future internet

MDPI

Article

THBase: A Coprocessor-Based Scheme for Big Trajectory Data Management

Jiwei Qin, Liangli Ma * and Jinghua Niu

College of Electronic Engineering, Naval University of Engineering, Wuhan 430033, China; 18602706401@163.com (J.Q.); niujh93@126.com (J.N.)

* Correspondence: maliangl@163.com; Tel.: +86-134-0710-4236

Received: 5 November 2018; Accepted: 27 December 2018; Published: 3 January 2019

Abstract: The rapid development of distributed technology has made it possible to store and query massive trajectory data. As a result, a variety of schemes for big trajectory data management have been proposed. However, the factor of data transmission is not considered in most of these, resulting in a certain impact on query efficiency. In view of that, we present THBase, a coprocessor-based scheme for big trajectory data management in HBase. THBase introduces a segment-based data model and a moving-object-based partition model to solve massive trajectory data storage, and exploits a hybrid local secondary index structure based on Observer coprocessor to accelerate spatiotemporal queries. Furthermore, it adopts certain maintenance strategies to ensure the colocation of relevant data. Based on these, THBase designs node-locality-based parallel query algorithms by Endpoint coprocessor to reduce the overhead caused by data transmission, thus ensuring efficient query performance. Experiments on datasets of ship trajectory show that our schemes can significantly outperform other schemes.

Keywords: trajectory data; HBase; coprocessor; spatiotemporal query

1. Introduction

In recent years, with the rapid development of mobile networks and sensor technologies, trajectory data of MO (Moving Object) has exploded, using traditional database in storing and querying the massive trajectory data cannot meet the requirements already [1]. With the rapid development of distributed computing, the rise of technologies such as Hadoop [2], Spark [3], and NoSQL [4] has made it possible to access mass data effectively. Compared with traditional schemes, the distributed solutions can easily use cluster resources to satisfy the needs of mass data management for computing and storage, and they have excellent features such as ease of maintenance, manageability, and scalability. Therefore, it gradually becomes a new tendency that massive trajectory data are managed based on a distributed solution.

Trajectory data is a type of spatiotemporal data, so the existing schemes for spatial data or spatiotemporal data management can be applied to it. Recently, some prototype systems have been proposed for processing such data, e.g., MD-HBase [5], SpatialSpark [6], STEHIX [7], and Simba [8]. However, the time attribute is not considered in the schemes for spatial data management, so a time-related query may result in unpredictable time requirements [7]. Compared with spatial data schemes, the schemes for spatiotemporal data management provide support for efficient time-related query as they encode the time attributes in their indexes. However, they ignore those important non-spatiotemporal attributes such as MOID (Moving Object Identifier), and it is difficult to perform efficient non-spatiotemporal-based queries [9]. In addition, there are some differences in query processing between the trajectory data and the usual spatiotemporal data. For example, when processing a trajectory-based spatiotemporal range query, in addition to satisfying the query conditions,

the query results should be output in the form of trajectories rather than points. For another, when processing a trajectory-based *k*-NN query, the query results must belong to different MOs. It can be seen that the trajectory-based queries usually involve MO-based grouping operations, but which are not supported in the spatiotemporal data management schemes.

On the basis of spatial or spatiotemporal data research, some schemes especially for trajectory data have also been proposed, such as SPDM [10], Elite [11], TrajSpark [12], UlTraMan [13], HBSTR-tree [14] et al. These schemes consider both spatiotemporal attributes and important non-spatiotemporal attributes, enabling support for trajectory-based spatiotemporal queries and non-spatiotemporal queries. However, most of these schemes do not consider the impact of data transmission on queries, it is mainly reflected in the following two aspects:

- Some schemes [10,14] treat index and data as independent units and thus, in a distributed setting, they can reside on different nodes. When processing queries, the isolation of index and data results in the additional cross-node data transmission for index access.
- Some schemes [10–13] split the trajectory data of the same MO into multiple subtrajectories and store them in different partitions on different nodes. During query execution, it is often necessary to obtain subtrajectories from different nodes, and merge or sort them into a whole trajectory for further process.

With the query data scale increment, the big data transmission will generate notable network communication, and also includes an abundance of serialization and deserialization operations, resulting in a large occupation of computing, storage, and network resources, and has an enormous impact on query efficiency [15].

It can be seen that the data transmission problem is a huge challenge to implement the trajectory data management system under the distributed environment. To solve the data transmission problem, we designed and implemented a trajectory data management scheme called THBase (Trajectory on HBase [16]) through HBase coprocessor. HBase is an open-source, distributed, non-relational database modeled after Google's Bigtable, which enables real-time random access support for big data in the Hadoop environment. HBase uses Region as the basic element of availability and distribution, and can achieve parallel access to Regions through Observer and Endpoint coprocessors. Compared to other distributed frameworks, HBase coprocessor can easily achieve the colocation between programs and data, thus effectively reducing data transmission cost. So we implement our massive trajectory data management schemes based on Hbase coprocessor.

Based on the consideration of reducing data transmission, and taking into account pruning, sort, and footprint factors, THBase proposes a segment-based data model and an MO-based partition model to store trajectories. To accelerate spatiotemporal query processing, it imports a local indexing structure for each Region through Observer coprocessor. Furthermore, we design certain maintenance strategies to ensure the colocation between Region and its corresponding index. On this basis, THBase designs node-locality-based parallel query algorithms by Endpoint coprocessor to reduce the data transmission overhead when querying. Our main contributions can be summarized as follows:

- We propose a segment-based data model and an MO-based partition model, which provides effective support for massive trajectory data storage in HBase.
- We introduce an Observer coprocessor-based local indexing framework, which provides efficient support for the spatiotemporal queries and achieves the colocation for index and the indexed data.
- We propose Endpoint coprocessor-based parallel algorithms for processing spatiotemporal queries to improve query efficiency.
- We evaluate our scheme with the dataset of ship trajectory, the experimental results verify the efficiency of THBase.

Section 2 introduces the preliminaries about the trajectory data, the basics in HBase, and the related work. The overview of THBase is introduced in Section 3. The three main modules, T-table,

L-index, and Query Processing Module, are described in Section 4, Section 5, and Section 6, respectively. Section 7 provides an experimental study of our system. Finally, we give a brief conclusion in Section 8.

2. Preliminary

2.1. Related Definition

There are usually two forms of trajectory data: discrete and continuous. In this paper, we use discrete form to represent trajectory data. The data types of trajectory are defined as follows:

Definition 1. *A trajectory point p is a location-based data element, denoted as <moid, l, t, o>, where moid is the MOID of an MO, l and t are the spatial and timestamp information, and o is the other information which includes speed, heading*, etc.

