Journal of Geographical Systems
https://doi.org/10.1007/s10109-019-00292-4

ORIGINAL ARTICLE

HadoopTrajectory: a Hadoop spatiotemporal data processing extension

Mohamed Bakli[1] · **Mahmoud Sakr**[2,3] · **Taysir Hassan A. Soliman**[1]

Received: 23 April 2018 / Accepted: 9 January 2019

Abstract

The recent advances in location tracking technologies and the widespread use of location-aware applications have resulted in big datasets of moving object trajectories. While there exists a couple of research prototypes for moving object databases, there is a lack of systems that can process big spatiotemporal data. This work proposes HadoopTrajectory, a Hadoop extension for spatiotemporal data processing. The extension adds spatiotemporal types and operators to the Hadoop core. These types and operators can be directly used in MapReduce programs, which gives the Hadoop user the possibility to write spatiotemporal data analytics programs. The storage layer of Hadoop, the HDFS, is extended by types to represent trajectory data and their corresponding input and output functions. It is also extended by file splitters and record readers. This enables Hadoop to read big files of moving object trajectories such as vehicle GPS tracks and split them over worker nodes for distributed processing. The storage layer is also extended by spatiotemporal indexes that help filtering the data before splitting it over the worker nodes. Several data access functions are provided so that the MapReduce layer can deal with this data. The MapReduce layer is extended with trajectory processing operators, to compute for instance the length of a trajectory in meters. This paper describes the extension and evaluates it using a synthetic dataset and a real dataset. Comparisons with non-Hadoop systems and with standard Hadoop are given. The extension accounts for about 11,601 lines of Java code.

Keywords Spatiotemporal · Hadoop · 3DR-tree · Trajectory data management · Big data

JEL Classification C6 · C8 · R4 · R53 · L86 · O3

✉ Mohamed Bakli
mohamed_bakli@aun.edu.eg

Extended author information available on the last page of the article

Published online: 01 February 2019

1 Introduction

Nowadays, knowledge of the spatial and temporal data is very important for you in all fields. Understanding the processes at the basis of movement will help to discover useful information. It is essential to forecast the impact of human-caused environmental changes and outline conservation strategies. Spatiotemporal data come in huge masses from various sources such as web data, IoT data, scientific data, etc. This data deluge, more commonly known as big data, have introduced unprecedented performance and scalability challenges to data management and processing systems. The large volume of spatiotemporal data is beyond the capability of legacy systems to store, process and analyze. These data come in different forms such as vehicle tracking logs and user activities on mobile. Vehicle tracking logs represent each trajectory as multiple of records. In contrast, each user activity data is represented as one record.

In many cases, data are characterized by two very important dimensions: (geographical) space and time. In a two-dimensional space, spatial data types only provide simple object structures, such as single points, lines, and simple regions. In a three-dimensional space, spatiotemporal data types enable the user to describe the dynamic behavior of spatial objects over time. The dynamic behavior refers to the continuous change in the locations of the spatial objects over time. The number of collected raw data increases exponentially so that any analysis task becomes more complex. Hadoop (http://hadoop.apache.org/) makes it possible to run applications on thousands of commodity hardware nodes and handling thousands of terabytes of data. So, the biggest enterprises use Hadoop as a back-end of their tools, such as Environmental Systems Research Institute (ESRI) (https://www.esri.com/). ESRI developed a geometry Application Programming Interface (API) for Java. It allows users to build geometrical functions for Hadoop-related systems. But, it lacked the capability of indexing the objects. Hadoop is powerful but there are two problems when using it.

The first problem is that Hadoop is not aware of the nature of the spatiotemporal data. Some of these data are correlated with each other like moving object trajectories. Each moving object consists of multiple lines of file input. The HDFS deals only with heap files. Big data files are split into block files that are stored and replicated on the data nodes across the cluster. The data splitting process is done at the record level. It means that the data of one moving object might get partitioned over multiple block files in HDFS. Therefore, any operation takes more time to access these files in order to reconstruct the moving object. Clearly, this will negatively affect the efficiency of queries on moving objects. We address this problem by injecting the file syntax of moving objects into the HDFS architecture. So, it is able to understand their storage structure and to respect this structure when splitting files. Most of the previous systems were built on the top of Hadoop and use Hadoop as a black box, whereas Hadoop is not aware of the nature of the data. All the data types are defined in the main application, not in Hadoop. Hadoop extension addresses this in many applications such as ParallelSECONDO (Güting and Lu 2015), HadoopGIS (Aji et al. 2013), TRUSTER (Yang et al. 2009), PRADASE (Ma et al. 2009).

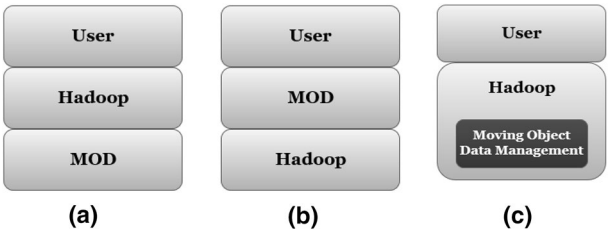

Fig. 1 The three integration alternatives

The second problem is that Hadoop does not provide index structures able to prune some data before performing an operation. Accordingly the whole dataset is partitioned and sent to the data nodes. In the first part of our contribution (Bakli et al. 2018), we extended Hadoop with a spatiotemporal Algebra. It consists of moving object types built in HDFS and operators built as MapReduce jobs. Operators can be chained and nested, both in sequence and in parallel to build complex analytics jobs. However, in the experiments, we observed that the dominant runtime cost was due to the HDFS overhead to copy data to all nodes. It also overtook the advantages gained from MapReduce. Therefore, we address this problem by introducing global spatiotemporal indexing and partitioning technique into the HDFS. So, it is able to filter the data before sending it to nodes.

The overall paper objective is to build a moving object data management system based on MapReduce. This shall be realized as a Hadoop extension. There are generally three approaches for such an integration:

1. Put Hadoop on top of a Moving Object Database (MOD). In such a way, the worker nodes have stand-alone MOD instances, and Hadoop orchestrates their work, as illustrated in Fig. 1a.
2. Put a MOD on top of Hadoop. In this setting, Hadoop would act as a file manager and a processing framework, while all user interaction is done via the MOD interface, by means of queries, as illustrated in Fig. 1b.
3. Extend Hadoop by spatiotemporal types and operators. This integrates the moving object data management into the core of Hadoop. So, MapReduce programs can define and process moving object types, as illustrated in Fig. 1c.

We applied the third approach of integration. It is the most challenging among these variants as it requires a non-naive change into the Hadoop system and a complete parallelism of MOD. Yet, it provides the maximum flexibility for writing scalable moving object data management programs. So, the contribution of this paper can be summarized as follows:

- Extending Hadoop with spatiotemporal types and operators.
- Building an index structure to optimize the data transfer between Hadoop nodes while processing queries.
- Introducing operators for index access.

- Adding index support to operators in the MapReduce where suitable.
- Showcase optimized queries based on the BerlinMOD benchmark.

The rest of this paper is organized as follows. Section 2 reviews the closely related work about this study. Section 3 explains the proposed HadoopTrajectory in details. Section 4 evaluates the performance of the proposed Algebra and compares it with the SECONDO system. Finally, Sect. 5 concludes the paper and mentions the future development in Sect. 6.

