International Journal of Geo-Information

MDPI

Article

Real-Time Spatial Queries for Moving Objects Using Storm Topology

Feng Zhang [1,2], Ye Zheng [1], Dengping Xu [3], Zhenhong Du [2,*], Yingzhi Wang [4,*], Renyi Liu [2] and Xinyue Ye [5,*]

1 School of Earth Sciences, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China; zfcarnation@zju.edu.cn (F.Z.); zy274508496@126.com (Y.Z.)
2 Zhejiang Provincial Key Laboratory of Geographic Information Science, 148 Tianmushan Road, Hangzhou 310028, China; liurenyi@163.com
3 Academy of Forest Inventory and Planning, State Forestry Administration, Beijing 100714, China; bj119xdp@163.com
4 Department of Public Order, Zhejiang Police College, 555 Binwen Road, Hangzhou 310053, China
5 Department of Geography, Kent State University, Kent, OH 44240, USA
* Correspondence: duzhenhong@zju.edu.cn (Z.D.); wangyingzhi@zjjcxy.cn (Y.W.); xye5@kent.edu (X.Y.); Tel.: +86-571-8827-3287 (Z.D.); +86-138-1919-8735 (Y.W.); +1-419-494-7825 (X.Y.)

Academic Editor: Wolfgang Kainz
Received: 27 July 2016; Accepted: 23 September 2016; Published: 29 September 2016

Abstract: With the rapid development of mobile data acquisition technology, the volume of available spatial data is growing at an increasingly fast pace. The real-time processing of big spatial data has become a research frontier in the field of Geographic Information Systems (GIS). To cope with these highly dynamic data, we aim to reduce the time complexity of data updating by modifying the traditional spatial index. However, existing algorithms and data structures are based on single work nodes, which are incapable of handling the required high numbers and update rates of moving objects. In this paper, we present a distributed spatial index based on Apache Storm, an open-source distributed real-time computation system. Using this approach, we compare the range and K-nearest neighbor (KNN) query efficiency of four spatial indexes on a single dataset and introduce a method of performing spatial joins between two moving datasets. In particular, we build a secondary distributed index for spatial join queries based on the grid-partition index. Finally, a series of experiments are presented to explore the factors that affect the performance of the distributed index and to demonstrate the feasibility of the proposed distributed index based on Storm. As a real-world application, this approach has been integrated into an information system that provides real-time traffic decision support.

Keywords: real time; spatial query; moving objects; Apache Storm

1. Introduction

Advanced technologies for sensing and computing have resulted in the creation of massive datasets consisting of trajectories of people and vehicles. The development of mobile phones and other handheld GPS devices as well as the widespread use of location-based services (LBS) have contributed to the acquisition of spatial information related to moving objects. Understanding and analyzing the large-scale and complex data that reflect moving objects is crucial for enhancing both quality of life and built environments. As a significant branch of the field of Geographic Information Systems (GIS), real-time GIS technology is devoted to the study of the collection, integration, management and analysis of spatial data streams in real-time contexts, such as emergency response and disaster monitoring.

Scientists in this domain need to store, manage, query and visualize large volumes of dynamic flow data to perform exploratory and analytical tasks. However, they are facing enormous challenges

because of a lack of sufficient computing power to support big-data-driven studies. A platform is needed that integrates scalable trajectory databases with intuitive and interactive visualization and high-end computing resources [1]. The absence of such tools that explicitly support interactive visual analytics is impeding progress in fully utilizing the trajectory data that are becoming available. To accomplish this goal, a fast and robust spatial query method is needed. An appropriate spatial index for moving objects could be used to reduce the time consumed for spatial queries. However, increasing accuracy requirements and the growing amounts of moving object data are driving a corresponding increase in the frequency of spatial data updates, thereby posing considerable challenges for the building of spatial indexes and the real-time execution of spatial queries.

To address this issue, cloud technologies are emerging as a new solution. For example, Hadoop MapReduce can be used to improve the efficiency of polygon overlay analysis in spatial big data (SBD) applications by building a grid spatial index [2]. S. You et al. proposed an approach using a distributed computing framework based on memory to address the problem of spatial join operations, because the intermediate calculation results of Hadoop must be stored on hard disk, which reduces analytical efficiency because of the excessive disk I/O cost [3]. Although these cloud technologies offer the benefits of high-performance computing for GIS applications based on static SBD, a novel approach to cloud computing is needed to cope with highly dynamic spatial data.

The major difference between static data and dynamic data is that the former are stored in a distributed file system in the form of data blocks, whereas the latter are streamed from an external sensor device with the help of distributed message-oriented middleware (DMOM), tuple by tuple. Compared with static SBD, for which the relevant problems are predominantly focused on the amount of data, the problems that must be solved to handle dynamic data are predominantly related to the rapid update of the data. Rapidly changing data are known as fast data [4].

To lay the foundations for effective visual analytics of trajectory datasets, this manuscript presents a method of building a distributed spatial index on an open-source cloud computing framework, namely, Apache Storm, which has been playing an important role in traffic statistics [5] and network anomaly detection [6]. Based on this technology, we conduct real-time spatial querying of spatial fast data (SFD), including range querying, KNN querying, and spatial join querying. In particular, we build a secondary index based on the grid-partition index for spatial join queries. According to the experimental results, we find that using a quadtree structure for the secondary index yields the best performance in spatial computing. The proposed approach offers the capability of large-scale data management and support for various types of queries by leveraging Cloud computing platforms. This research will help the research community to conduct mobility-related studies in a more efficient and productive way.

In summary, the main contributions of this paper are as follows:

1. We propose a distributed index based on a Storm topology for moving objects and then implement range queries and continuous KNN (CKNN) queries using this distributed index. In this way, the pressure of updating data streams in stand-alone nodes can be relieved, and real-time updating for moving objects can be realized.
2. We design a spatial join algorithm using Storm for moving object streams. Experiments show that a quadtree-based index outperforms other types as the secondary index for distributed querying and updating.

The remainder of the paper is organized as follows. Section 2 reviews the technological background and related work, including the Storm topology programming paradigm and spatial indexes for moving objects. In Section 3, we describe the problem setting, and the semantics of various types of spatial queries, in accordance with the characteristics of spatial datasets, are formally presented. Section 4 proposes two types of spatial query approaches based on Apache Storm. Section 5 reports several sets of experiment conducted to explore the possible factors that affect the distributed spatial index. Section 6 describes the conclusion of the study and potential applications of this research.

2. Related Work and Background

2.1. Related Work

2.1.1. Spatial Index for Moving Objects

Many previous studies have suggested that modifications to the traditional spatial index can help to cope with the storage of changing mobile spatial data [7,8]. According to the research content and spatial data type, the proposed indexes can be categorized into two main types [9]: 1. Spatial indexes for storing trajectories of moving objects. The temporal and spatial information of the objects' motion can be expressed by extending the traditional spatial data structure into the time dimension [10–12]. These indexes are built on the hypothesis that all data have been stored in advance. As a result, spatial indexes that belong to this category, such as MV3R-tree, STR-Tree, and TB-Tree, are still optimized for spatial queries; 2. Spatial indexes for storing current or near-future positions of moving objects. Some studies have focused on reducing the number of updates by describing the positions of moving objects in terms of linear functions and only updating the processes in the spatio-temporal databases when the actual location of a point diverges from the value indicated by the function by a certain error threshold [13–15]. Other studies have considered points based on periodic reports of an object's latest position at individual timestamps from peripheral equipment [9,16,17]. These studies represent efforts to improve the efficiency of data updating to adapt to the frequent changes in dynamic data.

In addition, multi-core processors, graphics processing units (GPUs) and other parallel technologies have been adopted to improve the spatial index updating and query efficiencies for computations on increasing amounts of data [18–20].

2.1.2. U-Grid and P-Grid

U-Grid (Uniform Grid) is one type of update-efficient spatial index that works in a single thread of the main memory, especially for moving objects. As shown in Figure 1, U-Grid divides the space into uniform grids, and each grid cell within the array stores a pointer to a linked list of buckets that contain the object data [21]. Moreover, it also stores a secondary index, which points directly to entities in the grid, in a table. When the data are updated locally, the index is refreshed in a bottom-up fashion [22], which improves the update efficiency. P-Grid (Parallel Grid), a method of parallel spatial computing on multiple threads and concurrency control, has also been proposed to improve performance [18,23].