Definition 2. *A trajectory T contains a sequence of ship trajectory points* $(p_1, \ldots, p_m)$*, orderly by their timestamps, and all points of T belong to the same MO.*

Definition 3. *A trajectory segment (TS) contains the trajectory points of an MO sampled in a TI (Time Interval). Where TI is generated by unified division in time dimension, namely all the TIs are of the same length, so the start time of a TI can be its identifier. From the definition, it is easy to know that a trajectory segment can be uniquely identified by MOID and TI attributes.*

In Euclidean Space with n dimension, given a coordinate point $q = (q_1, 2026, q_n)$ and a rectangle $E = (s,r)$, where $s = (s_1, \ldots, s_n)$ and $r = (r_1, \ldots, r_n)$ are the two vertices on the main diagonal of E, and for all $k = 1$ to n, $r_k \geq s_k$. For the distance between E and q, there are the following definitions:

Definition 4. *Mindist(E,q) is the shortest distance from E to q [17]; it is computed as below:*

$$\text{Mindist}(E, q) = (\sum_{i=1}^{n} |E_i - q_i|^2)^{1/2}. \tag{1}$$

where:

$$E_i = \begin{cases} s_i & \text{if } q_i < s_i \\ r_i & \text{if } q_i > r_i \\ q_i & \text{otherwise} \end{cases}. \tag{2}$$

Definition 5. *MinMaxdist(E, q) is the minimum overall dimensions distance from q to the furthest point of the closest face of E [17]; it can be expressed as:*

$$\text{MinMaxdist}(E, q) = (\min(|E_{mk} - q_k|^2 + \sum_{\substack{i \neq k \\ 1 \leq i \leq n}} |E_{mi} - q_i|^2))^{1/2}. \tag{3}$$

where:

$$\begin{aligned} &1 \leq k \leq n \\ &E_{mk} = \begin{cases} s_k & \text{if } q_k \leq (s_k + t_k)/2 \\ t_k & \text{otherwise} \end{cases} \\ &E_{mi} = \begin{cases} s_i & \text{if } q_i \geq (s_i + t_i)/2 \\ t_i & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

2.2. HBase

HBase is an open-source distributed key-value database. The technology is derived from Google's Bigtable concept. It dispersedly stores the data of a table over a cluster of nodes, in order to provide scalable storage to massive data.

Different from traditional relational data model, data in HBase is stored in tables, and each data cell is indexed by rowkey, column family, column qualifier, and time stamp. Among them, rowkey is the unique index for directly accessing data, it is closely related to data storage and query efficiency, and plays a very important role in data processing.

On the physical structure, HBase uses Region as the minimum unit for distributed storage. An HBase table is usually divided horizontally into multiple Regions scattered in each work node (named as RegionServer), each Region stores a certain range of data rows. A Region is decided to be split when store file size goes above threshold or according to defined region split policy, in order to make the data distribution of Regions more uniform. Split operation takes a row as the split point, and the rowkey range of the split Region changes after split [18].

In order to leverage the parallelism of HBase clusters, a coprocessor mechanism is introduced in HBase 0.92, which is similar to a lightweight MapReduce framework, enabling the execution of custom code in RegionServers. The coprocessor can be divided into two kinds, one is Observer, and the other is Endpoint. The Observer can be thought of like database triggers, and provides event hooks of table operation to capture and process certain events, while Endpoint is similar to the storage process of traditional database. One can invoke an Endpoint at any time from the client, the Endpoint implementation will then be executed remotely in RegionServers, and results from those executions will be returned to the client [19].

2.3. Related Work

Distributed Spatial Data Management Schemes: In recent years, many hadoop-based massive spatial data management schemes have been proposed. MD-HBase [5] is built on top of HBase, and uses Z-ordering curves algorithms to map spatial objects into single-dimension space separately. EDMI [20] implements spatial data indexing through a distributed multilevel index structure, and effectively avoids overlap of Minimum Bounding Rectangles (MBRs) and achieves better query efficiency. SpatialSpark [6] is implemented on the basis of Spark framework; compared to other schemes, it specially adds support for spatial joint queries. However, the abovementioned schemes do not consider time attribute, so they are inefficient to apply directly to spatiotemporal data management.

Distributed Spatiotemporal Data Management Schemes: In order to solve the above problem, recently, some schemes particularly implemented for big spatiotemporal data management have been implemented. STEHIX [7] builds a spatiotemporal hybrid index structure based on HBase storage layer, and provides efficient support for range query, *k*-NN query, and G-NN query on this basis. In order to support frequent updates of spatiotemporal stream data and real-time multidimensional queries, UQE-Index [21] indexes the current data and historical data at different levels of granularity, respectively, and adopts certain strategies to reduce the maintenance cost of the index. Simba [8] uses IndexTRDD to compress and store spatiotemporal data on the top of Spark, and combines global index and local index to implement efficient parallel queries. Since these schemes do not consider the important non-spatiotemporal attributes such as MOID, they are difficult to implement full support for trajectory data queries.

Distributed Trajectory Data Management Schemes: On the basis of spatial or spatiotemporal data research, some schemes especially for trajectory data have also been proposed. Ke et al. [14] propose a hybrid indexing structure named HBSTR-tree, which combines hash table, B*-tree, and spatiotemporal R-tree, and implemented a practical storage schema for massive trajectory data based on MongoDB. Aydin et al. [10] proposed four different table schemas for massive trajectory data storage in No-SQL database, and implemented two index structures on this basis to achieve full support for spatiotemporal and non-spatiotemporal queries. Jie Bao et al. [22] designed a trajectory data management system on

Microsoft Azure, this system optimizes the storage schema, index method and query strategy based on different functionalities to guarantee the efficient trajectory updates and perform the services in real time. Trajspark [12] proposes a new abstraction RDD structure to manage trajectories as a set of trajectory segments, and imports the global and local indexing mechanism to provide support for real-time query. Some above schemes [10,14] isolate index and data on different nodes. When processing queries, this data distribution results in the additional cross-node data transmission for index access. Moreover, most schemes [10,12,22] split the trajectory data of the same MO into multiple subtrajectories and store them in different partitions on different nodes. During query execution, they usually require more overhead of data transmission and sort to merge the subtrajectories.

3. Overview

Figure 1 gives a full picture of THBase. It is composed of three modules:

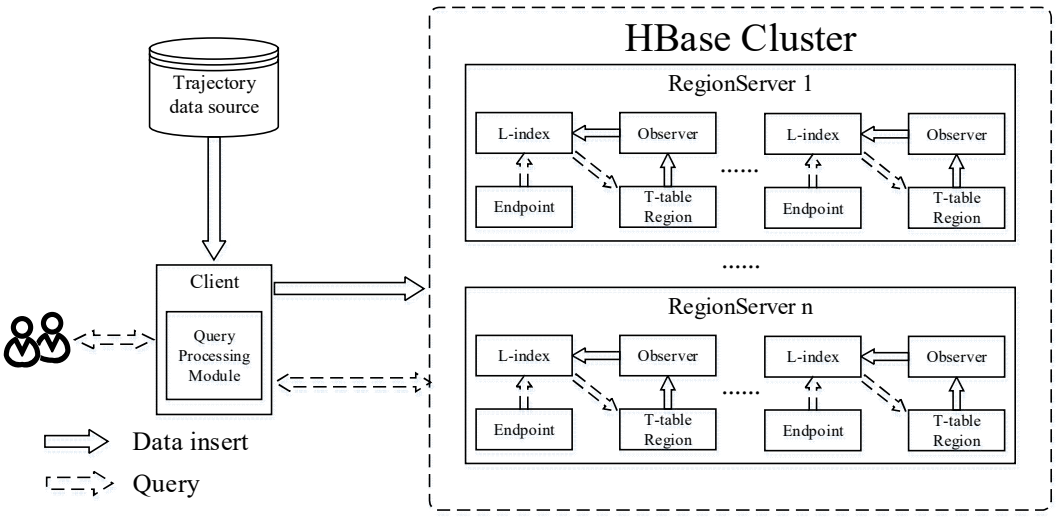

Figure 1. THBase Architecture.

