A FRAMEWORK FOR SPATIO-TEMPORAL TRAJECTORY DATA SEGMENTATION AND QUERY

Huaqiang Kang

A thesis
in
The Department
of
Electrical and Computer Engineering(ECE)

Presented in Partial Fulfillment of the Requirements
For the Degree of Mast of Applied Science(MASc)
Concordia University
Montréal, Québec, Canada

March 2019

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Huaqiang Kang**

Entitled: **A Framework for Spatio-Temporal Trajectory Data Segmentation and Query**

and submitted in partial fulfillment of the requirements for the degree of

Mast of Applied Science(MASc)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Wahab Hamou-Ljhad

_____ Examiner
Dr. Wahab Hamou-Ljhad

_____ Examiner,
External to the Program
Dr. Tristan Glatard (CSSE)

_____ Supervisor
Dr. Yan Liu

Approved _____
Dr. William E. Lynch
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dr. Amir Asif Dean,
Gina Cody School of Engineering and Computer Science

Abstract

A Framework for Spatio-Temporal Trajectory Data Segmentation and Query

Huaqiang Kang

Trajectory segmentation is a technique of dividing sequential trajectory data into segments. These segments are building blocks to various applications for big trajectory data. 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 thesis, a distributed trajectory segmentation framework that includes a greedy-split segmentation method is created. This framework consists of distributed in-memory processing and a cluster of graph storage respectively. For fast trajectory queries, distributed spatial R-tree index of trajectory segments is applied. Using the trajectory indexes, this framework builds queries of segments from in-memory processing and from the graph storage. Based on this segmentation framework, two metrics to measure trajectory similarity and chance of collision are defined. These two metrics are further applied to identify moving groups of trajectories. This study quantitatively evaluates the effects of data partition, parallelism, and data size on the system. The study identifies the bottleneck factors at the data partition stage, and validate two mitigation solutions. The evaluation demonstrates the distributed segmentation method and the system framework scale as the growth of the workload and the size of the parallel cluster.

Acknowledgments

I would like to thank my thesis supervisor Dr. Yan Liu of the Department of Electronic and Computer Engineering. She gave me the chance to focus on what I am interested and pursuit it. It is with her supervision that this work came into existence. For any faults I take full responsibility.

Nobody has been more important to me in the pursuit of my Masters than the members of my family. I also wish to thank my parents for their support both financially and emotionally. Their encouragement and love give me the faith to face the difficulties, whose love and guidance are with me in whatever I pursue.

Finally I would like to thank my friend Zach who encouraged me throughout the time of my research and my life.

Contents

List of Figures

List of Tables

Chapter 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 wearable devices, vehicles to study their behaviors, track vehicle's routes, and produce diverse applications including points of interests, recommendation, health monitoring safety regulation, and fleet management.

1.1 Problem Statement

GPS raw data are usually recorded as spatio-temporal points. For example, an IMU sensor [32] 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 [70]. As the scale of managed cars expands dramatically, the trajectory data becomes a source of Big Data. Processing of trajectory data timely or in real-time inherits Big Data challenges.[61] When the data size is beyond the capacity of a single processing node or the processing latency has a higher requirement shorter than a single node processing time, the trajectory data are required to be distributed and processed in more salable multiple nodes. [29] [30] For self-join query operation, the time complexity is $O(N^2)$. Partitioning the data site into smaller set can significantly reduce the time cost of each single query. Should the framework distribute the data based on file objects or based on geolocation? If geolocation partitioning is considered, which strategy is the best practice to handle

the data density varying from urban to rural? [16]

After data are distributed, how to express the trajectories is a big challenge. Is there any possibility to transform a set of points into more efficient geometry shapes? How the system organizes the shapes and utilizes any index to optimize the queries is another challenge. After the transformation, to what extent of the shape characteristic of the trajectory is preserved during trajectory similarity search? Based on the index and segmentation, there should be an algorithm to evaluate the trajectory similarity and finding the most one out. [23] [42]

In this thesis, the study focuses 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: When trajectory segments are queried for analysis, how much performance the result gains in cluster processing compared to the single node processing of trajectories.

1.2 Objective

The objective of this thesis is to develop a framework that can parallel process the trajectory into segments and distribute the trajectories into multiple nodes using data partitioning strategies. The load and transformation should be parallel to increase the efficiency. There should also be a parallel trajectory query framework that enables intersection query to get the desired trajectory results. The framework should ensure the low latency in the streaming processing scenario and it can also adapt to the data warehouse's large quantity of data case. The framework can be elastic to adapt to the size of data to be processed and the latency requirement.

Furthermore, trajectory metrics could be developed based on these queries for further analysis model. Last but not least, we should have a story in real data to demonstrate the ease of use in this framework and track down the bottleneck and data skew of the system to avoid this when put it into practice.

1.3 Methodology

In this thesis, it requires a parallel trajectory segmentation method that scales horizontally as the size of the trajectory data, the load of queries and the number of worker nodes grow. This method consists of a split algorithm for parallel trajectory partition and workflows for queries of segments for trajectory analysis. This workflow is to be implemented to investigate the parallelism factors of scalability using two frameworks, one is distributed in-memory processing and the other is NoSQL graph storage. Spatial indexes and query operations in each framework are built.

To realize this workflow, this study further requires a data model for representing trajectory segments and associated geometric operations. A long trajectory should be transformed into small segments and then being warped with Minimum Bounding Rectangles(MBRs). For the lowest latency, the MBRs as well as the indexes of the MBRs is going to be stored in each cluster node's memory. On the other hands, when the data exceed the size of cluster memory,geometric objects are indexed and stored in a NoSQL graph database as an alternative.

To demonstrate the usage of the proposed method, two metrics that are integral to applications such as trajectory clustering analysis are defined. One metric estimates the similarity of trajectories. This study also defines a threshold of Euclidean distances of trajectories to count if any moving objects are within this threshold at a certain period. The other metric detects the collision chance by measuring the intersections of two trajectories.

This research evaluates the method proposed by two means, (1) system-level performance evaluation and (2)comparison of results from the trajectory clustering workflow with another clustering method. For the system level performance evaluation, workflows on the Amazon Elastic MapReduce (EMR) platform are developed. The trajectory data are from Microsoft GeoLife [74] data with the size of 1.6GB and 24 million moving object records. The experiments show the performance on GeoSpark has an improvement speedup ranging from 2 to 2.5 with 8 node cluster or 16 cluster compared with the single node system. The evaluation on Neo4j framework has a maximum speedup of 17.5 times improvement when expanding the cluster size from 1 node to 16. In term of accuracy of clustering results, it reaches 87.2% accuracy compared to GPFinder [68] as ground truth.

Through this research study, it identifies that the bottleneck is data skew due to

geographic imbalance. A dynamic partitioning method to adjust each partition's load of objects is adopted. Furtherly a data skew mitigation solution by involving other non-geographical property as secondary key when distributing data is devised.

1.4 Contributions

The main contribution of the thesis is four-fold. The implementation is accessible from Github-*https://github.com/kanghq/SparkApp* [31].

1. We firstly apply the Greedy-Split trajectory segmentation algorithm **with MapReduce** programming model on a distributed system;
2. A system workflow that queries trajectories segments **in-memory** and **in a graph database** is developed. Integration indexing techniques for fast queries is also performed.
3. This study **defines metrics** that are further utilized by applications such as trajectory clustering analysis
4. **A data skew migration solution** is developed to balance the workload as both the size of the data and the computing cluster grow.

1.5 Thesis Structure

The thesis follows this structure:

- Chapter 2 introduces the related works of trajectory segmentation, storage, and access.
- Chapter 3 presents the framework system architecture, segmentation, indexing and query flow.
- Chapter 4 illustrates a real life use case for evaluating.
- Chapter 5 shows the experiments results.
- Chapter 6 discusses the data skewness, evaluating reliability, and validity.

Chapter 2

Background and Related Works

2.1 MapReduce Model

Previously, the parallel computing is restricted to parallel algorithms. Traditional parallel algorithms include dense matrix algorithms, sorting, graph algorithm, search algorithms, dynamic programming, and fast Fourier transform [36]. This limits the usage of high-performance parallel computers. In 2008, Google proposed MapReduce programming model to make it possible for large datasets in parallel processing. This model can be described in five stages.1) `mapping` input as `<Key, Value>` pair. 2) computing on single `<Key, Value>` record. 3) grouping intermediate data. 4) computing on intermediate data. 5) `reducing` for final result. Figure 1 gives an intuitive impression on MapReduce model.

MapReduce provides a simple and universal parallel programming model. However, there are also cons. There are a large number of algorithms can not be rewritten to MapReduce model. Not like SQL, it is a lower level language. You have to focus on how to retrieve data and how to sort these. Recently, there are some middlewares like Hive [70] providing similar SQL language to manipulate data as well as the indexing support. Another drawback is its low efficiency. It enables only single dataflow in the whole framework. The stages between map and reduce are block operations. The shuffling stage consuming large of I/O also needs attention when programming.

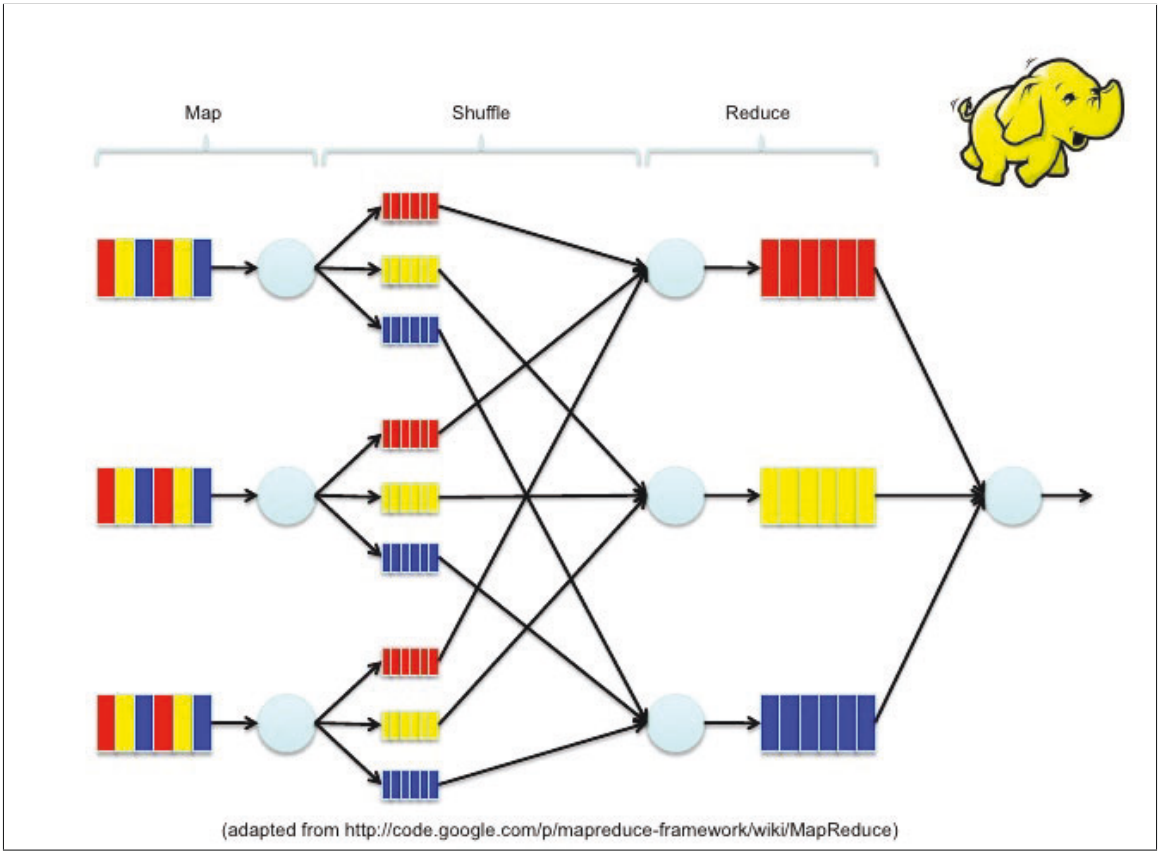


Figure 1: How Map Reduce works

2.2 NoSQL Database

2.2.1 NoSQL Fundamental Concepts

NoSQL or Not Only SQL database emerges at the background of big data analysis. The appearance of NoSQL databases aims to solve the following weakness that traditional RDBMS can not handle.

Here are the motivations of creating NoSQL databases [59]:

- To avoid unnecessary ACID complexity;
- Giving high throughput in big data analysis;
- Ability of horizontal expatiation on non-dedicated servers;
- More flexibility than database norms;
- Low administration and setting up cost.

2.2.2 Imperfect NoSQL

NoSQL is not a replacement of DBMS. However, it is more an alternative when the data are more flexible where it is hard to fit for the RDBMS. In traditional RDBMS, all data fields should be defined property with constraints. RDBMS can provide the maximum robust data integrity. The simple SQL query method makes optimization easy and reliable.

For this system, the choice of graph database gives the system more flexibility in the query of graph theory algorithms. For example, as a graph database, Neo4j provides some path finding algorithms, community detection like *Louvain algorithm* [6] or *connected components algorithm*.

2.3 Open GIS Support

Spatial topology refers to the relationship between spatial objects. Applying topology in a GIS system has three benefits.

- Topology is necessary for route planning. Without topology, it is impossible to route to a certain destination via the road network.
- Topology can be used to validate data for better data quality. For example, a utility hole should be outside polygon objects where the shape of roads are represented.
- By creating the topology relationship between features and objects, it is possible to synchronize the features to make them consistent.

This framework imports a third party library to support spatial topology called *JTS Topology Suit*. It's a group of core APIs for processing geometry. The UML chart is shown in Figure 2 [14]. With JTS library, it is possible to read standard WKT or WKT format shapes, building indexes, to query desirable objects and to compute metrics.

It has complete 2-D linear geometry model supporting *Point*, *LineString*, *LinearRing*, *Polygon*, *MultiPoint*, *MultiLineString*, *MultiPolygon*, *GeometryCollection*. JTS follows OpenGIS API standards, and follows Dimensionally Extended nine-Intersection Model(DE-9IM) [60], which means you can compute the spatial relationships with the predicates like *intersects*, *contains*, *within*, *equanls*, *disjoint*, *touches*,

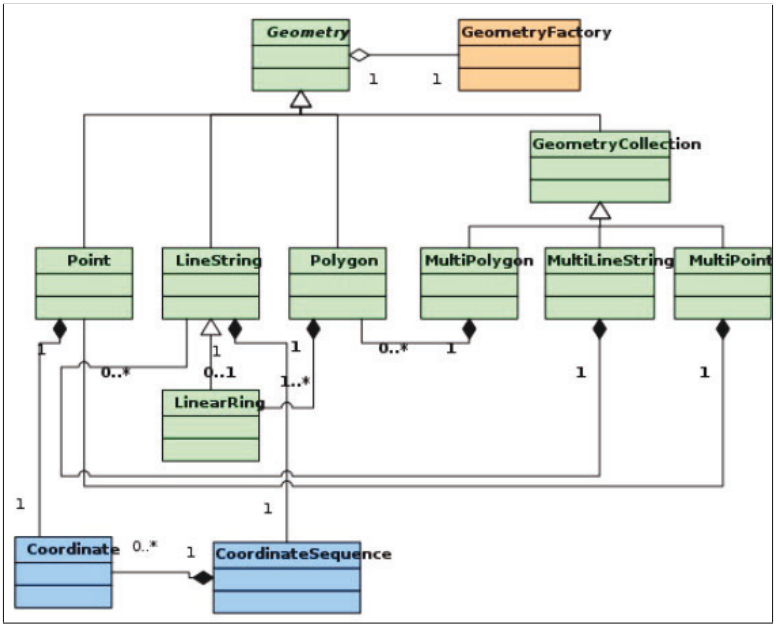


Figure 2: JTS UML Chart

crosses, *overlaps*, *covers*, *coveredBy*. These 9 intersection relationships are shown in Figure 3.

There are four overlay methods in JTS. They are *intersection*, *union*, *difference*, *symmetric difference* as heterogeneous overlay.

The precision Model provides floating and fixed coordinate models. It can give different capacities putting points in the grid.

JTS also provides the metrics to measure the spatial objects including area, length, distance, WithinDistance and Hausdorff Distance.

The supported spatial indexes are Quadtree,StRtree, kD-tree, Bintree, MonotoneChains, and SweepLine.

2.4 Algorithms and Queries

Trajectories need to be simplified by cutting into smaller, less complex primitives. Anagnostopoulos et al. [4] illustrate a method of segmentation that adapts to Nearest-Neighbor search and analyzes the segmentation problem in a global view. Cudre-Ma uroux et al. [13] give a solution for large size of trajectories on disk. They maintain an optimal index and the data are dynamically co-located. To reduce the I/O, the system also adapts to queries for optimization. Mokbel et al. [45] show us three major

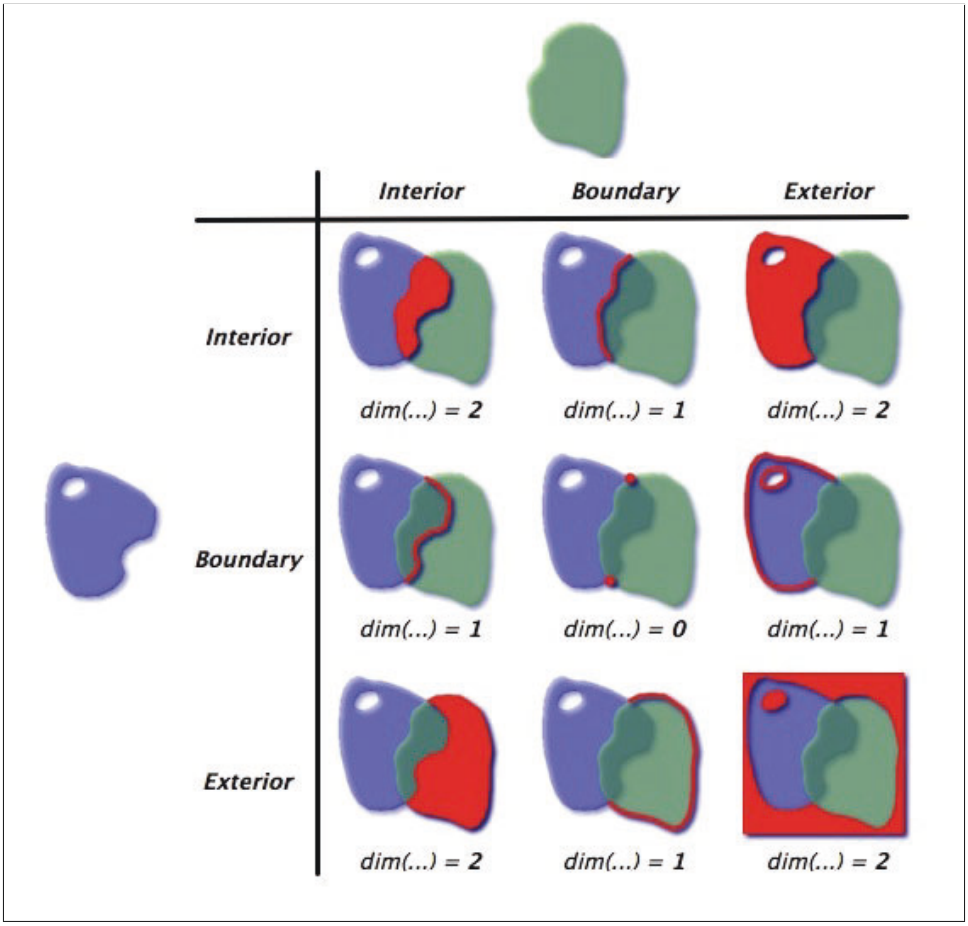


Figure 3: Modelling Object Interactions. [7]

spatio-temporal access methods, namely "Indexing the Past", "Indexing the Current Positions" and "Indexing the Current and Future Positions". The mostly used one is indexing the past positions, such as STR-tree or RT-tree [27]. Another method commonly used is to index the current positions, such as LUR-tree [37] and Hashing. The third method of indexing is to index the current and future positions, including PMR-quadtree [49] and SV-Model [11]. A new trend of accessing trajectories is using parametric rectangles [11]. They do not enclosure the trajectories directly; they create the bounding rectangles as a function of time. The moving objects would be in the same rectangles for a time instance t.

