International Journal of Digital Earth

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tjde20

Optimizing segmented trajectory data storage with HBase for improved spatio-temporal query efficiency

Yi Bao, Zhou Huang, Xuri Gong, Yuyang Zhang, Ganmin Yin & Han Wang

To cite this article: Yi Bao, Zhou Huang, Xuri Gong, Yuyang Zhang, Ganmin Yin & Han Wang (2023) Optimizing segmented trajectory data storage with HBase for improved spatio-temporal query efficiency, International Journal of Digital Earth, 16:1, 1124-1143, DOI: 10.1080/17538947.2023.2192979

To link to this article: https://doi.org/10.1080/17538947.2023.2192979

INTERNATIONAL JOURNAL OF DIGITAL EARTH
2023, VOL. 16, NO. 1, 1124–1143
https://doi.org/10.1080/17538947.2023.2192979

Optimizing segmented trajectory data storage with HBase for improved spatio-temporal query efficiency

Yi Bao[a,b], Zhou Huang [a,b], Xuri Gong[a,b], Yuyang Zhang[a,b], Ganmin Yin[a,b] and Han Wang[a,b]

[a]Institute of Remote Sensing and Geographical Information Systems, Peking University, Beijing, People's Republic of China; [b]Beijing Key Lab of Spatial Information Integration & Its Applications, Peking University, Beijing, People's Republic of China

ABSTRACT

The surging accumulation of trajectory data has yielded invaluable insights into urban systems, but it has also presented challenges for data storage and management systems. In response, specialized storage systems based on non-relational databases have been developed to support large data quantities in distributed approaches. However, these systems often utilize storage by point or storage by trajectory methods, both of which have drawbacks. In this study, we evaluate the effectiveness of segmented trajectory data storage with HBase optimizations for spatio-temporal queries. We develop a prototype system that includes trajectory segmentation, serialization, and spatio-temporal indexing and apply it to taxi trajectory data in Beijing. Our findings indicate that the segmented system provides enhanced query speed and reduced memory usage compared to the Geomesa system.

ARTICLE HISTORY
Received 19 December 2022
Accepted 14 March 2023

KEYWORDS
Trajectory storage; HBase; trajectory segmentation; spatio-temporal query

1. Introduction

With the rapid development of mobile Internet, Internet of Things (IoT) technology and wireless infrastructures, trajectory data has been in an explosive growth in recent years. Trajectory data is a type of data recording the movement of people or objects at specific movements by storing sequences of points with locational attributes and timestamps (Zheng 2015). It has been widely implemented in commercial services, traffic monitoring, and administration applications and brought essential insights for further understanding urban systems (Liu et al. 2012; Wang et al. 2021).

Despite bringing richer mobility information, the surge in the trajectory data amount causes difficulties. For instance, DiDi, a Chinese taxi corporation, generates and collects trajectory data reaching 100TB per day (Yang, He, and Phoebe Chen 2018). The gigantic quantity of trajectory data and its continuous growth momentum pose substantial challenges for data storage and management systems, which further entail the necessity for tools with more robust storage capabilities and management efficiencies (Zimányi et al. 2019). Storage systems designed explicitly for trajectory data, such as Geomesa (Hughes et al. 2015), TrajMesa (Li, He, Wang, Ruan et al. 2020), and JUST (Li, He, Wang, Huang et al. 2020) have been developed to support storage and queries with massive data sizes. These systems are primarily based on non-relational databases (NoSQL)

CONTACT Zhou Huang huangzhou@pku.edu.cn

such as HBase (Vora 2011) and MongoDB (Chodorow 2013), which sacrificed the ACID characteristics of relational databases to achieve the capabilities for handling larger data quantities in distributed approaches. These systems adopt spatial filling curves (Liu et al. 2014) as the Z curve or extended Z curve (XZ) (BÖxhm, Klump, and Kriegel 1999) for geohash coding to record the geographic location of points and trajectories.

Current trajectory data storage systems often apply two approaches: storage by point or storage by trajectory. Storage by point refers to building a spatio-temporal index on each point of a trajectory. When a query is made, each point of the trajectory is retrieved from the database and merged into the original trajectory. On the other hand, storage by trajectory refers to building a spatio-temporal index on the entire trajectory whose points are serialized into a sequence in the Value of the database. Despite having simple algorithms, each of these two methods has a drawback. In spite of its high spatial efficiency, storage by point is inefficient for data compression. Conversely, the storage by trajectory method has better compression efficiency but low query efficiency due to the large size of the entire trajectory's minimum bounding box (MBB). Segmented trajectory storage has been proposed to balance the deficiencies between the two approaches. The concept of segmented storage means splitting the trajectory into segmentations as basic units in the database. Existing studies have investigated the optimization of the segmentation processes with algorithms such as dynamic programming and greedy (Chakka, Everspaugh, and Patel 2003; Hadjieleftheriou et al. 2002; Rasetic et al. 2005). However, such trajectory segment optimizations are presently only proposed in single-machine trajectory storage systems and have yet to be implemented in NoSQL trajectory data storage and distributed systems. With the exponential increase of trajectory storage data, single-machine systems are inadequate for accommodating the storage and retrieval demands of such massive data amount of trajectory data. Hence, constructing a NoSQL-based prototype for segmented trajectory storage becomes imperative.

In this study, we investigate the efficacy of segmented trajectory data storage with HBase optimizations for trajectory spatio-temporal queries. Figure 1 demonstrates the logical flow and a brief description of each component of this study. We develop a prototype of distributed segmented trajectory data storage system that consists of trajectory segmentation, serialization, and spatio-temporal index with the capability of performing ID query, spatio-temporal query, OD(Origin to Destination) query, as well as similar trajectory query. We apply the developed segmented trajectory data storage system to 100,000 taxi trajectories containing 864 million points in Beijing, and conduct several experiments to evaluate the efficiency of the optimization approaches. Furthermore, we compare the segmented system's query speed and memory usage to the Geomesa system (Hughes et al. 2015), which can only store trajectory data through storage by point.

2. Related work

2.1. Trajectory segmentation

In the field of trajectory mining, trajectory segmentation is commonly used to split long trajectories into shorter ones based on factors such as speed, activity, and dwell points. This allows for the identification of different activities or movements within the trajectory. For example, Lee, Han, and Whang (2007) utilized the MDL (Minimum Description Length) principle to segment the trajectories before trajectory clustering for anomaly detection purposes. Alternatively, Zhou and Huang (2008) borrowed concepts of natural language processing, treating trajectories as sentences, and applied EM (Expectation Maximization) algorithm for effective segmentation in trajectory similarity calculation and classification. Additionally, Guo et al. (2018) employed a data segmentation method based on probabilistic logic to analyze vehicle driving habits and extract parking points from trajectory data sequences. Buchin et al. (2011) also proposed algorithms for quickly splitting trajectories into the minimum number of segments based on attributes such as position, heading, velocity, and curvature.

