The VLDB Journal (2021) 30:287–310
https://doi.org/10.1007/s00778-021-00652-x

REGULAR PAPER

Dragoon: a hybrid and efficient big trajectory management system for offline and online analytics

Ziquan Fang[1] · Lu Chen[1] · Yunjun Gao[1,2] · Lu Pan[1] · Christian S. Jensen[3]

Received: 4 June 2020 / Revised: 7 January 2021 / Accepted: 14 January 2021 / Published online: 3 February 2021

Abstract
With the explosive use of GPS-enabled devices, increasingly massive volumes of trajectory data capturing the movements of people and vehicles are becoming available, which is useful in many application areas, such as transportation, traffic management, and location-based services. As a result, many trajectory data management and analytic systems have emerged that target either offline or online settings. However, some applications call for both offline and online analyses. For example, in traffic management scenarios, offline analyses of historical trajectory data can be used for traffic planning purposes, while online analyses of streaming trajectories can be adopted for congestion monitoring purposes. Existing trajectory-based systems tend to perform offline and online trajectory analysis separately, which is inefficient. In this paper, we propose a hybrid and efficient framework, called *Dragoon*, based on Spark, to support both offline and online big trajectory management and analytics. The framework features a mutable resilient distributed dataset model, including RDD Share, RDD Update, and RDD Mirror, which enables hybrid storage of historical and streaming trajectories. It also contains a real-time partitioner capable of efficiently distributing trajectory data and supporting both offline and online analyses. Therefore, Dragoon provides a hybrid analysis pipeline. Support for several typical trajectory queries and mining tasks demonstrates the flexibility of Dragoon. An extensive experimental study using both real and synthetic trajectory datasets shows that Dragoon (1) has similar offline trajectory query performance with the state-of-the-art system UlTraMan; (2) decreases up to doubled storage overhead compared with UlTraMan during trajectory editing; (3) achieves at least 40% improvement of scalability compared with popular streaming processing frameworks (i.e., Flink and Spark Streaming); and (4) offers an average doubled performance improvement for online trajectory data analytics.

Keywords Trajectory system · Data management · Data analytics · Distributed processing

1 Introduction

With the wide availability of GPS-enabled devices and increasing use of mobile computing services, massive trajectory data from various moving objects (e.g., people, vehicles, and animals) are continuously growing at a high speed [48]. Trajectory data with its analytics may benefit many real-life applications, including urban computing [58], transportation [45], animal behavior studies [31], and security [23], to name but a few. As an example, DiDi, the largest ride-sharing company in China, is collecting more than 106 TB trajectory data daily in order to provide services such as route planning, travel time estimation, and urban capacity analyses [6]. As a result, researchers have devoted their efforts to trajectory analyses, which led to a variety of trajectory data management and analytic systems have emerged [14,17,20,47], which can be categorized as offline and online settings.

✉ Yunjun Gao
gaoyj@zju.edu.cn

Ziquan Fang
zqfang@zju.edu.cn

Lu Chen
luchen@zju.edu.cn

Lu Pan
panlu96@zju.edu.cn

Christian S. Jensen
csj@cs.aau.dk

1 College of Computer Science, Zhejiang University, Hangzhou, China

2 Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China

3 Department of Computer Science, Aalborg University, Aalborg, Denmark

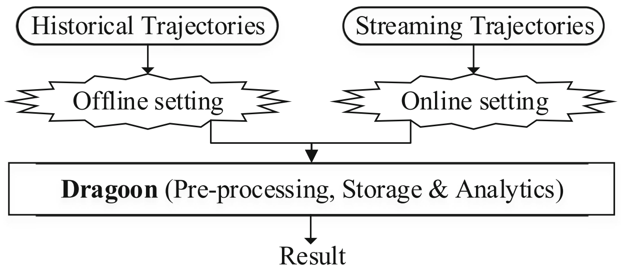

Fig. 1 Hybrid trajectory data analysis pipeline

The offline systems [20,47] focus on the management and analysis of historical trajectory data, while the online systems [14,17] aim at the real-time processing of streaming trajectories [37].

However, some real-life applications need both offline and online trajectory analyses. Consider a real-world traffic management application, where the online processing of streaming trajectories generated by urban taxis can be adopted for real-time congestion monitoring [55], while the offline analyses of big historical trajectory data can be used for traffic planning purposes, such as traffic signal optimization [51] and region function characterization [53]. Unfortunately, existing trajectory management systems offer sub-optimal support as relying on more than one system to achieve both offline and online analyses, which forces users to constantly switch between offline and online systems to accomplish hybrid analysis scenarios mentioned above. Moreover, it is also inefficient because (i) it needs to repeat a similar trajectory preprocessing process when *extracting*, *transforming*, and *loading* (ETL) historical versus streaming trajectory data and (ii) it incurs disk storage duplication for different offline or online scenarios. Consequently, we aim to develop an efficient hybrid trajectory management and analytic system.

In this paper, we present *Dragoon*, a new hybrid and efficient big trajectory management system for offline and online analytics. Dragoon is designed to be a holistic solution, as illustrated in Fig. 1, which supports the full pipeline of both historical and streaming trajectory data preprocessing, management, and analytics in a single system. There are two approaches that we have considered to develop such a system. The first is to extend an existing offline or batch-oriented system (e.g., Spark [54]) to enhance its online real-time processing capabilities for streaming trajectories. The second is to extend an existing online or streaming-oriented system (e.g., Flink [3]), equipping it with the ability to manage and process large-scale historical trajectory data. Nonetheless, we observe that the online systems or frameworks have limitations related to their handling of data updates [50], which cannot be ignored when designing such a trajectory management system. More specifically, the data update mechanisms of online systems are stream-driven and passive, meaning that data updates are triggered only when new data arrives, while no updates occur without any incoming data. In contrast, many real-world applications that require trajectory editing [49] or trajectory cleaning [22] call for data updates even when there is no incoming trajectory data. To this end, we adopt the first approach and leave the second one as a future research direction. Specifically, we choose to extend the Spark platform to achieve the Dragoon system, as Spark is a popularly used and efficient offline distributed processing system in both academic and industry fields. Note that Spark Streaming also extends Spark to process streaming data. However, it can be only used to process streaming trajectory in a batch way, i.e., it takes micro-batch strategy that cuts trajectory streams into several data RDDs and manages those RDDs separately. In contrast, we aim to develop a single system to manage both historical and streaming trajectories in a hybrid fashion by enhancing the storage layer of the Spark core, based on which it enables better hybrid trajectory analytics. Two key challenges exist as follows when adopting the first approach.

Challenge I: How to extend Spark to support dynamic and unified storage of streaming trajectory data? Existing Spark Streaming stores streaming data by cutting data streams into data batches, where each data batch can be viewed as a resilient distributed dataset (RDD) [54]. However, such RDDs are immutable and read-only, and any update (e.g., map or union operations) on an RDD will create a new data RDD. In the settings of streaming trajectories or trajectory data editing scenarios, data updates occur frequently. Hence, we aim to provide a dynamic storage that can support efficient data updates, which can avoid unnecessary data copies and time costs. To address this challenge, we propose a mutable RDD model, termed as mRDD, which includes RDD Share, RDD Update, and RDD Mirror. Specifically, the RDD Share detects those unchanged parts that can be shared in an RDD; the RDD Update offers three update strategies for different update related scenarios; and the RDD Mirror supports effective read and write access controls, thus avoiding data conflicts and inconsistencies caused by concurrent reads and writes in the distributed environment. The mRDD model is designed to be compatible with the original RDD model of Spark, so that Dragoon enables to store both historical and streaming trajectory data in the mRDDs via a hybrid manner. Dragoon also provides flexible real-time partitioners. Specifically, the partitioners enable cutting historical and streaming trajectories to multiple data partitions for parallel processing based on different trajectory characteristics (e.g., the id, spatial, or temporal information of trajectories) while considering data balance of the system. Meanwhile, historical and streaming trajectories in the same data partition are merged together to achieve unified and efficient storage.

Challenge II: How to build hybrid processing pipeline for both offline and online trajectory data analytics? The techniques used for offline and online trajectory analyses are different. The offline techniques focus on the management and batch processing of historical trajectory data, while the online techniques aim at the real-time processing of trajectory streams. In view of this, Dragoon provides a hybrid data analytic pipeline based on its hybrid storage. Besides, the offline analysis cases that contain trajectory editing and ID/Range/*k* nearest neighbor (*k*NN) queries as well as the online analysis cases that include online ID/Range/*k*NN queries and the real-time co-movement pattern detection are implemented and supported by Dragoon.

Based on the proposed mRDD model, Dragoon is able to serve as a scalable, efficient, and flexible hybrid platform for both offline and online big trajectory data management and analytics. To sum up, the contributions of this paper are summarized as follows.

- We propose a hybrid and efficient system for integrated offline and online scalable trajectory data management and analytics.
- We offer a mutable RDD model with real-time partitioners to manage historical and streaming trajectories in a hybrid fashion on underlying storage.
- We design a hybrid analysis pipeline for both historical and streaming trajectory data, and demonstrate its utility by showing and supporting typical offline and online trajectory queries and mining tasks.
- Extensive experiments on both real and synthetic trajectory datasets offer insights into the efficiency and scalability of Dragoon, and also include comparisons with existing state-of-the-art trajectory processing systems or frameworks.

The rest of the paper is organized as follows. Section 2 reviews the related work, and Sect. 3 covers the preliminaries. Section 4 presents an overview of the Dragoon system. Sections 5 and 6 detail storage and analytic techniques, respectively. Section 7 presents trajectory case studies that demonstrate the system's support for hybrid analyses. The experimental results based on real and synthetic data sets are reported in Sect. 8. Finally, Sect. 9 concludes the paper and offers research directions.

2 Related work

In this section, We proceed to survey briefly related system architectures, and clarify the differences between them and Dragoon by the following.

2.1 Offline/online trajectory analytic systems

To begin with, offline trajectory systems enable batch processing of historical trajectory data and come in centralized and distributed variants. Prominent centralized systems for trajectory storage or analysis include BerlinMOD [21], TrajStore [17], and SharkDB [42], to name just a few. However, due to the limitation of a single machine's capability, they cannot scale to massive volumes of trajectory data management and analytics.

Thanks to the distributed MapReduce [18] framework and its open-source implementations Hadoop [1] and Spark [4], several distributed trajectory management or analytic systems exist. CloST [39], which is Hadoop-based, provides distributed query processing for big trajectory data. LocationSpark [40], Simba [47], and Elite [48], which are Spark-based, utilize specific partitioning and index strategies to manage big trajectory data using innovation storage scheme. Several distributed big trajectory data analytic systems also exist. Shang et al. [38] propose the DITA, which is also Spark-based to support top-*k* trajectory similarity join. Xie et al. [46] present a Spark-based query framework for distributed trajectory similarity search. Yuan et al. [52] proceed to study the trajectory similarity search and join on the Spark platform while considering the road networks. In addition, cloud and other schema-based (e.g., distributed NoSQL) trajectory systems [10,28,29,36] that offer distributed storage and processing also exist.

More recently, an efficient platform called UlTraMan [20] has proposed that offers a unified pipeline for big trajectory data ETL, storage, management, and analytics. In that way, users can finish various offline trajectory data analysis in a single system with provided operation interfaces. The aforementioned systems are all designed to support offline management and analytics for static and historical trajectories, and thus fail to support real-time processing of streaming trajectories. The reason is that the streaming trajectory data is unbounded and arrives in real time, while the storage and processing engines of all the above trajectory systems are designed for static historical data. In light of this, our proposed Dragoon adopts a flexible framework for both offline and online trajectory data management and analytics. Note that Dragoon's offline component shares all the features of state-of-the-art system UlTraMan including scalable, efficient, and unified, since the offline techniques used in Dragoon are similar to that of UlTraMan. The main difference is that we propose the mRDD model in Dragoon, which is compatible with the original RDD of Spark but enabling efficient data updates in the underlying storage.

As real-time trajectory data analytics [14] is increasingly important in real-life applications (e.g., real-time event detection [35]), online trajectory systems are being developed for streaming trajectory processing. Existing online trajectory data systems aim to efficiently store [10], query [9,30,34, 43], and mine [14,32] trajectory streams. Nonetheless, they address the storage, querying, and mining tasks separately, and there is still no unified system like Dragoon that supports the full pipeline of big streaming trajectory data management and analytics. With the recent advent of distributed stream processing, several general distributed stream processing platforms [2–5,16,25] are proposed. However, they do not fully exploit the characterizes of trajectory data. In contrast, Dragoon fully considers the characteristics of the trajectories, such as Id, spatial, and temporal information, to support efficient and dynamic management for trajectory streams. Although trajectory architectures based on those platforms for streaming trajectory analytics have been studied [56,57], those online trajectory systems focus on the real-time processing of the latest trajectory streams, which are not efficient for the large-scale, offline trajectory data management and analytics due to the passive update limitations as we have discussed in Sect. 1. In contrast, we proposed the Dragoon system that is based on the mRDD model extended from Spark, which can support both online streaming trajectory data management and offline trajectory data management (e.g., trajectory data editing).

2.2 General-purpose hybrid approaches

A hybrid system is able to process both historical and streaming data in a single system, and several general-purpose hybrid systems exist. Kumar et al. [26] extend MapReduce online to support batch data processing, but still faces the limitation of passive updates as mentioned in Sect. 1. With the emergence of the Lambda architecture [24], several hybrid systems [7,8,11,15,19,50] are developed that combine two platforms to process both historical and streaming data. For instance, the Summingbird [11] combines Hadoop and Storm, where Hadoop is used for offline processing, and Storm is used for online processing. The idea of Lambda-based systems is simple, but they must maintain two separate systems or different operator APIs to support hybrid analytics, while we aim to develop a single system to support both offline and online data management and analytics. In other words, we target a unified system to support both historical and streaming trajectory data analysis. Further, none of the above systems target trajectory data processing. To the best of our knowledge, Dragoon is the first proposal that targets hybrid trajectory data management and analytics, and that includes a detailed description of its implementation.