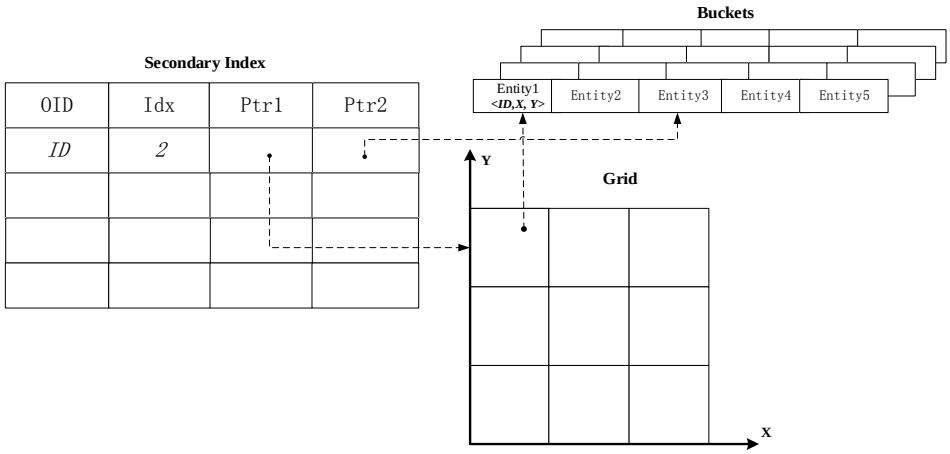


Figure 1. U-Grid index.

U-Grid and P-Grid enable the implementation of the data structure and algorithm for the mobile data spatial index in the main memory and the performance of spatial querying and updating by means of multithreading on multi-core processors, respectively. However, a single node is unable

to meet the requirements of these approaches when faced with a rising number of spatial data and an increasing update frequency due to higher accuracy requirements.

2.1.3. Spatial Querying on a Distributed Platform

Recently, many distributed platforms have been applied in GIS applications to improve spatial querying on massive spatial data [24]. Hadoop is the most commonly used distributed platform for handling spatial queries. In [25,26], the authors implemented spatial join queries and KNN join queries using MapReduce. Afsin Akdogan et al. improved the efficiency of parallel queries by building a distributed Voronoi diagram as a flat spatial index [27]. VegaGiStore implements an "indexing + MapReduce" data processing architecture to provide efficient spatial query processing over SBD and numerous concurrent user queries [28]. In addition, many spatial processing platforms based on Hadoop, such as Hadoop-GIS [29] and SpatialHadoop [30], have been proposed. Hadoop-GIS is optimized for spatial partitioning, partition-based parallel processing over MapReduce using the Real-time Spatial Query Engine (RESQUE) and multi-level spatial indexing. SpatialHadoop employs a simple high-level spatial language, a two-level spatial index structure, basic spatial components constructed inside the MapReduce layer, and three basic spatial operations: range queries, KNN queries, and spatial joins. These platforms have proven to be powerful GIS tools for spatial querying.

Beyond the above, with the development of efficient distributed memory computing platforms, research has begun to focus on increasing the efficiency of spatial join queries with the help of Apache Spark. SpatialSpark [3] implements partition-based spatial joins using Spark, which is more universal and efficient than the earlier approach. GeoSpark [31], which consists of three main layers—an Apache Spark Layer, a Spatial RDD Layer and a Spatial Query Processing Layer—extends the regular Apache Spark RDD to support geometrical and spatial objects with data partitioning and indexing. In addition, SparkGIS [32] and Simba [33] also support high-performance spatial querying on SBD with the help of Apache Spark. However, the spatial data that are handled by the current GIS distributed platforms are offline static data stored on hard disk or HDFS. Because of the rapidly changing nature of online mobile data, update messages may be sent from a terminal at any time; consequently, we cannot store these data in such media in advance. Consequently, the traditional distributed batch processing framework is not applicable, and a novel cloud computing approach should be developed.

2.2. Technological Background

2.2.1. Distributed Streaming Processing Framework

Since 2010, when the open-source distributed stream processing platform S4 was released by Yahoo!, many platforms that are similar but possess different characteristics have been proposed. Among them, Apache Storm, Yahoo S4, SparkStreaming and Samza are representative examples. They each have their own characteristics, and the main differences between them are summarized in Table 1.

Apache Storm was chosen over other distributed streaming platforms for use in our work for the following reasons: (1) Spark Streaming addresses stream data in a micro-batch-based manner. It handles real-time streaming data in micro-batches created via time slicing and transforms stream processing into the batch processing of time slices. In contrast, Storm handles streaming data tuple by tuple, which is more suitable for the frequent updating of mobile data; (2) In Samza, each step in a job workflow is an independent entity, and those entities are connected using Kafka. In Storm, all steps are connected by an internal system, resulting in much lower latency and satisfying the requirements of real-time querying for moving objects; (3) Compared with S4, Storm offers higher reliability and a more flexible routing method, which make it easier to implement a distributed spatial index.

Table 1. Comparison of distributed streaming processing frameworks.

Item	Storm	S4	Spark Streaming	Samza
Time	2011	2010	2012	2013
Grain	Single Record	Single Record	Micro-batching	Single Record
Data Model	Tuple	Event	Object	Object
Routing	User-defined	By Key	/	Depends on Kafka
State Management	Not Built In	Stateful	Dedicated DStream	Stateful Operations
Latency	Very Low	Low	Medium	Low
Throughput	Low	Low	High	High
Maturity	High	Low	High	Medium
Intermediate Result Storage	Local Memory	Local Memory	Local Memory or File System	Kafka
Guarantees	At Least Once	None	Exactly Once	At Least Once

In summary, real-time spatial querying for moving objects requires a high-speed event processing platform that allows for incremental computations. Consequently, Storm is most suitable for our purposes because of its characteristics of low latency, high reliability and maturity.

2.2.2. Storm Topology Programming Paradigm

A topology is a workflow that describes an entire job in Storm. A Storm topology is a directed topology graph that includes a number of components, including spouts and bolts [34]. A spout is a unit that produces a data stream, and a bolt is a unit for processing a data stream. Each tuple generated by a spout flows to a bolt in a particular way. Each bolt also sends a tuple to the next bolt in a specifically defined grouping until the tuple is stored in the persistent memory.

Apache Storm provides multiple ways to group data streams. Three main types of grouping are considered in our discussion: ShuffleGrouping, FieldsGrouping and AllGrouping. When streams are grouped via ShuffleGrouping, each tuple is randomly emitted to a bolt instance. FieldGrouping is another type of stream grouping in which tuples are emitted to bolt instances based on a specific field. Streams that have the same value in this field are guaranteed to flow to the same bolt instance. The last stream grouping method on which we focus is AllGrouping. In this method, all tuples are copied to each bolt instance and are then processed in parallel.

All processes have independent resources that are distributed to each node in a Storm topology. Each bolt instance processes only one tuple at a time, unless the bolt instance itself creates threads. For this reason, a Storm program is thread-safe under normal circumstances. Notably, a Storm transaction topology can ensure the order of the submitted computations. However, a tuple will be sent again if its processing fails. To simplify the problem, this paper does not consider such scenarios.

3. Problem Setting and Semantic Information

3.1. Problem Setting

Based on the characteristics of the vast majority of application scenarios, we regard moving objects as discrete points in a two-dimensional $|X| * |Y|$ space. $|X|$ ($|Y|$) represents the number of different positions in the horizontal (vertical) dimension. The spatial index stores points, each of which contains an ID and a quantity P_{store} (X, Y), which represents the spatial position information of the corresponding moving object. When the distance between the stored position and the actual position is greater than $\Delta Tolerance$ (Dist (P_{store} (X_1, Y_1), P_{actua1} (X_2, Y_2)) > $\Delta Tolerance$, where $\Delta Tolerance$ is the tolerance distance), the point will send an update message (*OID*, *Xold*, *Yold*, *Xnew*, *Ynew*), which represents the movement of the point from location (*Xold*, *Yold*) to location (*Xnew*, *Ynew*), to the central server [35]. Under this premise, if no new update message has been delivered, we regard the index running in the server as the most recently updated data. All spatial queries we discuss in this article are based on this index. The spatial information of each moving object can be represented by

an 8-byte tuple [36]. Therefore, less than 8 GB of storage is required for a spatial index for 100 m moving objects, that is, a normal PC can be used. However, the update speed of these dynamic spatial data may reach 1 m/s or even higher.

