applied sciences

MDPI

Article

# A Resilient Large-Scale Trajectory Index for Cloud-Based Moving Object Applications

**Omar Alqahtani * and Tom Altman**

Department of Computer Science and Engineering, College of Engineering, Design and Computing, University of Colorado Denver, Denver, CO 80204, USA; tom.altman@ucdenver.edu

* Correspondence: omar.alqahtani@ucdenver.edu; Tel.: +1-720-238-3808

Received: 1 September 2020; Accepted: 13 October 2020; Published: 16 October 2020


**Abstract:** The availability of location-aware devices generates tremendous volumes of moving object trajectories. The processing of these large-scale trajectories requires innovative techniques that are capable of adapting to changes in cloud systems to satisfy a wide range of applications and non-programmer end users. We introduce a Resilient Moving Object Index that is capable of balancing both spatial and object localities to maximize the overall performance in numerous environments. It is equipped with compulsory, discrete, and impact factor prediction models. The compulsory and discrete models are used to predict a locality pivot based on three fundamental aspects: computation resources, nature of the trajectories, and query types. The impact factor model is used to predict the influence of contrasting queries. Moreover, we provide a framework to extract efficient training sets and features without adding overhead to the index construction. We conduct an extensive experimental study to evaluate our approach. The evaluation includes two testbeds and covers spatial, temporal, spatio-temporal, continuous, aggregation, and retrieval queries. In most cases, the experiments show a significant performance improvement compared to various indexing schemes on a compact trajectory dataset as well as a sparse dataset. Most important, they demonstrate how our proposed index adapts to change in various environments.

**Keywords:** moving objects; big data; spatial indexing; machine learning for indexing

## 1. Introduction

Enormous volumes of moving object trajectories are generated rapidly due to the availability of low-cost geospatial chipsets that can take advantage of the advanced technologies used in many fields. In particular, GPS, which has become ubiquitous due to the growth of embedded systems and increased use of electronic gadgets, creates massive moving object trajectories. Most of our daily devices (e.g., smartphones, smartwatches, navigation systems, tablets, etc.) are able to accurately pinpoint our location. As a result, they open new horizons, and many wide-ranging commercial applications have become feasible. Ridesharing (e.g., Uber, Lyft, etc.) is a distinct example of the influence of the location-aware devices on transportation services. These applications rely on the availability of smartphones and wireless networks to automate a procedure that used to require human interaction. Moreover, new services such as electric bike and scooter rentals, carsharing, security, and monitoring are also leveraging the use of GPS tracker devices. Nowadays, most corporations' fleet vehicles use real-time GPS trackers to maximize efficient use of resources. As a result, tremendous historical moving object trajectories are produced on a scale that requires innovative storing and processing techniques.

Historical moving object trajectories are fundamental in studying and analyzing numerous fields, such as smart cities, human crowds, navigation, animal migration, etc. They play a significant role in planning smart cities by analyzing trajectory-driven factors related to environmental or economic

issues, such as congestion and collision hotspots. They can be employed to improve city sustainability. Moving object trajectories are used in crowd behavior and movement studies that aim to enhance crowd management and minimize areas of contention or crisis. Transportation services and smart navigation heavily depend on moving object trajectories, which can be used to analyze a hotspot area and recommend routing based on specific preferences, such as time of arrival or green routing. The trajectories can also be used by wildlife biologists who study the impacts of urban infrastructure on wildlife habitats and/or animal migrations.

As a consequence, advanced techniques and large-scale computing platforms have become a necessity to cope with storing and processing vast volumes of big spatial data [1]. MapReduce is a typical framework for distributed computations, and it is used by many large-scale data processing platforms like Apache Spark [2]. Spark is a general purpose in-memory computing platform that is supported by most of the major cloud computing systems (e.g., Amazon AWS, Google Cloud Engine, IBM Cloud, Microsoft Azure, Cloudera, etc.).

However, the adoption of distributed platforms and cloud systems, the nature of the trajectories, and querying diversity create many challenges and obstacles to process large-scale trajectories. In general, cloud systems provide a wide range of configuration options for CPU, memory, and networking capacity, which require any running framework to be fine-tuned for maximizing the benefits of the available resources. Additionally, moving objects reflect the space topology, which might accumulate as a scattered or compacted dataset. Selective queries are mostly related to self-skewed trajectories and cause computation skewness, which affects performance by reducing cluster utilization, i.e., creating hotspots within the cluster. Some queries on compact trajectories, especially multistage and aggregation queries, require tremendous amounts of communications, which can create performance bottlenecks. On the other hand, space-based queries on a sparse dataset often do not utilize the available parallelism capacity, especially on small spatial selectivity.

Our goal is to develop a resilient index that is capable of adapting in different cloud environments and can be used by the end-users without the need for fine-tuning. Three factors are essential for an adaptive index: computational resources, nature of the data, and query types. An adaptive index should be able to maximize the benefit from the available computing resources (e.g., number of nodes, memory size, number of cores, etc.) and build the index structure accordingly. For example, suppose there are two clusters and the first one supports more parallel computation. The index structure on the first cluster needs to maximize the cluster utilization more than the index on the second cluster if we consider the second index as the baseline. This could be resolved by increasing the number of memory partitions, especially when space-based queries are the majority. However, creating too many partitions and exceeding the parallelism capacity could cripple the performance of the index by increasing the memory access times.

Additionally, an adaptive index should be able to recognize the fundamental characteristics of a historical trajectory dataset, such as sparseness, geographic topology and scope, or moving object classes. For instance, object-based queries on compact trajectories tend to have more data transportation compared to those on sparse trajectories, which might cause a communication jam and dramatically affect the system performance. Finally, the index has to perceive the needs of each query type with a given resource and dataset, and it needs to comprehend the different impacts of query types in the long run and optimize the structure accordingly. In order to have effective adaption capability, the index needs to consider a combination of the three aspects and find a middle ground in contradictory or ambiguous situations.

In this paper, we propose a Resilient Moving Object Index (RMOI) for in-memory processing of large-scale historical trajectories. RMOI achieves the adaptation capability by controlling the major participants' localities. It uses two novel machine learning models to predict the locality pivot $LP$, which is used to balance the spatial and object localities. The prediction depends on the adaptation factors of availability of resources, nature of the trajectories, and query types in order to maximize the

overall performance in any cloud environment. *LP* boosts RMOI's flexibility, thus making it suitable for a wide range of analytic applications and queries, and thereby more appealing for cloud platforms.

RMOI is equipped with two prediction models: a Compulsory Prediction Model (CPModel) and a Discrete Prediction Model (DPModel). The main goal of both models is to predict a convenient *LP* where each model is suitable for specific situations. The models are trained on real trajectory datasets and comprehensive independent variables (features). The derived features are proportional factors, which cover different aspects without being tied-down to specific factors such as memory size, number of worker nodes, etc. In addition, we introduce a query Impact Factor Model (IFModel), which is responsible for predicting the computational requirement of a given query type. It helps DPModel to focus on the most expensive queries for a long-run scenario.

The absence of sufficient training sets led us to conduct large-scale experiments to generate them. We used different combinations of features and *LP* values, which generated more than 66,000 result factors. Moreover, we conducted extensive performance experiments to test our proposed approach. The experiments were designed to evaluate each prediction model when compared to cutting edge indexes. We included stress testing for a more realistic scenario that reveals the complications of dealing with a pack of query types instead of narrowing down to a specific type. We covered the essential queries of the space-based, time-based, and object-based query types. Our evaluation considered two different system settings: high and low availability of resources. In most cases, the experiments showed significant performance improvement.

The main contributions of this work are as follows.

- We introduce RMOI as an adaptive index for analytic applications.
- We develop two novel machine learning models—CPModel and DPModel—to control both spatial and object localities through *LP*.
- We also develop an IFModel, which predicts a query impact factor.
- We provide a framework to extract proportional features and generate training sets.
- We formalize spatial, temporal, spatio-temporal, continuous, aggregation, and retrieval queries. Furthermore, we provide efficient query processing algorithms.
- We evaluate our work by conducting extensive performance experiments comparing varies indexing schemes. The experimental study includes two testbeds on three datasets.

The related work is discussed in Section 2. The index structure and the prediction models are introduced in Section 3. The query processing algorithms are presented in Section 4. An extensive experimental study and our concluding remarks are discussed in Sections 5 and 6, respectively.

## 2. Related Work

Spatio-temporal data can be divided into three main groups: historical, current, and future data. Each group represents different obligations that result in different indexing and queries. Our work is only focusing on historical trajectories. From the computing platforms perspective, we classify the prior work on historical data into three subgroups: centralized systems, parallel database systems, and MapReduce-based systems. Before discussing these specific bodies of literature, we first review some of the access methods and index structures used in most of the related work.

### 2.1. Access Methods

Hierarchical trees are among the most used access methods. In general, spatial access methods depend on object grouping or space splitting techniques. R-tree [3] and its variants, such as R*-tree [4] and $R^+$-tree [5], depend on grouping the objects in a minimum bounding rectangle (MBR) in a hierarchical manner. Simple grid, k-d-tree [6], and its variants (e.g., k-d-B-Tree [7] and Quadtree [8]) depend on space-splitting instead. In this method, the overlapped objects are duplicated or trimmed. Otherwise, an overlapping enlargement is enforced.

However, a moving object trajectory is an example of time-series spatial data. As a result, many versions of the previous structures had to be adapted for moving object trajectories.

These structures can be grouped into augmented multidimensional indexes or multi-version structure indexes. Augmented multidimensional indexes can be built using any of the previous hierarchical indexes (in practice, mostly R-trees) with augmentation on the temporal dimension, as seen in Spatio-Temporal R-tree and Trajectory-Bundle tree (TB-tree) [9]. A Spatio-Temporal R-tree keeps segments of a trajectory close to each other, while a TB-tree ensures that the leaf node only contains segments belonging to the same trajectory, meaning that the whole trajectory can be retrieved by linking those leaf nodes together. On the other hand, multi-version indexes, such as Historical R-tree (HR-tree) [10], rely mostly on R-trees to index each timestamp frame. Then, the resulting R-trees are also indexed by using a 1-d index, such as a B-tree. Nodes that are unchanged from time frame to time frame do not need to be indexed again. Instead, they will be linked to the next R-tree.

### 2.2. Centralized Systems

In centralized systems, the authors of [11] implement an in-memory two-level spatio-temporal index, where the first level is a $B^+$ tree on temporal windows. The second level consists of an inner R-trees with two bulk-loading techniques. GAT [12] uses centralized architecture to process a top-k query on activity trajectories, where the points of the trajectory represent some set of events, such as tweeting or posting on Facebook. It uses a simple grid to partition the space and some auxiliary indexes to process the events and trajectories. Scholars have also focused on a specific query type, such as the work in [13], which divides the road network into sub-graphs based on the position of interests for efficient time-period most-frequented path query.

### 2.3. Parallel Databases

On the other hand, the authors of [14] implement a parallel spatial-temporal database to manage both network transportation and trajectory and to support spatio-temporal SQL queries. They use a space-based index that partitions the data based on a space-splitting technique. Any trajectory that crosses a partition boundary is split into sub-trajectories, whereas any sector of the transportation network that crosses a partition boundary is replicated in all of the crossed partitions. Grid indexes [15,16] that function as in-memory grid indexes are designed for thread-level parallelism. TwinGrid [15] maintains two grids for moving object updates and queries. However, PGrid [16] depends on a single grid structure for running queries and data updating, where it is optimized for up-to-date query results and relies on an atomic concurrency mechanism.

