UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics

Xin Ding[*,†] Lu Chen[‡] Yunjun Gao[*,†] Christian S. Jensen[‡] Hujun Bao[*,†]

[*]State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou, China
[†]College of Computer Science, Zhejiang University, Hangzhou, China
[‡]Department of Computer Science, Aalborg University, Aalborg, Denmark
{dingxin@, gaoyj@, bao@cad.}zju.edu.cn {luchen, csj}@cs.aau.dk

ABSTRACT

Massive trajectory data is being generated by GPS-equipped devices, such as cars and mobile phones, which is used increasingly in transportation, location-based services, and urban computing. As a result, a variety of methods have been proposed for trajectory data management and analytics. However, traditional systems and methods are usually designed for very specific data management or analytics needs, which forces users to stitch together heterogeneous systems to analyze trajectory data in an inefficient manner. Targeting the overall data pipeline of big trajectory data management and analytics, we present a unified platform, termed as *UlTraMan*. In order to achieve *scalability*, *efficiency*, *persistence*, and *flexibility*, (i) we extend Apache Spark with respect to both data *storage* and *computing* by seamlessly integrating a key-value store, and (ii) we enhance the MapReduce paradigm to allow flexible optimizations based on random data access. We study the resulting system's flexibility using case studies on data retrieval, aggregation analyses, and pattern mining. Extensive experiments on real and synthetic trajectory data are reported to offer insight into the scalability and performance of UlTraMan.

PVLDB Reference Format:
Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *PVLDB*, 11(7): 787-799, 2018.
DOI: https://doi.org/10.14778/3192965.3192970

1. INTRODUCTION

With the proliferation of GPS-equipped devices, increasingly massive volumes of trajectory data that captures the movements of humans, vehicles, and animals has been collected. This data is used widely in transportation [36], location-based services [43], animal behavior studies [28], and urban computing [42], to name but a few application areas. Systems that are used to manage and analyze trajectory data are important not only in scientific studies, but also in real-world applications. As an example, DiDi, the largest ride-sharing company in China, utilizes trajectory data to provide services such as travel time prediction, demand forecasting, and carpool scheduling [6]. The explosive increase in data volumes and the rapid proliferation of new data analysis methods expose three shortcomings of traditional trajectory data management platforms.

Proceedings of the VLDB Endowment, Vol. 11, No. 7

DOI: https://doi.org/10.14778/3192965.3192970

First, in real-life applications, trajectory data is collected at a rapid pace. For instance, the Daisy Lab at Aalborg University currently receives some 100 million points per day from about 40,000 vehicles in Denmark. Consequently, traditional centralized systems [13, 23, 35] are or will be inefficient at or incapable of managing and processing real-world trajectory data. In recent years, various systems [17, 37] based on MapReduce [14] have also been proposed for spatial data analytics, which have been shown to be scalable in cloud environments and to be more efficient than centralized systems. However, those systems are designed for generic *spatial* data and cannot fully exploit techniques for *spatio-temporal* trajectory data. Further, they intrinsically target data analysis and hence are suboptimal for data management tasks. Therefore, it remains a challenge to develop a *scalable* and *efficient* platform architecture for managing and processing big trajectory data.

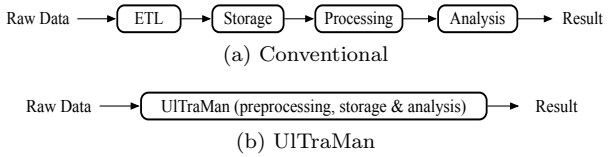


Figure 1: Trajectory data analysis pipelines.

Second, existing solutions for trajectory data analytics are composed of heterogeneous systems for data storage, data processing, and data analysis. For example, as depicted in Fig 1(a), a user may need one system to *extract*, *transform*, and *load* (ETL) the raw dataset [40], and may need other systems for data *storage* [9, 38], data *processing* [25, 39], and data *analysis* [19]. Thus, several systems are needed to accomplish even a simple trajectory analytics task. The resulting heterogeneous workflows are suboptimal: (i) the dataset has to be transferred across multiple system boundaries, involving expensive network traffic as well as costs of serialization and format transformation; (ii) additional expertise is needed to maintain and operate different systems with disparate data models, APIs, and internal implementations; and (iii) many potential optimization techniques can-

not be applied across multiple storage systems and analysis systems. Consequently, a holistic solution, as illustrated in Fig. 1(b), that supports the full pipeline of data management, processing, and analysis is an important step towards applications that involve massive trajectory data. Again, a new system architecture is called for.

Last but not least, in real analysis scenarios involving trajectory data, multiple *data formats* (e.g., points [12], segments [25]), *index structures* (e.g., TB-tree [31], TPR-tree [33]) and *processing techniques* (e.g., segmentation [25], map-matching [39]) are needed in different applications. The diversity of techniques calls for an underlying system that is flexible in two main respects: (i) *pluggable system components*, meaning that the system should be extensible to the adoption of new components such as index structures and analysis algorithms; and (ii) *customizable processing pipelines*, meaning that the system should make it possible for its users to design particular data preprocessing and analysis pipelines. The needs for flexibility challenge the system design in terms of both architecture modules and APIs.

Aiming to achieve an efficient and holistic solution that meets the needs just described, we present *UlTraMan*, a unified platform for big trajectory data management and analytics. UlTraMan achieves its design goals by (i) adopting a unified storage and computing engine and by (ii) proposing an enhanced distributed computing paradigm based on MapReduce with flexible application interfaces.

UlTraMan's unified engine is built on Apache Spark [40], a popular distributed computing framework. Spark enables high performance distributed computing, but is, however, in itself suboptimal for big trajectory data management due to (i) a lack of indexing mechanisms, (ii) limited and inefficient runtime data persistence, and (iii) its significant pressure on the JVM garbage collector. Hence, UlTraMan integrates Chronicle Map [7], an embedded key-value store, into the internal block manager of Spark. This way, UlTraMan provides efficient data processing and reliable data management.

Based on the unified storage and computing engine, the overall process of trajectory analytics can be pipelined in UlTraMan without unnecessary data copying and serialization. The system enables optimizations across storage, processing, and analysis. For instance, the data organization and index structures can be changed according to analysis requirements, and computing tasks can be scheduled according to the data distribution.

Moreover, with the help of the unified engine, we enhance MapReduce to become a more powerful and flexible distributed computing paradigm. MapReduce [14] and Spark's Resilient Distributed Datasets (RDD) [40] adopt the functional programming concept, which facilitates *sequential* operations on the data. In contrast, many important techniques and optimizations are realized based on *random data access*, such as hash-maps and indexes. Motivated by this, we improve MapReduce and integrate an abstraction called *TrajDataset* into UlTraMan. This abstraction enables random access at both local and global levels. At the local level, data in each partition can be accessed randomly with the help of new programming interfaces provided by the unified engine; at the global level, data partitions are explicitly managed by two types of data generalization: (i) *global indexes* maintained in a driver node to organize small partition features, and (ii) *meta tables* distributed in executors to manage large features. As a result, we provide a computing paradigm in UlTraMan that is compatible with MapReduce while providing possibilities for optimization techniques.

Based on the unified engine and the TrajDataset abstraction, UlTraMan is able to serve as an efficient and flexible platform for an open-ended range of trajectory data management and analytics techniques. The contributions of UlTraMan are summarized as follows:

- UlTraMan offers an innovative and holistic solution for big trajectory data management and analytics, based on a novel unified engine.
- UlTraMan adopts an enhanced MapReduce paradigm that enables efficient and flexible management and analysis of massive trajectory data .
- UlTraMan provides flexible application interfaces to support customizable data organization, preprocessing, and analysis pipelines. Typical analytics cases, including data retrieval, aggregation analysis, and pattern mining, are studied and implemented to demonstrate the flexibility of UlTraMan.
- Extensive experiments on real and synthetic massive trajectory data are covered to offer insight into the efficiency and scalability of UlTraMan, in comparison to existing state-of-the-art methods.

The rest of this paper is organized as follows. Section 2 introduces necessary background. Section 3 provides a systematic overview of UlTraMan. Section 4 then elaborates on the design and implementation of UlTraMan's unified engine. Section 5 describes the TrajDataset API and the operations it provides. Case studies are presented in Section 6, and experimental results are reported in Section 7. Section 8 reviews related work, and Section 9 concludes the paper and offers research directions.