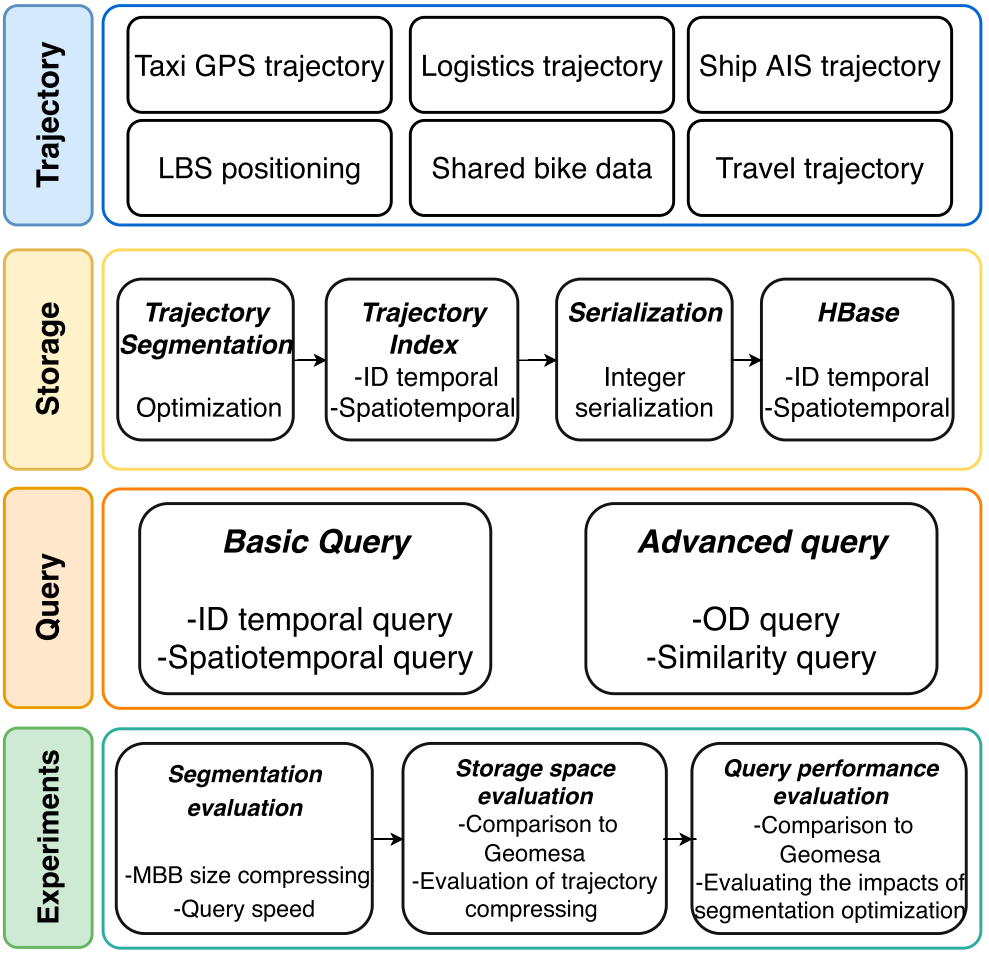

Figure 1. The framework of the study with brief descriptions of the components.

In the context of trajectory storage, segmentation is often used to optimize data compression and query efficiency. Hadjieleftheriou et al. (2002) proposed that the trajectory segments should be optimized as much as possible to minimize the sum of the generated trajectory segment volumes to reduce the candidate trajectory segments in spatio-temporal queries and improve efficiency. Concretely, they proposed the DPSplit and MergeSplit algorithms for segmentation, with the former utilizing dynamic programming for optimal segmentation and the latter employing a greedy approach to gradually merge adjacent segments. Rasetic et al. (2005) further defined trajectory segmentation optimization as minimizing the number of disks I/Os required for spatio-temporal queries and proposed an optimal algorithm based on dynamic programming. TrajStore is a large-scale trajectory storage system that utilizes adjacent storage of trajectory segments and a series of compressions to minimize storage space occupation (Cudre-Mauroux, Wu, and Madden 2010).

2.2. Trajectory query and storage

In the realm of trajectory query and storage, several relational database-based trajectory storage systems have been developed. Pelekis et al. (2015) built the trajectory database system HERMES on the object-relational database system Oracle, which supports a query language similar to SQL and can

support queries such as trajectory similarity and KNN. Zimányi et al. (2019) based on PostgreSQL/PostGIS, extended the built-in GiST and SP-GiST indexes and query optimizer, and thus can support 3D point (tgeogpoint) and 4D point (tgeompoint) SQL query functions. GCOTraj, developed by Yang, He, and Phoebe Chen (2018) utilized space-filling curves to sort data blocks and the Graph-Based Ordering technique for optimization, resulting in improved query performance. Memory-based trajectory storage systems, such as SharkDB (Zheng et al. 2018), have also been developed to enhance query speed. SharkDB arranges all points at a given moment together, enabling efficient compression, reducing memory I/O bandwidth, increasing CPU cache hit rates, and simplifying parallel computations on multicore CPUs.

As the volume of trajectory data increases exponentially, a single machine's storage and computing capabilities become limiting factors (Huang et al. 2011; Chen et al. 2010). As a result, distributed trajectory storage solutions have become a popular research area (Huang et al. 2017; Wan, Huang, and Peng 2016). MDHBase is an earlier spatio-temporal data storage system based on the distributed database HBase (Nishimura et al. 2011). MDHBase leverages space-filling curves to transform high-dimensional point data into one-dimensional values stored in the database. With the horizontal scalability of HBase, the query and storage performance of MDHBase is significantly superior to that of a stand-alone system. Since the development of MDHBase, various distributed NoSQL database-based trajectory management systems have been proposed (Zhang et al. 2014; Whitby, Fecher, and Bennight 2017; Van Le and Takasu 2018; Qin, Ma, and Niu 2019; Li, He, Wang, Huang et al. 2020). Among those systems, Geomesa is one of the most widely used open-source spatio-temporal engines (Hughes et al. 2015). Utilizing NoSQL databases such as HBase, Bigtable and Accumulo as the storage backend, Geomesa implements the OGC (Open Geospatial Consortium) standard interface to support the storage of point, line, polygon and spatio-temporal data. Additionally, TrajMesa, a system based on Geomesa specifically applied to manage trajectory data, is also developed (Li, He, Wang, Ruan et al. 2020). TrajMesa supports point and trajectory storage as well as ID, spatio-temporal and trajectory similarity queries.

In this research, we focus on addressing the shortage of studies on trajectory segmentation in distributed NoSQL environments. We introduce a solution for trajectory segments storage in such environments and test various optimization methods for improved query efficiency, reduced storage space. A prototype system is built using the HBase database and evaluated with actual data, showing that the proposed solution outperforms existing trajectory data storage systems in terms of storage space and query efficiency.

3. Methods and materials

3.1. Trajectory storage and query

Figure 2 presents the system's architecture for storing and querying trajectory segments. The system is divided into five major parts, each described in detail below.

(1) Segmentation of the input trajectory using the MergeSplit algorithm: This step involves partitioning the input trajectory into smaller segments, allowing for efficient storage and querying.
(2) Construction of a trajectory data index and serialization of the trajectory segments: The index is built to enable fast lookup of trajectory segments, and their corresponding serialized byte arrays.
(3) Design and implementation of HBase tables for storing the indexed trajectory data: The HBase tables are designed to store the indexed trajectory data in a distributed manner, allowing for scalable storage and computation.
(4) Key-based trajectory query and optimization: The system provides an interface for users to query the entire trajectory while the underlying storage and indexing are performed segmentally. This allows for efficient querying of large trajectories.

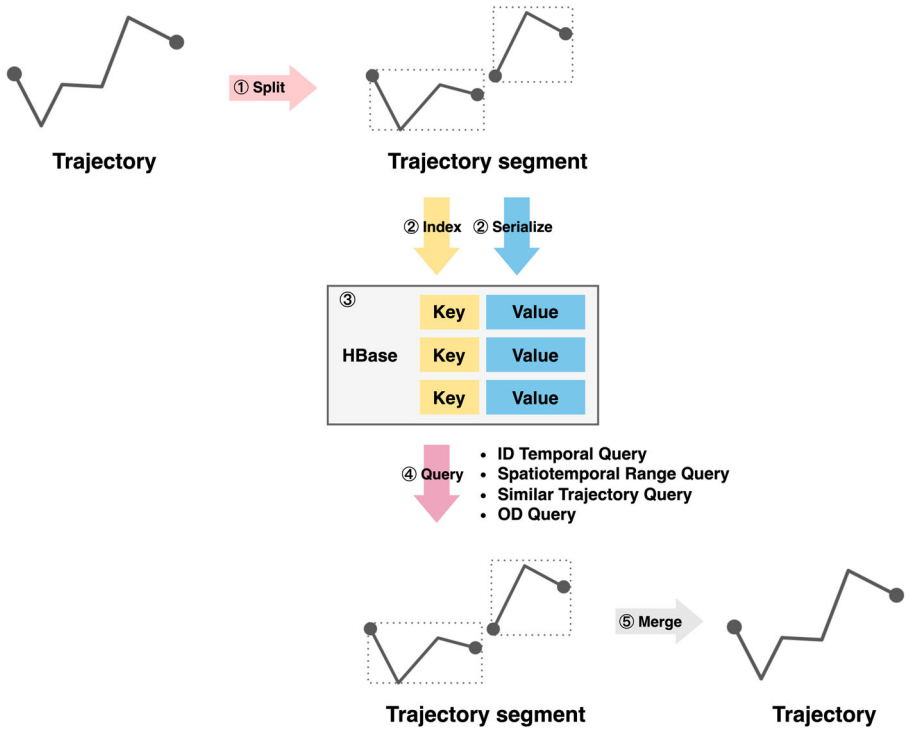

Figure 2. The processes of segmented trajectory data storage and query.