Although Spark may process data in a hybrid fashion with its extensions Spark Streaming or Structured Streaming, those systems do not support efficient and scalable hybrid trajectory management and analytics. There are two reasons accounting for this. To begin with, Spark Streaming is an extension of the core Spark API that enables stream processing by dividing data streams into data batches and then processing each data batch using the Spark engine, which we call that as micro-batch processing [4]. Nonetheless, batches of streaming trajectory data are still based on immutable RDDs and are managed separately in Spark. Thus, Spark Streaming cannot support efficient offline analyses due to the massively redundant data storage caused by updates using immutable RDDs. On the other hand, Structured Streaming is an extension of the Spark SQL engine that enables stream processing by appending structured data in an Unbounded Table. However, trajectory data are usually not stored in a table as the structured data due to its important spatial and temporal characteristics. For example, it is inconvenient to build a spatial STR tree [27] index based on such table-based management. Notably, non-trivial internal modifications are needed to support indexing in Spark SQL [47], and the flexibility for doing so is limited due to its structured storage scheme [20]. In contrast, we enhance the Spark Core in terms of its underlying storage mechanism for both historical and streaming trajectory data. Based on these, we develop the Dragoon system, a hybrid and efficient big trajectory management system for offline and online analytics.

3 Background

In this section, we proceed to offer background related to the implementation of Dragoon system, including resilient distributed datasets in Spark, and hybrid management and analytics of trajectories.

3.1 Resilient distributed datasets

Resilient Distributed Datasets (RDD) is the core concept in Spark. Specifically, a data RDD is a set of data that can be divided into multiple data partitions logically in the distributed environment, where each partition is a subset of the entire dataset and corresponds to a physical data block in the underlying storage layer. The data partitions of an RDD can be sent to different nodes in a distributed environment, and each node can manage several partitions of an RDD. In addition, there is a block manager in each node that controls the data partitions that are sent to it, so that computations can be performed in parallel on different nodes. Note that although the recent Dataset concept in Spark is proposed with richer optimizations, it is more suitable for semi-structured and structured data. In contrast, RDD can provide low-level and more general data management and access.

However, the immutable model of RDDs (i.e., RDD with its partitions are read-only and cannot be modified directly) results in the limitations when related to trajectory data management scenarios (e.g., trajectory editing). This is because any data update on an RDD will create a new RDD. For example, in Fig. 2a, the original RDD$_o$ contains three data objects (i.e., o_1, o_2, and o_3), where o_1 is the one that needs to be edited during the trajectory editing. A simple solution for a data update in an immutable RDD of Spark is as below: (i) using the filter operation on the original RDD$_o$ to obtain an RDD$_e$ and another RDD$_u$, where RDD$_e$ contains o_1 of RDD$_o$ that needs to be edited and RDD$_u$ represents the remaining data objects in RDD$_o$ that keep unchanged; (ii) using the map operation to update the RDD$_e$ and get the RDD$_r$ (i.e., the edited RDD); (iii) using the union operation between RDD$_u$ and RDD$_r$ to provide a new and "updated" RDD$_n$ to users; and (iv) returning RDD$_n$ for further possible trajectory editings in the future. As shown in Fig. 2a, what we want is just the final result RDD$_n$ in the trajectory editing, but several intermediate but unnecessary data RDDs (e.g., RDD$_e$, RDD$_r$, and RDD$_u$) are created during above editing processing due to the immutability of RDDs, incurring huge data copies. The storage cost will be further exacerbated as data updates are very common in trajectory management scenarios. Therefore, the mutable data RDDs that enable direct update operations are necessary for trajectory editing scenarios. Based on that, we can execute a data update operation simply as shown in Fig. 2b.

In addition, the mutable RDDs can not only provide an efficient way to deal with above trajectory editing scenarios but also enable efficient online trajectory analysis tasks. Although Spark Streaming can be used for streaming trajectory processing, it simply transforms the newly arrived trajectory points into a data batch and stores it in a newly created data RDD. Then, Spark Streaming appends this new RDD to the end of historical dataset. In this case, data batches of trajectory streams are managed separately and sorted by the temporal information, which is inefficient for some trajectory analytics. Taking online range query as an example that aims to find trajectories in a specific spatial region, Spark Streaming needs to visit each data batch to get the final result. In contrast, the mRDD equipped Dragoon system can store locations close to each other in the same RDD, i.e., the newly arrived locations are inserted into the previous corresponding RDDs according to their spatial information. Based on that, Dragoon only visits a small number of (instead of all) RDDs to get the final online range query result. Motivated by these, we extend the underlying storage under RDD of Spark for both historical and streaming trajectory data, based on which we develop a hybrid big trajectory management system for both offline and online analytics.

3.2 Hybrid management and analytics

Trajectory data generated by moving objects (e.g., people and vehicles) can be classified into historical trajectory data and streaming trajectory data. The management and analytics on these two types of data are different. We first describe the

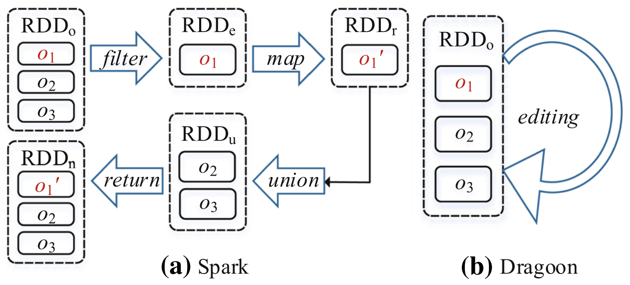

Fig. 2 Trajectory editing

trajectory management phase and then present the trajectory analytic phase.

In terms of trajectory data management, historical and streaming trajectory data involve different storage formats. Historical trajectory data, also called static or batch trajectory data, are typically represented by a sequence of trajectory points and are stored in the file blocks. In contrast, streaming trajectory data, also known as dynamic or unbounded trajectory data, are often loaded as the most recent trajectory points into the main memory for the subsequent real-time processing.

In terms of trajectory data analytics, historical trajectory data call for the offline batch processing, while streaming trajectory data call for the online real-time processing. For historical data, the existing work aims to achieve high scalability and efficiency by using a variety of optimization techniques (e.g., data partitioning [20,47] and trajectory index structures [33]). For streaming data, existing work [14] aims at real-time processing that achieves high throughput as well as low response latency. However, some real-world applications need both offline and online analysis, which we call it as hybrid analysis. For example, in traffic management applications, offline analyses of historical trajectories can be used for better traffic planning purposes, while online analyses of streaming trajectories can be adopted in traffic monitoring to detect congestion in real time. Existing studies on trajectory data management and analytics tend to solve these problems separately [37], thus causing an inefficient way to provide hybrid analysis for users. In contrast, Dragoon proposes a hybrid approach based on Spark to support simultaneously offline analysis of historical trajectories and online analysis of streaming trajectories in one single system.

3.3 Chronicle map

Chronicle Map[1] is a popular storage technique for low-latency and multiple-process access in the area of high-frequency trading (HFT), such as trade and financial market applications. Chronicle Map possesses two essential features

[1] https://github.com/OpenHFT/Chronicle-Map.

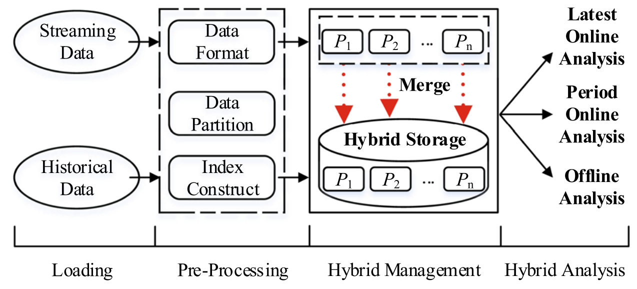

Fig. 3 Overview of dragoon

to the implementation of Dragoon, including off-heap memory persist and embedded key-value store, as explained in the following. (i) *Off-heap memory persist.* To support high performance of distributed data computing and processing, the original RDD based Spark chooses to cache data on the on-heap memory, which could incur significant overhead on the garbage collector (GC), especially for the big data processing. In contrast, Chronicle Map adopts the off-heap memory mechanism, which can not only provide a similar process performance compared with that of Spark's built-in memory cache but also significantly relieve the GC pressure for providing better data scalability. In other words, this feature provides a basic guarantee for big trajectory data management of Dragoon. (ii) *Embedded key-value store.* Recall that we target a hybrid and efficient big trajectory management system for offline and online analytics, and thus, efficient data access is required, especially for online trajectory management and analytics. Fortunately, Chronicle Map provides an embedded key-value store, which enables efficient and random data access for different data formats. For instance, the key can refer to a moving object's ID while the value can represent its various trajectory formats. Specifically, the formats can denote an observed GPS point, a sub-trajectory that consists of several GPS points, or the whole trajectory of this moving object. Moreover, we have confirmed its effectiveness in our previous work [20]. By seamlessly integrating Chronicle Map into Spark as the underlying storage engine, Dragoon can provide an **efficient**, **scalable**, and **flexible** trajectory management for both offline and online trajectory data analytics.

4 System overview

In this section, we present an overview of Dragoon, including loading, preprocessing, hybrid management, and hybrid analysis, as depicted in Fig. 3.

Loading In the first stage, Dragoon ingests historical trajectory data from a static trajectory data source or continuously loads the latest trajectory data from a streaming trajectory source.

Preprocessing The preprocessing includes format transformation, data partitioning, and index construction. First, the raw trajectory data are represented as a sequence of GPS records (id, l, t), where id denotes the ID of a moving object, l represents a location that contains latitude and longitude, and t is the time when the location was observed. Next, both historical and streaming trajectories can be divided into several data partitions according to specific partitioning rules. As shown in Fig. 3, the trajectory data are partitioned into n data partitions P_i $(1 \leq i \leq n)$. Finally, the index construction includes local-index construction on each data partition and a global-index construction over all data partitions, with the details to be provided in Sect. 6.

Hybrid Management To manage both historical and streaming trajectory data, we design the hybrid storage, which is the core part of Dragoon. The hybrid storage is physically based on the Chronicle Map. In order to store the hybrid trajectory data, we need to merge streaming and historical trajectories. The merging process is not a simple union operation as implemented in Spark, but needs to be able to support data updates at the physical Chronicle Map layer. To support data updates, we design the mRDD model, to be detailed in Sect. 5. In addition, the local indexes and global index are also stored in the Chronicle Map, and need to be updated after the data update processing.

Hybrid Analysis The hybrid analysis includes offline analysis for historical trajectory data and online analysis for streaming trajectory data. Specifically, Dragoon supports two types of online analyses, as depicted in Fig. 3. The first type of online analysis, called latest online analysis, focuses on the latest location data values of the observed moving objects. The second type of online analysis, termed as period online analysis, is performed on both the latest location data and previous historical trajectory data of moving objects (i.e., the data pertaining to a certain recent time period). The hybrid analysis details are given in Sect. 6.

5 Hybrid storage

The streaming trajectory data are collected continuously. Instead of using mini-batch strategy that divides incoming data into small batches and manages them separately, our system Dragoon appends the incoming data into the previous data partitions, which incurs an update in the underlying storage for each arriving trajectory point. However, the original RDDs of Spark are immutable, and any update on an RDD will create a new RDD, resulting in high storage costs due to unnecessary data copies. Hence, how to support frequent RDD updates of streaming data is the key to implement the hybrid storage. In view of this, we propose a mutable RDD (mRDD) model that is designed to be compatible with the original RDD model of Spark. The mRDD model includes

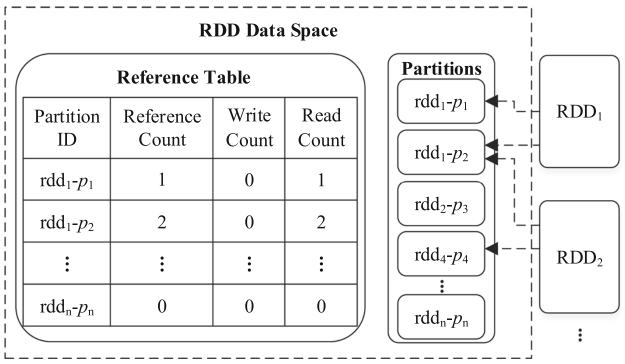

Fig. 4 An example of RDD share

three parts, i.e., RDD Share, RDD Update, and RDD Mirror, in order to support hybrid storage for both historical and streaming trajectory data. In the next of this section, we will detail these three parts by orders. To distinguish, we use mRDDs to denote our proposed mRDD model of Dragoon, while we use RDDs to represent the original RDD model of Spark.

5.1 RDD share

In Spark, data are logically stored in the distributed data partitions, but are stored physically in the distributed data blocks. That is to say, each data partition corresponds to a data block in the underlying storage layer. Inspired by the multi-version data structures [41], we develop an effective mechanism for data sharing across different RDDs, termed as **RDD Share**. When updating some data in an RDD, the RDD Share first identifies data partitions without any data updates, and then, the newly created data RDD directly reuses those data partitions that have not been recently updated. This mechanism can avoid large amounts of unnecessary data copies for the data update processing.

In Spark, the data partitions of an RDD are managed in an RDD-specific space, and thus, different data RDDs are maintained in separated storage spaces. To support RDD Share across different RDDs, we assign to each data partition (i.e., physical data block) **a unique ID**, as shown in Fig. 4. To generate a unique ID for each data partition in the data space, we combine the RDD ID and the sequence number of the data partition in its data RDD. For example, in Fig. 4, the data partition ID "rdd_1-p_1" combines the RDD ID "rdd_1" and the sequence number "p_1" of the data partition in its belonging data RDD.