2 Related work

A moving object database system MOD is a database system that is able to store, query and manage moving object data, also called trajectories. There exist few prototypes for such a system: such as SECONDO (Gting et al. 2004), Hermes (Pelekis et al. 2006), and DEDALE (Grumbach et al. 1998). These three systems manage trajectories of moving objects. DEDALE (Grumbach et al. 1998) uses a constraint DB approach for managing moving objects. In such a representation, a spatial object is represented as a set of constraints that represent its area. Adding other dimensions to the data is by definition supported by introducing additional variables such as a third spatial dimension and the time and by expressing more constraints over these variables. Both SECONDO and HERMES use an abstract data type model for moving objects (Güting et al. 2000). In contrast DEDALE uses a constraint database model. The abstract data types for moving objects encapsulate the trajectory information in types that are supported by the database system. For instance, SECONDO provides the types: *mpoint* for a temporal spatial point, *mregion* for a temporal spatial region, *mreal* for a temporal numeric value, etc. Using these types, it is possible to store within a tuple a trajectory of a car, represented as an *mpoint* instance. Query operators can then be expressed on such a trajectory. For instance, an operator *speed*(*mpoint*) would yield the time-dependent speed of the car, represented as an *mreal*. HERMES implements the same model as SECONDO, yet on top of PostgreSQL and ORACLE. The ADT approach facilitates building indexes and query optimization methods. There are many spatiotemporal indexes for instance, 3DR-tree (Theodoridis et al. 1996), TB-tree (Pfoser et al. 2000), STR-tree (Pfoser et al. 2000), FNR-tree (Frentzos 2003) and MON-tree (De Almeida and Güting 2005). Despite the optimization that can be done in MOD systems to speed up the query processing, they remain non-scalable. Such systems are expected to be run on a single node server.

Scalable moving object databases Since these MOD prototypes are not scalable and hence cannot support big spatiotemporal data processing, it was natural to extend them in the direction of distributed databases. For example, ParallelSECONDO (Güting and Lu 2015) is a version of SECONDO that uses Hadoop (http://hadoop.apache.org/). Hadoop is used as a communication manager for scheduling and coordinating the tasks between worker nodes, each of which runs a regular SECONDO instance, and contains a complete copy of the data. Each node in the cluster contains Hadoop and mini SECONDO to run the job. The user query is converted by the master node into a set

of parallel query statements. These statements are scheduled to worker nodes. Their individual results are communicated back to the master, which in turn compiles them and generates the final query answer. Distributed SECONDO (Nidzwetzki and Güting 2015) is another SECONDO version that implements a distributed moving object database without using Hadoop. It uses CASANDRA as a storage layer, instead of BerkelyDB which is used in the original SECONDO. It thus inherits the high availability and the fast updates from CASANDRA. Both the management and query processing are performed by SECONDO nodes.

GeoMesa (Fox et al. 2013) is a spatiotemporal database used to store, query and transform the spatiotemporal data at a large scale. It is built on the top of Apache Accumulo, which is a key value store built on the top of Hadoop. GeoMesa organizes the data using geohashes and timestamps. The keys are generated as a combination the geohash and the temporal value.

Extending Hadoop with spatiotemporal functionality In contrast to MOD systems, this approach seeks to extend Hadoop (or other big data frameworks) with spatiotemporal support. Hadoop is used not only for task scheduling and monitoring, but rather extends to task execution. This is the approach that is used in this paper. One recent work also exist following this approach, called ST-Hadoop (Alarabi and Mokbel 2017) short for spatiotemporal Hadoop. It is a temporal extension of SpatialHadoop (Eldawy and Mokbel 2015). SpatialHadoop pushes the spatial types and operators inside the core of Hadoop to be as built-in. This includes the basic GIS types of point and region. It implements two levels of indexing, a global index functioning on the master node and a local index functioning on the data slice of every worker node. SpatialHadoop uses three types of indexes: grid files, R-tree, and R+-tree. It provides a set of spatial operators, with index integration wherever possible. Our work builds on some SpatialHadoop data types and operators. We also utilize the concept of building two levels of indexing, which was originally introduced in SpatialHadoop, in indexing the spatiotemporal data.

ST-Hadoop extends it with a temporal support. It implements a single spatiotemporal type called *STPoint*, which is a triple of (latitude, longitude, time). It can process big files of STPoints and evaluate selectivity and join predicates on them (e.g., overlap and within_distance). The global index of SpatialHadoop is extended by a temporal slicer that partitions the input files into slices according to a given temporal granularity: day, week, month. Within every slice, the SpatialHadoop kind of index is built. Accordingly the operators will first filter by time, fetch the corresponding slices, and process them using temporal extensions of SpatialHadoop operators. ST-Hadoop cannot express the notion of a trajectory and cannot accordingly express trajectory level operations such as *speed, intersects*. It is limited to processing sets of discrete temporal points. A detailed comparison between ST-Hadoop and our work will be discussed in Sect. 8.

3 HadoopTrajectory architecture

The goal of the proposed HadoopTrajectory is to develop a spatiotemporal data processing framework that is highly scalable and highly available. Therefore, we inject the spatiotemporal logic into the core of Hadoop and take advantage of

MapReduce running on commodity clusters. On top of this framework, spatiotemporal big data application can be written.

The proposed HadoopTrajectory includes the following components:

1. **Indexing** in the form of a spatiotemporal 3DR-tree structure, that is used as a global index on the whole dataset of moving objects and their trajectories.
2. **Partitioning** the big data files of moving objects into multiple chunks. Partitioning is done based on disk page size, preserving the semantic and the structure of moving objects (e.g., one moving object shall not be split over multiple partitions).
3. **Linking** each moving object with its metadata that can be in other files. It helps answer queries both at the trajectory level and at the moving object level. Note that one moving object may have multiple trajectories, e.g., a car performing multiple trips.
4. **Operators** to process trajectories, including the computation of movement attributes (e.g., speed, direction), predicates, and trajectory restriction to space and time. Many of these operators utilize the index.
5. **Jobs** in the form of the user MapReduce task that invokes the aforementioned components to perform big spatiotemporal data processing.

These components are grouped into the following architectural layers, as depicted in Fig. 2. The following enumerates the layers and their components. These will be explained in detail in the following sections.

- **The Storage Layer** We extend the HDFS so that it can manage spatiotemporal data. The extension has four parts: the spatiotemporal types, the global index, the moving object data partitioner and the linker. The global index is a spatiotemporal 3DR-tree used to index and filter the moving objects. A job can take the advantages of the index if some of its predicates are index supported, such as *passes* and *intersects*. Accordingly, an index filter can be invoked, so that only the result candidates are sent to worker nodes. The partitioner works in combination with the index to split the big data file into blocks and to link the moving objects inside the block files to the index. The linker is used to link the moving object trajectory information. We build some operators for creating and scanning the index: *IndexCreate*, *IndexScan* and *Multiple-IndexScan*. For partitioning the big input files, the *Partition* and the *IDScan* are implemented. Finally, the *Link* operator is implemented for linking the trajectories with other attributes that may exist in the input files (e.g., the car license number, owner, the trip purpose, etc).
- **The MapReduce Layer** We extend the Hadoop MapReduce with the support functions needed to interact with the storage layer components, e.g., index, partitioner, and linker. This includes the functions: *IndexFilter*, *ObjectFilter*, *ObjectFileSplitter*, and *ObjectRecordReader*. The IndexFilter scans the global index and retrieves the moving object identifiers that intersect a given spatiotemporal box. The ObjectFilter is used for queries which require a specific

