partition limitation, the 1% sample MBRs cannot generate R-tree leaf grids covering all the spatial range of MBRs. R-tree leaves cover only portions of the whole range to be partitioned. The rest of MBRs are assigned to the ”overflow” partition. This causes the data skew problem.



!t

Fig. 21. Segmentation Repartition Shuffling Ratio

10.1 Replacing Partitioning Strategy

The experiments further measure the latency and the number of MBR records in RDD processed during the join query stage. Table 4 can give a insight of the partition data size distribution and the execution latency distribution among the tasks in query stage. For in-memory processing framework, the largest partition size (354, 524 records) is 145 times of median partition size (2, 450 records) in 1GB data input. In the meantime, the largest partition's execution time is 504 times than the median partition's execution time. The variance between partition record number suggests a severe data skew between partitions and the variance between task execution time indicates the garbage collection overhead takes too much time when processing the largest partition. This is a sign of lacking resources that the framework can not handle so much data in one single partition.

To solve this data skew issue, we replace the R-tree partition strategy to the Quad-tree partition strategy [37] within the in-memory processing framework. Unlike the R-tree partition, the Quad-tree partition has no overflow partition. The depth of a Quad-tree is adapted to the MBR density. The denser of MBRs in a defined spatial range, the deeper of the Quad-tree and the more partitions in this range.

Table 4 shows under 1GB data size, the largest partition generates 8, 878 records compared to 986 records for median partition, which is only approximately 8 times larger. Meanwhile, the maximum partition's processing time (53s) is only 25.5 times longer than the median partition's processing time. The lower variance results in a higher level of parallelism we mentioned in Section 9.2.1.

10.2 Introducing Time Dimension When Partitioning

We observe increasing the number of partitions incurs uneven distribution that leads to data skew and long tail of the processing time. Since the graph storage framework has file systems for data storage, it has sufficient capacity to hold larger size but fewer number of partitions. Our solution to reduce the number of partitions of a graph storage framework is introducing the time dimension as a new factor when partitioning data.

We repartition the MBRs in one geographical partition into multiple map layers by introducing time dimension.

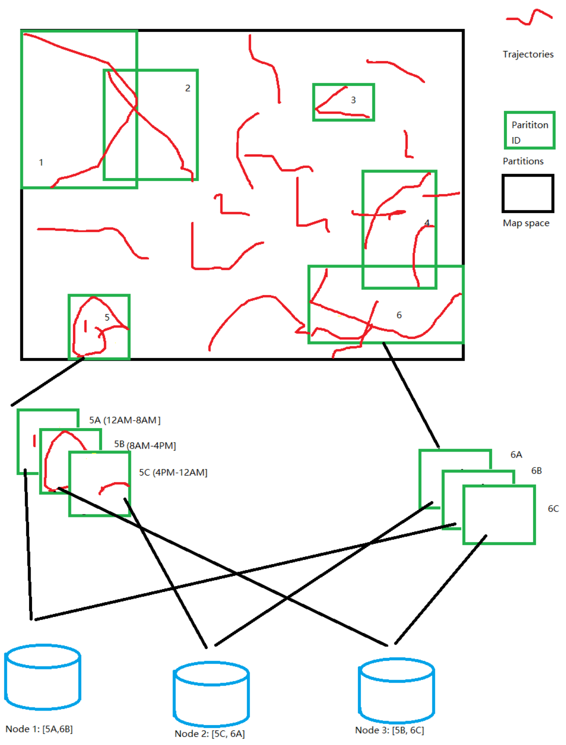


Fig. 22. Multiple Map Layers Routing to Neo4j Nodes

Each map layer is labeled to contain sub-trajectories occurring in a certain period of time. That can be a few minutes or several days depending on the density. The MBRs as a representation of sub-trajectories have the property indicating when the sub-trajectories start and end. We are able to dispense the MBR into certain map layer during the Stage 3. We follow the *replication* method in Section 5.1 Stage 3 when an MBR is between two map layers. The following stages treat each map layer in graph database as an individual partition in Spark system. Fig. 22 demonstrates the trajectory segments are routed to different map layers.