3.2. Semantic Information

Spatial queries can be classified into two categories. One consists of spatial queries on a single dataset. Range queries and KNN queries, for example, are included in this category. The other category consists of spatial queries between two datasets, which ignore the spatial relationships within either individual dataset, such as spatial joins.

Definition 1. Range Query: *Suppose that there is a given set of moving points represented by O and that F(O) represents the spatial properties of O. The process of obtaining O′ as a subset of O in the spatial attribute range (F(O′).min>F(Range).min∩F(O′).max<F(Range).max) is defined as a range query* [19,37,38].

Definition 2. KNN Query: *Suppose that there is a given set of moving points represented by O and that a query point q is specified. A KNN query refers to the process of obtaining a subset of O that satisfies the following conditions: 1.* $|O'|=K$. *2. For any point* P_{out} *outside of O′ and any point* P_{in} *inside O′, the distance between* P_{out} *and q is larger than the distance between* P_{in} *and q* ($\forall o \in (O\text{-}O'), F_{dist}(q,o > Max\{F_{dist}(q,o') \mid o' \in O'\}$) [17].

Definition 3. Range Spatial Join Query: *Suppose that there are two given sets of moving points, L and R. A spatial join query is a spatial query for a pair of points* $P_L \in L$ *and point* $P_R \in R$ *that satisfy the inequality* $F_{dist}(P_L,P_R)<d$, *where* $F_{dist}(P_L,P_R)$ *refers to the Euclidean distance between* P_L *and* P_R *and d is a constant known as the join distance (JD). If the join query result is also required to satisfy* $P_L, P_R \in$ *Range, then the query is called a range spatial join query* [39–41].

Definition 4. Continuous Query: *Unlike static spatial queries, continuous queries* [42,43] *can be classified into three types—moving queries on stationary objects, stationary queries on moving objects and moving queries on moving objects* [44]. *In our research, we regard the result of a continuous query as a continuous series of snapshots of a set of results, which is calculated based on the constantly changing spatial positions of the moving objects of interest. That is,* $q' = (I^{t1}(a_1), I^{t2}(a_2), \ldots, I^{tn}(a_n))$, *where* $I^{ti}(a_i)$ *represents the snapshot of the result set obtained based on condition* a_i *at time* t_i.

For range queries and range spatial join queries as defined above, a rectangular range is the most commonly used type of query condition. The corresponding update message is thus a tuple in the form (X_{min}, Y_{min}, X_{max}, Y_{max}).

4. Structure and Algorithms for Spatial Data in a Storm Topology

4.1. Storm Topology Algorithm for a Single Dataset

The general concept of the establishment of a Storm topology for a single dataset has three steps: 1. The spatial data are grouped according to the ID field via FieldsGrouping; 2. Distributed indexes are established in each bolt instance, and local results are obtained by means of the main memory spatial query algorithm; 3. After the local results are summarized, the global outcome for the dataset is integrated.

As shown in Figure 2, there are two spout components, the query spout and the update spout, where data streams are generated in the topology. The Update Spout connects to the external MOM and receives real-time transmissions of update message from external mobile devices. In this way, spatial information is sent to the Storm topology in the form of a tuple stream for data operations. The Query Spout converts continuous queries into tuples entering the topology in the form of a stream and performs a distributed spatial query. In Figure 2, the transmission of data from the Update Spout is represented by dashed lines because the update messages are grouped into clusters via FieldsGrouping.

For all update messages corresponding to spatial points for a specific ID, there exists a unique bolt instance to accept them. In contrast, the transmission of data from the Query Spout is represented by solid lines because the query messages are copied to every Moving Objects Algorithm Bolt (MOAB) instance via AllGrouping. Initially, the MOABs are a batch of executors with the same data structure that are randomly assigned to each work node. The data structure in each bolt represents a partition of the spatial index separated by point ID and stores the corresponding local point values. These bolts are the core of the entire topology because all updating and querying operations are executed in these bolts in parallel. When it enters the topology, a tuple message is assessed to determine whether it is an update message or a query message. Then, depending on the type of message, a query or update operation is performed. A MOAB instance is an implementation of the algorithm for moving objects in the main memory and requires considerable resources for its intensive calculations. A Result Aggregation Bolt then obtains the global solution through aggregation and calculation of the result set from each partition. It obtains the tuples from the MOABs via FieldsGrouping based on the QueryID field. A range query then requires only adding the results from each relevant partition. The iSEE algorithm is implemented for continuous KNN (CKNN) queries [17]. Initially, TopK is calculated for each partition. When a MOAB receives an update message, according to the obtained tuple and the local critical distance, it is determined whether expansion or contraction operations are needed. If a new local critical distance is generated, further comparisons with the global critical distance are performed at the Result Aggregation Bolt.

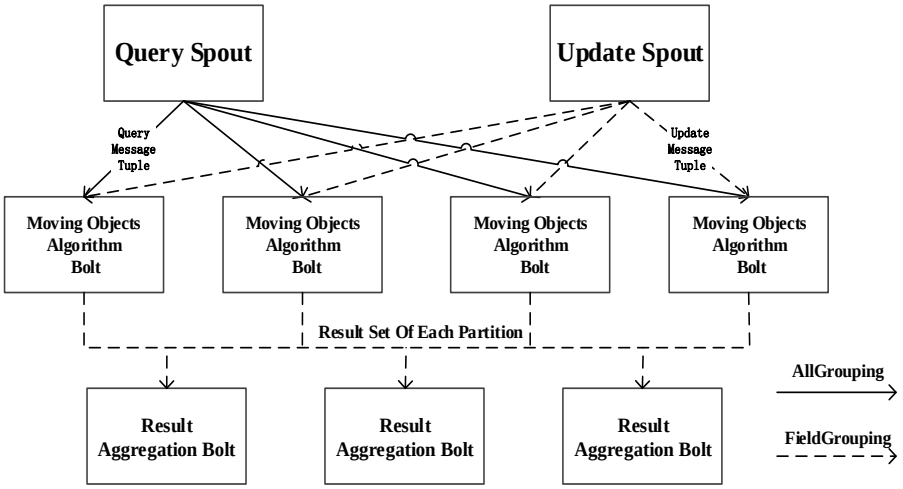


Figure 2. Storm topology for a single dataset.

From the above topology, we can see that for spatial queries within a single dataset, the data structure is distributed from a single executor to multiple executors on one or more work nodes. Update messages for spatial information are allocated to these executors based on point IDs, thereby alleviating the pressure of frequent updates and real-time computations and analyses of a large number of moving objects. For spatial analyses of two or more datasets, however, stream groups based on point ID cannot guarantee that spatially correlated points will flow into the same MOAB. Therefore, we usually establish spatial index partitions as the basis of stream grouping.

4.2. Spatial Joins for Moving Objects in a Storm Topology

In this section, we describe how to build a distributed index in a Storm topology for range spatial join queries on moving objects. As mentioned above, the data cannot be grouped by point ID in this case. If no index is available, pairwise computations should be performed for points from the two datasets. For improved computational efficiency, a grid index is built for data partitioning. The Storm topology for spatial joins is depicted in Figure 3.

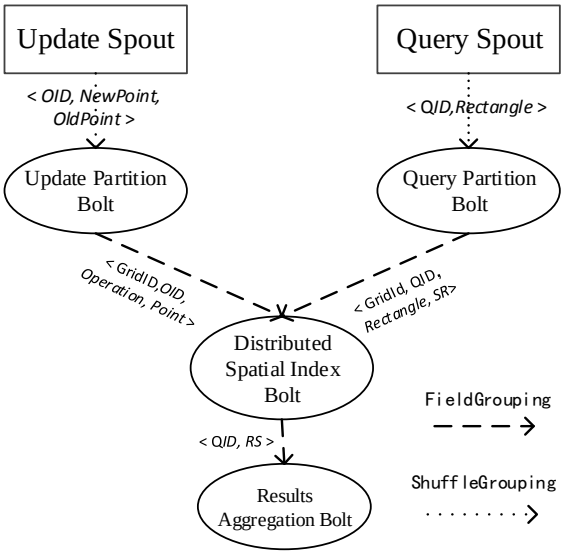


Figure 3. Storm topology for spatial joins.