In the original RDD design of Spark, each data partition only belongs to one specific data RDD and each RDD only manages its own data partitions. When a data RDD is released, its data partitions are also liberated. However, in our mRDD model, each data partition can be shared among multiple mRDDs. Therefore, the life cycle of each data partition needs to be maintained individually to support consistent data management. In other words, when an RDD is released, its data partitions cannot be released if they are shared with other mRDDs. As an example, in Fig. 4, the data partition "rdd_1-p_2" is shared by both RDD_1 and RDD_2. When RDD_1 is released by the system, it's data partition "rdd_1-p_2" cannot be liberated directly, since it is also shared by another RDD_2. Otherwise, inconsistent data issues can be raised. To avoid incorrect release of data partitions, we extend the block manager in each node by introducing a **Reference Table** that maintains a reference count for each data partition. The reference count of a data partition indicates the number of mRDDs that share this partition. An example of the reference table is depicted in Fig. 4, where the reference count of data partition rdd_1-p_1 is 1, as it is only used by mRDD_1, while the reference count of data partition "rdd_1-p_2" is 2, the reason is that it is shared by both mRDD_1 and mRDD_2.

When a data partition is created by an RDD, a new row corresponding to this data partition is inserted into the reference table, and its reference count is initialized to 1. When a new RDD shares this data partition, its reference count is incremented by 1. Similarly, when an RDD that shares this data partition is released, the corresponding reference count of this data partition is decremented by 1. Finally, when the reference count of a data partition equals 0, this data partition and its physical data block can be released safely. Note that the reference table can be implemented using Chronicle Map. In addition, each node in the Dragoon system maintains a reference table by its block manager.

RDD Share mechanism could share data partitions across different data RDDs, and it decouples an RDD's life cycle from the life cycles of its data partitions. Assume that there is an RDD and it has 1000 data partitions and only one partition needs to be updated, which could be caused by trajectory data editing or incoming streaming trajectory management. In order to support these kinds of data updates, Spark creates a new RDD that maintains one updated partition, and copies the remaining 999 partitions. In contrast, our system Dragoon creates a new RDD that maintains the updated partition but shares the remaining 999 partitions. Thus, RDD share can avoid large amounts of unnecessary data copies caused by data updates.

5.2 RDD update

In this subsection, we detail how to update those data partitions that are needed to be updated because of trajectory editing or management of trajectory streams, which we called that as **RDD Update**. The RDD Update contains three different data update strategies, including the Newest-Only, the Share-Append, and the Share-Update, which offer supports for different relevant update scenarios in trajectory management. In order to show how to perform data updates on

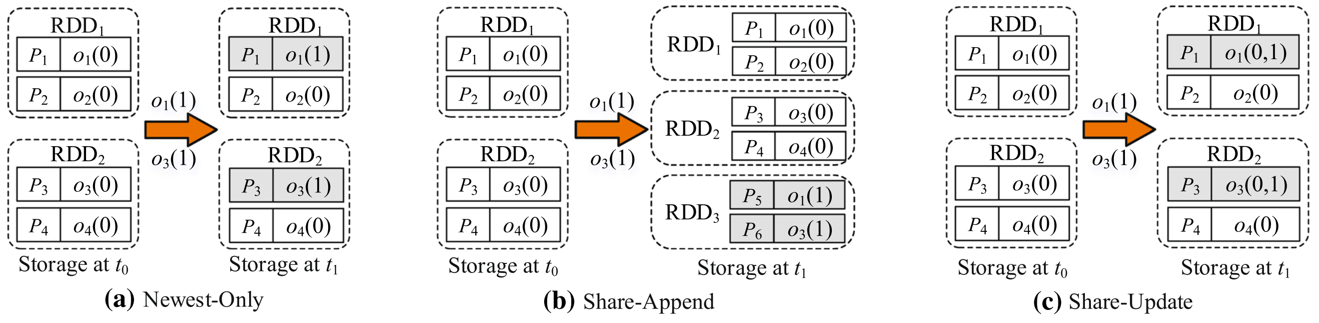

Fig. 5 A running example of RDD update

our mRDDs using these three update strategies, we use a running example as illustrated in Fig. 5 that includes four moving objects o_1, o_2, o_3, and o_4. At time t_0, their locations are stored separately in four data partitions (i.e., P_1, P_2, P_3, and P_4) according to the id information of moving objects, where $o_1(0)$ represents the location of o_1 at time t_0. At time t_1, new locations $o_1(1)$ and $o_3(1)$ are generated by moving objects o_1 and o_3, respectively. Hence, Dragoon has three update strategies for these two new location updates in this running example. (i) replacing the old values with these two new locations; (ii) appending these two new locations to the end of historical data; and (iii) inserting these two new locations into corresponding historical data partitions. In the next of this subsection, we proceed to detail how these three update strategies work using the running example, and analyze the strengths, shortcomings, and adapted scenarios of each update strategy.

The Newest-Only update strategy assumes users only focus on the latest location data values when the objects are moving, meaning that the historical trajectory data values of moving objects can be ignored directly. This scenario is common in streaming settings, where both stateful and stateless stream computing are based on the latest data. In addition, it is also common when editing and cleaning trajectory data to store only the latest correct values to save storage space. The Newest-Only strategy does not need to create a new RDD to perform an update, but replaces the old values directly in the old mRDDs with the new ones. Specifically, to perform an update, we first identify the data partition where it locates, and then, we update this data partition directly. Here, a data partition is mutable, as it is implemented by Chronicle Map [20]. However, the RDD and its data partitions are not changed in logically. An example of applying the Newest-Only strategy is depicted in Fig. 5a, where $o_1(0)$ in P_1 is replaced by $o_1(1)$, while $o_3(0)$ in P_3 is replaced by $o_3(1)$ after these two data updates. The Newest-Only update strategy is useful due to three reasons as follows.

– First, the Newest-Only strategy provides the possibility for the efficient data updates in RDDs using Chronicle Map, which is required in many application scenarios such as trajectory editing, trajectory cleaning, and streaming trajectory processing.
– Second, the Newest-Only update strategy does not create a new data RDD for a data update operation, which avoids the overheads associated with data copying compared with the original Spark.
– Last but not the least, the Newest-Only strategy always keeps the latest data in mRDDs when the data is updated. Consequently, we can directly obtain the latest data values without interruptions when continuously fetching data in real time, which is meaningful in streaming trajectory processing settings.

However, the Newest-Only update strategy has two limitations. First, it replaces the previous data values with the latest new data values, and thus, the historical trajectory data are no longer available for further analysis. Therefore, the Newest-Only strategy cannot support period online analysis. Second, it updates the data in the distributed environment, which may cause data inconsistencies when reading and writing the data simultaneously. Thus, we need to introduce access permissions before reading or writing the mRDDs, resulting in some additional time costs, which will be detailed in “RDD mirror” subsection.

The Share-Append update strategy assumes that the latest data values are appended to the end of the historical data values, where the historical data do not need to be updated. This scenario aligns with historical analyses that involve both the latest and previous locations of a streaming trajectory. An example is to find trajectories of moving objects that have moved together during the past 30 min [37]. The Share-Append strategy is based on the RDD Share, where the new data are placed in newly created data partitions. An example of Share-Append is shown in Fig. 5b, where new data partitions P_5 and P_6 containing the two updated locations $o_1(1)$ and $o_3(1)$ are appended to the end of the historical storage space. There are three advantages by using the Share-Append update strategy.

- First, the Share-Append update strategy does not significantly affect the data distribution. Specifically, when the historical data are uniformly distributed in the system, a Share-Append data update does not yield a skewed data distribution.
- Second, the data partitions under the Share-Append are naturally divided according to the temporal information of trajectories, which is beneficial for trajectory analytics with the temporal filtering.
- Last but not the least, the latest data are maintained in newly created data partitions and has no effect on the shared data partitions. Thus, the shared data partitions can be regarded as immutable with Share-Append strategy, which avoids data inconsistency issues caused by concurrent reads and writes.

Although the Share-Append strategy is simple and does not cause updates on historical trajectory data, it has two drawbacks. First, the data are distributed according to its temporal information. To obtain a different data distribution (e.g., according to the spatial information) of trajectories, the historical data need to be repartition and rebuild indexes periodically. Second, although a single update does not significantly affect the data distribution, the continuous data updates may result in declining of query performance.

The Share-Update update strategy assumes that only a small fraction of the entire dataset is updated at a time. This is because the locations generated by moving objects at a specific time are much fewer than all archived historical locations in the entire dataset. The Share-Update strategy is based on RDD Share, where a newly created RDD shares data partitions without data changes, and copies the remaining data partitions by inserting the new incoming data. Note that, unlike the Newest-Only strategy, the update in Share-Update is an insertion operation instead of a replace operation. An example of Share-Update is shown in Fig. 5c, where the data in P_1 are updated to $o_1(0, 1)$, while the data in P_3 are updated to $o_3(0, 1)$. The Share-Update combines the RDD update mechanism with RDD Share, which has the following strengths.

- First, trajectory data can be divided using different data partition strategies in order to support flexible data balancing and efficient global management.
- Second, the new incoming data can be distributed uniformly to different cluster nodes in order to improve parallel and overall performance.
- Last but not the least, the latest trajectory data are maintained in the newly copied data partitions and have no effect on those shared data partitions.

The Share-Update strategy can efficiently overcome the disadvantages of the Newest-only and Share-Append strategies. Nonetheless, it still needs additional time costs for applying write and read permissions, as does the Newest-Only update strategy.

In summary, the Dragoon system provides three different data update strategies to form RDD Update for different trajectory related updating scenarios. Besides, each update strategy is based on different assumptions, and has its own advantages and adaptation scenarios. It is worth mentioning that the proposed mRDD model of Dragoon is totally compatible with the original RDD model of Spark. Therefore, the existing transformation and action operations in mRDD are the same with those supported by RDD. The difference between mRDD and RDD is that mRDD introduces the update mechanism to support efficient data updates, which belongs to one of the transformation operations that also follows the lazy evaluation of Spark.

5.3 RDD mirror

In the original design of RDD, an RDD is read-only and cannot be updated directly; thus, there does not exist any data inconsistencies. However, as discussed earlier, we introduced the mRDD model to support data updates in mRDDs, which makes the RDD be writeable and readable. To avoid the data inconsistencies caused by concurrent data reading or writing, data updates require permission controls and locking mechanisms to manage permissions. As data in a mutable RDD are stored and managed at the granularity of data partitions, the read and write permissions can also be controlled at the granularity of data partitions.

In this subsection, we introduce the RDD Mirror mechanism, which further extends the reference table in the block manager to implement read/write permissions and locks in a distributed environment. Specifically, the RDD Mirror records permission information (i.e., a read count and a write flag) for each data partition in the reference table, as depicted in Fig. 4. When the write flag of a data partition is 0, meaning that no task is writing this data partition; otherwise, when its write flag is 1, it indicates that a task is writing this data partition now. The read count records the number of concurrent readings. As an example in Fig. 4, assume that data partition $\text{rdd}_1\text{-}p_2$ is currently being read by two applications and its read count is 2.

When a task applies for a read or write permission of an RDD, it creates an RDD mirror that can be used for recovery. RDD Mirror shares all the unchanged partitions via the RDD Share, and it updates the permission information of the changed data partitions. The RDD Mirror needs to check for permission conflicts before establishing an RDD mirror. After both the reference count and the permission information are successfully updated, the RDD mirror is fully established. The RDD mirrors can be classified into readable RDD mirrors and writable RDD mirrors as follows.

Readable RDD Mirror When an application reads an RDD, it needs to obtain a read permission before creating a readable mirror of this RDD. The readable RDD mirror shares all the data partitions with the original RDD and checks its write flag for each data partition. If a data partition exists whose write flag equals 1, the read permission request fails; otherwise, if all the write flags of the data partitions equal 0, their read counts are incremented by 1. After the reading is completed, their read counts are decremented by 1.

Writable RDD Mirror When an application updates an RDD, it needs to apply for the write permission and create a writable mirror of this RDD. The writable RDD mirror requires that none of its data blocks are read or written by other tasks, i.e., the read count and the write flag of every data partition are equal to 0. In Fig. 4, if a write permission request for rdd$_1$-p_2 is issued by an application, the request will fail as the read count of rdd$_1$-p_2 is 2. If there are no conflicts for this write permission request, we set the write flags of all the data partitions in this RDD to 1. After completing the writing, the write flags are set as 0.

5.4 Fault tolerance

In this subsection, we introduce the fault tolerance mechanisms of the Dragoon system related to the distributed processing process and read/write permissions. To begin with, the Spark platform offers multiple levels of data persistence, including the task and process levels. If an RDD is not cached, it can be regarded as being persisted at the task level, meaning that the data will be lost if a task fails. While a cached RDD, either in memory or on disk, is persisted at the process level, which denotes that the data can be recovered if a task fails, it will be lost if the process crashes. To persist data at higher levels, users have to manually and regularly save the dataset by using other services (e.g., HDFS), which is inconvenient and time-consuming.

In Dragoon, trajectory data persisted at the Chronicle Map storage are saved transparently through reliable services at runtime. Moreover, this persistence does not sacrifice the performance of in-memory data access. To achieve this, a Chronicle Map instance is created by default upon a file in shared memory (e.g., /dev/shm in Linux). Data in this file can survive task failures and can be accessed at in-memory speed. Furthermore, to serve as a reliable distributed storage, several techniques are applied to prevent both historical and streaming data storage from failures. For example, Dragoon asynchronously backs up the files in shared memory or on disk to a reliable file system (e.g., HDFS), so that the data can survive task failures and node crashes. As a result, missing data can be reloaded automatically under the Spark recomputation mechanism. In addition, when updating data mRDDs in Dragoon, the RDD Mirror function is integrated into the block manager of each node to implement read/write permissions and locks. In addition, every update request is saved as a log file in Chronicle Map in order to ensure that the data updates are executed in order.