2. BACKGROUND

The design of UlTraMan represents a cross-domain study covering distributed systems, data management, and trajectory data analytics. Consequently, we cover the background of the design of UlTraMan in these three domains.

2.1 Apache Spark

Spark [40] is a general purpose distributed computing framework for tackling big data problems, which has attracted substantial interests in both academia and industry. In Spark, distributed computing tasks are submitted through an abstraction called *Resilient Distributed Datasets (RDD)* and are physically conducted by *executors*. A Spark executor is a JVM process running on a worker node, which internally uses a *block manager* to manage its cached data splits. If a task fails and a split is lost, Spark traces back the task lineage until cached data or the data source is found, and then it schedules recomputation tasks to recover the data.

Taking advantage of in-memory caching and recomputation-based fault tolerance, Spark enables high performance distributed computing. However, Spark's design trade-offs also introduce shortcomings in relation to big data management. First, massive on-heap caching may produce significant pressure on the garbage collector (GC), thus resulting

in unexpected overhead. Second, recomputation is expensive for some data management tasks, such as repartition and index construction.

Spark SQL [8] is a potential interface for big data management in Spark. It manages *structured data* through a new DataFrame API, which enables off-heap storage to save in-memory space and to relieve GC pressure at the same time. However, non-trivial internal modifications are needed to support indexing in Spark SQL [37], due to the limited functionalities for data management. In addition, the structured storage scheme cannot provide the flexibility needed for supporting different trajectory data formats (e.g., sub-trajectories). Therefore, Spark SQL is limited to serve as a unified platform for big trajectory data management and analytics.

2.2 Chronicle Map

Chronicle Map [7] is an in-memory, embedded key-value store designed for low-latency and multiple-process access, which is popular in the area of high frequency trading (HFT).

Chronicle Map is chosen among many storage techniques, as it enables meeting the design goal of the proposed system, namely, *efficiency*, *flexibility*, and *persistence* , as explained next. (i) Efficiency: to achieve high performance data access that is comparable to Spark's built-in cache, we need an in-memory storage that has a *Java-compatible* implementation so that cross-language transformation is avoided; we also need the store to be *embedded* in Spark executors to avoid both network transfer and inter-process communication. (ii) Flexibility: To support different trajectory data formats, flexible storage structures and random access are needed. (iii) Persistence: Data should be persisted at runtime to support efficient failure recovery. Distributed stores (e.g., Apache Ignite [4]), stand-alone stores (e.g., Redis [2]), or conventional database engines (e.g., H2 [1]) do not satisfy the needs stated above. Thus, they are not adopted in UlTraMan.

To the best of our knowledge, this is the first study of integrating Chronicle Map with Spark, and Chronicle Map is the only existing technique that meets our requirements. Specifically, the key-value store instance is embedded in the block manager to provide efficient data access. Furthermore, data is stored in off-heap memory to relieve GC pressure, and the data is persisted at runtime through the support of simultaneous access from multiple processes. By seamlessly integrating Chronicle Map and Spark in the underlying engine, UlTraMan is convenient, efficient, and reliable, for both users and developers.

2.3 Trajectory Data

Trajectory data is generated by sampling spatio-temporal locations of moving objects. In real applications, trajectories can be represented in a variety of formats, such as points [12], segments [25], and sub-trajectories [32]. For convenience, we use a generalized notation called *element* to represent trajectory data according to the context.

A broad range of methods for querying and analyzing trajectories have been proposed over the past decades, and new ones are being proposed. For example, multiple trajectory queries, including range queries [31] and k nearest neighbor (kNN) queries [20], are supported by different indexing and query processing methods; and a range of trajectory data mining techniques are also explored [41].

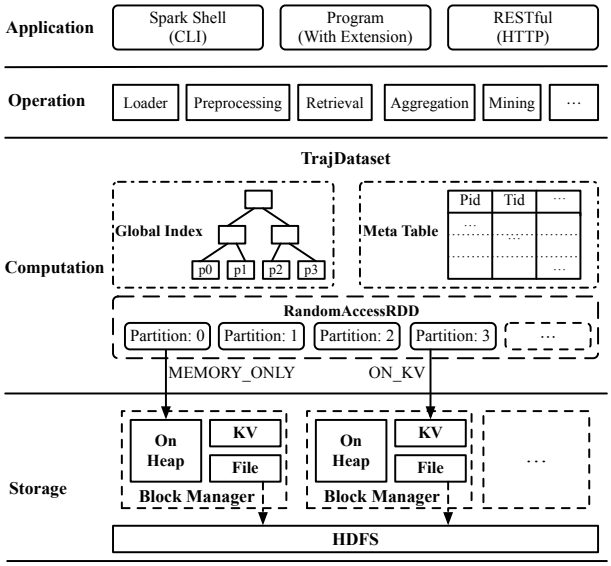


Figure 2: The architecture of UlTraMan.

Further, a wide variety of index structures and preprocessing techniques have been proposed to improve the efficiency of trajectory data analytics. Specific index structures contain TB-tree [31], TPR-tree [33], TrajTree [32], etc. Typical preprocessing techniques encompass *segmentation* [25], *synchronization* [19], *compression* [15], and *map-matching* [39]. These techniques can only be conducted through the use of specific systems, and there is a lack of a single holistic and flexible solution.

Along with the specific techniques, some systems have also been designed and built. COMPRESS [23], Elite [38], SharkDB [35], and TrajStore [13] are developed to manage trajectory data using innovative storage scheme. Implementations of specific tasks such as clustering [25] and pattern mining [19] are also reported in the literature.

Since existing indexing, preprocessing, querying, and mining techniques are designed for specific scenarios, they also rely on different data models, procedures, and optimization methods. Existing systems fail to provide a holistic solution for the full pipeline of trajectory data management and analytics, thereby leading to inefficient heterogeneous solutions. Based on this state of affairs, we develop UlTraMan, aiming to offer the flexibility needed to accommodate a variety of techniques and to support a broad range of pipelines on trajectory data management and analysis pipelines.

3. SYSTEM OVERVIEW

In this section, we provide an overview of the system architecture and data pipeline of UlTraMan.

3.1 System Architecture

Physically, UlTraMan adopts the master-slave architecture, which consists of one driver node and multiple worker nodes. The driver node is responsible for task scheduling, while data storage and computing are distributed across worker nodes. Conceptually, as illustrated in Fig. 2, the system architecture contains four layers, namely storage, computation, operation, and application. Among these layers,

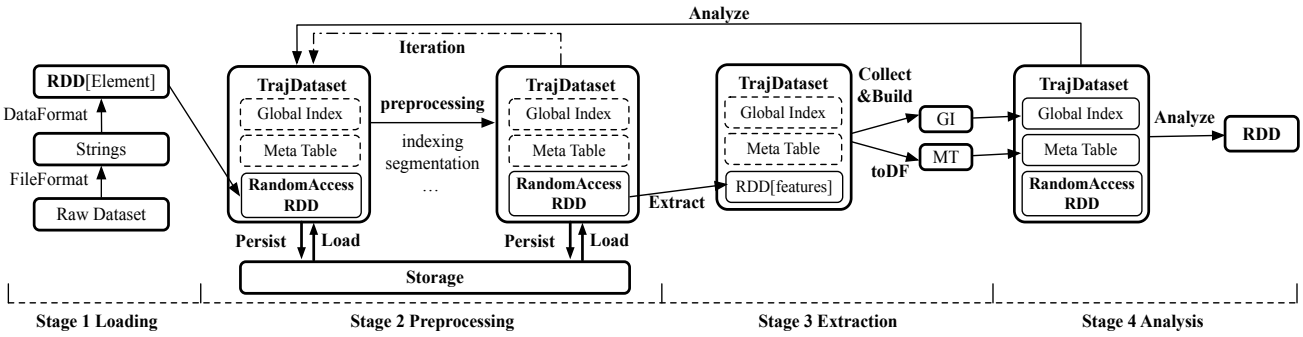


Figure 3: The data pipeline in UlTraMan.

the *storage* and *computation* layers form UlTraMan's underlying unified engine, the *operation* layer realizes reusable components to the developers, and the *application* layer provides interfaces to end users. We proceed to describe each layer in detail.