(5) Distributed storage and computation of the segmented trajectory data: The system employs a distributed architecture to store and compute the segmented trajectory data, ensuring scalability and high performance.

Overall, the system provides a seamless interface for users to write and query entire trajectories while the underlying storage and indexing are performed segmentally. This makes the system easy to use and efficiently handles large trajectory datasets.

3.1.1. Trajectory segmentation

The trajectory segmentation algorithm takes in a complete original trajectory and outputs multiple segments of it. The rules of implementing trajectory segmentation could highly vary from each other but the quality of the segments ensued are substantially affected by the difference among the splitting approaches. The longer the trajectory segment, the better the data compression effect but with larger bounding boxes and lower spatial-temporal query efficiency. On the other hand, shorter trajectory segments have smaller bounding boxes and higher spatial-temporal query efficiency but weaker data compression ability. Therefore, an appropriate segmentation rule is needed to balance the trade-off between compression efficiency and query efficiency, ensuring that the segments have greater lengths while the bounding box sizes remain under a reasonable scale. In this research, the process is carried out in two main steps: 1. breaking the trajectory at points of outlier or missing data to generate sub-trajectories, and 2. utilizing the greedy MergeSplit method (Hadjieleftheriou et al. 2002) to obtain the desired trajectory segments. The workflow is illustrated in Algorithm 1.

Algorithm 1: Trajectory segmentation Algorithm

```
Data: Trajectory
Result: Segments
SubTrajs ⟵ ∅;
n ← length of Trajectory;
last ⟵ 0
for i in range(1, n) do
    if Disance(point(i), point(i − 1) > dist_max or
     Interval(point(i), point(i − 1)) > interval_max then
        Add Traj[last, i] into SubTrajs;
        last ⟵ i + 1;
    end
end
Add Trajectory[last, n] into SubTrajs;
Segments ⟵ ∅;
for subTraj in SubTrajs do
    Segments.add(MergeSplit(subTraj));
end
return Segments;
```

During the splitting process, we apply a filtering method to the points of each trajectory based on their spatial and temporal distances from other points to improve storage efficiency. There are two underlying reasons for filtering. Firstly, there is the potential for 'deviant points' to arise, which are unreasonably distant from the overall trajectory due to positioning errors in the data. Secondly, there is the possibility of data loss over a long period, where a point may have a timestamp that is too far from the previous point in the sequence. Both of these scenarios can lead to inaccuracies and biases in the data and can ultimately reduce the storage efficiency by increasing the MBB size. Therefore, we have implemented a filtering process that removes deviant points by comparing the distances of each point to its neighboring points, using a maximum distance threshold. Additionally, point pairs with excessively long temporal lags are divided into two parts and treated as independent trajectories.

The next step is to segment the obtained trajectory after removing the deviation points. The optimization goal of trajectory segmentation is to minimize the size of the bounding box of the trajectory segments after segmentation. Currently, the most commonly used methods are the DPSplit and MergeSplit proposed by Hadjieleftheriou et al. (2002). The former uses dynamic programming to optimize segmentation with a time complexity of $O(n^2l)$, while the latter employs a greedy approach to gradually merge adjacent trajectory segments with a time complexity of $O(nlogn)$. Although MergeSplit cannot guarantee the optimal solution, its actual performance is close to the optimality with relatively lower time complexity (Hadjieleftheriou et al. 2002), making it a favorable choice for trajectory segmentation in this research. The MergeSplit algorithm involves two steps: 'split' and 'merge'. Firstly, a trajectory is divided into lines with their MBB sizes calculated. Following that, these lines are merged back into a limited number of trajectory segments, as defined by the user's input. During the merge process, the algorithm greedily selects the existing two sub-trajectories that have the smallest MBB size if merged to form a new trajectory segment. The merge process eventually terminates when the number of trajectory segments reaches its limit.

3.1.2. Trajectory index and serialization construction

The design of a trajectory index must satisfy the query requirements. ID temporal queries and spatio-temporal range queries queries represent the basic query capabilities in trajectory storage tasks. In this study, we have developed ID temporal and spatio-temporal indexes for the segmented trajectory storage system.

The ID temporal index is defined in Formula (1). The *ID* represents the identifier of the trajectory, while the *StartTime* indicates the initial timestamp value of a trajectory segment. The time interval in which *StartTime* is located is expressed by *TimeBin*. In other words, the *TimeBin* equals $\lfloor StartTime/TimeBinLength \rfloor$, where the user defines the length of the time intervals

(*TimeBinLength*) according to the data distribution. The inclusion of *StartTime* is essential for guaranteeing the uniqueness of the index as different trajectory segments of the same trajectory can have the same *TimeBin*. When a query with trajectory *ID*, start time (st), and end time (et) is made, the system scans the database in a range of Keys shown in Formula (2). To mitigate the risks of data skew of queries where the queried trajectories' *IDs* are concentrated in a few ranges, we implemented the HashMap algorithm on the *ID* temporal index (*Hash*(*ID*)) to randomly disperse the distributions of *ID*.

$$Hash(ID) :: TimeBin :: StartTime \tag{1}$$

$$[ID :: \lfloor st/TimeBinLength \rfloor - 1, ID :: \lfloor et/TimeBinLength \rfloor] \tag{2}$$

The spatio-temporal index is defined in Formula (3). The *TimeBin*, *ID*, and *StartTime* have the same meanings as in the ID temporal index Formula 1. The XZ refers to the geohash value of the trajectory segmentations, which is the space-filling curve calculated from the extended Z ordering curve. Similar to the HashMap function, we implemented the sharding technique (*Shard*), which adds a random single-byte shard before the Key to avoid data skews.

$$Shard :: TimeBin :: XZ :: ID :: StartTime \tag{3}$$

In addition to implementing indexes for the trajectory segments, we further realize the trajectory serialization. Moreover, we make further optimizations in storage space usage for coordinates and timestamps. The unoptimized serialized data is in the sequential form of $[(x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_n, y_n, t_n)]$. The x and y represent the latitude and longitude coordinates, declared double types that take up 8 bytes. The t represents the timestamp, declared as a *long* type that takes up 8 bytes. Therefore, the total memory consumption of a trajectory segment with n points is 24n bytes. Depending on the characteristics of the trajectory segment, we can perform specific optimizations during serialization to reduce storage occupation. As a primary concern, excessive precision is unnecessary when storing latitude and longitude, as GPS positioning is unable to achieve an accuracy of seven decimal places. To optimize the coordinate space usage, we changed the data type of latitude and longitude from *long* to *int*, which saved half of the space usage as *int* only takes up 4 bytes.