Algorithm 1: IdPartitioning Algorithm

```
   Input: the historical trajectory dataset $S$, a partition key value $k$
1  for each trajectory $T \in S$ do
2      PartitionId ← $\langle \lfloor T.id/k \rfloor \rangle$
3      output (PartitionId, $T$)
```

Algorithm 2: GridPartitioning Algorithm

```
   Input: the historical trajectory dataset $S$, a grid cell width $l_g$
1  for each trajectory $T \in S$ do
2      for each GPS record $r \in T$ do
3          GridId ← $\langle \lfloor r.l.x/l_g \rfloor, \lfloor r.l.y/l_g \rfloor \rangle$
4          output (GridId, $r$)
```

Algorithm 3: STRPartitioning Algorithm

```
   Input: the historical trajectory dataset $S$, the number of leaf
          nodes in R-tree $n_{nodes}$
1  initialize the R-tree $rt \leftarrow \emptyset$
2  SampleDataset = sample($S$)
3  $rt$ = STR.build(SampleDataset, $n_{nodes}$)
4  for each trajectory $T \in S$ do
5      for each GPS record $r \in T$ do
6          PartitionId ← $rt$.locate($r$)
7          output (PartitionId, $r$)
```

6 Hybrid analysis

In this section, we present the detailed hybrid analysis pipeline for historical and streaming trajectory data. As discussed in Sect. 4, the pipeline contains four stages, i.e., loading, preprocessing, management, and analysis. Thus, we offer the details of four stages in the offline and online trajectory data analysis pipelines, respectively.

6.1 Offline analysis pipeline

The offline analysis pipeline for historical trajectory data is simple, since the entire trajectory dataset is known in advance and can be loaded into the Dragoon system at once for the subsequent processing.

Loading The first stage is to load the entire historical trajectory dataset. The historical trajectory dataset is usually stored in the HDFS system, and hence, it can be loaded into Dragoon in parallel. In this stage, a customizable data loader is provided to support different file formats (e.g., txt, csv, or xml).

Preprocessing The second stage includes format transformation (e.g., trajectory points to segments), data partitioning, and index construction. The customizable data partitioners are provided as follows, which use different features (e.g., the id, spatial, or temporal information) of trajectories to partition the loaded trajectory data considering different characteristics of trajectories. In other words, trajectories with the same or similar id/spatial location/time features will be sent to the same data partition for parallel processing.

- *IDPartitioner* is based on the HashPartitioner of Spark, with the pseudo-code depicted in Algorithm 1. It takes as inputs the whole historical trajectory dataset S and a partition key value k. For each trajectory T in S, the algorithm first computes the $PartitionId$ of the data partition that T belongs to (line 2). Specifically, the $T.id$ represents the ID of the moving object that generated this trajectory T. Next, it sends T to this data partition, where trajectories with the same PartitionIds are distributed to the same data partition (line 3).
- *GridPartitioner* is inspired by the Grid-index [44], with the pseudo-code shown in Algorithm 2. It takes as inputs the whole historical trajectory dataset S and a grid cell width l_g. For each location r in each trajectory T in S, the algorithm first computes the $GridId$ of the spatial grid cell that r belongs to (line 3). Note that the $r.l$ is the spatial location of this GPS record r, which includes $r.l.x$ and $r.l.y$. Next, it sends r to its corresponding data partition, where locations in the same spatial grid cell are distributed to the same data partition (line 4).
- *STRPartitioner* is inspired by the STR tree [27], with its pseudo-code presented in Algorithm 3. It takes as inputs the whole historical trajectory dataset S and the number of leaf nodes n_{nodes} in R-tree. The algorithm first randomly samples the whole trajectory dataset S (line 2) to get a sampled dataset $SampleDataset$, and then, it uses the $SampleDataset$ to build an R-tree by applying the STR algorithm [27] with n_{nodes} (line 3). Next, it partitions the S into several disjoint data partitions with approximately equal size based on the R-tree (line 4-7). The STRPartitioner can achieve better load balance compared with the former GridPartitioner, because it divides the dataset according to the data distribution of the entire dataset.
- *TimePartitioner* is also based on the HashPartitioner of Spark, with the pseudo-code depicted in Algorithm 4. It also takes as inputs the whole historical trajectory dataset S and a partition key value k. For each GPS record r of each trajectory T in S, the algorithm first computes the $PartitionId$ of the data partition that r belongs to (line 3) based on temporal information of GPS records. It is worth mentioning that $r.time$ represents the specific timestamp of this GPS record r when it is generated.

Algorithm 4: TimePartitioning Algorithm

```
  Input: the historical trajectory dataset $S$, a partition key value $k$
1 for each trajectory $T \in S$ do
2     for each GPS record $r \in S$ do
3         $PartitionId \leftarrow \langle \lfloor r.time/k \rfloor \rangle$
4         output $(PartitionId, r)$
```

Next, the TimePartitioner will send each r to its corresponding data partition, indicating that trajectories with similar timestamps are distributed to the same data partition (line 4).

With different trajectory data partitioners, the Dragoon system could support more flexible data balancing. After the data partitioning, Dragoon builds the local indexes and a global index similar to the way how it is done in UlTraMan [20]. Specifically, Dragoon builds a local index in each data partition, and then builds a global index based on the features of all data partitions. For example, to build a global R-tree, Dragoon needs to collect pids and MBRs from every data partition, where the pid is the partition ID, and the MBR is the partition's spatial minimum bounding.

Management The historical trajectory data management includes how to store the data and the indexes. The storage in Dragoon is implemented based on Chronicle Map, which is an off-heap memory and an embedded key-value storage designed for low-latency and multiple-process access. The reason why we adopt Chronicle Map to store the trajectory data is that it provides efficient data updates and can ease the garbage collector pressure in Spark. Compared with Spark, our system based on Chronicle Map achieves better scalability, as confirmed by our experiments in Sect. 8.

Analysis The offline trajectory analysis contains typical querying and mining tasks of historical trajectory data. The offline analysis can be accelerated by global filtering using the global index, and then, its result can be refined by local analyses aided by local index in each data partition. More details about the offline analysis tasks of historical trajectory data that we focused on in this work will be discussed in Sect. 7.

6.2 Online analysis pipeline

The online analysis pipeline for streaming trajectory data is illustrated in Fig. 6. The online analysis of streaming trajectory data not only focuses on the latest trajectory data values but also needs to merge the latest data with historical trajectory data. It is worth mentioning that the latest data is the data that has arrived at the most recent time point. For the latest online analysis, we aim at the latest spatial data values of all moving objects instead of new incoming data at the

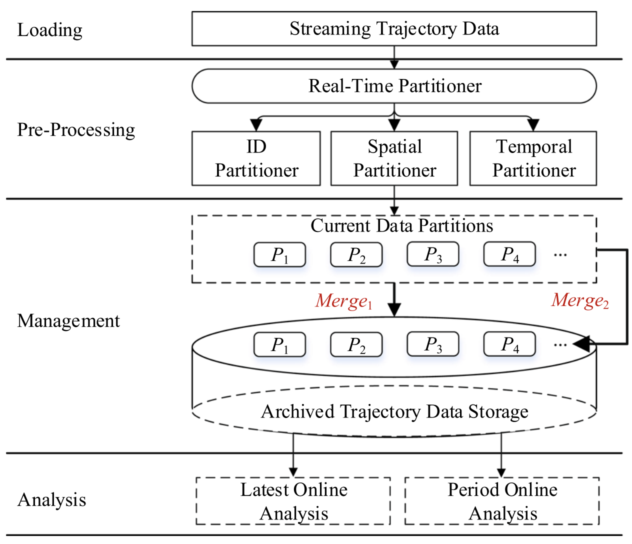

Fig. 6 Online analysis pipeline for trajectory streams

Algorithm 5: Online IdPartitioning

```
Input: a set $S_t$ of the newly arriving trajectory points at time $t$,
       a partition key value $k$
1 for each GPS point $r \in S_t$ do
2     PartitionId ← $\langle \lfloor r.id/k \rfloor \rangle$
3     output (PartitionId, r)                          ▷ IdPartitioner
```

Algorithm 6: Online GridPartitioning

```
Input: a set $S_t$ of the newly arriving trajectory points at time $t$,
       a grid cell width $l_g$
1 for each GPS point $r \in S_t$ do
2     GridId ← $\langle \lfloor r.l.x/l_g \rfloor, \lfloor r.l.y/l_g \rfloor \rangle$
3     output (GridId, r)                               ▷ GridPartitioner
```

most recent time point. This is because not all moving objects update their locations at every time point. As an example, in Fig. 5, the latest (i.e., the newly arriving) data includes $o_1(1)$ and $o_3(1)$, whereas the latest data values of all the four moving objects at time t_1 are $o_1(1)$, $o_2(0)$, $o_3(1)$, and $o_4(0)$, where o_2 and o_4 do not update their locations at time t_1. In addition, the latter is our focused latest online analysis. In contrast, for another period online analysis, we aim at both the latest and previous data values of moving objects.

Loading In the first stage, Dragoon continuously reads the newly arriving data from a trajectory stream. The system reads the data stream similar to how it is done by Spark Streaming, which reads the data stream during the last few seconds (e.g., every 5 s) as a mini-batch, and each batch of streaming data is processed as an independent RDD. The customizable data loader used in the offline analysis can be also used here.

Preprocessing Although both the online and offline trajectory preprocessing need data partitioning process that is based on the id, spatial location, or temporal information of trajectories, and the index construction process, there are several differences between the online and offline preprocessing. First, offline trajectory data partitioning takes the entire historical trajectory dataset as an input and executes the data partitioning only once, while the streaming trajectory data are partitioned by real-time partitioners that take trajectory data points of moving objects that are observed at each time as inputs, and execute the data partitioning at each time in real time. Second, the index construction of offline analysis also usually executes only once and the (local and global) indexes generally do not need to be updated. However, in the streaming trajectory settings, not only trajectory data in data partitions but also the local and global indexes need to be updated during the next merge process stage.

– *Online IdPartitioner* is similar as the IdPartitioner of offline trajectory preprocessing, with the pseudo-code depicted in Algorithm 5. The difference is that the Online IdPartitioner takes as inputs a set S_t of the newly arriving trajectory points at time t and a partition key value k. For each GPS point r in S_t, the algorithm first computes the $PartitionId$ of the data partition that r belongs to (line 2). Next, it sends r to its corresponding data partition (line 3). Note that the mRDD update process, which merges the newly arriving trajectory points to historical trajectory data, will be executed at the next stage. Different from Algorithm 1, Algorithm 5 repeats the above processing at every time.
– *Online GridPartitioner*, as the pseudo-code is shown in Algorithm 6, which is similar as Algorithm 2. The only difference is that the online method takes a set S_t of the newly arriving trajectory points at time t as an input, while the offline method takes the entire trajectory dataset as an input. Note that a subsequent merge process will be performed by merging the new coming data with historical trajectory data in this data partition. However, the grid cells are fixed for all locations generated by the moving objects at every time, which could result in a skew partitioning. To get better data partitioning and data balancing, the Online STRPartitioner is also provided as follows.
– *Online STRPartitioner* is also similar as the STRPartitioner of the offline trajectory preprocessing, with its pseudo-code presented in Algorithm 7. It takes a set S_t of the newly arriving trajectory points at time t instead of the entire trajectory dataset as an input. The online STRPartitioner can achieve better load balance compared with the online GridPartitioner, because it divides the dataset according to the data distribution of each S_t in real time.
– *Online TimePartitioner* can be easily implemented by the Share-Append strategy of Dragoon since the trajectory stream comes in chronological time order. Thus, the detailed algorithm is omitted here.

Algorithm 7: Online STRPartitioning

```
Input: a set $S_t$ of the newly arriving trajectory points at time $t$,
       the number of leaf nodes in R-tree $n_{nodes}$
1 initialize the R-tree $rt \leftarrow \emptyset$
2 $SampleDataset$ = sample($S_t$)
3 $rt$ = STR.build($SampleDataset$, $n_{nodes}$)
4 for each GPS point $r \in S_t$ do
5     $PartitionId \leftarrow rt$.locate($r.l$)
6     output ($PartitionId$, $r$)                    ▷ STRPartitioner
```

Management The online trajectory management involves the merge process about merging newly arriving trajectory points with the historical trajectory data. The real-time partitioner in the previous preprocessing stage may generate the same partitions all the times (e.g., IDPartitioner and GridPartitioner) or the different partitions at different time (e.g., TimePartitioner and STRPartitioner). In the former case, we directly distribute the partitioned S_t to the corresponding existing data partitions using the Newest-Only or Share-Update strategies based on mRDD model, which is highlighted as a red line "$Merge_1$" in Fig. 6. In the latter case, we append newly created data partitions of S_t to the existing partitions using the Share-Append strategy, to realize the "$Merge_2$" in Fig. 6. During the merging, both the writable RDD mirror and readable RDD mirror are used to avoid data inconsistencies: One is for updating processing and the other is for result reading processing. In addition, both local and global indexes also need to be updated after data merging process.

Analysis The online analysis of streaming trajectory data includes the latest online analysis and period online analysis, as depicted in Fig. 6. The latest online analysis considers only the latest data for all moving objects. However, the period online analysis needs both the latest and historical data in a given time period. More details about the online analysis of trajectory streams will be discussed in Sect. 7.

7 Analytic case studies

We demonstrate Dragoon's flexibility through several typical offline and online trajectory data analytic case studies, including the (online) ID queries, the (online) range queries, the (online) k nearest neighbor (kNN) queries, the offline trajectory editing, and the real-time co-movement pattern detection on trajectory streams.

To begin with, let $O = \langle o_1, o_2, ..., o_m \rangle$ be a set of moving objects, in which each $o_i \in O$ $(1 \leq i \leq m)$ is a moving object that can produce a trajectory or a streaming trajectory as defined below.

Definition 1 (Trajectory). A trajectory T_i, generated by the moving object o_i $(1 \leq i \leq m)$ in the geographical space, is usually represented by a sequence of chronologically ordered GPS records, i.e., $T_i = \langle r_1, r_2, ..., r_n \rangle$. Each GPS record r_j $(1 \leq j \leq n) \in T_i$ can be represented as a triple $(id, l, time)$, where l is the spatial location of the object o_{id} $(id = i)$ at the timestamp $time$. Note that the l usually consists of a geospatial coordinate set (*longitude*, *latitude*).

Definition 2 (Streaming Trajectory). A streaming trajectory generated by a moving object o_i $(1 \leq i \leq m)$ is also a time-ordered sequence of GPS records, i.e., $ST_i = \langle r_1, r_2, ... \rangle$. Note that, different from the trajectory T_i, the streaming trajectory ST_i of the moving object o_i is unbounded.