- T-table: T-table is an HBase table for persisting trajectory data. Considering the pruning and footprints factors, it organizes the trajectory data based on segment model, and processes data partitioning by MO-based model to ensure centralized distribution of all data of the same MO. This module is detailed in Section 4.
- L-index: L-index is an in-memory local secondary index structure for each Region of T-table, and is built by Observer coprocessor when trajectory data is inserted in Region. It uses a hybrid grid structure to index the spatiotemporal attribute of all trajectory segments. During query processing, each L-index generates the candidate rowkey collection for querying the corresponding Region by pruning and sorting. (Detailed in Section 5).
- Query Processing Module: The Query Processing Module implements efficient support for three typical queries by calling Endpoint coprocessor in the HBase cluster. We give a detailed description in Section 6.

4. T-Table

T-table is used to persist trajectory data; we design the schema of T-table by considering factors such as data transmission, pruning, footprints, and sort, like the following:

- Rowkey: Rowkey design is the key of table schema design. Rowkey not only affects the access efficiency of HBase table, but also influences the secondary index (The secondary index of HBase usually maps specific columns or expressions to rowkeys [23]). For trajectory data, the point-based rowkey model can maximize the predictive accuracy of pruning, but greatly increases the space

overhead of secondary index; conversely, the MO-based rowkey model reduces the space overhead of secondary index, but it weakens the pruning power. Based on the consideration of space and pruning, we designed a rowkey structure based on trajectory segment. According to Definition 3, a T-table rowkey is shown as follows:

$$rowkey = \text{in}(moid) + ti, \tag{5}$$

where *moid* is the MOID of an MO, and *ti* is the identifier of a TI. The in(*moid*) function is used to inverse MOID. Because the initial digits of MOID usually have a certain distribution pattern, it is relatively easier to make data unevenly distributed. The final digits of MOID show a certain randomness, so in(*moid*) ensure that the MOID values are randomly distributed in each Region. Corresponding to the rowkey model, the processing of data insert and index update are performed based on trajectory segment.

- Column family and column: While HBase does not do well with multiple column families, one column family named *Traj* is assigned to store all trajectory data. The column qualifier is represented by the timestamp information of trajectory point, in order to ensure that trajectory points are stored in chronological order. Location and other information are stored as data cell.
- Partitioning: In terms of data partitioning, a MO-based data partitioning model is adopted. During initialization of T-table, we evenly divide the range of the inversing MOID to determine the rowkey range of each Region, to ensure that all data of the same MO is stored in one Region. When the store file size of a Region goes above threshold, this Region needs to split to adjust the load. In order to avoid distributing the data of the same MO to different Regions after split, we should implement a MO-based split strategy. With the aid of KeyPrefixRegionSplitPolicy [16] interface of Region split, we can simply implement the MO-based split strategy by specifying the inversing MOID structure as the prefix of rowkey. Following this strategy, HBase will ensure that rows with the same prefix locate in the same Region after split, namely all data of the same MO is still stored in a Region.

Figure 2 displays the storage process of trajectory data in T-table, where trajectories T_1 and T_2 are preprocessed as trajectory segments, and then inserted into rows with rowkeys rk_{11}, rk_{12}, rk_{21}, rk_{22}, respectively. Based on the dictionary order of HBase, the trajectory points of T_1 or T_2 are stored in chronological order. Therefore, if the query rowkeys are orderly, there is no need to sort the query results. In addition, the data of T_1 or T_2 always locate in the same Region, so when reconstructing the trajectory points into a trajectory, there is no need to request data from other Regions, thus avoiding data transmission and ensuring query efficiency. In order to avoid ambiguity, in the following we use tRegion to refer to the Region of T-table.

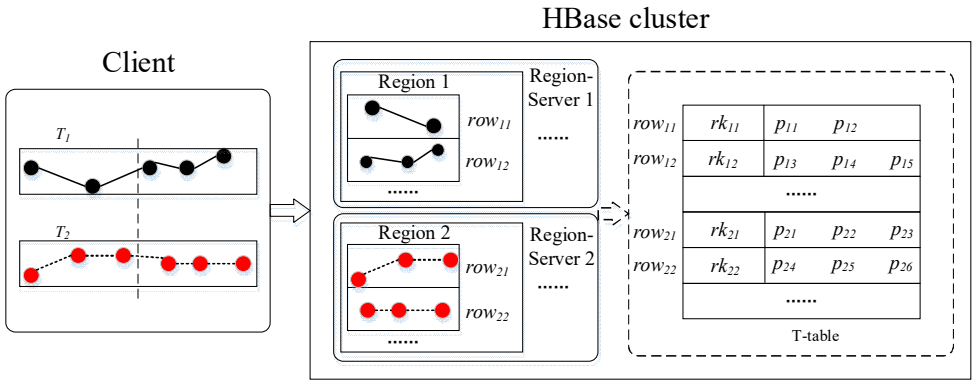

Figure 2. Storage process of trajectory data in T-table.

5. L-Index

In HBase, the rowkey provides the only access function as a primary index, and a query for the attributes which are not rowkey structures means a full table scan. Because the rowkey structure of T-table does not include spatiotemporal attribute, it is difficult to satisfy the complex spatiotemporal query requirements by using only rowkey. Therefore, an index structure capable of supporting spatiotemporal queries is necessary. In addition, for the purpose of reducing communication overhead, we hope that the tRegion and the corresponding index data are distributed as much as possible on the same node. Therefore, if a tRegion is split or migrated, the corresponding index data also needs to be split or migrated. In this regard, with the aid of Observer coprocessor, we built a local index structure named L-index for indexing the spatiotemporal attribute of trajectory segments to effectively support complex spatiotemporal query, and implement certain maintenance strategies to ensure that the L-index is colocated with the corresponding tRegion.

This section first introduces the basic structure of L-index, and then discusses the query, update, and maintenance of it.

5.1. Structure

L-index is a spatiotemporal local (Region level) index structure, which resides in memory so that it can respond quickly to queries. As shown in Figure 3, L-index has a two-level structure: The level-1 is a time periods index called T-index, and the level-2 is named as G-index which indexes spatial attribute by multilevel grid.

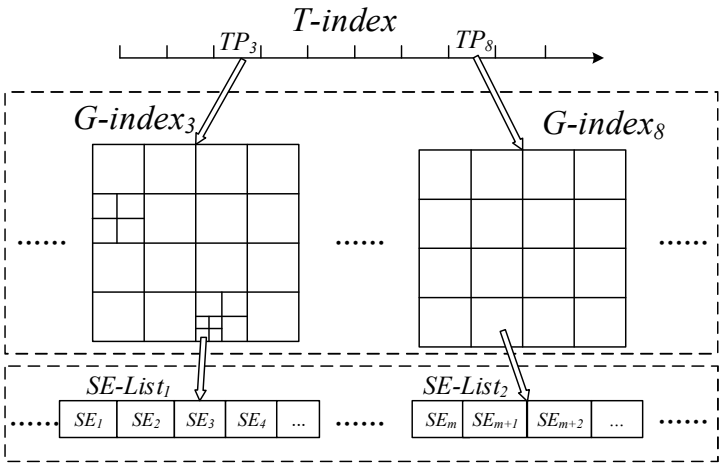

Figure 3. L-index Structure.

