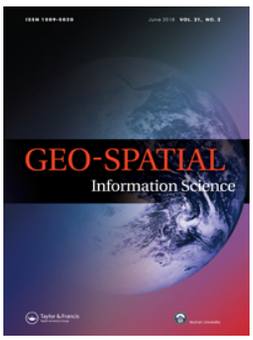

Geo-spatial Information Science

ISSN: 1009-5020 (Print) 1993-5153 (Online) Journal homepage: http://www.tandfonline.com/loi/tgsi20

A spatiotemporal algebra in Hadoop for moving objects

Mohamed S. Bakli, Mahmoud A. Sakr & Taysir Hassan A. Soliman

To cite this article: Mohamed S. Bakli, Mahmoud A. Sakr & Taysir Hassan A. Soliman (2018) A spatiotemporal algebra in Hadoop for moving objects, Geo-spatial Information Science, 21:2, 102-114, DOI: 10.1080/10095020.2017.1413798

To link to this article: https://doi.org/10.1080/10095020.2017.1413798

GEO-SPATIAL INFORMATION SCIENCE, 2018
VOL. 21, NO. 2, 102–114
https://doi.org/10.1080/10095020.2017.1413798

Taylor & Francis
Taylor & Francis Group

A spatiotemporal algebra in Hadoop for moving objects

Mohamed S. Bakli[a], Mahmoud A. Sakr[b,c] and Taysir Hassan A. Soliman[a]

[a]Faculty of Computers and Information, Assiut University, Assiut, Egypt; [b]Faculty of Computer and Information Sciences, Ain Shams University, Cairo, Egypt; [c]Université libre de Bruxelles, Bruxelles, Belgium

ABSTRACT

Spatiotemporal data represent the real-world objects that move in geographic space over time. The enormous numbers of mobile sensors and location tracking devices continuously produce massive amounts of such data. This leads to the need for scalable spatiotemporal data management systems. Such systems shall be capable of representing spatiotemporal data in persistent storage and in memory. They shall also provide a range of query processing operators that may scale out in a cloud setting. Currently, very few researches have been conducted to meet this requirement. This paper proposes a Hadoop extension with a spatiotemporal algebra. The algebra consists of moving object types added as Hadoop native types, and operators on top of them. The Hadoop file system has been extended to support parameter passing for files that contain spatiotemporal data, and for operators that can be unary or binary. Both the types and operators are accessible for the MapReduce jobs. Such an extension allows users to write Hadoop programs that can perform spatiotemporal analysis. Certain queries may call more than one operator for different jobs and keep these operators running in parallel. This paper describes the design and implementation of this algebra, and evaluates it using a benchmark that is specific to moving object databases.;

ARTICLE HISTORY
Received 31 December 2016
Accepted 12 November 2017

KEYWORDS
Spatiotemporal algebra; Hadoop; MapReduce; moving objects

1. Introduction

The tremendous increasing volume of spatiotemporal data presents challenges in storing, managing, processing, and analyzing data. Spatiotemporal data are produced by various devices, such as smart phones, satellites and telescopes, medical devices, and traffic analysis. Some projects are dealing with big spatial data, such as the Blue Brain Project (Markram 2006), which studies the brain's architectural and functional principles through modeling brain neurons as spatial data. In addition, epidemiologists use spatial analysis techniques to identify cancer clusters (Pickle et al. 2006) and track infectious disease (Auchincloss et al. 2012). Spatiotemporal data projects are also applied in motion planning of intelligent vehicles (González et al. 2016). Other projects include visual exploration of big spatiotemporal urban data (Ferreira et al. 2013), location-based social networks (Cao et al. 2015), and spatial crowd-sourcing (Chung et al. 2015).

In terms of exploring cloud computing technologies, geographic information system (GIS) is moving ahead from traditional GIS to cloud GIS, as in the case of ESRI. In addition, Chen, Wang, and Shang (2008) built a high-performance workflow system MRGIS, which used MapReduce clusters to execute GIS applications efficiently. Park et al. (2010) used Hadoop distributed file system (HDFS) and MapReduce on massively parallel processing of 3D GIS data, this method reduces the processing time by a very high factor. Aji et al. (2013) presented Hadoop GIS, depending on a spatial data warehousing system, to tackle large-scale spatial queries. Song et al. (2015) built a spatiotemporal cloud platform, which uses HDFS for managing image data, MapReduce computing service, and workflow of high-performance geospatial analysis, to provide optimization scaling algorithms and visualization.

Many researchers also took the advantage of the MapReduce environment in supporting large-scale spatial data, such as Hadoop GIS (Aji et al. 2013), Parallel SECONDO (Güting and Lu 2015; Lu and Güting 2012), and MD-HBase (Nishimura et al. 2013). The main drawbacks of these previous systems are the lack of integration with the core of Hadoop. Eldawy (2014), Eldawy and Mokbel (2014, 2015) proposed SpatialHadoop which is an open source framework. SpatialHadoop has the following advantages: (1) handling spatial data, (2) having a simple high-level language with spatial data types and operations, (3) analyzing data, and (4) handling efficient indexing. For the very high need of spatiotemporal data in the big data era, these spatiotemporal elements will be proposed as an extension to SpatialHadoop (Eldawy and Mokbel 2015).

CONTACT Mohamed S. Bakli mohamed_bakli@aun.edu.eg

Much of the research in this field focuses on defining data models for continuously changing geometric objects, typically known as moving objects. To handle big spatiotemporal data, it is natural to use distributed computing technologies, such as Hadoop MapReduce and HDFS. In this case, a query has two forms: a spatial form and a temporal form. The query can be done by the period of time through which one can retrieve the spatial objects done during this time. It also can be done for the spatial object to find the period of time. Thus, the efficient techniques for spatiotemporal data processing are highly required. There are two main approaches: (1) extending Hadoop with spatiotemporal functionality, as in SpatialHadoop of Eldawy and Mokbel (2015), (2) extending a spatiotemporal database management system (DBMS) with Hadoop support, as in Parallel SECONDO of Lu and Güting (2012).

In this paper, an extension of Hadoop is proposed in the form of a spatiotemporal algebra, having the following main contributions:

(1) Extending the Hadoop type system with moving object types.
(2) Extending Hadoop with spatiotemporal operators.
(3) Demonstrating the utility of the proposed algebra using the BerlinMOD benchmark (Düntgen, Behr, and Güting 2009).

The rest of this paper is organized as follows. First, it reviews the closely related work about this study; and explains the moving object databases (MOD) model that we implement in this paper, and the benchmark used in the evaluation. Then the proposed Hadoop extension is explained. Finally, it evaluates the performance of the proposed algebra, and is compared with the SECONDO system.

2. Related work

There have been few proposals for MOD in the literature. MOD system provides database capabilities for spatiotemporal data, including storage, query, user interface, etc. Probably, the most mature MOD so far is the SECONDO system (Güting et al. 2004). It implements the moving object model in (Güting et al. 2000). It consists of a kernel, a GUI, and a query optimizer that allows for SQL-like queries. SECONDO kernel contains various query algebras that cover other spatial and spatiotemporal processing. Recently, two versions of SECONDO are proposed for parallel processing: Parallel SECONDO (Lu and Güting 2012), and Distributed SECONDO (Nidzwetzki and Güting 2015).