Storage Layer. All data as well as indexes are managed in the storage layer via the extended block manager. When an RDD is cached, the block manager on each executor stores its data partitions in on-heap arrays or off-heap Chronicle Map instances, according to the storage level assigned by users. Based on this storage engine, both on-heap and off-heap data can be accessed randomly, enabling optimizations on the upper layers.

Computation Layer. The computation layer supports the distributed computing paradigm by using the abstraction of `TrajDataset`. Being compatible with MapReduce and RDDs, the TrajDataset implementation is composed of an extended RDD type called `RandomAccessRDD` and two global structures: global index and meta table. Locally, the `RandomAccessRDD` takes advantage of the storage layer to enable random data access on cached data. Globally, all data partitions are generalized to specific features in global indexes and meta tables, so that distributed tasks can be scheduled at specific partitions.

Operation Layer. UlTraMan offers programming interfaces in the operation layer. Trajectory data is managed and processed through operations on TrajDatasets, including loading, filtering, mapping, repartition, etc. Advanced data processing and analysis techniques can be supported based on the interfaces. To verify the system flexibility, we have implemented several modules in UlTraMan, including a csv file loader, segmentation, range querying and clustering.

Application Layer. To serve as a platform, UlTraMan supports multiple methods of interaction as its Application Layer. Most simply and interactively, users can run data management and analysis tasks through the Spark shell. Advanced developers and users are also able to submit jobs via programs along with customized modules. Finally, UlTraMan is packed with an HTTP server to answer web requests and to support frontend visualization.

In summary, the architecture of UlTraMan offers three main innovations: (i) it equips the computing engine with a reliable storage capability to support unified data processing and management; (ii) it enables a new computing model to support scalable global-local random access; and (iii) it supports complex trajectory data analytics by extensible and reusable modules and techniques.

3.2 Data Pipeline

With the help of its flexible system architecture, UlTraMan is customizable in terms of both system modules and processing pipelines. The data pipeline for a typical analysis task in UlTraMan includes four stages, as depicted in Fig. 3.

Stage 1: loading. In the first stage, UlTraMan loads a raw trajectory dataset and extracts trajectory elements to the storage layer. The raw dataset is usually stored in HDFS, so that it can be loaded in parallel. A customizable data loader is provided to support different file formats (e.g., csv or xml) and data formats (e.g., points or segments).

Stage 2: preprocessing. The second stage includes flexible procedures that are applied before storing and analyzing the trajectory data, such as format transformation and segmentation. In addition, some data management tasks (such as index construction) are also carried out at this stage. Having been properly processed, a dataset is persisted in UlTraMan, in order to support efficient analysis and fast failure recovery.

Stage 3: extraction. Extraction is a special stage that serves to facilitate analysis of a TrajDataset. Since TrajDataset enables global scheduling based on global indexes and meta tables, we need to extract and build the global information before analysis. For instance, to build a global R-tree, we need to extract and collect the features of *pid* and *mbr* from each partition, where *pid* is the partition ID and *mbr* is the bounding box of the partition. Then, a global R-tree is built on the collected features.

Stage 4: analysis. Since an open-ended range of analysis scenarios exist for trajectory data, UlTraMan is designed to serve as a platform that can support most (if not all) of them. To accomplish a complex task, the stages may need to be applied iteratively. As an example, in order to perform co-movement pattern mining (to be discussed in Section 6.5), the preprocessing and analysis stages are both conducted twice.

4. THE UNIFIED ENGINE

We proceed to offer details on the design and implementation of UlTraMan's unified storage and computing engine. The engine is realized based on Spark, which is designed for high performance distributed computing. Our goal is to enhance Spark's data management capabilities while retaining its computing performance. We introduce the enhancement in terms of random access support, customizable serialization, indexing support, and runtime persistence.

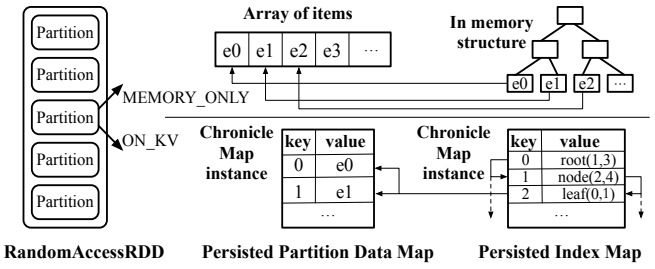


Figure 4: Data management in UlTraMan.

4.1 Random Access

The support for random access is fundamental to a functional data management engine, as without random access, operations have to rely on *brute force* scans on the dataset, resulting in many missed optimizations opportunities.

To support random access in Spark, we first investigate its internal storage models. For RDDs that are not cached, the data partitions are computed lazily for use, and are discarded immediately after use and thus cannot be accessed any more. For cached RDDs, several *storage levels*[1] are supported by the block manager. However, since Spark serializes data sequentially in an output stream without position information, the serialized data can only be read and deserialized sequentially, and hence, only the fully deserialized level, called `MEMORY_ONLY`, potentially supports random access. In fact, in the `MEMORY_ONLY` level, a data partition is stored in an array, and individual items can thus be randomly accessed through the array index.

Caching all data in an on-heap array is efficient for computation, but it may incur unexpected GC overhead; and more importantly, it suffers from data loss if a task crashes the executor (e.g, through an *out of memory* exception). To address this, we introduce a new storage level `ON_KV` through the modified block manager. `ON_KV` can be used in the same way as the original storage levels. When a data partition is about to persist in this new level, the block manager creates a Chronicle Map instance with integers as key and the data items as values. The items are then put in the KV store with an incremental counter as keys, as shown in Fig. 4. Hence, random access is supported by using the mapped key. Moreover, to support Spark's original access approach, data in Chronicle Map can be accessed sequentially through iterations over the map entries. Taking advantage of Chronicle Map's features, which are described in Section 2.2, the data is stored in off-heap memory and is persisted outside the executors. We detail our improvements on serialization and persistence later in this section.

Finally, a derived API called RandomAccessRDD is provided by UlTraMan to support random access on the data persisted at these two levels, namely `MEMORY_ONLY` and `ON_KV`.

4.2 Serialization

Data serialization is unavoidable for an off-heap and persisted store. Chronicle Map executes serialization and deserialization automatically when an item is put or retrieved. Therefore, data can be viewed by *users* as always being deserialized. However, how to serialize data in Chronicle Map is determined by the *system designer*. The serialization is critical to both data retrieval performance and the persisted data size. In our experiments, we find that properly optimized serialization can yield 5x faster data access as well as 10x smaller persisted data sizes.

The original Chronicle Map uses the java serializer for generic data types. In UlTraMan, to take advantage of Spark's optimizations, we adopt Spark's serializer as the default implementation for data persisted at the `ON_KV` level.

Nevertheless, there is space for further optimization. For generic data types, the type is unknown until run time (as abstract classes or interfaces can be used at compile time), and thus a general serializer should write the full class name along with the real data. When the data is deserialized, the type instance is constructed according to the full class name, which involves expensive reflection operations. Since UlTraMan is designed for trajectory data and the data format is usually determined at compile time, we can bind each format with a specific serializer, and the serializer can be detected and used by Chronicle Map at run time in order to save computation cost and persisted data size. This technique is also available for user-defined trajectory formats.

4.3 Local Indexing

To take full advantage of the random access capabilities, UlTraMan provides a flexible mechanism for indexing on each executor. Three considerations underlie the design of this mechanism: (i) the indexes are used to accelerate in-memory data access and thus should be in-memory; (ii) the index structures should be highly customizable to support diverse indexing techniques; and (iii) since index construction is expensive, the indexes should be persisted by the same way as the dataset. To realize the mechanism, we introduce an *index manager* in Spark to work with the block manager. It manages the process of index construction, index persistence, and index-based data access according to the storage level of the dataset. Users are allowed to implement specific *index constructors* and *queriers* to support their index structures, to be discussed in Section 5.

If the dataset is persisted in `MEMORY_ONLY`, an index constructor takes the cached item array as input and builds an in-memory index on the array. Since the items can be accessed directly through Java references without data copying, the constructed index can be used as a *primary index*. Indexes on this kind are cached on-heap after construction and can be fetched or released via the index manager.