4.2.1. Data Structure

For moving objects, the frequency of data updates is usually much greater than the query frequency. Using a tree structure to organize the data will increase the time complexity of the updating process. In some special cases, the entire tree structure will change, from bottom to top. Consequently, a uniform grid index or hierarchical grid index is designed for spatial partitioning [21].

Similar to U-Grid, the proposed distributed grid spatial index consists of two parts. The grid-partition index is used as the primary index, and all points in each cell are further stored in different bolt instances, as indicated by a secondary index.

Because different instances of the same bolt cannot communicate with each other, each cell stores not only all of the points within that cell but also the points that are related to that cell. As shown in Figure 4, the shortest distance between a point P and a cell C is denoted by *DistMin (P, C)*. When the point is contained in the grid cell, *DistMin (P, C)* = 0. It is tempting to conclude that *DistMin (P, C)* is the vertical distance from the cell edge or the linear distance from one of the cell's vertexes. When *DistMin (P, C)* < *JD*, the cell and the point are deemed to be spatially correlated. All cells that are spatially correlated with a given point are defined as the affected cells (ACs) of that point. The circle with a radius of *JD* and point *P* as its center is defined as the critical circle (CC) of *P*. Because a point located near a cell vertex may have multiple ACs, replicas of these points are stored in multiple different cells.

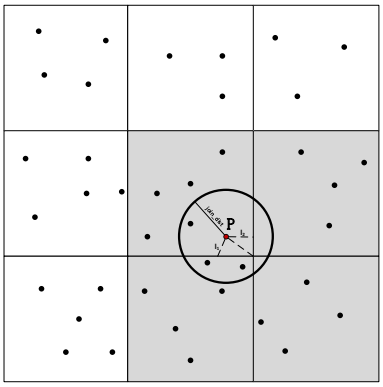


Figure 4. Grid cell partitioning.

As shown in Figure 5, the data structure in each grid cell consists of two sets of secondary indexes and a result table. The secondary indexes are used to individually organize the spatial data from the two datasets. The result set (RS) is used to store the spatial join results. If the secondary index is an R-tree, then the entire data structure is referred to as D-Rtree (Distributed R-Tree); similarly, secondary indexes based on hash table and quadtree structures correspond to D-HashTable and D-QuadTree. If RS is a two-dimensional array, $A_{[Lm][Rn]}$ represents two pointers to two entities, one from each of the two secondary indexes. A null value in this two-dimensional array indicates that the two points corresponding to the indexes do not conform to a spatial join relationship. By taking advantage of the two-dimensional array, we can navigate the point entities with a time complexity of O(1). However, in practice, considering that only a small portion of all pairs of points from the two datasets in the same cell will satisfy a spatial join relationship, the high incidence of null values in the two-dimensional array is expected to result in storage waste. Therefore, an element table is proposed to replace the two-dimensional array when there are few spatial join tuple pairs. For a given pair, both their OIDs and the pointers to them are stored in a row of the element table. The time complexity of querying a specific pair by their OIDs is O(*M***N*), where M and N are the numbers of spatially joined points from each of the two layers. When M and N are not excessively large, this data structure can effectively decrease the required storage space while maintaining an acceptable query efficiency.

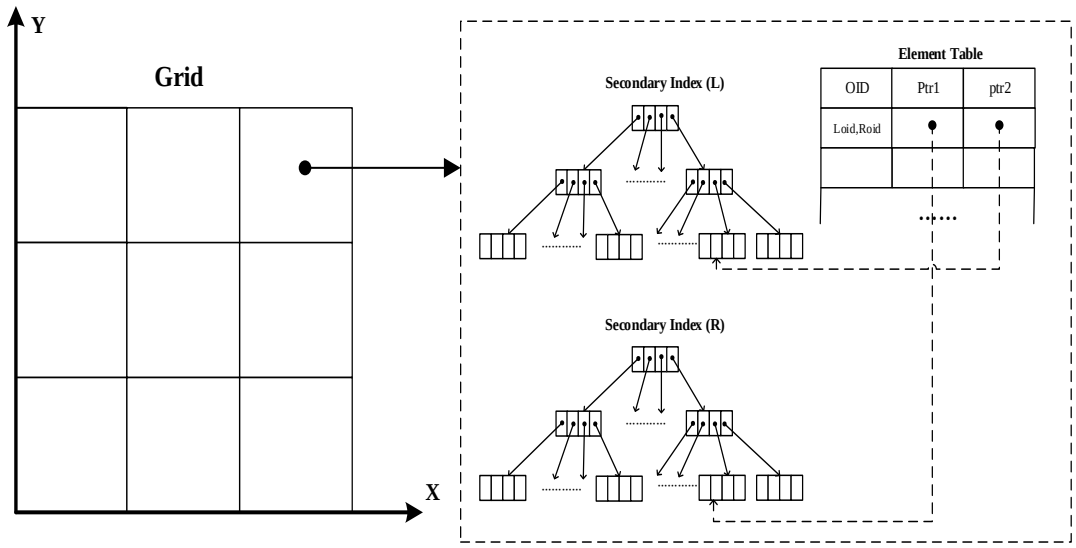


Figure 5. Distributed index.

4.2.2. Update Partition Bolt

The process of data updating using the proposed distributed index is performed by two bolts—the Update Partition Bolt and the Distributed Spatial Index Bolt. The Update Spout sends data update messages to the Update Partition Bolt as tuples of the form (*Id*, *oldX*, *oldY*, *newX*, *newY*), which contain the information on the coordinates before and after the update. Similar to the local and non-local update operations in U-Grid, the proposed distributed spatial index has the equivalent operations of equivalent and discrepant AC updates. When the affected cells of a point are exactly equal before and after the point update, the update is an equivalent AC update; otherwise, the update is a discrepant AC update. The Update Partition Bolt judges the update type based on the incoming tuple. If it is an equivalent AC update, the Update Partition Bolt needs only to emit tuples contain the calculated cell ID, Update Operation and point entity to the Spatial Index Bolt via FieldsGrouping based on the CellId field. If it is a discrepant AC update, then the Spatial Index Bolt must compare the different ACs before and after the update.

Algorithm 1: UpdatePartBolt (Tuple UpdateMessage)

```
1.  OldCC←GetOldCC(JoinDistance, UpdateMessage.OldPoint)
2.  NewCC←GetOldCC(JoinDistance, UpdateMessage.NewPoint)
3.  OldACList←GetOldACList(Grid, oldCC)
4.  NewACList←GetNewACList(Grid, newCC)
5.  foreach Cell in OldACList
6.    if Cell In NewACList then
7.      emit(CellId, UpdateOperation, Point)
8.    else
9.      emit(CellId, DeleteOperation, Point)
10.   end if-then
11. end-foreach
12. foreach Cell in NewACList then
13.   if not in OldACList then
14.     emit(CellID, DeleteOperation,Point)
15.   end if-then
16. end-foreach
```

4.2.3. Query Partition Bolt

The algorithm for the Query Partition Bolt is relatively simple. First, the extent of the query rectangle is obtained, and the relevant cells (intersected or contained by the query rectangle) are computed. Then, the Query Partition Bolt emits the CellIDs and the corresponding spatial relationships (SRs) with the query rectangle to the Distributed Spatial Index Bolt via FieldsGrouping based on the CellId field.

Algorithm 2: QueryPartBolt (Tuple QueryMessage)

```
1.  AffectQueryCellList←CalculateByRange(QueryMessage.Range)
2.  foreach Cell in AffectQueryCellList
3.    if cell PartialCovered by Range then
4.      emit (CellID, SR:Intersect)
5.    end-if
6.    if Cell TotalCovered by Range then
7.      emit(CellID, SR:Contain)
8.    end-if
9.  end-foreach
```

4.2.4. Distributed Spatial Index Bolt

The Distributed Spatial Index Bolt is the core of this topology for spatial join queries. It operates based on the concept of global shared execution [45,46], and the streams generated by the Query Partition Bolt and the Update Partition Bolt are aggregated in this bolt. The data structures, including the various cells with their corresponding secondary indexes and spatial join result tables, are stored in an element table with a hash index in the bolt instances. All of the above data structures run in memory; consequently, the data query and update efficiencies are relatively high because no I/O operations are involved.

