MapReducing GEPETO or Towards Conducting a Privacy Analysis on Millions of Mobility Traces

Sébastien Gambs
Université de Rennes 1
INRIA/IRISA
Rennes, France
sgambs@irisa.fr

Marc-Olivier Killijian
CNRS; LAAS
Université de Toulouse;
UPS; INSA; INP; ISAE; LAAS
Toulouse, France
marco.killijian@laas.fr

Izabela Moise
INRIA
Rennes, France
izabela.moise@inria.fr

Miguel Núñez del Prado Cortez
CNRS; LAAS
Université de Toulouse;
UPS; INSA; INP; ISAE; LAAS
Toulouse, France
mnunezde@laas.fr

***Abstract*—GEPETO (for GEoPrivacy-Enhancing TOolkit) is a flexible software that can be used to visualize, sanitize, perform inference attacks and measure the utility of a particular geolocated dataset. The main objective of GEPETO is to enable a data curator (e.g., a company, a governmental agency or a data protection authority) to design, tune, experiment and evaluate various sanitization algorithms and inference attacks as well as visualizing the following results and evaluating the resulting trade-off between privacy and utility. In this paper, we propose to adopt the MapReduce paradigm in order to be able to perform a privacy analysis on large scale geolocated datasets composed of millions of mobility traces. More precisely, we design and implement a complete MapReduce-based approach to GEPETO. Most of the algorithms used to conduct an inference attack (such as sampling, k-means and DJ-Cluster) represent good candidates to be abstracted in the MapReduce formalism. These algorithms have been implemented with Hadoop and evaluated on a real dataset. Preliminary results show that the MapReduced versions of the algorithms can efficiently handle millions of mobility traces.**

***Index Terms*—Location Privacy; Big Data Mining; MapReduce; Hadoop; Data-Intensive Applications.**

I. Introduction

The advent of ubiquitous devices and the growing development of location-based services have lead to a large scale collection of the mobility data of individuals on a daily basis. For instance, the location of an individual can be (1) deduced from the IP address of his computer, (2) collected by applications running on a smartphone providing information tailored to the current location or used for collaborative tasks such as traffic monitoring[1], (3) attached in the form of a geotag to a picture he has taken without him noticing or (4) given explicitly by checking-in to a geo-social network such as Foursquare[2]. Among all the *Personally Identifiable Information* (PII), learning the location of an individual is one of the greatest threats against privacy. In particular, an inference attack can use mobility data (together with some auxiliary information) to deduce the points of interests characterizing his mobility, to predict his past, current and future locations or even to identify his social network. Therefore, we believe that the design of tools that can assess the privacy risks incurred by the collection or dissemination of location data is of paramount importance.

GEPETO (for *GEoPrivacy-Enhancing TOolkit*) [9] is a flexible software that can be used to visualize, sanitize, perform inference attacks and measure the utility of a particular geolocated dataset. The main objective of GEPETO is to enable a data curator (*e.g.*, a company, a governmental agency or a data protection authority) to design, tune, experiment and evaluate various sanitization algorithms and inference attacks as well as to visualize the following results and evaluate the trade-off between privacy and utility. The aim of this paper is to describe a new version of GEPETO based on the MapReduce paradigm. This MapReduced version of GEPETO is highly distributed and can efficiently handle up to millions of mobility traces.

The outline of this paper is the following. First, we provide a brief overview on the location data and geoprivacy, respectively in Sections II and Section III, before introducing the MapReduce programming model and its Hadoop implementation in Section IV. Afterwards, in Section V, we describe in generic terms how it is possible to implement GEPETO based on the MapReduce paradigm before presenting the testbed and the dataset used during the experiments. In the following sections, we give concrete examples of algorithms from GEPETO that we have MapReduced, namely sampling, k-means and DJ-Cluster. More precisely, the adaption of these algorithms to the MapReduce paradigm are detailed respectively in Section VI, Section VII and Section VIII. For each algorithm, we first describe its underlying concepts, then we identify the *map* and the *reduce* phases for each of them and detail the tasks of the mapper and the reducer in their Hadoop implementation and finally, we report on some experimental results. Finally, we conclude in Section IX with some possible extensions.