To support indexing of a dataset persisted at level `ON_KV`, the index should be persisted and accessed through Chronicle Map as well, and thus the index structures should be realized in a map-like fashion. Consider the R-tree [22] as a typical example of a tree-structured index, as depicted in Fig. 4. We use a map of (*nid*, *node*) pairs to represent an R-tree, where *nid* is an ID for each tree *node*. The ID of the root node is hard-coded to 0. Hence, if 0 is not found in the map, the index is empty. In a non-leaf node, the children are stored as an array of node IDs and minimum bounding rectangles (MBRs, omitted in the figure). A leaf node stores the keys of the items in the *data map* to enable access to the data. As a result, the constructed indexes are managed and used in the same way as the dataset.

The indexes in Chronicle Map are used as *secondary indexes* because data is accessed indirectly through the keys in a data map. With the help of customizable index struc-

[1]https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

tures and query algorithms, this can be further optimized. For example, when each element is small (e.g. a point), it is reasonable to store the data directly in the leaf nodes in the index map, thus saving the cost of a `get` operation for each data access.

4.4 Runtime Persistence & Fault Tolerance

Spark offers multiple levels of data persistence, such as *task*, *process*, and *node*. If an RDD is not cached, it can be taken as being persisted at the *task* level, meaning that the data is lost if a task fails. A cached RDD, either in memory or on disk, is persisted at the *process* level, since the data can be recovered on task failures, but is lost if the process crashes. To persist data at higher levels, users have to manually save the dataset by using other services (e.g., HDFS), which is inconvenient and time consuming.

In UlTraMan, data persisted at level `ON_KV` is saved transparently through reliable services at runtime. Moreover, this persistence does not sacrifice the performance expected of in-memory data access. To realize 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 survives task failures, and can be accessed at in-memory speed. However, for big datasets, the amount of shared memory may be insufficient. To tackle this problem, UlTraMan estimates the total data size and pre-allocates memory space before the creation of any Chronicle Map instance. If the remaining memory space is insufficient, a file on *disk* is used as the underlying storage. Since data in Chronicle Map is serialized internally, the corresponding performance on disk is better than that on the original Spark.

Furthermore, to serve as a reliable distributed storage, two techniques are applied to recover data from failures. First, UlTraMan 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. After that, the lineage of the `ON_KV` cached RDD is changed, and the parent operator (i.e., a special loader) is able to load the persisted files directly to Chronicle Map. As a result, missing data can be reloaded automatically under Spark's recomputation mechanism.

5. THE TRAJDATASET

Here, we present the abstraction of TrajDataset that aims to provide flexible operational interfaces based on distributed random data access. TrajDataset explicitly manages data by partitions, so that distributed computing can be scheduled and optimized at both local and global levels. Locally, data partitions are self-managed with flexible local indexes. Globally, the partitioning strategy is controlled by the user, and two scales of data generalization are enabled, namely the *global index* and the *meta table*.

5.1 Data Partition

A data partition is the basic operational unit. Operations on each data partition can be seen as running in a non-distributed environment, and thus a multitude of optimization techniques are available. To take full advantage of these, in addition to the original operations provided by Spark, UlTraMan supports index-based random data access on `RandomAccessRDDs` through two additional operations, `buildLocalIndex` and `query`.

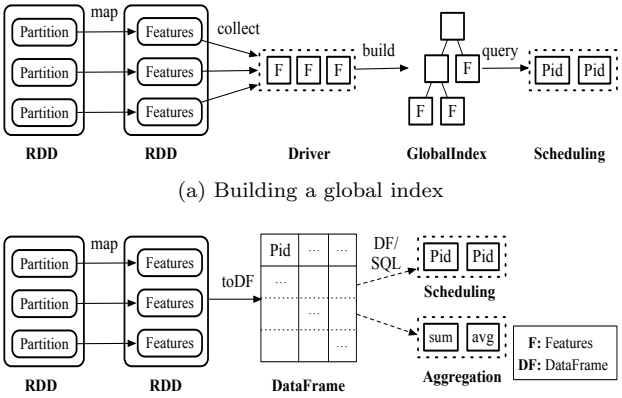


Figure 5: Data generalizations in UlTraMan.

A customizable *index constructor* is provided to support the process of building an index. The builder reads the partition data and outputs an index structure with an index name. As described in Section 4, a constructed index is consistent with the data on the persistence method. After the construction, the index manager in UlTraMan maintains the index along with its name.

In the query process, a customizable *querier* is adopted to provide a query algorithm and the names of its necessary indexes. At the beginning of the process, UlTraMan fetches the necessary indexes through the index manager. If all indexes are available and prepared, the query is conducted using the indexes and the algorithm. If indexes are missing, the query performs brute force scans.

5.2 Partitioning

Data partitioning is critical for efficient global scheduling. For instance, if trajectories are partitioned according to their spatial information, a spatial range query would be accelerated, since the partitions to search can be largely pruned with the range. In contrast, if trajectories are partitioned according to their time information, a global schedule may be useless for a range query because most partitions are search candidates. TrajDataset provides a `repartition` operation to support disparate partitioning strategies, and several partitioners are implemented in UlTraMan, including the STRPartitioner [37].

5.3 Global Index

To conduct global operations on the unified engine, a generalization of the dataset is often helpful. Building a global index is the most straightforward method of data generalization and has been adopted in existing systems [17, 37, 38]. Those systems usually apply a specific index structure (such as the R-tree) and a particular data partitioning strategy. In contrast, UlTraMan enables flexible partitioning and provides a `buildGlobalIndex` operation in TrajDataset that enables user-defined global indexes.

As depicted in Fig. 5(a), the global index construction process consists of three steps. First, a query is conducted on every data partition to create features. The features could be spatial bounding boxes, time spans, ID ranges, or something else that users have defined. Then, all the features are collected at the UlTraMan driver, and a global index is

built with the features as keys and the corresponding partition IDs as values. Finally, the built global index is stored in the TrajDataset for global scheduling. When a query is conducted on the global index, an array of *candidate partition IDs* is returned, and other partitions are pruned using an operation `globalFilter`.

5.4 Meta Table

Although the global index is effective for global scheduling, it is only applicable when the generalized features are of constant size. Otherwise, the collected features may become too large as the data volume increases, resulting in limited system scalability. For example, in the kNN query to be described in Section 6.3, collecting all trajectory IDs to the global index endangers system scalability.

To handle this problem, the *meta table* construction is adopted. Instead of collecting features at the UlTraMan driver, a meta table manages the features in *distributed* fashion. As an important component in UlTraMan's architecture, the meta table construction is designed and integrated for the following purposes:

- The meta table construction supports global random access with reasonable system scalability, making meta tables a natural replacement for global indexes in certain circumstances for scheduling and optimization.
- Meta table store pre-computed accounting information for each element/partition, such as *min*, *max*, and *average* values, in order to improve the efficiency of trajectory queries and mining tasks.
- In real applications, most analytical tasks are simple statistical queries and are usually expressed in SQL. In such cases, queries on a meta table are more efficient and more convenient to optimize than are queries on raw RDDs.

The meta table construction is realized with Spark SQL APIs [8], to take advantage of Spark's off-heap structured data encoding and built-in query optimizations. Fig. 5(b) illustrates how to build a meta table. TrajDataset provides an operation `extractMT` that allows users to extract features from data partitions and to transform the features to a DataFrame. Extracted DataFrames are maintained in a TrajDataset in the same way as the global indexes.

6. ANALYTICS CASE STUDIES

We demonstrate the system's flexibility through several trajectory data analytics case studies.

6.1 ID Query

The ID query is a simple trajectory data query. Three types of identities exist in a trajectory dataset: the element ID (e.g., a point ID), the trajectory ID (also called a trip ID), and the moving object ID. Since a moving object may produce multiple trajectories and a trajectory may contain multiple elements, the queries on different ID types are usually different. Here, we focus on the *trajectory ID query*, due to its utility in applications.