- T-index: For constructing T-index, we divide the time dimension into multiple Time Periods (TPs) in accordance with time series, where each TP element corresponds to a G-index to index all trajectory segments over it. Each TP element is obtained by combining several consecutive TI elements to avoid a single trajectory segment being indexed by multiple T-indexes.
- G-index: For building G-index, we first divide the space evenly with a certain size grid, and each grid cell should be subdivided by using a quad-tree structure if it overlaps with too many trajectory segments. Such consideration is more helpful for dealing with the skewed data. Therefore, a G-index can be divided into multigrade level, and a trajectory segment may intersect with multiple grid cells of different levels at the same time. In order to reduce the redundancy and ensure the efficiency of index, a trajectory segment is indexed by the deepest level cell that can completely enclose its MBR (Minimum Bounding Rectangle). If a trajectory segment MBR cannot be completely enclosed in any of a grid cell, it is indexed in all of the first-level grid cells that intersect its MBR. Each G-index grid cell corresponds to a list of index entries, and all of the index entries (denote the index entry of trajectory segment as **SE**, and the list as **SE-List**) in SE-List

are organized in rowkey value order, which is convenient for querying the G-index grid cell to directly obtain the orderly rowkey results. We use a tuple <*rk*, *m*> to present the data model of SE, where *rk* is the rowkey of the trajectory segment, *m* is the MBR attribute.

Compared to traditional quad-tree, G-index avoids data redundancy inside each first-level cell, so it requires less storage overhead than the former. In addition, G-index implements the orderly organization of index entries, so it can effectively reduce the sorting overhead of subsequent processing.

In contrast with CIF quad-tree (Caltech Intermediate Form quad-tree) [24], G-index requires more storage due to the non-redundancy characteristic of CIF quad-tree. However, in contrast, in addition to the sorting optimization mentioned above, G-index also has the following advantages: G-index can quickly use grid structure to locate the more precise spatial range, and CIF quad-tree needs to recursively query the node and the corresponding index entries from the root node. If the upper nodes contain too many index entries, the query efficiency of CIF quad-tree will be significantly affected. Furthermore, in *k*-NN query processing (detailed in Section 6.3), the incremental iterative strategy can be applied to G-index to avoid access to duplicate cells, while this strategy is difficult to apply to CIF quad-tree.

5.2. Query

Given a time range *tr* and a spatial range *sr*, we perform the spatiotemporal query in L-index like the following.

We first use *tr* to query T-index to obtain the overleaping TP elements. For each of these TP elements, we query the corresponding G-index by using *sr*, if a G-index grid cell overleaps with *sr*, we treat it as the candidate cell and determine whether its subgrids overleap with *sr*, finally we can obtain the candidate cells. Next, in the SE-List corresponding to each candidate cell, each SE is pruned based on MBR and rowkey attributes (through the TI attribute contained in rowkey) to obtain more accurate rowkey query results.

In particular, since the rowkey query results of each G-index grid cell are orderly, a quick sorting and even grouping for all the rowkey results can be achieved by a mergesort operation, the time complexity of this sort operation is O(n), where n is the total number of rowkeys in the results. Through the above sorting operation, we can use the sorted rowkeys to access the corresponding tRegion, thereby avoiding sorting the trajectory point results.

5.3. Update

The update of L-index is performed by Observer coprocessor. After the client inserts trajectory data through Put API, Observer triggers the update of L-index by capturing the postPut event [16]. Given a trajectory segment TS_i, its MBR is m_i, and the rowkey is $rk_i = \text{in}(moid_i) + ti_i$. The update process of L-index is as follows.

We first use ti_i to query T-index to obtain the matching TP element, and then query the first-level grid cells in the corresponding G-index through m_i, there are two cases: (1) If m_i can be enclosed in a single first-level grid cell, the subgrid cells of this cell are queried level by level to select the deepest level grid cell that can enclose m_i to index TS_i. (2) If m_i intersects with multiple first-level grid cells, all the matching first-level grid cells are used to index TS_i.

After TS_i is indexed, we count the number of SE in SE-List of each matching G-index grid cell. If the threshold is exceeded, the grid cell is split based on quad-tree, and the SE-List is divided into corresponding subsets according to the spatial location. If a SE cannot be enclosed in any of the subgrid cells, it is still indexed by the current grid cell.

5.4. Maintenance

The maintenance of L-index includes two aspects: one is to synchronously split the corresponding L-index when a tRegion is split, to ensure one-to-one correspondence between tRegion and L-index.

The second is to synchronously migrate the corresponding L-index when a tRegion is assigned to another RegionServer, to ensure that both always coreside on same node. The maintenance of L-index is also performed by Observer coprocessor.

- TRegion split: After a tRegion is split, Observer captures the postCompleteSplit event to trigger the split of the corresponding L-index [16]. During split processing, the existing grid structure of G-index remains unchanged, each SE-List is directly divided into two sublists by the rowkey of split point, and then each sublist is linked to the corresponding G-index grid cell of new tRegion. The above split operation of L-index avoids the heavy overhead caused by quad-tree splitting, thereby ensuring splitting efficiency. Its time complexity is O(n), where n is the total number of SE in the L-index.
- TRegion assignment: The tRegion assignment usually occurs during load balancing, Observer triggers the migration of L-index by capturing postBalance event [16]. First, the information of assigned Regions, source RegionServers and destination RegionServers are obtained by reading Region Plan parameter [16], and then according to these information, each corresponding L-index is migrated to the same destination RegionServer.

6. Query Processing Module

On the basis of T-table and L-index, the Query Processing Module supports multiple types of query, such as single-object query, spatiotemporal range query, k-NN query, and so on. This section describes above 3 types of query algorithms in detail.

6.1. Single-Object Query

A single-object query retrieves the trajectory T_{so} by the following parameters: $moid_{so}$ and tr_{so}, where $moid_{so}$ is the MOID of a MO and tr_{so} denotes the time range. Since the single-object query does not contain spatial condition, it can be completed by directly scanning the corresponding rowkey range. Algorithm 1 introduces the detailed steps. Firstly, we use $moid_{so}$ and tr_{so} to construct rowkey query range rkr_{so} (line 1). Next, we scan T-table through rkr_{so} to get intermediate results irs_{so} (line 2). Finally, irs_{so} are filtered by tr_{so} to get result T_{so} (it only needs to filter the start and end trajectory segments) and then return it (line 3 to 4).

Algorithm 1. Single-Object Query

Input: Query MOID $moid_{so}$, Query time range tr_{so};
Output: Trajectory T_{so};

1 $rkr_{so} \leftarrow getRowkeyRange(moid_{so}, tr_{so})$;
2 $irs_{so} \leftarrow T\text{-}table.scan(rkr_{so})$;
3 $T_{so} \leftarrow irs_{so}.filter(tr_{so})$;
4 **return** T_{so};

6.2. Spatiotemporal Range Query

A spatiotemporal range query retrieves a set of trajectories according to a time range tr_{st} and a spatial range sr_{st}.