For motion classification, Fu et al. proposed a similarity-based pattern grouping method compared with fuzzy K-means [24]. Giannotti et al. [25] also presented 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 used a dense based discretization method to find out the popular regions. The authors Panagiotakis et al. [52] introduce a methodology to find out the most representative sub-trajectories. They represent the trajectories based on multiple attributes and then sampled them without supervision.

2.4.1 R-tree Building and Query

R-tree is widely used both in partitioning technique and indexing method. R-tree is a depth-balanced tree, so the root should have at least two children to be balanced. The children number of a node (except leaf or root) is between m and M, where $m \in [0, M/2]$. M is the max number of children of a node. The depth d of an R-tree:

$$\lfloor log_m N - 1 \rfloor < d < \lfloor log_M N - 1 \rfloor$$

.

Algorithm 2 gives an example `query(root,q)` to find object q from root of R-tree.

Algorithm 1 Algorithm for R-tree Query query(u,r)

Input: a query object r wrapped by MBR mbr_r, search root entry u.
Output: objects overlapped with r.

```
1: if u is a leaf then
2:    return all objects overlapped with r
3: else
4:    for each each child v in node u do
5:       if mbr_v overlapped with r then
6:          query(v,r)
7:       end if
8:    end for
9: end if
```

The time complexity of search can be considered into two conditions: 1) if Bounding boxes do not overlap the query object q, the complexity is $O(log_m N)$. in the worst case, when all objects' bounding boxes are overlapping on q, it is $O(N)$ [63].

Given an object p to be inserted, the study illustrates the `insert(root, p)` algorithm to show how R-tree is built.

For the `choose-subtree(u,p)` function in Algorithm 3, the aim of this function is to reduce the volume growth when adding a new object p. When the new leaf exceeds

Algorithm 2 Algorithm for R-tree Insertion insert(u,p)

Input: a new object p wrapped by MBR mbr_p, R-tree root entry u.

```
1: if u is a leaf then
2:    add p in node u
3:    if u overflows then
4:       handle-overflow(u)
5:    end if
6: else
7:    v := choose-subtree(u,p)
8:    insert(v,p)
9: end if
```

its capacity during the inserting, it triggers handle-overflow shown in Algorithm 4 to split one node into two to ensure the tree is balanced.

Algorithm 3 Algorithm for R-tree sub tree choosing choose-subtree(u,p)

Input: root entry u, new object p to be inserted.
Output: the

```
1: for v_i as one of the child of root u do
2:    vol_i := volume(mbr_{v+p} - mbr_v)
3: end for
4: get the smallest vol_k
5: return v_k
```

Algorithm 5 illustrates the split operation in R-tree.

For a good split solution, there are two standards to evaluate it. 1) the total area of the two nodes is minimized and the overlapping of the two nodes is minimized. Study shows that the complicity of finding an optimized split solution is $O(2^{M+1})$. Figure 4 is a typical R-tree structure, where $M = 3$, $m = 2$.

2.4.2 Quad-tree Building and Query

Quad-tree is another hierarchical spatial data structure. It is a rooted tree and each node has a fixed number of 4 children. Each node expresses a square area in the space and each child of this node expresses one quadrant of the space. It's a non-balanced tree and can be used to express non-uniform meshes. Algorithm 6 shows how to insert a point p from a Quad-Tree with $insert(root, p)$ function. From the algorithm, each leaf contains one object most. It is easy to be very unbalanced when points lie close

Algorithm 4 Algorithm for R-tree overflow handling handle-overflow(u)

Input: root entry u.

split(u) into two parts u and u'
if u is the root **then**
  create a new root and connecting u and u'
else
  w:= parent(u)
  update w := new MBR(u)
  add new child u' to w
  if w overflows **then**
    handle-overflow(w)
  end if
end if

Algorithm 5 Algorithm for R-tree split split(u)

Input: root entry u, parameter $\beta \geq 3$.
Output: two new child MBRs mbr_{s1} and mbr_{s2} covering u

m := size of objects in u
sort objects under u in x-dimension
for i := $\lceil 0.4\beta \rceil$ to $m - \lceil 0.4\beta \rceil$ **do**
  S_1 := first i objects in list
  S_2 := the other i objects in list
  get mbr_{S_1} and mbr_{S_2}
end for
repeat 2-6 line with the respect of another dimension
return mbr_{S_1} and mbr_{S_2} with the best solution

together. The depth of a Quad-tree

$$d = log(s/c) + 3/2$$

where s is the initial square length and c is the smallest distance between two points. It has $O((d+1)n)$ nodes and the construction time complexity is $O((d+1)n)$ too [46].

In the case of rectangles, it is resembled that the point as zero height zero width polygon. During the insertion, it should make sure each cell is not big enough to fit the whole polygon and the polygon does not need to be stored in the leaf node.

2.4.3 DTW for Trajectory Similarity

Dynamic Time Wrapping (DTW) [5] algorithm is firstly introduced for speech recognition. Then it is used to time series analysis applications. It can be used to compare

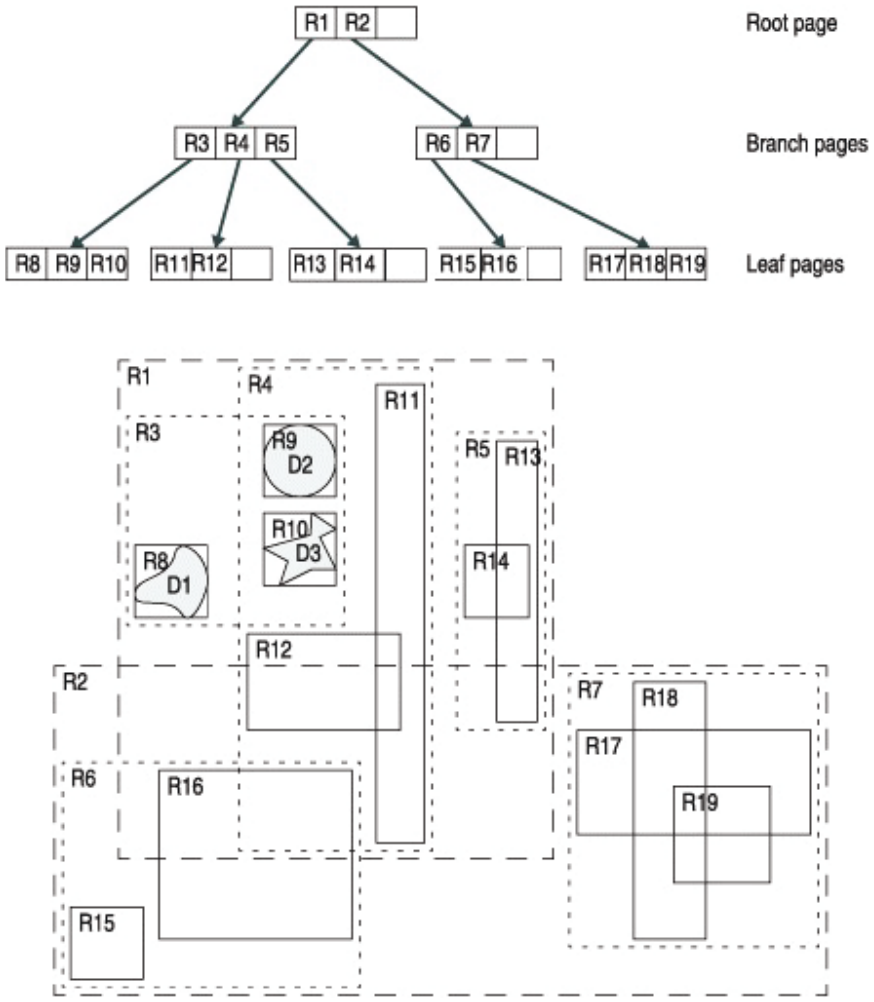


Figure 4: An R-tree Structure [2]

two trajectories with different length. To find out the mating points, the pairwise Euclidean distance matrix should be prepared first. A path satisfies "monotonic" and "continuity" from bottom left to top right makes the alignment between sampling nodes.

It should be noticed that the DTW algorithm only compares the sampled "points", not the trajectory. So the GPS sampling interval or video stream frame rate may interfere the accuracy between two trajectories. Also, DTW origins from time series algorithms, which lacks the consideration of temporal attribute [43].

Algorithm 6 Algorithm for Quad-Tree Construction insert(u,p)

Input: root entry u, point to be inserted p

```
1: if u's boundary not contain p then
2:    return false
3: end if
4: if  v is empty then
5:    add p to cell v
6:    return true
7: else
8:    subdivide NW, NE, SW, SE four quadrants
9:    if insert(NW,p) then
10:      return true
11:   end if
12:   if insert(NE,p) then
13:      return true
14:   end if
15:   if insert(SE,p) then
16:      return true
17:   end if
18:   if insert(SW,p) then
19:      return true
20:   end if
21:   return false
22: end if
```

2.5 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. The first one is raster-based and the second one is vector-based [66]. The object is represented as a series of lines connected to form a polygon. The RDBMS can easily store the coordinates of vertices. In the raster-based system, the real world object is formatted by cells and represented as a series of contiguous cells. ESRI company develops a spatial database engine providing the middle layer to store GIS data in RDBMS like DB2, SQL Server or Oracle. It is also popular to use geometry database to store this geological information. DISASTER [73] is a Portuguese GIS database based on the most popular open source MySQL database engine. It stored floods and landslides for the period of the year 1865 to the year 2010. In [40], they have developed mechanisms to integrate multiple data sources and finally

a seamless database was achieved. A data warehouse supporting streaming data was designed [51]. The data warehouse supports trajectory properties such as average velocity, maximum acceleration as well as aggregation operations.

2.5.1 RDBMS SQL Server 2008 Spatial Indexing

This study uses SQL Server as an example to introduce how traditional database handles spatial data. SQL Server has built-in geometry support. "GEOMETRY" and "GEOGRAPHY" types express points, lines, polygons or multi-polygons. The expression format can be Well-Known Text (WKT), Well-Known Binary (WKB) as well as GML [12].

In traditional RDBMS like SQL Server, it utilizes B-tree to achieve the support of 2-dimensional spatial data. The entire space is decomposed into a grid hierarchy. The cells are numbered in Hibert space-filling curve. There are four levels grid hierarchy and each level can be configured as *HIGH*, *MEDIUM* or *LOW* density divisions to decide the density of cells per layer. If a cell is contained in an object, it is not tessellated further. If an object is covered by multiple cells, the database records these cells respectively. Figure 5 is an example of SQL Server indexing [20].

2.6 Processing Framework

For distributed spatial analytic systems, they can be divided into two camps. The one is Hadoop based systems that store the intermediate data in the shared disk system. The other one is Spark based, which processes data in memory only.

2.6.1 Simba

Simba [69] 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. Multiple varieties of space operations like *kNN Query*, *kNN join distant join* are supported in this framework. It reconstructs the fundamental RDD architecture to *IndexRDD*. It is possible to have the indexes persisted on disk and loaded back but it does not support full data disk persistence.

An important feature for Simba is that it provides SQL planer. With the SQL

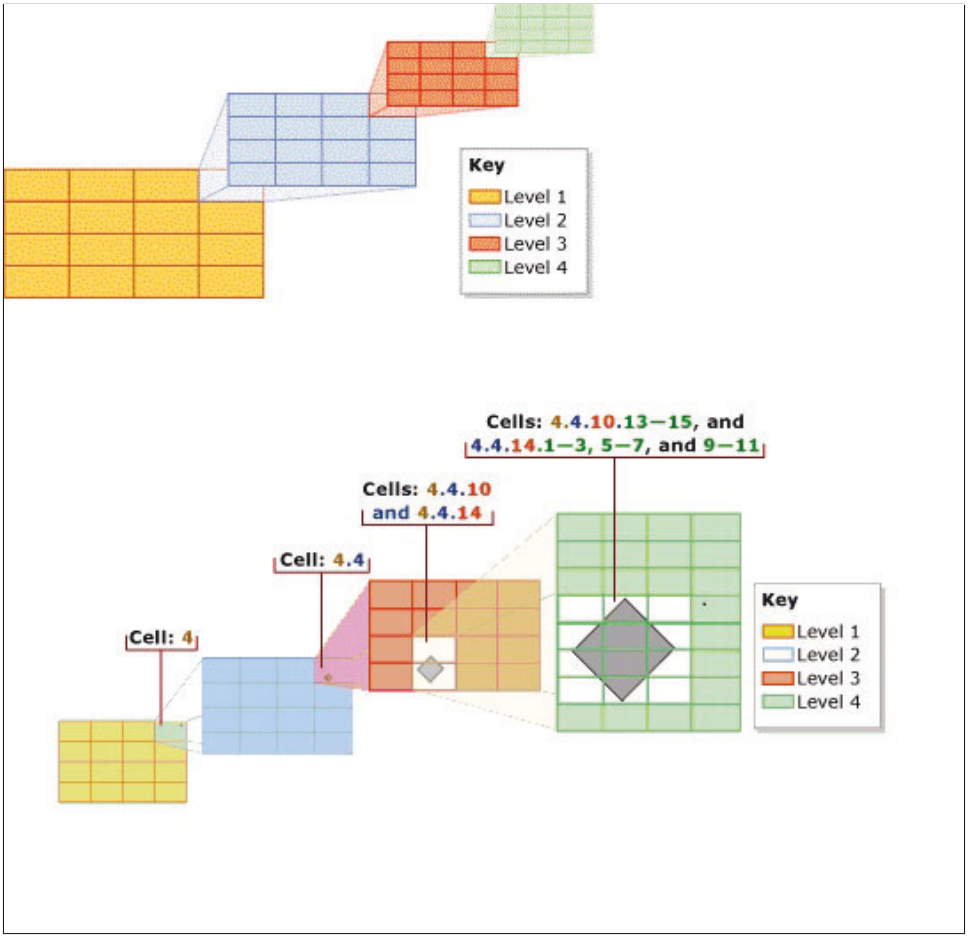


Figure 5: SQL Server Indexing [1]

planer, SQL can be used as input and the optimizer in SQL planer can make the best use of existing indexes and statistics. This planer is based on *optimization rules* and *cost-based optimizations*.

Simba has two-level indexing strategy. The first level index gives quick access to the partition where the spatial object belongs; the second level R-tree indexes optimize the spacial operations like range query, kNN join or distance join. Simba supports concurrent query execution by deploying a thread pool in the query engine. This is a platform level concurrency strategy that does not need the involvement of users.

As a cluster based system, Simba has the ability of fault tolerance inherited from Spark. When a master fails in the multiple masters environment, recovery mechanism ensures the global indexes are not missing. Also the query job in Simba triggering Spark transformation job can be guaranteed to recover from failure with the help of

Spark native design.

2.6.2 SpatialHadoop

SpatialHadoop [15] is another distributed processing framework based on Hadoop. It provides native support of spatial data. Not like Hadoop-GIS [3] treats the Hadoop framework as a black box; Spatial Hadoop realizes *range query*, *kNN query*, and *spatial join* functions which enable the user to develop a higher level application.

In SpatialHadoop, there are four layers: **language layer**, **operation layer**, **MapReduce layer**, and **storage layer**. In language layer, SpatialHadoop supports SQL-like scripting by applying Pig Latin [50] extension. To support spatial data, Pegeon [17] is also integrated into SpatialHadoop. Casual users can directly do Ad-hoc query with SQL-like scripts. In Operation layer, the spatial operations are encapsulated for developer use. Higher level functions can be expanded in SQL-like scripts. In MapReduce layer, SpatialHadoop involves *SpatialFileSplitter* to split input file by blocks so that the indexes can be built up efficiently. *SpatialRecordReader* transforms spatial data into key-value pairs, extracts indexes and sends the spatial data to map function in blocks. Lastly, the storage layer provides grid partition file storage and R-tree or R+-tree support. It archives up to 4.6 TB data processing in the prototype test.

2.6.3 Others

LocationSpark [62] and Magellan [58] 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. Table 1 lists the features of different frameworks.

Table 1: Comparison Between Different Spatial Processing Frameworks.

Features	GeoSpark	Simba	SpatialHadoop	Magellan	LocationSpark	**This Framework**
Data Dimensions	2	Multiple	2	2	2	**3**
Spatial Indexing	R-tree/Quad-tree	R-tree	Grid/R-tree	ZOrderCurves	Grid/R-tree/Quad-tree	**R-tree**
In-memory	Yes	Yes	No	Yes	Yes	**Yes**
SQL	No	Yes	Yes	No	Yes	**No**
Data Persistence	Index Only	Index Only	No	No	No	**Yes**

2.7 Distributed Parallel Data Analysis System

2.7.1 Overview

While data size exceeds the capacity of a single machine, especially in today's Web 2.0 era, a new way to share the resource between multiple computing nodes are required. **Distributed computing** consists of software components, hardware components as well as network components. Compared to the centralized system, there are many benefits that distributed system can offer:

- Scalability: a centralized system can be limited by the microelectronics to increase the capacity or power to boost the scalability of a system. The distributed system scalability can be easily expanded by adding more computing nodes as required.
- Redundancy: centralized system reserves all the resources in the server. When the central server is unavailable, the whole system is down. Distributed system can duplicate the data into multiple copies to ensure more accessibility and avoid single node failure.
- Price/performance ratio: since many smaller machines can be used to scale out, the total cost is lower than one powerful machine.

2.7.2 Requirement

The design of a distributed system should achieve the following goals:

- Openness: the communication protocols or infrastructures should be easy to access, which will make it easier for troubleshooting in a large scale distributed system.
- Transparency: The framework designed should conceal heterogeneous architecture. The fact that the resources are distributed across the network and should provide a universal way of retrieving the resources even though the resources are relocated or part of the system is in a failure status.
- Scalability: A scalable distributed system is a system that can be flexible with the size, geographical location, and administration.

2.8 Cluster Manager

In this cloud distributed system, Yarn is selected as the cluster manager. Yarn [65] is a resource manager for scheduling jobs and monitoring the CPU, memory, disk and network usage. There are two components called Resource Manager(RM) and Node Manager(NM). Between the Resource Manager and Node Manager, there is a frame specific Application Master(AM) which is responsible for negotiating with RM and NM. The NM is responsible for nodes and responds to the requests from RM. RM is the interface accepting jobs from clients and schedules it. The manager communication between nodes uses RPC service [48]. The whole architecture of YARN can be found in Figure 6.