In Spark, the ID query can only be conducted with a brute force filtering, i.e., each element is scanned to check its trajectory ID. In UlTraMan, the local filtering on each data partition can be accelerated easily by introducing a hash map. Specifically, a hash map with trajectory IDs as keys and arrays of elements corresponding to the IDs as values can be built as a local index. Since Chronicle Map itself is a hash map, the index can be realized easily at level `ON_KV`. As a result, the brute force scans of a ID query can be improved to direct accesses with $O(1)$ complexity.

6.2 Range Query

The range query finds trajectory data with respect to a spatial or spatio-temporal range. UlTraMan maximizes the benefits of multi-dimensional indexes (e.g., the R-tree) through its support for partitioning and global-local indexing.

To achieve optimal partitioning, UlTraMan implements an STRPartitioner [37]. It packs data to partitions in the same way as building R-tree leaf nodes. By applying this partitioning technique, a global R-tree can be constructed that enables effective global filtering. In addition, local R-trees can be constructed within each partition, so that local range queries in the partitions resulting from the global filtering are accelerated.

6.3 kNN Query

A k nearest neighbor (kNN) query on trajectory data finds k nearest *trajectories* for a query spatial location. Here, the distance between a trajectory and a spatial location is computed as the distance from the location to the nearest *trajectory point* [43], although other distance functions can also be implemented in UlTraMan.

Existing algorithms [37] that find k nearest *elements* cannot be used to compute the kNN query, as fewer than k trajectories could be returned because some nearest elements may belong to the same trajectory. Although the kNN trajectory query is useful in real-life applications, it has not been supported fully in a distributed environment. Traditional centralized kNN trajectory algorithms [20] simply extend queries on elements with a buffer to record selected trajectories. This extension is not useful in a distributed algorithms [17, 37] because it is expensive to count trajectories across multiple, distributed partitions.

In UlTraMan, we propose and implement an *R-tree variant* that enables efficient kNN queries. In this R-tree, each tree node maintains a count of distinct trajectories in its covering partitions. As a result, a kNN trajectory query in UlTraMan is processed as follows.

Partitioning (optional). A global partitioning according to the spatial distribution (e.g., STRPartitioner) can improve the global pruning ability and the query efficiency.

Indexing and Extraction. First, local R-trees are built on the data partitions. Then, we extract trajectory IDs and partition IDs *(tid, pid)* to construct a meta table. Next, the bounding boxes and the partition IDs are extracted as features to construct a global R-tree. For each tree node of the global R-tree, the partitions covered are found, and the trajectory count is calculated using the following operation on the meta table.

```
metaTable.filter("pid in <covering partitions>")
         .agg(countDistinct("tid"))
```

1st Global Filtering. First, we find the nearest partition P with respect to the query location in the global R-tree. If P contains k or more trajectories, it forms the candidate partition set C_1. Otherwise, we find the leaf node that contains P. If the leaf node contains k or more trajectories, the candidate partitions in this leaf node are assigned

to C_1; if the leaf node contains fewer than k trajectories, we check its parent node, and this process is applied recursively until a node that contains k or more trajectories is found.

2nd Global Filtering. First, we conduct local kNN trajectory queries on the C_1 partitions to get k nearest trajectories. The k-th distance is a upper bound of the final result trajectories, based on which a range query is conducted on the global R-tree to find qualified partition candidates C_2 that contain the final result trajectories.

Local kNN. Local kNN queries are conducted on the C_2 partitions. The results are sorted globally by their distances, and the top-k trajectories are returned.

The support for the kNN query showcases the ability of UlTraMan to support existing as well as emerging indexes and algorithms for trajectory data analytics.

6.4 Aggregation Analysis

UlTraMan supports efficient aggregation analyses (e.g., `max`, `min`, `count`, and `avg`) through the meta table construct. As an example, *to get the average trajectory length of a dataset*, we first extract the trajectory ID and the length of each element (denoted as `tid` and `length`) to the meta table and then submit the following query:

```
metaTable.groupBy("tid")
         .agg(sum("length") as "tlength")
         .agg(avg("tlength"))
```

With the optimizations provided by Spark SQL, aggregation analysis on a meta table is more efficient than that on the original dataset. Further, with the flexibility in the extraction process, a meta table itself can be used as an optimization technique. For instance, in the above example, we can merge the (*tid, length*) tuples in each partition during extraction, so that the efficiency of the analysis over the whole dataset is further improved.

6.5 Co-Movement Pattern Mining

Co-movement pattern mining constitutes advanced mining functionalities for trajectory data and has been explored in several studies [21, 24, 27, 26]. Recently, Fan et al. [19] has proposed a distributed framework to mine general co-movement patterns on massive trajectories. This framework can also be easily realized in UlTraMan. Moreover, necessary preprocessing tasks that are not covered by their framework can be supported efficiently in UlTraMan, hence avoiding unnecessary data transfer. We detail the process of co-movement pattern mining in UlTraMan as follows.

Preprocessing: format transformation. The trajectory data should be transformed to a required format, such as meter-measured spatial coordinates (instead of latitude and longitude).

Preprocessing: synchronization. After format transformation, we synchronize the trajectories by a global timestamp sequence. During the process, the meta table is used to obtain the overall time period, and a specific partitioner is used to repartition the dataset via timestamp ranges.

Analysis: clustering. To find co-movements, we need to cluster the data at every timestamp. In UlTraMan, the clustering algorithm (e.g., DBSCAN [18]) can be accelerated by an R-tree built in advance.

Mining: co-movement pattern. The existing distributed mining algorithm [19], including star partitioning and apriori enumeration, can be implemented in UlTraMan with the APIs provided by TrajDataset and RDD.

Table 1: Statistics of the dataset used.

Attributes	Taxi	Shopping	Brinkhoff
# points	276,753,114	607,086,634	3,508,915,737
# trajectories	15,789	137,502	4,016,000
raw size	27.5GB	37.5GB	250.2GB
# snapshots	1,996	3,593	100,000
ϵ of DBSCAN	16	5,000	3.0

The implementation of co-movement pattern mining again showcases the advantage of UlTraMan's unified, efficient, and reliable data processing and analysis.

7. EXPERIMENTAL EVALUATION

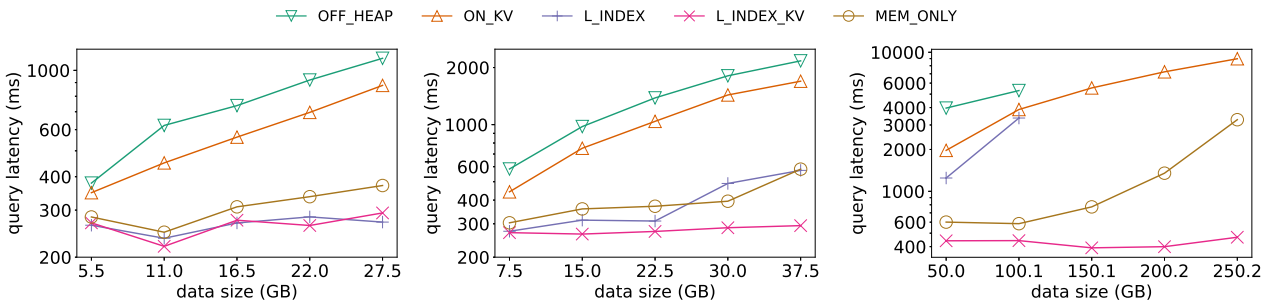
In this section, we experimentally evaluate the performance and scalability of UlTraMan. We conduct experiments on data processing, data retrieval, and trajectory clustering. The results, when compared to results obtained for baselines, generally demonstrate the effects of optimizations enabled by UlTraMan. We omit the results on the aggregation analysis and the pattern mining because their performance is determined by the underlying Spark and Spark SQL in the UlTraMan engine.

Experimental Setup: All experiments were conducted on a cluster consisting of 12 nodes. Each node is equipped with two 12-core processors (Intel Xeon E5-2620 v3 2.4GHz), 64GB RAM, and the Gigabit Ethernet. Each cluster node is equipped with a Ubuntu 14.04.3 LTS system with Hadoop 2.7.1, Spark 2.1.1, and Chronicle Map 3.14.0. In the cluster, one node is chosen as the master node and the driver node, and the remaining nodes are slave nodes. In each slave node, we allocate 40GB main memory for UlTraMan, of which 20GB is preserved for data storage and the other 20GB is used for temporary objects. All system modules and algorithms were implemented in Scala.