Table 4 shows the MBR distribution between partitions when we apply the R-tree and time dimension partition method in graph storage. Since the Neo4j graph database has a better ability handling large partition, we set the repartition number smaller than the in-memory processing framework. Under 1GB data size scenario, we observe the largest partition has 43, 180 MBRs compared to the median partition contains 41, 410 MBRs (increasing only 4% in size). The corresponding execution time is only 10% more than the median partition case. The less variance in the measurement indicates the mitigation technique is efficient in solving the data skew issue for the graph storage framework, thus produces scalable trajectory queries.

11 Conclusion

In this paper, we develop a distributed trajectory segmentation framework that transforms sequences of trajectories into queryable data blocks to build trajectory analysis applications. We design the system architecture and workflows to discover trajectory patterns using both distributed

TABLE 4
Tasks Latency and Records Distribution of the Join Query Stage

Partition Method	Data Size	25 %(Records)	Median(Records)	75 %(Records)	Max(Records)
In-memory (R-tree only)	200M	1 S(502)	3 S(1218)	12 S(3898)	1.5 Min(26772)
In-memory(R-tree only)	1G	3 S(1198)	10 S(2450)	27 S(4966)	1.4 H(354524)
In-memory(Quad tree)	200M	0.1 S(138)	0.5 S(344)	1 S(1050)	17 S(7206)
In-memory(Quad tree)	1G	0.2 S(182)	2 S(986)	5 S(2100)	53 S(8878)
Graph storage (R-tree&Time Dimension)	200M	36 S(6408)	41 S(6266)	43 S(6657)	48 S(7208)
Graph storage (R-tree&Time Dimension)	1G	5.7 Min(40114)	6.4 Min(41410)	7 Min(42369)	7.7 Min(43180)

in-memory processing framework and a cluster of graph database nodes. To boost the performance, we evaluate the effects of data partition, parallelism and the workload size on the system. We observe the bottleneck to higher scalability is caused by data screw. Accordingly, we propose a balancing method based on the R-tree index to adjust individual partition size and thus balance the data distribution. We also evaluate a better partition method when facing a large amount of data. For our future work, we aim to extend this framework with other indexing methods and increase the redundancy.

REFERENCES

[1] A. R. Jiménez, F. Seco, J. C. Prieto, and J. Guevara, "Indoor pedestrian navigation using an ins/ekf framework for yaw drift reduction and a foot-mounted imu," in *2010 7th Workshop on Positioning, Navigation and Communication*, March 2010, pp. 135–143.

[2] W. Xu, N. R. Juri, A. Gupta, A. Deering, C. Bhat, J. Kuhr, and J. Archer, "Supporting large scale connected vehicle data analysis using hive," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 2296–2304.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[4] 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. New York, NY, USA: ACM, 2015, pp. 70:1–70:4. [Online]. Available: http://doi.acm.org/10.1145/2820783.2820860

[5] Y. Xian, Y. Liu, and C. Xu, "Parallel gathering discovery over big trajectory data," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 783–792.

[6] Y. Zheng, H. Fu, X. Xie, W.-Y. Ma, and Q. Li, *Geolife GPS trajectory dataset*, July 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/

[7] K. Huaqiang. Spark trajectory processing repository. [Online]. Available: http://github.com/kanghq/SparkApp

[8] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. Keogh, and P. S. Yu, "Global distance-based segmentation of trajectories," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150411

[9] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 109–120.

[10] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods," *IEEE Data Eng. Bull.*, vol. 26, no. 2, pp. 40–49, 2003.

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

[12] D. Kwon, S. Lee, and S. Lee, "Indexing the current positions of moving objects using the lazy update r-tree," in *Proceedings Third International Conference on Mobile Data Management MDM 2002*, Jan 2002, pp. 113–120.

[13] R. C. Nelson and H. Samet, "A population analysis for hierarchical data structures," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '87. New York, NY, USA: ACM, 1987, pp. 270–277. [Online]. Available: http://doi.acm.org/10.1145/38713.38744

[14] H. D. Chon, D. Agrawal, and A. E. Abbadi, "Storage and retrieval of moving objects," in *Mobile Data Management*, K.-L. Tan, M. J. Franklin, and J. C.-S. Lui, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 173–184.

[15] Z. Fu, W. Hu, and T. Tan, "Similarity based vehicle trajectory clustering and anomaly detection," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 2. IEEE, 2005, pp. II–602.