To execute a distributed job. There are several steps happening in the cluster :

1. Client decides which input data is required for execution and fetches its meta-data.
2. Client generates descriptor HDFS files that contains the location of each partition.
3. Client triggers AM and RM.
4. AM negotiates resource containers with a set of nodes.
5. Application executes in the container.
6. Containers are deregistered and shut down after work has been finished.

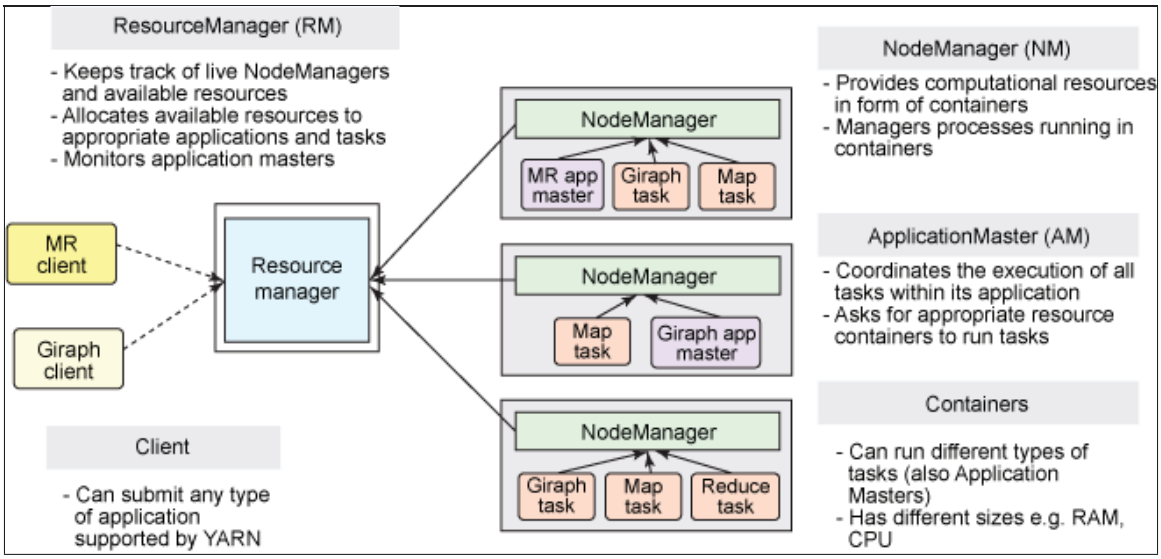


Figure 6: Overview of YARN Architecture [34]

Chapter 3

On Cloud Data Processing Framework

3.1 System Components

The system designed has the following key components.

Data Pool: Data pool is the source of data to be processed. It can be from other OLTP databases, some offline wearable devices, offline storage drives like tape drives or any other distributed file system such as HDFS.

Processing Framework: This thesis uses Spark as the processing framework. Inside Spark, Remote Procedure Call (RPC) and event loop mechanisms are used to communicate with each other. There is a subsystem called Netty achieving these. For bulk data transportation like shuffling, Spark uses Java Non-blocking I/O (NIO) to transfer the data. There is also some broadcast data transportation delivered by Jetty [33] subsystem. As an extent to the spatial field, Java Topology Suit (JTS) is selected as the GIS format data support.

Distributed System Management Software: To sufficiently manage these resources and to schedule tasks, Yarn a cluster manager is introduced to track these resources. This component will be explained in the following section.

Distributed Persistent Software: In this system, two persistent methods are used, one is an in-memory method based on Spark native RDD, another one is based on graph-database implemented by Neo4j.

3.1.1 On Cloud Data Pool

In this system, Amazon S3 is selected to store the raw dataset gathered from the Internet.

Amazon S3 stands for Simple Storage Service. As an object storage system, it mostly is used for backup and restore, disaster recovery, archive, data lakes and big data analytics, hybrid cloud storage or cloud application data storage. As an object storage service, it can ensure 11 9's durability and 99.99% availability.

To use the S3 storage, users are required to create a bucket in a specific region. Then it is possible to use API like REST API or SOAP interface to upload objects into buckets. Each object consists of object data and metadata. An object can be identified by a key in the bucket and a version ID. Amazon S3 provides eventual consistency when multiple clients are writing at the same items.

3.1.2 Processing Framework Architecture

Apache Spark[72] is an In-memory computing framework based on the MapReduce programming model. It has multiple extension modules such as streaming computing, machine learning, graph theory, or SQL support. Spark is written in Scala but supports multiple languages including Python and Java. Resilient Distributed Dataset(RDD) is the primary storage structure in Spark. All computation operations on the dataset are the *transormations* towards RDDs.

RDD is cacheable and fault-tolerant. If one or more partitions are missing or failed, Spark can restore from the data source with the help of lineage transformation plan. RDDs are read-only. There are two ways of creating an RDD: *parallelizing* existing in-memory collection or *referencing* a dataset from external storage.

For a better optimization, Spark transforms RDD lineage into Directed Acyclic Graph(DAG) stages. The stage is the minimal schedule unit. It applies lazy load technique which means the stages are not executed until an action in the workflow requiring the results to produce non-RDD values.

Previous frameworks store the intermediate data on disk like HDFS, so the next task can retrieve the shared data from there. Spark provides a new approach to enable the intermediate data stored in memory and shared between nodes for parallel computing.

The framework architecture follows a layered architecture to support metrics calculation, topology modeling, parallel distributive processing, query, and storage. The architecture is shown in Figure 7.

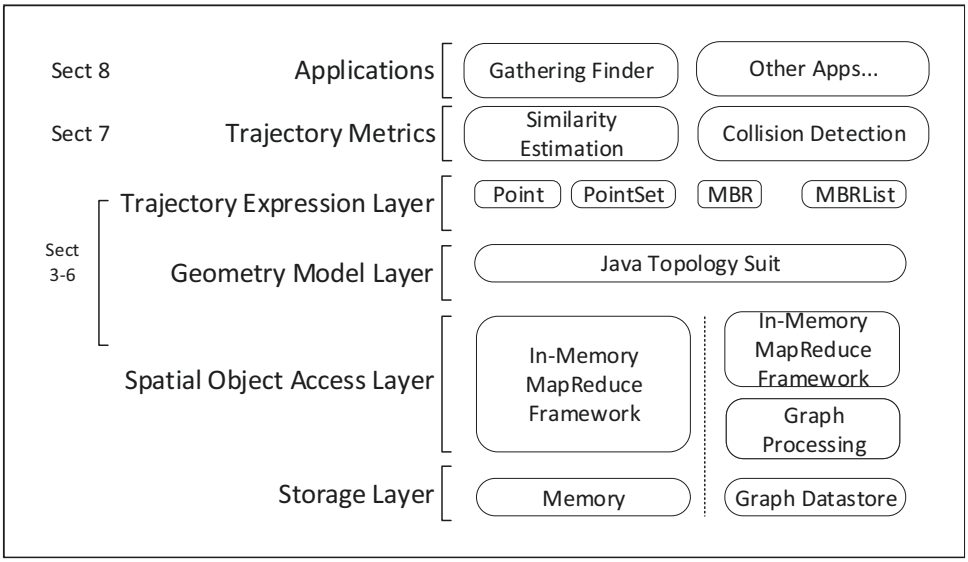


Figure 7: Overview of Framework Architecture

Application Layer

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

Middle-ware Layers

The second layer is the *Trajectory Metric layer*. In this layer, all trajectory metrics are calculated. Three dimensions are considered in the 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 Figure 7. And then the trajectories are further processed as rectangles shape called Minimum Boundary Rectangles (MBRs). The further discussion about MBR is elaborated in the next section.

The fourth layer is the *Geometry Model Layer*. At this layer, topology calculation is performed. It converts MBR objects to JTS objects. The benefit of converting to JTS objects is that the 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 4.3.

Infrastructure Layers

The fifth layer is the *Spatial Object Access Layer*. The operations on MBRs are specific to the data processing frameworks. In this thesis, two kinds of data processing frameworks are considered, namely a NoSQL graph database and Apache Spark in-memory processing framework. If MBRs are stored in a graph processing system (such as Neo4j), Cypher, a graph query language is programmed to operate these data. If the MBRs are operated by an in-memory geometry processing framework(such as GeoSpark [71]), Spark parallel functions to access MBRs are developed.

The sixth layer is *storage layer*. In the graph processing system, MBRs are indexed and stored in the directed graph structure. They are distributed on multiple database nodes. For the in-memory processing framework, MBRs are indexed and stored in a customized tree structure. The tree structure is organized in Spark RDD structure for distributing.

3.2 Trajectory Expression

Trajectories can be collected from various types of devices. The most commonly seen trajectory data are collected from GPS tracker. A GPS tracker periodically receives the signals from GPS satellites and calculates the current position. This requires the device to be exploded to an open area to receive signals. For indoor trajectory collection, GSM cell stations, WI-FI hot spots or RFID labels are used to get the approximate current location. Furthermore, the trajectories can also be extracted from video stream file like surveillance cameras.

A trajectory is a collection of unique points organized in time series order. For the unprocessed data, this thesis uses $Tr =< pt_1, pt_2, ..., pt_n >$ to express a trajectory. A point can be expressed with four elements: $pt_k = (id_k, loc_k, t_k, A_k)$ in k^{th} position. id_k is the position identifier; loc_k is the spatial location of the position; t_k is the time at which the position was recorded; A_k is the additional data like altitude or temperature. For loc_k, it can be expressed as coordinate data (x, y) if the data is collected from GPS-based device or the loc_k will be marked as the cell ID of a GSM base station, Wi-Fi hot spot, or an RFID label. If the location is marked by cell ID, these data should be transformed into coordinates so that they can be expressed in Euclidean space.

Further more, the trajectory can be simplified as a rectangle or an envelope formed by the minimum and maximum latitude and longitude coordinate. The rectangle is called **Minimum Bounding Rectangle (MBR)**. The MBR is an approximation of the trajectories and that transforms the discrete point problem into topology problem. Figure 8 shows how an MBR bounding a trajectory in 3 dimensions.

In this project, the trajectory files generated by GeoLife devices are GPS position logs. Each trajectory is a single "plt" file. It records the latitude, longitude, altitude in feet, data and time. The framework firstly loads the data in bulk from Amazon S3 datastore. Then it parses each single pt_k into **Point** datatype and a sub trajectory Tr can be expressed as a **PointSet** datatype.

A segmentation method splits one trajectory into maximal K segments. In this thesis, segments are the first class entities for indexing, storage, and query. In this segmentation method, a data model with components to encapsulate the operations on segments is designed. Then a greedy split algorithm to split each trajectory into segments is presented. Since trajectories are independent, this greedy-split method is

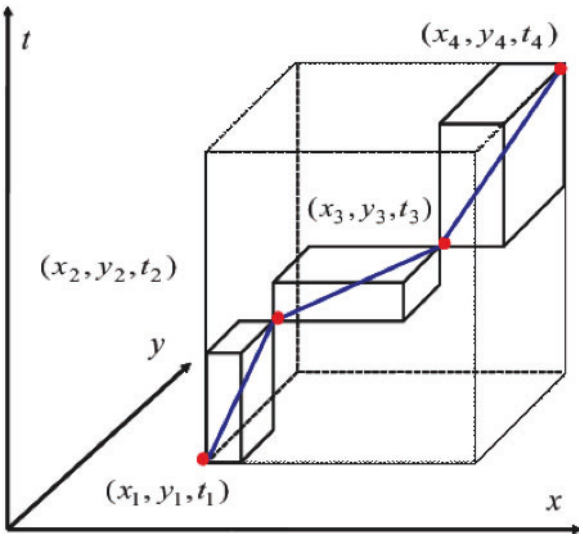


Figure 8: The Trajectory MBR in 3-D. [9]

processed in parallel. The commonly used notations in this thesis is listed in Table 2.

Table 2: 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 coordinate point in trajectory Tr_p, sequence i, $pt \in Tr$
$pt.x$	coordinate longitude
$pt.y$	coordinate latitude
$R_{c,v}(resp.R_{q,u})$	Candidate MBR relation, partition v (resp. query MBR, 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

3.3 The Data Model

The data model's entities and their relationships are presented in Figure 9.

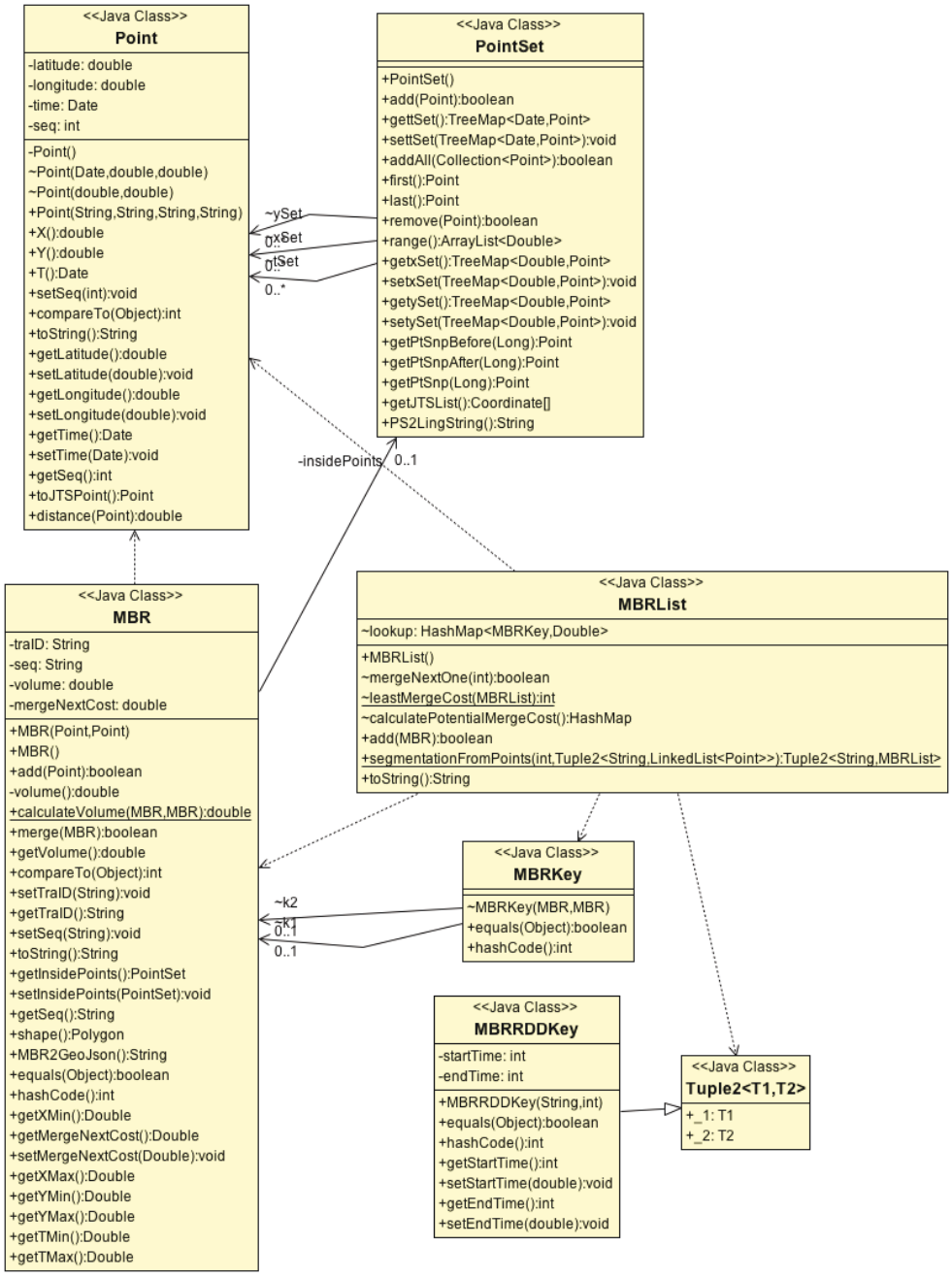


Figure 9: Data Model of Trajectory Segmentation

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. This framework supports an extra attribute of the timestamp T. Ours also supports the Euclidean distance calculation when required.

PointSet. A trajectory consists of a cluster of points as time elapses. `PointSet` class to express the sub-trajectories is created. At least one point can be the smallest sub-trajectory. These sub-trajectories can be linked to form a longer sub-trajectory by using `addAll()` function or only adding one point to extend this trajectory. All these internal points have their sequence order by sorting their timestamps.

To get one position snapshot at a specific time, this framework uses the function `getPtSnp()`. As an estimation in between two real points, a virtual position is calculated based on the average velocity between the two adjacent positions.

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 merge procedure will be presented in detail in this section 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. The framework can get the MBR vertexes of its sub-trajectory by using the `range()` function and get its volume by using `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.

3.3.1 Trajectory Segmentation Methods

Dieter Pfoser et al. [53] compare several query approaches of moving objects. The naive way is to query the sampled points directly. Using the coordinate-based query such as nearest-neighbor, range query is possible. It can not reflect all the movement of the objects especially for the times in-between the sampled points. Further, the linear interpolation can be used. the algorithm connects the points as endpoints of segments. The queries are trajectory-based topological queries which can handle speed, acceleration information or more.

In this system, it puts the sub-trajectories into Minimum Bounding Rectangles (MBRs). MBRs can simplify the topology query and as an approximation spatial query as well as for spatial indexing propose.

For time-series analysis, the objective of trajectory segmentation is to provide

homogeneous pieces. A high standard trajectory segment can expose clear information in high level representation, reduce the chance of noise and finally give a better expression for the algorithm to analyze the behavior behind the trajectories [10]. The commonly used trajectory segmentation methods including three thoughts: fixed length split, probatlity splitting, and greedy algorithm.

Fixed Length Split

Ferreira et al. [21] provide a fixed time length trajectory segmentation method. They transform the sub-trajectories into vectors for K-Means clustering. It cannot ensure the sub-trajectories are evenly divided, so the accuracy is limited.

Probability Theory Split

Lee et al. [41] presents a partition algorithm using Minimum Description Length(MDL). They turn the optimal partitioning into best hypothesis using MDL principle. Since it's used for trajectory clustering, line segments with the best similarities are clustered together. The time complexity is $O(n)$

3.3.2 The Greedy Split Algorithm

Stage 1: Trajectory segmentation. The segmentation process transforms a trajectory expressed by coordinate points into a sequence of MBRs. An illustrating process is depicted in Figure 10. In this process, smaller MBRs are aggregated into larger MBRs. Initially, each two consecutive points in a trajectory sequence resemble as diagonal vertexes of an MBR.