Datasets: We employ real and synthetic trajectory data sets to study the performance of UlTraMan. The two real data sets used fit in memory, which is the common use case targeted by UlTraMan. To evaluate the performance in extreme cases, a synthetic data set is generated that large enough that the allocated memory is insufficient to cache the whole data set. Statistics of the datasets listed next are stated in Table 1:

- `Taxi`[2]: This is a real trajectory dataset generated by taxis in Hangzhou, China. Trajectories are not separated by trips, and thus, each trajectory represents the trace of a taxi during a whole month.
- `Shopping`[3]: This dataset contains real trajectories of visitors in the ATC shopping center in Osaka. This is a dataset of free-space trajectories, in which the visitor locations are sampled every half second.
- `Brinkhoff`[4]: This dataset is generated on the real road network of Las Vegas via the Brinkhoff generator [11]. The moving objects in this dataset are generated step by step, i.e., in each step, an object moves along a road with random but reasonable direction and speed. Therefore, this dataset is naturally synchronized.

[2]This is a proprietary dataset.
[3]http://www.irc.atr.jp/crest2010_HRI/ATC_dataset/
[4]https://iapg.jade-hs.de/personen/brinkhoff/generator/



(a) Taxi (b) Shopping (c) Brinkhoff

Figure 6: The latency of ID queries on different data sizes.

Table 2: Preprocessing times(s).

Preprocessing	Taxi	Shopping	Brinkhoff
Hash Partitioning	11.204	20.648	61.578
STR Partitioning	19.529	42.892	115.725
On_KV Persistence	3.337	6.877	19.596
Meta Table Extract	1.474	3.097	8.533
Hash-map Loc-Index	3.068	7.619	24.219
R-tree Local Index	5.818	12.296	28.510
R-tree Global Index	0.388	0.398	0.416
R-tree (kNN) Glob.	16.622	20.988	50.539

When evaluating the scalability of UlTraMan, we partition the above datasets into smaller ones with different sizes, namely, 20%, 40%, 60%, 80%, and 100% of the dataset. By default, all of `Taxi` and `Shopping` and 40% of `Brinkhoff` are used to evaluate the performance when the data fits in memory. The data in `Taxi` and `Shopping` is partitioned according to time spans, and the `Brinkhoff` dataset is partitioned according to moving objects.

Parameters: Every measurement we report is an average of 100 queries, where 10 query cases are randomly generated according to the data distribution and each case is conducted 10 times. The default capacity of an R-tree node is set to 64. For DBSCAN clustering, the number of snapshots after synchronization and the applied ϵ for each dataset are shown in Table 1, and the minimum number of 15 moving objects is used when generating clusters. All figures with experimental results are shown with a logarithmic scale on the y-axis.

7.1 Preprocessing

We first evaluate the running times of different preprocessing techniques in UlTraMan. The results are presented in Table 2.

The first observation is that the running time for repartition is proportional to the dataset size. In particular, STR-partitioning is slower than hash-based partitioning, because an additional sampling procedure is needed to compute partition boundaries. Further, data repartition is one of the most time-consuming preprocessing tasks. This is because the data shuffling process in Spark stores data on disk internally before real network transfer, thus incurring extensive serialization, deserialization, and disk and network I/O costs. Hence, co-located computing and data persistence is important. The unified engine of UlTraMan is designed to reduce the need of frequent data shuffling.

Second, the times for `ON_KV` persistence and meta table extraction are small and almost equal to the time for scanning and copying the dataset in main memory. `ON_KV` persistence is about two times slower than the meta table extraction since additional time is needed to serialize the data to be stored in off-heap memory.

Third, the time for local index construction depends on the specific index structure. In this set of experiments, we construct local indexes on the `ON_KV` persisted datasets. The construction of a hash map is similar to the process of `ON_KV` persistence; thus, the time consumed is also similar to that consumed by `ON_KV` persistence. R-tree construction needs more time to pack data according to the spatial information. However, it is still fast due to in-memory data access, considering the large sizes of the datasets.

Finally, the construction of a global R-tree is very fast because the number of partitions is small compared to the dataset volume and because the spatial information of each partition has been computed during local index construction. Compared with the normal R-tree, the modified R-tree for the kNN query has a much longer construction time. The reason is that we have to conduct a query on the meta table for each tree node to get the count of trajectories in the covered partitions. Nonetheless, this time is acceptable, as the index saves substantial time during kNN querying, as to be shown in Section 7.4. We can conclude that a well-designed global index can help achieve better query performance with low construction cost.

7.2 ID Query

Next, we conduct ID queries on the datasets, measuring the query latencies. The results are shown in Fig. 6.

Fig. 6(a) shows the ID query performance on the `Taxi` dataset, where the lines labeled `MEM_ONLY` and `OFF_HEAP` show results for baselines that conduct a brute force filtering on the on-heap and off-heap persisted RDDs, respectively.

The first observation is that the filtering on the persisted RDDs scales linearly with respect to the data size. This is because the computation cost is linear to data size. However, the latency of queries on the off-heap persisted RDDs increases faster than those on the on-heap RDDs. The reason is that for a simple filtering process as in an ID query, the deserialization cost of off-heap data exceeds the computation cost.

The second observation is that the data persisted on `ON_KV`, which is provided by UlTraMan, can be queried faster than that persisted on `OFF_HEAP`, due to the optimized serializa-

tion mechanism based on Chronicle Map and trajectory data (as described in Section 4.2).

Finally, the performance of the ID queries using local indexes (`L_INDEX` and `L_INDEX_KV`) is better than that of the brute force methods. Also their latency increases only slightly as the data size increases. The reason is that a query on the hash map has an amortized $O(1)$ computation cost, which is much less than the $O(N)$ complexity of the baseline methods. It is reasonable to believe that the performance improvement brought by the indexes would be more significant for larger data volumes. The results confirm the importance of being able to support index structures in UlTraMan. Further, the results show that in spite of the additional cost of deserialization, the performance of the KV-persisted indexes is comparable to that of the on-heap indexes.

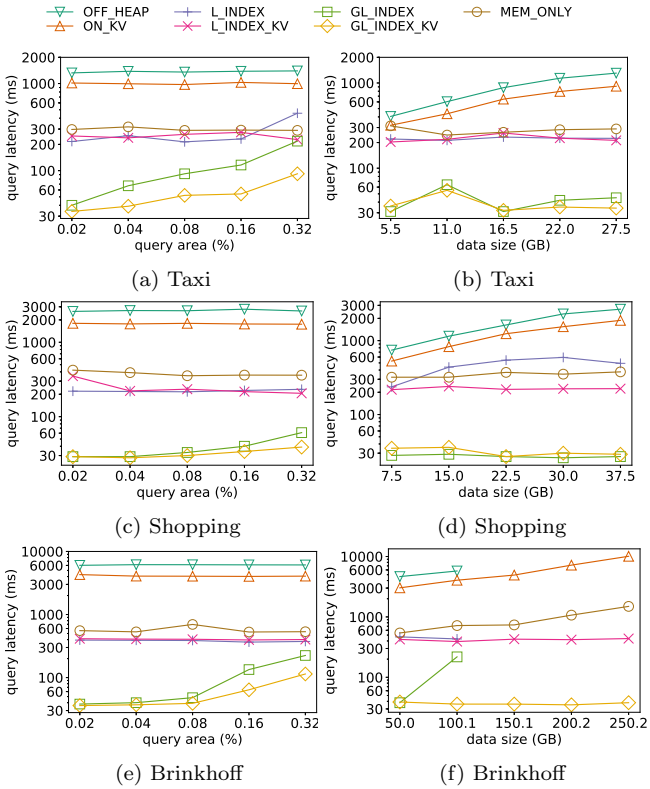
Fig. 6(b) shows the ID query performance on the `Shopping` datasets. The results are similar as those on `Taxi`, the reason being that the ID query is a light-weight data retrieval procedure that is affected little by the differences among datasets. However, the performance when using on-heap indexes deteriorates as the data volume grows. The cause of the large latency could be unexpected GC overheads in some queries. For the same query input, the best query case on `L_INDEX` returns results as fast as those on `L_INDEX_KV`, but other cases may be much slower due to GC processes, thus incurring a larger average latency. This is unavoidable when the data volume becomes large and the main memory is insufficient. Compared with `L_INDEX`, the KV-based indexes in UlTraMan have smaller in-memory footprints and generate little (or even no) garbage during ID queries, resulting in a stable and reliable performance.