Hence, a static trajectory dataset (***STD***) contains all the trajectories generated by O, i.e., $STD = \langle T_1, T_2, ..., T_m \rangle$. A dynamic trajectory dataset (***DTD***) contains all the streaming trajectories generated by O, i.e., $DTD = \langle ST_1, ST_2, ..., ST_m \rangle$. In the sequel, we proceed to introduce the typical offline and online trajectory analysis cases on both historical and streaming trajectory data. In addition, an experimental evaluation of these analysis cases will be detailed in Sect. 8.

7.1 Trajectory editing

Given a static trajectory dataset *STD*, there may have noisy sample points [57]. For example, when a GPS receiver is in an urban canyon and satellite visibility is poor, inaccurate locations may occur. To clean such trajectory noisy points, trajectory editing operations are used. More specifically, the trajectory editing aims to update the noisy trajectory points with correct ones. As depicted in Fig. 2a, the trajectory editing in Spark will result in many unnecessary data copies due to the immutability of RDDs. In contrast, Dragoon provides the mRDD model that enables to update data RDDs directly, as shown in Fig. 2b, providing a more effective and powerful mechanism for trajectory data management and maintenance.

7.2 ID and online ID queries

ID and online ID queries aim to find one specific trajectory according to its ID information, which are useful in trajectory monitoring scenarios, as defined below.

Definition 3 (ID and Online ID Query). Given a specific id, the ID query and online ID query find the trajectory T_i in *STD* and streaming trajectory ST_i in *DTD* respectively, which is generated by the moving object o_i, where $i = id$.

For the offline ID query, the IdPartitioner first partitions the entire trajectory dataset *STD* into multiple data partitions according to IDs of moving objects to enable subsequent parallel processing. As shown in Algorithm 1, if we set k =

10000, the IdPartitioner could assign (i.e., partitionID = id/k) moving objects whose IDs between 0 and 10000 to data partition P_0, and whose IDs between 10001 and 20000 to data partition P_1, etc. Then, we build a local Hash index in each partition, where the object ID is used as the key and the trajectory points are regarded as the values. Since the Chronicle Map itself is a hash map, the local Hash index can be implemented easily based on the Chronicle Map storage. Next, a global index can be built based on the features of all data partitions. The features of a partition include the partition ID and the moving object ID range in this data partition. Considering the example described above, the global index contains $\langle(P_0.ID, [0, 10000]), (P_1.ID, [10001, 20000]), ...\rangle$. Finally, an ID query can use the global index to filter unnecessary data partitions based on the given id, and it can use a local Hash index to find the final result directly.

For the online ID query, the Real-Time IdPartitioner is used to assign the newly arriving trajectory points at time t (S_t) to several data partitions according to the IDs of the streaming trajectories, and then merges them with the historical locations in each data partition. The online ID query is an instance of the latest online analysis, as it returns the current location of a streaming trajectory whose ID equals the given ID at each time; thus, the online ID query is conducted on the current locations of all streaming trajectories at each time point. Therefore, all three different update strategies covered in Sect. 5 can be used for this merging processing. In addition, we also need to update the local and global indexes after trajectory merging.

7.3 Range and online range queries

Range and online range queries aim to find trajectories in a certain spatial range region, which are useful in applications such as traffic monitoring and hotspot detection [57], as defined below.

Definition 4 (Range and Online Range Query). Given a certain spatial search region Q, an *STD*, and a *DTD*, the range query finds the trajectories in *STD* that have intersected with Q, while the online range query finds the streaming trajectories in *DTD* whose current spatial locations locate inside Q. Note that the search region Q is usually represented as a rectangle $\langle[min_x, min_y], [max_x, max_y]\rangle$.

For the offline range query, we adopt the GridPartitioner to partition *STD* based on spatial information of trajectories, where a local R-tree is built on each data partition and a global grid index is built on the whole trajectory dataset *STD*. Similar to the ID query, the global grid index can filter those data partitions that are not intersected with search region Q, and then, local R-trees can be utilized to support range queries on the candidate partitions to obtain the final result.

For the online range query, the online GridPartitioner is first used to partition the new incoming points (S_t) from the streaming trajectories and then merge the incoming points with the historical data of corresponding data partitions. The online range query is also an instance of the latest online analysis. It returns the steaming trajectories whose current locations are contained in the search region Q, and the query is performed on the latest locations of all streaming trajectories. Thus, all three update strategies can be employed to merge the data. In addition, the corresponding local and global indexes are updated after data merging.

7.4 *k*NN and online *k*NN queries

kNN and online kNN queries aim to find k nearest trajectories for a specified spatial location, which are useful in location-based services such as trajectory classification and ride sharing [13], as defined below.

Definition 5 (kNN and Online kNN Query). Given a location l, an *STD*, a *DTD*, and an integer $k \geq 1$, the k nearest neighbor (kNN) query finds k nearest trajectories in *STD* for l, i.e., $R_k = \{T_i | 1 \leq i \leq k, d(T_i, l) \leq d(T_j, l)(T_j \notin R_k)\}$, and online kNN query finds k streaming trajectories in *DTD* whose current locations are closest to l, i.e., $OR_k = \{ST_i | 1 \leq i \leq k, d(ST_i.r_t, l) \leq d(ST_j.r_t, l)(ST_j \notin OR_k)\}$.

Note that the distance computation $d(T_i, l)$ and $d(ST_i, l)$ between location l to a trajectory T_i or ST_i follow the work [13], although other distance functions can also be implemented in the Dragoon system.

For the offline kNN query, we adopt the STRPartitioner [47] to uniformly partition the trajectory data according to its spatial information. Here, the R-tree variant [20] is used to improve kNN query efficiency, where each node of R-tree maintains both the minimal bounding rectangle (MBR) and a count of distinct trajectories contained in its MBR. During the kNN query, global filtering is utilized to obtain candidate data partitions with trajectory counts no less than k, and then, local kNN queries are performed in the candidate partitions individually. Finally, the local results from the candidate partitions are sorted in ascending order of their distances to the query location, and then, the global top-k trajectories are returned.

For the online kNN query, we use the MBRs in the global R-tree index instead of a STRPartitioner to partition the latest locations from the streaming trajectories. This is because the STRPartitioner could cause additional cost for data merge as discussed in Sect. 6.2. Next, we merge the data and update the local and global indexes. Similar to the online ID and Range queries, the online kNN query is an instance of latest online analysis. The online kNN query returns k trajectories whose current locations are closest to the query location at each time; thus, it is performed on the latest locations of all streaming trajectories. Hence, all three update strategies can be used for data merging.

7.5 Co-movement pattern detection

Co-movement pattern detection aims to discover co-moving objects that satisfy specific spatiotemporal constraints [14], including the *closeness*, the significance M, the duration K, the consecutiveness L, and the connection G. Here, we focus on the real-time co-movement pattern detection on streaming trajectories, which is useful in many applications such as future movement prediction. The real-time co-movement pattern detection is an example of period online analysis, and thus, we must consider the latest locations as well as the historical locations of each streaming trajectory.

We directly adopt the state-of-the-art method [14] for the real-time co-movement pattern mining. However, the underlying system of Dragoon is different from that of the Flink platform, which can achieve better scalability. This is because, the state[2] storage in Flink is based on on-heap memory to enable high performance, which has a significant pressure on the garbage collector (GC), especially for the big trajectory data maintenance and processing. In contrast, our system uses Chronicle Map for physical storage, which is off-heap memory that relieves the GC pressure, while guaranteeing the efficiency at the same time. In addition, the streaming trajectory data maintained by states in Flink are released after completing the stream analysis. In contrast, Dragoon provides permanent storage of streaming trajectory data, and hence, the data can be used for subsequent trajectory analytics in the future.

Since the real-time co-movement pattern detection is an instance of the period online analysis, the Newest-Only update strategy is not applied anymore. This is because the Newest-Only strategy directly discards the previous location values of moving objects. The co-movement pattern detection method updates the indexes during querying to further improve query efficiency. In addition, the intermediate results (e.g., the clusters) also need to be stored. The method contains two phases, clustering and enumeration. For the clustering phase, both the Share-Append and Share-Update strategies can be employed to merge data; for the enumeration phase, only the Share-Update strategy can be used since we need to partition the data according to its ID in order to realize parallel pattern detection (i.e., ID partitioning is used in the method [14]).

[2] https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/state/.

Table 1 Statics of datasets

Attributes	GeoLife	Taxi	Brinkhoff
#trajectories	18,670	20,151	100,000
#locations	24,876,978	2,273,039,208	4,769,059,392
#snapshots	92,645	6,030,792	71,800
Ave. length	1,332	112,800	47,690
Data size	1.5G	102G	201G

8 Experimental evaluations

In this section, We evaluate the performance of Dragoon with typical trajectory analytic cases as discussed in Sect. 7, and we compare Dragoon with the existing state-of-the-art offline trajectory management system UlTraMan and also the general-purpose online processing systems, including Spark Streaming and Flink. Recall that the core component of Dragoon is the mRDD model, based on which hybrid trajectory data analytics including offline and online analytics can be efficiently supported. Hence, in the subsequent experiments, we evaluate the performance of two components of Dragoon, i.e., mRDD model and hybrid trajectory data analytics. (i) To verify the performance of mRDD model, we first report the performance of offline trajectory editing that are based on mRDD model, and then, we report the data update performance during online trajectory queries. This is because, when performing online ID/Range/kNN query analytics, Dragoon first merges the newly arriving trajectory points with historical trajectory data in the underlying storage. This merge process is mainly based on the mRDD model, where three update strategies (i.e., Newest-Only, Share-Append, and Share-Update) provided by Dragoon are evaluated. (ii) To evaluate the performance of hybrid trajectory data analytics, we use several typical offline (i.e., ID/Range/kNN queries) and online (i.e., real-time moving pattern detection and ID/Range/kNN queries) trajectory analytic cases, as discussed in Sect. 7.

Datasets In our experiments, we use two real-life datasets (i.e., GeoLife and Taxi) and one synthetic dataset (i.e., Brinkhoff), whose detailed information including the number of trajectories, the number of locations, the number of snapshots, average length, and data size are summarized in Table 1.

- GeoLife[3]: This dataset keeps the GPS records of each user during a period of more than 3 years. The GPS information is collected periodically, and 91% of the trajectories are sampled every 5 s.

[3] https://research.microsoft.com/en-us/projects.

– Taxi[4]: This dataset is generated by taxies in Hangzhou, China. Trajectories are identified by the number plate of taxis, and each trajectory represents the trace of a taxi for 3 months with a sampling rate of every 5 s.
– Brinkhoff[5]: This dataset is generated via the Brinkhoff generator [12]. The trajectories are generated on the real road network of Las Vegas city. Positions of moving objects are updated every 1 second, and an object moves along a road network with random but reasonable direction and speed.

Parameters In experiments, we study the effect on the performance of varying settings for five parameters, as summarized in Table 2, where default values are shown in bold. Here, n represents the number of editing times on a static trajectory dataset in trajectory editing. Second, O_r denotes the percentage of moving objects w.r.t. the entire trajectories in the offline trajectory analytic cases, which also represents the percentage of moving objects who issue updates at every time during the online analysis. Besides, ϵ represents the query region used in (online) range queries w.r.t. the size of the whole region that contains all trajectory points. k denotes the k used in (online) kNN queries. N represents the number of slave worker nodes for Dragoon system. In addition, parameters used in real-time co-movement pattern detection are set as the default values in previous work [14], i.e., $M = 15$, $K = 180$, $L = 30$, and $G = 30$. Note that we adopt the L_1-norm as the similarity distance between two locations of trajectories in the following experiments for simplification. However, it is easy to support other distance functions.

Performance Metrics (i) For the ID/Range/kNN query and trajectory editing, the execution time (i.e., the average processing time for each query or editing) is employed to evaluate their performance. (ii) For the online ID/Range/kNN query and co-movement pattern detection, we use both the latency and throughput as performance metrics. More specifically, for the online ID/Range/kNN query, we verify the update and query processing phases, separately. In the update phase that targets finding corresponding data partitions and then inserting the newly arrived trajectory points into the historical data partitions, the update latency denotes the average time for inserting current S_t (all the points arrived at time t) into historical trajectories, and the update throughput represents the number of S_t inserted by the system per second. In the query phase processing, the query latency is defined as the average response time for each online ID/Range/kNN query, and the query throughput denotes the number of ID/Range/kNN queries processed by the system per second. (iii) Last but not the least, we also report the memory occupation of system during data updating processing scenarios

Table 2 Parameter ranges and default values

Parameter	Range
The number of editings n	100, 200, 300, 400, 500
The ratio of objects O_r (%)	10, 20, 40, 60, 80, **100**
The query area ϵ (%)	0.01, 0.02, 0.04, **0.08**, 0.16, 0.32
The value of k	4, 6, 8, **10**, 12, 14
The slave workers N	1, 2, 4, 8, **10**

including both trajectory editing and update phases of online ID/Range/kNN queries.

Experimental Setup All experiments were conducted on a cluster consisting of 11 nodes, where one node serves as the master node, and the remaining nodes serve as worker nodes. Each node is equipped with two 12-core processors (Intel Xeon E5-2620 v3 2.40 GHz), 64 GB RAM, and a Gigabit Ethernet. Each cluster node runs the Ubuntu 14.04.3 LTS system with Hadoop 2.7.1, Spark 2.1.1, and Chronicle Map 3.14.0. In each slave node, we allocate 40GB main memory for trajectory data storage and computation. All system modules were implemented in Scala.