Parallel SECONDO architecture consists of multiple nodes, each of which has a stand-alone SECONDO installation. Hadoop is used only for scheduling the tasks over the SECONDO nodes, and monitoring the execution. Queries to the parallel SECONDO are received by a master node, also running SECONDO, and re-written into parallel queries that divide the processing of the whole query into pieces. Hadoop is then used to schedule these pieces over the SECONDO nodes in the cluster. The final result has to be collected back from the nodes contributing to the query. Distributed SECONDO, on the other hand, does not build on Hadoop. It replaces the SECONDO storage layer, which is based on BerkelyDB (Olson, Bostic, and Seltzer 1999), by Casandra. Hence, it is highly available and allows for fast updates. The management and the query processing are performed by SECONDO nodes.

Pelekis et al. (2006) developed the Hermes MOD, which extends on top of PostgreSQL by spatiotemporal types and operators. The moving object model of Hermes is similar to that of SECONDO. Yet, the implementation of Hermes is not yet mature. For instance, it approximates regions by their bounding boxes. The scalability of Hermes, hence, is equivalent to that of PostgreSQL. GeoMesa (Fox et al. 2013) uses Geohash code and builds a spatiotemporal index structure on the top of Apache Accumulo. It achieves acceptable results when storing data using 35-bit Geohash strings. However, the performance depends on the number of Geohash characters in the row key, and on the resolution level.

Few other proposals exist to add spatiotemporal data types on top of extensible database system. Raza (2012) proposes an approach to modeling space and time to provide the basis for implementing a temporal GIS. The approach that depends on new spatiotemporal data type is called spatiotemporal type (STT) built on top of PostgreSQL. STT is an extension of the STGeometry type for PostgreSQL. It contains constructors and functions supporting spatial and spatiotemporal queries. In addition, MD-HBase (Nishimura et al. 2013) extends HBase (a non-relational database runs on top of Hadoop to support multi-dimensional indexes (HBase 2012)).

DEDALE (Grumbach et al. 1997) is an object-oriented framework of a constraint DBMS for spatiotemporal information. It provides an abstract and non-specialized data model and query language for representing the geographic objects. The data model allows a uniform representation of n dimensional data, such as geometric objects and spatiotemporal data. It is implemented on the top of O_2 DBMS. The spatial objects have an identifier, thematic and spatial attributes expressed as a constraint relation. It consists of kernel, Basis with storage capabilities, and a set of relational algebra operators.

Natalija and Dragan (2015) had developed a distributed application that performs spatial join between trajectory data-set and spatial areas of interests. Then, aggregations join the results to provide analysis of movement. They use parallel/distributed programming frameworks, message passing interface (MPI), and open multi-processing (OpenMP) as programming interfaces. MPI is used for process communication between

multicore nodes; OpenMP is used for thread communication within each multicore node. While PRADASE (Ma et al. 2009) is a new framework that supports spatial query processing over trajectory data based on MapReduce in a computer cluster. The execution engine is built on MapReduce which can simplify many computational data-intensive tasks. The master node is responsible for storing the keys to data on slave nodes and searching by key. Then the data can be returned automatically if the key is given.

Yang et al. (2009) proposed a distributed trajectory data processing system called TRUSTER. It is built on the top on Hadoop and designed to manage large amount of trajectory data in a distributed environment. TRUSTER partitions the whole trajectory data-set into sets of static partitions according to the predefined spatial grid. These partitions are distributed across multiple nodes. For each partition, index is just built over temporal dimension for all trajectories in the partition.

Yet none of these works is cloud native. They are built on top of existing database systems, thus this limits the applications to the data models and application architectures posed by these database systems. Hadoop, on the other hand, is a very flexible framework for building applications on the cloud. Many existing applications nowadays are built on Hadoop and need to benefit from spatiotemporal processing capabilities. This calls for a Hadoop spatiotemporal extension.

The work that had inspired our work is the SpatialHadoop (Eldawy 2014; Eldawy and Mokbel 2015). It extends Hadoop with spatial types and operators. Since its proposal, it has been used in many real-world applications and research works (SpatialHadoop 2013). SpatialHadoop can be seen as a library of spatial types and operators that can be called within Hadoop MapReduce programs. Additionally, spatial indexes are included to speed up the data transfer between HDFS and the work nodes, and to speed up the processing inside the work nodes. This paper adopts a similar approach for spatiotemporal types and operators.

3. Technique background

This section gives a concise description of other works that are used as basis of this paper.

3.1. Hadoop

Hadoop (Apache Software Foundation 2010) is an open-source MapReduce implementation for processing big volumes of data in a distributed manner on large clusters. It is freely available and licensed under the Apache License 2.0. MapReduce is a programming model that follows a certain functional style to process large data-sets. HDFS is primarily a file system similar to many of the already existing ones. However, it is also a virtual file system. MapReduce and HDFS are the main components of Hadoop.

With MapReduce, it is very easy to develop scalable parallel programs to process data-intensive applications on clusters of commodity machines. Data in MapReduce are organized into files and directories and stored in the distributed file system, like the Google file system (GFS) or HDFS. Files are divided into uniform sized blocks, and distributed across cluster nodes. Blocks are replicated to handle hardware failure.

3.2. Data model

The proposed algebra implements the data model as in that of Güting et al. (2000). Abstractly, a moving object is a function from time to value. The value can be scalar of spatial. This model defines abstract data types for moving objects such as mpoint, mreal, mbool, etc. An mpoint object (stands for moving point) represents a spatial point that moves (e.g. car, animal, etc.). An mreal represents a scalar number that changes with time (e.g. temperature, speed of a car, etc.). An mbool represents a time-dependent Boolean. It is useful for representing the result of time-dependent predicate (e.g. the speed of car below 60 km/h). The answer of such a predicate is true for some time intervals and false during the rest.

This model is implemented using the so called sliced representation of moving object (Forlizzi et al. 2000). Basically, the moving object is represented as a set of non-overlapping temporal units. Every unit describes the motion during its associated time interval. Within a single unit, the motion is approximated to a constant, linear, or quadratic function. The list of non-overlapping units together constructs an approximation of the whole movement. An example (consisting of two units, each has a time interval and a linear function) of an mpoint object can be represented as follows:

```
{[2015-05-16-07:30:00-2015-05-16-07:30:05]
(50 30 100 130)
[2015-05-16-07:30:05-2015-05-16-07:30:08]
(100 130 90 100)
}
```

The spatiotemporal types are used to represent results from operations on moving point types or inputs to these operations. An abstract representation allows to represent moving point as a continuous curve in the 3D space. The data model allows the integration of abstract data types involving: base types (int, real, bool, string), spatial types (point, region), time types (instant), temporal (moving), and spatiotemporal types (mpoint, mreal, and mbool). Embedding this in a DBMS and building a query language allows the use of a query language for spatiotemporal data and moving objects. Its data model is implemented in SECONDO DBMS.

The proposed Hadoop extension differs strongly from SECONDO. SECONDO is implemented as DBMS, but Hadoop extension is used as a backbone for big spatiotemporal data analytical processing. Also, the other versions of SECONDO use Hadoop only for scheduling the tasks. In the extension, we define the algebra in the core of Hadoop, and modify the HDFS splitters and formatters for reading different kinds of moving object. Also, we make a layer on the top of HDFS which helps in linking the data-set files. SECONDO relies on the implementation of the algebra, and manages queries inside DBMS to take the advantages of Database. SECONDO is built on the top of BerkeleyDB, but the proposed extension storage system is built inside HDFS. SECONDO has a relation between geometric objects. Also our extension builds a relation between the input files, and can query them in a simple way. A relation can be partitioned into the slave nodes using HDFS new splitters. When the data becomes huge, SECONDO has serious problems which do not exist in our Hadoop extension. Our extension reduces I/O access and differentiates jobs among the master node and the slave nodes. This will be further explained in the result section at scale factor 10 with size 561 GB.