On the other hand, we adopted serialization of increments for the temporal attributes. As timestamps of a trajectory are strictly increasing, it is plausible to store the increment of each timestamp rather than the actual values because of the previously asserted time interval length limit, which can be stored in 2 bytes. The timestamp of the start point remains unchanged at 8 bytes. Therefore, the total memory consumption of the integer serialization timestamps is 2n + 6 bytes for a trajectory segment with n points.

Conclusively, our optimization changes the original space usage of 24n bytes to 10n + 6 bytes for a trajectory segment with n points, saving more than half of the memory consumption. We name this optimization method '*integer off setserialization*' The integer means setting data types as *int*, while offset refers to the practices of storing increments of timestamps. Figure 3 illustrates the summary of storage usage before and after the optimization of integer offset serialization

Length (Bytes)	4n				4n				2n + 6			
Type	*Int*	*Int*	*Int*	*Int*	*Int*	*Int*	*Int*	*Int*	*Long*	*Short*	*Short*	*Short*
Data	X_1	X_2	...	X_n	Y_1	Y_2	...	Y_n	t_1	Δt_2	...	Δt_n

Figure 3. The summarization of storage usage after optimization of integer offset serialization.

3.1.3. Structural design of HBase

For each trajectory segmentation, the ID temporal and spatio-temporal index generates the Key. The Value is generated by serialization. The subsequential Key-Value pairs are then stored in two HBase tables for the two index types (Figures 4 and 5). The spatio-temporal index table keeps the segment extent for optimizing spatio-temporal range queries.

3.1.4. Trajectory query and optimization

The segmented trajectory storage system can carry out several queries: ID temporal query, spatio-temporal query, OD query, and similar trajectory query.

The ID temporal query refers to the query of a trajectory with a given trajectory ID and time range. The ID temporal index only stores the start time of the trajectory segment, and upon querying, it first searches for the trajectory segments with start times within [t_start, t_end], assuming they are ordered in time as s1, s2, s3. Upon successful completion of the query, it additionally examines the preceding trajectory segment of s1, if it exists, which shall be included in the output. To clarify, the system will only return the trajectory within the interval, not the entire trajectory.

In spatio-temporal queries, trajectories within the given spatio-temporal ranges are retrieved. As with ID temporal query, the inputs of the spatio-temporal query are transformed into several Key ranges. Likewise, the results returned from a spatio-temporal query are within the given query range rather than the entire trajectory. When the queries are being conducted, the *Multi Row Range Filter* function in HBase is implemented, which improves the query efficiency by merging the related queries. As the spatial filling curve's precision is insufficient, many retrieved trajectories cannot intersect with the queried spatial ranges (Figure 6). We solve this problem by utilizing the segment extents in the spatio-temporal index table to determine if the spatial query ranges intersect with the trajectory segments and invalid query trajectories are discarded. In this case, the optimization is termed '*spatial filter optimization*'.

The OD query refers to the query of trajectories that pass through the origin and destination spatial ranges under a certain period given by the user. Algorithm 2 depicts the process of the OD query in the system. First, the one with smaller spatio-temporal ranges between the origin (O) and destiny (D) is selected. A spatio-temporal query is then performed to retrieve the intersected trajectories. When the number of retrieved exceeds a certain threshold(k), the system directly

ID Temporal Index KEY	VALUE
Hash(ID) :: ID :: Timebin :: StartTime	Segment Points
Key1	*list[STPoint]*
Key2	*list[STPoint]*
Key3	*list[STPoint]*
Key4	*list[STPoint]*

Figure 4. The HBase structure for ID temporal index.

Spatiotemporal Index KEY	VALUE	
Shard :: Timebin :: XZ :: ID :: StartTime	Segment Points	Segment Extent
Key1	*list[STPoint]*	$x_{min}, y_{min}, t_{min}, x_{max}, y_{max}, t_{max}$
Key2	*list[STPoint]*	$x_{min}, y_{min}, t_{min}, x_{max}, y_{max}, t_{max}$
Key3	*list[STPoint]*	$x_{min}, y_{min}, t_{min}, x_{max}, y_{max}, t_{max}$
...	*list[STPoint]*	$x_{min}, y_{min}, t_{min}, x_{max}, y_{max}, t_{max}$

Figure 5. The HBase structure for spatio-temporal index.

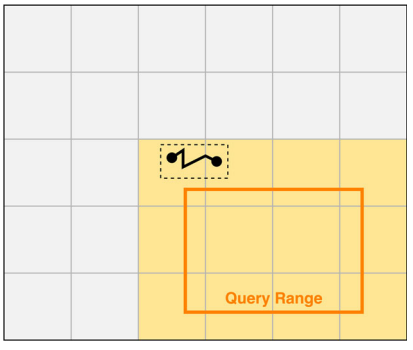

Figure 6. The visualization of spatial filtering by segment extent.

performs a spatio-temporal query on another range and intersects it with the previously retrieved trajectories. Conversely, the system applies ID temporal query on the retrieved trajectories and judges whether these trajectories intersect with the other spatio-temporal range when the number of retrieved trajectories is less than k. The aim of setting the different query strategies based on the number of candidate trajectories is to improve query efficiency. The rationale for such a design is based on the fact that the query speed of the ID temporal query is faster than the spatio-temporal query. If the number of candidate trajectories retrieved by the spatio-temporal query on the smaller spatio-temporal domain is below a reasonable size, it is plausible to apply the ID temporal query to check if they intersect with the other spatio-temporal domain. However, if the total number of initial candidate trajectories is too large, applying ID temporal trajectory queries to all of them would be cumbersome. Under these circumstances, applying the spatio-temporal query to the other spatio-temporal domain and finding the intersections would be more advantageous because the expected number of query operations would be lower.

Algorithm 2: The OD query algorithm

```
   Data: Two spatio-temporal ranges O and D
   Result: The ID of trajectories that travel through O and D
 1 E1 ← The one with smaller spatio-temporal range between O and D;
 2 E2 ← The one with larger spatio-temporal range between O and D;
 3 CandidateIDs ← spatio-temporal_query(E1);
 4 if CandidateIDs.length ≥ k then
 5 |   E2IDs ← spatio-temporal_query(E2);
 6 |   ResultSet ⟵ CandidateIDs∩ E2IDs;
 7 end
 8 else
 9 |   ResultSet ⟵ ∅;
10 |   for Candidate in CandidateIDs do
11 |   |   if ID_temporal_query(Candidate) ∩ E2 = True then
12 |   |   |   ResultSet.add(Candidate);
13 |   |   end
14 |   end
15 end
16 return ResultSet;
```

The similarity query refers to the process of finding the most similar trajectories to a given trajectory in the database. This type of query is beneficial in mining information on co-travellers, such as passengers in the same taxi. Our segmented storage system implements the temporally weighted similarity algorithm proposed by Gong et al. (2020). The algorithm is outlined in Algorithm 3. When a user inputs the target trajectory, it is divided into segments for querying nearby candidate trajectory segments. The similarity is calculated by summing the degrees of similarity between the target and candidate trajectory segments with the same ID. The similarity score is output for all candidate trajectories. Further selection based on the score can be performed later to get similar trajectories with a given threshold or top-k ranking.

Algorithm 3: The similarity trajectory query algorithm

```
Data: Target_trajectory
Result: The trajectories that are most similar to the target trajectory
Segments ← split(Target_trajectory);
for Segment in Segments do
    Proximate_trajectory ← spatio-temporal_query(Segment);
    K1 ← Key(Proximate_trajectory);
    K2 ← Key(Segment);
    Similarity_scores[K1][K2] ← Similarity_score(Proximate_trajectory, Segment);
end
Similarity_trajectories ← Similarity_scores.group_by(Trajectory_ID).sum();
return Similarity_trajectories;
```

3.1.5. Distributed system implementation

The utilization of the Spark NewHadoopRDD API for reading trajectory segments from Hadoop can greatly improve the performance of data analysis in a distributed manner. By leveraging the MapReduce framework, Spark is able to effectively read and process the data from the nearest node, taking advantage of the parallel capabilities of multiple nodes to accelerate the reading and calculation processes. This reduces the time required for data analysis and avoids the need for extensive network communication transmission between nodes, further optimizing the system's performance. Furthermore, the ability of Spark to adapt to the specific structure and division of the data tables allows for a more flexible and efficient data analysis process. By adapting to the data's specific characteristics, Spark can optimize the reading and calculation processes, leading to improved performance and accuracy of the analysis.