### 2.4. MapReduce-Based Contributions

SpatialHadoop [17], an extension of Hadoop, is designed to support spatial data (Point, Line, and Polygon) by including global and local spatial indexes in order to speed up spatial query processing for range queries, k-Nearest Neighbors (k-NN), spatial join, and geometry queries [18]. Hadoop-GIS [19] extends Hive [20], a warehouse Hadoop-based database, to process spatial data by using a grid-based global index and an on-demand local index. A Voronoi-based index is used in [21] to process the nearest neighbor queries. However, none of the previous systems support trajectories directly. ST-Hadoop [22] extends SpatialHadoop to support spatio-temporal data. It partitions the data into temporal slices and constructs a SpatialHadoop index on each slice. PRADASE [23] concentrates on processing trajectories, but it only covers range queries and trajectory-retrieve queries. It partitions space and time by using a multilevel grid hash index as a global index where no segment crosses the partition boundary. Another index is used to hash all segments on all the partitions belonging to a single trajectory to speed up the object retrieving query. Nevertheless, all Hadoop-based contributions inherit the continuous disk access drawback.

GeoSpark [24] is implemented on top of Spark, and it is identical to SpatialHadoop in terms of indexing and querying. LoctionSpark [25] reduces the impacts of query skewness and network communication overhead. It tracks query frequencies to reveal cluster hotspots and cracks them by repartitioning. Network communication overhead is reduced by using the embedded bloom filter

technique in the global index, which helps avoid unnecessary communication. SpatialLocation [26] is designed to process the spatial join through the Spark broadcasting technique and grid index. However, the trajectories are not directly supported by any of the previous contributions. DTR-tree [27] uses an R-tree as a local and global index where the data partitioning only depends on one dimension. The work in [28] processes the top-k similarity queries (a trajectory-based query) by using a Voronoi-based index for spatial dimension, where each cell is statically indexed on the temporal dimension. Any trajectory that crosses a partition boundary is split, and all segments belonging to that trajectory are traced with Trajectory Track Table. SharkDB [29] indexes trajectories based only on time frames in a column-oriented architecture to process range query and k-NN. TrajMesa [30] provides a key-value compressed horizontal storage scheme for large-scale trajectories based on GeoMesa [31], an open source suite of tools for geospatial data. The key is produced based on temporal indexing and spatial indexing which duplicates the data table. DITA [32] only focuses on trajectory similarity search and join queries by leveraging a trie-like structure on representative points at the local index. The global index uses a packed R-tree on the first points of trajectories and then the last points to increase object locality.

Generally, most of the prior work has focused on static spatial data (Point, Polygon, and Line), which does not sufficiently account for moving object trajectories. On the other hand, the research focusing on trajectories has relied on spatial or temporal distribution, i.e., data distribution depends on partitioning space and time dimensions. Most often, the resulting distribution can partially preserve spatial and object localities, but will not provide a mechanism to control both of them.

In our previous work [33], we proposed a Universal Moving Object index (UMOi) that is capable of controlling spatial and object localities. There, the locality preservation degree needs to be set manually by the end-user, i.e., the index will not be able to self-adjust to the best locality preservation degree. It requires fine-tuning while considering the resources availability and the nature of the data and queries. The global index of UMOi depends on separated structures of k-d-B-tree and hash table. The pruning of partitions during a Spark job depends on an internal pruning mechanism and is achieved by tracking local trees' identifiers and updating the hash table accordingly. Neither the global index nor the local index supports the temporal dimension. Consequently, UMOi does not support temporal queries. For trajectory retrieving queries, UMOi depends on a secondary index that scans the whole dataset to index objects based on their identifiers.

On the other hand, RMOI uses machine learning models to predict the best locality pivot based on many factors. The global index of RMOI is a single hierarchical structure, where the top part follows a k-d-B-tree splitting mechanism and the bottom part is an R-tree. Both global and local indexes support temporal indexing and querying. RMOI uses different partitions pruning that depends on the Spark built-in filter transformation. In addition, it leverages the partitioning mechanism to serve trajectory retrieving queries without using a secondary index.

Finally, the locality is an essential key to improve the performance and enhance the adaptation capability. The nature of a trajectory (consecutive timestamped spatial points) creates contradictory domains, which can be seen in spatial locality and object locality. As a result, some of the previous contributions optimize their systems to contain this contradiction by focusing on spatial locality and spatial queries (e.g., range query, k-NN, etc.) with an object-based auxiliary index, or by narrowing it down to a particular operator and building an ad hoc index for that purpose. To the best of our knowledge, no work has been conducted on trajectory indexing for distributed environments that would simultaneously balance the losses and gains of both localities and target a large variety of queries.

## 3. Resilient Moving Object Index

In this section, we present our adaptive approach for trajectory indexing. Based on the work in [34], most of the MapReduce spatial indexes generally follow three steps: partitioning, building the local index, and building the global index. In the partitioning phase, data is partitioned into smaller pieces based on specific measures such as space, time, object, etc. Then, the resulted partitions are

distributed to the worker nodes and each worker node builds a local index for each partition. Finally, the driver node collects the needed information from worker nodes to build the global index.

The main goal of RMOI is to have a flexible index that considers both space-based and object-based partitioning techniques. It is capable of balancing both spatial and object localities by providing a locality preservation mechanism, which gives the flexibility to satisfy different applications' demands. RMOI uniquely provides a Locality Pivot (*LP*) parameter that is predicted based on several different features. Consider the trajectory set in Figure 1 with $\alpha = 3$ and $\beta = 2$, where $\alpha$ is the number of the required spaced-based partitions, and $\beta$ is the required object-based partitions per $\alpha$. RMOI starts by partitioning the global space into $\alpha$ spatial groups. Then, each spatial group is hashed into $\beta$ partitions. The total partitions number (*tpn*) is 6. The result is a combination of spatial and object partitioning, which provides a balance of both localities simultaneously.

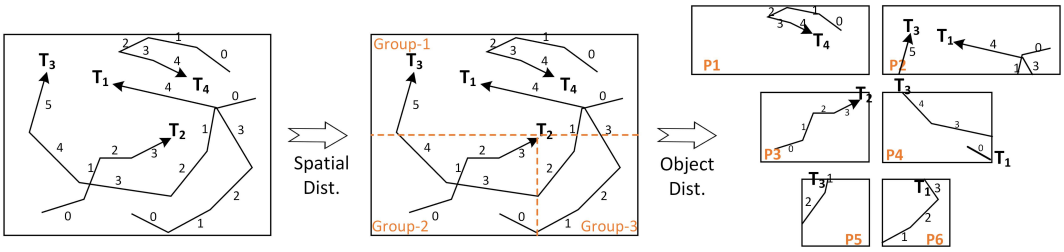


**Figure 1.** Partitioning phase in Resilient Moving Object Index (RMOI) with $\alpha = 3$ and $\beta = 2$.

### *3.1. Index Structure*

RMOI consists of a global index, a local index, and prediction models. Each node of the global index, denoted by GlobalRMOI, contains a Minimum Bounding Rectangle (MBR) and time interval, as illustrated in Figure 2. The MBR is used to specify the minimal spatial area covered by the contained sub-trees, while the time interval is also used to specify the minimum time range. Each leaf node has the corresponding partition identification number (*Pid*), and there are *tpn* leaf nodes. GlobalRMOI is influenced by the partitioning mechanism, which depends on the value of the *LP*. In general, GlobalRMOI is a combination of k-d-B-tree [7] and R-tree [3]. However, when $\beta = 1$, GlobalRMOI is similar to k-d-B-tree, and it only reflects the space-based partitioning. Comparing to the rest $\beta$ values, this scenario gives the maximum spatial-locality and the least object-locality. Alternatively, when $\alpha = 1$, R-tree dominates the structure of the GlobalRMOI, as this is a scenario when it only depends on object-based partitioning. In this case, RMOI guarantees full trajectory-preservation where all the segments (*Segs*) of a trajectory (*Traj*) reside in one partition, which equals one GlobalRMOI leaf node.

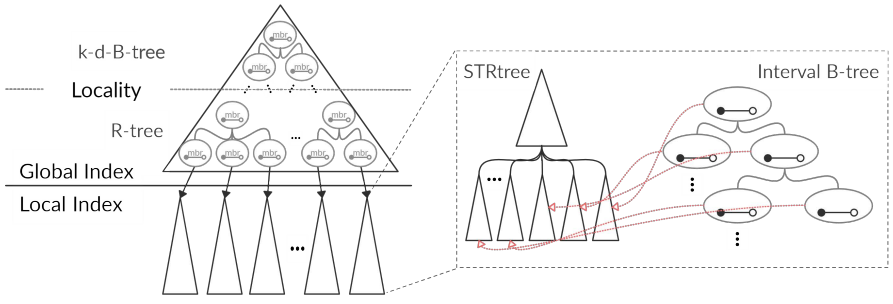


**Figure 2.** RMOI structure.

The local index, denoted by LocalRMOI, consists of STRtree [35] and interval B-tree [36]. STRtree is a packed R-tree that uses the Sort-Tile-Recursive (STR) algorithm. Unlike the GlobalRMOI, the LocalRMOI favors the temporal dimension, where the interval B-tree is kept separately from STRtree. Each interval node is associated with a subtree of the local STRtree allowing us to traverse the local index based on the interval B-tree, as seen in Figure 2.

The prediction model is one of the essential components in RMOI. It is responsible for capturing the differences in computation resources, storage availability, nature of data, and query types. The goal of the prediction model is to determine the value of the Locality Pivot ($LP$) in constant time without an overhead on top of the index construction. RMOI uses polynomial regression on different features by applying a low-degree polynomial transformation. The model consists of two parts: a Compulsory Prediction Model (CPModel) and a Discrete Prediction Model (DPModel). The CPModel is used as a cold start when there is no hint of the incoming query types, while DPModel takes into consideration the frequency of requests for various query types and their impacts. The DPModel depends on another model, the query Impact Factor Model (IFModel), when there is more than one query type. The IFModel is used to distinguish between query types and determine the heaviness of each type. Both models are already trained on real datasets and do not need to be fit again. For simplicity, RMOI only incorporates the final polynomial equations without the need for further training.

### 3.2. Prediction Models

Here, we discuss the features and the dependent variable ($LP$). Next, we will explain the training datasets and how they are extracted. Finally, we will talk about our models' training and how we plan to test them.

#### 3.2.1. Features of the Models

The features of a model are crucial for making an accurate prediction. After testing and analyzing different direct and derived features, we found that Computation Power Ratio (ComR), Memory Usage Ratio (MemR), and Trajectory Overlapping Indicator (TOver) are the most influential features for our models. All of the selected features depend on the proportionalities between two or more factors, which gives an advantage of not being tied to a specific value or size. Moreover, it reduces the number of features, which might have a negative impact on the regression. The polynomial regression needs to transfer the features into higher-degree ones (polynomial features) based on the polynomial degree. The number of the higher-degree features scale polynomially with respect to the number of the features and exponentially with respect to the degree. By only focusing on the proportional features, we decrease the computational overhead of predicting $LP$.

ComR is computed as follows,

$$ComR = \frac{tpn}{ExecCoreNum \times ExecNum} \ , \tag{1}$$

where $ExecCoreNum$ is the number of cores with respect to Spark's definitions, and $ExecNum$ is the number of cluster executors (workers). ComR reflects the computation power of a cluster available to a specific task. As a result, it eliminates the need to specifically report the cluster scale and the data size.

MemR reflects the ratio of the used memory by a specific task. It depends on the data size, which we could obtain by using file system calls. However, to speed up the process, RMOI depends only on a sample set ($ST$) of trajectories. It estimates the data size based on a segment size and $ST$:

$$MemR = \frac{(s'/f) \times Seg.Size}{ExecMemSize \times ExecNum} \ , \quad (0 < f \leq 1), \tag{2}$$

where $s'$ is the number of $Segs \in ST$ and $f$ is the sampling fraction.