A distributed index data update actually consists of three processes [44]: 1. Perform add, update or delete operations on the secondary indexes where the point to be updated is stored; 2. Perform query operations on the other secondary indexes; 3. Update the spatial join results. From the above description, we can see that both the update efficiency and the query efficiency

should be considered with regard to the secondary indexes. As demonstrated by the experimental comparison presented in Section 5, when a quadtree is used as the secondary index, it outperforms secondary indexes based on the hash table and R-tree structures. For a given query tuple, we must determine the relevant cells according to the tuple identifier. If a cell is contained by the query rectangle, then all pairs in RS are emitted as tuples to the next bolt. If a cell intersects with the query rectangle, then further judgment is needed to determine whether a pair is emitted.

Algorithm 3: SpatialIndexBolt(Tuple Message)

```
1.  Cell←FindCellByID(Message.CellId)
2.  if Message from QueryPartBolt then
3.    if Message.SR = contain then
4.      foreach Pair in Cell.RS
5.        emit Pair
6.      end-foreach
7.    end if-then
8.    if Message.SR = Intersect then
9.      foreach Pair in Cell.RS
10.       if Pair.Point1∈Message.Range&Pair.Point2∈Message.Range then
11.         emit Pair
12.       end if-then
13.     end-foreach
14. if Message from UpdatePartBolt then
15.   foreach SpatialIndex In cell.SI
16.     if Message.Point∈SpatialIndex then
17.       SpatialIndex.Operate(Message.Point, Message.Operate)
18.     else
19.       PairList←SpatialIndex(Message.Point, JoinDistance)
20.       update Cell.RS by PairList
21.     end if-then
22.   end-foreach
23. end if-then
```

4.2.5. Results Aggregation Bolt

In the Results Aggregation Bolt, the final global results are aggregated from the results of each distributed index query. Because multiple copies are stored of point entities located near grid vertexes, duplicate tuple pairs must be removed in this bolt.

5. Experimental Study

In this section, we present experiments using the topology described above and analyze its performance results in terms of data updating and querying. In Section 5.1, we describe the software and hardware environment, the workload and other preparations involved in the experiments. Of the experiments presented Section 5.2, Study 1 is an experiment conducted to investigate the design of the distributed index for a single dataset. U-Grid, U-R-tree, R-tree and Grid were chosen, and this experiment was performed to compare their efficiency in spatial querying and updating with different numbers of Storm worker processes. Studies 2–4 are range spatial join query experiments in which D-Rtree, D-HashTable and D-QuadTree were chosen as the distributed index.

5.1. Experimental Setting and Workloads

All experiments were performed using seven servers as work nodes, each of which had a 6-Core Intel Xeon(R) 2.6 GHz CPU with 24 hardware threads and 8 GB of RAM. All seven nodes used a 64-bit Linux operating system (Suse Enterprise Server SP2), with Apache Kafka, an open-source distributed publish/subscribe messaging system, serving as the DMOM. Three of the nodes also had Apache ZooKeeper installed to coordinate the distributed systems.

The workloads were produced by a modified version of an open-source network-based moving object trace generator based on Brinkhoff's algorithm [47]. The network data were obtained from Jiangsu Province, China (see Figures 6 and 7). In the single-dataset experiment, 1 million dynamic points were sent to Kafka at an update rate of 0.7 m/s, and the Storm spout pulled tuples from Kafka into the Storm topology. The four indexes mentioned above were compared with regard to their performance for continuous range queries, CKNN queries and update rates. In the spatial join experiments, we compared the spatial update ratios for different numbers of grid cells and different numbers of moving objects. For the range spatial join queries, instead of using pure query tuples, we mixed update tuples and query tuples in different proportions as inputs to the Storm topology. For these workloads, searches with various scopes were performed. Table 2 shows the parameters of the workloads for the single-dataset spatial queries. The workloads for the range spatial join queries included two sets of moving objects produced using the same parameters presented in Table 3. The preparation time is not included in the results for any experiment.

Figure 6. Road network in Jiangsu Province.

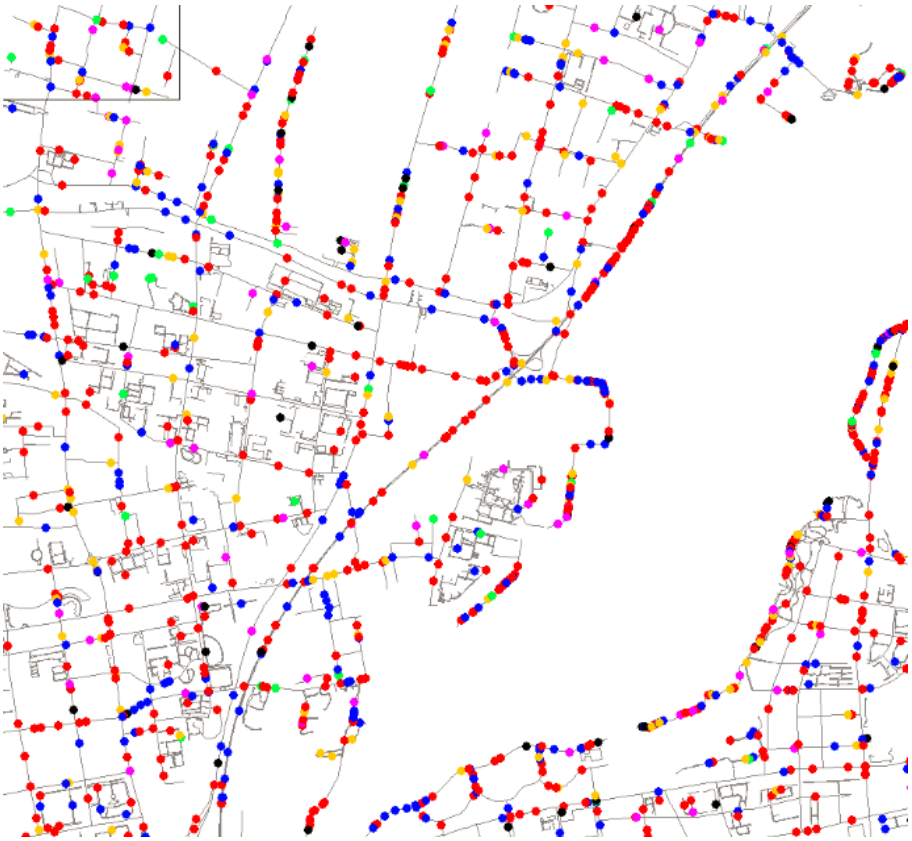


Figure 7. Moving objects in the road network.

Table 2. Workload configuration for single-dataset spatial queries.

Parameter	Value
Number of objects, 10^6	1
Update ratio, M/S	0.7
Query selectivity	0.5%
K in nearest neighbor analysis	100
Nodes	232237895
Monitored region (km^2)	Jiangsu Province, 511 × 494

Table 3. Workload configuration for range spatial join queries.

Parameter	Value
Number of objects, 10^6	0.25, 0.5, 0.75, 1
Query/update ratio	5%, 10%, 15%, 20%
Query selectivity	0.01%, 0.04%, 0.16%, 0.64%
Monitored region (km^2)	Jiangsu Province, 511 × 494
Nodes	232237895
Update ratio, M/S	0.7

5.2. Experimental Results

- Study 1: Influence of the Storm bolt parallelism on spatial queries for a single dataset

In Figure 8a, the update efficiencies of the indexes are plotted versus the number of parallel bolts. Figure 8b,c depict the efficiencies of range queries and CKNN queries, respectively. From Figure 8a, we can conclude that all four indexes exhibit a trend of an increasing update rate with an increasing

degree of parallelism. Once the index has saturated, however, the update efficiency no longer increases and even starts to decrease. This is because the number of different update processes entering different work nodes will increase as the parallelism increases and the transmission of information between work nodes requires considerable time. Figure 8b,c suggest that increasing the parallelism has a relatively small effect on spatial query performance. Increasing the parallelism will reduce the number of moving objects in each Distributed Spatial Index Bolt executor while giving rise to more inter-node communications. Consequently, whether increasing the degree of parallelism has a positive or negative impact depends on which factor dominates.