II. Location Data

Nowadays, the rapid growth and development of location-based services and geolocated devices have multiplied the potential sources of location data. The location data generated by these diverse applications varies in its exact form and content but it also shares some common characteristics. Within the context of GEPETO, we focus mainly on location data

[1] www.waze.com
[2] www.foursquare.com

represented in the form of *mobility traces*. A mobility trace is usually characterized by:

- An *identifier*, which can be the real identifier of the device (*e.g.*, "Alice's phone"), a pseudonym or even the value "unknown" when full anonymity is required. A pseudonym is generally used as a first protection mechanism to hide the identity of the user while still being able to link different mobility traces that have been generated by him.
- A *spatial coordinate*, which can be a GPS position (*e.g.*, latitude and longitude coordinates), a spatial area (*e.g.*, the name of a neighborhood in a particular city) or even a semantic label (*e.g.*, "home" or "work").
- A *timestamp*, which can be the exact date and time or just an interval (*e.g.*, between 2PM and 6PM).
- Additional information such as the speed and direction for a vehicle, the presence of other geolocated devices or individuals in the direct vicinity or even the accuracy of the estimated reported position. For instance, some geolocated systems are able to estimate the precision of their estimated position as a function of the number of GPS satellites they can detect.

A *trail of traces* is a collection of mobility traces that correspond to the movements of an individual over some period of time. A geolocated dataset D is generally constituted by a set of trails of traces from different individuals. Technically, this data may have been collected by recording locally the movements of each geolocated device for a certain period of time, or centrally by a server that can track the location of these devices in real-time. The Crawdad project[3] is an example of a public repository giving access to geolocated datasets, which can be used for research purpose. GeoLife is another example of a geolocated dataset [30], which contains the mobility traces of researchers of Microsoft Asia on a period of several years.

III. Geoprivacy

One of the main challenge for geoprivacy is to balance the benefit for an individual of using a location-based service with the privacy risks he incurs by doing so. For example, if Alice's car is equipped with a GPS and she accepts to participate in the real-time computation of the traffic map, this corresponds to a task that is mutually beneficial to all the drivers but at the same time Alice wants to have some privacy guarantees that her individual locations will be protected and not broadly disclosed. In practice, we clearly advocate to follow the "privacy by design" paradigm that explicitly takes into account the privacy issues in the design process of a location-based service, rather than simply deploying it and wait for the possible disastrous consequences.

An *inference attack* is an algorithm that takes as input some geolocated dataset D, possibly with some auxiliary information aux, and produces as output some additional knowledge [19]. For example, an inference attack may consist in identifying the house or the place of work of an individual. The auxiliary information reflects any *a priori* knowledge that the adversary might have gathered (*e.g.*, through previous attacks or by accessing some public data sources) that may help him in conducting an inference attack. An adversary attacking some geolocated data may have various objectives ranging from identifying the home of the target to reconstructing his social network, or even obtaining knowledge of his favorite jogging tracks. More precisely, the objective of an inference attack may be to:

- *Identify important places*, called *Points Of Interests* (POIs), characterizing the interests of an individual [18]. A POI may be for instance, the home or place of work of an individual or locations such as a sport center, theater or the headquarters of a political party. Revealing the POIs of a particular individual is likely to cause a privacy breach as this data may be used to infer sensitive information such as hobbies, religious beliefs, political preferences or even potential diseases. For instance, if an individual has been visiting a medical center specialized in a specific type of illness, then it can be deduced that he has a non-negligible probability of having this disease.
- *Predict the movement patterns of an individual* such as his past, present and future locations [19]. From the movement patterns, it is possible to deduce other PII such as the mode of transport, the age or even the lifestyle[4]. According to some recent work [25], [13], our movements are easily predictable by nature. For instance in [25], the authors have explored the limits of predictability in human mobility by analyzing mobility patterns of 50000 individuals within an anonymized geolocated dataset obtained from a mobile phone company that has more than 10 million users. By measuring the entropy of individuals' trajectories, these authors have found a 93% potential predictability in user mobility.
- *Learn the semantics of the mobility behavior of an individual* from the knowledge of his POIs and movement patterns. For instance, some mobility models such as *semantic trajectories* [3], [26] do not only represent the evolution of the movements of an individual over time but they also attach a semantic label to the visited places. From this semantic information, the adversary can derive a clearer understanding about the interests of an individual as well as his mobility behavior than simply from his movement patterns. For instance, the adversary might be able to infer that, on a typical weekday, the considered individual generally leaves his home (POI 1) to bring his kid to school (POI 2) before going to work (POI 3), which is a more deep knowledge than simply knowing the movement pattern "POI 1 $\Rightarrow$ POI 2 $\Rightarrow$ POI 3".
- *Link the records of the same individual*, which can be contained in different geolocated datasets or in the same dataset, either anonymized or under different

[3]http://crawdad.cs.dartmouth.edu/

[4]See for instance http://www.sensenetworks.com/.

pseudonyms. This is the geoprivate equivalent of the *statistical disclosure risk* where privacy is measured according to the risk of linking the record of the same individual in two different databases (*e.g.*, establishing that a particular individual in the voting register is also a specific patient of a hospital [27]). In a geolocated context, the purpose of a linking attack might be to associate the movements of Alice's car (contained for instance in dataset A) with the tracking of her cell phone locations (recorded in another dataset B). As the POIs of an individual and his movement patterns constitute a form of fingerprinting, simply anonymizing or pseudonymizing the geolocated data is clearly not a sufficient form of privacy protection against linking or deanonymization attacks. Indeed, a combination of locations can play the role of a *quasi-identifier* if they characterize almost uniquely an individual in the same way as the combination of his first name and last name. For example, Golle and Partridge [12] have shown that even the pair home-work becomes almost unique per individual, and thus acts as a quasi-identifier, if the granularity is not coarse enough (*e.g.*, if the street is revealed instead of the neighborhood).

- *Discover social relations* between individuals by considering for instance that two individuals that are in contact during a non-negligible amount of time share some kind of social link (of course, false positive may happen) [16]. This information can also be derived from mobility traces by observing that certain individuals are in the vicinity of each other on a frequent basis.

IV. MapReduce and Hadoop

This section briefly discusses the main principles of the MapReduce paradigm and its Hadoop implementation.

A. MapReduce

The MapReduce programming model was introduced in 2004 [8] by Google as a possible solution to process extremely large datasets (up to Terabytes in size). The MapReduce approach is able to efficiently handle such large datasets by taking advantage of data independence (*i.e.*, it assumes that the dataset can be split into chunks that can be processed independently of one another). In particular, MapReduce integrates automatic mechanisms for distributing and parallelizing data computation on a large scale. A developer designing a MapReduce-based application is left with the task of specifying two primary functions: *Map* and *Reduce*. The *map* phase applies a filter on the input data outputting only relevant information while the *reduce* step aggregates the map outputs into a final one.

When launching a MapReduce computation, the input data is first partitioned into blocks of equal size called *chunks*. These chunks are stored in a distributed file system deployed across the participating machines. The scheduler launches as many map tasks as possible, each chunk being processed by a different map task. In the MapReduce approach, data is represented as key-value pairs. Each mapper applies a filter on its input data, selecting only the records satisfying a predefined condition. The mapper processes its associated chunk and outputs intermediate key-value pairs for a reduce task. This reducer collects and aggregates the data produced by the mapping phase. More precisely, all values that have the same key are presented to a single reducer, through a data shuffle and sorting step, in which the data coming from several mappers is transferred to a particular reducer responsible for dealing with it. This phase represents the only communication step in MapReduce. The reduce phase is responsible for applying the user computation on each intermediate key and its corresponding set of values. The result of this phase is also the final output of MapReduce process. However, it is possible that some applications require to pipeline several MapReduce computations.