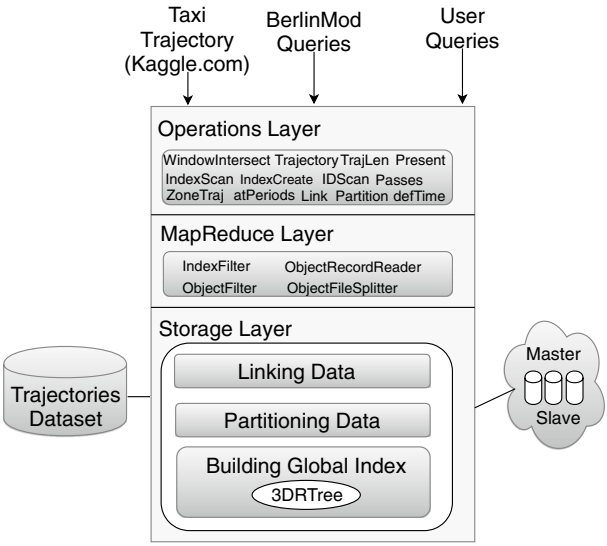

Fig. 2 The proposed architecture of the HadoopTrajectory

moving object by its ID. The ObjectFileSplitter extends the Hadoop *FileSplitter* function, so as to be able to split the input files. This function is the main component of the *Partition* operators of the storage layer. Finally, the ObjectRecordReader is an extension of the HDFS RecordReader. It allows Hadoop to understand the different formats/structures of moving object trajectories. For example, we used three datasets in the experiments that come in different file structures. Implementing multiple RecordReader classes in hence necessary to cope with this.

- **The Operation Layer** It groups the spatiotemporal operators that can deal with the extension types.

4 Storage layer

In this section, we describe our HDFS extension. First, the idea of the global index and its use is explained. We then illustrate the index implementation into HDFS. Finally, we explain the partitioner and the linker and their implementation in HDFS.

4.1 The global index

The global index is applied to the big moving object data files. As Hadoop is schemaless and does not have the notion of a relation, we had to implement couple of hooks to integrate the index in the HDFS environment. The ultimate goal is to filter out the data that has no chance to be among the query results, before dispatching the data and the jobs to the cluster nodes. The idea of the global index has been proposed in SpatialHadoop. Here, we reuse this idea in

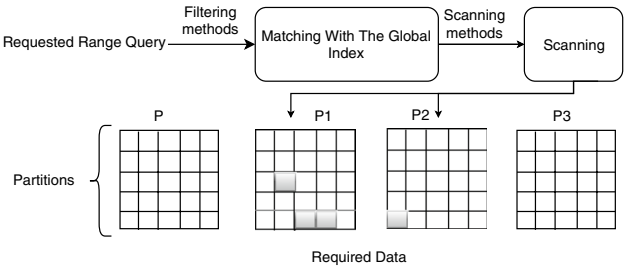

Fig. 3 Global index description

the context of spatiotemporal data. The global index directly links to the block files and understands their partitions and their metadata (i.e., all the information that describe the trajectory such as speed, type, etc). All the information about the global index is stored in the master file. It is used to organize the data in the other nodes for more efficient update and faster access. The data is divided into many partitions and stored in the worker nodes. The global index serves as a primary mean to uniquely identify partitions in the worker nodes.

We have implemented two index structures, from which the user can choose Grid and R-tree. The Grid is a space partitioning structure that splits the 3D extent of the input data into cubes, each of which is mapped to one or many files. This structure allows that one trajectory spans multiple grid cells. The 3DR-tree is a data partitioning structure that stores the 3D bounding boxes of the trajectories. Both indexes support queries based on spatiotemporal windows. All the information about the index is stored in the main memory of the master node. The index maps the bounding boxes directly to the partitioned files, so that the big input files are not anymore used. We allow for two granularities in creating an index: the trajectory and the moving object. In the trajectory granularity, one 3D box per trajectory is stored into the index, whereas in the moving object granularity, the 3D boxes of all the trajectories that belong to the same moving object are unioned into a single 3D box, and this box is stored into the index.

Consider a job of MapReduce that needs to perform a spatiotemporal filter on the data. As illustrated in Fig. 3, it will pass the requested spatiotemporal range of the query to the index. The index will apply filtering methods and search its internal structure for the overlapping trajectories and return their identifiers. The scanner (i.e., partitions scanning process) will then identify the files/partitions that contain the data of these trajectories, open each file only once, and retrieve all its associated trajectory data. The opening and reading of a file is implemented via a *TrajectoryInputSplit*, which is our extension to the Hadoop *InputSplit* object. To this end, the index access terminates. As will be explained in Sect. 6, the data of these trajectories will be again partitioned over multiple files and distributed over the worker nodes to further execute the task.

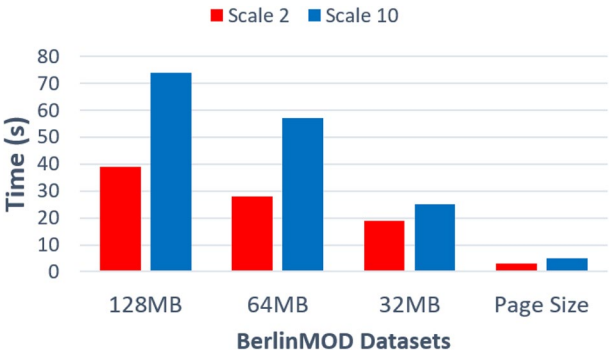

Fig. 4 The access times of selecting moving objects

4.2 The partitioner

The goal of the partitioning is to split the big input files into multiple smaller ones, so that they can be managed more efficiently. Additionally the partitioner will create a master file that maps trajectory identifiers to the partitions/files that contain their data (i.e., a hash map). The partitioning is done independently from the indexing. There are two major considerations for trajectories data partitioning. The first is to avoid the examination of many trajectories to obtain the desired ones (i.e., favoring small partitions). The second is to avoid the distribution of the trajectory segments over many files. That is, a single moving object trajectory consists of a list of segments, each of which describes a part of the movement. This requirement in contrast to the previous one might result in bigger partitions. Yet, it is important to avoid scanning many blocks to retrieve the information about a specific moving object.

We have performed an experiment with multiple partition sizes to assess a good size choice. Figure 4 shows the access time of selecting a random trajectory from the HDFS blocks. The data were generated using BerlinMOD benchmark (Düntgen et al. 2009) with scale factors of 2, 10. The *scale factor* is the parameter by which one can control the size of the generated data by BerlinMOD. For instance, when it is set to be 1, trajectories of 2000 vehicles running on the street network of Berlin for 28 days are simulated, requiring about 11 GB of disk space. Increasing this number will exponentially increase both the number of vehicles and the simulation duration and vice versa. BerlinMOD generates the data in the SECONDO format, which follows the sliced representation of moving objects in Forlizzi et al. (2000). In this representation, a trajectory is a sequence of so-called *units*, each of which is a tuple <start time, end time, start point, end point>.

With a block size of 128 MB, the access time to retrieve the units of the trajectories increases because it needs to iterate over many objects in one file. Note that the file is the data access unit. The default block size of Hadoop 1, 2 is 64 MB and 128 MB, respectively. In our case, a file of such size will contain many objects with many trajectories which can negatively affect the performance gain of the index. Therefore, we alter this setting and use smaller block sizes. We have experimented

Table 1 The cost of partitioning

Block size	Scale 2 cost (s)	Scale 10 cost (s)
128 MB	648	2516
64 MB	537	2164
32 MB	589	2278
Default page size	712	2449

Trajectories File		Meta data files		
Id	Trajectory	Id	Model	Plate
101	((2007-05-28-08:36,2007-05-28-08:37,4.11,1308,12793,1310),(2007-05-28-08:37,2007-05-28-08:38,12793,1310,12808,1314),...)	101	BMW	B-RL01
102	((2007-05-28-08:52:53,2007-05-28-08:52:55,9716,-55,9716.93,-46.7183),...)	102	Toyota	B-TX13
103	((2007-05-28-09:03:00,2007-05-28-09:03:02,7324,7804,7298,7832))	103	BMW	B-ZV11

Fig. 5 Stored trajectories information

with different block sizes and found that the time constantly decreases with the decrease in the block size, e.g., 64 MB, 32 MB, and up to the last value, which is called the default block size of the operating system. As it is not possible to go beyond the OS block size, we choose it. Using the default page size is also an advantage when the memory is small since it does not require using a large RAM space in the worker nodes.