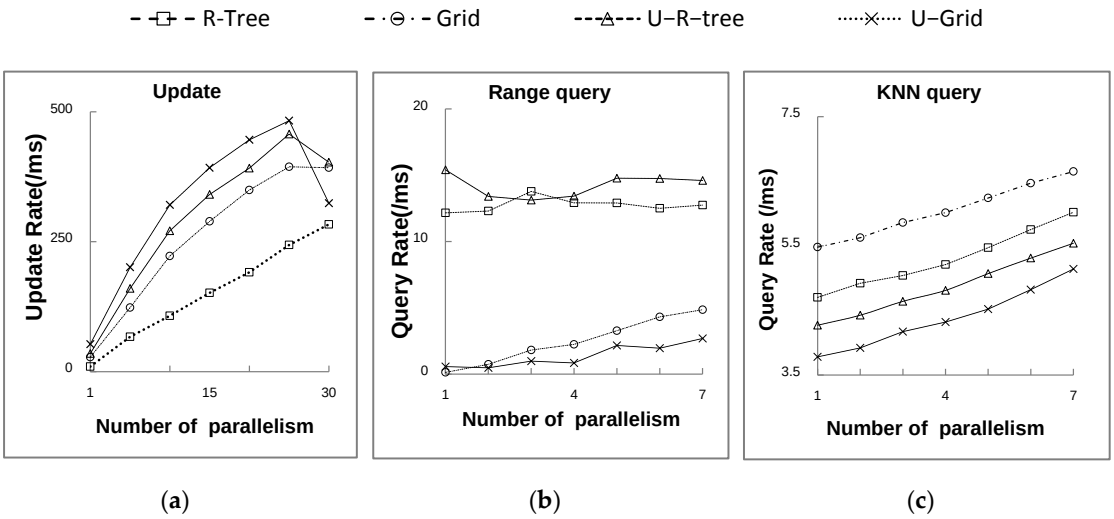


Figure 8. Single-dataset performance with different degrees of parallelism. (**a**) Update rate of spatial index; (**b**) query rate of range query; (**c**) query rate of KNN query.

The following three experiments demonstrate the potential factors affecting the spatial join performance. Studies 2 and 3 investigate the update performance of the proposed distributed spatial index for different numbers of grid cells over different periods of time. Study 4 addresses how the percentage of query tuples in the workload influences the total throughput of spatial join queries. To ensure reliability, the workloads imported into Kafka for the different experiments were exactly the same. The rate of workload input was 0.7 m/s, which was sufficiently fast to ensure that the data read rate would not become a bottleneck.

- Study 2: Influence of the number of moving objects on the spatial join performance

From Figure 9a–c, we can conclude that the update efficiency of D-Quadtree is significantly higher than those of the other indexes. In general, the update rates of the three indexes increase over time, initially rapidly and then more gradually. Because the secondary index does not initially store any data, the query step of the spatial join updating process described in Section 4.2 initially consumes less time. However, as an increasing amount of data becomes stored in the secondary index, the update efficiency decreases. The number of moving objects has a small effect on D-Quadtree index and a greater effect on D-Hashtable. When the number of objects is 1m, the update efficiency is quite slow. For D-Rtree, the update efficiency is initially rapid and then slows with an increasing number of objects. The update speed is the fastest when the number of objects is 50 m–75 m.

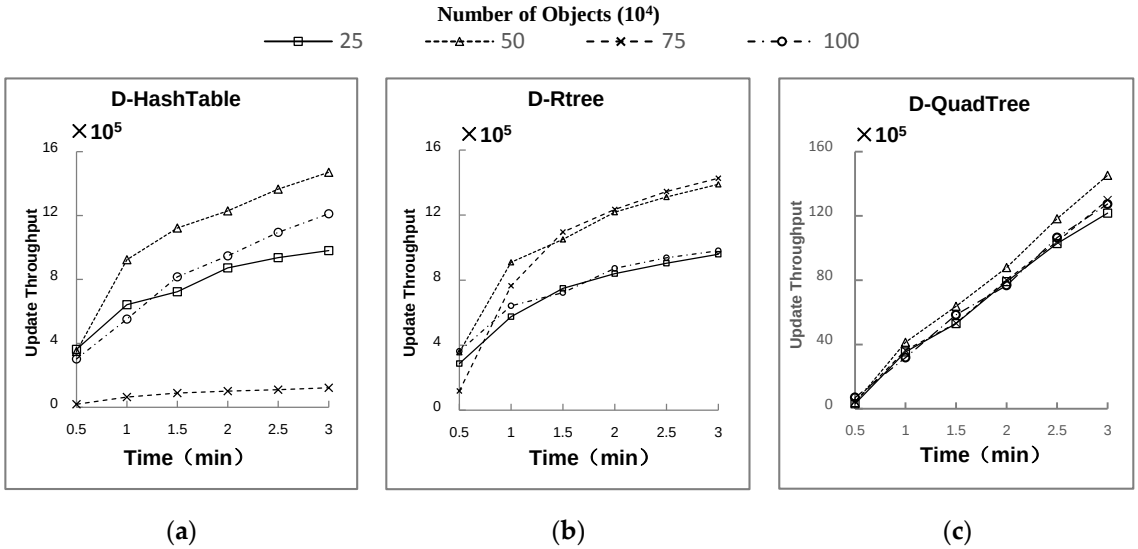


Figure 9. Effect of the number of objects on the spatial join update throughput. (**a**) Update throughput of D-Hahtable; (**b**) update throughput of D-Rtree; (**c**) update throughput of D-Quadtree.

- Study 3: Influence of the number of grid cells on the spatial join performance

The effect of the number of grid cells on the spatial join performance has two main characteristics: (a) For a specific point, the number of cells into which the grid is partitioned and the number of ACs are directly related, that is, a more finely partitioned grid leads to more ACs, which increases the replication of points and reduces the update efficiency; (b) A grid with larger partitions will lead to an imbalance in the numbers of moving objects in different grid cells. Cells in denser regions of the road network will store more points. As Figure 10a–c show, for D-Rtree, the first factor dominates, whereas for D-HashTable, the second dominates. Both factors have some impact on D-Quadtree.

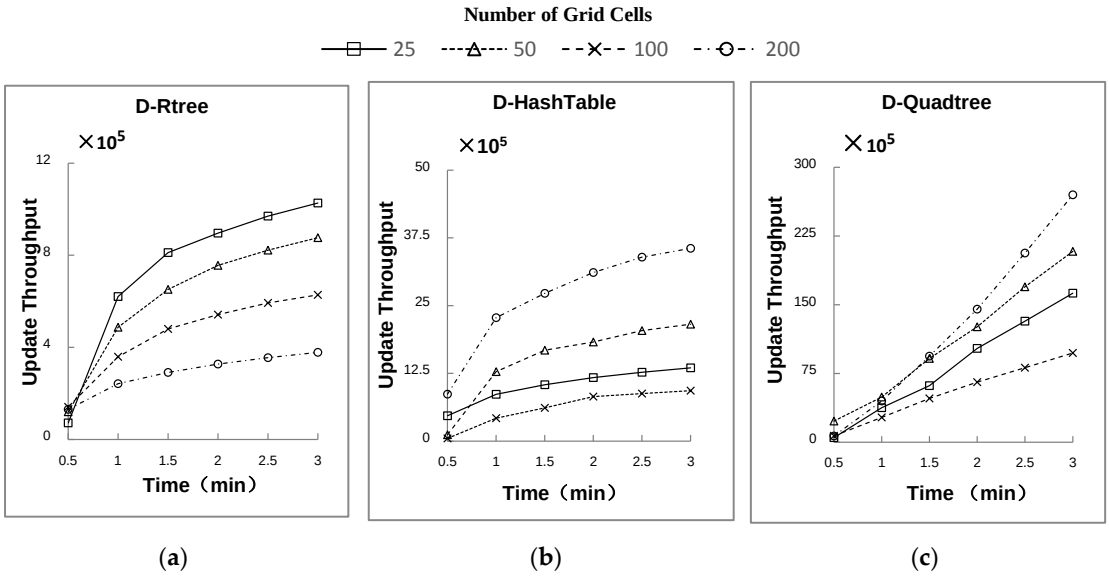


Figure 10. Effect on the update throughput of different partition sizes over time. (**a**) Update throughput of D-Rtree; (**b**) update throughput of D-Hahtable; (**c**) update throughput of D-Quadtree.