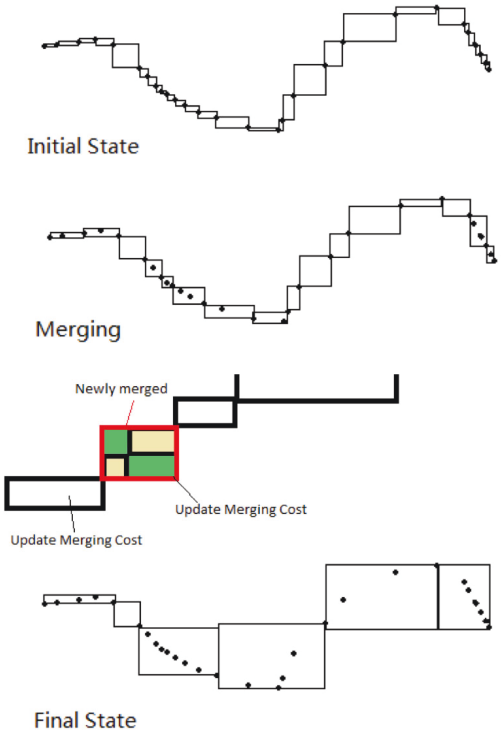


Figure 10: Main Steps of the Greedy Splitting Process

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 [54] :

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

Where Vol denotes the volume function of MBRs.

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.

The framework adopts a greedy-split algorithm [67] to balance the cost and approximation quality. The implementation uses the data model defined. The full details of greedy-split are listed in Algorithm 7. The merging action is listed in Algorithm 8.

Algorithm 7 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. 2)

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

Algorithm 8 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: $P_{a\prime} := P_a \cup P_b$, where P_a and P_b is mbr_a and mbr_b's inside PointSet

2: get $x_{max} := Max\,(P_{a\prime}.X)$,
$x_{max} := Max\,(P_{a\prime}.Y)$,
$x_{min} := Min\,(P_{a\prime}.X)$,
$y_{min} := Min\,(P_{a\prime}.Y)$

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 $=P_{a\prime}$

5: **return** a new MBRmbr_{ab} covering mbr_a and mbr_b

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

3.3.3 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 contains a number of trajectories that are segmented in parallel.

To enable parallel processing on data partitioning, this study develops the greedy-split algorithm using Resilient Distributed Datasets (RDDs) in Apache Spark [72]. RDDs are first created by reading the dataset from stable storage such as HDFS into the partitioned collection of records. These RDDs are further transformed by operations such as `map`, `filter`, `groupBy`, `reduce` and so on all elements in the dataset. So RDDs are immutable. RDDs are distributed datasets processed in memory of worker

nodes. Accordingly, different RDDs containing different data types correspond to objects defined in this data model. The transformations and operations on RDDs realize the greedy-split algorithm.

When the raw data are stored in HDFS or any other file system with distributed blocks like S3, the initial RDDs created by reading files from that are already distributed and partitioned. The partition size and their distribution are inherited from HDFS's block size and the partitions will be distributed into multiple nodes. At this time, the trajectories in the same partition origin from the same block in HDFS host.

Each Geolife trajectory is a text-based file in the dataset. After reading to Spark, each trajectory record is a `<key, value>` pair in the RDD. In each RDD record, the key is the file name of that trajectory, and the value is the raw content of that file. Followed by that, the content is read line by line to create position records with latitude, longitude and the timestamp to a `Point` in this data model.

After this transformation, it has a new `<key,value>` pair RDD, where the key is still the file name and the value is this trajectory's point list, as `LinkedList<Point>`.

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

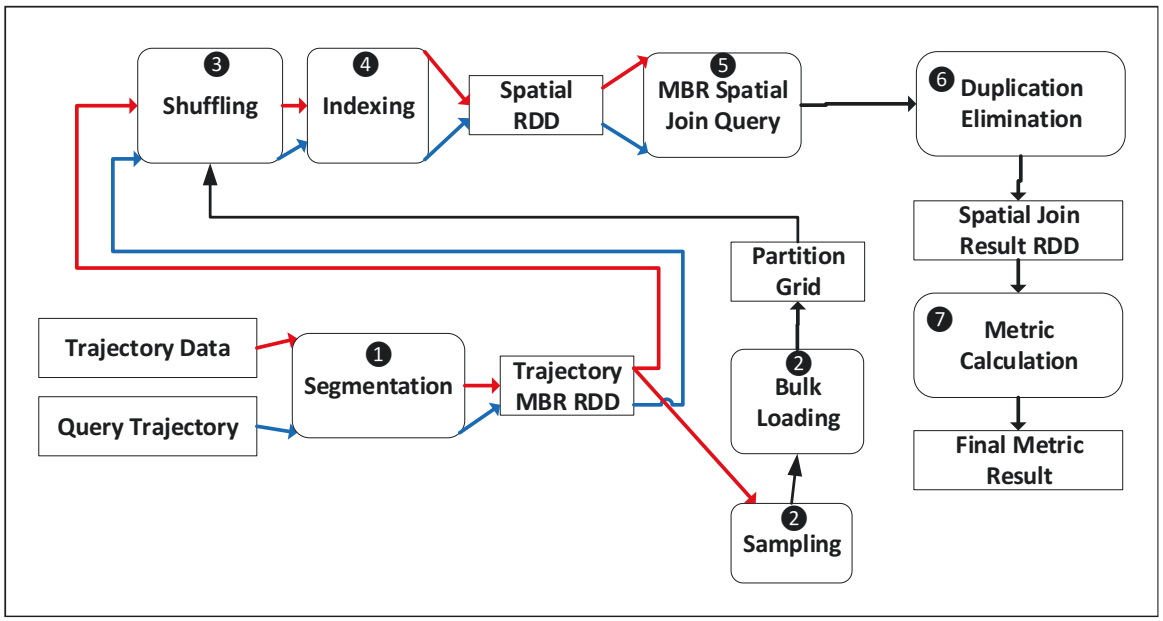
Afterwards, system uses the `flatMap` transformation to flatten RDD's value that is represented as `MBRList` to a sequence of `MBRs`. The new RDD has the compound key called `MBRRDDKey` that consists of two attributes. One attribute is the trajectory name, and the other is trajectyr's MBR sequence number in a chronological order acquired from the `MBRList`.

3.4 Partition and Indexing

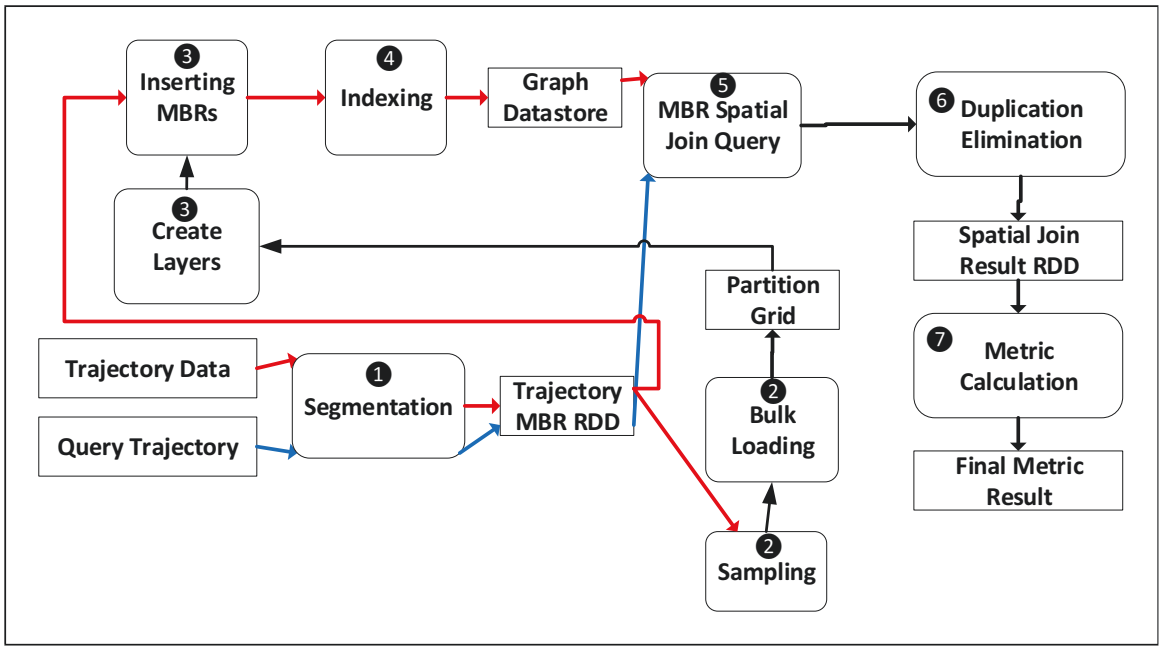
The trajectory repartitioning shuffles all MBRs within a certain geographic boundary to the same partition. These spatio-temporal partitions form closures. Inside this closure, the framework can perform the intersection join operation. It further extends the spatial boundary with the temporal boundary, that means MBRs within a certain time period are also partitioned to the same node. Under this spatio-temporal repartitioning, an intersection query to sub-trajectories occurs within the same partition.

This is different from the initial MBR based partition discussed in Section 3.3.3. 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 partitioning significantly reduces the data shuffling in the following processing.

The workflow is depicted in Figure 11. This thesis notates activities of this data flow with numbers to present the techniques involved and the mapping of activities in the distributed cluster deployment (depicted in Figure 12). 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.



(a) In-memory Processing Workflow



(b) Graph Database Workflow

Figure 11: Framework Workflows

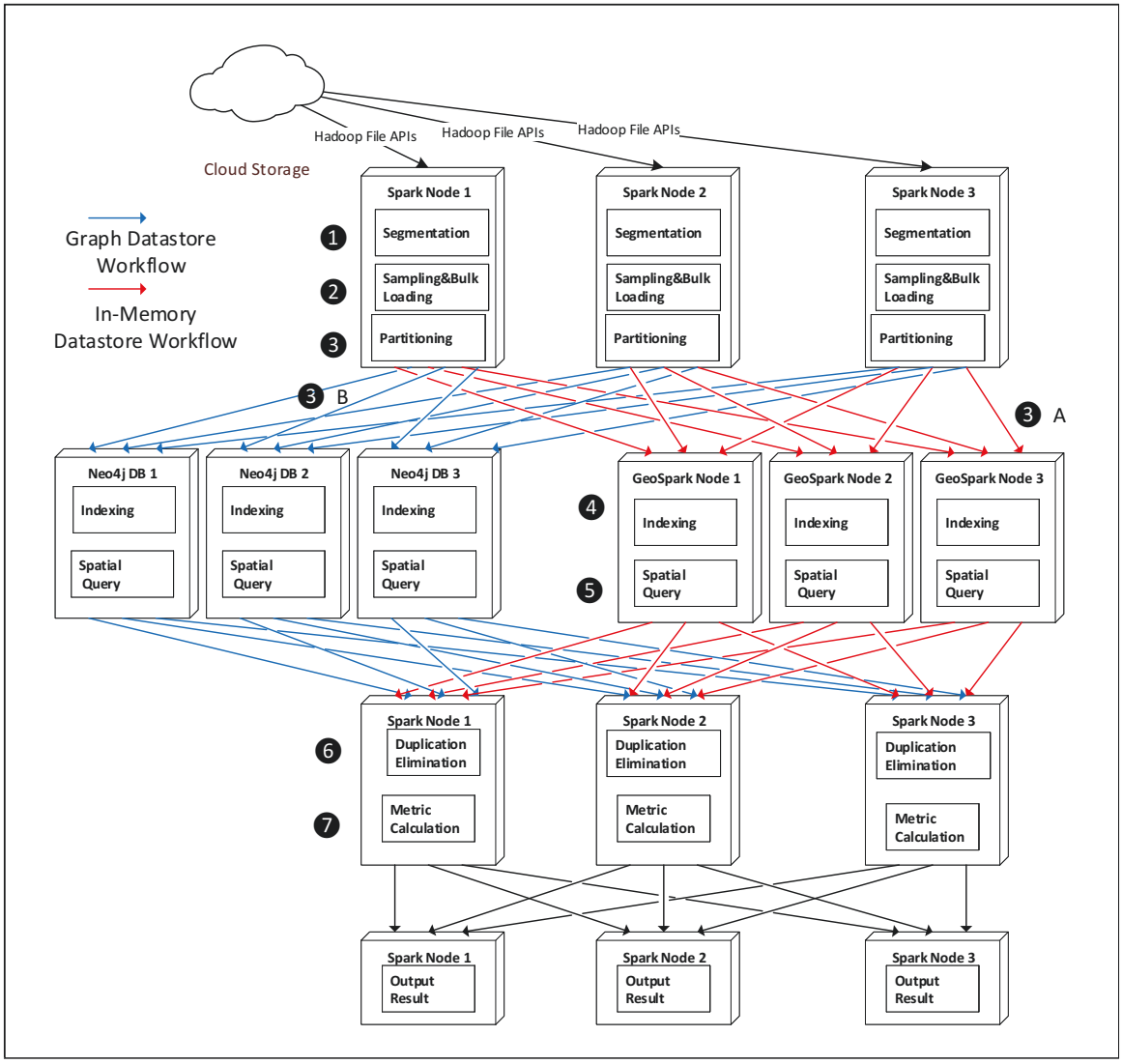


Figure 12: Dataflow Between Nodes

3.4.1 Partitioning Techniques

The *Spatial Partition* activity in the data flow uses a spatial partition method. The spatial partitioning methods include Equidistant, Hibert, Voronoi, Quad-tree and R-tree [16].

This study develops an R-tree partition to achieve balanced partition of MBRs. Since it aims to put MBRs within the same spatio-temporal boundary to the same partition. Therefore, 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 there are 2000 MBRs after the segmentation process, and sample 1% MBRs to build the R-tree. Then it is 20 MBRs for building up boundaries for partitions. Assume further that the system can get 10 partitions, then boundary ranges are divided into the 10 ranges based on the 20 MBRs's spatio-temprol span. In addition, there is one more range for any MBRs that are beyond the spatio-temporal range from the samples called overflowed partition. In general, if it targets p number of partition, it finally has $p + 1$ ranges.

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 a larger range than average; if the objects in an area are dense, this leaf node contains a smaller range than average.

The thesis develops the R-tree spatial partition in-memory using GeoSpark. The major tasks and techniques are presented below.

Stage 2: Creating a geographical partitioning grid. In this step, the framework creates a geological partitioning strategy. This is a spatial grid based on R-tree rectangles. There are two sub-steps including step 2.1 and 2.2.

Stage 2.1: Data sampling. The framework randomly samples 1% of the whole MBRs for establishing range boundaries of the partitions with R-tree. This sampling method avoids building a global index thus helps to reduce the computation cost.

Stage 2.2: Bulk loading. With the samples, the system builds the R-tree using the Sort-Tile-Recursive (STR) algorithm [27] 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 rest of range of boundaries outside 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 like

`Distance Join`, `Range Query`, `KNN Query` or even saving as text files.

Stage 3: Data shuffling or migration. In this step, there are several sub-steps marked as 3.1, 3.2 and 3.3.

Stage 3.1: Partition assignment. When an MBR imported 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 ID number becomes the key to the MBR's RDD. A *replication* method is further developed 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 of partitions 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 9.

Algorithm 9 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_Trc then
4:       partitionID := i
         /*Contain only check*/
5:       containFlag := true
6:    else if Partition_i intersects mbr_Trc 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
```

3.4.2 Data Shuffling

All MBRs within the same geographical partition should be located on the same Spark worker node. This redistribution process involves *shuffling* RDDs in GeoSpark or *migration* from RDDs in memories to Neo4j data nodes.

In the following steps, the thesis uses A suffix to distinguish the stages occurring in the in-memory framework and use B suffix to express the stages occurring in the graph storage framework.

Stage 3.2A: Repartition. The framework applies 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.

3.4.3 Data Persistence

The persistent method of MBRs migrates the MBRs in Spark RDDs to the NoSQL database, Neo4j. To support the spatial data model, the framework deploys the Neo4j-spatial extension [64] 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, a map layer is refereed as as `partition` too.

Stage 3.2B: Create partitions on Neo4j. The framework deploys a total number of s Neo4j data nodes. Each node runs independently with distinct data partitions. Therefore it deploys 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 it assigns $p + 1$ map layers (or partitions) to s nodes using the binning method. To identify the node destination, $NodeNumber = MBRKey\%s$ and $LayerNumber = MBRKey$. The framework builds a router to route each MBR to a designated map layer (or partition). Before inserting the MBRs, the map layers with the exact $LayerNumber$ will be created explicitly.

Stage 3.3B: Inserting MBRs to Neo4j partitions. Neo4j uses the WKT format to represent the geometry when inserting or query. Since the MBRs in Spark RDDs are serialized objects stored in memory, the framework further inserts MBR's RDDs to Neo4j's nodes. Due to the Neo4j `procedure call` limitation, the insertion of polygons in bulk is not supported. The solution is to traverse all the MBRs in each partition, and partitions execute the insertion operation in parallel. If MBRs are assigned to a partition on its local Neo4j node, the data is 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.

3.4.4 Spatial Indexing

There are a number of data structures supporting spatial indexing. Using fixed cell methods [28] cannot ensure the best performance because the cell size should be determined in advance. Quad Tree [22] is more efficient in update-intensive applications and requires fine tune to realize the best performance [35]. K-dimensional B-tree (KDB tree) [55] is only useful in point data. R-tree is more robust and can achieve high performance without too much tuning optimization.

3.4.5 Local R-tree Indexing

Inside each partition, local R-tree indexes are built for fast retrieving. Since the MBRs are stored in different frameworks, the implementation of R-tree indexing may differ.

Stage 4A: In-memory R-tree indexing. The JTS library [57] provides an R-tree index data structure. The framework can create an `STRtree` and use `insert()` function to build that R-tree index as part of `SpatialRDD` in Spark heap. In each partition, an R-tree index is built by traversing the MBRs in their partition. All partitions' local R-trees are stored in GeoSpark's `SpatialRDD`.

Stage 4B: In graph storage R-tree indexing. The R-tree indexes are built simultaneously when inserting MBRs into the Neo4j map layers. The key of each MBR is a compound key including `Partition ID` and `Time Slot ID`. The `Time Slot ID` is a method of rebalancing the data distribution over time. The framework divides one day into multiple time periods and number each slot. More details can be found in Section 6.1. Based on this compound key not only the `Partition ID`, the framework hashes the key and map the MBR object to a Neo4j map layer. The visualization of an R-tree structure in Neo4j is shown in Figure 13. 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 covering all the geometries. The BBOX is accessed following the `RTREE_ROOT` relation. The non-top level BBOXes are connected by `RTREE_CHILD`

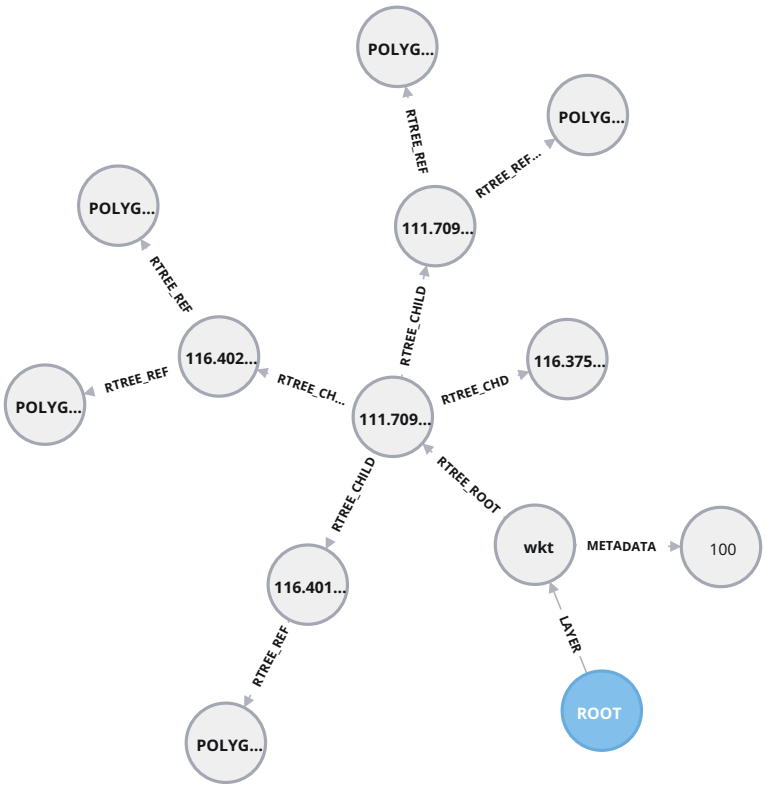


Figure 13: The Neo4j Local R-tree Visualization.

relations. At the leaf level, MBRs are serialized in the WKT format and stored as one property.