All the steps of the computation, namely the data partitioning, the scheduling of mapper/reducer tasks and the transfers of records across nodes are rendered transparent to the users by MapReduce. In particular, the system assigns tasks to the nodes in the system based on the locations of the data chunks. In practice, the reducers are spread across the same nodes as the mappers.

B. Hadoop

In 2006, Yahoo! proposed Hadoop as an open-source implementation of the MapReduce programming model [1], [2]. Since then, Hadoop has quickly become the reference implementation of the MapReduce paradigm and also by far the most popular framework for performing data intensive computation (*i.e.*, also referred to as "Big Data"). Hadoop is currently used as a support for web indexing and smart advertising by many companies such as Facebook [7], Twitter, Youtube and Flickr. In the domain of scientific applications, Hadoop is used in the particular fields of Image Processing and Machine Learning [11], [22], as well as Astronomy [28].

Hadoop is designed to efficiently process large datasets by connecting together many commodity computers and making them work in parallel, in a so-called *Hadoop cluster*. In a Hadoop cluster, the nodes in charge of storing the chunks are called *datanodes* while a centralized *namenode* is responsible for keeping the file metadata and the location of the chunks. The Hadoop Distributed File System (HDFS) [23] handles possible failures of nodes through chunk-level replication (by default 3 replicas are created). When distributing the replicas to the datanodes, the HDFS employs a rack-aware policy: the first copy is always written locally, the second copy is stored on a datanode in the same rack as the first replica, and the third copy is shipped to a datanode belonging to a different rack chosen at random. The namenode decides and maintains the list of datanodes storing the replicas of each chunk.

The Hadoop architecture follows the master-slave design; a single master, called the *jobtracker*, is in charge of multiple slaves named the *tasktrackers*, one mapped on each node. The input data is split into chunks of equal size, usually of 64 MB but the chunk size is parametrable. A MapReduce job in the Hadoop implementation is split into a set of tasks

executed by the tasktrackers as assigned by the jobtracker. Each tasktracker has at its disposal a number of available slots for running tasks. Each active task uses one slot, thus a tasktracker usually executes several tasks simultaneously. When dispatching map tasks to tasktrackers, one of the main objectives of the jobtracker is to keep the computation as close as possible to the data. This is made possible due to the data-layout information previously acquired by the jobtracker. If the work cannot be hosted on the actual node in which the data resides, priority is given to neighboring nodes (*i.e.*, belonging to the same network rack). The jobtracker schedules the map tasks as the reducers have to wait for the completion of the map phase execution to generate the intermediate data. Apart from the data splitting and scheduling responsibilities, the jobtracker is also responsible for monitoring tasks and handling failures.

V. MapReducing GEPETO

Efficiently analyzing large geolocated datasets composed of millions of mobility traces requires both distribution and parallelization. In particular, partitioning the dataset into independent small chunks assigned to distinct nodes will speed up the execution of algorithms implemented within GEPETO. Therefore, we believe that these algorithms (in particular the clustering ones) represent good candidates to be abstracted in the MapReduce formalism. In the following subsections, we describe how some algorithms of GEPETO can be adapted to the MapReduce programming model. Basically, this adaptation requires to structure the algorithm/application into Map/Reduce phases (which is sometimes a highly non-trivial task), before implementing it on top of Hadoop. More precisely, a developer must define three classes that extend the native Hadoop classes and interfaces:

- the *Mapper* class implements the *map* phase of the application,
- the *Reducer* class deals with the *reduce* phase of the application and
- the *Driver* class specifies to the Hadoop framework how to run and schedule the MapReduce processes.