Fig. 6(c) plots the ID query performance on the `Brinkhoff` datasets. Since the datasets become too large to fit in memory, the persistence tasks of all-in-memory levels, namely, `OFF_HEAP` and `L_INDEX`, fail and produce no results. Next, `L_INDEX` has poor performance in extreme cases, due to increasingly frequent GC procedures. Due to the fault tolerance techniques, data in the `MEM_ONLY` and `ON_KV` levels can work under memory overflow. However, the query performance deteriorates dramatically because some data is persisted on disk. As observed, the query latency of `ON_KV` increases slower than that of `MEM_ONLY`. This is because the KV store has better data retrieval and deserialization performance, due to its usage of the optimizations applied in UlTraMan. Finally, `L_INDEX_KV` provides very stable performance even when data does not fit in memory because the $O(1)$ algorithm saves I/O cost as well. As a conclusion, UlTraMan excels for large datasets.

7.3 Range Query

Next, we investigate the performance of range queries. Fig. 7 depicts two sets of range query experiments on each dataset, in order to study the impact of the query area and the data size on the performance. For range queries with respect to different data sizes, a 0.02% query area is used by default.

The first observation is that the latency of brute force methods (`MEM_ONLY`, `OFF_HEAP`, and `ON_KV`) stays stable when the query area is varied. These methods scan the whole dataset, so the cost remains the same. In contrast, the index-based methods gain speedups of up to a factor of 100 when the query area is small, because I/O and computing



(a) Taxi (b) Taxi

(c) Shopping (d) Shopping

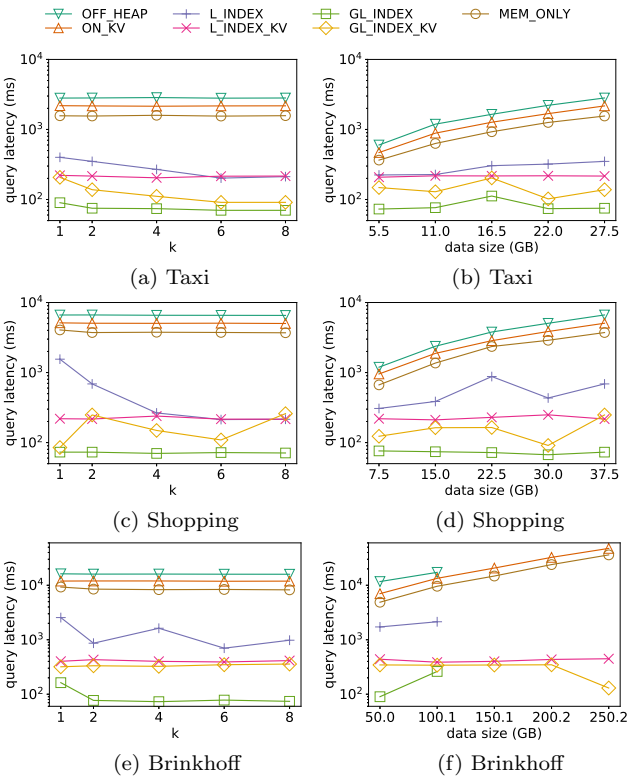
(e) Brinkhoff (f) Brinkhoff

Figure 7: Performance of range queries.

costs are effectively saved. The query time of index-based methods increases when the query area is large. This is because the main cost of a range query with a large query area is the cost of I/O and GC for the output, in which the improvement caused by indexing is limited. In real-life scenarios, a range query usually has a small area with respect to the whole data space. Hence, the index-based methods can achieve high query efficiency in real life applications.

The `ON_KV` persisted method performs better than Spark's built-in `OFF_HEAP` RDDs, which is because the deserialization cost is optimized. `L_INDEX` shows better performance in most cases (except for too large query areas) than brute force methods, and `L_INDEX_KV` performs better and is more stable than `L_INDEX` because it avoids garbage collection. In particular, `GL_INDEX`, which applies STR-partitioning and a global R-tree, shows a small speedup comparing with `L_INDEX` on all three datasets. This speedup is the result of global filtering using the global indexes and the partitioning. However, the partitioning also distributes the data to natural clusters in partitions, and thus, the local query time in the remaining partitions increases. We can conclude that `GL_INDEX_KV` performs the best for range queries. It achieves stable performance speedup by using the off-heap local indexes and global pruning.

As shown in Fig. 7(f), the scalability of each of `OFF_HEAP`, `L_INDEX`, and `GL_INDEX` is limited by the memory capacity of the cluster. In contrast, `MEM_ONLY`, `ON_KV`, `L_INDEX_KV`, and `GL_INDEX_KV` are capable of handling larger datasets by using disk space. Because of an increased I/O cost on disk-stored data, the query times of all these methods grows accordingly. However, the techniques of optimized partitioning, indexing,



(a) Taxi (b) Taxi

(c) Shopping (d) Shopping

(e) Brinkhoff (f) Brinkhoff

Figure 8: Performance of kNN queries.

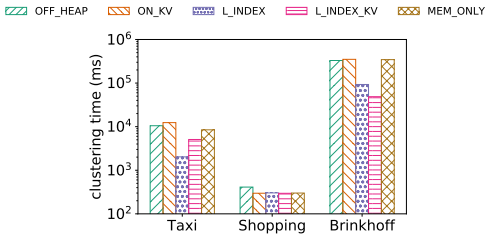


Figure 9: Performance of DBSCAN.

and serialization remain effective, which makes UlTraMan more powerful when handling very large datasets.

7.4 KNN Query

As shown in Fig. 8, we conduct two sets of kNN query experiments on each dataset in order to study the impact of the number k and the data size on the latency. For experiments with varying data sizes, k is set to 2 by default.

Unlike the two previous query types, the kNN query is both computation intensive and I/O intensive. The brute force method of computing a kNN query takes advantage of the RDD's `takeOrdered` API, which involves expensive distributed sorting. Hence, the queries without using indexes are slow. Relatively, `OFF_HEAP` is the worst due to expensive general deserialization. Next, `ON_KV` saves some deserialization time, and `MEM_ONLY` is the best among the three brute force approaches. The performance of the brute force methods decrease as the data size grows, due to the larger search space.

The use of local indexes yields better performance than the baseline methods by an order of magnitude, which shows the efficiency of indexes for complex data retrieval and analysis. However, the performance of `L_INDEX` is not stable, especially for the `Shopping` dataset. This is because the average trajectory length is shorter in the `Shopping` dataset. Hence, more temporary objects are produced during kNN queries, and the performance is affected by occasional process of garbage collection. In contrast, `L_INDEX_KV` shows stable performance across different k values and data sizes, which illustrates an important benefit of UlTraMan.

The global-local index based methods (i.e., `GL_INDEX` and `GL_INDEX_KV`) adopt specialized data partitioning and global indexes. This way, they are able to achieve lower latency than the methods that only use local indexes, and they are able to achieve a latency below $100ms$, which is very fast considering the scheduling and setup costs for distributed tasks. More specifically, there are two reasons behind this speedup: (i) most data partitions and distributed tasks are pruned by our kNN query algorithm; and (ii), more importantly, from a system perspective, the support for random access on both data partitions and local indexes enables algorithmic optimizations. For the two methods using global-local indexes, `GL_INDEX` performs slightly better. This is because the rate of garbage collection is very small for only a few local kNN queries due to the strong global pruning ability, while `GL_INDEX_KV` has additional deserialization costs in local queries.

Fig. 8(f) shows the query time for each method when data size exceeds the memory capacity. When the dataset becomes too large, `OFF_HEAP`, `L_INDEX`, and `GL_INDEX` fail to provide results, while `MEM_ONLY` and `OFF_HEAP` remain functional, but with an increased query time. In comparison, `L_INDEX_KV` and `GL_INDEX_KV` reduce the computation and I/O cost effectively even for the large datasets, thus providing relatively stable performance. The results again demonstrate the benefits of UlTraMan for big trajectory data analytics.