3.3. BerlinMOD benchmark

BerlinMOD (Düntgen, Behr, and Güting 2009) is a benchmark for MOD systems. It is used in many research works. It consists of a data generator, based on SECONDO, and an extensive set of benchmark queries. The data generator simulates a number of cars driving on the road network of Berlin for a given period of time (e.g. a month), and captures their positions at least every two seconds. Some random events are added to the simulation to simulate real-time events that might disturb the car path. The user can create BerlinMOD data-sets of arbitrary sizes by setting a scale factor. For example, when it is set to one, trajectories of 2000 vehicles running on the street network of Berlin in 28 days are simulated, yielding 11 GB data.

BerlinMOD benchmark represents vehicle movements and defines six relations containing random spatial and temporal data to build query points and ranges within the benchmark queries. Its queries cover multiple classes: aggregate/non-aggregate, static/temporal, etc. We use BerlinMOD in many places during the paper to evaluate the performance of our algebra.

4. Methodology

4.1. Hadoop extension

Our Hadoop extension is used to support spatiotemporal processing. This extension is not a layer on top of Hadoop. Figure 1 abstracts how it is integrated into the Hadoop framework. The algebra types are added to the

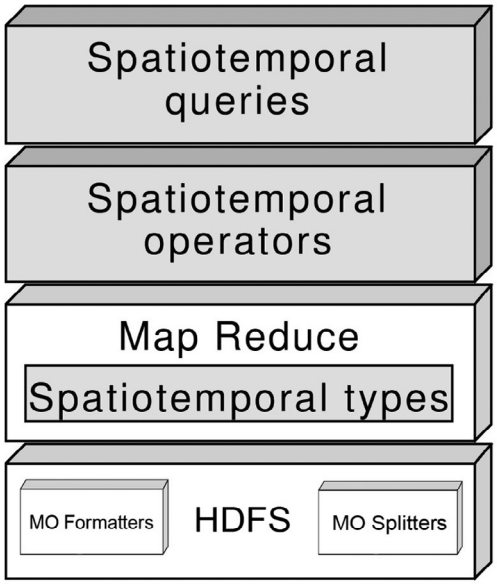

Figure 1. The proposed spatiotemporal extension.

Hadoop core classes, so that they can be referenced as native types in the MapReduce. These types are carefully integrated with HDFS input and output. The operators are implemented as MapReduce jobs. Spatiotemporal queries can then be written as MapReduce jobs invoking both the types and the operators. To demonstrate the algebra, we use it to implement the BerlinMOD queries.

4.1.1. Algebra types

The spatiotemporal types (i.e. the MOD model in the works of Güting et al. (2000)) are implemented as an extension to the MapReduce class hierarchy, along with corresponding HDFS input/output formatters and splitters. We have implemented the mpoint, mreal, mbool, upoint, ureal, and ubool types. The types that start with m (moving) are temporal types, which represent a complete moving object. Types that start with u (unit) denote unit types, where a unit is the building block for a moving object type. For instance, an mpoint consists of a list of upoint.

The type system is defined by a signature. Any signature consists of sorts (called kinds) and operators (called type constructors). It generates a set of terms that describes the types available in the type system. Some data types defined by signature are range (instant) or moving (point). There are many classes of types: Base, Spatial, Temporal, Range, Unit and Moving. The Base type represents the constant values of types: int, bool, string, and real. The Spatial type represents 2-D spatial objects of types: point, points, and region. The Temporal type is used to represent single instants of time. The Range type represents a collection set of intervals or pairs of time instants. It is defined on Base or Temporal types.

There are two important kinds for moving objects (Unit and Mapping). They are important for describing

the moving object. Given the type of α argument in Base or Spatial, it builds a type moving (α). The values are represented as functions from time into the domain α. The Moving type is used to represent a trajectory of moving object. Also, the types moving (int), moving (real), and moving (bool) are defined for storing a temporal evolutions of the Base types.

The types (Range and Moving) are applicable to all the types in the Base. For example: a moving (point) is function from time into point values. The mpoint data type is the basic abstraction of an object moving in the plane or in an area of higher dimensions. It is represented as set of temporal units (Figure 2). This mpoint consists of three unit points. Each unit is represented as (Tstart, Tend, p1, p2). Hence, a value of type mpoint is a continuous function *f*: instant → point. The type mpoint can be viewed as mapping from time into space. The mapping type describes the complete movement of a moving object and represented as a set of units. The moving object movement is decomposed into slices, each of which describes the movement during time interval. A single slice is represented as Unit type. Multiple slices are represented as Mapping type.

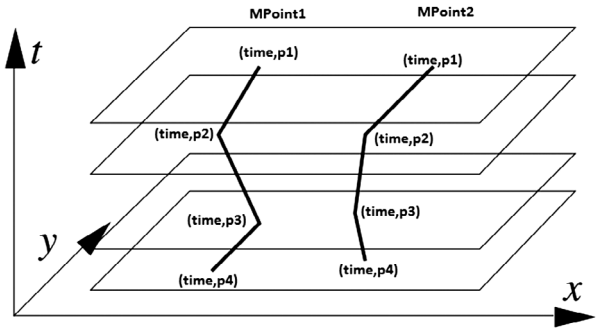

Figure 2. Abstract sliced representation of moving point.

4.1.2. Integrating algebra types into Hadoop

Special input and output formatters are required, because the moving object types have complex formats. These new formatters require well-defined file formats to deal with. A file might contain multiple trajectories, each of which consists of an arbitrary number of units. Each trajectory is identified by a numeric identifier that is repeated with all its units. The end of a trajectory and the beginning of the following one can, hence, be detected by comparing the identifiers. We choose to represent a unit as a line of comma separated values. The parsing rules of this file format are as follows:

```
trajectories-data::= trajectory-data | trajectories-data |<empty>
trajectory-data::= <identifier>, unit-data <new line>
unit-data::= time-interval, unit-value
time-interval::= time-stamp, time-stamp, <lc>, <rc>
time-stamp::= yyyy-MM-dd HH:mm:ss.SSS
unit-value::= upoint-value | ureal-value | uint-value |ubool-value
upoint-value::= point, point
point::= <X>, <Y>
ureal-value::= <real>, <real>
uint-value::= <int>
ubool-value::= <bool>
```

Such a comma-separated format is convenient, because it can be easily exchanged with MOD systems (e.g. SECONDO (Güting et al. 2004) and Hermes (Pelekis et al. 2006)). It is also trivial to transform the GPS exchange files from the GPS tracking devices into this format. Figure 3 is a section from a sample file representing a moving point trajectory.

For the representation of spatiotemporal data, a relevant issue is the representation of their time values. Each unit is represented as TrajID, tStart, tEnd, xStart, yStart, xEnd, yEnd, where <timevalue> is represented as (day month year [hour minute [second [millisecond]]]). For that, we use GregorianCalendar to deal with different forms of date and time. Also, we use it because of trajectory time represented in some cases without one of time items, such as milliseconds or seconds.

The moving object data are distributed across multiple files as we mentioned previously. We need to read and link these files to each other. This is a more complicated task that requires to make an analysis by combining information from two or more different file structures. Therefore, we build some data types and operators for reading these files and building a relation between them. First of all, the user should determine which file contains the moving objects data which may be. Then he should determine the other files, any columns representing the link between them, and the file containing the trajectory of moving objects.