TOver is an indicator that reveals the nature of the trajectories. It is the ratio of the trajectories' MBRs with respect to the global space. For example, a set of sparse trajectories yields a small percentage score on TOver, whereas TOver increases for a compact set of trajectories. TOver is the only feature to capture the differences between trajectory datasets. While the previous features are computed in constant time by plugging in the configuration parameters, TOver needs to scan $ST$ as follows,

$$TOver = \frac{\sum_{i=1}^{m'} Traj_i.MBR}{m' \times ST.MBR} \quad , \tag{3}$$

where $m'$ is the number of $Traj \in ST$. However, the $ST$ only contains $Segs$; therefore, it needs to scan $ST$ first to compute the MBR of each $Traj$. To keep the complexity linear, we only scan $ST$ once and use a hash set based on $Seg.Tid$ to update the MBRs.

In addition, it is worth mentioning the relationship between $\alpha$ and $\beta$, as they represent the label value for the training data. It is evident that $\alpha \times \beta = tpn$. Thus, to reveal the performance difference between $\alpha$ and $\beta$, we need to fix $tpn$. Most of the time, the system shows a noticeable change in running time only when we double their values. This is due to the uniform merging of partitions. For example, suppose we have 32 partitions where $\beta = 8$ and $\alpha = 4$. Now, we want to increase $\beta$, which means decreasing $\alpha$ to keep the same $tpn$. Decreasing $\alpha$ means merging at least two spatial groups (MBRs) into one. However, we do not want to affect only part of the global space. We apply a consistent merging of the spatial groups, which will result in reducing $\alpha$ by half to 2 and doubling $\beta$ to 16. On the other hand, using $\alpha$ or $\beta$ as a label when training the prediction model will not be smooth because of the doubling in their values. For instance, following the previous example, the possible values for $\alpha$ or $\beta$ are 1, 2, 4, 8, 16, or 32. Sometimes, the doubling in the label value affects the regression. Therefore, we use $LP$, which represents the states of changing rather than the actual values. We compute $LP$ as follows,

$$LP = 1 - \log_2 \frac{\beta}{tpn} \quad . \tag{4}$$

The range of $LP$ is $\{1, 2, 3, \dots\}$. When $LP = 1$, it means $\beta = tpn$ where the index guarantees a full trajectory-preservation.

#### 3.2.2. Training Sets

One of the obstacles in our work is the absence of available training data. In response, we have generated our own training sets. Each training set should give the best $LP$ for any possible configuration. To find the best $LP$, we need to have the result for all the possible $LP$ values. Therefore, we conducted substantial experimentation on two real datasets with all the different feature values on every $LP$ to generate the training sets.

**Table 1.** Moving object trajectory datasets used in training and evaluation.

| Dataset | Trajectories | GPS Points | Ellipsoid Area | Data Type | Moving Object Class |
|---|---|---|---|---|---|
| GT | 163 K | 132 M | 636,408 km$^2$ | real | mixed |
| UK + IE | 149 K | 179 M | 937,586 km$^2$ | real | mixed |
| RioBuses | 296 K | 114 M | 6588 km$^2$ | real | 1 |
| SF | 140 K | 119 M | 10,721 km$^2$ | generated | 20 |

We used the German Trajectory dataset (GT) [37] and the RioBuses dataset [38] described in Table 1. The GT dataset, extracted from Planet GPX of OpenStreetMap database [39], contains spatio-temporal moving object trajectories and covering Germany and parts of its neighboring countries. It contains trajectories for different moving object classes such as vehicles, humans, airplanes, and trains. The RioBuses dataset is generated by the public buses of the city of Rio de Janeiro, Brazil. The TOver for RioBuses is 0.102, which is high compared to the TOver of GT 0.009. The reason for choosing these two datasets is because of the contrast between them in the global space scope. Moreover, we cover most of the essential queries of space-based, time-based, and trajectory-based query types as we will discuss them further in Section 4. The complete list of selected query types is as follows.

- Lookup Query: contains 20 random *Tids*
- Aggregation Query: uses trajectory length as the aggregation function which returns the longest trajectory
- Range Query: contains 100 random queries with 0.1%, 0.3%, 1%, 10%, and 30% spatial selectivity
- Interval Query: contains 100 random queries with 0.1%, 0.3%, 1%, 10%, and 30% temporal selectivity
- Small Selectivity k-Continuous Range Query: contains 50 random queries with 0.3% spatial selectivity on k = 2, 3, 4, 5, and 6
- Large Selectivity k-Continuous Range Query: contains 50 random queries with 10% spatial selectivity on k = 2, 3, 4, 5, and 6
- Small Selectivity k-Continuous Interval Query: contains 50 random queries with 1% temporal selectivity on k = 2, 3, 4, 5, and 6
- Large Selectivity k-Continuous Interval Query: contains 50 random queries with 10% temporal selectivity on k = 2, 3, 4, 5, and 6
- Small Selectivity k-Continuous Spatio-Temporal Query: contains 50 random queries with 0.3% spatial selectivity and 1% temporal selectivity on k = 2, 3, 4, 5, and 6
- Large Selectivity k-Continuous Spatio-Temporal Query: contains 50 random queries with 10% spatial selectivity and 10% temporal selectivity on k = 2, 3, 4, 5, and 6.

Moreover, the features are set to cover the maximum, medium, and minimum values. Therefore, we set ComR to 2, 4, and 8. Furthermore, we set MemR to 0.16, 0.31, and 0.62. The TOver scores, as we mentioned before, are 0.009 and 0.102 for GT and RioBuses, respectively. *LP* takes the values from 1 to 7, which represent 1, 2, 4, 8, 16, 32, and 64 for $\beta$. The *tpn* for both datasets is 64 partitions. We only changed one value at a time, which leads to 126 different combinations. We use the same system settings detailed in Section 5.1.

Next, we processed the raw results of 126 runs. They contained more than 66k result factors. The goal was to generate three training sets for the CPModel, DPModel, and IFModel. The first set was used to train the CPModel, which is used when there is no prior information about the queries and their frequencies. We analyzed the result for all the query types and selected the proper *LP* for each feature combination. The selected *LP* is expected to be reasonably suitable for all the query types even if it is not the absolute best choice for a particular query. We considered the running time as the primary performance indicator. The second training set was used to train the DPModel, where the queries and their frequencies are known. We selected the best and the second-best values of *LP* for each query type (10 query types) on every feature combination. The IFModel model is used to predict the impact factor (iFactor) that is used to adjust each query type's frequency. We used the lookup query as the baseline since it is the lightest type in all of the different runs. For each feature combination, we take the ratio of other types to lookup query, and that serves as the label for the IFModel's training set.

#### 3.2.3. Training the Models

All of the models (CPModel, DPModel, and IFModel) are polynomial regression models. Polynomial regression is a special case of multiple linear regression where the features are modeled in the $d$th polynomial degree. To train the models, we first transfer the features into degree 2 for CPModel and degree 3 for DPModel and IFModel. The total number of the higher-degree features is $(d+n)!/(d! \times n!)$, where $d$ is the degree number, and $n$ is the number of features. However, we only have three features on low polynomial degrees, which is another advantage of using proportional features. Moreover, using low degrees ensures we avoid the risk of overfitting. Next, we use sklearn library to fit a linear model that depends on minimizing the residual sum of squares. Finally, we generate the corresponding coefficients, which are traveled to RMOI to be used for prediction.

We evaluated all the index components together, including prediction models. Our main goal is to build a resilient index by controlling spatial and object localities based on the prediction outcomes. Therefore, it is meaningful to test the produced index against state-of-the-art indexes.

### *3.3. Index Construction*

Algorithm 1 outlines the fundamental steps for RMOI construction, which consists of four main stages: prediction, partitioning, building the local index, and building the global index. However, before explaining these stages, we need to discuss some essential steps. We highlight the worst case upper bound of the time and space complexity for the index construction stages.

**Algorithm 1:** Index Construction.

**Input:** Trajectory Dataset T
**Output:** RMOI

$ST \leftarrow$ Sample.$T$
$ComR, MemR, TOver \leftarrow$ `ComputeFeatures(`$ST, CoresNum, ExecNum, ExecMem, \ldots$`)`
**if** *PredictionMode* $=$ *Compulsory* **then**
  $LP \leftarrow$ `CPModel(`$ComR, MemR, TOver$`)`
**else**
  $LP \leftarrow$ `DPModel(`$ComR, MemR, TOver$`)`
$\alpha, \beta \leftarrow$ `Compute(`$LP$`)`
*SK-tree*.build($ST, \alpha, \beta$) `// leaf nodes' ids start from 0 and increment by` $\beta$
**Transformation** Map(*SK-tree*, $\beta$)
  **foreach** $Seg \in T$ **do**
    $Seg.Tag \leftarrow$ *SK-tree*.Traverse($Seg.mbr$).Apply(Func: $LeafNodeID + (Seg.Tid \bmod \beta)$)
**Transformation** GroupBy ()
  **return** *Seg.Tag as a Key* `//` $Seg.Tag \equiv Pid$ `and the final result is formed as a` $PairRDD < Pid, Segments >$
**Transformation** MapPartitions ()
  $Intervals[\,] \leftarrow$ Divide *PartitionSegment*[ ] based on timestamps into intervals
  $STRtrees[\,] \leftarrow$ `BuildSTRtree(`$Intervals[\,]$`)` `// Each` *Interval* `has the corresponding STRtree's index`
  $IntervalBtree \leftarrow$ `BuildIntervalBtree(`$Intervals[\,]$`)`
  $LocalRMOI \leftarrow$ `BuildLocalRMOI(`$IntervalTree.Root, STRtrees[\,]$`)` `// It'll continue building` $STRtrees[\,]$ `into one STRtree augmented with the interval B-tree`
  **return** *LocalRMOI* `//` `The result is a` $PairRDD < Pid, LocalRMOI >$
**foreach** *LocalRMOI* **do** $gNodes[\,] \leftarrow LocalRMOI.Root$ `//` `contains` $MBR, Interval, Pid$
**if** $LP = 1$ **then**
  $GlobalRMOI \leftarrow$ `BuildGlobalRMOI(`$gNodes$`)` `// as R-tree`
**else if** $\beta < 8$ **then**
  $GlobalRMOI \leftarrow$ `BuildGlobalRMOI(`$gNodes, SK\text{-}tree$`)` `// as k-d-B-tree`
**else**
  $GlobalRMOI \leftarrow$ `BuildGlobalRMOI(`$gNodes, SK\text{-}tree, \beta$`)` `// as k-d-B-tree for upper half and R-tree for lower half based on` $\beta$
**return** *GlobalRMOI*
**Function** `CPModel` *(ComR, MemR, TOver)*:
  $PolyFeatures[\,] \leftarrow$ `PolynomialFeatures(`$Degree = 2, ComR, MemR, TOver$`)`
  $LP \leftarrow$ `Predict(`$CPModel.Coeffs, PolyFeatures$`)`
  **return** $LP$
**Function** `DPModel` *(ComR, MemR, TOver)*:
  $PolyFeatures[\,] \leftarrow$ `PolynomialFeatures(`$Degree = 3, ComR, MemR, TOver$`)`
  $QFT\{QT, Freg\} \leftarrow$ `getQueryFrequencyTable()`
  $LP_1[\,] \leftarrow LP_2[\,] \leftarrow \phi$
  **foreach** $QT \in QFT$ **do**
    $LP_1[QT] \leftarrow$ `Predict(`$DPModel.Coeffs_1, PolyFeatures, QT$`)`
    $LP_2[QT] \leftarrow$ `Predict(`$DPModel.Coeffs_2, PolyFeatures, QT$`)`
    $iFactor \leftarrow$ `Predict(`$IFModel.Coeffs, PolyFeatures, QT$`)`
    $QFT$.update($QT$, $QFT$.getFreq($QT$) $\times$ $iFactor$)
  $CumulativeQFT \leftarrow$ `CumulateFreqOfSameLPs(`$QFT, LP_1, LP_2$`)`
  $LP \leftarrow CumulativeQFT$.getMaxFreq( )
  **return** $LP$

As seen in Algorithm 1, the driver node starts by reading the given trajectory dataset $T$, on space $S$, into Spark RDD. Then, it generates $ST \subset T$ as a sample set, which can fit in the driver node's memory. For sampling, we use the Spark built-in function with replacement to capture the trajectories' characteristics. After that, RMOI computes the features ComR, MemR, and TOver based on Equations (1)–(3), respectively. The required parameters to conduct the computation of the feature are already given when initializing the cluster. These include executor number, cores number, executor memory size, and more. The time and space complexity to compute the features are dominated by the TOver, which is $O(s' + m')$, where $s' = |ST|$ and $m'$ is the number of $Traj \in ST$.