```
1  JavaPairRDD<TrajectoryID , String> input = JavaSparkContext.
       wholeTextFiles(Path);
2  /* Rading from text */
3  JavaPairRDD<TrajectoryID , Set<Point>> pointRDD = Formatting(input);
4  /* Coverting log file to trajectory, each trajectory represening as a
       point set */
5  JavaPairRDD<TrajectoryID , List<MBR>> MBRListRDD = GeedySplit(pointRDD);
6  /* Using greedy-split to transform to MBRs*/
7  JavaPairRDD<<TrajectoryID ,MBRID>,MBR> MBRRDD= Flatten(MBRListRDD);
8  /* Flatten the RDD, each key is unique*/
9  JavaPairRDD<<<TraID1 ,MBRID1>,<TraID2 ,MBRID2>>, SEValue> SERDD =
       SimilarityEstimationCalculating(MBRRDD);
10 /* pairwise similarity estimation calculation*/
11 JavaPairRDD<<<TraID1 ,MBRID1>,<TraID2 ,MBRID2>>, isCollided> CDRDD =
       collisionDetecting(MBRRDD);
12 /* pairwise collision detection stage*/
13 JavaPairRDD<<TraID1 , TraID2>,SEValue> SERecords = aggregate(SERDD);
14 /* Aggregation to generate trajecory scale result*/
15 JavaPairRDD<<TraID1 , TraID2>,CDValue> CDRecords = aggregate(CDRDD);
16 /* Aggregation to generate trajecory scale result*/
```

Listing 3.1: Spark RDD Workflow Summary

3.5 The Query Workflow

Above steps generate spatio-temporal data indexes and store the data partitions in a cluster of nodes. The framework further develops queries on trajectories and output MBRs that meet certain predicates. The queries enable to 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 Figure 14. 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. The outline of the in-memory implementation part of the workflow is presented in List 3.5.

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 (referred as **Stage 1: Trajectory segmentation** in Figure 14), partitioning (referred as **Stage 3: Data shuffling or migration**) and indexing (referred as **Stage 4: Local R-tree**

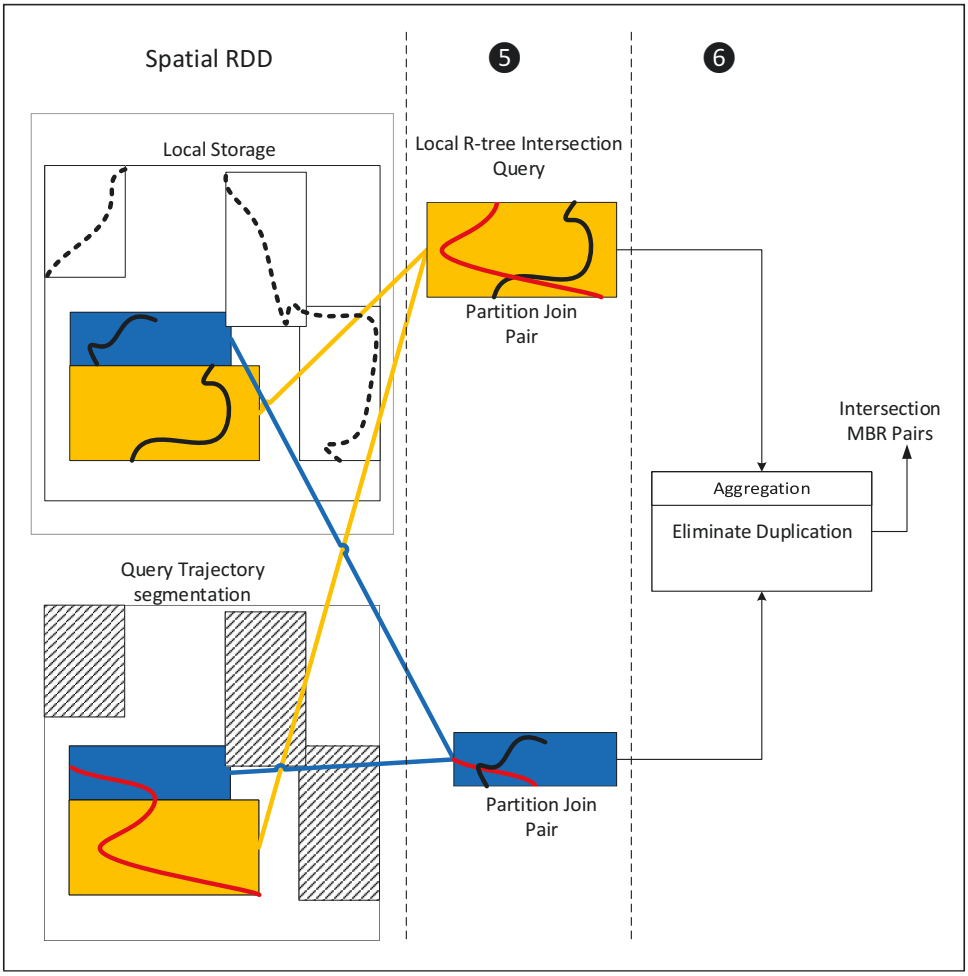


Figure 14: The Trajectory Query Workflow in Two Parallel Partitions.

indexing) as discussed in previous sections. Due to the difference of the storage architecture, queries on GeoSpark and on Neo4j have separate workflows.

3.5.1 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.

This thesis defines $R_c = \bigcup_{v=1}^{p+1} R_{c,v}$ 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:

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

Where the `Intersects(a,b)` predicate is defined in Dimensionally Extended nine-Intersection Model (DE-9IM) [60]. It is to be 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, the study first defines an *intersection closure* as $R_{q,mbr_{Tr_q},j} \cup R_{c,mbr_{Tr_c},i}$ that covers the overlapping MBRs, where

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

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

It is known that because of the *replication* strategy, it is possible to make sure $u = v_1 = v_2 = \cdots = v_n$. This is true with intersection predicate, not KNN or others.

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 $R_{q,u} \cup R_{c,u}$. Therefore, MBRs within separate partitions are aggregated by the reduce stage of Spark to generate the super partition closure set. Therefore the intersection join is consistent with the sequential and centralized processing whereby the overlapping MBRs are within the same partition.

An example is illustrated in Figure 14. 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 sub queries in Stage 5.

3.5.2 In-memory Query

Stage 5A.1: Range query pre-screening. The framework utilizes the local R-tree on each partition to find out the intersected MBRs with the query trajectory's MBR.

Each query MBR produces a `JavaRDD< QueryMBR,HashSet<MBR>>` record. This involves the join operation that is converted to multiple rounds of range queries.

Stage 5A.2: Intersection. The framework retrieves the candidate MBRs that are within the query range. It traverse the candidate MBRs to execute intersection predicate.

Algorithm 10 Algorithm for Join Query in Map Stage

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 11 Algorithm for Join Query in Reduce Stage

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>
```

3.5.3 On Graph Store Query

Stage 5B: Direct intersection query. The query on Neo4j is simple since the R-tree index on Neo4j involves the execution plan automatically. The intersection calls the `intersect` procedure to return MBRs.

The Map Reduce procedures for parallel spatial join are shown in Algorithm 10 and Algorithm 11. In Algorithm 10, from line 2 to line 6, the intersection query procedure is implemented in different platform based on what you choose. If you choose GeoSpark method, the indexed query is completed on Spark framework; if you choose Neo4j method, the indexed query is completed on Neo4j internally.

3.5.4 Duplication Elimination

Stage 6: Duplication eliminate. The intersection operations take place in each partition. The system gets a result pair `<Candidate MBR, Intersection MBR>` showing that there is an intersection between the candidate trajectory MBR and query trajectory MBR. After this it groups the results by `Candidate MBR`. Duplicated intersection MBRs to one `Candidate MBR` in different partitions will be grouped and removed. All distinct `Intersection MBRs` in the result pair to this trajectory make up the final results.

3.5.5 Raw Data Separation Technique

For the in-memory framework, the framework embeds the original text raw data into MBR objects. This is redundancy and this causes more network traffic when shuffling data. Actually, it do not use these raw data until the last step before outputting the final result.

It is a chance to split these raw data into a separate RDD aside and add an extra *join* step to combine the results later before output. This costs extra time to join but reduces the shuffling time especially when additional data are extra large. By applying this technique, it can reduce the memory cost when transforming the RDDs, which can fit in smaller memory size node cluster. The shuffling transformation execution time is also reduced based on the experiments.

Chapter 4

Trajectory Metrics

Stage 7: Metrics calculation. This thesis defines two basic metrics that are computed using the framework architecture and workflows developed above. These two metrics are basic and composing elements to applications such as clustering analysis of trajectory data. The study presents one metric to estimate the trajectory similarity and the other metric to detect the collision of trajectories.

4.1 Trajectory Similarity Estimation

This thesis measures 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 $mbr_{Tr_p,i}$ that has six attributes that 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 to get the Tr_p's i_{th} MBR and Tr_p's j_{th} MBR intersection volume in time axis, the thesis defines the `intersection` operation in the time axis as:

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

where (p) denotes the partial intersection on the time axis.

It's possible to get the following property:

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

or

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

Next, the norm for the time axis is defined as

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

Likewise, it defines the intersection in longitude $Intersection_x$ and in latitude $Intersection_y$.

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

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

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

Then, it defines the intersection volume:

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

where V denotes volume in 3D.

For any trajectory Tr_p and query trajectory Tr_r, the `Intersection Volume` is calculated as:

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

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

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

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

where $length(Tr_p)$ is the trajectory lasting time 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 Figure 15 whereby the yellow area is the similarity estimation value of these two trajectories(refer the gray existing MBRs in Figure 10.). The algorithm is shown in Algorithm 12.

Algorithm 12 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

```
<<QueryMBR>,[<IntersectedMBRs>]>
queryResult := MBRRDD.spacejoin(MBRRDD);
for each < K, Y > pair in queryResult do
   for each mbr in Y list [< IntersectedMBRs >] do
      intersectedShape := QueryMBR.intersection(mbr)
      volume2D := intersectedShape.getArea();
      intersectedTimePeriod :=
      QueryMBR.getTimeInterval().overlap(
      mbr.getTimeInterval());
      volume3D := volume2D*intersectedTimePeriod;
      return <<QueryMBR,mbr>,<volume3D>>
   end for
end for
```

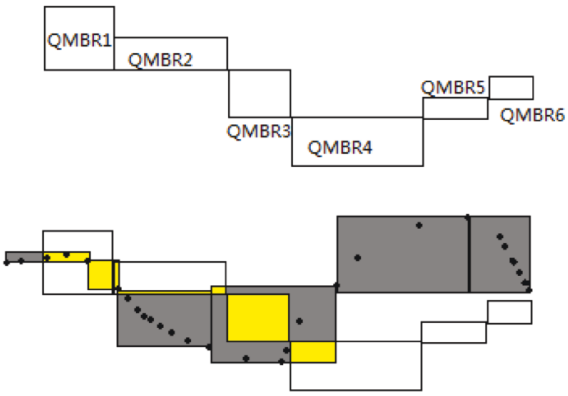


Figure 15: Similarity Estimation Metric in 2D

4.2 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.

The framework pre-screens the collision detection candidates requiring the MBR pairs whose *similarity estimation* > 0. So the collision detection operation is the downstream sector after similarity estimation. However, when two MBRs border each other or corner each other, the *similarity estimation* $= 0$, this may cause a false negative result. It gives a margin to expand the range fo MBRs so that in this scenario the *similarity estimation* > 0. Usually, the margin is set slightly larger than half of the threshold.

To test the condition of collision detection, the framework samples location points on sub-trajectory in an MBR. Certainly, the more points sampled, the more precise the result is. It has three sub steps to calculate this metric. Stage 7.1, it finds out the timestamps of the checkpoints. A checkpoint is a trajectory position point at a certain timestamp. Stage 7.2, it calculates the interpolation between two real data points as checkpoints. And step 7.3, it examines the distance between a series of checkpoint pairs.

Stage 7.1: Checkpoint timestamp selection.

For any time span as a result of the operation

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

defined as

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

and

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

.

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

$$\begin{aligned} T_{ckp}[l] =& \frac{l \times (T_{max} - T_{min})}{L} + T_{min}, \\ & 0 \leq l < L, l \in N. \end{aligned} \tag{7}$$

Stage 7.2: Checkpoint coordinate calculation.

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

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

It is possible to 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 $T_{ckp}[l]$:

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

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

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

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

Stage 7.3: Collision detection

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

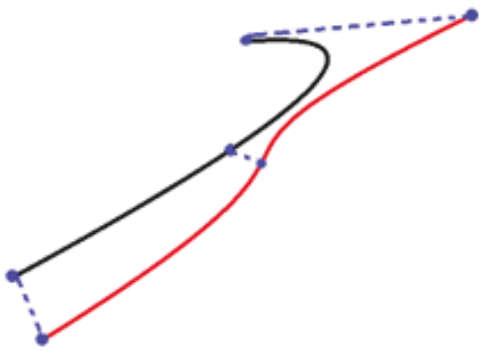


Figure 16: The Illustration of Three Blue Checkpoints for Collision Detection

To define collision detection is true as a condition that:

$$\begin{aligned}&\exists \quad l \in N, l < L,\\&Dist((x_{Tr_{p,T_{chk[l]}}}, y_{Tr_{p,T_{chk[l]}}}), (x_{Tr_{r,T_{chk[l]}}}, y_{Tr_{r,T_{chk[l]}}}))\\&< threshold.\end{aligned}$$

The study draws a diagram to illustrate the collision check among three check points in Figure16. If any one of the three dotted lines' length is smaller than threshold, the study decides there is a collision between this pair of sub-trajectories. The algorithm is listed in Algorithm 13. In the system, it sets L constantly as 3. When two MBRs collide, it records the MBR IDs and collision time.