Table 1 shows, on the other hand, the time cost of partitioning. The data is partitioned into chunks of different sizes: 128 MB, 64 MB, 32 MB, and the default page size of 4 MB. The smaller the block size, the more time the partitioning takes.

4.3 The linker

We allow that the input files include other attributes besides the trajectory attribute. These attributes are required to be given in separate files, using the same identifier as the one used in the trajectory files. For instance, an input can reflect the schema:

<Id, car model, plate number, trajectory>

To pass this input, it is required to have in one file the attributes: Id, car model, and license number. The trajectory file must then contain the attributes: Id, trajectory. A sample of the input files of such a schema is illustrated in Fig. 5. The job of the linker is similar to that of the partitioner, yet on the attribute files.

4.4 Spatiotemporal data types

The types that we have implemented have been inspired from the MOD model in Güting et al. (2000). Specifically, these types have been implemented: *instant*, *interval*, *periods*, *points*, *regions*, *trajSegment*, *trajectory*. The three time types *instant*, *interval*, and *periods* represent a single timestamp, a time interval, and a set of time

intervals, respectively. The two spatial types *points* and *regions* have been added to the spatial types in SpatialHadoop. The *trajSegment* type represents a segment of the trajectory (i.e., a linear interpolation between two consecutive location observations). The *trajectory* type is a list of *trajSegment*s, ordered by time. Most of these types of system has been proposed in a previous work (Güting et al. 2000).

These types have been carefully integrated into the core of Hadoop with corresponding HDFS input/output formatters and splitters. Since Hadoop accepts its input in the form of files, we have also defined certain file formats for each of these types. Conceptually, the data are organized in this hierarchy:

```
Moving Object: <Id, List<Trajectory> >
Trajectory:    <Id, List<Segment> >
Segment:   <Tstart, Tend, X1, Y1, X2, Y2>
TrajectoryData:      <Id, Car model, ...etc>
```

For more flexibility, multiple file formats have been defined for some types. Our extension will expect that the input files respect these formats. For the *trajectory* type, for instance, one file format would expect one trajectory per line, encoded in a nested way: one pair of brackets for the whole trajectory within which every x, y, t observation is enclosed in a pair of brackets. Another accepted format is to put each trajectory in multiple lines, where each line is a single segment: $Id, t_1, t_2, x_1, y_1, x_2, y_2$, where the *Id* identifies the trajectory and will be repeated for all its segments.

4.5 Index operators

The proposed HadoopTrajectory provides operators for building and scanning moving object trajectories. These operators are described in details in the following:

IndexCreate This operator creates for the input trajectory file, either a Grid or a R-Tree as described in Sect. 4.1. The index is stored in multiple files: a master file that contains the index structure and multiple data files. In the Grid, a data file will contain the identifiers of one cube cell. In the R-Tree, a data file will contain the identifiers in a leaf node in the tree. The operator can either be used to index individual trajectories or to index the moving objects, where one moving object might have multiple trajectories.

The operator accepts six parameters, namely input directory, output directory, index type, granularity, trajectory files, and the overwrite flag. The input directory in conjunction with the trajectory files defines the input data. The input directory might contain other files, for instance those containing descriptive attributes. These will be ignored by the operator. The index type can be set to either *Grid* or *R-Tree*. The granularity parameter determines whether to index individual trajectory or moving objects. In the case of trajectory granularity, the input file is expected to have a single identifier column. In the case of moving object granularity, two identifiers are expected: object identifier and a trajectory identifier. Finally, the overwrite flag instructs the operator to first delete existing indexes. If this flag is not set, and older

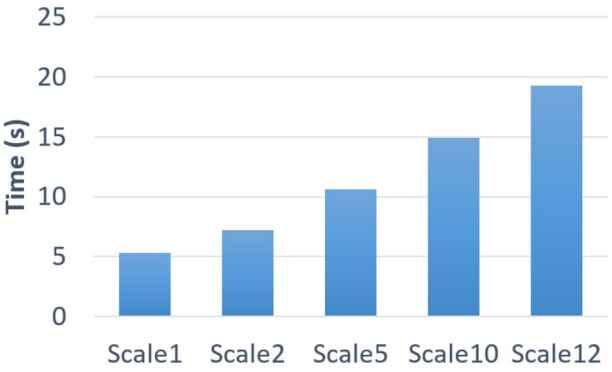

Fig. 6 Scanning the index on different BerlinMod scales

indexes exist, an error will be raised. The next command describes how to use the *IndexCreate* operator:

```
IndexCreate input: <input> output: <output>
indexType: <Grid or R-tree>
granularity: <MO or Traj>
trajFiles: <trajectories files>  -overwrite
```

IndexScan This is the operator for searching the index. It scans the index and extracts the chosen granularity identifiers that have chances to overlap the search window. The index is built on the 3DMBR or the trajectories; hence, a refine step is necessary by evaluating the predicate on the filter results. The resulting identifiers are returned as an in-memory list to the calling operator. Average run times for the different scale factors of BerlinoMOD data are shown in Fig. 6. It accepts three parameters: the input directory, output directory, and (queryRange or point). The input directory points to the index file. The index can be queried either using a range or point. In either cases, it will return the overlapping trajectory identifiers. The syntax of the operator is as follows:

```
indexScan indexInput:<Index> output:<Output>
queryRange:<t1,t2,x1,y1,x2,y2>
(or queryPoint: <x,y,t>)
```

MIndexScan MIndexScan stands for *Multiple IndexScan* is an efficient version of the previous operator whenever it is required to query the index using multiple query ranges and points. Calling this function once, in comparison with calling *IndexScan* multiple times, saves the time of loading the index into the main memory, which would happen many times in the case of calling the IndexScan many times. Additionally, if there are duplicates in the query points/ranges, the *MIndexScan* will scan the index one per different query.

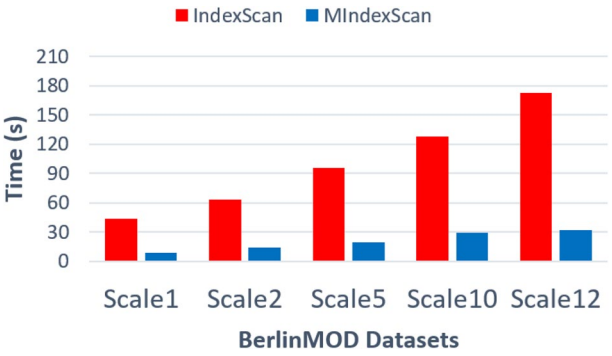

Fig. 7 The comparison between IndexScan and MIndexScan

The comparison in Fig. 7 is between the single call of the *MIndexScan* and the corresponding multiple calls of the *IndexScan* operator. The operator accepts three parameters: input directory, output directory, and (queryRanges or points). The input directory contains the index location. The (points or queryRanges) parameter points to a file that contains the query points or regions. The operator has the following syntax:

```
MIndexScan indexInput:<Index> output:<Output>
queryRanges: <file>
(or queryPoint: <file>)
```