[16] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory pattern mining," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 330–339. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281230

[17] C. Panagiotakis, N. Pelekis, I. Kopanakis, E. Ramasso, and Y. Theodoridis, "Segmentation and sampling of moving object trajectories based on representativeness," *IEEE Trans. on Knowl. and Data Eng.*, vol. 24, no. 7, pp. 1328–1343, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2011.39

[18] J. D. Vitek, J. R. Giardino, and J. W. Fitzgerald, "Mapping geomorphology: A journey from paper maps, through computer mapping to gis and virtual reality," *Geomorphology*, vol. 16, no. 3, pp. 233 – 249, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169555X96800031

[19] J. L. Zêzere, S. Pereira, A. O. Tavares, C. Bateira, R. M. Trigo, I. Quaresma, P. P. Santos, M. Santos, and J. Verde, "Disaster: a gis database on hydro-geomorphologic disasters in portugal," *Natural Hazards*, vol. 72, no. 2, pp. 503–532, Jun 2014. [Online]. Available: https://doi.org/10.1007/s11069-013-1018-y

[20] R. Laurini, "Spatial multi-database topological continuity and indexing: a step towards seamless gis data interoperability," *International Journal of Geographical Information Science*, vol. 12, no. 4, pp. 373–402, 1998. [Online]. Available: http://dx.doi.org/10.1080/136588198241842

[21] S. Orlando, R. Orsini, A. Raffaetà, A. Roncato, and C. Silvestri, "Trajectory data warehouses: Design and implementation issues," *Journal of computing science and engineering*, vol. 1, no. 2, pp. 211–232, 2007.

[22] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1071–1085. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915237

[23] 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.

[24] 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, pp. 1565–1568, Sep. 2016. [Online]. Available: https://doi.org/10.14778/3007263.3007310

[25] R. Sriharsha, "Geospatial processing made easy," Jul 2017. [Online]. Available: https://magellan.ghost.io/magellan-geospatial-processing-made-easy/

[26] C. Strobl, *Dimensionally Extended Nine-Intersection Model (DE-9IM)*. Cham: Springer International Publishing, 2017, pp. 470–476.

[27] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, "A trajectory splitting model for efficient spatio-temporal indexing," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 934–945. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083700

[28] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh, "Indexing multi-dimensional time-series with support for multiple distance measures," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 216–225. [Online]. Available: http://doi.acm.org/10.1145/956750.956777

[29] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatialhadoop," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1602–1605, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824057

[30] C. Taverner, "Neo4j spatial," Jul 2018. [Online]. Available: https://github.com/neo4j-contrib/spatial

[31] R. Mehran, A. Oyama, and M. Shah, "Abnormal crowd behavior detection using social force model," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 935–942.

[32] Y. Zheng, N. J. Yuan, K. Zheng, and S. Shang, "On discovery of gathering patterns from trajectories," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 242–253. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2013.6544829

[33] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=3001460.3001507

[34] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213840

[35] Y. Kwon, K. Ren, M. Balazinska, and B. Howe, "Managing skew in hadoop," *IEEE DATA ENG. BULL., VOL*, 2013.

[36] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *2012 IEEE 28th International Conference on Data Engineering*, April 2012, pp. 522–533.

[37] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, Jun. 1984. [Online]. Available: http://doi.acm.org/10.1145/356924.356930

Huaqiang Kang Huaqiang Kang is a graduate student at Concordia University. He is pursuing his Master degree since 2016. Previously he worked at Nokia Solutions and Networks as a database engineer. His main research interest includes database, large scale data processing and deep learning.

Yan Liu Yan Liu is an Associate Professor in Faculty of Engineering and Computer Science, Concordia University. Dr. Liu has over 15 years research experience of developing data intensive algorithms on distributed and parallel computing systems. Before her faculty position, she was a senior research scientist in Pacific Northwest National Laboratory (PNNL) in Washington State, delivering high performance and scalable data analysis platforms for domains of power systems, scientific computing and engineering simulation. Her recent research focuses on parallel and distributed machine learning, and automatically scaling back-end computing resources also by means of machine learning.

Weishan Zhang Weishan Zhang is a full professor at Department of Software Engineering, College of Computer and Communication Engineering, China University of Petroleum. His main research interests cover big data intelligent processing, cloud computing, Internet of Things, and software engineering. He has published over 100 papers, and his current h-index is 15 and i10 index is 27 as in Sept. 2018.