The proposed system uses the following steps to read the input data-set: (1) a map-only job to convert the input to a list of moving objects; (2) a map-only job to link each moving object with its metadata from the other files. Figure 4 shows the processing steps for reading and linking the input data-set files.

```
Moid,Tstart,Tend,Xstart,Ystart,Xend,Yend
1,2007-05-27,2007-05-28 08:36:47.846,12785,1308,12785,1308
1,2007-05-28 08:36:47.846,2007-05-28 08:36:49.846,12785,1308,12793,1310.2
1,2007-05-28 08:36:49.846,2007-05-28 08:36:51.846,12793,1310.2,12808.7,1314.5
1,2007-05-28 08:36:51.846,2007-05-28 08:36:53.846,12808.7,1314.5,12824.8,1318.9
1,2007-05-28 08:36:53.846,2007-05-28 08:36:55.846,12824.8,1318.9,12840.9,1323.3
2,2007-05-27,2007-05-28 08:52:53.673,9716,-55,9716,-55
2,2007-05-28 08:52:53.673,2007-05-28 08:52:55.673,9716,-55,9716.93,-46.7183
2,2007-05-28 08:52:55.673,2007-05-28 08:52:57.673,9716.93,-46.7183,9718.73,-30.5691
2,2007-05-28 08:52:57.673,2007-05-28 08:52:59.673,9718.73,-30.5691,9720.59,-14.0058
```

Figure 3. A section from a sample file representing a moving point trajectory.

We store moving object data as a relation. Each relation is stored in a new Tuple. Each Tuple contains one moving object consists of ID, UnitsTable, and other data that describe the object. The first column is identifier for the moving object. The second column is a data type called UnitsTable which stores all received Tuples as a table. The table consists of multiple tuples and each of them represents the unit. The third column contains the descriptive data that describe the moving object trajectory like license, model, and so on. Figure 5 describes the files stored in HDFS which contain all the moving objects data.

In the first, we choose the method of loading the data. It uses one of these ways: (1) building a new relation on the data-set files or (2) using the relation stored in HDFS. For example, assume you have a new data-set, then it needs to build a relation between these data before executing the first query. If it needs to run the next query, it can use the relation stored in HDFS. When the relation stored in HDFS is used, the query runs faster.

Next, we have implemented new input and output formatters to read and write the moving object data from and to HDFS. New input splitters had to be implemented too. Hadoop input splitters are responsible for dividing the input file into chunks of smaller sizes, and sending them to mappers. The new splitters, in contrast to the Hadoop splitters, can parse the moving object data items that are written over multiple file lines. For the mappers and reducers to consume the moving object data, the type system of (Güting et al. 2000) is implemented as native Hadoop types.

The class diagram (Figure 6) illustrates these extensions, and their integration with the Hadoop classes. A moving object consists of many units, each has a time interval. Several types inherited from the UnitValue class: UnitPoint, UnitBool, UnitReal, etc. The moving object classes in our implementation (in addition to the classes, instant, period, and point) have to implement the Hadoop Interface Writable, so that they can be written to HDFS. Similarly, each of our moving object types requires an InputFormat class that must inherit from the Hadoop class FileInputFormat. Input splits are generated using MovingObjectInputFormat class. Each split is assigned to map task. Input key-value pairs are generated from each split using MovingObjectRecord reader provided by MovingObjectInputFormat class. Every InputFormat class requires a corresponding RecordReader class that inherits from the Hadoop RecordReader.

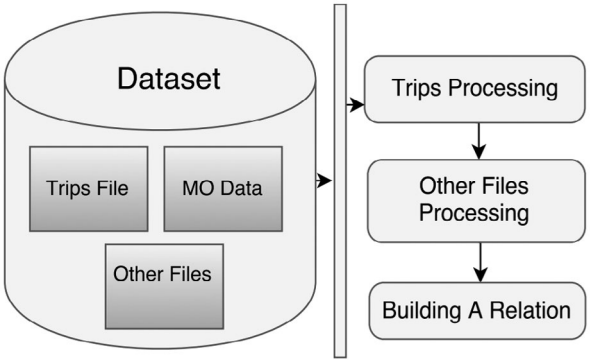

Figure 4. Building a relation.

4.2. Algebra operators

The richness of the data model that we have implemented allows for a numerous set of query and analysis operators. In this section, we describe representative examples of the algebra operators. Fortunately, there already exist two MOD systems based on the same model: SECONDO (Güting et al. 2004), and Hermes (Pelekis et al. 2006). The syntax and semantics of the operators could be easily copied from these two systems. We have already implemented many of their operators within the proposed algebra. This work aims at implementing the whole set of existing operators in these systems. In the scope of this paper, selected operators have been implemented, while the work continues to implement the whole set. In our selection, we fulfill the query requirements of the BerlinMOD queries. When an operator is sent to be executed, the operator would be parsed and represented as relational algebra. MapReduce works better because it does not need schema support, so it is well suited to processing unstructured data. Input formatters do not require Tuples to have the same number of fields.

Every operator is implemented as a MapReduce job. According to their arguments, most of the operators shall be divided into two categories: (1) unary operators

ID	UnitsTable	Other Data
1	((2007-05-28 08:36:47.846,2007-05-28 08:36:49.846,12785,1308,12793,1310.2), (2007-05-28 08:36:49.846,2007-05-28 08:36:51.846,12793,1310.2,12808.7,1314.5), (2007-05-28 08:36:51.846,2007-05-28 08:36:53.846,12808.7,1314.5,12824.8,1318.9))	B-RL 1,passenger,Audi
2	((2007-05-28 07:44:46.025,2007-05-28 07:44:48.025,12717.9,10542.4,12680.2,10532.7), (2007-05-28 07:44:48.025,2007-05-28 07:44:50.025,12680.2,10532.7,12642.5,10523))	B-XW 2,passenger,Audi
8	((2007-05-29 22:34:32.922,2007-05-29 22:34:34.053,17518.3,555.076,17508,571), (2007-05-29 22:34:34.053,2007-05-29 22:34:36.053,17508,571,17522.1,578.481), (2007-05-29 22:34:36.053,2007-05-29 22:34:38.053,17522.1,578.481,17536.8,586.307), (2007-05-29 22:34:38.053,2007-05-29 22:34:40.053,17536.8,586.307,17551.5,594.133), (2007-05-29 22:34:40.053,2007-05-29 22:34:42.053,17551.5,594.133,17566.2,601.958), (2007-05-29 22:34:42.053,2007-05-29 22:34:44.053,17566.2,601.958,17579.9,609.234), (2007-05-29 22:34:44.053,2007-05-29 22:34:46.053,17579.9,609.234,17591.8,615.563))	B-TS 8,bus,Audi

Figure 5. A sample file stored in HDFS which contains all the moving object data.