Algorithm 13 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_pcollides 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
```

4.3 An Evaluation Application

This thesis develops an application by finding out moving object crowd profiting by above similarity estimation and collision detection metrics.

A crowd is a set of objects that with a position of collision in a certain of time [44].

In this system, trajectory crowd analysis application is further developed based on collision detection metric. Finding out the collision MBRs, the study reveals the moving objects and their positions.it uses graph theory model to explain the crowd query procedure.

4.3.1 Graph Build Up

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 each two vertices with the numerical value of collision timestamp, marked as $(u, v) \in E(G)$. this associated value is called weight, marked as $D(u, v)$. Between these MBRs, a crowd is a set of MBRs in which MBRs are maximal connected to each other by the edges.

4.3.2 Search Crowds

A connected component [18] is called a crowd in graph G. A connected component is a maximal set of vertices such that each pair of vertices is connected by an edge. By searching for the connected components using the Breadth First Search (BFS) or Depth-first search (DFS), the crowds are derived. The graph creating and search procedures are listed in Algorithm 14. The result expressed in graph theory is shown in Figure17 (The figure uses trajectory nodes instead of the MBR nodes for a better illustration).

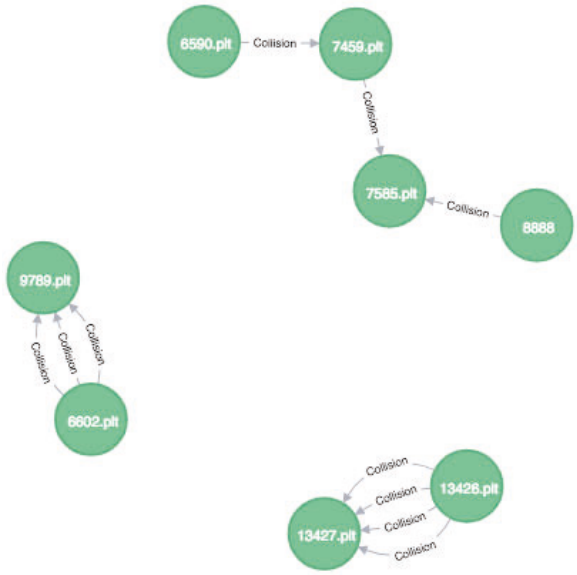


Figure 17: The Connected Components(Crowds) in Graph Database

Algorithm 14 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

If the data is processed in the graph storage framework, the framework has already store all MBRs into Neo4j graph database in Stage 3.3B. Otherwise it needs to store these MBRs into Neo4j explicitly when the previous processing is done in-memory

framework only. The framework uses Neo4j connected component algorithm [47] function `unionFind` with above weight $D(u, v)$ threshold to get the crowds.

4.3.3 Test Dataset

The study applies the clustering analysis on open data from Microsoft GeoLife project [74], 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. Since the trajectories are sparsely distributed in five-year range, the study ignores the date attribute but keep the time attribute to make the data denser as if happening in one day. The study randomly selects 3330 trajectories to find out crowds.

4.3.4 Existing Gathering Implementation

Zheng et al [75] define a gathering is a pattern occurring at a certain area or location in a certain time period indicating a non-trivial event.

A gathering pattern should satisfy five attributes-scale, density, durability, stationariness and commitment. A *crowd* is a cluster captures the first four attributes. They use the `DBSCAN`[19] algorithm to discover the crowd on snapshots, then detect the gathering patters.

The gathering crowd is the intermediate result when detecting the gathering patterns. This thesis compares the crowds produced by the workflow and ones produced by the gathering Spark implementation, referred as `GPFinder` [68] in the following analytic. The *collision threshold* = gathering finder application's *threshold* is set for further tests.

4.3.5 Small Size Trajectory Analytics

Evaluating the accuracy of the whole dataset is not quantifiable since the data is not labeled for ground truth. The analysis is non-supervised. To solve this problem, 13 trajectories are manually labeled and used as ground truth data for evaluation. The collision detection threshold and margin is set to 10 meters. The K value (max segmentation number per trajectory) is set to 20; The study compares the gathering crowds with [68]'s result.

This thesis visualizes the crowd trajectories for manual evaluation. From Figure 18, it shows 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. It is noticeable that D, E, and F gathering pairs have the same common trajectory whose MBRs are extremely large. The purple trajectory shows a pathological interpolation. One reason is the poor data quality that some coordinates of this trajectory location recording may be lost.

4.3.6 Medium Size Trajectory Analytics

400MB data are also selected for testing, which include 3330 trajectories for crowd finding. The collision detection threshold is set to 5 meters. The study finds 2740 trajectories are positive, which means they form crowds; while the rest 590 trajectories are isolated. The thesis also uses similar parameter settings with `GPFinder` algorithm to find out the gathering crowds with the same dataset. The confusion matrix is listed in Table 3.

The study notices that in this system, the sensitivity remains 86%, but the specificity is only 49%. There is a suspicion that it is the 590 isolated trajectories that interfered `GPFinder` to find out crowds.

One thing it should noticed is that the DBSCAN is a dynamic clustering algorithm, while ours is static. The study exacts the positive gathering trajectories from the application, which is 2740 from Table 3. Then put these 2740 trajectories as input to rerun both `GPFinder` and the application. This application remains the result that all 2740 are still positive. From Table 3 and Table 4 it is observed that `GPFinder` positive number rises a bit from 2484 to 2671.

This thesis does not do the whole set trajectory analytics because the lacking of the label regarding to the collision in the dataset.

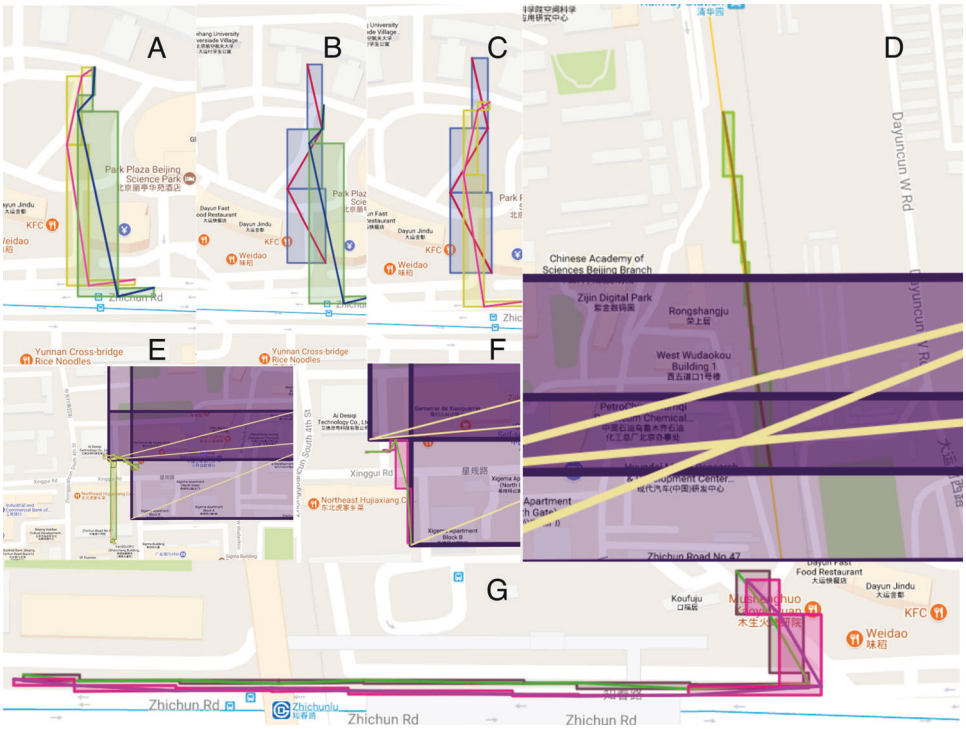


Figure 18: Positive Crowd Pair Trajectories Snapshot

Table 3: Confusion Matrix for Gathering Detection Result,a total of 3330 Trajectories

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

Table 4: Confusion Matrix for Positive Detected 2740 Trajectories as Input

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

4.3.7 Trajectory Transforming to MBR Visualization

This thesis creates a heat map showing the distribution of MBRs with the dataset of 400MB. Similarly, it also plots the heat map of these trajectories. Here the study visualizes 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 converting the point based trajectory into rectangle based MBRs. The heatmap can easily observe the distributing of trajectory density. It is noticeable the sketch of the trajectory heatmap in the top left corner is highly similar to that of MBRs in the center of Figure 19. The picture on the top left is the trajectory heat map, the picture at the center is the MBR heat map. It also highlights the three different parts where the roads are not horizontal or vertical.

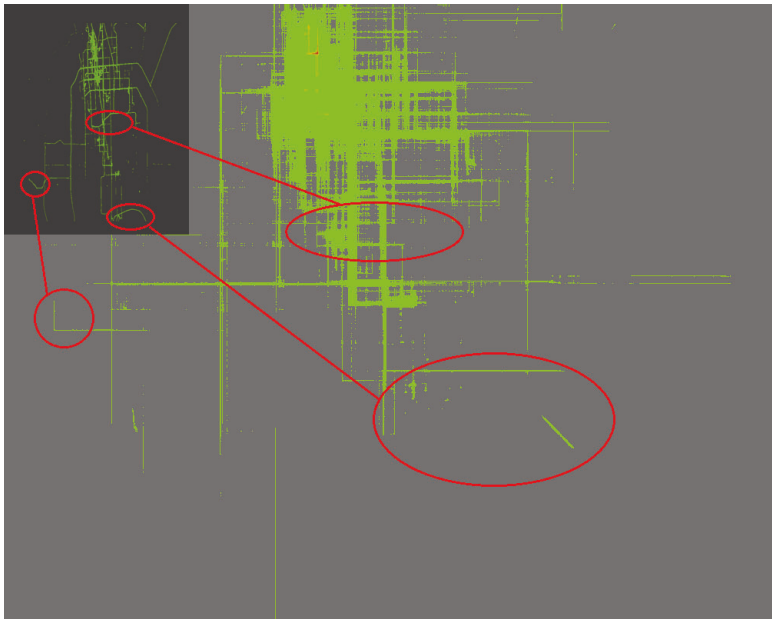


Figure 19: The Heat Map of Trajectories and MBRs

Chapter 5

System Performance Evaluation

In this section, the thesis aims 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.

These four factors vary the workload of the system. To further identify the performance bottleneck, the study decomposes the latency by stages. It applies the same set of metrics to both the in-memory framework and the graph storage framework to compare the effects of the system architecture. It adjusts the cluster size to foresee the capacity of this framework if given unlimited resources. The study controls the input data size to see if the framework can handle large data and keep the processing efficiency stable. It controls segmentation number for the scenario of a more precise result requirement. It varies the partition number to assess how the task granularity can affect the parallelism level. Also, it analyzes the latency decomposition for further optimization. The study has common metrics for both in-memory framework and graph database framework to make a horizontal comparison.

5.1 The Experiment Setup

Datasets. The thesis uses Microsoft GeoLife [74] as the trajectory source data. 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. It uses slices of datasets from the size of 100MB to the full size of 1.6GB for the varied size of the data input.

Cluster. The computing nodes are set up on Amazon EMR platform. All nodes are R4.2xlarge instances. Each R4.2 xlarge instance has 8 cores with 61 GB memory. The cluster size is the instance number of worker nodes, so the exactly running instances in this cluster number is cluster size plus one master node.

Evaluation tasks. The trajectory metrics evaluated are trajectory *similarity estimation* and *collision detection*. These two metrics can be evaluated in a single workflow. These tasks allow the framework to execute spatial self join with MBR intersection predicate and some simple Map-Reduce numeral calculations.

Performance metrics. There are common evaluation metrics for both on in-memory framework and graph database platform.

- *Latency* evaluates the end to end execution time.
- *Speedup* in latency 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* the execution time to observe the most time-consuming steps of a workflow.
- *Throughput* the effectiveness.

$$Throughput \ = \ DataSize/Latency$$

- *Shuffle Read/Write rate* :

  Input Data Size is the size of data the Spark is ingesting at this stage.

Shuffle Write Data Size is the sum of serialized data on all executors before transmitting in this stage.

Shuffle Read Data Size is the sum of serialized data on all executors after transmitting at the next stage.

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

This metric indicates to what extent that data is serialized to and from remote nodes. Reducing this rate helps reduce the I/O cost.

Global Settings. The cluster size is 4 if no further explanation. In collection detection metric, the margin is fixed to a half of threshold and the threshold is set to 5 meters.

5.2 Evaluations on In-memory Framework Based on GeoSpark

5.2.1 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.

The network communication bandwidth between AWS EMR R4 nodes is 10 GBps. A 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 5.2.2.

Data locality refers to how close the data to the processing code. Based on configuration, data and the processing code may reside on the 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, it keeps 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 Figure 21, studies 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 an extra cost of removing duplicated objects.

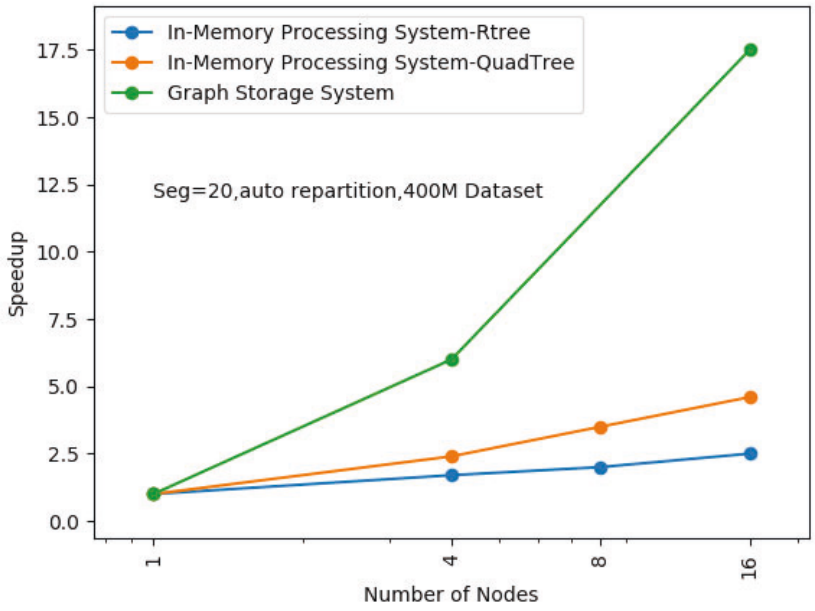


Figure 20: Speedup under Different Cluster Size

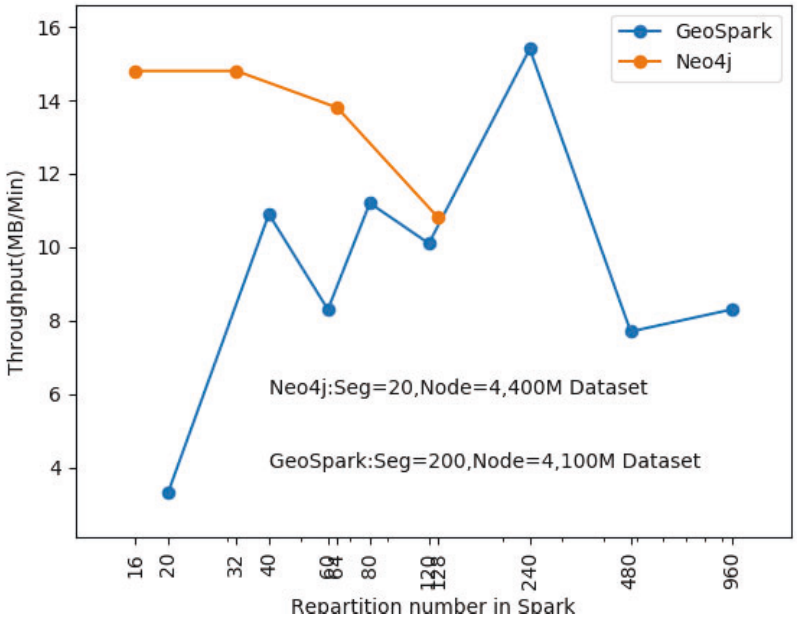


Figure 21: Throughput under Different Repartition Numbers

The speedup under the fixed data size of 400MB is plotted in Figure20. The framework automatically sets the $partition\quad number = \frac{total\quad MBR\quad number}{300}$ if not mentioned specifically to minimize the data skew. It can observed that 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. It is further discussed in Section 6.1 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 Figure22 and Figure23 for the in-memory processing system. Study notices some of the throughputs are missing because the failure of 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, K is set to 200

and use 16 worker nodes. Other configuration remains the same as the in-memory processing system.

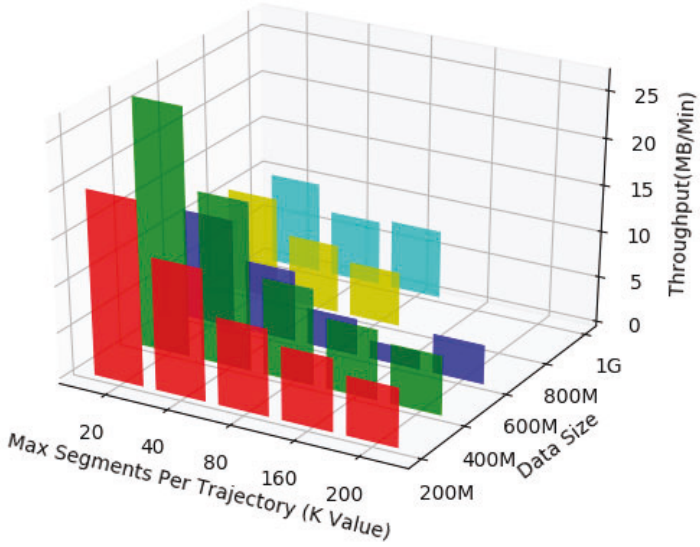


Figure 22: GeoSpark 8 Node Clustering

5.2.2 Data Size Effect

The throughput trend is displayed in Figure 24 when the data size increases. The K value (the maximum number of segmentation per trajectory) is set to 20. 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, The out-of-memory events due to garbage collection actions of Spark is observed. 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 thesis. For the graph data storage system, the throughput drops 37.5% when the data size increases from 1000MB to 1800MB.

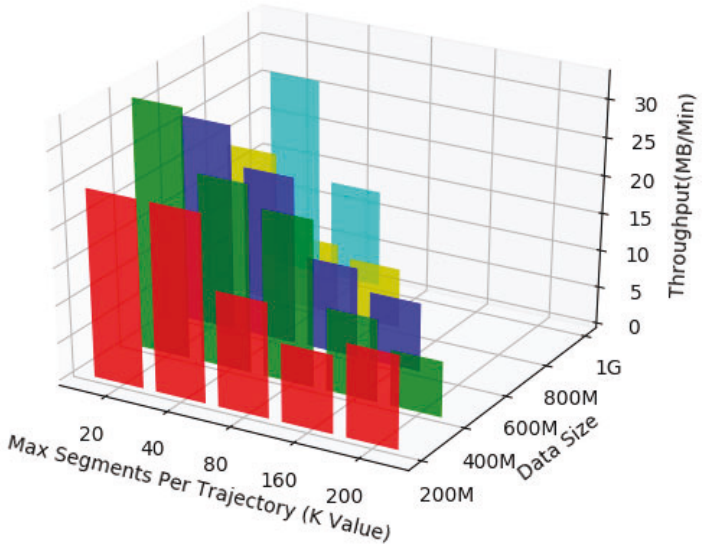


Figure 23: GeoSpark 16 Node Clustering

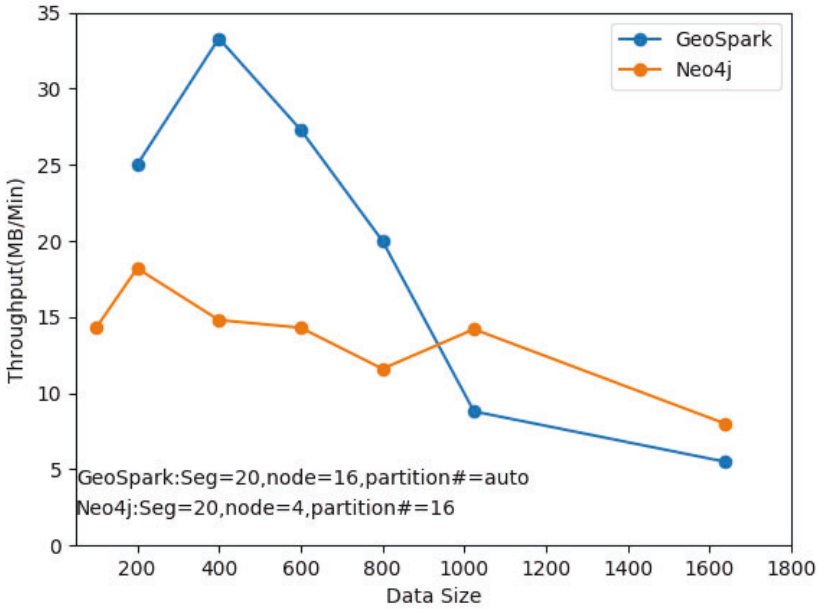


Figure 24: Throughput under Different Data Size

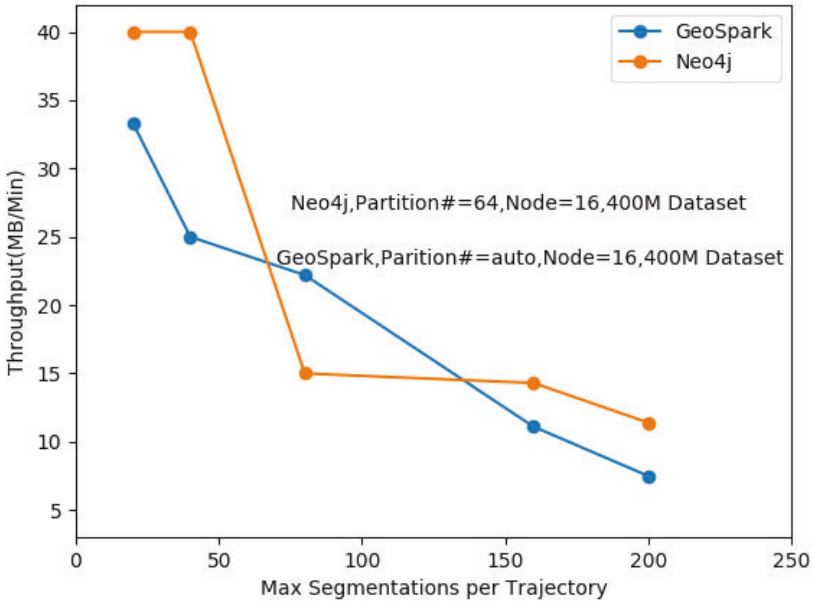


Figure 25: Throughput under Different Segments per Trajectory

Segmentation Effect

The study increases 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 Figure 25, 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 5.2.1.

Latency Decomposition

To further investigate the bottleneck, the latency decomposition for the in-memory processing system is shown in Figure 26. It shows that R-tree indexing and join query accounts for 75% of the execution time, which is 1.5 hours. It is impossible to distinguish the R-tree building and query time due to the lazy loading strategy

in each partition. After examining the execution log, the garbage collection time taking over 17% of this stage is observed. 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.

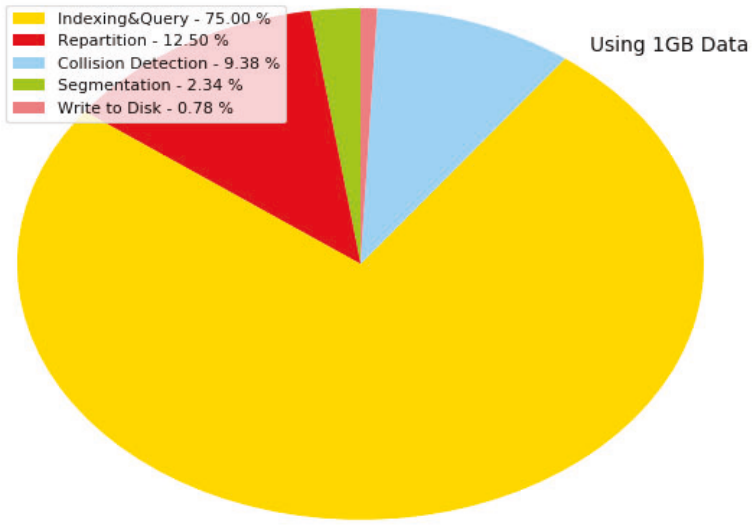


Figure 26: In-memory Processing Framework Latency Decomposition

Since R-tree indexing and join query operations are at stage 4 and stage 5 of the workflow, the study further measures the *Shuffle Read Rate* and *Shuffle Write Rate* under varied partition numbers as shown in Figure 27. 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 affects to the latency of the indexing and joint query stages when increasing the partition numbers.

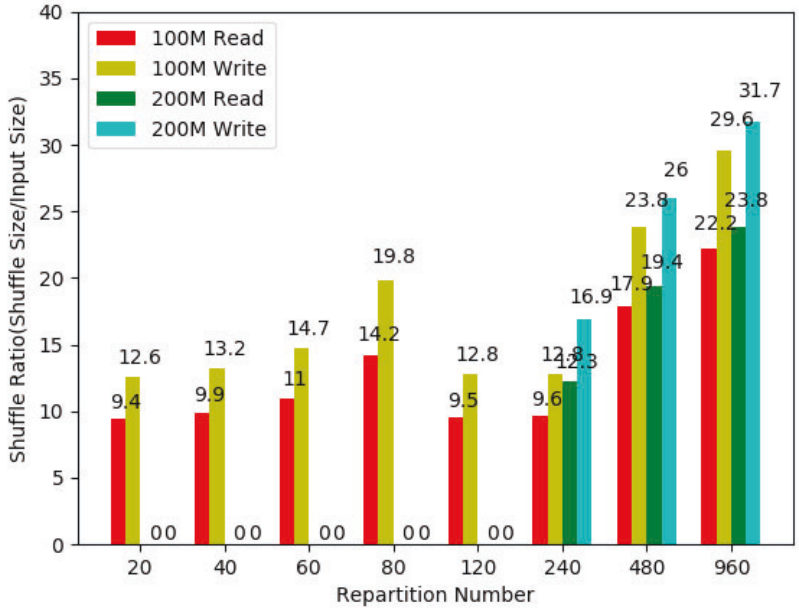


Figure 27: Segmentation Repartition Shuffling Ratio

The latency decomposition of the graph storage system is shown in Figure 28. 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.

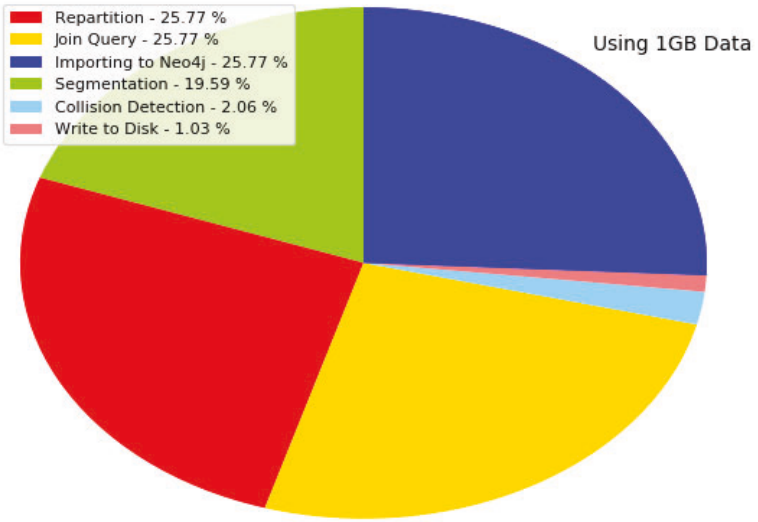


Figure 28: Graph Database Based Framework Latency Decomposition

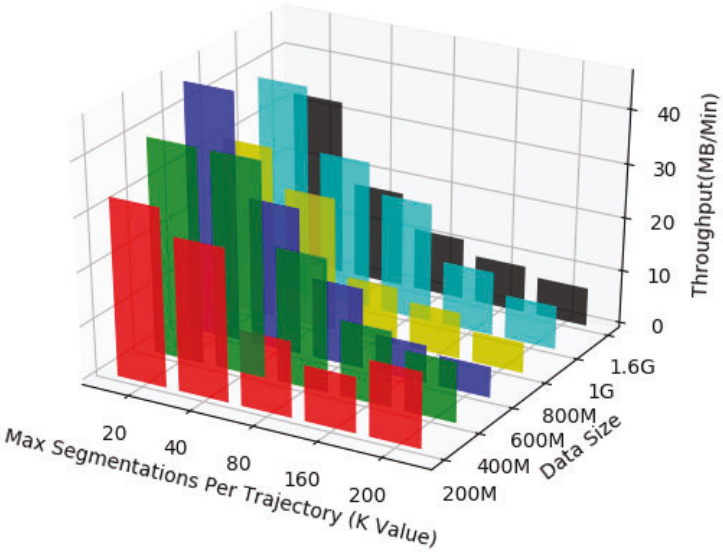


Figure 29: Neo4j 16 Node Clustering

Chapter 6

Discussion

6.1 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 [38][39][26]. 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 Figure 11 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.

6.2 Replacing Partitioning Strategy

The experiments further measure the latency and the number of MBR records in RDD processed during the join query stage. Table 5 can give an insight into the partition data size distribution and the execution latency distribution among the tasks in the 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.

Table 5: Tasks Latency and Records Distribution on Join Query Stage

Partition Method	Data Size	25 %(Records)	Median(Records)	75 %(Records)	Max(Records)
GeoSpark(R-tree only)	200M	1 S(502)	3 S(1218)	12 S(3898)	1.5 Min(26772)
GeoSpark(R-tree only)	1G	3 S(1198)	10 S(2450)	27 S(4966)	1.4 H(354524)
GeoSpark(Quad tree)(21Min Total)	200M	0.1 S(138)	0.5 S(344)	1 S(1050)	17 S(7206)
GeoSpark(Quad tree)(25Min Total)	1G	0.2 S(182)	2 S(986)	5 S(2100)	53 S(8878)
Neo4j(R-tree&Time Dimension)	200M	36 S(6408)	41 S(6266)	43 S(6657)	48 S(7208)
Neo4j(R-tree&Time Dimension)	1G	5.7 Min(40114)	6.4 Min(41410)	7 Min(42369)	7.7 Min(43180)

To solve this data skew issue, the R-tree partition strategy is replaced to the Quad-tree partition strategy [56] 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 certain spatial range, the deeper of the Quad-tree and the more partitions in this range.

Table 5 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 is mentioned in Section 5.2.1.

6.3 Introducing Time Dimension When Partitioning

The study observes increasing the number of partitions incurs uneven distribution that leads to data skew and the 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. The solution to reduce the number of partitions of a graph storage framework is introducing the time dimension as a new factor when partitioning data.

The study repartitions the MBRs in one geographical partition into multiple map layers by introducing the time dimension. 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 the representation of sub-trajectories have the property indicating when the sub-trajectories start and end. it is possible to dispense the MBR into certain map layer during the Stage 3. It follows the *replication* method in Section 3.4.1 Stage 3 when an MBR is between two map layers. The following stages treat each map layer in the graph database as an individual partition in Spark system. Figure 30 demonstrates the trajectory segments are routed to different map layers.

Table 5 shows the MBR distribution between partitions when applying R-tree and time dimension partition method in graph storage. Since the Neo4j graph database has a better ability handling large partition, the repartition number parameter is set

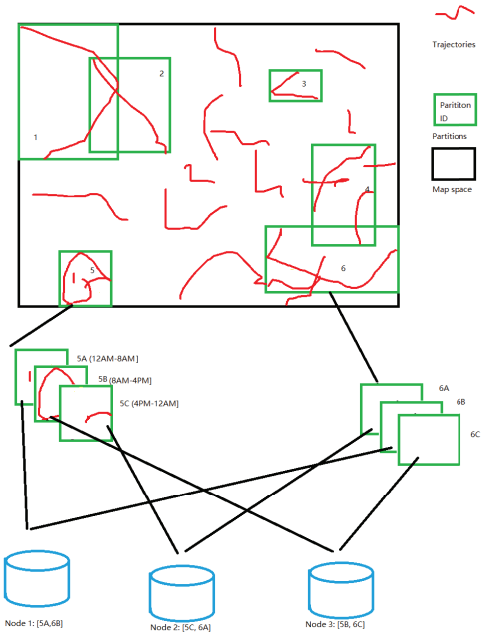


Figure 30: Multiple Map Layers Routing to Neo4j Nodes

smaller than the in-memory framework. So the record number in one partition is relatively larger compared to other partition methods based on in-memory framework. From the table shown in 1GB scenario, study reveals the largest partition is 43, 180 MBR records and the median partition record number is 41, 410, which is only 4% larger. Also, the largest partition's execution time is only 0.1 times longer than median partition. The least variance ensures all parallel tasks can complete simultaneously and gives the most efficiency.

6.4 Unaddressed Problems

Due to the time limitation, a full evaluation of the Quad-tree performance is not accomplished. The study does not test the latency using Quad-tree partition method and graph database. It does not have the time decomposition with Quad-tree partitioning workflow.

This framework is a distributed system consisting of multiple processing nodes and multiple graph storage nodes. If there is a failure on Spark processing node, the framework can recover from Spark's failover mechanism to reproduce the losing RDD.

Due to the lack of synchronization mechanism or high-availability between graph database nodes. Each database node is in standalone mode to get the maximum throughput. When facing a network partition, it is impossible to access all data partitions. We lost the accessibility of our system as CAP theorem [8] described. CAP theorem stands for **Consistency**, **Availability**, and **Partition Tolerance**.

6.5 Reliability Factors

Test-retest reliability: Due to the random sampling algorithm adopted in partitioning stage, re-run the test can not guarantee the same partition distribution as last time. Partition distribution is a big factor impacting the workflow performance.

Parallel-forms reliability: There is no standard to evaluate the cloud computing platform computing power. There is no mating to AWS M4.2XLarge instant type or similar from other cloud computing platforms like Google or Microsoft Azure. The non-universal computing node size standard limits our cross infrastructure platform test to examine our parallel-forms reliability.

6.6 Threads to Validity

History: Between two rounds of the trajectory processing tests, the Linux system cache some frequently used data as an optimization even though we delete the whole database folder and all Spark intermediate files. Building a whole new system to execute a single test and terminating it is cost-consuming. Also, the long distance network connection between S3 file storage and EMR cluster is also a concern that

we could not control. To solve this, it requires a dedicated private cluster and to reset all configurations between each round of tests.

Selection biases: When selecting the trajectory data, there is no measurement to evaluate the variance of trajectories. We just randomly select the trajectories that fulfill the required size of data. There is no more measurement against the trajectory lasting time, trajectory travel distance. So double the size of the dataset does not mean the double size of MBRs or double workload. To improve this, distribute short, medium and long trajectories in each size of test dataset on a pro-rata basis.

The trajectories are gathered from Beijing city. However, the road network in Beijing is one of the very few cities with ring roads. There are more than six ring roads in Beijing. The unique road network affects the accuracy when compare the two trajectories. Applying more trajectory data from other cities can improve this bias.

Chapter 7

Conclusion

In this thesis, a distributed trajectory segmentation framework that transforms sequences of trajectories into queryable data blocks to build trajectory analysis applications is developed.

The thesis designs the system architecture and workflows to discover trajectory patterns using both distributed in-memory processing framework and a cluster of graph database nodes.

This thesis designs the parallel trajectory segmentation algorithm based on MapReduce pattern. It is implemented on Spark which is a memory processing framework. Greedy-split algorithm is selected to transform trajectory data to indexable MBR shapes ,which is a balance between the accuracy and time complexity.

This whole system uses divide and conquer thoughts to divide a whole geographical area into multiple partitions. The experiments show two dynamic partitioning strategies one is R-tree partition and another is Quad-tree partitioning. The results show that Quad-tree has a better performance when handling data skew problem.

For acceleration of the similarity query, the R-tree indexing inside of each partition is stored in the node memory or in external database. The self-join query operation for finding similar trajectories is also implemented in both Spark in-memory framework and external Neo4j NoSQL database.

Based on the evaluation, we suggest users when facing small scales or dynamic queries for low latency processing like streaming or micro batches to use the in-memory processing framework. When the case is large scale static historical data analysis like data warehouse offline query, we suggest users using the Neo4j based

framework.

After defined two metrics of trajectories, more performance experiments are evaluated to testify how the parameters can affect the performance of the framework. The number of segments per trajectory affects the accuracy of trajectory transformation but also directly influences the raw data processing latency. Increasing the cluster node scale can get a good speedup because of the enlargement of memory. The study also evaluates how the partition numbers can affect throughput. Clusters can be fully utilized only by choosing high enough level of parallelism. However too many partitions can result in more overhead at later processing stage.

Finally, the bottleneck to higher scalability caused by data screw is observed. Accordingly, this thesis proposes a balancing method based on time dimension to adjust individual partition size and thus balance the data distribution. The study also evaluates a better partition method called "Quad-Tree" to solve the "overflow: partition skewness.

For the future work, it aims to extend this framework with streaming pipeline to handle real-time data, how to ensure the framework can recover from partial failure and use other container DevOps concepts to achieve elastic auto scaling.

Appendices

Appendix A

System Deployment

Our system is written in Java Maven. All dependencies can be handled by the repositories automatically.

To get the project, please use git took to folk the code.

Firstly, compile the modified GeoSpark with the support of Neo4j