- Study 4: Influence of the query range size on the spatial join performance

Figure 11a–c show that an increase in the update/query message ratio will cause an increase in the total spatial join throughput, especially for D-Quadtree and D-HashTable. This suggests that the query process has a lower latency than the update process. In addition, a larger query range size has a smaller but also positive influence on throughput. This is because Storm sends a query message to each bolt instance for execution. When the query scope is large, the number of working executors is greater.

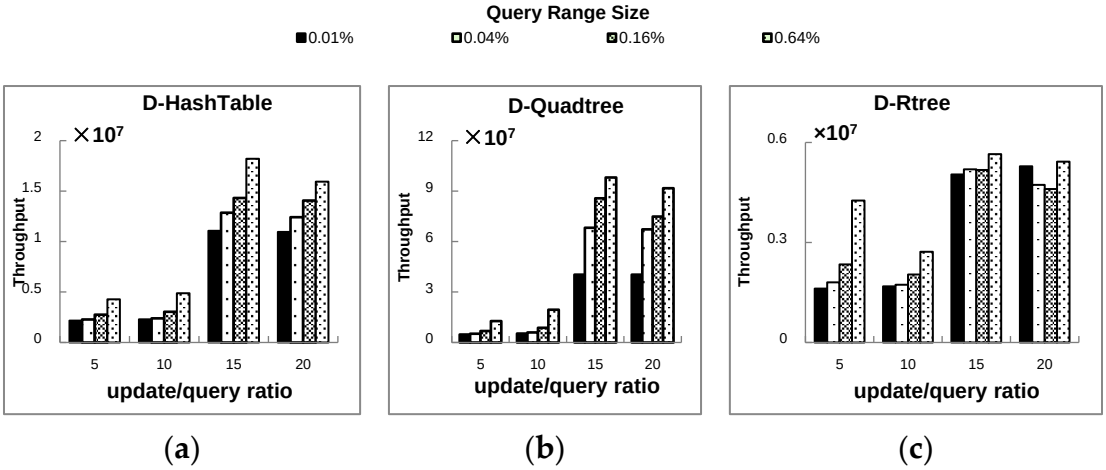


Figure 11. Effects of different query range sizes for various query ratios under a mixed workload. The search area is accessed in a spatially random manner. (**a**) Update and query throughput of D-Hahtable; (**b**) update and query throughput of D-Quadtree; (**c**) update and query throughput of D-Rtree.

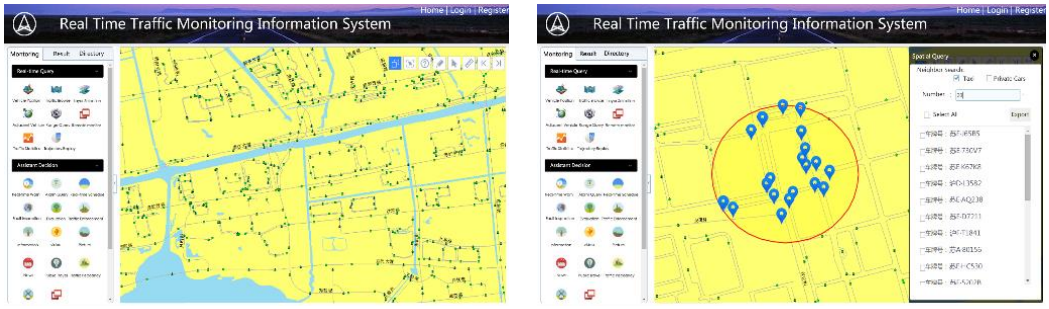
6. Conclusions and Applications

The mobility of people and vehicles in motion is a basic element of human society. Exploring the patterns and trends of human and vehicle mobility can advance the understanding of regional and urban dynamics and reveal the underlying socioeconomic driving forces at work [1]. In this paper, we develop distributed spatial indexes for storing rapidly changing SFD for moving objects based on Apache Storm. In particular, two different solutions are proposed, one for spatial queries on a single dataset and one for range spatial join queries between two datasets. Several factors that may affect the efficiency of the distributed index were explored through experiments. The experiments show that a quadtree index offers better computational efficiency regardless of the number of moving objects, the number of grid partitions or the query range size. For such an evaluation, we should consider not only the query efficiency but also the update efficiency of the secondary index. A quadtree index shows the best performance for the following reasons: 1. The add and delete operations in an R-tree may lead to the merging or splitting of leaf nodes. In some special cases, the height of the tree will increase or decrease, which will lead to low efficiency in the updating of an R-tree; 2. A spatial query must iterate through the all elements in a hash table. Consequently, the spatial query efficiency of a hash table index is relatively low. Compared with these two index structures, the quadtree structure offers better computational efficiency in both query operations and update operations and thus exhibits better performance.

Theoretically, for more rapidly changing spatial data, we can increase the number of work nodes to provide more computing resources. However, each tuple message will stream to a single executor when all kinematic points are assigned to the same grid partition. Therefore, in the future, we will investigate how to address hyper-skewed workloads in the proposed Storm-based method. In addition, emerging distributed streaming frameworks, such as Apache Flink, are showing strong vitality and excellent development prospects (in particular, Apache Flink offers much better throughput than

Apache Storm with relatively low latency). Our future research work will focus on comparing these frameworks with Storm and selecting the best distributed platforms for GIS applications.

This spatial query approach is an important server-side component in our web information system for real-time traffic monitoring. This system has been successfully applied for traffic monitoring and vehicle scheduling and provides traffic decision support for the government (see Figure 12). Our tool has facilitated the easy exploration of big trajectory data. The developed method will enable advancements in a broad spectrum of applications by assisting researchers in tackling the challenges posed by big data.



(**a**) (**b**)

Figure 12. The web interface of our real-time traffic monitoring system, which shows a visualization of moving objects that updates every minute, while the spatial index is updated every second. (**a**) Real-time traffic monitoring; (**b**) nearest neighbor search.

Acknowledgments: This research was funded by the National Natural Science Foundation of China (41471313, 41671391), the Science and Technology Project of Zhejiang Province (2014C33G20), the Public Science and Technology Research Funds Projects (2015418003), and the National Science Foundation (ACI-1535031, 1535081).

Author Contributions: Feng Zhang conceived and designed the study and also provided the funding; Ye Zheng contributed to the study design, made improvements to the algorithm and drafted the manuscript; Dengping Xu contributed to the data acquisition and experimental study; Zhenhong Du was involved in data acquisition and revision of the manuscript; Yingzhi Wang was involved in data acquisition and analysis, worked on aspects of the experiment evaluation, and drafted the manuscript; Renyi Liu edited the manuscript; and Xinyue Ye improved the conceptual framework and updated the manuscript. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Huang, X.; Zhao, Y.; Yang, J.; Zhang, C.; Ma, C.; Ye, X. TrajGraph: A graph-based visual analytics approach to studying urban network centralities using taxi trajectory data. *Vis. Comput. Graph.* **2016**, *22*, 160–169. [CrossRef] [PubMed]
2. Wang, Y.; Liu, Z.; Liao, H. Improving the performance of GIS polygon overlay computation with MapReduce for spatial big data processing. *Clust. Comput.* **2015**, *18*, 507–516. [CrossRef]
3. You, S.J.; Zhang, L.G. Large-scale spatial join query processing in cloud. In Proceedings of the IEEE International Conference on Data Engineering Workshops, Seoul, Korea, 13–17 April 2015.
4. Fast Data: The Next Step after Big Data. Available online: http://www.infoworld.com/article/2608040 (accessed on 13 September 2016).
5. Stojanović, D.N.; Turanjanin, J. Processing big trajectory and Twitter data streams using Apache STORM. In Proceedings of the 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), Niš, Serbia, 14–17 October 2015.
6. Zhao, S.; Chandrashekar, M.; Lee, Y. Real-time network anomaly detection system using machine learning. In Proceedings of the 11th International Conference on the Design of Reliable Communication Networks, Kansas City, MO, USA, 24–27 March 2015.