After computing the necessary parameters, the prediction stage is started. As seen in Line 3, RMOI needs to determine whether to go with the compulsory or with the discrete prediction. In the case of compulsory prediction, Line 28, RMOI transfers the features into a vector of polynomial features with degree 2. After that, RMOI carries out the computation to predict $LP$ as follows,

$$LP = \left\lfloor \sum_{i=1}^{n} Coeff_i \times P_i \right\rceil \ , \tag{5}$$

where $P$ is the set of the polynomial features, and $n$ is the vector size. However, in the case of discrete prediction (Line 32), RMOI needs the Query Frequency Table (QFT), which consists of Query Type (QT) and its Frequency (Freq). QFT could be dynamically collected from previous runs until it is needed, or the end-user could provide it. After transferring the features into a vector of polynomial features with degree 3, RMOI predicts the best locality pivot ($LP_1$) and the second best ($LP_2$) for every QT in QFT. While it loops over QTs, RMOI also predicts the query impact factor (iFactor). Then, it updates the Freq of the associated QT based on its own iFactor, as in Line 40. All the models conduct the prediction computations, per Equation (5), on the transferred features and the corresponding vectors of the stored regression coefficients. The prediction results are rounded, except for iFactor. Next, RMOI accumulates the Freqs of different QTs that have the same $LPs$. In cases where there is an identical Freq for different QTs, it returns the $LP_1$ that is equal to the $LP_2$ of the other QT. Otherwise, it returns the $LP_1$ of the highest Freq. Finally, all the returned $LPs$ from CPModel or DPModel are converted into $\alpha$ and $\beta$ based on Equation (4). The complexity of the prediction stage is constant, as the size of the polynomial vector and QFT are fixed.

After the prediction stage, RMOI enters the partitioning stage, which consists of two steps: space splitting and hashing. On the driver node, when $LP \neq 1$, RMOI builds a binary skeleton tree (SK-tree) on $ST$, shown in Line 8. The SK-tree is similar to k-d-B-tree [7] in the way it is constructed, but it is a lightweight tree which is only used to represent the required sub-regions (i.e., the required $\alpha$ spatial group). SK-tree only contains $\alpha$ leaf nodes. Each leaf node has a $LeafNodeID \in \{0, \beta, 2\beta, 3\beta, \ldots, tpn - \beta\}$. Next, the driver node broadcasts SK-tree to each worker on the cluster and launches a Map transformation to tag each $Seg \in T$, as shown in Line 9. Each worker traverses the SK-tree to tag each $Seg$ as follows.

$$Tag = LeafNodeID + (Seg.Tid \ \text{MOD} \ \beta) \ . \tag{6}$$

The $Seg.Tag$ is essential as it represents the RDD partition id ($Pid$). The first term in Equation (6), $LeafNodeID$, acts as an offset for $\alpha$ spatial groups where $Seg.Tid$ MOD $\beta$ represents the object-based partitioning for each spatial group. In case of having a segment that does not fit in an SK-tree's leaf, the segment is split into two segments and then reinserted again. At the end, RMOI uses GroupBy transformation on the $Seg.Tag$ to distribute $Segs$ on new RDD partitions such that $Seg.Tag = Pid$. Even though Algorithm 1 shows the general outlines of the partitioning stage, there is a special case when $LP = 1$. As mentioned before, this special case guarantees a full trajectory-preservation, which implies that the space-splitting is not needed. It does not require the construction of the SK-tree or the execution of the tagging procedure. Moreover, it combines the Map transformation and

GroupBy transformation, Lines 9 and 12, whereas a GroupBy transformation only returns the result of $Seg.Tid$ MOD $\beta$ as a key. There is no need for the *LeafNodeID* offset, as there is only one spatial group.

The first part of the partitioning stage is the construction of the SK-tree by the driver node. It has time complexity $O(\log x'(s' \log s'))$ and space complexity $O(x' + s')$, where $x'$ is the number of nodes in the SK-tree ($\approx 2\alpha + 1$). The time complexity of the second part is $O(s \log x')$, where $s = |T|$, as a result of the tagging and grouping procedures. However, they are carried out by the worker nodes which will parallelize the process over $ExecCoreNum \times ExecNum$ processors. Parallelism processing is straightforward since there are no data dependencies. The space complexity is linear even though Spark uses immutable data structure.

The next stage is building the local index (LocalRMOI). The data are already distributed over the cluster as RDD partitions. Each partition has a unique *Pid* and an ArrayList of *Segs*. RMOI launches a MapPartitions transformation (Line 14), which slices the segments into intervals based on the segments' timestamps for each partition. Then, it builds STRtree by using the Sort-Tile-Recursive algorithm [35] for each interval (slice) such that STRtrees only contain the ArrayList indices. Each interval has direct access to the corresponding STRtree. After that, it collects the STRtrees' roots and continues to build the top part. Finally, it builds an interval B-tree [36] on the intervals. The time complexity to build a LocalRMOI is $O(p \log p \log y)$, where $p = |\text{RDD partition}|$ and $y$ is the STRtree's node numbers. It is dominated by the STRtree construction complexity because $p > |intervals|$. The space complexity is linear. There are *tpn* LocalRMOIs that will be built by $ExecCoreNum \times ExecNum$ processors.

In the final stage (Line 20), the driver node collects the overall interval, MBR, and *Pid* from the roots of all the LocalRMOIs. After that, it checks whether the $LP = 1$ and proceeds to build the GlobalRMOI as an R-tree. Otherwise, it utilizes the SK-tree by reusing the tree hierarchy for the upper part. The lower part is built as R-tree, which contains the global nodes that do not exist in the SK-tree. If $\beta < 8$, then there is no need for the lower part. In general, the time complexity to build the GlobalRMOI is $O(tpn \log tpn \log x)$, where $x$ is the number of nodes of the GlobalRMOI. However, it reuses $x'$ nodes from SK-tree ($x' \leq x$). The space complexity is also linear.

## 4. Query Processing

In our previous work [33], we provided a detailed classification of space-based and trajectory-based query types. We extend that work by adding a time-based query type. Moreover, we improve the continuous queries by including a flag to indicate trajectory-in or trajectory-out conditions.

Our goal is to concentrate on space-based, time-based, and trajectory-based queries to reveal the performance level of the proposed approach in different scenarios. As a result, we focus on the following queries: Range Query, Interval Query, Continuous Range Query, Continuous Interval Query, Continuous Spatio-Temporal Query, Longest Trajectory, and Lookup Query.

### 4.1. Range Query

Given a range query $RQ =< P_{bl}, P_{ur} >$, where $P_{bl}$ is the bottom left point of the spatial range and $P_{ur}$ is the upper right point, on trajectory dataset $T$, RMOI needs to find any $Segs \in T$ such that $Seg_{space} \cap RQ_{space}$. It first determines the involved RDD partitions by traversing the GlobalRMOI based on the global nodes' MBRs. It returns the *Pids*, which are contained by the corresponding leaf nodes. After that, a Spark Job is initialized and targets only the required partitions. During the Job execution, engaged worker nodes traverse their own LocalRMOI. For range queries, LocalRMOI uses only the STRtree without the need for the interval B-tree as it only searches for spatial overlapping. The result is returned as a new RDD, which only contains the segments covered by *RQ*.

### 4.2. Interval Query

Given an interval query $IQ =< t_{start}, t_{end} >$ on $T$, RMOI needs to find any $Segs \in T$ s.t. $Seg_{time} \cap IQ_{time}$. As it does when handling a *RQ*, in this case RMOI specifies the needed *Pids* by traversing the GlobalRMOI based on the global nodes' intervals. Then, a Spark Job is initialized and only targets

the required partitions. The worker nodes only traverse the interval B-tree of the LocalRMOI. Finally, the result is formed as a new RDD containing the queried *Segs*.

In both *RQ* and *IQ*, RMOI depends on bulk loading when a node or interval is entirely covered by an *RQ* or *IQ*, respectively. In case of partially overlapping, RMOI applies a refinement process on leaf nodes or intervals to find intersected *Segs*.

### *4.3. Continuous Range Query*

Continuous Range Query (*CRQ*) consists of $k$ clauses where each clause has *RQ* and an indicator for a trajectory-in or trajectory-out. Therefore, when receiving a *k-CRQ* $= \{< RQ_1, flag_1 >, < RQ_2, flag_2 >, \ldots, < RQ_k, flag_k >\}$ on a trajectory set $T$, RMOI needs to find any $Traj \in T$ such that

$$\forall i \in [1, 2, \ldots, k],\ Traj_{space}\ \textcircled{s}\ RQ_i \ \text{ where } \ \textcircled{s} = \begin{cases} Traj_{space} \cap RQ_i: & (flag_i = True) \\ Traj_{space} \cap RQ_i = \phi: & (flag_i = False) \end{cases} . \quad (7)$$

Algorithm 2 outlines the required steps to process *k-CRQ*. First, RMOI traverses the GlobalRMOI based on the spatial property for each $RQ_i$, where $1 \leq i \leq k$. It determines the required *Pids* and returns two items: an overall set and an array of sets. The overall set contains all the *Pids* required for all *RQs* of *CRQ* and is used when initializing the Spark Job to filter undesired RDD partitions. The second item, denoted by ReqPids, is an array of sets, which contains all the required *Pids* as an individual set for each $RQ_i$. It is used to refine unnecessary LocalRMOI traversal, as shown in Line 8. At Line 5, RMOI identifies any trajectory intersects with any $RQ_i$ by using a transformation MapPartitions, which is running in parallel by the worker nodes on the given RDD partitions. Each engaged worker uses an array of hash sets for each RDD partition to collect the overlapped *Tids* (i.e., $Tids \cap RQ_i$ without considering the $flags$ values) and the corresponding clause's id ($ClauseID$). The Boolean value of the $flag$ does not matter at this point because RMOI needs to report all the overlapped *Tids* to check later on for the trajectory-out of other *RQs*' results. It uses a hash set to eliminate duplication among *Tids* of a particular $RQ_i$ and to speed up searching in the next step. In the case of partial trajectory-preservation, the results from different partitions are returned as lists of 2-tuple of *Tid* and $ClauseID$ and concatenated into a PairRDD< $Tid, ClauseID$ >. At this point, RMOI will finish the local reduction, where it is conducted on the RDD partitions level. After that, RMOI starts the global reduction by using a GroupBy transformation to reduce the PairRDD on *Tids*, Line 27. Finally, RMOI runs a Filter transformation to eliminate any *Tid* that does not fulfill the *CRQ*'s definition, as illustrated in Line 29. The purpose of the $ClauseID$ during filtration is to indicate that the associated *Tid* has been tested, and it intersected with $RQ_{ClauseID}$.

In the case of full trajectory-preservation, all the required computations are carried out during the first transformation, Line 5. As seen in Lines 10–23, RMOI picks a set of the resulted *Tids* where the associated $flag$ is true. Then, it iterates over the *Tids* of the picked set to eliminate any *Tid* that does not fulfill the *CRQ*'s definition. Finally, it returns the final result as an RDD of *Tids* without the need for a global reduction, i.e., GroupBy and Filter transformations.