```
1  git clone https://github.com/kanghq/GeoSpark.git
```

```
1  git clone https://github.com/kanghq/SparkApp.git
```

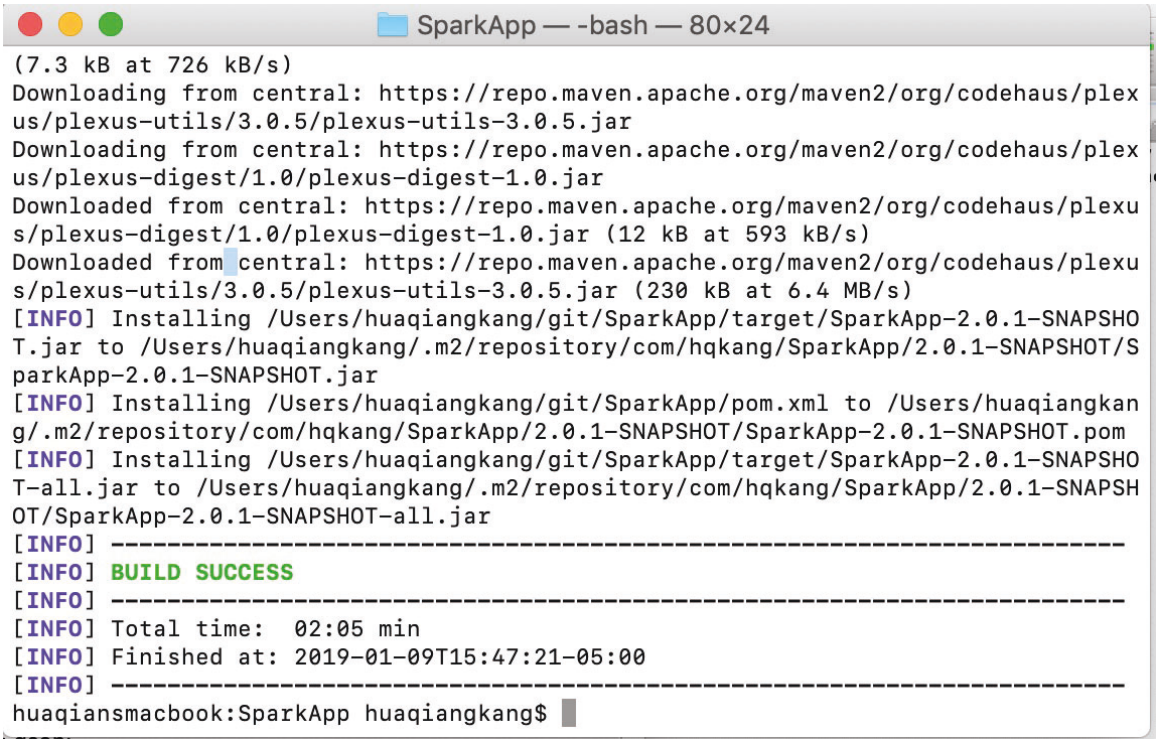
```
huaqiansmacbook:git huaqiangkang$ git clone https://github.com/kanghq/SparkApp.g
it
Cloning into 'SparkApp'...
remote: Enumerating objects: 55, done.
remote: Counting objects: 100% (55/55), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 872 (delta 6), reused 43 (delta 6), pack-reused 817
Receiving objects: 100% (872/872), 793.78 KiB | 852.00 KiB/s, done.
Resolving deltas: 100% (393/393), done.
huaqiansmacbook:git huaqiangkang$
```

If you want the support of Neo4j, include this customized package in your local repository path.

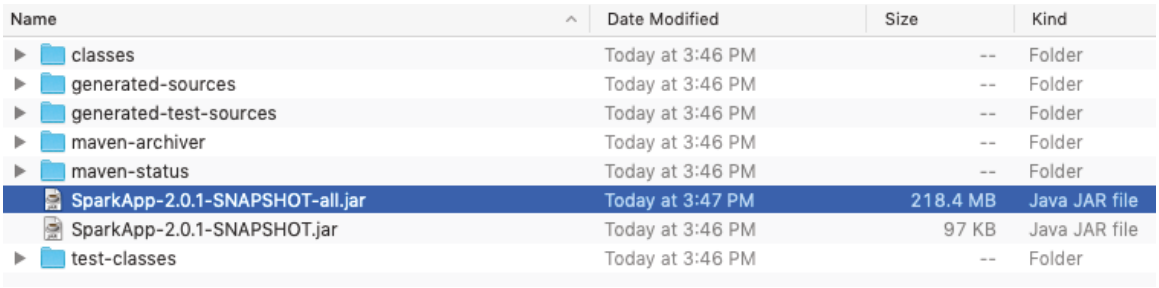
```
1  <groupId>org.datasyslab</groupId>
2  <artifactId>geospark</artifactId>
3  <version>0.6.1-hq</version>
```

Then compile the middleware. We skip the test cases to save time.