Partition This operator partitions the input files into smaller ones and stores them in HDFS. It is described in detail in Sect. 4.2. The mapping between trajectory identifiers and the files/partitions is stored in a master file. This facilitates accessing the trajectories by their identifiers. The operator accepts three parameters: the input directory, the output directory, and the granularity. Similar to the *IndexCreate*, this operator can perform on two granularities: trajectory and moving object. It has the following syntax:

```
Partition input:<input> output:<output>
granularity: <MO or traj>
```

Link As described before, the trajectory can have descriptive attributes in other files. The *Link* operator splits them into smaller files for the efficient access, in a similar way as the *Partition* operator. It accepts three parameters: input directory, output directory, and granularity. The syntax is as follows:

```
Link input:<input> output:<output>
granularity: <Mo or Traj>
```

IDScan To perform a query on the data of a specific moving object or a specific trajectory, the *IDScan* operator can be used. For example, find all trajectories of vehicle number "1025" from 02:30 to 23:00. The *IDScan* operator is used to identify and fetch the files/partitions that contain that trajectory data, as well as the descriptive attributes. It will then sequentially scan these files and fetch the specific objects of the query. The operator accepts four parameters: the two input directories, the output directory, and the moving object identifier (or trajectory identifier). The input is the partitioned files (i.e., the output directories of the Partition and the Link operators). Following is the syntax:

```
IDScan trajInput:<trajectory input>
descInput:<descriptive input>
<output> Id:<MOId or TrajId>
```

WindowIntersect The operator integrates both the *IndexScan* and the *IDScan* operators. It starts by calling the *IndexScan* operator using the given range or point. For the resulting identifiers, it calls the *IDScan* to fetch the trajectory data of these identifiers. This operator is the most used index access interface. It accepts four parameters: indexInput directory, trajectory directory, queryRange, and queryPoint, as in the following syntax:

```
WindowIntersect indexInput:<index directory>
partitionsInput:<trajectory directory>
output:<output> queryRange:<t1, t2, x1, y1, x2, y2>
(or queryPoint: <x, y, t>)
```

MWindowIntersect The operator is like the previous one, but it can process multiple ranges or multiple points by calling *MIndexScan*. It accepts four parameters: indexInput directory, trajectory directory, queryRanges and queryPoints, as in the following syntax:

```
MWindowIntersect indexInput:<index directory>
partitionsInput:<trajectory directory>
output:<output> queryRanges: <file>
(or queryPoint: <file>)
```

5 MapReduce layer

This section describes the part of the extension that is applied to the Hadoop MapReduce layer. The main role of this layer is to invoke the storage layer operators and prepare the individual tasks for the individual mappers. Figure 8 shows the architecture of this MapReduce layer extension. It consists of two main components: the file splitters and the trajectory record readers.

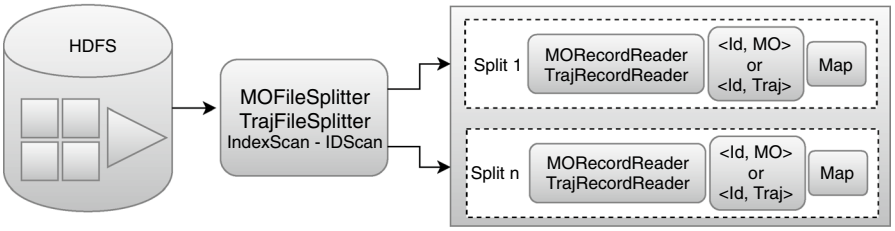

Fig. 8 The MapReduce extension architecture

Hadoop provides a FileSplitter class that is used to split the input files into n splits and distribute them to all mappers. It is a process that runs on the master node of the Hadoop cluster. The TrajectoryFileSplitter is an extension to the Hadoop FileSplitter which understands the file format of the trajectories. Whenever a MapReduce job asks to process big trajectory files, the TrajectoryFileSplitter is invoked to produce n splits of the file and distribute them over the worker nodes. Note that the file format of a trajectory file is fundamentally different than any file format that the Hadoop FileSplitter can deal with. Hadoop file splitters cannot deal with data values that span multiple lines. Our TrajectoryFileSplitter is so flexible to perform this task. As mentioned in Sect. 4.4, different trajectory file formats can be accepted. Accordingly, multiple TrajectoryFileSplitter class had to be implemented. Here, we refer to all of them using the name TrajectoryFileSplitter.

The TrajectoryFileSplitter also understands the global index of the storage layer. That is, instead of operating over the whole input files, it operates over the result of the global index, i.e., *IndexScan* and *IDScan*. The user, while writing a MapReduce task, has the possibility to invoke the global index. The TrajectoryFileSplitter applies the *IndexScan* or the *IDScan* functions on the master file to select the moving objects that will be processed by the mappers based on a query window or a query Id, respectively.

The TrajectoryRecordReader is a process that runs on every mapper. It will parse the received split into key value pairs, where the key is the trajectory Id and the value is the trajectory data. This is a necessary step that is required before the mapper function starts. The TrajectoryRecordReader extends the Hadoop RecordReader so that it can understand the trajectory file format. Again, we have implemented multiple TrajectoryRecordReader classes to deal with the different possible trajectory file formats. All the trajectory file splitters and the trajectory record readers have been implemented in two variants: one for the trajectory granularity and the second for the moving object granularity.

6 Operation layer

The combination of the storage layer extension with the new features in the MapReduce layer extension allows for efficient realization of many moving object operators. These operators are themselves MapReduce jobs. They can be used in

the users' MapReduce code to express more complex tasks. In this section, we describe some representative operators.

6.1 *Passes*

The *Passes* operator is a MapReduce job used to find all moving objects that have passed a specific spatial point. It calls the *WindowIntersect* operator to filter the trajectories using the index and stores the candidate trajectories on desk. Then it creates a new job, which is responsible for checking and retrieving the trajectories which passes the point (i.e., the refine step). This job consists of one mapper class. If the granularity is at the object level, this operator will retrieve and match all the trajectories that are related to this moving object and checks whether any of them passes the spatial point. In *trajectory granularity*, a bounding box is calculated for every trajectory, which increases the efficiency of the index. The operator accepts five parameters called indexInput, partitionsInput, output, point, and shape. The indexInput parameter is the HDFS directory that contains the index files. The partitionsInput parameter points to the partitions in HDFS. The point parameter is the spatial point on which the passes condition is checked. The syntax is as follows:

```
Passes indexInput:<Input>
partitionsInput:<Input> output:<output>
point:<x,y,t>
```

6.2 *Trajectory*

Trajectory operator is used to retrieve the specific moving object trajectory based on the identifier. It retrieves all the segments of a moving object trajectory. This operator calls the *IDScan* operator to access the storage layer for retrieving the trajectory segments. This operator can be applied on both object and trajectory granularities. For the object granularity, it produces the spatial projection of all the trajectories of the given object. For the trajectory granularity, it projects only the asked trajectory. This operator takes three or four parameters: input directory, output directory, the moving object identifier, or (the trajectory identifier). Following is the syntax:

```
Trajectory partitionsInput: <Input> output:<Output>
objectID (or trajID): <Identifier>
```

6.3 *Length*

The *Length* operator is used to compute the driving distance of the moving object trajectory. It is the path that a moving object follows through space as a

function of time. The operator firstly calls the *IDScan* operator for retrieving the segments of a moving object trajectory. Then, it creates a job for calculating the length of this trajectory. This job consists of one mapper and reducer classes. The mapper class is responsible for calculating the length of an individual segment, and the reducer class is responsible for summing the lengths of all trajectory segments. This operator can again work on both granularities: object and trajectory. This operator takes three or four parameters: input directory, output directory, the moving object identifier, or (the trajectory identifier). The input represents the partitions directory stored in HDFS. The trajectory identifier parameter is used to define a specific trajectory. The syntax

```
Length partitionsInput: <Input> output:<Output>
objectID (or trajID): <Identifier>
```