8.1 Offline analysis

For offline trajectory analysis, we explore the performance of Dragoon in terms of execution time and compare it with the state-of-the-art offline trajectory management system UlTraMan. First, we compare Dragoon and UlTraMan using offline ID/Range/kNN queries. Figures 7, 8, and 9 show the results on three datasets, where each measurement is an average of

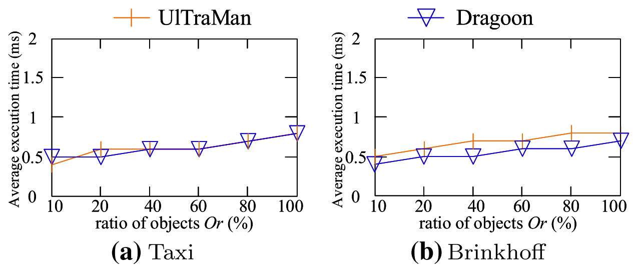

Fig. 7 Performance of ID queries versus O_r

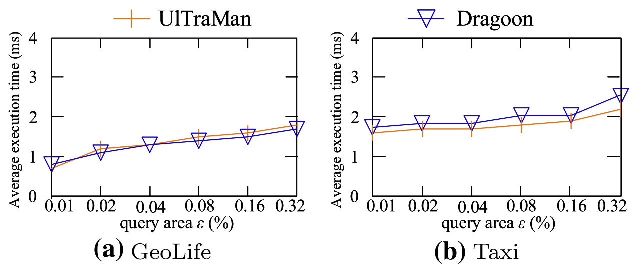

Fig. 8 Performance of range queries versus ϵ

[4] This is a proprietary dataset.

[5] https://iapg.jade-hs.de/personen/brinkhoff/generator/.

five random queries. Figure 7 verifies the impact of O_r on the performance of ID queries, Fig. 8 concerns the impact of ϵ on range query performance, and Fig. 9 depicts the impact of k on kNN query efficiency. Figures 7, 8, and 9 show that UlTraMan and Dragoon perform similarly in offline trajectory queries. This is because Dragoon extends Spark to support streaming trajectory data management and analytics, while the techniques used for offline trajectory analysis in Dragoon and UlTraMan are the same.

In addition, we proceed to use trajectory editing to evaluate the mRDD performance, and compare it with RDD of UlTraMan. Specifically, we report the execution time for trajectory editing and the memory occupied by the system when each editing is completed. Figure 10 plots the results of Dragon compared with that of state-of-the-art system UlTraMan on Taxi and Brinkhoff datasets by varying the number of editing times n from 100 to 500, where the execution time is denoted by polyline and the memory occupation is denoted by column. As we can see, the memory occupation of UlTraMan increases obviously, and UlTraMan fails to support trajectory editing when n reaches 400 (on Taxi dataset) or 300

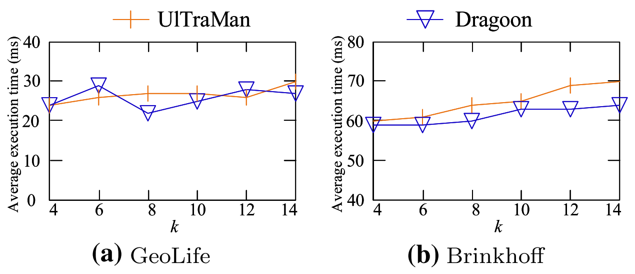

Fig. 9 Performance of kNN queries versus k

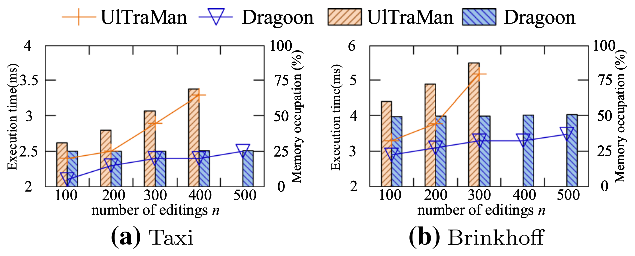

Fig. 10 Performance of trajectory editing versus n

(on Brinkhoff dataset). The reason is that UlTraMan is still based on immutable RDDs of Spark, meaning that a large number of unnecessary data copies are generated during the update process, incurring excessive memory pressure on the system, as we have discussed in Sect. 3.1. In contrast, Dragoon always occupies a low system memory, and offers low update time because its updates are directly and based on mutable RDDs. Not to mention that we only performed 500 trajectory editing operations here while there are massive trajectory points (e.g., GeoLife contains tens of millions points, and Brinkhoff includes hundreds of millions points). Note that the trajectory editing experiments are conducted on the whole trajectory dataset instead of a single RDD. To sum up, Dragoon is more suitable, efficient, and scalable than UlTraMan for trajectory updating scenarios.

Last but not the least, we offer insights into the offline scalability performance of Dragoon by modifying the number of slave workers N. The results are shown in Table 3. As expected, the ID/Range/kNN query performance first improves dramatically and then improves slowly when varying N from 1 to 10. This is because, on the one hand, the degree of parallelism ascends with the growth of N; on the other hand, the communication cost in the distributed environment also increases.

8.2 Online ID query

Next, we first evaluate the performance of Dragoon about latency and throughput using online ID queries in terms of data update and query phases, separately. Figure 11 shows the update latency, update throughput, query latency, and query throughput on the three datasets when O_r ranges from 10 to 100%. We consider the Newest-Only, Share-Append, and Share-Update strategies for both data updates and ID queries, and compare with Spark Streaming.

In terms of data updates, the first observation is the update latency increases and update throughput decreases with the growth of O_r. That is because, as O_r increases, the size of the latest trajectory data generated by moving objects that needed to be merged with the historical trajectory data grows. Second, Spark Streaming achieves the best update efficiency, followed by the Share-Append, Share-Update,

Table 3 Scalability evaluation of Dragoon (ms)

Slave workers	ID Query			Range Query			kNN Query		
	GeoLife	Taxi	Brinkhoff	GeoLife	Taxi	Brinkhoff	GeoLife	Taxi	Brinkhoff
1	1.51	2.15	2.55	10.71	24.16	35.16	268.68	402.77	899.19
2	1.08	1.52	1.75	5.64	12.26	16.74	119.41	169.23	359.67
4	0.83	1.16	1.30	3.32	7.13	9.46	65.61	91.48	191.32
8	0.62	0.87	1.12	2.21	4.69	6.03	40.57	55.44	106.88
10	0.51	0.70	0.82	1.71	3.53	4.34	27.24	36.22	64.10

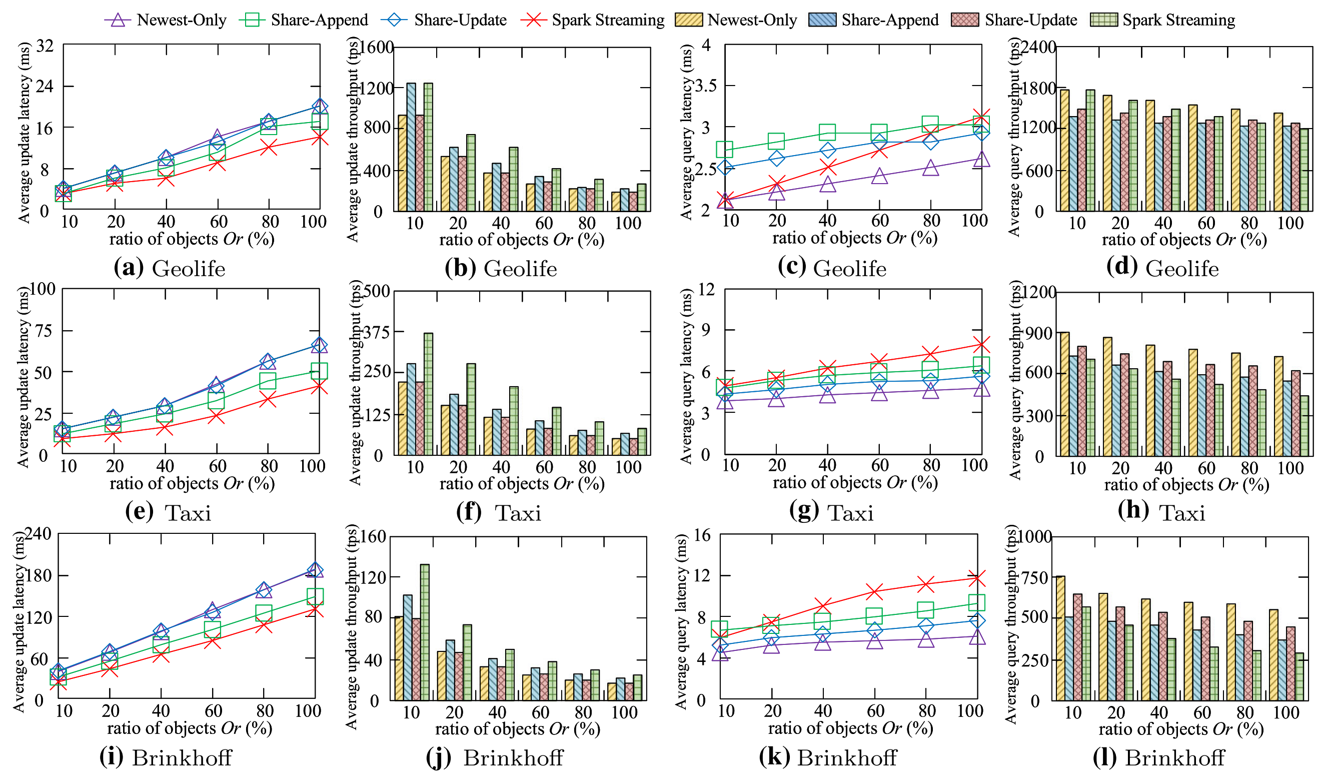

Fig. 11 Performance of online ID queries versus O_r

and Newest-Only. The Share-Append update performance is slightly lower than that of Spark Streaming. The reason is that the Share-Append needs additional time to serialize the trajectory data that need to be stored in the Chronicle Map, while Spark Streaming stores the data in the main memory directly. In addition, the update latencies and throughputs of Newest-Only and Share-Update are almost the same. This is because both of them handle data updates by using local Hash indexes. The difference is that the Newest-Only finds the corresponding historical trajectory data and then replaces its value, while the Share-Update finds the corresponding historical trajectory data and then appends data to it. However, the performance of both the Newest-Only and Share-Update are slightly slower than that of the Share-Append strategy due to the additional cost incurred by RDD Mirror that is needed to avoid data inconsistencies in the distributed environment of Dragoon.

For online ID queries, the first observation is that the latency increases and the throughput drops slightly as O_r grows. The reason is that the implementation of the ID query is based on a hash map, which means the performance of that is affected slightly by the trajectory data size. The second observation is that the Newest-Only strategy has the lowest query latency and the highest query throughput, followed by the Share-Update and Share-Append strategies, while Spark Streaming performs worst especially on the large datasets. This is because (i) Newest-Only only stores the latest locations of the moving objects, thus using only minimal storage; (ii) as discussed in Sect. 5.2, Share-Append divides the data according to the temporal information, while Share-Update and Newest-Only can achieve stronger ID filtering capabilities; and (iii) the bigger the data volume O_r, the larger data pressure on the Spark Streaming system.

Finally, we also evaluate the online update performance of mRDD in terms of memory occupation during the update phase of online ID query. The results are shown in Table 4, where N denotes Newest-Only, A stands for Share-Append, U denotes Share-Update, and S represents Spark Streaming. The first observation is that the memory occupation increases as O_r varies from 10 to 100. The second observation is that the Newest-Only strategy has the smallest memory occupation, since it only keeps the newest location points of moving objects in the system. Last but not least, both Share-Append and Share-Update occupy fewer system memory compared with that of Spark Streaming. The reason is that the underlying storage of Dragoon is based on the off-heap Chronicle Map, while Spark Streaming relies on the on-heap memory and JVM mechanism that lead to additional memory costs. Note that the memory occupation is related to data volumes and update strategies, but is not affected by the query types and data partitioning methods, so the memory costs of the online Range/kNN queries are omitted.

Table 4 Online updating performance of mRDD versus *memory occupation* (%)

O_r (%)	GeoLife				Taxi				Brinkhoff			
	N	A	U	S	N	A	U	S	N	A	U	S
10	0.05e−3	0.06	0.07	0.07	0.06e−3	3.49	3.37	4.17	0.33e−3	6.73	6.72	6.73
20	0.11e−3	0.13	0.14	0.14	0.13e−3	7.23	7.31	7.81	0.67e−3	12.11	12.23	13.14
40	0.23e−3	0.26	0.25	0.26	0.29e−3	15.08	15.26	16.99	1.43e−3	25.45	25.67	26.88
60	0.34e−3	0.41	0.42	0.41	0.40e−3	22.37	22.19	23.06	2.17e−3	37.88	37.34	38.15
80	0.44e−3	0.47	0.48	0.48	0.51e−3	28.83	28.67	30.15	2.79e−3	50.54	50.67	52.13
100	0.47e−3	0.53	0.54	0.55	0.59e−3	35.5	35.34	37.92	3.07e−3	65.46	65.53	67.78

8.3 Online range query

We proceed to explore the performance of Dragoon and compare it with Spark Streaming using online range queries on the Taxi and Brinkhoff datasets. Figure 12 plots the update and query latency/throughput for the two datasets when varying O_r from 10 to 100% and changing the query region area ϵ from 0.01 to 0.32%. Here, we only report the update latency and update throughput when varying O_r, as the update latency and throughput are not affected by ϵ. In addition, since we adopt the Grid index as the global index and the R-tree as the local index, we fix the grid cell side length to $\sqrt{0.08\%}$, and the R-tree node capacity to 64.

For data updates, the first observation is that the average update latency grows and the average throughput decreases with the increasing of O_r. The reason is that more recent spatial locations generated by moving objects need to be merged with historical trajectory data as the growth of O_r. Similar to the observations for the online ID queries, Spark Streaming has the best update performance, while Newest-Only and Share-Update have similar update efficiency, but both are slower than Share-Append strategy.