```
1  mvn install -DskipTests
```



After a successful build, we can find the jar file in the target folder.



Upload this file to AWS S3 storage.

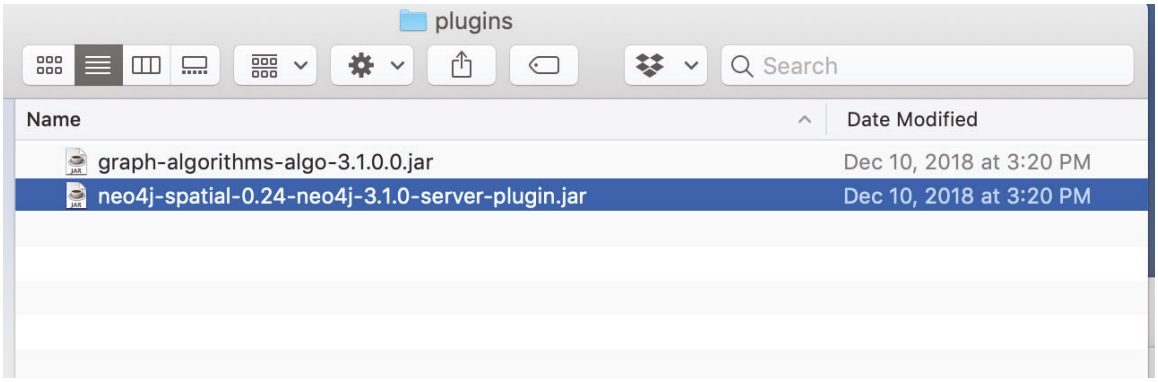
To use the graph database, download neo4j spatial plugin source code from git and switch to 3.1 branch which adds the feature with GeoSpark support.

```
1 git clone https://github.com/kanghq/spatial.git
2 git checkout 3.1
3
```

Follow the instruction to build this plugin.

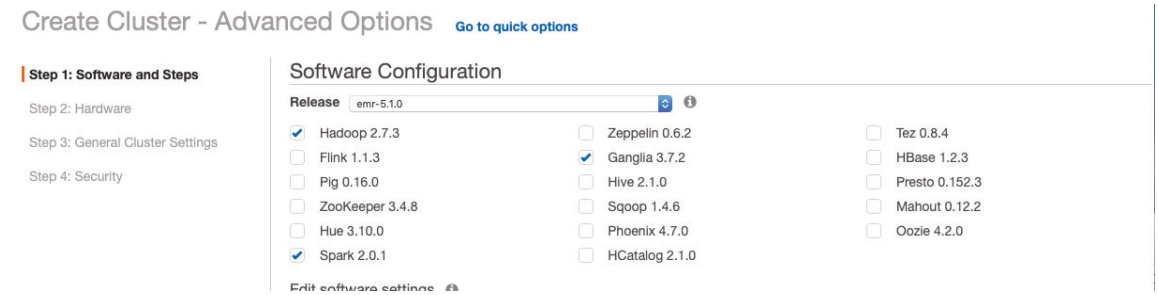
Download Neo4j Ver 3.1 Community.

Put this compiled plugin in the Neo4j plugin folder.

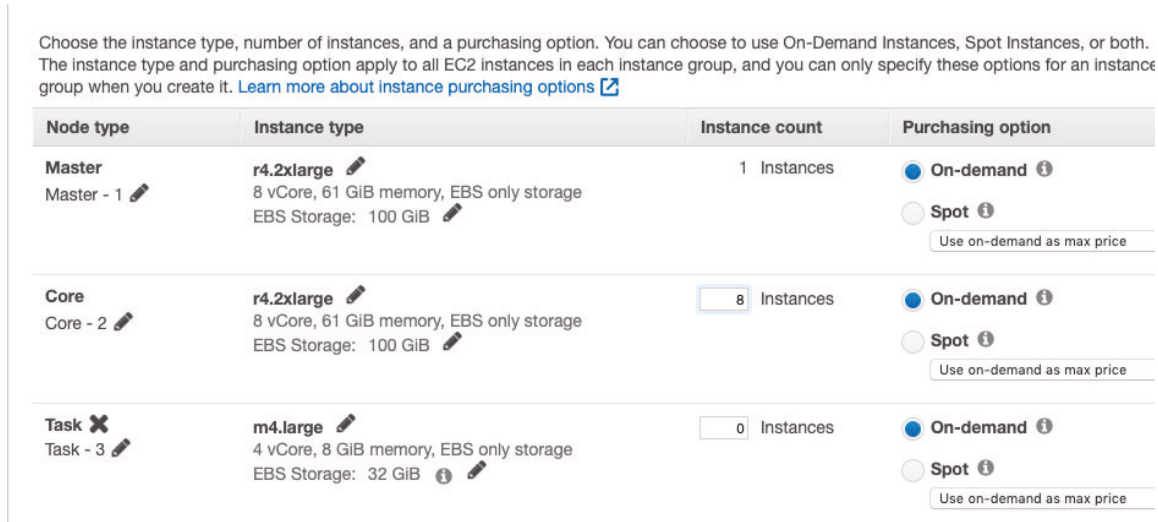


Then we log in to Amazon AWS console to start the cluster.

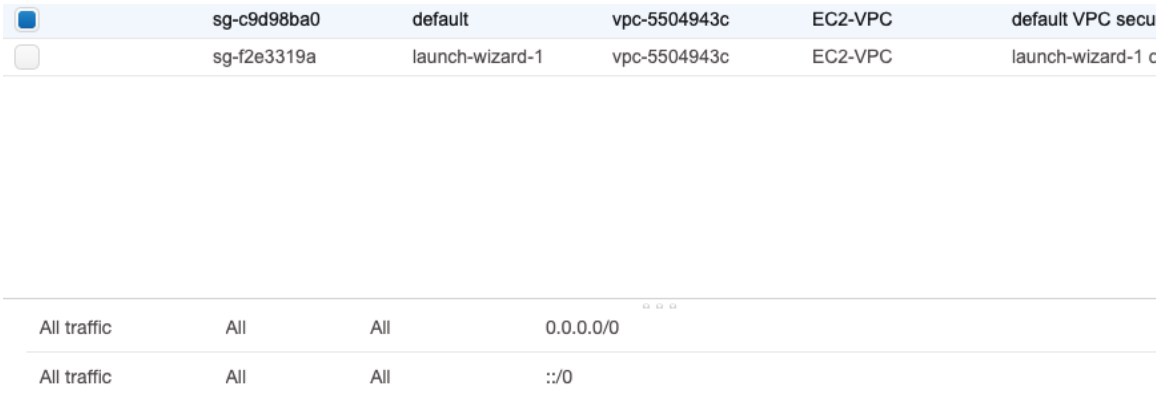
Please select the EMR version 5.1.0



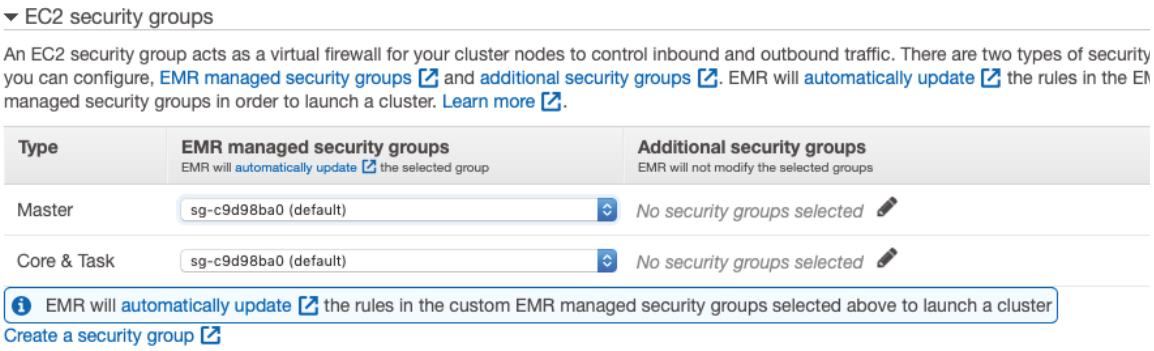
To avoid insufficient turnover disk space, please manually increase each node's EBS Storage to 100 GB.



Create a security group that allows SSH, Spark, and Neo4j to communicate with each other.



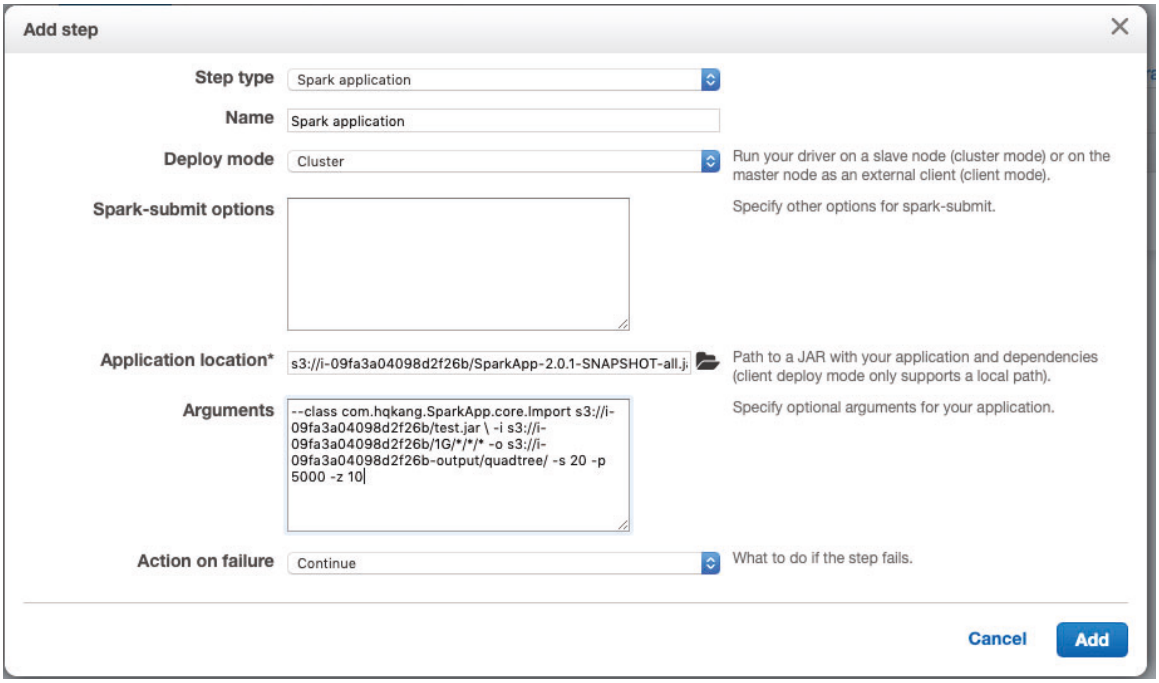
Assign this security group to both Master and core$task nodes.



Upload the Neo4j Community 3.1 software to each node and start the graph database server.

Add each work node an alternative hostname with the prefix DBSRV, for example, DBSRV1, DBSRV2 ...

Now you can submit the Spark task.


Add step
Step type
Spark application
Name
Spark application
Deploy mode
Cluster
Run your driver on a slave node (cluster mode) or on the master node as an external client (client mode).
Spark-submit options
Specify other options for spark-submit.
Application location*
s3://i-09fa3a04098d2f26b/SparkApp-2.0.1-SNAPSHOT-all.j
Path to a JAR with your application and dependencies (client deploy mode only supports a local path).
Arguments
--class com.hqkang.SparkApp.core.Import s3://i-09fa3a04098d2f26b/test.jar \ -i s3://i-09fa3a04098d2f26b/1G/*/*/* -o s3://i-09fa3a04098d2f26b-output/quadtree/ -s 20 -p 5000 -z 10
Specify optional arguments for your application.
Action on failure
Continue
What to do if the step fails.
Cancel
Add

Bibliography

[1] Sql server 2017 spatial indexes overview. Accessed: 2019-01-03.

[2] Ibm informix r-tree index user's guide, May 2008.

[3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.*, 6(11):1009–1020, August 2013.

[4] Aris Anagnostopoulos, Michail Vlachos, Marios Hadjieleftheriou, Eamonn Keogh, and Philip S.e Yu. Global distance-based segmentation of trajectories. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 34–43, New York, NY, USA, 2006. ACM.

[5] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, pages 359–370. AAAI Press, 1994.

[6] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[7] Boundless. Dimensionally extended 9-intersection model. Online:Introduction to PostGIS.

[8] Eric Brewer. Towards robust distributed systems. Keynote, July 2000.

[9] Mattia Broilo, Nicola Piotto, Giulia Boato, Nicola Conci, and Francesco G. B. De Natale. *Object Trajectory Analysis in Video Indexing and Retrieval Applications*, pages 3–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Maike Buchin, Anne Driemel, Marc Van Kreveld, and Vera Sacristán. Segmenting trajectories: A framework and algorithms using spatiotemporal criteria. *Journal of Spatial Information Science*, 2011(3):33–63, 2011.

[11] Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi. Storage and retrieval of moving objects. In Kian-Lee Tan, Michael J. Franklin, and John Chi-Shing Lui, editors, *Mobile Data Management*, pages 173–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] Simon Cox, Adrian Cuthbert, Paul Daisey, John Davidson, Sandra Johnson, Edric Keighan, Ron Lake, Marwa Mabrouk, Serge Margoulies, Richard Martell, et al. Opengis® geography markup language (gml) implementation specification, version, 2002.

[13] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 109–120. IEEE, 2010.

[14] Martin Davis. Jts topology suite - a library for geometry processing, 2011.

[15] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363, April 2015.

[16] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial partitioning techniques in spatialhadoop. *Proc. VLDB Endow.*, 8(12):1602–1605, August 2015.

[17] Ahmed Eldawy and Mohamed F Mokbel. Pigeon: A spatial mapreduce language. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pages 1242–1245. IEEE, 2014.

[18] P ERDdS and A R&WI. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei 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*, KDD'96, pages 226–231. AAAI Press, 1996.

[20] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. Spatial indexing in microsoft sql server 2008. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1207–1216, New York, NY, USA, 2008. ACM.

[21] Nivan Ferreira, James T Klosowski, Carlos E Scheidegger, and Cláudio T Silva. Vector field k-means: Clustering trajectories by fitting multiple vector fields. In *Computer Graphics Forum*, volume 32, pages 201–210. Wiley Online Library, 2013.

[22] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, March 1974.

[23] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 816–825, April 2007.

[24] Zhouyu Fu, Weiming Hu, and Tieniu Tan. Similarity based vehicle trajectory clustering and anomaly detection. In *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 2, pages II–602. IEEE, 2005.

[25] Fosca Giannotti, Mirco Nanni, Fabio Pinelli, and Dino Pedreschi. Trajectory pattern mining. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 330–339, New York, NY, USA, 2007. ACM.

[26] 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*, pages 522–533, April 2012.

[27] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[28] Antonin Guttman and Michael Stonebraker. Using a relational database management system for computer aided design data. *IEEE Database Eng. Bull.*, 5:21–28, 01 1982.

[29] J. Han, M. Song, and J. Song. A novel solution of distributed memory nosql database for cloud computing. In *2011 10th IEEE/ACIS International Conference on Computer and Information Science*, pages 351–355, May 2011.

[30] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[31] Kang Huaqiang. Spark trajectory processing repository.

[32] A. R. Jimnez, 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*, pages 135–143, March 2010.

[33] Martin Kalin. *Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction*. " O'Reilly Media, Inc.", 2013.

[34] Adam Kawa. Introduction to yarn, 2014.

[35] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: A comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 546–557, New York, NY, USA, 2002. ACM.

[36] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[37] Dongseop Kwon, Sangjun Lee, and Sukho 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*, pages 113–120, Jan 2002.

[38] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[39] Yongchul Kwon, Kai Ren, Magdalena Balazinska, and Bill Howe. Managing skew in hadoop. *IEEE DATA ENG. BULL., VOL*, 2013.

[40] Robert Laurini. Spatial multi-database topological continuity and indexing: a step towards seamless gis data interoperability. *International Journal of Geographical Information Science*, 12(4):373–402, 1998.

[41] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604. ACM, 2007.

[42] N. Magdy, M. A. Sakr, T. Mostafa, and K. El-Bahnasy. Review on trajectory similarity measures. In *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 613–619, Dec 2015.

[43] Yingchi Mao, Haishi Zhong, Xianjian Xiao, and Xiaofang Li. A segment-based trajectory similarity measure in the urban transportation systems. *Sensors*, 17(3):524, 2017.

[44] 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*, pages 935–942, June 2009.

[45] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[46] Alberto Mrquez. Geometric algorithms.

[47] Mark Needham, David Oliver, and Tomaz Bratanic. The connected components algorithm - chapter 5. community detection algorithms.

[48] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Pittsburgh, PA, USA, 1981. AAI8204168.

[49] Randal C. Nelson and Hanan Samet. A population analysis for hierarchical data structures. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 270–277, New York, NY, USA, 1987. ACM.

[50] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[51] Salvatore Orlando, Renzo Orsini, Alessandra Raffaetà, Alessandro Roncato, and Claudio Silvestri. Trajectory data warehouses: Design and implementation issues. *Journal of computing science and engineering*, 1(2):211–232, 2007.

[52] Costas Panagiotakis, Nikos Pelekis, Ioannis Kopanakis, Emmanuel Ramasso, and Yannis Theodoridis. Segmentation and sampling of moving object trajectories based on representativeness. *IEEE Trans. on Knowl. and Data Eng.*, 24(7):1328–1343, July 2012.

[53] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 395–406, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[54] Slobodan Rasetic, Jörg Sander, James Elding, and Mario A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 934–945. VLDB Endowment, 2005.

[55] John T Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981.

[56] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[57] Shashi Shekhar and Hui Xiong. *Java Topology Suite (JTS)*, pages 601–601. Springer US, Boston, MA, 2008.

[58] Ram Sriharsha. Geospatial processing made easy, Jul 2017.

[59] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20, 2011.

[60] Christian Strobl. *Dimensionally Extended Nine-Intersection Model (DE-9IM)*, pages 470–476. Springer International Publishing, Cham, 2017.

[61] L. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. Hung, and W. Peng. On discovery of traveling companions from streaming trajectories. In *2012 IEEE 28th International Conference on Data Engineering*, pages 186–197, April 2012.

[62] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proc. VLDB Endow.*, 9(13):1565–1568, September 2016.

[63] Yufei Tao. The r-tree. ITEE University of Queensland.

[64] Craig Taverner. Neo4j spatial, Jul 2018.

[65] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[66] John D. Vitek, John R. Giardino, and Jeffrey W. Fitzgerald. Mapping geomorphology: A journey from paper maps, through computer mapping to gis and virtual reality. *Geomorphology*, 16(3):233 – 249, 1996.

[67] Michail Vlachos, Marios Hadjieleftheriou, Dimitrios Gunopulos, and Eamonn 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*, KDD '03, pages 216–225, New York, NY, USA, 2003. ACM.

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

[69] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1071–1085, New York, NY, USA, 2016. ACM.

[70] 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)*, pages 2296–2304, Dec 2016.

[71] Jia Yu, Jinxuan Wu, and Mohamed 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*, SIGSPATIAL '15, pages 70:1–70:4, New York, NY, USA, 2015. ACM.

[72] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[73] 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*, 72(2):503–532, Jun 2014.

[74] Yu Zheng, Hao Fu, Xing Xie, Wei-Ying Ma, and Quannan Li. *Geolife GPS trajectory dataset*, July 2011.

[75] Yu Zheng, Nicholas Jing Yuan, Kai Zheng, and Shuo Shang. On discovery of gathering patterns from trajectories. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 242–253, Washington, DC, USA, 2013. IEEE Computer Society.