Because spatiotemporal range query involves multiple trajectories, to improve query efficiency, we implement the corresponding parallel query algorithm based on Endpoint Coprocessor. Compared with MapReduce framework, Endpoint is more customizable, with higher query precision, and the format conversion overhead between Region and MapReduce Partitioner is avoided. According to the introduction of T-table and L-index, tRegion and its L-index coreside on the same RegionServer, and the data of the same MO is distributed in the same tRegion. Therefore, when processing a query, each Endpoint-based task can obtain all the index and trajectory data to be processed from the RegionServer where it runs on, and has a high locality level. When performing parallelism, except for the node

where the client is located, there is no need to communicate across nodes, thus effectively controlling communication overhead and improving execution efficiency.

The implementation of Endpoint includes the steps of extending the CoprocessorProtol interface, implementing the interface function of Endpoint, extending the abstract class BaseEndpoint-Coprocessor and specifying the calling mode of the client [25]. Here, the interface function of Endpoint is mainly introduced.

The processing flow is shown in Algorithm 2, where lines 2 to 5 describe interface functions of Endpoint in each tRegion. In the query process of each Endpoint, we first query L-index to get an ordered set of rowkeys ors_{st} by tr_{st} and sr_{st} (see Section 5.2 for the query process in L-index) (line 2). Then, we use ors_{st} to access the corresponding tRegion, and filter the query results by tr_{st} and sr_{st} to generate trajectory points ps_{st} (line 3). After that, ps_{st} are merged into complete trajectories Ts_{st} as the results of tRegion (line 4). Finally, we summarize the results of all tRegions and then return them to users (line 6).

Algorithm 2. Spatiotemporal Range Query

Input: Query time range tr_{st}, Query spatial range sr_{st};
Output: A set of Trajectories;

1 **for** each *tRegion* **do in parallel**
2 $ors_{st} \leftarrow$*L-index.getRowkeys*(tr_{st}, sr_{st});
3 $ps_{st} \leftarrow$*tRegion.get*(ors_{st}).*filter*(tr_{st}, sr_{st});
4 $Ts_{st} \leftarrow ps_{st}$. *mergeByMOID*();
5 **end parallel**
6 **return** *client.collect*(Ts_{st});

6.3. k-NN Query

The k-NN query is defined as follows: Given a time range tr_k and a spatial location q_k, according to the selected distance function, the k trajectories are queried which are nearest to q_k within tr_k. Here, the distance between a trajectory and q_k is computed as the distance from q_k to the nearest trajectory point.

Existing k-NN spatial or spatiotemporal query algorithms do not support k-NN trajectory query very well. Because they do not consider MOID attribute, some candidate elements may belong to the same MO, and it is time-consuming to merge and count the trajectories across nodes [26].

In THBase, we implement an Endpoint-based parallel k-NN query algorithm, which takes full advantage of the parallelism of HBase cluster and reduces the overhead caused by data transfer.

Like spatiotemporal range query, we can also divide the k-NN query task into several independent subtasks, and each subtask can be independently performed through the Endpoint on the corresponding tRegion without cross-node communication. After the executions of all subtasks, we complete the global count task by the client. Here, we introduce the steps of a subtask in k-NN query as follows:

1. Range-based pruning: We first query L-index by tr_k and an estimated spatial range sr_k (see Section 5.2 for query process in L-index). For the obtained candidate SE elements, we group them according to the MOID attribute, and obtain a set $sess = \{ses_1, ses_2, \ldots, ses_n\}$, where each $ses_i = \{se_{i1}, \ldots, se_{im}\}$ $(1 \leq I \leq n)$ is a set of candidate SE elements of the same MO.
2. Pruning for the data of the same MO: Since only one point is generated as a result in the trajectory data of the same MO, we can prune those irrelevant ses_{ij} $(1 \leq j \leq m)$ elements in ses_i according to 1-NN query to further reduce the query range. On the abovementioned principle, we calculate the Mindist and MinMaxdist from the MBR of each se_{ij} to q_k, and then select the minimum MinMaxdist (named as *mMMd*) and the minimum Mindist (named as *mMd*). Based on Lemma 1, if the Mindist of a se_{ij} is above *mMMd*, it should be removed from ses_i.

3. Pruning for the data of different MOs: We first count the number of elements n_k contained in *sess*, namely, the number of candidate trajectories. If n_k is above k, according to Lemma 1, we can remove such ses_i element whose *mMd* is greater than the k-th minimum *mMMd*. Finally, we extract rowkeys from *sess* to get the ordered rowkey set ors_k for querying the corresponding tRegion.
4. Distance calculation: At first, the tRegion is queried by the interested ors_k to obtain trajectory points ps_k. After that, this Endpoint calculates the distance from each trajectory point in ps_k to q_k. For the trajectory points that belong to the same MO, the point with the smallest distance is selected. Finally the number of the selected trajectory points is counted; if it is above k, only the first k minimum distances and their corresponding points are returned to the client. Here, we use lps_k to represent the return results.

Lemma 1. *In k-NN query, if the Mindist of a MBR is above the k-th minimum MinMaxdist value, this MBR must not contain the final result, and it should be pruned [17].*

In order to enhance the readability, we use function *pruning* (q_k, tr_k, k, sr_k) and *distfrom* (q_k, k, ors_k), to represent the above pruning (steps 1 to 3) and distance calculation process (step 4), respectively.

Algorithm 3 introduces the detailed steps. First of all, we estimate a spatial range sr_k in line 1, and perform the subtasks in parallel through Endpoint from lines 3 to 6. Next, client collect the return results lps_k of each tRegion; if the total number is smaller than k, we expand the range of sr_k and run subtasks repeatedly until the number is not less than k (lines 2 to 8). Finally, we choose the k-most-nearest distance and its corresponding trajectory points as the final results (lines 9 to 10).

Algorithm 3. k-NN Query

Input: Trajectory number k, Query time range tr_k, Query spatial location q_k;
Output: the k-nearest neighbor trajectory points kps_k to q_k;

1 $sr_k \leftarrow estimateSpatialRange()$;
2 **repeat**
3 **for** each *tRegion* **do in parallel**
4 $ors_k \leftarrow pruning\ (q_k, tr_k, k, sr_k)$;
5 $lps_k \leftarrow distfrom(q_k, k, ors_k)$;
6 **end parallel**
7 $sr_k \leftarrow sr_k.expand()$;
8 **until** $client.sum(lps_k.size) \geq k$
9 $kps_k \leftarrow client.collect(lps_k).top(k)$;
10 **return** kps_k;

Incremental iterative: Different from spatiotemporal range query, it is difficult for k-NN query to ensure that a sufficient number of candidate trajectories are obtained with only one parallel processing, so numerous iterative processing for query is required. In the iterative process, the generated candidate trajectories and the enclosed areas do not need to participate in the next iteration, so it can be considered to reduce the computational scale by incremental iterative computation. In order to implement the incremental iterative model through Endpoint, and further reduce the communication and computing cost, we cache the Endpoint calculations by deploying an in-memory database locally (such as Redis [27]), which is helpful for obtaining parameters directly from the local to process the following iteration. Here, we cache the parameters $cids_k$ and $gcells_k$ for avoiding repeated processing in next iteration, where $cids_k$ are the MOIDs of the selected trajectories, and $gcells_k$ are the G-index grid cells which are enclosed by the previous sr_k.