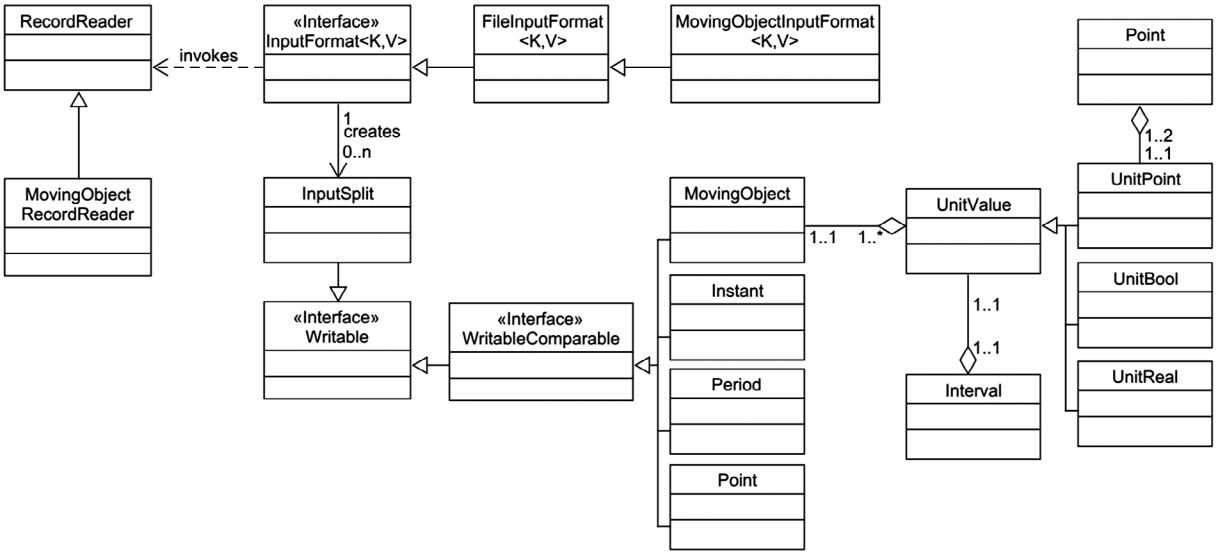

Figure 6. The class diagram of the integration of the MOD type system into Hadoop.

accepting a single set of moving object. These operators require at most one mapper without a reducer; (2) binary operators accepting at least two sets moving objects. These operators may require up to two mappers and one reducer in their implementation, according to the input size.

We illustrate one example operator in the following. Table 1 lists the whole set of operators we have implemented so far.

The passes operator is a predicate that checks whether a moving point trajectory ever intersects a given point position. In a MOD system, it would have the following signature: mpoint × point → bool.

A major differentiation should be made here between a query system (such as MOD systems) and an analytics system (such as Hadoop). In a query system, the data are streamed one by one into the chain of query operators. Operators are thus designed to accept individual elements in their input. Hadoop, on the other hand, does a bulk split and distributes for the whole input set before the MapReduce functions are invoked. The operator needs accordingly to accept the two sets of input in whole (i.e. the set of trajectories and the set of points), and to return the whole set of pairs (trajectory, point) that fulfill the predicate. The predicate signature

Table 1. Operators description.

Operator	Type	Description
Atperiods	Binary	Restricts the definition time of a given moving object to a given periods object, yielding a moving object
Atinstant	Binary	Restricts the definition time of a given moving object to a given instant object, yielding a pair of instant and value
Passes	Binary	A predicate that yields true if the given mpoint object ever intersects the given spatial object
Present	Binary	A predicate that yields true if the given mpoint object is defined at the given time instant
Deftime	Unary	Yields the definition time of the given moving object as periods
Initial/final	Unary	Yields the pair of instant and value corresponding to the initial/final instant of the given moving object
Length	Unary	Computes the total distance traveled by the mpoint object

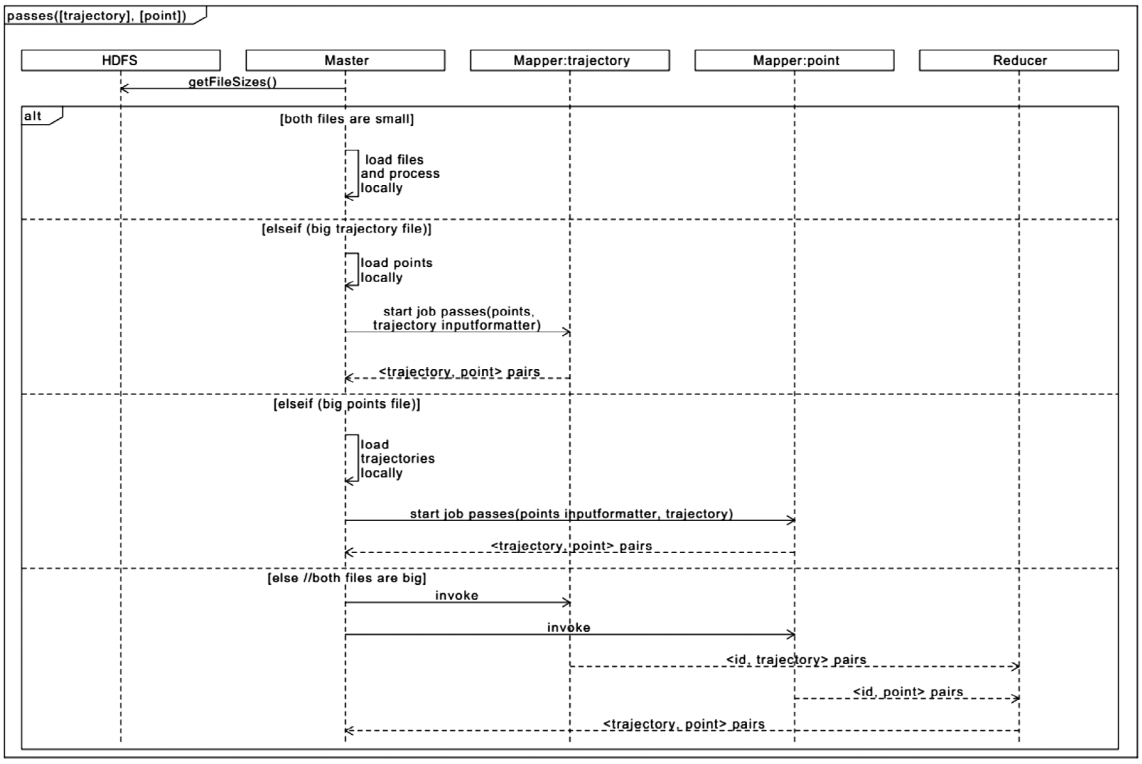

Figure 7. The passes operator.

in Hadoop would conceptually be: set(mpoint) × set(-point) → set(<mpoint, point>).

Yet, we include more details in the result in our implementation, such as the intersection time(s). Note: a single trajectory might pass the same point multiple times. Moreover, the result format and contents can be configured by the user to fit the requirements of the analysis task. The user might wish to obtain only the identifiers in the result, rather than the actual trajectory and point data. Such multiple choices are included in the implementation of the operator and can be set through the Hadoop job configuration.

One more implementation aspect is to make the operator able to adapt with the input size. The passes operator accepts two sets of input, each of which can be of an arbitrary size. Small sized data are better loaded and processed locally on one node, to avoid the latency of the splitting and the network transmission. Therefore, we have included such logic in the implementation of all the algebra operators. The passes operator, as illustrated in Figure 7, starts by checking the input file sizes in the HDFS. If the two files holding the trajectory data and the point data are small, the master node performs the whole job of loading the files and computing the result. If only one of the files is small, the master node loads and adds it to the mapper configuration. The mapper would receive the data of the smaller file in its configurations, and would consume the larger file via its input formatter. Therefore, we have two mappers for the passes operator: a mapper for the case when the trajectories file is big, and a mapper for the case when the points file is big. It should be noted that, in these two cases, a reducer is not required. The mapper can do the job, as it already has all the required input. Finally, if the two files are big, two mappers are required. This is necessary, because one mapper can only receive one input formatter corresponding to one big file. The task of the mappers is simple in this case, as they have to parse the file and emit the trajectories and points, respectively. A reducer is therefore required to join the two sets and implement the logic of the passes operator.