For online range queries, the first observation is that the query latency increases and the query throughput drops as O_r grows due to the larger search spaces. Second, the query latencies and throughputs of Dragoon using the three update strategies, Newest-Only, Share-Append, and Share-

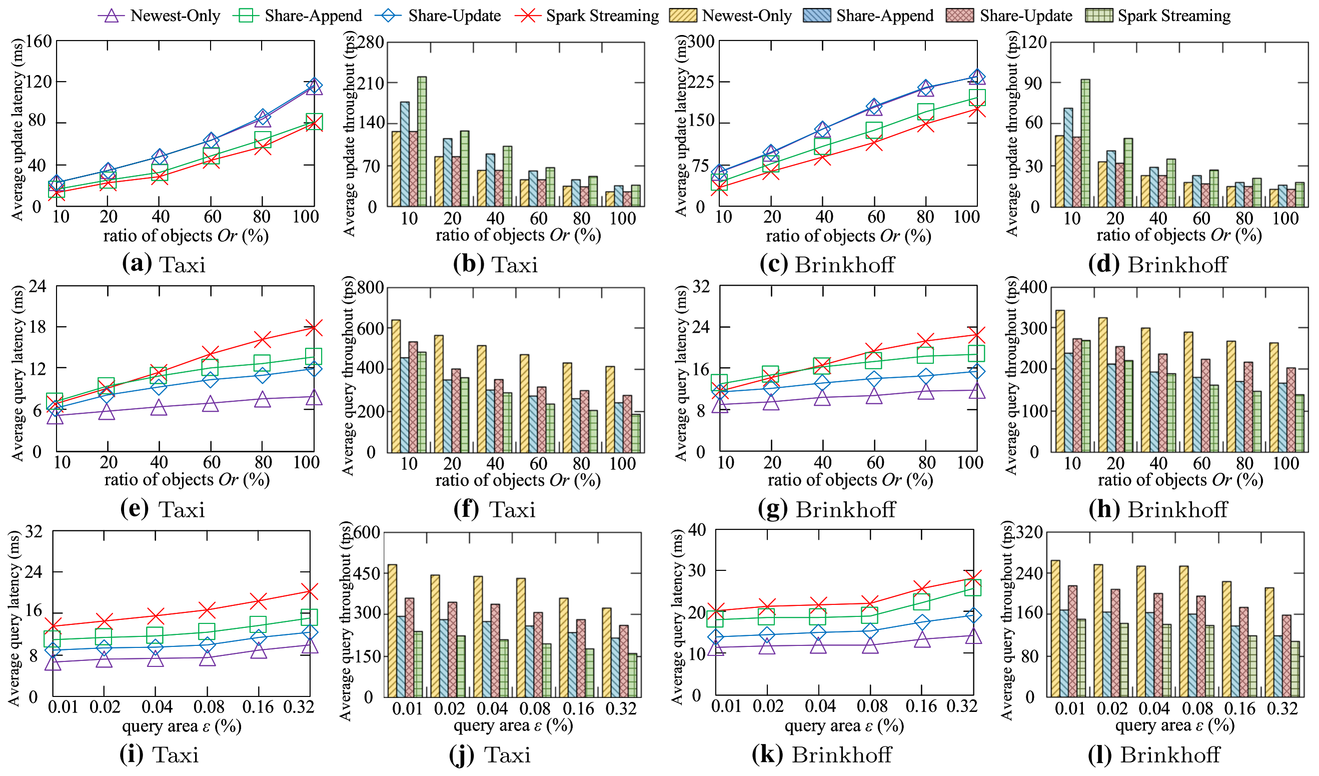

Fig. 12 Performance of online range queries versus O_r and ϵ

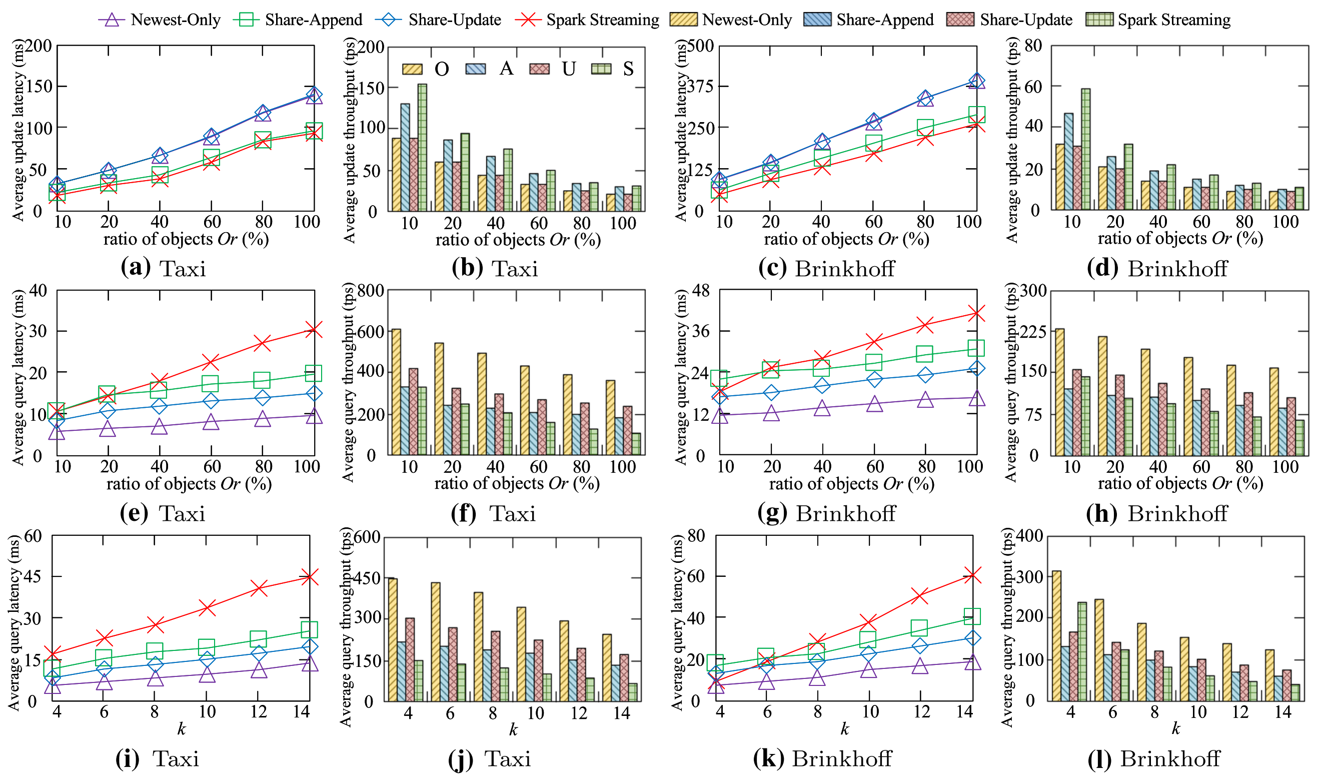

Fig. 13 Performance of online kNN queries versus O_r and k

Update, perform better than that of Spark Streaming on large datasets (i.e., when O_r exceeds 40%). This is because, during the online range queries, Spark Streaming suffers from the garbage collector (GC) pressure caused by the large big volumes of data. The third observation is that the query latency and throughput remain stable at first but then increases when ϵ exceeds 0.08%. This is because the grid side length is fixed to $\sqrt{0.08\%}$. When ϵ exceeds 0.08%, the search region size exceeds the area of each grid cell, and thus, it contains much more grid cells, resulting in the decrease of query efficiency in terms of latencies and throughputs.

Last but not the least, the query performance of the Newest-Only is the best, because it only maintains the latest values in the Dragoon system. In addition, the query performance of Share-Update is better than that of Share-Append. The reason behind is that Share-Update can use the Grid index to partition the data, i.e., Share-Update partition the trajectory data according to location information, which is more efficient for the online range queries, while Share-Append partitions data according to its temporal information, as discussed in Sect. 5.2.

8.4 Online *k*NN query

Furthermore, we proceed to investigate the performance of Dragoon on online kNN queries using the Taxi and Brinkhoff datasets. Figure 13 shows the update and query performances using latency and throughput performance metrics when varying O_r from 10 to 100% and changing k from 4 to 14. As for online range queries, parameter k does not affect the update performance, and thus, the update latency and throughput performance are also omitted when varying k.

For the data updates, the first observation is that the latency increases and the throughput decreases as O_r grows, as more new incoming trajectory data that are needed to be merged. The second observation is that Spark Streaming achieves the lowest update latency because its update operations are performed in main memory. As in previous experiments, Share-Append performs best for data updates, while Newest-Only and Share-Update have similar update performance.

For the online kNN queries, the first observation is that Dragoon using the three update strategies performs better about latencies and throughputs than that of Spark Streaming on large datasets (i.e., when $O_r \geq 40$) due to the high GC pressure in Spark Streaming. In other words, all Newest-Only, Share-Append, and Share-Update have a more stable query performance than Spark Streaming. The second observation is that the query latency increases, and the query throughput decreases with the growth of k due to larger search spaces. The third observation is that the Share-Update strategy has a better query performance than that of the Share-Append, as the Share-Update partitions the data spatially,

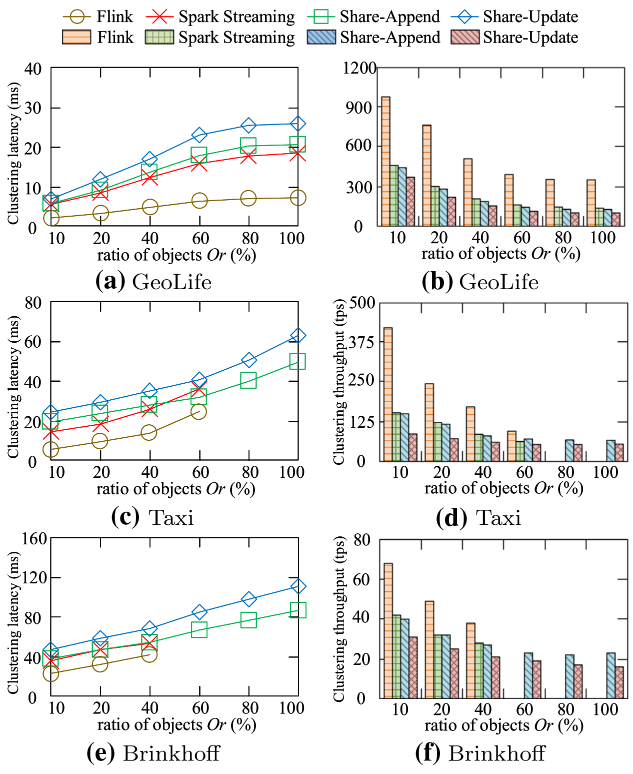

Fig. 14 Clustering performance of pattern detection

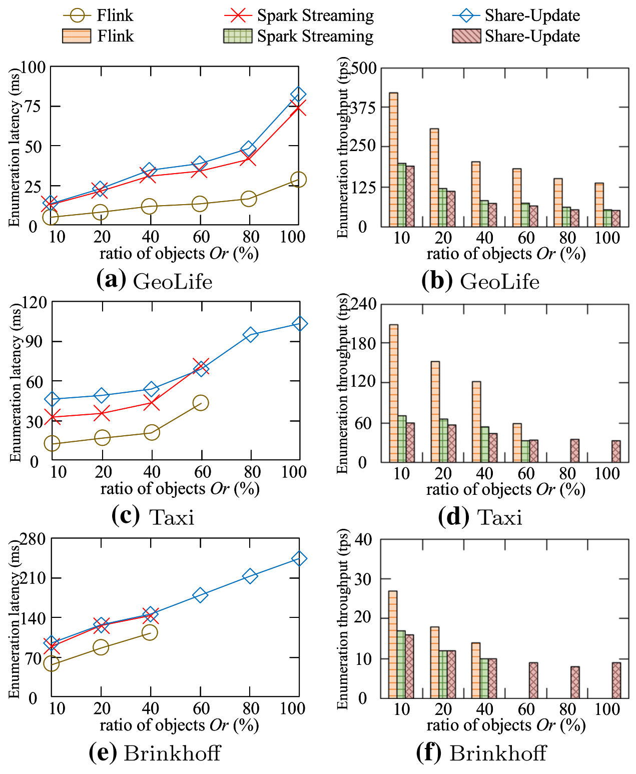

Fig. 15 Enumeration performance of pattern detection

which is more suitable for the online *k*NN queries. The last observation is that the query performance of Newest-Only is the best as only the latest locations of the moving objects are managed in the system, and the data storage is much lower than the two other update strategies (i.e., Share-Append and Share-Update). In other words, the search space is much smaller when compared to the other update strategies.

8.5 Co-movement pattern detection

Finally, we study the performance of Dragoon using real-time co-movement pattern detection while comparing it with general-purpose platforms Flink and Spark Streaming. We compare with these two representative general systems, because no unified online trajectory management and analytic system exist, and therefore, we directly adopt the latest work [14] on Flink and extend it to work with Spark Streaming as comparison methods. Figures 14 and 15 show the clustering and enumeration performance of pattern detection when varying O_r from 10 to 100%. In particular, the real-time co-movement pattern detection includes two stages, clustering and enumeration, where the data and indexes are updated during the clustering process to improve efficiency. In the clustering phase, we evaluate the Share-Append and Share-Update strategies, and compare them with Flink and Spark Streaming. Note that the Newest-Only strategy cannot be used for pattern detection since pattern detection needs historical trajectory data, while Newest-Only only keeps the latest data values. In the enumeration phase, Share-Update is used to compare with Flink and Spark Streaming. This is because Share-Append does not support ID partitioning [14], as discussed in Sect. 5.2.