**Algorithm 2:** RMOI: Processing *k-CRQ*.

**Input:** *k-CRQ*
**Output:** $RDD < Tid >$

$ReqPids[k]\{\} \leftarrow \phi$
**for** $i \leftarrow 1$ **to** $k$ **do**
  $ReqPids[i] \leftarrow GlobalRMOI$.spatialTraverse$(RQ_i.space)$
$OverallReqPids\{\} \leftarrow ReqPids$ // It is used in partitions filtration
**Transformation** MapPartitions$(k\text{-}CRQ, ReqPids)$
  $R[k]\{\} \leftarrow \phi$ // An array of sets to hold overlapped $Tids$
  **for** $i \leftarrow 1$ **to** $k$ **do**
    **if** $this.Pid \in ReqPids[i]$ **then** // $this.Pid$ is the current RDD partition's id
      $R[i]\{\} \leftarrow LocalRMOI$.spatialTraverse$(RQ_i.space)$
      /* It returns all $Tid$s intersect with $RQ_i$ space */
  **if** $LP = 1$ **then** // Full Trajectory-Preservation
    $L[\,] \leftarrow \phi$
    **for** $i \leftarrow 1$ **to** $k$ **do**
      **if** $flag_i = True$ **then**
        $Qid \leftarrow i$
        **Break**
    **foreach** $Tid \in R[Qid]\{\}$ **do**
      $Insert \leftarrow True$
      **for** $i \leftarrow 1$ **to** $k$ **do**
        **if** $R[i].contains(Tid) \neq flag_i$ **then**
          $Insert \leftarrow False$
          **Break**
      **if** $Insert = True$ **then** $L$.add$(Tid)$
    **return** $L$ *as list of* $< Tid >$// Formed as $RDD < Tid >$
  **return** $R$ *as list of 2-tuple* $< Tid, i >$// $i$ acts as *CRQ*'s clause id, and the final result is formed as $PairRDD < Tid, ClauseID >$
**if** $LP = 1$ **then**
  **return** $RDD < Tid >$// No need for global reduction
**Transformation** GroupBy$(PairRDD < Tid, ClauseID >)$
  **return** *Tid as Key*
/* The result is formed as a $PairRDD < Tid, ClauseIDs[\,] >$ where $ClauseIDs$ of each $Tid$ is grouped into a list */
**Transformation** Filter$(k, PairRDD < Tid, ClauseIDs[\,] >)$
  **for** $i \leftarrow 1$ **to** $k$ **do**
    **if** $ClauseIDs[\,].contains(i) \neq flag_i$ **then**
      **return** *False*
  **return** *True*
**return** $RDD < Tid >$

### *4.4. Continuous Interval Query*

Given a continuous interval query $k\text{-}CIQ = \{< IQ_1, flag_1 >, < IQ_2, flag_2 >, \ldots, < IQ_k, flag_k >\}$ on a trajectory set $T$, the result is any $Traj \in T$ such that

$$\forall i \in [1,2,\ldots,k],\ Traj_{time}\ \textcircled{t}\ IQ_i \ \text{ where } \ \textcircled{t} = \begin{cases} Traj_{time} \cap IQ_i: & (flag_i = True) \\ Traj_{time} \cap IQ_i = \phi: & (flag_i = False) \end{cases} . \quad (8)$$

The processing of *k-CIQ* is similar to what we have seen in *k-CRQ*. The only differences occur when traversing the GlobalRMOI and LocalRMOI. When traversing the GlobalRMOI for the ReqPids, it uses the global nodes' intervals. Furthermore, it only uses the interval B-tree when traversing the LocalRMOI. Otherwise, it follows the same processing steps from Algorithm 2. The optimization of the special case, full trajectory-preservation, holds true when processing *k-CIQ*. The reason is that each trajectory is held in one partition, thus there is no temporal existence of a trajectory in another partition. As a result, the local reduction is enough to fulfill the *k-CIQ* definition.

### *4.5. Continuous Spatio-Temporal Query*

A Continuous Spatio-Temporal Query (*CSTQ*) is a query that retrieves *Trajs* based on three dimensions: 2-D space and time. Therefore, when $k\text{-}CSTQ = \{< RQ_1, IQ_1, flag_1 >, < RQ_2, IQ_2, flag_2 >, \ldots, < RQ_k, IQ_k, flag_k >\}$ is given on a trajectory set $T$, the result must include any $Traj \in T$ such that

$$\forall i \in [1,2,\ldots,k],\ Traj_{space}\ \textcircled{s}\ RQ_i \wedge Traj_{time}\ \textcircled{t}\ IQ_i\ , \quad (9)$$

where the operators ⓢ and ⓣ are already defined in Equations (7) and (8), respectively.

As k-CSTQ follows the same processing steps as in Algorithm 2, we will only focus on the important and novel steps. RMOI starts collecting the ReqPids sets by traversing the GlobalRMOI on both spatial and temporal properties. It traverses down the tree if and only if the global node's MBR and interval intersect with the query clause's $RQ_i$ and $IQ_i$, consecutively. After that, it begins the local reduction on all the engaged RDD partitions by traversing all the corresponding LocalRMOI for each *CSTQ's* clause. This traversing of LocalRMOI based on *CSTQ's* clauses is conducted on the interval B-tree and the lower part of the STRtree. It starts from the interval B-tree such that, for any interval intersecting with the $IQ_i$, RMOI will traverse the corresponding sub-STRtree on $RQ_i$. The retrieved *Trajs* intersects with both $IQ_i$ and $RQ_i$.

### *4.6. Longest Trajectory Query*

The longest trajectory query (*LTQ*) is an aggregation query, which depends on the trajectory-length aggregation function. When receiving this query on T, it results in the maximal $Tid_{length}$. However, the data is in the form of *Segs* and distributed over the cluster based on the *LP*. The solution relies on aggregating the trajectory-length, which could be divided into local aggregation and global aggregation. We adopt the same algorithm implemented in our previous work [33]. The driver node starts a MapPartitions transformation, where each worker node executes a local aggregation on the trajectory-length for each partition. The result of the local aggregation is a list of 2-tuples of a *Tid* and its local length. In the case of full trajectory-preservation, the algorithm stops and returns the maximum $Tid_{length}$. Otherwise, it aggregates the resulted trajectory lengths on their *Tids* and returns the longest trajectory.

### *4.7. Lookup Query*

Given a *Tid*, RMOI needs to return all the *Segs* of the given *Tid*. As shown in Algorithm 3, the driver node starts by computing the required partitions (ReqPids). It loops over the spatial groups and reuses the offsetting from tagging (Equation (6)) on the given *Tid*. A trajectory cannot be in more

than one partition of each spatial group. After that, it uses a MapPartitions transformation, on the ReqPids, to retrieve the segments of the given *Tid*.

**Algorithm 3:** RMOI: Processing Lookup Query.

```
Input: Tid
Output: Traj
ReqPids[ ] ← ϕ
counter ← 0
while counter < tpn do
    ReqPids[ ] ← counter + (Tid MOD β)
    counter ← counter + β
Transformation MapPartitions()
    Traj[ ] ← ϕ
    foreach Seg ∈ partition.Segs do
        if Tid = Seg.Tid then
            Traj[ ] ← Seg
    return Traj as list of < Seg >
return Traj
```

### 4.8. Time Complexity for Query Processing

Most of the previous queries are processed in two steps. First, the driver node traverses the GlobalRMOI to find the ReqPids, if needed. The average case for the first step is $O(|\text{ReqPids}| \log x)$ and $O(|\text{ReqPids}|k \log x)$ for multistage queries. Lookup query needs $\lceil tpn/\beta \rceil$ steps to find the ReqPids. Second, the worker nodes process the selected RDD partitions and conduct a global reduction for the multistage queries. For simplicity, we will show the complexity of the average cases for the sequential processing of the second step. Then, we will analyze the parallelism factor. The complexity of the range query is $O(|ReqPids|R \log y)$, where $R$ is the number of results. The parallelism depends on the distributions of the ReqPids, where the worst case is to have the maximum ReqPids on one worker node. The interval query has similar complexity except that it traverses the interval B-tree. The complexity of processing *k-CRQ* is $O(|\text{ReqPids}|kR \log y + |\text{ReqPids}|k^2R)$. The first term is the complexity of the local reduction. During the local reduction, the data are being prepared for the global reduction by using a hash set on *Tid*, which offers, in the average case, constant time for the add and contain operations. The second term is the complexity of the global reduction which is dominated by the Filter transformation. Spark leverages a hash map technique to keep the GroupBy complexity linear in the average case. The parallelism factor depends on the distribution of ReqPids among the cluster, especially on the local reduction. When $LP = 1$, there is no need for a global reduction, which reduces the complexity into $O(|\text{ReqPids}|kR \log y)$. The other continuous queries have similar complexity, and the difference is in the LocalRMOI traversing. For the lookup query, the complexity of the average case is $O((tpn/\beta)p)$, and the parallelism factor depends on the number and the distribution of the required partitions to the number of the processors. The complexity of the average case of *LTQ* is $O(s + m)$, where the worker nodes need to scan $T$ and aggregate segments on *Tids* locally by using a hash map to keep the complexity linear. After that, they conduct a global aggregation on *Tids* and return the max length.

## 5. Experimental Study

In this section, we discuss the evaluation of our approach RMOI. The testbed is designed to evaluate RMOI's prediction models (CPModel and DPModel) on different features and datasets. We adopt DTR-tree (DTR) [27] and LocationSpark (LSpark) [25] in our evaluations. Both approaches

are designed for large-scale in-memory computation by using Spark. DTR is dedicated to moving object trajectories and only depends on a 1-dimension distribution. LSpark depends on a hierarchical tree for distribution, similar to RMOI. LSpark is generalized for spatial data, but it is considered as a Spark implementation for PRADASE [23], which is devoted to trajectory processing. We extend both approaches on the global and local indexes to support temporal queries.

### 5.1. Experiment Settings

The experiments were conducted on a six-node cluster using YARN as a resource manager and HDFS as a file system. The nodes were Dell OptiPlex 7040 desktop computers, with quad-core i7 (3.4 GHz), 32 GB of RAM, a 7200 rpm 1TB HD, and 8 M L3 cache, running CentOS 7. Our implementation used Apache Spark 2.2.0 with Java 1.8. We adopted Java ParallelOldGC as a garbage collector and Kryo for serialization. From Spark's perspective, the driver node used one node with 8 threads and 24 GB of memory. The worker nodes (executors) used 4 nodes with maximum settings of 32 threads and 64 GB of main memory.

From the data side, we used SF, UK + IE, and RioBuses, described in Table 1. The first dataset, SF, is a generated dataset over a real road network of the San Francisco Bay Area, CA. We used the network-based moving object generator [40]. SF's TOver is 0.223, which is the highest among our datasets. UK + IE [37] is only a spatial dataset, and it covers the UK, Ireland, the Irish Sea, and the English Channel. It has the lowest TOver (0.007) and contains mixed moving object classes similar to GT dataset. Furthermore, we reused RioBuses for the stress testing only. As we are interested about in-memory computation, the data were always cached to the main memory during experiments.

### 5.2. Construction of Indexes

Table 2 shows the average construction times for indexes on the SF, RioBuses, and UK + IE datasets. We only show the *LPs* that have been reported during the empirical study, as the other *LPs* are irrelevant. In other words, RMOI is unlikely to use the unreported *LPs* on the given datasets and resources. Furthermore, we exclude the construction time of the local index as all of the participant indexes have similar structures.