Taking the function of generating a statistical heat map as an example, this system utilizes Spark's distributed reading capabilities to obtain trajectory segments data within a specified time period, dividing the data spatial extent into uniform 50-meter grids and calculating the number of trajectory segments within each grid. The overall process is as follows:

(a) Based on the time period to be plotted, the time slot range is calculated and the Key query range is generated, resulting in the construction of a Scan.
(b) Spark's NewAPIHadoopRDD is utilized to construct an RDD based on the Scan.
(c) The map function is utilized to deserialize the byte arrays in the RDD, transforming them into trajectory segments.
(d) Interpolation of the trajectory segments is performed in order to obtain denser trajectory sampling points (optional).
(e) The flatmap function is employed to calculate the grid in which each point of the trajectory segment belongs, transforming it into a grid ID.
(f) The countByKey function is utilized to count the number of points within each grid.
(g) The grid is plotted based on the number of points contained within it.

3.2. Experiment design

This study uses GPS trajectories from Beijing cabs over 92 days from May 1 to July 23, 2013. The data has been anonymized to protect privacy. The original dataset includes 26.4 million trajectories and 2.8 billion points, with an average sampling interval of 73.9 seconds. Given the sparsity of the data, with a sampling interval that is longer than typical real-world applications such as GPS (which typically has a sampling interval of less than fewer seconds), we resampled the first 100,000 trajectories in chronological order. After resampling, the sampling interval is reduced to 10 seconds, resulting in 846 million points. In this experiment, we used the full trajectory data for comparison with the Geomesa storage performance, and the resampled trajectory data for other experiments.

Figure 7. Example heat maps of the trajectory data for different time periods. (a) 4:00–5:00. (b) 9:00–10:00. (c) 14:00–15:00. (d) 19:00–20:00.

Figure 7 illustrates the heat map of the trajectory data for the different time periods we utilize in this experiment.

The system introduced in this paper is implemented using Scala, a programming language known for its concurrency and functional programming capabilities. After establishing the segmented trajectory storage system, we stored the Beijing taxi trajectory data and conducted several experiments. We used Spark to read and compute the trajectory data from the HBase of the system. The Spark NewHadoopRDD API allows for distributed reading of data from Hadoop and the creation of RDDs for concurrent computing.

Table 1. The softwares used in this experiment.

Software	Version
Java	1.8.0_281
Hadoop	3.2.2
HBase	2.2.6
Spark	3.1.1
Geomesa	3.0.1

Table 2. The default parameters in this experiment.

Default parameters	Value
Segment Length (Average Points per Segment)	100
Compression	LZ4
Data Block Coding	FAST DIFF
Serialization	Integer offset serialization
Spatio-temporal Query Range	1 h&1 km
ID temporal Query Range	5 h

The experiments of this study are conducted on a group of three service nodes and one user interface(UI) node. Each service node is a virtual machine with 16 cores, 32GB RAM, and 1800GB hard drives. These nodes communicated with one another using 1000MB bandwidth networks. A summary of the environment settings is presented in Table 1.

During the experiments, we apply to write and query operations and tested the system performance with different compression methods, query ranges, and division lengths. For each condition, we vary one attribute while keeping others at their default values. The default parameters are summarized in Table 2. After evaluating the effect of several optimization methods, we used Geomesa and our system to conduct the same operations for comparisons.

4. Results

4.1. The effects of integer offset serialization

We implement several strategies in Section 3.1.2 of our study to optimize space usage. These strategies include adjusting the coordinate data type to *int* and using sequential increments to store timestamps. Our experiments with varying segment lengths demonstrated the efficacy of these approaches, resulting in a significant reduction in memory consumption(as depicted in Figure 8).

Figure 8(a) displays the effects of varying trajectory segment lengths on storage space utilization. As the length of each segment increases, the amount of storage space required for the trajectory decreases, as does the space occupied by integer offset serialization. This is due to the fact that

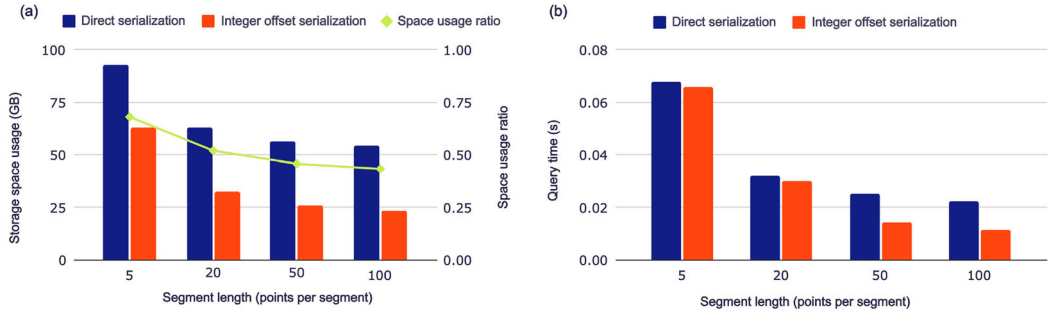

Figure 8. The comparison of direct serialization and integer offset serialization in the consumption of storage space (a) and ID temporal query time (b).

more trajectory points are stored together, allowing for more efficient compression. Additionally, as each segment requires storing a Key and other information in HBase, shorter segments result in more segments and need more additional space. If the trajectory is stored point by point, the maximum amount of space is occupied, while storing the entire trajectory results in the minimum amount of space. These findings demonstrate the potential benefits of utilizing integer offset serialization and longer trajectory segments for optimizing storage space utilization.

The integer offset serialization strategy significantly reduces ID temporal query time, as shown in Figure 8(b). The bottleneck of HBase queries often lies in disk reading and network IO time, and serialization optimization reduces the amount of data read and transmitted, resulting in faster query times. Furthermore, longer segment lengths result in faster ID temporal queries. On the one hand, with a fixed-length time window, longer segment lengths result in fewer segments and, therefore, fewer rows to scan during a query. On the other hand, longer segment lengths provide better compression and hence fewer data to be transferred, leading to shorter query times.

4.2. The effects of spatial filter optimization

Spatial filter optimization, which incorporates segment extents for intersection judgments (as discussed in Section 3.1.4), effectively filters out invalid trajectories during spatio-temporal queries. Figure 9 demonstrates the effects of this optimization on query time in a 1km*1h window with varying segment lengths. In all tested conditions, spatial filter optimization reduces query time by over 50%, indicating its effectiveness in improving query efficiency. The ability of spatial filter optimization to filter out invalid trajectory segments based on the bounding boxes of HBase nodes reduces the amount of unnecessary data transfer, thus improving query efficiency.

4.3. The effects of segmentation optimization

Segmentation optimization, using the greedy merging method (as mentioned in Section 3.1.1), aims to minimize the MBB sizes of trajectory segments while maintaining a fixed number of segments. Such optimization can effectively reduce the size of MBB, especially when the segment length is low, as shown in Figure 10.