6.4 *ZoneTraj*

This operator is used to retrieve set of moving objects in a specific region surrounding a point of interest. It is very similar to the *Passes* operator except that the query argument is a circle defined by the user. The query zone is passed as x, y, r, t where x and y are the coordinates of the center of attention, r is the radius of the coverage, and t is the timestamp. The operator calls the *WindowIntersect* operator to retrieve all the objects in the zone. This operator is useful in discovering the vehicles that are in an accident place. It works both in the object granularity and in the trajectory granularity. The operator takes the four inputs called indexInput, partitionsInput, output directories, and zone attribute. The value of zone attribute is defined as x,y, and radius. The following is the syntax:

```
ZoneTraj indexInput:<Input>
partitionsInput:<Input>
output:<output> zone:<x,y,radius,t>
```

7 Using HadoopTrajectory

This section illustrates the use of HadoopTrajectory in the MapReduce jobs. First, the user needs to decide the data access method, either processing the whole input files or using one of the index access methods of the storage layer. Clearly, this decision depends on the task to be implemented. Most of the spatiotemporal filtering tasks can benefit the *IndexScan* operator.

The first example answers the query: what is the driving distance that each of the visitors of a shopping mall had to drive, assuming that we have all the car trajectories of the city. Such a query can be expressed by aggregating the length of trajectories that passed the mall location. Since all the operators needed to express this query are implemented in HadoopTrajectory, the user function is simply a sequence of operator calls as follows:

```
AnalyzeTrajectories(TrajectoryDirectory, IndexDirectory, QueryPoint)
{
    IndexScan(IndexDirectory, IndexScanOutDirectory, QueryPoint);
    IDScan(TrajectoryDirectory, IDScanOutDirectory,
           IndexScanOutDirectory);
    Passes(IDScanOutDirectory, PassesOutDirectory, QueryPoint);
    Length(PassesOutDirectory, TrajLengthOutDirectory);
}
```

In this example, the *Passes* operator will be used; thus, it makes sense to use the *IndexScan* access function. Its result is passed to the *IDScan* operator for fetching the data of all the trajectories that pass the query point. Then the *Length* operator is called on these trajectories to compute their travel distance in meters. Because the operators *Passes, Length* exist in HadoopTrajectory, they can be directly called in the user code, without the need to specifically write MapReduce jobs for them. Indeed calling each of them will initialize a MapReduce job in the back-end.

The second example analyzes the trajectories with the goal of extracting trip information (start point, end point, start time, end time). For this, there is no direct operator in HadoopTrajectory, rather a MapReduce job has to be implemented by the user using HadoopTrajectory types and primitive operations.

```
ExtractTripInformation( InputDirectory, AnalysisOutDirectory )
{
        JobConf conf=new JobConf("job.class");
        conf.setJobName("job");
        conf.setJarByClass(job.class);
        conf.setInputFormat(TrajInputFormat.class);
        TrajInputFormat.addInputPath(conf,
           new Path(InputDirectory));
        conf.setOutputFormat(TextOutputFormat.class);
        TextOutputFormat.setOutputPath(conf,
           new Path(AnalysisOutDirectory));
        conf.setMapperClass(AnaysisMapper.class);
        conf.setOutputKeyClass(LongWritable.class);
        conf.setOutputValueClass(Text.class);
        conf.setNumReduceTasks(0);
        JobClient.runJob(conf);
}
AnalysisMapper(K, V)
{
    String result=V.getStartPoint();
    result+=","+V.getLastPoint();
    result+=","+V.segments.get(0).timeStart;
    result+=","+V.size();
    Text text=new Text(result);
    outputCollector.collect(K,text);
}
```

It sets the configurations of a MapReduce job, i.e., the object *conf*, and then runs the job. The mapper function is illustrated in the *AnalysisMapper* code segment. It basically computes trajectory attributes such as the starting point, ending point, etc.

8 Experimental evaluation

This section experimentally evaluates HadoopTrajectory on different datasets and different analysis tasks. In the following, three experiments are presented and discussed.

For the purpose of comparison, four environments are setup; Distributed SECONDO, standard Hadoop, the HadoopTrajectory, and ST-Hadoop. A dedicated cluster is used in all the experiments. It consists of three nodes. Two of them have quad cores i7-3770 CPU 3.40 GHz, 32 GB RAM, HDD of 1 TB running Ubuntu 14.04. The third node has 2 core i5-4210U CPU 2.40 GHz, 8 GB RAM, running Ubuntu 14.04.

To make a comprehensive comparison, we use two different datasets: (1) BerlinMOD benchmark dataset (Düntgen et al. 2009) as synthetic data and (2) the taxi trajectory dataset as real data (https://www.kaggle.com). These file formats of these datasets are quite different, yet our HadoopTrajectory contains the methods that can deal with them. On the two datasets, we execute a set of queries that will be described in more detail in the coming section.

BerlinMOD is a benchmark for moving object databases. It contains a data generator that simulates the movement of cars in the city of Berlin and records their tracks. The generator simulated the commutes from home to work and back to home. With some probability, the cars also do leisure trips. The simulator is adjusted in many ways to mimic real-life scenarios. The size of the generated data can be configured with a parameter called scale factor as described in Sect. 4.2. Additionally, BerlinMOD has a set of 26 benchmark queries covering the functions of a moving object database system; selections, joins, nearest neighbor, etc. In this experiment, we evaluate 8 queries for which the required operators have been implemented.

BerlinMOD generates a table *Cars* and another table *Trips*. The first stores a single trajectory per vehicle that represents its complete track over all the simulation days (i.e., object granularity). The *trips* table stores one trajectory per every individual trip of the vehicle (i.e., trajectory granularity). Other look-up tables are created to form the queries as follows:

- *QueryLicences* 100 vehicle licenses chosen at random from the base table of car data. It contains vehicle identifier, license, car type, and car model.
- *QueryPoints* 100 point chosen at random from trips data table. Thus each point is traversed by at least one trip.
- *QueryRegions* 100 random regions (convex polygons) with varying sizes in Berlin.
- *QueryInstants* 100 instant chosen at random that fall in the simulation timespan.

- *QueryPeriods* 100 temporal intervals generated at random that overlap the simulation timespan.

The taxi trajectory dataset contains the GPS observations of a complete year (from 01/07/2013 to 30/06/2014) of 442 taxis running in the city of Porto, in Portugal (https://www.kaggle.com). These taxis operate through a taxi dispatch central, using mobile data terminals installed in the vehicles. The dataset contains 1,710,671 trajectories and has a disk storage size of 1.9 GB.

8.1 Query efficiency

We compare four platforms in terms of execution time: (1) HadoopTrajectory, (2) Standard Hadoop, (3) Distributed SECONDO (Nidzwetzki and Güting 2015), and (4) ST-Hadoop (Alarabi and Mokbel 2017). The used Hadoop version is 2.7.3. The Distributed SECONDO version was downloaded from the SECONDO Website[1] on 6/7/2017. The used ST-Hadoop version was downloaded from GitHub[2] on 10/8/2018.