7.5 Clustering: DBSCAN

Finally, the efficiency of clustering in UlTraMan is reported in Fig. 9. Since the trajectory data has been synchronized to snapshots and the clustering is conducted within each snapshot (cf. Section 6.5), global indexes have no effect in this process and hence are not used in the experiments.

For the `Shopping` dataset, the number of customers in a snapshot is small, and clustering in snapshots is very fast: DBSCAN can finish in less than $400ms$ with or without the use of indexes. In particular, `OFF_HEAP` is slower than other methods, since the deserialization cost is substantial for the intensive data access made by DBSCAN. Nonetheless, `ON_KV` shows to be as efficient as the other methods, which is attributed to its optimized serialization.

For the `Taxi` dataset, a snapshot size is usually large (near the number of taxis) because most taxis are traveling all the time. This is also true for the `Brinkhoff` dataset, because locations of the moving objects are generated snapshot by snapshot. As a result, the clustering time is large on `Taxi` and `Brinkhoff`. All three brute force methods have similar performance, and they consume tens of seconds to answer the query. With the help of pre-constructed local indexes, the processing time can be reduced by up to 4/5.

It is interesting that `L_INDEX_KV` is slower than `L_INDEX` on `Taxi` but faster than `L_INDEX` on `Brinkhoff`. On `Taxi`,

L_INDEX_KV is slower than L_INDEX because (i) the data size and the number of snapshots is smaller in Taxi, resulting in less pressure of garbage collection on L_INDEX, while (ii) the cost of deserialization on L_INDEX_KV is substantial. In contrast, the pressure on main memory space and garbage collection are large for Brinkhoff, which hurts the performance of L_INDEX.

8. RELATED WORK

We proceed to survey the related system architectures and clarify the difference between them and UlTraMan.

Several systems for trajectory data management and analysis exist. PIST [10], BerlinMOD [16] and TrajStore [13] propose specific storage structure and indexes targeting traditional database engines. SharkDB [35] adopts a columnar data schema to provide better query and analysis performance. However, these systems are designed for centralized architectures, and thus they are inefficient for, or incapable of, handling massive volumes of trajectory data. In addition, as the storage structures are constrained by the underlying database engine, the system flexibility is limited.

SpatialHadoop [17] and Simba [37] enable distributed spatial analytics based on the MapReduce paradigm. Nonetheless, they are unable to exploit the characterizes of trajectory data for efficient data management and analytics. CloST [34], PARADASE [29], Elite [38], and a cloud-based system [9] provide distributed solutions for big trajectory data. They utilize specific partitioning strategies in distributed environments to support data retrieval. In contrast, UlTraMan adopts a flexible framework that supports customizable data formats, partitioning strategies, index structures, processing methods, and analysis techniques, which offers better support to realize optimizations and complex analytics.

Other systems that offer distributed storage and computing also exist. SnappyData [30] integrates Apache Spark and Apache GemFire [3] to support efficient streaming, transactions, and interactive analytics. Apache Ignite [4] also integrates Apache Spark with its key-value store to enable data sharing across RDDs. The IndexedRDD project [5] maintains an index within each partition. Although these systems provide solutions that enhance Spark and eliminate inefficiencies of heterogenous systems, they do not provide flexible operations and optimizations for trajectory data analytics. In contrast, UlTraMan adopts a unified engine accompanied with a TrajDataset abstraction that support efficient trajectory data management and analytics.

9. CONCLUSIONS

In this paper, we present UlTraMan, a flexible and scalable distributed platform for trajectory data management and analytics. We demonstrate that UlTraMan is able to outperform state-of-the-art systems, which is achieved by (i) providing a unified engine for both efficient data management and distributed computing, and by (ii) offering an enhanced computing paradigm in a highly modular architecture to enable both pipeline customization and module extension. Several case studies with efficient algorithms are implemented in UlTraMan. Experimental studies on large-scale real and synthetic data sets that include comparisons with the state-of-the-art methods offer insight into the efficiency and scalability of UlTraMan. In the future, one promising direction of improving UlTraMan is to integrate the meta table construction into global indexes in order to provide more efficient and convenient global scheduling. It is also of interest to design and integrate additional indexes and analytical algorithms into UlTraMan, in order to further support trajectory analysis use cases from real-world applications.

10. ACKNOWLEDGMENTS

Yunjun Gao is the corresponding author of this work. The work was supported in part by the 973 Program Grants No. 2015CB352502 and 2015CB352503, NSFC Grant No. 61522208, NSFC-Zhejiang Joint Fund No. U1609217, the DiCyPS project, and a grant from the Obel Family Foundation. Also, the authors of the work would like to thank Rui Chen, Guanhua Mai, Yi Ren, Renfei Huang, and Zhengyi Yang for their help on the system implementation and evaluation.

11. REFERENCES

[1] H2 database engine. http://www.h2database.com, 2006.
[2] Redis. https://redis.io, 2010.
[3] Apache geode. https://geode.apache.org, 2015.
[4] Apache ignite. https://ignite.apache.org, 2015.
[5] Indexedrdd. https://github.com/amplab/spark-indexedrdd, 2015.
[6] The DiDi Research. http://research.xiaojukeji.com/index.html, 2016.
[7] Chronicle map. https://github.com/OpenHFT/Chronicle-Map, 2017.
[8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.
[9] J. Bao, R. Li, X. Yi, and Y. Zheng. Managing massive trajectories on the cloud. In *SIGSPATIAL*, pages 41:1–41:10, 2016.
[10] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. PIST: an efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.
[11] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
[12] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266, 2010.
[13] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.
[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[15] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica*, 10(2):112–122, 1973.

[16] C. Düntgen, T. Behr, and R. H. Güting. Berlinmod: a benchmark for moving object databases. *VLDB J.*, 18(6):1335–1368, 2009.

[17] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[18] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[19] Q. Fan, D. Zhang, H. Wu, and K. Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *PVLDB*, 10(4):313–324, 2016.

[20] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *GeoInformatica*, 11(2):159–193, 2007.

[21] J. Gudmundsson and M. J. van Kreveld. Computing longest duration flocks in trajectory data. In *GIS*, pages 35–42, 2006.

[22] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[23] Y. Han, W. Sun, and B. Zheng. COMPRESS: A comprehensive framework of trajectory compression in road networks. *ACM Trans. Database Syst.*, 42(2):11:1–11:49, 2017.

[24] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.

[25] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[26] X. Li, V. Ceikute, C. S. Jensen, and K. Tan. Effective online group discovery in trajectory databases. *IEEE Trans. Knowl. Data Eng.*, 25(12):2752–2766, 2013.

[27] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1):723–734, 2010.

[28] Z. Li, J. Han, M. Ji, L. A. Tang, Y. Yu, B. Ding, J. Lee, and R. Kays. Movemine: Mining moving object data for discovery of animal movement patterns. *ACM TIST*, 2(4):37, 2011.

[29] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query processing of massive trajectory data based on mapreduce. In *CloudDB*, pages 9–16, 2009.

[30] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *CIDR*, 2017.

[31] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *VLDB*, pages 395–406, 2000.

[32] S. Ranu, D. P, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, pages 999–1010, 2015.

[33] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[34] H. Tan, W. Luo, and L. M. Ni. Clost: a Hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, pages 2139–2143, 2012.

[35] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. W. Sadiq. Sharkdb: An in-memory column-oriented trajectory storage. In *CIKM*, pages 1409–1418, 2014.

[36] Y. Wang, Y. Zheng, and Y. Xue. Travel time estimation of a path using sparse trajectories. In *KDD*, pages 25–34, 2014.

[37] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.

[38] X. Xie, B. Mei, J. Chen, X. Du, and C. S. Jensen. Elite: an elastic infrastructure for big spatiotemporal trajectories. *VLDB J.*, 25(4):473–493, 2016.

[39] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G. Sun. An interactive-voting based map matching algorithm. In *MDM*, pages 43–52, 2010.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[41] Y. Zheng. Trajectory data mining: An overview. *ACM TIST*, 6(3):29, 2015.

[42] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: Concepts, methodologies, and applications. *ACM TIST*, 5(3):38:1–38:55, 2014.

[43] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.