4.3. BerlinMOD queries

This section applies the proposed algebra to implement the widely accepted BerlinMOD benchmark queries (Düntgen, Behr, and Güting 2009). This benchmark was originally proposed to compare MOD systems. Although our algebra falls into the analytics rather than the MOD type of systems, we argue that by implementing the benchmark queries, and prove the utility of the algebra in terms of coverage and generousness. Each benchmark query is implemented by running and combining the results of multiple operators (i.e. multiple MapReduce jobs). The developer should make an execution plan, which requires knowledge of the input. A declarative language and an optimizer are in the scope of our future work. Here, we start by illustrating Query 17 of the benchmark:

Query 17: Which points from QueryPoints have been visited by a maximum number of different vehicles?

In SQL, this query would be expressed as follows:

```
Step1:
OutputCollector(key,value) countOutput= SELECT p.Pos as Pos, count(c.License)
as Hits FROM QueryPoints AS p, Trips AS t, dataMCar AS c
WHERE t.Trip passes p.Pos AND t.VehicleID = c.VehicleID
group by p.Pos,c.Moid

Step2:
output=select Pos from countOutput
where Hits=(select max(Hits) from countOutput)
```

This query needs to consume the three relations as follows:

- QueryPoints[Id: int, Pos: point]: a set of query points, randomly chosen from the nodes that constitute the trips. It is a small file, of size 1.4 KB in scale factor 10.
- dataMTrip[Moid: int, TripID: int, Trip: mpoint]: contains the mpoint objects representing the actual movement data. This is a big file, of size 51.5 GB in scale factor 10.
- dataMCar[Moid: int, licence: string, Type: string, Model: string]: contains the vehicle identifiers and meta data. It is a small file, of size 200.5 KB in scale factor 10.

The last two files are correlative each other. The dataMTrip file contains a trajectories and any descriptive data existing in another file called dataMCar. The challenge is how to check each moving point inside dataMTrip file, and then, to get the other attributes in the relation (such as licence), if one of them passes any point in QueryPoints file. Here, comes the benefit of modifying the HDFS splitters for distributing the moving objects in a way to all slave nodes. Also, building a relation between the input files helps in getting the other attributes quickly.

We choose to implement the plan as illustrated in Figure 8. The query starts by invoking the passes operator for the two files: QueryPoints and Trips. The result would be the pairs from the two files that fulfill the passes predicate. It should be noted that the duplicate pairs are possible, since a single trip might pass the same point multiple times. This result is of a small size, proportional to the size of QueryPoints. It is hence processed locally in the master node to produce the query result.

The master node joins the resulting pairs with the dataMCar file, to add the column licencelicense. Again, this step might produce duplicates, because one licencelicense can have multiple trips. Duplicates are removed, so that we obtain unique combinations of (licence, pos). Finally, this list is grouped by the pos, a count per position is calculated, and the pair with the highest count is returned.

Next, we will illustrate Query 16. It involves calling multiple operator instances on parallel, and MapReduce jobs.

Query 16 List the pairs of licences for vehicles. The first is from QueryLicences1, and the second is from QueryLicences2, where the corresponding vehicles are both presented within a region from QueryRegions1 during a period from QueryPeriod1, but do not meet each other there and then.

The query plan is illustrated in Figure 9. The query starts by joining the Trip file with the QueryLicences1 and with the QueryLicences2 in parallel, to construct the two sets of input Trip1 and Trip2. Each of the two sets is then used as input to the atperiods operator then to the intersection operator from our algebra. The atperiods operator restricts the definition time of the input trips to the periods given in QueryPeriods. The intersection operator spatially restricts the trips to the regions given in QueryRegions. The result so far consists of two sets of trips that are restricted/clipped in time and space to the given periods and regions. It remains to compute the spatiotemporal intersection of these two sets, and yield the licences of the pairs that do not intersect. This is done using two mappers (one for each set), and a reducer. The reducer gets a sorted list of triples (Region, Period, Trip). It joins the pairs of trips that occur in the same region and period, and filter out the intersected. To compute the

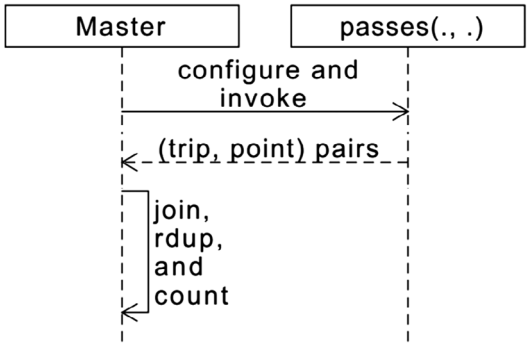

Figure 8. A sequence diagram of Query 17.

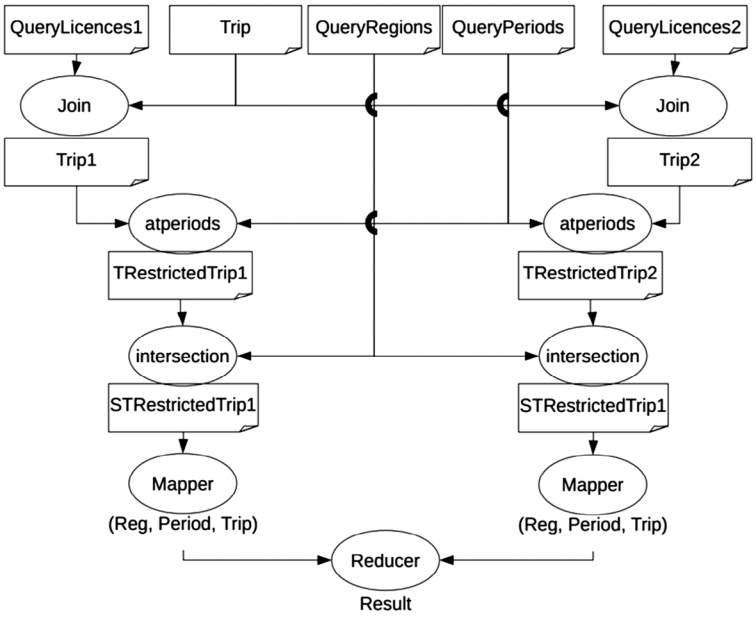

Figure 9. An explanation of Query 16.

intersection, it uses the corresponding member function provided in our implementation of the mpoint type.

5. Performance evaluation

In this section, we illustrate an experiment that compares the performance of the proposed Hadoop extension with the SECONDO MOD system. The objective of this comparison is to give the reader an idea about the performance of the proposed algebra, by having SECONDO as a base line. It should be noted, though, that the two systems are intended for different scenarios: our Hadoop extension is in place when the amount of data is larger than what can be processed on a single SECONDO node. It is also in place when adding spatiotemporal processing capabilities for existing or planned Hadoop application.

The setting of this experiment includes a SECONDO instance on a single machine of quad core i7–3770 CPU 3.40 GHz, 32 GB RAM, running Ubuntu v12.04, and SECONDO v3.4. This node is used to evaluate the run times on SECONDO. The cluster that is used for evaluating the run times on Hadoop consists of 3 nodes. Each node is an Intel Xeon 8 core CPU (E5–2650L 1.80 GHz), 16 GB RAM, running Ubuntu v12.04 and the Hadoop v1.1.2. One of the three nodes has only 8 GB RAM. The performance is evaluated by running the BerlinMOD queries on the two setups. The queries used in this experiment are of multiple classes: non-spatiotemporal, temporal, and spatiotemporal (Düntgen, Behr, and Güting 2009) (Table 2).