7. Experiment

All experiments were conducted in a cluster consisting of 5 nodes, with one master node and 4 data nodes. Each node runs Ubuntu 16.04 LTS on 6 cores Intel Xeon E5-2620 V2 CPU, 16GB memory and 2TB disk, nodes are connected through a Gigabit Ethernet switch. THBase is implemented based on Hadoop 2.7.3, HBase 1.3.0, and Redis 3.0.

The experiment uses the global AIS data (Automatic Identification System) [28] in July 2012 as the dataset. It contains about 1.66 billion messages sent by more than 200,000 ships and is about 200 GB in size. The AIS messages in the dataset have been decoded and sorted in ascending order of time, and each AIS message can be regarded as a trajectory point.

In THBase, we set the lengths of TI and TP to 2 h and 24 h, respectively, and initialize the first-level grid size of G-index to 1°. According to the reporting interval of AIS, a single trajectory segment can contain up to 3600 trajectory points. We split the dataset into multiple chronological sets, each of which is about 40 GB, and then we insert them into HBase in turn. We test the insert efficiency and query performance when each set is appended. Since the data in the dataset is sorted in ascending order of time, the incremental data will only increase the time range and will have little influence on the density of the inserted data.

We compare to the performance of THBase with CTDM (Column-Fragmented Traditional Data Model) and SPDM (Segmented-Trajectory Partitioned Data Model) proposed in the literature [10]. CTDM stores the trajectory data of the same MO in one row of an HBase table, and organizes the in-row data in chronological order. It uses a variant of SETI, a global index structure named G-IT (Grid-mapped Interval Trees Index), to index the spatiotemporal attributes of all trajectories. SPDM divides each trajectory into trajectory segments based on a spatiotemporal partitioning mechanism, and encodes the spatiotemporal attributes of trajectories in the rowkeys to provide spatiotemporal query support. In addition, it built a reverse-sort index structure to support the MOID-based query.

Compared to THBase, the coprocessor mechanism is not appropriate for CTDM and SPDM. The reasons are as follows:

- The isolation of index and data. Both CTDM and SPDM treat index and data table as independent units, and deploy them on different nodes. Therefore, the coprocessor-based query algorithm is not conducive to the reduction of network overhead.
- The segments of the same MO are distributed on different nodes in SPDM. It is difficult to implement an efficient coprocessor-based shuffling algorithm to merger subtrajectories into whole trajectories.

7.1. Performance of Data Insertion

First, we study the performance of data insertion. Figure 4a presents the insertion performance of different schemes when the dataset size changes from 40 GB to 200 GB. A natural observation is that the insert latency of the three schemes increases linearly with the increase of the dataset size. For each scheme, the performance of THBase is between CTDM and SPDM, and CTDM takes the least time. The main reason is that compared to CTDM, THBase needs to update L-indexes by coprocessor while inserting the trajectory data, and the frequency of index update (segment-based) is higher than CTDM (MO-based). Otherwise, SPDM may require multiple serialization and deserialization processes to complete the data insert of a data cell, resulting in being the most time-consuming.

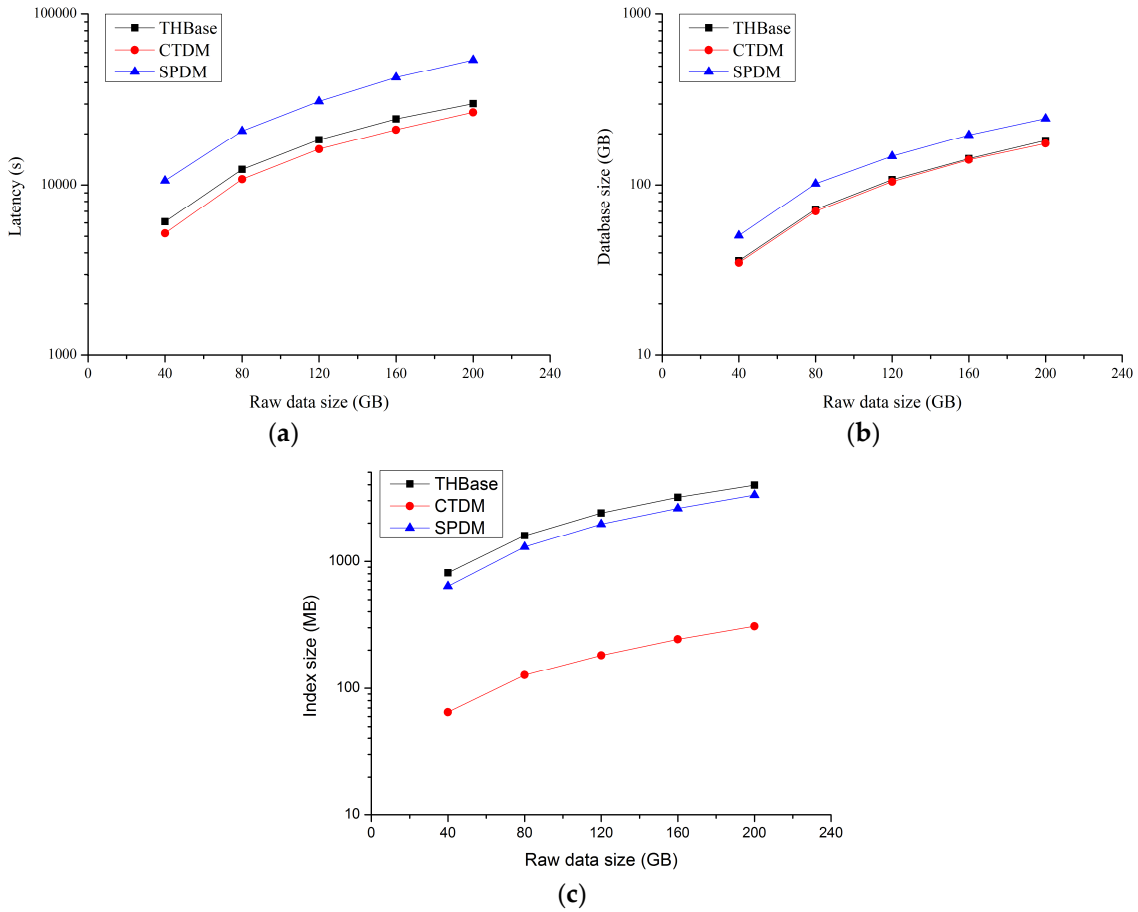

Figure 4. Time and storage cost for inserting data: (**a**) time cost, (**b**) database cost, (**c**) index cost.

Figure 4b,c show the data and index footprint of different schemes. In the aspect of storing data, CTDM has the lowest storage cost due to the MO-based data model, while THBase which has the segment-based data model consumes the larger footprint. SPDM needs to insert data into one data cell multiple times, resulting in multiple versions, so SPDM takes up the most cost. Also, in the aspect of index, the G-IT index of CTDM takes up the least cost due to its MO-based data granularity and relatively simple index entry structure. The footprint of index in THBase is larger than that in CTDM, but it only requires about 2.0% of actual storage, so the index of THBase can be completely stored in memory.

7.2. Performance of Single-Object Query

In terms of query performance, we first experiment with the single-object query. In the experiment, we randomly select 100 MOIDs and 50% of the time range as the query conditions, and use the average latency result as the evaluation criteria, as shown in Figure 5.