7. Iwerks, G.S.; Samet, H.; Smith, K.P. Maintenance of K-nn and spatial join queries on continuously moving points. *ACM Trans. Database Syst.* **2006**, *31*, 485–536. [CrossRef]
8. Park, K. An efficient scalable spatial data search for location-aware mobile services. *J. Inf. Sci. Eng.* **2015**, *31*, 165–178.
9. Kwon, D.; Lee, S. Indexing the current positions of moving objects using the lazy update R-tree. In Proceedings of the Third International Conference on Mobile Data Management, Singapore, Singapore, 8–10 January 2002.
10. Pfoser, D.; Jensen, C.S.; Theodoridis, Y. Novel approaches to the indexing of moving object trajectories. In Proceedings of the 26th VLDB Conference, Cairo, Egypt, 10–14 September 2000.
11. Xu, J.; Guting, R.H.; Zheng, Y. The TM-RTree an index on generic moving objects for range queries. *Geoinformatica* **2015**, *19*, 487–524. [CrossRef]
12. Tao, Y.; Papadias, D.; Sun, J. The TPR-tree: An optimized spatio-temporal access method for predictive queries. In Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 9–12 September 2003.
13. Tao, Y.; Papadiasa, D. *MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries Dept*; Hong Kong University: Hong Kong, China, 2000.
14. Jensen, C.S.; Lin, D.; Ooi, B.C. Query and update efficient B $\pm$ Tree based indexing of moving objects. In Proceedings of the 30th VLDB Conference, Toronto, ON, Canada, 31 August–3 September 2004.
15. Šaltenis, S.; Jense, C.S.; Leutenegger, S.T. Indexing the positions of continuously moving objects. In Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 16–18 May 2000.
16. Chen, N.; Shou, L.D.; Chen, G. Adaptive indexing of moving objects with highly variable update frequencies. *J. Comput. Sci. Technol.* **2008**, *23*, 998–1014. [CrossRef]
17. Wu, W.; Tan, K. ISEE: Efficient continuous K-nearest-neighbor monitoring over moving objects. In Proceedings of the 19th International Conference on Scientific and Statistical Database Management, Banff, AB, Canada, 9–11 July 2007.
18. Šidlauskas, D.; Ross, K.A.; Jensen, C.S. Thread-level parallel indexing of update intensive moving-object workloads. In Proceedings of the 12th International Symposium on Spatial and Temporal Databases, Minneapolis, MN, USA, 24–26 August 2011.
19. Deng, Z.; Wu, X.; Wang, L. Parallel processing of dynamic continuous queries over streaming data flows. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *82*, 834–846. [CrossRef]
20. Xiong, D.; Marble, D.F. Strategies for real-time spatial analysis using massively parallel SIMD computers: An application to urban traffic flow analysis. *Int. J. Geogr. Inf. Syst.* **1996**, *10*, 769–789. [CrossRef]
21. Šidlauskas, D.; Šaltenis, S.; Jensen, C.S. Trees or grids? Indexing moving objects in main memory. In Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 4–6 November 2009.
22. Lee, M.L.; Hsu, W.; Jense, C.S. Supporting frequent updates in R-trees: A bottom-up approach. In Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 12–13 September 2003.
23. Šidlauskas, D.; Šaltenis, S.; Jensen, C.S. Processing of extreme moving-object update and query workloads in main memory. *VLDB J.* **2014**, *23*, 817–841. [CrossRef]
24. You, S.; Zhang, J.; Le, G. Spatial join query processing in cloud: Analyzing design choices and performance comparisons. In Proceedings of the International Conference on Parallel Processing Workshops, Beijing, China, 16–19 August 2015.
25. Zhang, S.; Han, J.; Liu, Z. SJMR: Parallelizing spatial join with MapReduce on clusters. In Proceedings of the IEEE International Conference on Cluster Computing & Workshops, New Orleans, LA, USA, 31 August–4 September 2009.
26. Lu, W.; Shen, Y.; Chen, S. Efficient processing of k nearest neighbor joins using MapReduce. *Proc. VLDB Endow.* **2012**. [CrossRef]
27. Akdogan, A.; Demiryurek, U.; Banaeikashani, F.; Shahabi, C. Voronoi-based geospatial query processing with MapReduce. In Proceedings of the IEEE Second International Conference on Cloud Computing Technology & Science, Indianapolis, Indiana, IN, USA, 15–19 November 2010.

28. Zhong, Y.Q.; Han, J.Z.; Zhang, T.Y. Towards parallel spatial query processing for big spatial data. In Proceedings of the Parallel & Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012.
29. Aji, A.; Wang, F.; Vo, H. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *Proc. VLDB Endow.* **2013**. [CrossRef]
30. Eldawy, A.; Mokbel, M.F. A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data. *Proc. VLDB Endow.* **2013**. [CrossRef]
31. Yu, J.; Wu, J.; Sarwat, M. A demonstration of GeoSpark: A cluster computing framework for processing big spatial data. In Proceedings of the IEEE International Conference on Data Engineering, Helsinki, Finland, 16–25 May 2016.
32. Baig, F.; Mehrotra, M.; Wang, F. SparkGIS: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *VLDB Workshops*; Big-O(Q) and DMAH: Waikoloa, HI, USA, 2015; pp. 134–146.
33. Xie, D.; Li, F.; Li, G. Simba: Efficient in memory spatial analytics. In Proceedings of the 2016 International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016.
34. Allen, S.T.; Jankowski, M.; Pathirana, P. Basic Storm concepts. In *Storm Applied: Strategies for Real-Time Event Processing*; Manning Publications: Shelter Island, NY, USA, 2015; pp. 17–29.
35. MouRatidis, K.; Papadias, D.; Hadjieleftheriou, M. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, USA, 13–16 June 2005.
36. Dittrich, J.; Blunschi, L.; Salles, M.A. Movies: Indexing moving objects by shooting index images. *Geoinformatica* **2011**, *15*, 727–767. [CrossRef]
37. Bentley, J.L.; Friedman, J.H. Data structures for range searching. *ACM Comput. Surv.* **1979**, *11*, 397–409. [CrossRef]
38. Wang, H.; Zimmermann, R. Processing of continuous location-based range queries on moving objects in road networks. *IEEE Trans. Knowl. Data Eng.* **2011**, *23*, 1065–1078. [CrossRef]
39. Tauheed, F.; Heinis, T.; Ailamaki, A. Thermal-join: A scalable spatial join for dynamic workloads. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Melbourne, Australia, 9–16 July 2015.
40. Corral, A.; Torres, M.; Vassilakopoulos, M. Predictive join processing between regions and moving object. In Proceedings of the 12th East European Conference, Pori, Finland, 5–9 September 2008.
41. Ward, G.D.; He, Z.; Zhang, R. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *VLDB J.* **2014**, *23*, 965–985. [CrossRef]
42. Kalashnikov, D.V.; Prabhakar, S.; Hamrusch, S.E. Main memory evaluation of monitoring queries over moving objects. *Distrib. Parallel Databases* **2004**, *15*, 117–135. [CrossRef]
43. Gedik, B.; Liu, L. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Advances in Database Technology—EDBT 2004*; Springer: Philadelphia, PA, USA, 2004; Volume 2992, pp. 67–87.
44. Zhang, R.; Qi, J.Z.; Lin, D. A highly optimized algorithm for continuous intersection join queries over moving objects. *VLDB J.* **2012**, *21*, 561–586. [CrossRef]
45. Mokbel, M.F.; Xiong, X.; Aref, W.G. PLACE: A query processor for handling real-time spatio-temporal data streams. In Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, ON, Canada, 29 August–3 September 2004.
46. Xiong, X.P.; Mokbel, M.F.; Aref, W.G. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In Proceedings of the 21st International Conference on Data Engineering, Tokyo, Japan, 5–8 April 2005.
47. Brinkhoff, T. A framework for generating network-based moving objects. *Geoinformatica* **2002**, *6*, 153–180. [CrossRef]

© 2016 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) license (http://creativecommons.org/licenses/by/4.0/).