For the input, we use the BerlinMOD data generator to prepare two datasets as described in Table 3.

On the SECONDO machine, we note that the response time (i.e. elapsed time between issuing a query and the beginning of its response), and the CPU time (i.e. excluding from the response time, the time where the CPU was busy at processing other programs). These measures are obtained from a SECONDO system relation that stores run statistics for every query. On the cluster, we note two time values: one is the time for HDFS to transfer all the blocks of the input files to every work node, and the other is the average time taken on each node after receiving the input files until the whole query return. In some cases, when the query consists of multiple jobs, such as Query 16, we additionally note the time for every job.

The results (Table 4) show that for both scale factors 2 and 10, SECONDO performs faster than the proposed Hadoop extension, except for Queries 4 and 9. The reasons for these results can be summarized as follows:

- Most of the time spent on Hadoop in this experiment goes for reading the input. As indicated in Table 4, uploading the data from HDFS to the work nodes requires 126 s for scale factor 2, as the trip file is 10.4 GB.
- At the worker nodes, again, a lot of time goes to reading the moving object data from the input files.
- The queries on SECONDO use indexes (b-tree and r-tree) that are generated during the data generation. This leads to a significant speed-up.
- Queries 4 and 9 perform slower on SECONDO, because they involve aggregations, which require sorting on the limited memory space of the machine. On the other hand, grouping is one of the common MapReduce patterns, and can be processed very fast.

Additionally, the speed-up evaluation in Query 16 is also made between different data-set scale factors to show that Hadoop extension is outperform SECONDO, when the data become huger (Figure 10).

Table 2. Experimental queries.

Query	Type	Description
Query 1	Non-ST	What are the models of the vehicles with licence plate numbers from QueryLicences?
Query 2	Non-ST	How many vehicles are passenger cars?
Query 3	ST	Where have the vehicles with licences from QueryLicences1 been at each of the instants from QueryInstants1?
Query 4	ST	Which licence plate numbers belong to vehicles that have passed the points from QueryPoints?
Query 8	ST	What are the overall travelled distances of the vehicles with licence plate numbers from QueryLicences1 during the periods from QueryPeriods1?
Query 9	ST	What is the longest distance traveled by a vehicle during each of the periods from QueryPeriods?
Query 16	ST	List the pairs of licences for vehicles, the first is from QueryLicences1, the second is 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.
Query 17	ST	Which points from QueryPoints have been visited by a maximum number of different vehicles?

Table 3. Experimental datasets.

Berlin-MOD scale factor	Vehicles	Trips	Units	Trips file size (GB)	DB size (GB)
2	2828	599,584	115,447,671	10.4	160.6
10	6325	2,957,137	567,005,829	51.5	548.1

Table 4. List of run times (in unit of s).

	SECONDO		Hadoop extension	
BerlinMOD	Response	CPU	Response	Data uploading
Scale 2				
Query 1	0.17	0.03	12.73	126
Query 2	0.03	0.03	9.75	126
Query 3	0.65	0.10	387.25	126
Query 4	158.50	68.00	53.50	126
Query 8	0.61	0.53	1054.65	126
Query 9	4157.50	3833.7	753.40	126
Query 16	164.30	63.00	86.41	126
Query 17	59.00	56.90	395.80	126
Scale 10				
Query 1	1.72	1.59	14.35	423
Query 2	0.05	0.04	9.75	423
Query 3	75.93	74.83	4153.61	423
Query 4	1991.86	479.14	267.19	423
Query 8	1.91	0.93	3680.50	423
Query 9	12,430.13	11,590.30	6724.00	423
Query 16	2571.60	578.68	394.27	423
Query 17	2172.58	395.30	3914.40	423

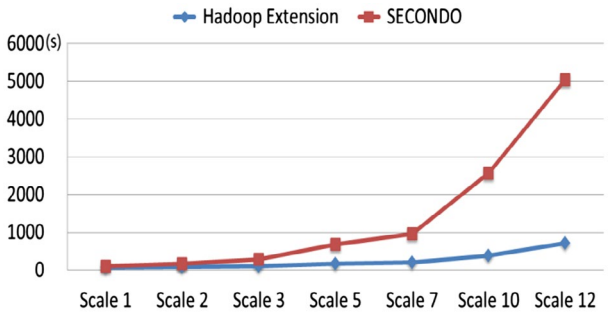

Figure 10. Query16 run times on different scale factors of BerlinMod data-set.

The purpose of comparing SECONDO to Hadoop is to give the reader an idea on the performance. Here, SECONDO is a DBMS targeted to querying, while Hadoop is targeted to analytics. In application, our algebra will be used in longer analytics scenarios, with a longer pipeline of spatiotemporal operators processing the data. Only then, the HDFS overhead will be overtaken by saving the execution time. We can see this in some queries, such as Queries 4, 9, and 16. One more reason that SECONDO can be faster than Hadoop in certain cases, is the ability of SECONDO of using indexing, and only processing potential candidates.

6. Conclusions

This paper has described a spatiotemporal algebra extension to Hadoop. The algebra adds spatiotemporal types to the Hadoop core, so that MapReduce jobs could use them as native Hadoop types. Spatiotemporal operators are built on top of these types, as MapReduce jobs. The utility of the proposed algebra has been demonstrated using the widely accepted BerlinMOD benchmark. Essentially, queries can be expressed as MapReduce jobs that invoke operators, which are MapReduce jobs themselves. As a result, this algebra gives space for scalable Hadoop applications that require spatiotemporal processing, this is not available so far, up to our knowledge.

This work needs to be further developed in many directions. More operators can be added to increase the expressiveness. A query language will be taken into consideration, so that end users can perform heavy spatiotemporal processing without writing MapReduce programs. Yet, the most important development required in this phase is to add local and global indexes to reduce the amount of data being transferred from HDFS to worker nodes, and to reduce the IO access at worker nodes.

Notes on contributors

Mohamed S. Bakli is a teaching assistant in the Department of Information Systems, Faculty of Computers and Information, Assiut University, Egypt. His research interests include big data, databases, moving object databases, query processing, spatial data, and spatiotemporal data.

Mahmoud A. Sakr is an assistant professor in the Department of Information Systems, Faculty of Computers and Information, Ain Shams University, Egypt. Currently he is a postdoctoral researcher in the Université libre de Bruxelles, Belgium. His research interests include moving object databases, big data, spatial and spatiotemporal databases.

Taysir Hassan A. Soliman is a professor in the Department of Information Systems, Faculty of Computers and Information, Assiut University, Egypt. Currently he is the Vice Dean for Graduate Studies and Research, Faculty of Computers and Information, Assiut University. His research interests include bioinformatics, data mining, big data, recommender systems, spatial and spatiotemporal databases.

References

Aji, A., F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. 2013. "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce." *Proceedings VLDB Endowment* 6 (11): 1009–1020. doi:10.14778/2536222.2536227.

Apache Software Foundation. 2010. "Hadoop." https://hadoop.apache.org.

Auchincloss, A. H., S. Y. Gebreab, C. Mair, and A. V. Diez Roux. 2012. "A Review of Spatial Methods in Epidemiology, 2000–2010." *Annual Review of Public Health* 33: 107–122. doi:10.1146/annurev-publhealth-031811-124655.