In this study, we investigate the impact of segmentation optimization on the performance of ID temporal and spatio-temporal queries. Our results show hardly any differences between scenarios with or without segmentation optimization for ID temporal query, as shown in Figure 11(a). This is

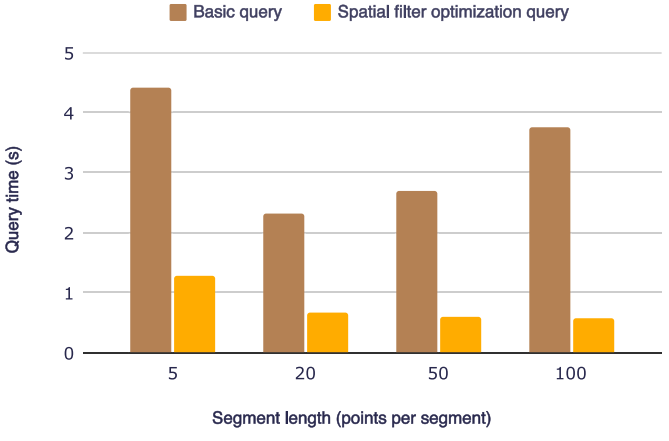

Figure 9. The comparison between basic query and spatial filter optimization query.

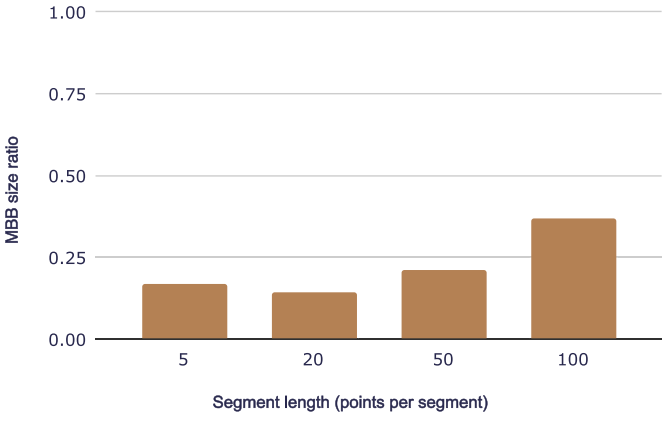

Figure 10. The MBB size ratio of segmentation optimization in different segment lengths.

in line with our expectations, as ID temporal queries are aspatial and therefore do not depend on the manner in which segmentation is conducted.

However, our results also show that segmentation optimization can generally reduce the time of spatio-temporal queries and improve query efficiency. This is evident in Figure 11(b), which shows the system's performance with and without segmentation optimization. While the system with segmentation optimization activated does have slightly worse speed performance when the segment length is at its lowest value, the overall improvement in query efficiency more than compensates for this minor trade-off.

4.4. Advanced queries and distributed calculation performance

We further evaluate the impact of trajectory segmentation on the efficiency of advanced queries such as OD and similar trajectory queries. In the experiment, different trajectory segment lengths are set for the OD query, as shown in Figure 12(a). The OD query is mainly composed of spatio-temporal range queries, so its query time is consistent with the trend of spatio-temporal range queries: within a reasonable range of trajectory segment lengths, the longer the average trajectory segment, the shorter the query time and the higher the query performance. In addition, to evaluate the performance of similarity trajectory query under different trajectory segment lengths, we adopt different trajectory segment lengths to perform similarity trajectory query, as shown in Figure 12(b). The results show that the time consumption of similarity trajectory query is much larger compared

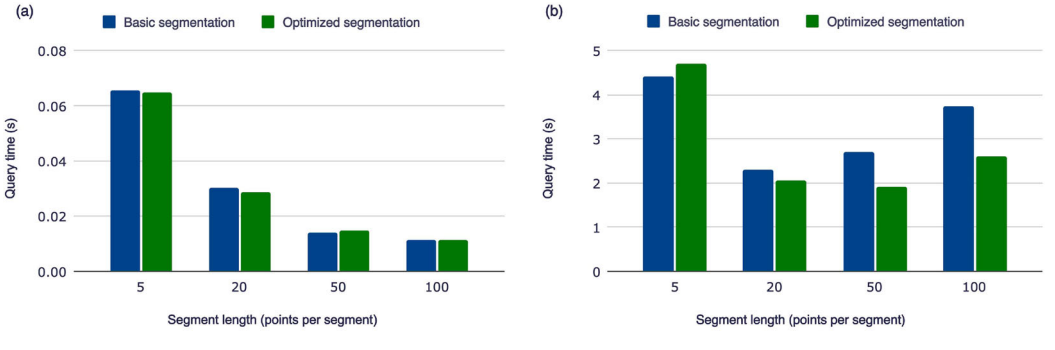

Figure 11. The comparison between basic segmentation and optimized segmentation in ID temporal query (a) and spatio-temporal query (b) with different segment lengths.

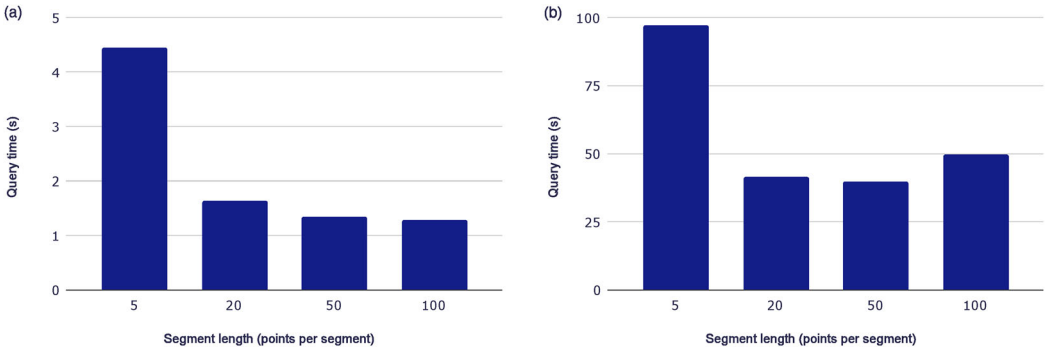

Figure 12. The performance of OD query (a) and trajectory similarity query (b).

to spatio-temporal range query and ID temporal query. The target trajectory is segmented into multiple trajectory segments, resulting in multiple spatio-temporal range queries and consequently a longer time for the similarity trajectory query process.

We validate the advantages of utilizing Spark for distributed trajectory computation by allocating the daily trajectory points into 50-meter grid units, tallying the quantity of trajectory points per hour within each unit, and yielding a graphical representation (illustrated in Figure 7). The statistics indicate that the single-machine client requires 1962.8 seconds to execute the spatio-temporal query calculations, whereas the distributed query procedure requires only 625.5 seconds. The distributed method significantly reduces data loading time and increases the speed of queries. The system, integrated with Spark's distributed computing system, effectively processes large amounts of data and efficiently produces detailed statistical charts.

4.5. Comparison to Geomesa

After conducting a thorough analysis and verification of the effects of integer offset serialization, spatial filter optimization, and segmentation optimization on the trajectory data storage system, we compare our approach to the Geomesa. The results show that our distributed trajectory data storage system outperform Geomesa in both query time and memory consumption.

Among the four HBase data compression approaches, the system with integer offset serialization consistently exhibits a lower space usage, as shown in Figure 13. In addition, our system

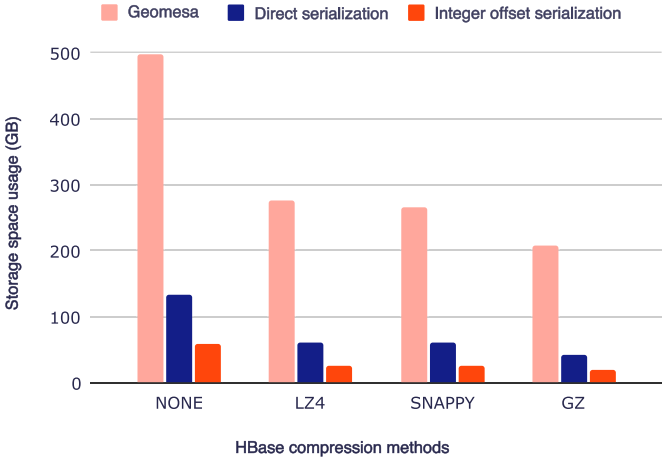

Figure 13. The space usage under different compression methods for Geomesa, direct serialization, and integer offset serialization.

demonstrates superior query speed for ID temporal queries, with a savings of approximately 70% in query time, as shown in Figure 14.