The experimental results show that the query latency of the three schemes gradually increases with the growth of the dataset size. This is due to the increase of the query time range. As expected, THBase works more efficiently than the two others. Benefitting from MOID and time attributes contained in rowkey structure, THBase can more accurately locate the query data range. While the rowkey structure of CTDM does not contain the time attribute, so it needs to filter all the data of the queried MO to get the final result. Compared with THBase and CTDM, WPDM supports parallel single-object query, but the data size of a single MO is not large so that the advantages of parallelism are not reflected. On the other hand, WPDM requires a larger deserialization overhead to restore each data cell to one or more trajectory points, so the efficiency of WPDM is lower than the former two.

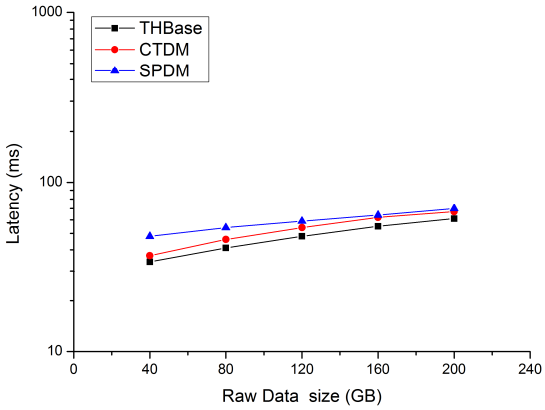

Figure 5. Performance of single-object query.

7.3. Performance of Spatiotemporal Range Query

Subsequently, we tested the spatiotemporal range query. Firstly, we randomly selected 100 spatiotemporal ranges as the query conditions when the amount of dataset are increased from 40 GB to 200 GB. The spatial range is 1%, and the time range is 24 h, the results of average latency are shown in Figure 6a. It can be seen that as the dataset size increases, the response time of CTDM increases gradually, while THBase and SPDM do not change much. This is because CTDM can only perform pruning and query based on MO, namely, it is impossible to use time conditions to exclude irrelevant trajectory data before querying HBase, and the filter overhead after accessing HBase increases with the increase of dataset. While THBase and SPDM can process pruning and query according to time condition, so the query is less affected when dataset increases and the query time range is unchanged. Among the three schemes, THBase has the highest query performance due to the following reasons: (1) all the index entries and trajectory data of the same MO coreside on same node, so the data transmission overhead in the HBase cluster can be avoided; (2) the distributed indexing scheme can better utilize the parallelism of cluster compared to the centralized index; (3) the orderly data organization for index entries and trajectory points effectively reduces the computing overhead for sorting; (4) the segment-based storage model and index model have a higher pruning precision.

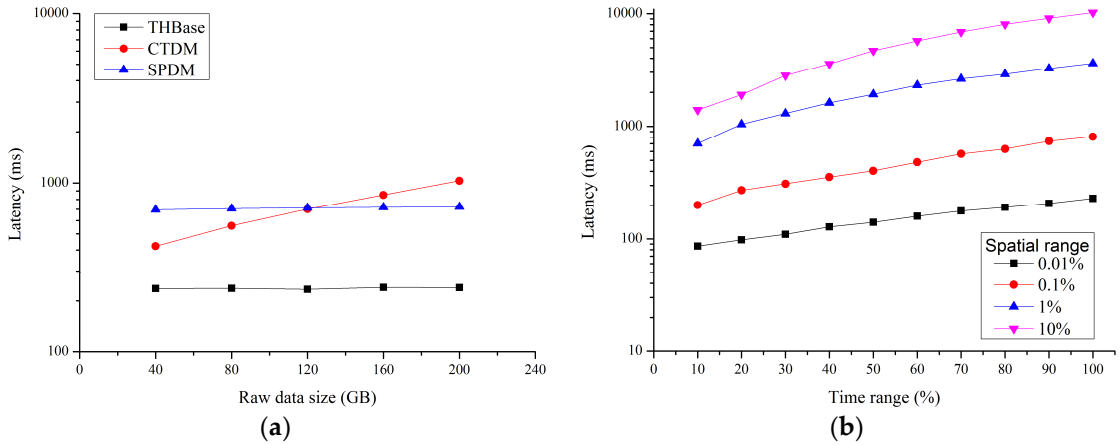

Figure 6. Performance of spatiotemporal range query: (**a**) result of changing data size, (**b**) result of changing spatiotemporal range.

In addition, we also conducted experiments on our scheme under various spatiotemporal ranges. Experiments are performed under the condition of 200 GB, the spatial range varies from 0.01% to 10%, and the time range varies from 10% to 100% (namely, from 3 days to 31 days). The result is shown in Figure 6b; as the spatiotemporal range increases, the query latency also increases. However, due to the

nonuniform spatiotemporal distribution of AIS data, the performance is not essentially linear to the change of spatiotemporal range.

7.4. Performance of k-NN Query

In the *k*-NN query experiment, we first evaluate the impact of different dataset sizes on the query performance. We randomly select 10 location points as query reference, and set the value of *k* and the query time range to 10 and 24 h, respectively. Figure 7a shows that THBase outperforms CTDM and SPDM; in addition to the reasons mentioned in Section 7.3, there are two reasons as follows: (1) For trajectory data belonging to the same object, THBase can use a certain pruning strategy to filter out more irrelevant data. (2) Incremental iteration strategy further reduces communication and computing overhead.

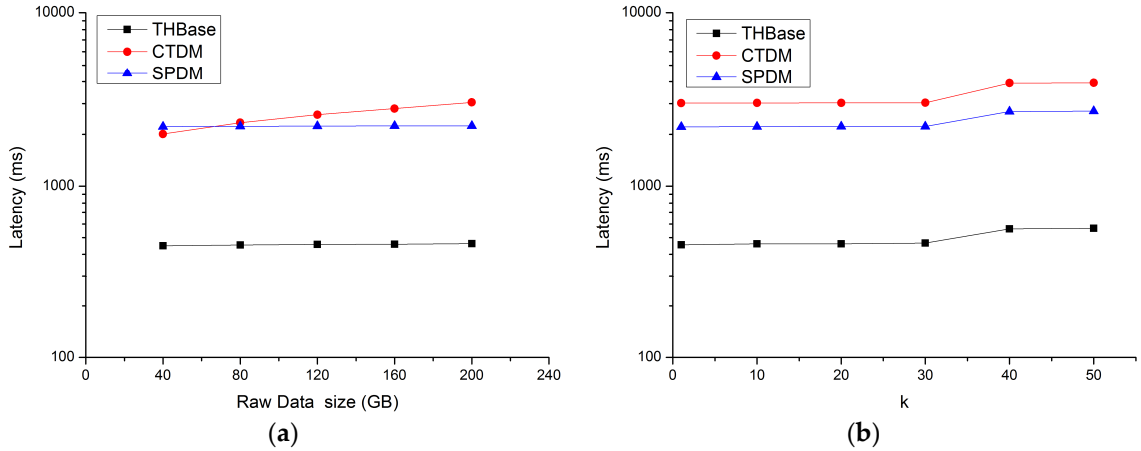

Figure 7. Performance of *k-NN* query: (**a**) result of changing data size, (**b**) result of changing *k*.

Next, we test the impact of different *k* values on queries. The dataset size is 200 GB, and the time range is still 24 h. The experimental results are shown in Figure 7b. In a certain range of *k* values, the query delay of the three schemes is not affected by the change of *k* value, This is because the candidates obtained when $k = 1$ already contain a sufficient number of trajectories to cover a sufficiently large *k* value. When the *k* value is large enough, the spatial range required to complete the query further increases, resulting in a longer query latency.