**Table 2.** The construction time (in seconds) for LSpark, DTR, and RMOI, where $LP_x$ means $LP = x$.

| Dataset | Resource Availability | LSpark | DTR | RMOI | |
|---|---|---|---|---|---|
| | | (Cons. Time) | (Cons. Time) | Cons. Time | Locality Pivot |
| SF | High | 67.08 | 49.05 | 53.29<br>52.33<br>47.09 | $LP_3$<br>$LP_2$<br>$LP_1$ |
| | Low | 100.17 | 78.09 | 74.60 | $LP_1$ |
| RioBuses | High | 68.78 | 40.51 | 53.11<br>35.85 | $LP_4$<br>$LP_1$ |
| | Low | 112.11 | 61.35 | 63.59<br>58.91 | $LP_2$<br>$LP_1$ |
| UK + IE | High | 60.92 | 62.61 | 68.30<br>60.92<br>56.91<br>55.96 | $LP_5$<br>$LP_3$<br>$LP_2$<br>$LP_1$ |
| | Low | 106.77 | 101.84 | 101.20<br>108.44 | $LP_6$<br>$LP_2$ |

Partitioning and distributing the trajectories dominate the construction time. DTR only depends on one dimension in distributing the data, while LSpark uses a hierarchical spatial tree. The partitioning phase in RMOI consists of space-based (space-splitting) and object-based (hashing) partitioning based

on the predicted $LP$. With $\alpha = 1$, RMOI only depends on object-based partitioning; when $\beta = 1$, it depends on spatial-splitting partitioning. On SF and RioBuses datasets, RMOI's construction times are decreased with increasing $\beta$, as space-splitting is more expensive. However, hashing partitioning is more expensive on UK + IE, because the dataset is sparse and contains more segments with respect to the number of trajectories, as shown in Table 1. Even though RMOI is slightly better than other indexes in some cases, there are no significant differences. However, given the overhead of both the features' computations and the prediction models, RMOI's construction complexity is at a competitive level.

### *5.3. Performance Evaluation*

We divided the testbed into two major stages to show the performance of RMOI and its prediction models in different scenarios. The first stage tested RMOI on all the query types in scenarios where it only depends on the compulsory model. In the second stage, RMOI used the discrete model on random queries limited by a predetermined frequency. Each stage was designed with two different cluster settings: high resource availability and low resource availability. On the SF and RioBuses datasets, each worker was set to 12 GB and 4 GB of memory with 6 threads and 2 threads. The UK + IE dataset has the same settings except for the memory, which is set to 6 GB instead of 4 GB.

Starting with the SF dataset, Figures 3 and 4 show the average running time for all the query types with high resource availability (Memory = 12 GB and Threads = 6, per worker). Figure 3a,b shows the average running time of 100 random range queries and 100 random interval queries on different selectivity levels. The running times of all methods increase with more selectivity. None of the algorithms show a significant speedup. However, RMOI shows a significant speedup on lookup queries as shown in Figure 3c, where it outperforms LSpark and DTR by a factor of 3.1× on the average. Figure 3d shows the running time of the longest query, where RMOI outperforms DTR by a factor of 1.9× and LSpark by a factor of 1.3×. Figure 3d also shows the number of sub-trajectories that need to be processed globally. It reveals how different partitioning techniques reflect on the local and global aggregations.

For continuous query, we ran 100 random queries with k = 2, 3, 4, 5, and 6 on two selectivity levels: Small Selectivity (SS) and Large Selectivity (LS), as shown in Figure 4. The spatial and temporal selectivities for SS were set to 0.3% and 1%, respectively, while both selectivities were set to 10% for LS. The spatial and the temporal selectivities were combined for spatio-temporal queries in SS or LS. In general, RMOI shows a significant speedup by a factor of 1.8× on the average.

Figures 5 and 6 show the average running time on low resource availability (4 GB of memory and 2 threads) on the SF dataset. Both RMOI and DTR have better performance than LSpark on RQ and IQ with large selectivity, as shown in Figure 5a,b. With the lookup query, RMOI shows a speedup by a factor of 4.2× on the average, as seen in Figure 5c. RMOI outperforms DTR and LSpark on the longest query by factors of 1.9× and 3.6×, respectively. With continuous queries, RMOI shows a speedup range from 1.5× to 7× the speed of competitors, Figure 6.

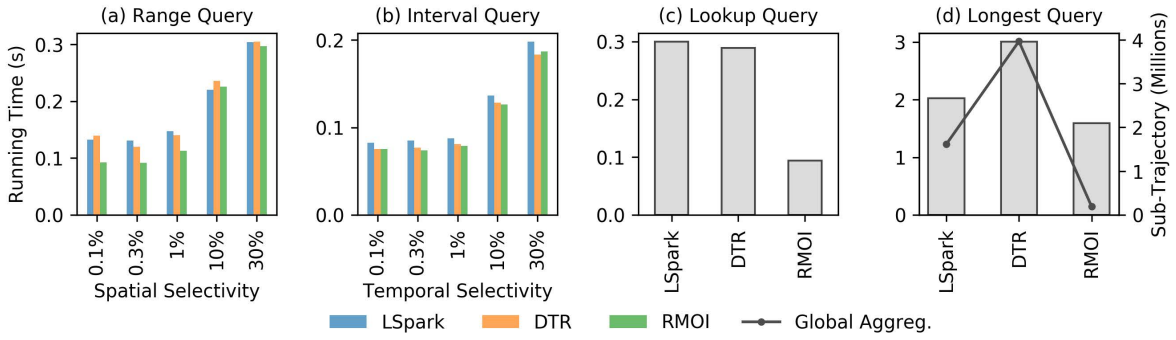


**Figure 3.** Average execution time for the SF dataset with high resource availability on (**a**) range query, (**b**) interval query, (**c**) lookup query, and (**d**) longest query.

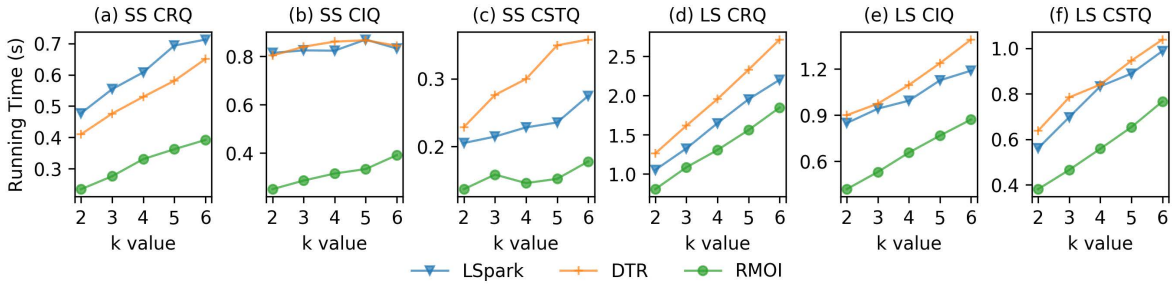


**Figure 4.** Average execution time for the SF dataset with high resource availability on (**a**) continuous range query with small selectivity, (**b**) continuous interval query with small selectivity, (**c**) continuous spatio-temporal query with small selectivity, (**d**) continuous range query with large selectivity, (**e**) continuous interval query with large selectivity, and (**f**) continuous spatio-temporal query with large selectivity.

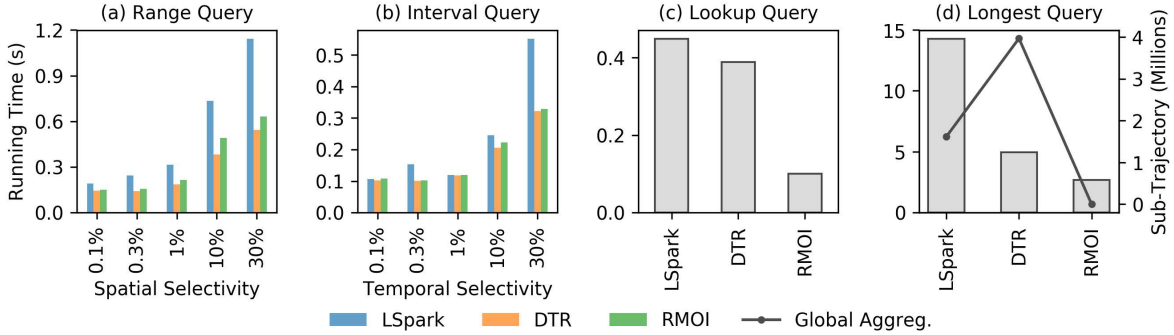


**Figure 5.** Average execution time for the SF dataset with low resource availability on (**a**) range query, (**b**) interval query, (**c**) lookup query, and (**d**) longest query.

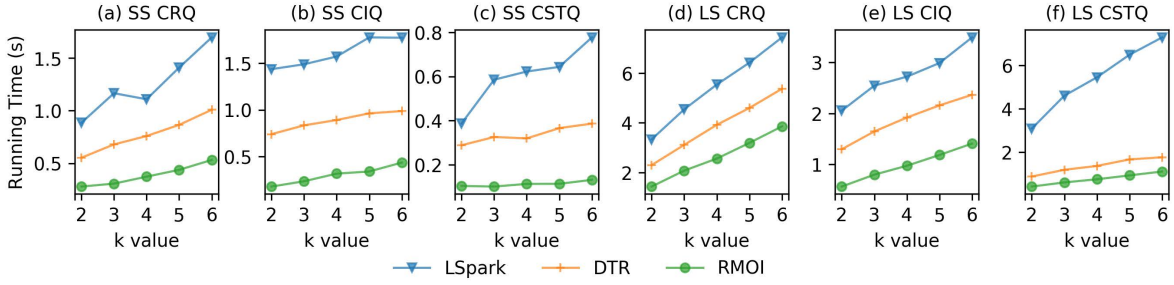


**Figure 6.** Average execution time for the SF dataset with low resource availability on (**a**) continuous range query with small selectivity, (**b**) continuous interval query with small selectivity, (**c**) continuous spatio-temporal query with small selectivity, (**d**) continuous range query with large selectivity, (**e**) continuous interval query with large selectivity, and (**f**) continuous spatio-temporal query with large selectivity.

In general, LSpark is more sensitive to resource availability, i.e., storage and computation resources. Its performance tends be better than DTR when there is high resource availability. However, DTR outperforms LSpark with low resource availability, especially on the longest query, even though the global aggregation is higher in DTR. RMOI predicts $LP$ to be 3 in the high resource case and 1 (full trajectory preservation) in the low resource case.

It is worth mentioning that RMOI used CPModel to predict a reasonable $LP$ that would be suitable for all the query types and not the absolute best choice for a particular query. Sometimes, it is impossible to find an $LP$ that would significantly improve the performance for all the queries because some queries prefer opposite directions. In such cases, CPModel considered the majority

without causing a significant performance decrease on the minority. This is the main reason why RMOI did not show a significant improvement on RQ or IQ, especially with DTR. However, it kept the performance of those queries at a competitive level and did not sacrifice their performance for the sake of other queries.

We excluded temporal and spatio-temporal queries in the UK + IE dataset since it does not include timestamps. Figures 7–9 show the averages of the running time in conditions of high resource availability (Memory = 12BG and Thread = 6) and low resource availability (Memory = 6BG and Thread = 2). Overall, RMOI shows better performance, especially on low resource availability. LSpark seems to slightly overtake DTR on most of the queries. In the case of a very low TOver dataset, the competition is challenging because most of the query types prefer spatial locality over object locality. As a result, RMOI predicts $LP$ to be 5 and 6 (the highest value is 7) for the high resource case and the low resource case, respectively. There is not enough room for improvement as all the approaches generally depend on space-based partitioning. We also observe that, when reducing resource availability, RMOI relies more on spatial locality on a dataset with low TOver, such as the UK + IE and the GT datasets. On the other hand, it relies more on object locality when applied to a dataset with higher TOver, such as the SF and RioBuses datasets.

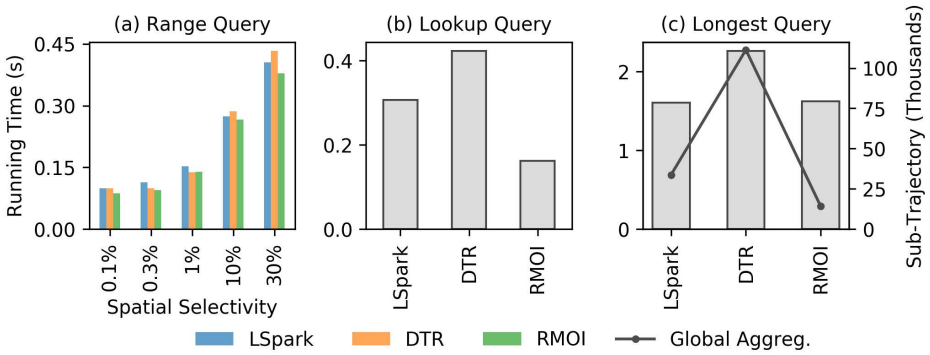


**Figure 7.** Average execution time for the UK + IE dataset with high resource availability on (**a**) range query, (**b**) lookup query, and (**c**) longest query.