Specifically, we use both the latency and throughput to evaluate the clustering and enumeration separately. The first observation is that Flink performs best. This is because Flink processes each tuple in real time, while Spark Streaming and Dragoon apply mini-batch semantics. However, Flink is unable to support pattern detection on the large Taxi dataset when $O_r \geq 60\%$ and the larger Brinkhoff dataset when $O_r \geq 40\%$. The reason is that the intermediate results are so large that they exceed the Flink system's capability, while Dragoon uses mutable RDDs based on Chronicle Map to reduce the reliance on main-memory storage. Second, Dragoon performs better than Spark Streaming on large datasets (i.e., Taxi with $O_r \geq 60\%$ and Brinkhoff with $O_r \geq 40\%$), but is slower than Spark Streaming on small datasets. This is because, on the one hand, Dragoon needs additional costs for storing data in Chronicle Map instead of putting trajectory data in the system's memory directly; on the other hand, Dragoon utilizes the mRDD model to reduce the pressure on the system's main-memory storage, especially when the dataset size is large. Similar to Flink, Spark Streaming cannot support real-time co-movement pattern detection on

Table 5 Workload of Dragoon versus min, max, and average location points ($\times 10^3$) in each data partition

Methods	GeoLife			Taxi			Brinkhoff		
	Min	Max	Average	Min	Max	Variance	Min	Max	Variance
IdPartitioning	9	1106	248	51	97,132	22,730	32,881	53,148	47,691
GridPartitioning	433	810	248	20,156	53,381	22,730	44,157	60,322	47,691
STRPartitioning	198	356	248	19,400	37,171	22,730	45,169	52,158	47,691

large datasets. The third observation is that the Share-Append strategy performs better than the Share-Update strategy during the clustering phase, as additional RDD Mirror cost is needed for Share-Update to avoid data inconsistency in the distributed environment. Finally, clustering and enumeration latency increases while clustering and enumeration throughput decrease when O_r grows due to larger search spaces.

To conclude, although Flink achieves the best efficiency for online co-movement pattern detection on streaming trajectories, Dragoon offers the best scalability performance. Overall, Dragoon gives consideration to both efficient and scalable management and analytics on big trajectory data.

8.6 System workload

To give deep insights into the workload performance of Dragoon, we conduct a set of experiments to study the system's workload balance considering different data partitioning methods. Specifically, we first count the number of trajectory location points in each data partition after all the points in the dataset are loaded and updated in Dragoon. Then, we calculate the min, max, and average trajectory points among all data partitions when Dragoon finishes merging all the arriving trajectory points to the historical trajectory dataset. Table 5 depicts the results under three different partitioning methods (i.e., IdPartitioning, GridPartitioning, and STRPartitioning). As observed, the data distribution with IdPartitioning is the most unbalanced. This is because IdPartitioning only considers the ID information of moving objects, while GridPartitioning and STRPartitioning consider the spatial information of trajectories with a more balanced data distribution. In addition, STRPartitioning performs better than GridPartitioning. The reason is that STRPartitioning divides the dataset according to the data distribution of newly arriving trajectory points in real time. It is worth noting that Dragoon can achieve better workload balance by repartitioning the data periodically.

8.7 Summary

Overall, we conclude that Dragoon is an efficient and scalable system for trajectory data management that is able to support both offline and online analytics. For offline analytics, Dragoon has a similar query performance as the state-of-the-art offline trajectory management system, UlTraMan. Nevertheless, Dragoon has better update performance than UlTraMan due to its mutable RDD model. For online analytics, Dragoon has better scalability than the general-purpose online processing systems, Spark Streaming and Flink. Moreover, Dragoon achieves better online ID/Range/kNN query performance than Spark Streaming on larger datasets. Moreover, when comparing the three proposed update strategies, Newest-Only strategy achieves the best performance for the latest online queries, while the Share-Update has the best flexibility for period online queries.

9 Conclusions

In this paper, we propose the Dragoon system, a new hybrid and efficient trajectory data management and analytic system. To support the management of both historical and streaming trajectories, Dragoon adopts the mRDD model, including the RDD Share, the RDD Update, and the RDD Mirror. The RDD Share is used to avoid unnecessary data copies for unchanged data blocks during data updates, while the RDD Update provides three update strategies, including Newest-Only, Share-Append, and Share-Update, that target different scenarios, and the RDD Mirror enables read/write controls to avoid data inconsistencies in a distributed environment. In addition, Dragoon's hybrid analysis pipeline offers support for both historical and streaming trajectories. Experimental studies on large real and synthetic datasets offer insight into the scalability and performance of Dragoon, and compare with state-of-the-art systems, yielding the following findings.

– For the historical trajectory data, Dragoon achieves similar performance as the existing trajectory system UlTraMan in terms of offline trajectory queries. However, Dragoon decreases up to doubled storage overhead during trajectory editing scenarios.
– For the streaming trajectory data, although existing general streaming systems Spark Streaming and Flink are capable of higher update efficiency for small workloads, Dragoon achieves at least 40% improvement of scalability for period online analytics and offers an average doubled performance improvement for latest online analytics.

– Share-Append achieves the best update efficiency, while the Newest-Only achieves the best query efficiency. However, Newest-Only is unsuitable for period online analyses, and Share-Append only supports temporal data partitioning.

In the future, it is of interest to apply Dragoon for the bigger trajectory data management and processing, and design more effective indexes to support additional types of trajectory data analysis (e.g., offline trajectory similarity computing and online sub-trajectory clustering). In addition, extending the proposed mRDD model for general big data management or developing another new dataset-enhanced model for big structured data analytics is also promising direction.

Acknowledgements This work was supported in part by the NSFC under Grant Nos. 62025206 and 61972338, the National Key R&D Program of China under Grant No. 2018YFB1004 003, and the NSFC-Zhejiang Joint Fund under Grant No. U1609217. Yunjun Gao is the corresponding author of the work.

References

1. Apache Hadoop. http://hadoop.apache.org/ (2008)
2. Apache Samza. http://samza.apache.org/ (2013)
3. Apache Flink. http://flink.apache.org/ (2014)
4. Apache Spark. http://spark.apache.org/ (2014)
5. Apache Storm. http://storm.apache.org/ (2014)
6. DiDi Brain. https://www.didiglobal.com/science/brain (2018)
7. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. PVLDB **2**(1), 922–933 (2009)
8. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB **8**(12), 1792–1803 (2015)
9. Ali, M., Chandramouli, B., Raman, B.S., Katibah, E.: Real-time spatio-temporal analytics using microsoft streaminsight. In: SIGSPATIAL, pp. 542–543 (2010)
10. Bao, J., Li, R., Yi, X., Zheng, Y.: Managing massive trajectories on the cloud. In: SIGSPATIAL, pp. 41:1–41:10 (2016)
11. Boykin, P.O., Ritchie, S., O'Connell, I., Lin, J.J.: Summingbird: a framework for integrating batch and online MapReduce computations. PVLDB **7**(13), 1441–1451 (2014)
12. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
13. Brunsdon, C., Zheng, Y., Zhou, X.: Computing with spatial trajectories. IJGIS **27**(1), 208–209 (2013)
14. Chen, L., Gao, Y., Fang, Z., Miao, X., Jensen, C.S., Guo, C.: Real-time distributed co-movement pattern detection on streaming trajectories. PVLDB **12**(10), 1208–1220 (2019)
15. Cho, H., Shiokawa, H., Kitagawa, H.: JsFlow: integration of massive streams and batches via JSON-based dataflow algebra. In: NBIS, pp. 188–195 (2016)
16. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: NSDI, pp. 313–328 (2010)
17. Cudré-Mauroux, P., Wu, E., Madden, S.: TrajStore: an adaptive storage system for very large trajectory data sets. In: ICDE, pp. 109–120 (2010)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
19. DeWitt, D.J., Halverson, A., Nehme, R.V., Shankar, S., Aguilar-Saborit, J., Avanes, A., Flasza, M., Gramling, J.: Split query processing in polybase. In: SIGMOD, pp. 1255–1266 (2013)
20. Ding, X., Chen, L., Gao, Y., Jensen, C.S., Bao, H.: UlTraMan: a unified platform for big trajectory data management and analytics. PVLDB **11**(7), 787–799 (2018)
21. Düntgen, C., Behr, T., Güting, R.H.: BerlinMOD: a benchmark for moving object databases. VLDB J. **18**(6), 1335–1368 (2009)
22. Ge, Y., Xiong, H., Zhou, Z., Ozdemir, H.T., Yu, J., Lee, K.C.: Top-eye: top-*k* evolving trajectory outlier detection. In: CIKM, pp. 1733–1736 (2010)
23. Gudmundsson, J., Laube, P., Wolle, T.: Computational Movement Analysis, pp. 423–438. Springer, Berlin (2012)
24. Hasani, Z., Kon-Popovska, M., Velinov, G.: Lambda architecture for real time big data analytic. In: ICT Innovations, pp. 133–143 (2014)
25. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter Heron: stream processing at scale. In: SIGMOD, pp. 239–250 (2015)
26. Kumar, V., Andrade, H., Gedik, B., Wu, K.: DEDUCE: at the intersection of MapReduce and stream processing. In: EDBT, pp. 657–662 (2010)
27. Leutenegger, S.T., Lopez, M.A., Edgington, J.: STR: a simple and efficient algorithm for R-tree packing. In: ICDE, pp. 497–506 (1997)
28. Li, R., He, H., Wang, R., Huang, Y., Liu, J., Ruan, S., He, T., Bao, J., Zheng, Y.: Just: Jd urban spatio-temporal data engine. ICDE (2020)
29. Li, R., He, H., Wang, R., Ruan, S., Sui, Y., Bao, J., Zheng, Y.: Trajmesa: a distributed nosql storage engine for big trajectory data. ICDE (2020)
30. Li, R., Ruan, S., Bao, J., Li, Y., Wu, Y., Zheng, Y.: Querying massive trajectories by path on the cloud. In: SIGSPATIAL, pp. 77:1–77:4 (2017)
31. Li, Z., Han, J., Ji, M., Tang, L., Yu, Y., Ding, B., Lee, J., Kays, R.: Movemine: mining moving object data for discovery of animal movement patterns. TIST **2**(4), 37:1–37:32 (2011)
32. Ma, S., Zheng, Y., Wolfson, O.: Real-time city-scale taxi ridesharing. TKDE **27**(7), 1782–1795 (2015)
33. Mahmood, A.R., Punni, S., Aref, W.G.: Spatio-temporal access methods: a survey (2010–2017). GeoInformatica **23**(1), 1–36 (2019)
34. Patroumpas, K., Kefallinou, E., Sellis, T.: Monitoring continuous queries over streaming locations. In: SIGSPATIAL, pp. 41:1–41:10 (2008)
35. Patroumpas, K., Pelekis, N., Theodoridis, Y.: On-the-fly mobility event detection over aircraft trajectories. In: SIGSPATIAL, pp. 259–268. ACM (2018)
36. Ruan, S., Li, R., Bao, J., He, T., Zheng, Y.: Cloudtp: a cloud-based flexible trajectory preprocessing framework. In: ICDE, pp. 1601–1604 (2018)
37. Salmon, L., Ray, C.: Design principles of a stream-based framework for mobility analysis. GeoInformatica **21**(2), 237–261 (2017)
38. Shang, Z., Li, G., Bao, Z.: DITA: distributed in-memory trajectory analytics. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) SIGMOD, pp. 725–740 (2018)
39. Tan, H., Luo, W., Ni, L.M.: CloST: a hadoop-based storage system for big spatio-temporal data analytics. In: CIKM, pp. 2139–2143 (2012)

40. Tang, M., Yu, Y., Malluhi, Q.M., Ouzzani, M., Aref, W.G.: Locationspark: a distributed in-memory data management system for big spatial data. PVLDB **9**(13), 1565–1568 (2016)
41. Tao, Y., Papadias, D.: MV3R-tree: a spatio-temporal access method for timestamp and interval queries. In: VLDB, pp. 431–440 (2001)
42. Wang, H., Zheng, K., Xu, J., Zheng, B., Zhou, X., Sadiq, S.W.: Sharkdb: an in-memory column-oriented trajectory storage. In: CIKM, pp. 1409–1418 (2014)
43. Wang, L., Cai, R., Fu, T.Z., He, J., Lu, Z., Winslett, M., Zhang, Z.: Waterwheel: realtime indexing and temporal range query processing over massive data streams. In: ICDE, pp. 269–280 (2018)
44. Wang, W., Yang, J., Muntz, R.R.: STING: a statistical information grid approach to spatial data mining. In: PVLDB, pp. 186–195 (1997)
45. Wang, Y., Zheng, Y., Xue, Y.: Travel time estimation of a path using sparse trajectories. In: SIGKDD, pp. 25–34 (2014)
46. Xie, D., Li, F., Phillips, J.M.: Distributed trajectory similarity search. VLDB **10**(11), 1478–1489 (2017)
47. Xie, D., Li, F., Yao, B., Li, G., Zhou, L., Guo, M.: Simba: efficient in-memory spatial analytics. In: SIGMOD, pp. 1071–1085 (2016)
48. Xie, X., Mei, B., Chen, J., Du, X., Jensen, C.S.: Elite: an elastic infrastructure for big spatiotemporal trajectories. VLDB J. **25**(4), 473–493 (2016)
49. Xu, W., Zhou, K., Yu, Y., Tan, Q., Peng, Q., Guo, B.: Gradient domain editing of deforming mesh sequences. ACM Trans. Graph. **26**(3), 84 (2007)
50. Yang, F., Merlino, G., Ray, N., Léauté, X., Gupta, H., Tschetter, E.: The RADStack: open source lambda architecture for interactive analytics. In: HICSS, pp. 1703–1712 (2017)
51. Yu, L., Yu, J., Zhang, M., Zhang, X., Liu, Y., Zhang, H., Min, W.: Large scale traffic signal network optimization: a paradigm shift driven by big data. In: ICDE, pp. 1832–1840 (2019)
52. Yuan, H., Li, G.: Distributed in-memory trajectory similarity search and join on road network. In: ICDE, pp. 1262–1273 (2019)
53. Yuan, J., Zheng, Y., Xie, X.: Discovering regions of different functions in a city using human mobility and POIs. In: SIGKDD, pp. 186–194 (2012)
54. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)
55. Zhan, X., Zheng, Y., Yi, X., Ukkusuri, S.V.: Citywide traffic volume estimation using trajectory data. TKDE **29**(2), 272–285 (2017)
56. Zhang, M., Wo, T., Lin, X., Xie, T., Liu, Y.: Carstream: an industrial system of big data processing for internet-of-vehicles. PVLDB **10**(12), 1766–1777 (2017)
57. Zheng, Y.: Trajectory data mining: an overview. TIST **6**(3), 29:1–29:41 (2015)
58. Zheng, Y., Capra, L., Wolfson, O., Yang, H.: Urban Computing: Concepts, Methodologies, and Applications. TIST **5**(3), 38:1–38:55 (2014)