Figure 15 compares the spatio-temporal range query performance of our system to Geomesa. For basic queries, the average query time of our system is longer than that of Geomesa due to the higher indexing efficiency for points in the spatial filling curve and the presence of a certain spatio-temporal range for trajectory segments, which leads to a larger number of ineffective trajectory segments being queried. This is a drawback of segment-based and trajectory-based storage. However, when the query range is larger and data volume are larger, the impact of data transfer

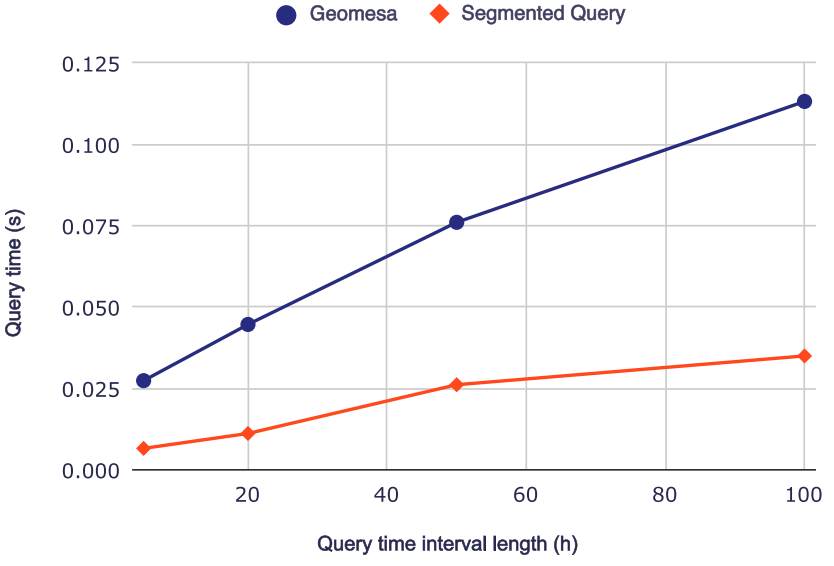

Figure 14. The comparison of the ID temporal query performance of Geomesa and proposed system under different interval length.

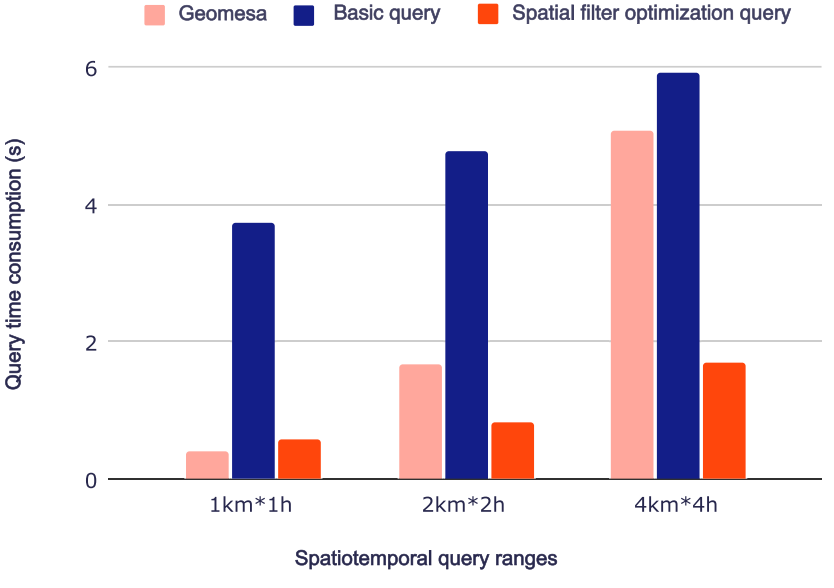

Figure 15. The comparison of the spatio-temporal query performance of Geomesa and proposed system under different spatio-temporal ranges.

on query time is larger and the impact of ineffective trajectory segments on query diminishes. As a result, the difference between our system and Geomesa's point-based storage decreases. For queries with filter optimization, when the query range is larger, the query time of our system is shorter than Geomesa's because filter optimization reduces the number of ineffective trajectory segments in the query and data compression reduces the data volume, resulting in a lower overall query time and improved query efficiency.

Overall, our approach to optimizing the trajectory data storage system demonstrates superior performance in both space usage and query speed compared to Geomesa. The implementation of integer offset serialization, spatial filter optimization, and segmentation optimization greatly improve the efficiency of the system and provided valuable insights into the optimization of trajectory data storage systems.

5. Conclusion and discussions

The trajectory storage and management, as the basis of computation and mining, can provide the function of trajectory data storage and basic query, and plays an important role in various fields of trajectory data application. We have investigated the trajectory segments storage method and its associated optimization techniques in a distributed NoSQL database, covering aspects such as segmentation, indexing, serialization, query optimization, and calculation. Based on the HBase database, we implement the prototype system of segmented trajectory storage, verify the effect of various designs and optimizations with real data, and compare it with Geomesa to show the advantages and characteristics of the segmented storage model.

The research achievements and contributions of this paper can be summarized as follows:

- Distributed trajectory segmentation and storage method. With the explosive growth of trajectory data, its data scale exceeds the management ability of a single-machine storage system. Although relevant trajectory distributed storage systems have been developed, these systems are based on point or trajectory storage. In this research, we propose a scheme of using trajectory segments storage in distributed storage environment; that is, when writing a trajectory, the trajectory is segmented into shorter trajectory segments and stored in the database as trajectory segments, and when querying, the trajectory segments are spliced to generate the whole trajectory. By segmented storage, we can avoid the point model consuming a large amount of storage space, the trajectory model index query efficiency is low, and the query performance is unstable.
- The optimization design of the trajectory segments storage system. To improve the query performance of the trajectory segments storage system, we design several optimization methods, including the optimization of trajectory segmentation, indexing, serialization, query processing and calculation. Through the optimization design, we can improve the query performance, reduce the storage space and improve the efficiency of the system.
- The prototype implementation and performance evaluation of the trajectory segments storage system. Based on the HBase database, we implement the trajectory segments storage system's prototype system and used actual data to verify the performance of various designs and optimization methods. The experimental results show that the proposed system performs better in storage space, query efficiency and stability than the existing system.

Based on the experiments and analysis discussed above, the main findings of this study can be summarized as follows:

- Using serialization compression during the storage of trajectory segments can reduce storage space and the amount of data transmitted during queries, resulting in improved query speeds.
- Filtering optimization based on bounding boxes during spatial and temporal range queries can effectively filter out invalid trajectories, significantly enhancing the performance of such queries.

- Optimizing trajectories' segmentation to minimize the volume of bounding boxes can improve the efficiency of spatial and temporal range queries, with noteworthy improvements observed in basic spatial and temporal queries.
- In distributed storage environments, trajectory segmentation models constitute an effective method of organizing and storing data, with optimized implementations offering superior storage and query speed performance.

While this study has achieved notable success, there are still opportunities for improvement and further research. One key area for future work is the development of more advanced algorithms for trajectory segmentation. Current approaches focus on optimizing for MBB, but additional factors such as segment length, spatial range, and temporal duration could also be considered. The effects of these factors on storage and querying performance would need to be evaluated to determine the optimal segmentation strategy. To further investigate the system's performance, a scalability experiment could be conducted in future work to show the execution speed as a function of the number of clients and/or size of the data. In addition, future work could aim to automatically determine segmentation parameters based on the characteristics of the trajectory, such as speed and sampling intervals, as well as query window size.

Another area for improvement is implementing a range of spatial indexes for trajectory segments and comparing their performance. The current system uses the XZ curve for spatial indexing, but alternative approaches, such as XZ3 indexing and grid-based indexing, could also be considered. This would provide insight into the most effective indexing methods for trajectory data.