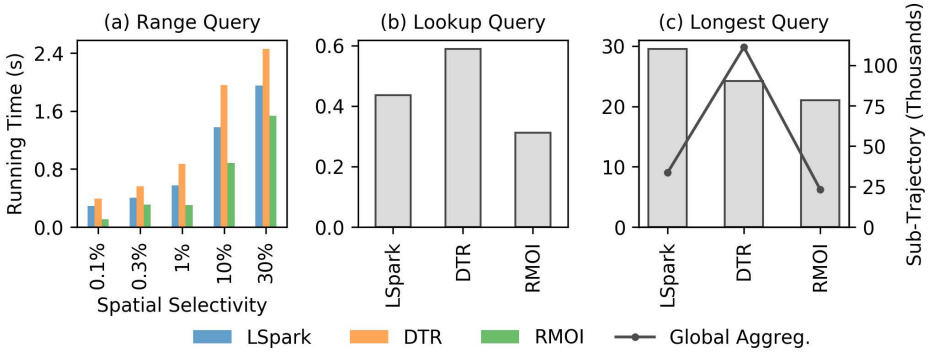


**Figure 8.** Average execution time for the UK + IE dataset with low resource availability on (**a**) range query, (**b**) lookup query, and (**c**) longest query.

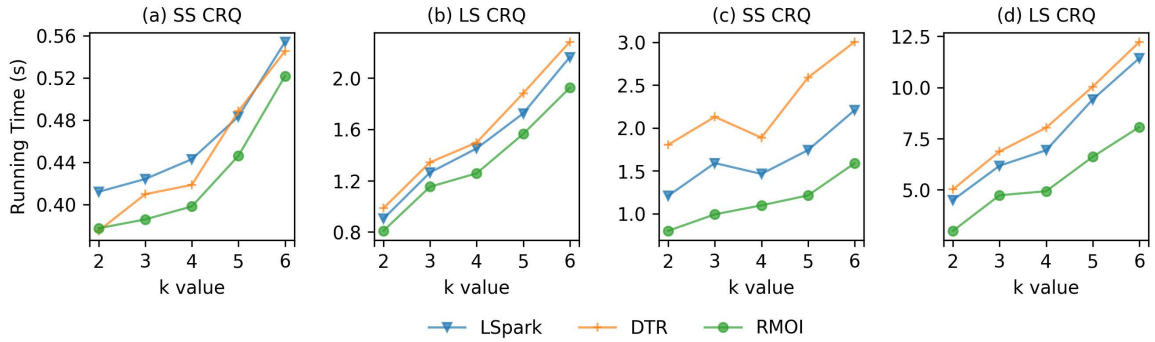


**Figure 9.** Average execution time of the continuous range query for the UK + IE dataset with (**a**) high resource availability and small selectivity, (**b**) high resource availability and large selectivity, (**c**) low resource availability and small selectivity, and (**d**) low resource availability and large selectivity.

The second testbed is a stress test designed to capture the situation of a long run and the impacts of different query types. It depends on reporting the results of 100-query cycle, which consists of random query types with random frequencies. Here, RMOI depends on DPModel and IFModel. Figures 10 and 11 show the cumulative running time of 6 query sets on the RioBuses dataset, where each color represents a query type. We identify continuous queries running on SS and LS as two query types, because of the huge difference in running time. The first and second query sets consist of 3 query types, as seen in Figures 10a,b and 11a,b. The third and the fourth sets have 7 query types (Figures 10c,d and 11c,d), and the last two sets contain all the query types. Each query type in every set has random frequency. Queries are drawn from query pools, which contain different spatial and temporal selectivities and k values similar to the first testbed. The cluster nodes have two settings as before. With high resource availability (memory = 12BG and thread = 6), RMOI shows a significant speedup. It outperforms LSpark and DTR by factors of 2.7× and 3.6× on the average, respectively. In a low resource availability scenario (memory = 4BG and thread = 2), RMOI outperforms LSpark and DTR again, by factors of 11.5× and 2.5× on the average, respectively, as seen in Figure 11. In both resource settings, aggregation queries and continuous range queries on large selectivity are the most dominant query types.

We also ran the second testbed on the UK + IE dataset. With high resource availability, RMOI gains a speedup by a factor 1.6x on the average, as seen in Figure 12; while on average it outruns LSpark by a factor 2.1× and DTR by a factor of 2.1× on low resource availability, Figure 13. However, it did not do as well on the second query set. RMOI predicts *LP* to be 2 on the second query set, while it predicts *LP* to be 6 and 7 on the other sets. The overall performance of RMOI is outstanding, especially on a very sparse dataset. Similar to the previous results, LTQ and LS CRQ are the most expensive query types, especially with low resource availability. The first query set has 35% LTQ, and the third set has 22% LTQ and 23% LS CRQ. These high percentages reflect on the cumulative running time, as seen in Figure 13a,c. One of the main reasons for running time surging is the GC's full-scan (Garbage Collector), which is triggered on heavy queries with low system resources. Moreover, it might cause a series of reactions that affect other performance factors, such as network, resource manager, cluster utilization, etc.

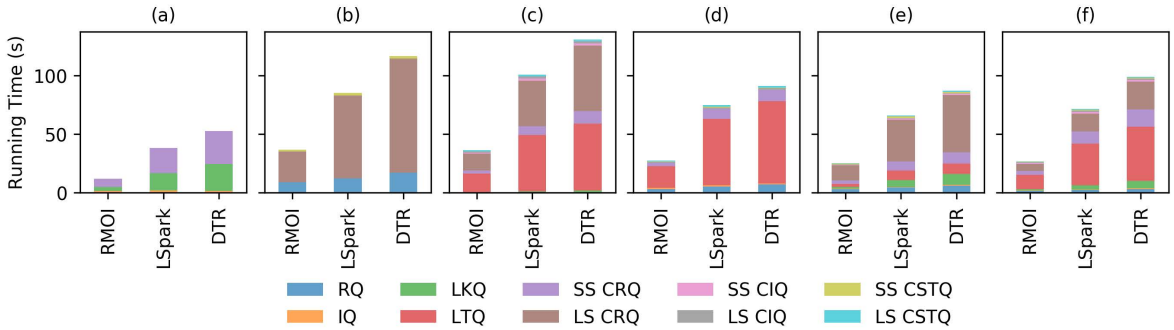


**Figure 10.** Stress testing for the RioBuses dataset with high resource availability on (**a**) 1st query set with 3 QTs, (**b**) 2nd query set with 3 QTs, (**c**) 3rd query set with 7 QTs, (**d**) 4th query set with 7 QTs, (**e**) 5th query set with 10 QTs, and (f) 6th query set with 10 QTs.

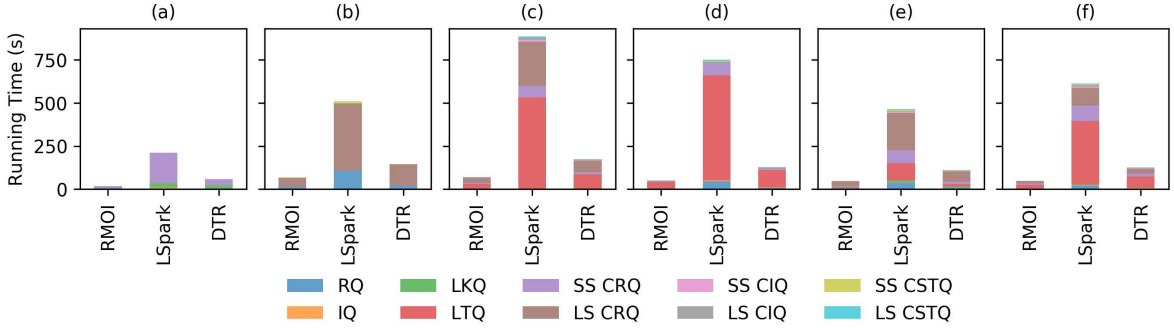


**Figure 11.** Stress testing for the RioBuses dataset with low resource availability on (**a**) 1st query set with 3 QTs, (**b**) 2nd query set with 3 QTs, (**c**) 3rd query set with 7 QTs, (**d**) 4th query set with 7 QTs, (**e**) 5th query set with 10 QTs, and (**f**) 6th query set with 10 QTs.

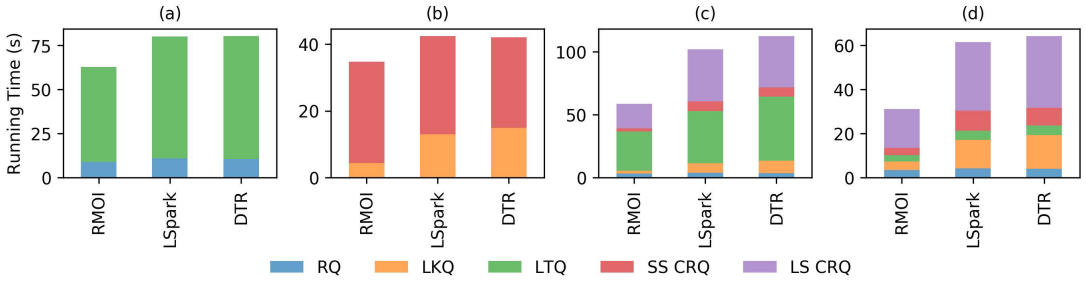


**Figure 12.** Stress testing on the UK + IE dataset with high resource availability on (**a**) 1st query set with 2 QTs, (**b**) 2nd query set with 2 QTs, (**c**) 3rd query set with 5 QTs, and (**d**) 4th query set with 5 QTs.

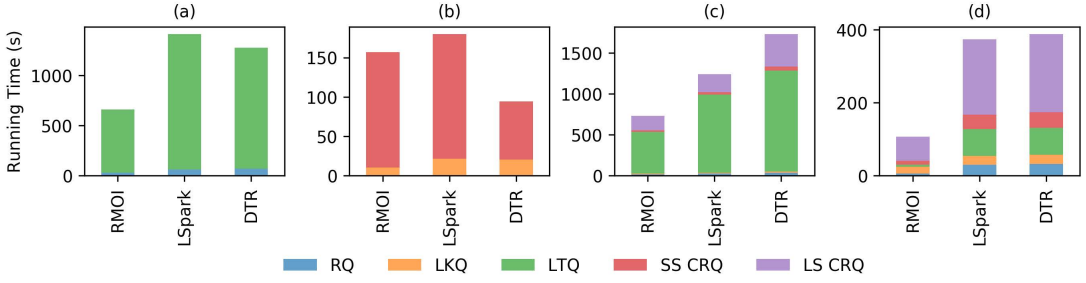


**Figure 13.** Stress testing on the UK + IE dataset with low resource availability on (**a**) 1st query set with 2 QTs, (**b**) 2nd query set with 2 QTs, (**c**) 3rd query set with 5 QTs, and (**d**) 4th query set with 5 QTs.

## 6. Conclusions

Our goal was to develop a resilient index that can satisfy a wide range of application demands and queries on the cloud, while overcoming the challenges raised by the configuration of cloud resources, the adopted distributed system, the nature of the trajectories, and the diversity in the queries. Cloud systems offer vast options for computing, storing, and communicating resources. A resilient index should adjust its structure to maximize the benefits of the available resources without the need for any fine-tuning by the end user. A resilient index also needs to reduce the impact of the distributed system's drawbacks and find a middle ground in contradictory situations in order to maximize overall performance.