8. Conclusions

With the rapid development of positioning technology, more and more trajectory data needs to be processed. Extending the existing distributed framework to support spatiotemporal query capabilities will benefit the development of moving objects databases. In the paper, we design a trajectory data management scheme called THBase through HBase coprocessor. THBase proposes a storage and partition model suitable for trajectory data management in HBase, and implements L-index structure based on Observer coprocessor to accelerate spatiotemporal queries. Additionally, it utilizes Observer coprocessor to ensure the reasonable distribution of trajectory data and index data. On these bases, the node-locality-based parallel algorithms are proposed for processing spatiotemporal queries through Endpoint coprocessor, to realize the effective control to data transmission overhead, thus ensuring efficient query performance. We validate the storage and query performance of THBase by experiments on real dataset. Experimental results show that THBase outperforms CTDM and SPDM in terms of query. For future work, we plan to support more query types by utilizing THBase.

Author Contributions: Conceptualization, J.Q.; Data curation, J.Q.; Methodology, J.Q.; Supervision, L.M.; Validation, J.Q.; Writing—original draft, J.Q.; Writing—review & editing, L.M. and J.N. All authors have read and approved the final manuscript.

Funding: This research was funded by National Natural Science Foundation of China, grant number 61802425.

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

References

1. Salmon, L.; Ray, C. Design principles of a stream-based framework for mobility analysis. *GeoInformatica* **2017**, *21*, 237–261. [CrossRef]
2. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th symposium on Mass Storage Systems and Technologies (MSST), Lake Tahoe, NV, USA, 6–7 May 2010. [CrossRef]
3. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster Computing with working sets. In Proceedings of the Usenix Conference on Hot Topics in Cloud Computing, Boston, MA, USA, 22–25 June 2010.
4. Nishimura, S.; Das, S.; Agrawal, D.; El Abbadi, A. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In Proceedings of the 12th IEEE International Conference on Mobile Data Management (MDM), Lulea, Sweden, 6–9 June 2011. [CrossRef]
5. Cattell, R. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* **2011**, *39*, 12–27. [CrossRef]
6. You, S.; Zhang, J.; Gruenwald, L. Large-scale spatial join query processing in cloud. In Proceedings of the 2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW), Seoul, Korea, 13–17 April 2015. [CrossRef]
7. Chen, X.Y.; Zhang, C.; Ge, B.; Xiao, W.D. Efficient historical query in HBase for spatio-temporal decision support. *Int. J. Comput. Commun.* **2016**, *11*, 613–630. [CrossRef]
8. Xie, D.; Li, F.; Yao, B.; Zhou, L.; Guo, M. 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. [CrossRef]
9. Wang, H.; Zheng, K.; Zhou, X.; Sadiq, S. Sharkdb: An in-memory storage system for massive trajectory data. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Australia, 31 May–4 June 2015. [CrossRef]
10. Aydin, B.; Akkineni, V.; Angryk, R.A. Modeling and Indexing Spatiotemporal Trajectory Data in Non-Relational Databases. In *Managing Big Data in Cloud Computing Environments*; Zongmin, M., Ed.; IGI Global: Hershey, PA, USA, 2016; pp. 133–162, ISBN 1466698349.
11. Xie, X.; Mei, B.; Chen, J.; Du, X.; Jensen, C.S. Elite: An elastic infrastructure for big spatiotemporal trajectories. *VLDB J.* **2016**, *25*, 473–493. [CrossRef]
12. Zhang, Z.; Jin, C.; Mao, J.; Yang, X.; Zhou, A. TrajSpark: A scalable and efficient in-memory management system for big trajectory data. In Proceedings of the Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data, Beijing, China, 7–9 July 2017. [CrossRef]
13. Ding, X.; Chen, L.; Gao, Y.; Jensen, C.S.; Bao, H. UlTraMan: A unified platform for big trajectory data management and analytics. *VLDB J.* **2018**, *11*, 787–799. [CrossRef]
14. Ke, S.; Gong, J.; Li, S.; Zhu, Q.; Liu, X.; Zhang, Y. A hybrid spatio-temporal data indexing method for trajectory databases. *Sensors* **2014**, *14*, 12990–13005. [CrossRef] [PubMed]
15. Shang, Z.; Li, G.; Bao, Z. DITA: A distributed in-memory trajectory analytics system. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018. [CrossRef]
16. Apache HBase. Available online: https://hbase.apache.org/ (accessed on 4 November 2018).
17. Roussopoulos, N.; Kelley, S.; Vincent, F. Nearest neighbor queries. *SIGMOD Rec.* **1995**, *24*, 71–79. [CrossRef]
18. Vora, M.N. Hadoop-HBase for large-scale data. In Proceedings of the 2011 International Conference on Computer Science and Network Technology (ICCSNT), Harbin, China, 24–26 December 2011. [CrossRef]
19. Vashishtha, H.; Stroulia, E. Enhancing query support in HBase via an extended coprocessors framework. In Proceedings of the European Conference on a Service-Based Internet, Poznan, Poland, 26–28 October 2011. [CrossRef]
20. Zhou, X.; Zhang, X.; Wang, Y.; Li, R.; Wang, S. Efficient distributed multi-dimensional index for big data management. In Proceedings of the International Conference on Web-Age Information Management, Beidaihe, China, 14–16 June 2013. [CrossRef]

21. Ma, Y.; Rao, J.; Hu, W.; Meng, X.; Han, X.; Zhang, Y.; Liu, C. An efficient index for massive IOT data in cloud environment. In Proceedings of the 21st ACM International Conference on Information and Knowledge Management, Maui, HI, USA, 29 October–2 November 2012. [CrossRef]
22. Bao, J.; Li, R.; Yi, X.; Zheng, Y. Managing massive trajectories on the cloud. In Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Burlingame, CA, USA, 31 October–3 November 2016. [CrossRef]
23. D'silva, J.V.; Ruiz-Carrillo, R.; Yu, C.; Ahmad, M.Y.; Kemme, B. Secondary indexing techniques for key-value stores: Two rings to rule them all. In Proceedings of the 20th International Conference on Extending Database Technology and 20th International Conference on Database Theory 2017 Joint Conference, Venice, Italy, 21–24 March 2017.
24. Samet, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv.* **1984**, *16*, 187–260. [CrossRef]
25. Xia, Y.; Chen, J.; Lu, X.; Wang, C.; Xu, C. Big traffic data processing framework for intelligent monitoring and recording systems. *Neurocomputing* **2016**, *181*, 139–146. [CrossRef]
26. Xie, D.; Li, F.; Phillips, J.M. Distributed trajectory similarity search. *Proc. VLDB Endow.* **2017**, *10*, 1478–1489. [CrossRef]
27. Han, J.; Haihong, E.; Le, G.; Du, J. Survey on NoSQL database. In Proceedings of the Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on, Port Elizabeth, South Africa, 26–28 October 2011. [CrossRef]
28. Harati-Mokhtari, A.; Wall, A.; Brooks, P.; Wang, J. Automatic identification system (AIS): Data reliability and human error implications. *J. Navig.* **2007**, *60*, 373–389. [CrossRef]