Both HadoopTrajectory and Distributed SECONDO can represent trajectories and manipulate them. Therefore, we compare the two frameworks using BerlinMOD queries and data. Comparing with standard Hadoop has been accomplished by using the data types of our extension, yet without the other components of indexing, partitioning, and linking. The goal is to evaluate the gain of the index. We also compare with ST-Hadoop (Alarabi and Mokbel 2017). As ST-Hadoop does not implement a type for trajectories, and rather uses a timestamped point type, it was not possible to express all the experimental queries on it, so we discuss it later in this section.

BerlinMOD data is generated at different scale factors starting from 1 (5 GB) to 12 (63 GB). The goal is to test the index efficiency using windows of different sizes. First, we compare the index creation process in HadoopTrajectory and Distributed SECONDO. Index creation in our HadoopTrajectory takes less time than Distributed SECONDO for big datasets as illustrated in Fig. 9. Note that the indexing, partitioning and linking can all run in parallel, which makes the whole process quite efficient.

Figure 10 illustrates the run times of the following BerlinMOD queries:

(a) What are the models of the vehicles with license plate numbers from QueryLicences? It is not a spatiotemporal query.
(b) Where have the vehicles with licenses from QueryLicences1 been at each of the instants from QueryInstants1? It is a spatiotemporal query.
(c) Which license plate numbers belong to vehicles that have passed the points from QueryPoints? It is a spatiotemporal query.

[1] http://dna.fernuni-hagen.de/ secondo.
[2] https://github.com/lmarabi/st-hadoop.

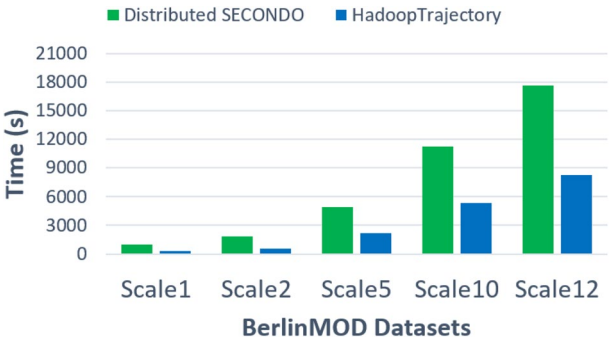

Fig. 9 Time to index different BerlinMOD scale factors

(d) What is the longest distance that was travelled by a vehicle during each of the periods from QueryPeriods?. It is a spatiotemporal query.
(e) List the pairs of licenses for vehicles, the first from QueryLicences1, the second from QueryLicences2, where the corresponding vehicles are both present within a region from QueryRegions1 during a period from QueryPeriod1, but do not meet each other there and then.? It is a spatiotemporal query.
(f) Which points from QueryPoints have been visited by a maximum number of different vehicles?. It is a spatiotemporal query.

As illustrated in Fig. 10a, distributed SECONDO takes less time than the other two solutions because the Hadoop-based solutions have the high overhead of distributing the required trajectories to the worker nodes. For simple queries such as this one, the overhead is more dominant over the distribution benefit.

Figure 10b shows the ability of our framework to apply the aggregation functions by using filtering methods before running these aggregations. Hadoop takes the largest time because it needs to check all moving objects trajectories. As shown in Fig. 10c, the proposed HadoopTrajectory framework utilizes the HDFS extension to achieve the best performance and minimize the running time. It is similar to query (d), but this query contains more operations. After filtering the moving objects, we need to calculate the length of each trajectory which meet the conditions. Then, we apply max function to get the largest distance between all moving objects.

In Fig. 10d, our HadoopTrajectory takes less time than distributed SECONDO and Hadoop. The implementation of the query in Distributed SECONDO is very complicated and it needs to call many operations and to use many indexes. Figure 10e demonstrates the *MIndexScan* operator. This operator filters all moving objects data and selects objects that overlap with the points in QueryPoints file.

On the taxi trajectory dataset, the following analysis tasks have been implemented:

- What is the longest trip?
- What are the stop points for each trip?
- How many trips starts from the city center, another place or random street?

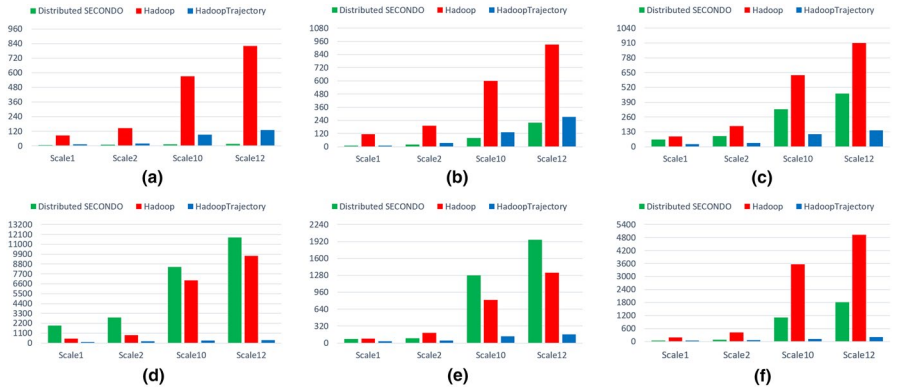

Fig. 10 The run times of the six BerlinMOD queries on the three platforms

- How many trips occurred based on phone call?
- What is the average number of trips per day?

These tasks have been executed for different durations; a specific day, a month, or a year. They have also been executed for different grouping levels; per vehicle and all trips. For instance, the first analysis task when executed for a January 2014 and the vehicle grouping level reads as follows: What is the longest trip per taxi vehicle, during January 2014. When the same task is executed for all trips, it reads as follows: what is the longest trip among all the input trips during January 2014.

Figure 11a illustrates the average run times of the five analysis tasks, on the vehicle grouping level. The comparison is done between standard Hadoop and HadoopTrajectory. The run time comparison for the all trips grouping level is illustrated in Fig. 11b. As expected, the proposed extension out performs the standard Hadoop, thanks to the indexing, partitioning, and linking components.

Another comparison between the proposed extension and ST-Hadoop, we implemented this comparison by using one node. The specifications are Intel i7-7800 3.4 GHz of 6 cores, 32 GB DDR4 memory, and 1 TB SSD storage. We use Hadoop

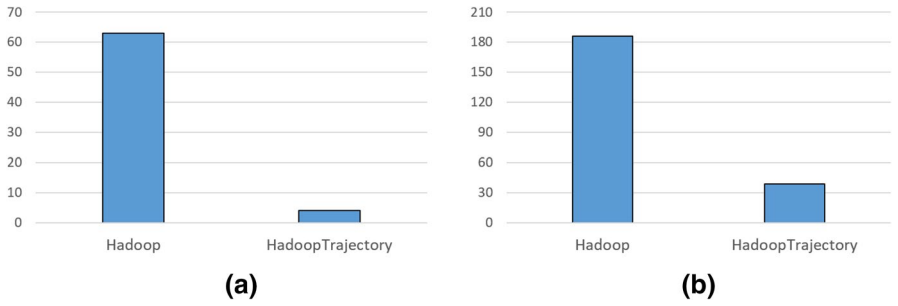

Fig. 11 **a** Runtime of the query *find the longest trip per taxi vehicle*, **b** runtime of the query *find the longest trip among all the input trips*