In this work, we proposed RMOI as a resilient spatio-temporal index for historical moving object trajectories on top of Spark. RMOI uses two novel machine learning models to predict the locality pivot *LP*, which is used to balance the spatial and object localities. The adaptation capability comes as a result of controlling both localities. The prediction of *LP* depends on three adaptation factors: resources availability, nature of the trajectories, and query types, to maximize the overall performance.

The CPModel, which is used for a general prediction of *LP*, does not require any prior information about the upcoming queries. It is carefully designed not to be biased toward any query types, allowing the predicted *LP* to be convenient for any query. We also presented the DPModel, which is capable of measuring queries by using the IFModel and predicts *LP* based on the most dominant queries. IFModel is a prediction model that is responsible for determining the query impact factor (iFactor) via an appropriate query ranking. In order to reduce the overall computational requirement, the features are derived as proportional variables. The models depend on three features: ComR, MemR, and TOver. ComR and MemR are computed to satisfy the first adaptation factor, while TOver is for the second factor. The third adaptation factor is considered in IFModel. The models are designed to be as compact as possible in order to keep the RMOI's construction time on a competitive level. All the models use a polynomial regression on small degrees (degree = 2 for CPModel and degree = 3 for DPModel and IFModel). The complexity of the polynomial regression relies on the higher-degree features and the coefficients, where they scale polynomially on the number of the features and exponentially on the degree. Our work in reducing the number of features and the choice to use small degree polynomials results in models that have almost no overhead.

One of the main obstacles to fit these models is the absence of available training data. To generate such training sets would require the outputs of all the possible *LPs* on the most critical values of the features. As a result, we conducted thorough experiments on two real datasets, which produced 66K result factors. They covered all the intended query types on different environment settings. This allowed us to extract three nontrivial training sets for each model.

Finally, we conducted extensive experiments to validate our method and to test the RMOI models. The empirical study included various spatial-driven, temporal-driven, and object-driven queries. The results showed significant performance improvement when using RMOI on compact trajectory as well as sparse datasets. Moreover, we found that in realistic situations RMOI outperforms competitors on our stress testing. More importantly, our results strongly suggest that RMOI is very capable of adapting to different environments in both concise and long-run testing.

**Author Contributions:** Conceptualization, O.A. and T.A.; methodology, O.A.; software, O.A.; validation, O.A.; formal analysis, O.A. and T.A.; investigation, O.A. and T.A.; resources, O.A. and T.A.; data curation, O.A.; writing—original draft preparation, O.A.; writing—review and editing, O.A. and T.A.; visualization, O.A.; supervision, T.A.; project administration, T.A. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References


1. Abadi, D.; Agrawal, R.; Ailamaki, A.; Balazinska, M.; Bernstein, P.A.; Carey, M.J.; Chaudhuri, S.; Dean, J.; Doan, A.; Franklin, M.J.; et al. The Beckman Report on Database Research. *Commun. ACM* **2016**, *59*, 92–99, doi:10.1145/2845915.
2. Apache Spark. Available online: https://spark.apache.org/ (accessed on 31 August 2020).
3. Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.* **1984**, *14*, 47–57, doi:10.1145/971697.602266.
4. Beckmann, N.; Kriegel, H.P.; Schneider, R.; Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Rec.* **1990**, *19*, 322–331, doi:10.1145/93605.98741.
5. Sellis, T.K.; Roussopoulos, N.; Faloutsos, C. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Proceedings of the 13th International Conference on Very Large Data Bases, VLDB '87, Brighton, UK, 1–4 September 1987; pp. 507–518.
6. Bentley, J.L. Multidimensional Binary Search Trees in Database Applications. *IEEE Trans. Softw. Eng.* **1979**, *SE-5*, 333–340, doi:10.1109/TSE.1979.234200.
7. Robinson, J.T. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, SIGMOD '81, Ann Arbor, MI, USA, 29 April–1 May 1981; pp. 10–18, doi:10.1145/582318.582321.
8. Finkel, R.A.; Bentley, J.L. Quad trees: A data structure for retrieval on composite keys. *Acta Inform.* **1974**, *4*, 1–9, doi:10.1007/BF00288933.
9. Pfoser, D.; Jensen, C.S.; Theodoridis, Y. Novel Approaches to the Indexing of Moving Object Trajectories. In Proceedings of the 26th International Conference on Very Large Data Bases, VLDB 2000, Cairo, Egypt, 10–14 September 2000; pp. 395–406.
10. Nascimento, M.A.; Silva, J.R.O. Towards Historical R-trees. In Proceedings of the 1998 ACM Symposium on Applied Computing, SAC '98, Atlanta, GA, USA, 27 February–1 March 1998; pp. 235–240, doi:10.1145/330560.330692.
11. Zhang, T.; Yang, L.; Shen, D.; Fan, Y. An Efficient In-Memory R-Tree Construction Scheme for Spatio-Temporal Data Stream. In *Proceedings of the Workshops on Service-Oriented Computing (ICSOC)*; Springer International Publishing: Cham, Switzerland, 2019; pp. 253–265.
12. Zheng, K.; Shang, S.; Yuan, N.J.; Yang, Y. Towards efficient search for activity trajectories. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, QLD, Australia, 8–12 April 2013; pp. 230–241, doi:10.1109/ICDE.2013.6544828.
13. Luo, W.; Tan, H.; Chen, L.; Ni, L.M. Finding Time Period-Based Most Frequent Path in Big Trajectory Data. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13), New York, NY, USA, 22–27 June 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 713–724, doi:10.1145/2463676.2465287.
14. Ding, Z.; Yang, B.; Chi, Y.; Guo, L. Enabling Smart Transportation Systems: A Parallel Spatio-Temporal Database Approach. *IEEE Trans. Comput.* **2016**, *65*, 1377–1391, doi:10.1109/TC.2015.2479596.
15. Šidlauskas, D.; Ross, K.A.; Jensen, C.S.; Šaltenis, S. Thread-Level Parallel Indexing of Update Intensive Moving-Object Workloads. In *Proceedings of the Advances in Spatial and Temporal Databases*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 186–204.
16. Šidlauskas, D.; Šaltenis, S.; Jensen, C.S. Parallel Main-Memory Indexing for Moving-Object Query and Update Workloads. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12), Scottsdale, AZ, USA, 20–25 May 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 37–48, doi:10.1145/2213836.2213842.
17. Eldawy, A.; Mokbel, M.F. SpatialHadoop: A MapReduce framework for spatial data. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 1352–1363, doi:10.1109/ICDE.2015.7113382.
18. Eldawy, A.; Li, Y.; Mokbel, M.F.; Janardan, R. CG_Hadoop: Computational Geometry in MapReduce. In Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13), Orlando, FL, USA, 5–8 November 2013; pp. 294–303, doi:10.1145/2525314.2525349.

19. Aji, A.; Wang, F.; Vo, H.; Lee, R.; Liu, Q.; Zhang, X.; Saltz, J. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *Proc. VLDB Endow.* **2013**, *6*, 1009–1020, doi:10.14778/2536222.2536227.
20. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Liu, H.; Wyckoff, P.; Murthy, R. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.* **2009**, *2*, 1626–1629, doi:10.14778/1687553.1687609.
21. Akdogan, A.; Demiryurek, U.; Banaei-Kashani, F.; Shahabi, C. Voronoi-Based Geospatial Query Processing with MapReduce. In Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Indianapolis, IN, USA, 30 November–3 December 2010; pp. 9–16, doi:10.1109/CloudCom.2010.92.
22. Alarabi, L.; Mokbel, M.F.; Musleh, M. St-hadoop: A mapreduce framework for spatio-temporal data. *GeoInformatica* **2018**, *22*, 785–813, doi:10.1007/s10707-018-0325-6.
23. Ma, Q.; Yang, B.; Qian, W.; Zhou, A. Query Processing of Massive Trajectory Data Based on Mapreduce. In Proceedings of the First International Workshop on Cloud Data Management (CloudDB'09), Hong Kong, China, 6 November 2009; pp. 9–16, doi:10.1145/1651263.1651266.
24. Yu, J.; Wu, J.; Sarwat, M. 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, Seattle, WA, USA, 3–6 November 2015; pp. 70:1–70:4, doi:10.1145/2820783.2820860.
25. Tang, M.; Yu, Y.; Malluhi, Q.M.; Ouzzani, M.; Aref, W.G. LocationSpark: A Distributed In-memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.* **2016**, *9*, 1565–1568, doi:10.14778/3007263.3007310.
26. You, S.; Zhang, J.; Gruenwald, L. Large-scale spatial join query processing in Cloud. In Proceedings of the 31st IEEE International Conference on Data Engineering Workshops (ICDEW), Seoul, Korea, 13–17 April 2015; pp. 34–41, doi:10.1109/ICDEW.2015.7129541.
27. Wang, H.; Belhassena, A. Parallel trajectory search based on distributed index. *Inf. Sci.* **2017**, *388-389*, 62–83, doi:10.1016/j.ins.2017.01.016.
28. Peixoto, D.A.; Hung, N.Q.V. Scalable and Fast Top-k Most Similar Trajectories Search Using MapReduce In-Memory. In *Proceedings of the Databases Theory and Applications*; Springer International Publishing: Cham, Switzerland, 2016; pp. 228–241.
29. Wang, H.; Zheng, K.; Xu, J.; Zheng, B.; Zhou, X.; Sadiq, S. SharkDB: An In-Memory Column-Oriented Trajectory Storage. In Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM '14), Shanghai, China, 3–7 November 2014; pp. 1409–1418, doi:10.1145/2661829.2661878.
30. Li, R.; He, H.; Wang, R.; Ruan, S.; Sui, Y.; Bao, J.; Zheng, Y. TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data. In Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 20–24 April 2020; pp. 2002–2005, doi:10.1109/ICDE48307.2020.00224.
31. GeoMesa. Available online: https://www.geomesa.org/ (accessed on 31 August 2020).
32. Shang, Z.; Li, G.; Bao, Z. DITA: Distributed In-Memory Trajectory Analytics. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18), Houston, TX, USA, 10–15 June 2018; ACM: New York, NY, USA, 2018; pp. 725–740, doi:10.1145/3183713.3183743.
33. Alqahtani, O.; Altman, T. A Universal Large-Scale Trajectory Indexing for Cloud-Based Moving Object Applications. In Proceedings of the Eleventh International Conference on Advanced Geographic Information Systems, Applications, and Services (Geoprocessing) (IARIA), Athens, Greece, 24–28 February 2019; pp. 42–51.
34. Eldawy, A.; Mokbel, M.F. The Era of Big Spatial Data: A Survey. *Found. Trends Databases* **2016**, *6*, 163–273, doi:10.1561/1900000054.
35. Leutenegger, S.T.; Lopez, M.A.; Edgington, J. STR: A simple and efficient algorithm for R-tree packing. In Proceedings of the 13th International Conference on Data Engineering, Birmingham, UK, 7–11 April 1997; pp. 497–506, doi:10.1109/ICDE.1997.582015.
36. Ang, C.H.; Tan, K.P. The interval B-tree. *Inf. Process. Lett.* **1995**, *53*, 85–89, doi:10.1016/0020-0190(94)00176-Y.
37. OpenStreetMap Contributors. Planet GPX. Available online: https://planet.openstreetmap.org/gps/ (accessed on 31 August 2020).
38. Dias, D.; Costa, L.H.M.K. CRAWDAD Dataset Coppe-ufrj/RioBuses (v. 2018-03-19). 2018. Available online: https://crawdad.org/coppe-ufrj/RioBuses/20180319 (accessed on 31 August 2020). doi:10.15783/C7B64B.

39. OpenStreetMap Contributors. Available online: https://www.openstreetmap.org (accessed on 31 August 2020).
40. Brinkhoff, T. A Framework for Generating Network-Based Moving Objects. *GeoInformatica* **2002**, *6*, 153–180, doi:10.1023/A:1015231126594.