Expanding the range of trajectory calculation and analysis algorithms available within the system can enable more sophisticated analyzes. The current system includes capabilities for similarity calculation, clustering, and compression, but additional algorithms could be implemented in future work. This could include algorithms for trajectory simplification, matching, and prediction, among others.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

We acknowledge the financial support from the National Natural Science Foundation of China (42271471, 42201454, 41830645), the International Research Center of Big Data for Sustainable Development Goals (CBAS2022GSP06). We also appreciate the detailed comments from the Editor and the anonymous reviewers.

ORCID

Zhou Huang http://orcid.org/0000-0002-1255-1913

References

BÖxhm Christian, Gerald Klump, and Hans-Peter Kriegel. 1999. "Xz-Ordering: A Space-Filling Curve for Objects with Spatial Extension." In *International Symposium on Spatial Databases*, 75–90. Springer.

Buchin Maike, Anne Driemel, Marc Van Kreveld, and Vera Sacristán. 2011. "Segmenting Trajectories: A Framework and Algorithms Using Spatiotemporal Criteria." *Journal of Spatial Information Science*2011 (3): 33–63. doi:10.5311/JOSIS.2011.3.66.

Chakka V. Prasad, Adam Everspaugh, and Jignesh M. Patel. 2003. "Indexing Large Trajectory Data Sets with SETI." In *CIDR*, Vol. 75, 76. Citeseer.

Chen, Bin, Fengru Huang, Yu Fang, Zhou Huang, and Hui Lin. 2010. "An approach for heterogeneous and loosely coupled geospatial data distributed computing." *Computers & Geosciences* 36 (7): 839–847. doi:10.1016/j.cageo.2010.01.002.

Chodorow Kristina. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc.
Cudre-Mauroux Philippe, Eugene Wu, and Samuel Madden. 2010. "Trajstore: An Adaptive Storage System for Very Large Trajectory Data Sets." In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, 109–120. IEEE.
Gong Xuri, Zhou Huang, Yaoli Wang, Lun Wu, and Yu Liu. 2020. "High-Performance Spatiotemporal Trajectory Matching Across Heterogeneous Data Sources." *Future Generation Computer Systems* 105: 148–161. doi:10.1016/j.future.2019.11.027.
Guo Sini, Xiang Li, Wai-Ki Ching, Ralescu Dan, Wai-Keung Li, and Zhiwen Zhang. 2018. "GPS Trajectory Data Segmentation Based on Probabilistic Logic." *International Journal of Approximate Reasoning* 103: 227–247. doi:10.1016/j.ijar.2018.09.008.
Hadjieleftheriou Marios, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. 2002. "Efficient Indexing of Spatiotemporal Objects." In *International Conference on Extending Database Technology*, 251–268. Springer.
Huang Zhou, Yiran Chen, Lin Wan, and Xia Peng. 2017. "GeoSpark SQL: An Effective Framework Enabling Spatial Queries on Spark." *ISPRS International Journal of Geo-Information* 6 (9): 285. doi:10.3390/ijgi6090285.
Huang, Zhou, Yu Fang, Bin Chen, Lun Wu, and Mao Pan. 2011. "Building the distributed geographic SQL workflow in the Grid environment." *International Journal of Geographical Information Science* 25 (7): 1117–1145. doi:10.1080/13658816.2010.515947.
Hughes James N., Andrew Annex, Christopher N. Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. 2015. "Geomesa: A Distributed Architecture for Spatio-Temporal Fusion." In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, Vol. 9473, 94730F. International Society for Optics and Photonics.
Lee Jae-Gil, Jiawei Han, and Kyu-Young Whang. 2007. "Trajectory Clustering: A Partition-and-Group Framework." In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 593–604. Beijing, China.
Li Ruiyuan, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. "Just Jd Urban Spatio-Temporal Data Engine." In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 1558–1569. IEEE.
Li Ruiyuan, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. "Trajmesa: A Distributed Nosql Storage Engine for Big Trajectory Data." In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2002–2005. IEEE.
Liu Yu, Chaogui Kang, Song Gao, Yu Xiao, and Yuan Tian. 2012. "Understanding Intra-Urban Trip Patterns From Taxi Trajectory Data." *Journal of Geographical Systems* 14 (4): 463–483. doi:10.1007/s10109-012-0166-z.
Liu Jiajun, Haoran Li, Yong Gao, Hao Yu, and Dan Jiang. 2014. "A Geohash-Based Index for Spatial Data Management in Distributed Memory." In *2014 22nd International Conference on Geoinformatics*, 1–4. IEEE.
Nishimura Shoji, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. "Md-hbase: A Scalable Multi-Dimensional Data Infrastructure for Location Aware Services." In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1, 7–16. IEEE.
Pelekis Nikos, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. 2015. "HERMES: A Trajectory DB Engine for Mobility-Centric Applications." *International Journal of Knowledge-Based Organizations (IJKBO)* 5 (2): 19–41. doi:10.4018/IJKBO.
Qin Jiwei, Liangli Ma, and Jinghua Niu. 2019. "THBase: A Coprocessor-Based Scheme for Big Trajectory Data Management." *Future Internet* 11 (1): 10. doi:10.3390/fi11010010.
Rasetic Slobodan, Sander Jörg, Elding James, and Nascimento Mario A. 2005. "A Trajectory Splitting Model for Efficient Spatio-temporal Indexing." In *Proceedings of the 31st International Conference on Very Large Data Bases,* 934-945. VLDB Endowment.
Van Le Hong, and Atsuhiro Takasu. 2018. "G-HBase: A High Performance Geographical Database Based on HBase." *IEICE TRANSACTIONS on Information and Systems* 101 (4): 1053–1065. doi:10.1587/transinf.2017DAP0017.
Vora Mehul Nalin. 2011. "Hadoop-HBase for Large-Scale Data." In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1, 601–605. IEEE.
Wan Lin, Zhou Huang, and Xia Peng. 2016. "An Effective NoSQL-Based Vector Map Tile Management Approach." *ISPRS International Journal of Geo-Information* 5 (11): 215. doi:10.3390/ijgi5110215.
Wang Sheng, Zhifeng Bao, J. Shane Culpepper, and Gao Cong. 2021. "A Survey on Trajectory Data Management, Analytics, and Learning." *ACM Computing Surveys (CSUR)* 54 (2): 1–36. doi:10.1145/3440207.
Whitby Michael A., Rich Fecher, and Chris Bennight. 2017. "Geowave: Utilizing Distributed Key-Value Stores for Multidimensional Data." In *International Symposium on Spatial and Temporal Databases*, 105–122. Springer.
Yang Shengxun, Zhen He, and Yi-Ping Phoebe Chen. 2018. "GCOTraj: A Storage Approach for Historical Trajectory Data Sets Using Grid Cells Ordering." *Information Sciences* 459: 1–19. doi:10.1016/j.ins.2018.04.087.
Zhang Ningyu, Guozhou Zheng, Huajun Chen, Jiaoyan Chen, and Xi Chen. 2014. "Hbasespatial: A Scalable Spatial Data Storage Based on hbase." In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, 644–651. IEEE.
Zheng Yu. 2015. "Trajectory Data Mining: An Overview." *ACM Transactions on Intelligent Systems and Technology (TIST)* 6 (3): 1–41. doi:10.1145/2743025.

Zheng Bolong, Haozhou Wang, Kai Zheng, Han Su, Kuien Liu, and Shuo Shang. 2018. "Sharkdb: An in-memory Column-Oriented Storage for Trajectory Analysis." *World Wide Web* 21 (2): 455–485. doi:10.1007/s11280-017-0466-9.

Zhou Yue, and Thomas S. Huang. 2008. "'Bag of Segments' for Motion Trajectory Analysis." In *2008 15th IEEE International Conference on Image Processing*, 757–760. IEEE.

Zimányi Esteban, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. 2019. "Mobilitydb: A Mainstream Moving Object Database System." In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*, 206–209.