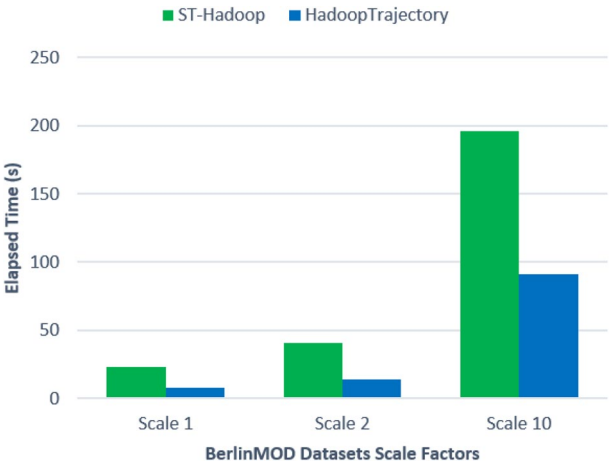

Fig. 12 Runtimes of query (c) by using different dataset sizes

Table 2 Expressiveness comparison with ST-Hadoop

	ST-Hadoop	HadoopTrajectory
Types	STPoint, STPointTweets, STPointTrajectory, Slice, Point	MovingObject, Trajectory, TrajSegment, TrajData, Points, Regions, Periods, Interval, Instant, Point
Operators	STJoin, STJoins, STRangeQuery, STHash	Passes, Traj, Length, ZoneTraj, AtInstant, AtPeriod, DefTime, WindowIntersect, Present, Partition, Link, IDScan, IndexScan, MIndexScan, MWindowIntersect

2.7.2 and Java 1.8 running on Ubuntu 18.04. We have implemented query (c) in ST-Hadoop and compared it with HadoopTrajectory. This was the only query among our experimental queries that could be implemented with the available types and operations in ST-Hadoop.

Figure 12 shows that HadoopTrajectory outperforms ST-Hadoop in processing such those kinds of trajectories. This query requires to load QueryPoints and DataMCar files to find what are the moving objects that have passed each point in the QueryPoints file. In HadoopTrajectory, this maps to an *MIndexScan* followed by the *Passes* predicate. In ST-Hadoop, the query is implemented as a MapReduce job, where the reduce is not really required. The mapper called the ST-Hadoop index scan operator (STRangeQuery) for every point in QueryPoints to retrieve the candidate objects. We had to implement in the mapper the logic of *line intersects point* to check whether a query point falls on the interpolated line between two trajectory points. We had to do this, as ST-Hadoop does not provide a passes operator nor

a trajectory data type. Table 2 lists the supported types and operators in both ST-Hadoop and HadoopTrajectory.

9 Conclusion

In order to process massive amounts of moving objects data, a HadoopTrajectory is proposed in this paper. Initially, we have carried out an extensive survey of the literature, which confirmed our initial hypothesis that there is a lack of support for big spatiotemporal data processing. Accordingly, we worked to fill-in this gap by extending Hadoop with trajectory data types. The standard behavior of HDFS would distribute all the moving object data of the big input file to the cluster nodes. In many cases, it is possible to filter the trajectories beforehand by means of indexing and ignore those that have no chance to contribute to the result. Accordingly, a 3DR-tree index was implemented. Again, it had to be carefully integrated into the Hadoop framework. We have extended all the Hadoop framework layers to accommodate the index. The index itself and the associated physical file management routines have been added to the HDFS storage layer. The MapReduce layer has been extended with index access and management methods, that can be invoked from within the operators and the user programs. Our spatiotemporal data management operators have been further extended to make use of this indexing infrastructure. Finally, we have rewritten the BerlinMOD queries using this optimization and extensively evaluated the performance gain, which showed a notable advantage.

References

Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J (2013) Hadoop gis: a high performance spatial data warehousing system over mapreduce. Proc VLDB Endow 6(11):1009. https://doi.org/10.14778/2536222.2536227

Alarabi L, Mokbel M (2017) A demonstration of st-hadoop: a mapreduce framework for big spatio-temporal data. Proc VLDB Endow 10(12):1961

Bakli MS, Sakr MA, Soliman THA (2018) A spatiotemporal algebra in hadoop for moving objects. Geospatial Inf Sci 21(2):102. https://doi.org/10.1080/10095020.2017.1413798

De Almeida VT, Güting RH (2005) Indexing the trajectories of moving objects in networks. Geoinformatica 9(1):33. https://doi.org/10.1007/s10707-004-5621-7

Düntgen C, Behr T, Güting RH (2009) Berlinmod: a benchmark for moving object databases. VLDB J 18(6):1335. https://doi.org/10.1007/s00778-009-0142-5

Eldawy A, Mokbel MF (2015) In: 2015 IEEE 31st international conference on data engineering, pp 1352–1363. https://doi.org/10.1109/ICDE.2015.7113382

Environmental systems research institute (ESRI). https://www.esri.com/. Accessed 15 Apr 2018

Forlizzi L, Güting RH, Nardelli E, Schneider M (2000) In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, SIGMOD '00. ACM, New York, pp 319–330. https://doi.org/10.1145/342009.335426

Fox A, Eichelberger C, Hughes J, Lyon S (2013) In: 2013 IEEE international conference on big data, pp 291–299. https://doi.org/10.1109/BigData.2013.6691586

Frentzos E (2003) In: Proceedings of the 8th international symposium on spatial and temporal databases (SSTD). Springer, pp 289–305

Grumbach S, Rigaux P, Scholl M, Segoufin L (1998) DEDALE, a spatial constraint database. Springer, Berlin, pp 38–59

Gting RH, Behr T, Almeida V, Ding Z, Hoffmann F, Spiekermann M (2004) Secondo: an extensible dbms architecture and prototype. Technical report, FernUni-Hagen

Güting RH, Lu J (2015) Parallel secondo: scalable query processing in the cloud for non-standard applications. SIGSPATIAL Spec 6(2):3. https://doi.org/10.1145/2744700.2744701

Güting RH, Böhlen MH, Erwig M, Jensen CS, Lorentzos NA, Schneider M, Vazirgiannis M (2000) A foundation for representing and querying moving objects. ACM Trans Database Syst 25(1):1. https://doi.org/10.1145/352958.352963

Hadoop http://hadoop.apache.org/. Accessed 15 Apr 2018

Ma Q, Yang B, Qian W, Zhou A (2009) In: Proceedings of the first international workshop on cloud data management, CloudDB '09. ACM, New York, pp 9–16. https://doi.org/10.1145/1651263.1651266

Nidzwetzki JK, Güting RH (2015) In: Advances in spatial and temporal databases. Springer, pp 491–496

Pelekis N, Theodoridis Y, Vosinakis S, Panayiotopoulos T (2006) Hermes—a framework for location-based data management. Springer, Berlin, pp 1130–1134

Pfoser D, Jensen CS, Theodoridis Y (2000) In: Proceedings of the 26th international conference on very large data bases, VLDB '00. Morgan Kaufmann Publishers Inc., San Francisco, pp 395–406. http://dl.acm.org/citation.cfm?id=645926.672019. Accessed 15 Apr 2018

Taxi trajectory analytics. https://www.kaggle.com. Accessed 15 Apr 2018

Theodoridis Y, Vazirgiannis M, Sellis T (1996) In: Proceedings of the third IEEE international conference on multimedia computing and systems, pp 441–448. https://doi.org/10.1109/MMCS.1996.535011

Yang B, Ma Q, Qian W, Zhou A (2009) In: Proceedings of the 14th international conference on database systems for advanced applications, DASFAA '09. Springer, Berlin, pp 768–771

Affiliations

Mohamed Bakli[1] · Mahmoud Sakr[2,3] · Taysir Hassan A. Soliman[1]

Mahmoud Sakr
mahmoud.sakr@cis.asu.edu.eg

Taysir Hassan A. Soliman
taysser.soliman@fci.au.edu.eg

[1] Assiut University, Asyut, Egypt

[2] Ain Shams University, Cairo, Egypt

[3] Université libre de Bruxelles, Bruxelles, Belgium