Cao, G., S. Wang, M. Hwang, A. Padmanabhan, Z. Zhang, and K. Soltani. 2015. "A Scalable Framework for Spatiotemporal Analysis of Location-based Social Media Data." *Computers Environment & Urban Systems* 51: 70–82. doi:10.1016/j.compenvurbsys.2015.01.002.

Chen, Q., L. Wang, and Z. Shang. 2008. "MRGIS: A MapReduce-enabled High Performance Workflow System for GIS." The Fourth IEEE International Conference on Escience, Indianapolis, IN, December 10–12. doi: 10.1109/escience.2008.169.

Chung, T. Y., K. T. Chuang, C. M. Hsu, and W. S. Ku. 2015. "Spatiotemporal Crowdsourcing Behavior: Analysis on OpenStreetMap." The Conference on Technologies and Applications of Artificial Intelligence, Tainan, Taiwan, November 20–22. doi: 10.1109 /taai.2015.7407072.

Düntgen, C., T. Behr, and R. H. Güting. 2009. "BerlinMOD: A Benchmark for Moving Object Databases." *VLDB Journal* 18 (6): 1335–1368. doi:10.1007/s00778-009-0142-5.

Eldawy, A. 2014. "SpatialHadoop: Towards Flexible and Scalable Spatial Processing Using Mapreduce." The 2014 SIGMOD PhD Symposium, Snowbird, Utah, June 22–27. doi: 10.1145/2602622.2602625.

Eldawy, A., and M. F. Mokbel. 2014. "Pigeon: A Spatial MapReduce Language." The IEEE International Conference on Data Engineering, Chicago, IL, USA, March 31–April 4. doi: 10.1109/ICDE.2014.6816751.

Eldawy, A., and M. F. Mokbel. 2015. "SpatialHadoop: A MapReduce Framework for Spatial Data." The IEEE International Conference on Data Engineering, Seoul, South Korea, April 13–17. doi: 10.1109/ICDE.2015.7113382.

Ferreira, N., and J. Poco, H. T. Vo, J. Freire, and C. T. Silva. 2013. "Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips." *IEEE Transactions on Visualization & Computer Graphics* 19 (12): 2149–2158. doi:10.1109/TVCG.2013.226.

Forlizzi, L., R. H. Güting, E. Nardelli, and M. Schneider. 2000. "A Data Model and Data Structures for Moving Objects Databases." The ACM SIGMOD International Conference on Management of Data, Dallas, TX, May 15–18. doi: 10.1145/342009.335426.

Fox, A., C. Eichelberger, J. Hughes, and S. Lyon. 2013. "Spatio-temporal Indexing in Non-relational Distributed Databases." The IEEE International Conference on Big Data, Santa Clare, CA, October 6–9. doi: 10.1109/BigData.2013.6691586.

González, D., J. Pérez, V. Milanés, and F. Nashashibi. 2016. "A Review of Motion Planning Techniques for Automated Vehicles." *IEEE Transactions on Intelligent Transportation Systems* 17 (4): 1135–1145. doi:10.1109/TITS.2015.2498841.

Grumbach, S., P. Rigaux, M. Scholl, and L. Segoufin. 1997. "DEDALE, A Spatial Constraint Database." The 6th International Workshop on Database Programming Languages, Colorado, August 18–20.

Güting, R. H., T. Behr, V. Almeida, Z. Ding, F. Hoffmann, M. Spiekermann, and F. Informatik. 2004. "SECONDO: An Extensible DBMS Architecture and Prototype." Technical Report. http://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/forschung/berichte/bericht_313.pdf.

Güting, R. H., M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. 2000. "A Foundation for Representing and Querying Moving Objects." *ACM Transactions on Database Systems* 25 (1): 1–42. doi:10.1145/352958.352963.

Güting, R. H., and J. Lu. 2015. "Parallel SECONDO: Scalable Query Processing in the Cloud for Non-standard Applications." *Sigspatial Special* 6 (2): 3–10. doi:10.1145/2744700.2744701.

HBase. 2012. "Apache HBase." http://hbase.apache.org/.

Lu, J., and R. H. Güting. 2012. "Parallel SECONDO: Boosting Database Engines with Hadoop." The 18th IEEE International Conference on Parallel and Distributed Systems, Singapore, December 17–19. doi: 10.1109/ICPADS.2012.119.

Ma, Q., B. Yang, W. Qian, and A. Zhou. 2009. "Query Processing of Massive Trajectory Data based on MapReduce." The First International Workshop on Cloud Data Management, ACM, Hong Kong, China, November 2. doi: 10.1145/1651263.1651266.

Markram, H. 2006. "The Blue Brain Project". The 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, November 11–17. doi: 10.1145/1188455.1188511.

Natalija, S., and S. Dragan. 2015. "A Hybrid MPI+OpenMP Application for Processing Big Trajectory Data." *Studies in Informatics and Control* 24 (2): 229–236.

Nidzwetzki, J. K., and R. H. Güting. 2015. "Distributed SECONDO: A Highly Available and Scalable System for Spatial Data Processing." The Advances in Spatial and Temporal Databases, 14th International Symposium, Hong Kong, China, August 26–28.

Nishimura, S., S. Das, D. Agrawal, and A. E. Abbadi. 2013. "\(\mathcal{MD}\)-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services." *Distributed & Parallel Databases* 31 (2): 289–319.

Olson, M. A., K. Bostic, and M. Seltzer. 1999. "Berkeley DB." The Annual Conference on USENIX Annual Technical Conference, Monterey, CA, June 06–11.

Park, J. W., W. Yong, L. Chang, H. Yun, H. K. Park, S. I. Chang, and I. Pyoung. 2010. "Cloud Computing for Online Visualization of GIS Applications in Ubiquitous City." The First International Conference on Cloud Computing, GRIDs, and Virtualization, Lisbon, Portugal, November 21–26.

Pelekis, N., Y. Theodoridis, S. Vosinakis, and T. Panayiotopoulos. 2006. "Hermes – A Framework for Location-Based Data Management." The 10th International Conference on Advances in Database Technology, Munich, Germany, March 26–31. doi: 10.1007/11687238_75.

Pickle, L. W., M. Szczur, D. R. Lewis, and D. G. Stinchcomb. 2006. "The Crossroads of GIS and Health Information: A Workshop on Developing a Research Agenda to Improve Cancer Control." *International Journal of Health Geographics* 5 (1): 1–13. doi:10.1186/1476-072X-5-51.

Raza, A. 2012. "Working with Spatio-temporal Data Type." The XXII International Society for Photogrammetry and Remote Sensing (ISPRS) Congress, Melbourne, Australia, August 25–September 01.

Song, W. W., B. X. Jin, S. H. Li, X. Y. Wei, D. Li, and F. Hu. 2015. "Building Spatiotemporal Cloud Platform for Supporting GIS Application." *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* II-4-W2: 55–62. doi: 10.5194/isprsannals-II-4-W2-55-2015.

SpatialHadoop. 2013. "SpatialHadoop, A MapReduce Framework for Spatial Data." http://spatialhadoop.cs.umn.edu.

Yang, B., Q. Ma, W. Qian, and A. Zhou. 2009. "TRUSTER: Trajectory Data Processing on ClUSTERs." Paper presented at the 14th International Conference on Database Systems for Advanced Applications, Brisbane, Australia, April 21–23.