A. Experimental setup

Platform. Our experiments were carried out on the Grid'5000 [17], [5] experimental platform. Grid'5000 is a testbed that can be used by researchers to conduct experiments related to distributed and parallel computing. The infrastructure of Grid'5000 consists in a highly-configurable environment, enabling users to perform experiments under real-life conditions for all software layers ranging from network protocols up to applications. The Grid'5000 platform is physically distributed on different sites across several French cities. In particular, more than 20 clusters spread over 10 sites are available, and each cluster includes up to 64 computing multi-core nodes, summing up to more than 7000 CPU cores overall. The particular testbed that we have used comprises nodes belonging to the Parapluie cluster located in Rennes. In Parapluie, each node is equipped with 2 AMD @1.7GHz CPUs, equipped with 12 cores per CPU, 48 GB of RAM and 232 GB of storage per node. The standard deployment environment used for our experimental setup, allocates one node to the jobtracker, one node to the namenode, while the rest of the nodes is assigned to datanodes and tasktrackers.

Dataset. In our experiments, we used the *GeoLife GPS trajectories* [29], [30], [31], which is a geolocated dataset collected by Microsoft Research Asia from 178 users during a time-span of several years (from April 2007 to August 2012). The whole dataset is composed of 18000 files summing up to 1,61 GB in total. Basically, each file contains a single trajectory for a specific user and it is contained in a directory named after this user's identifier (this folder stores all the user's GPS trajectories). In the GeoLife dataset, a trail of traces (*i.e.*, GPS trajectory) consists of a sequence of mobility traces belonging to the same user. The generic structure of a line in a GeoLife file is shown in Figure 1, along with a concrete example of a GeoLife log. The *latitude*, *longitude* and *altitude* fields specify the spatial coordinate in decimal degrees. The third field in Figure 1 has no meaning for this particular dataset. The fifth field represents the date as the number of days elapsed since $12/30/1899$. Finally, the two remaining fields contain the date and time as string values, thus acting as the timestamp of the trace.

latitude	longitude	0	altitude	date(days)	date	time
39.993207	116.32682	0	217.19159	2000-01-01	36526.96901	23:15:23

Fig. 1. Example of the structure of a GeoLife mobility trace.

The dataset contains 17621 trajectories (with an average of approximately 124 794 traces per user) for a total distance of about 1.2 millions kilometers and a total duration of more than 48 000 hours. This data has been collected from different GPS loggers and GPS-phones at different sampling rates. However, most of the trajectories are very dense (*i.e.*, a mobility trace is recorded every 1 to 5 seconds or every 5 to 10 meters) and correspond to outdoor movements of users. For instance, apart from daily life activities, such as going home or going to work, the collected data also contains several activities such as shopping, dining, hiking, sightseeing and cycling.

VI. A first illustrative example: sampling

(Down)sampling is a form of temporal aggregation in which a set of mobility traces that have occurred within a time window are merged into a single mobility trace, which is called the *representative* trace. Thus, sampling summarizes several mobility traces into a single one.

Considering a time window of size t composed of a sequence of mobility traces that have occurred during this time window, we have implemented two sampling techniques.

1) The first sampling technique takes the trace closest to the upper limit of the time window as the representative one (Figure 2) while

2) the second technique chooses the trace closest to the middle of the time interval (Figure 3).

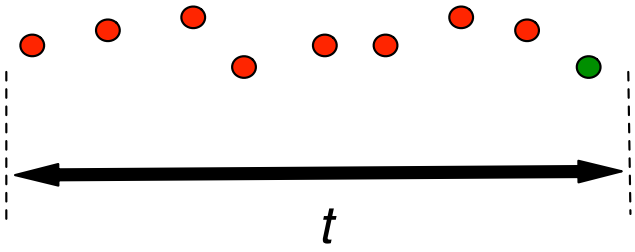

Fig. 2. Sampling by taking the mobility trace closest to the upper limit of the time window as the representative one.

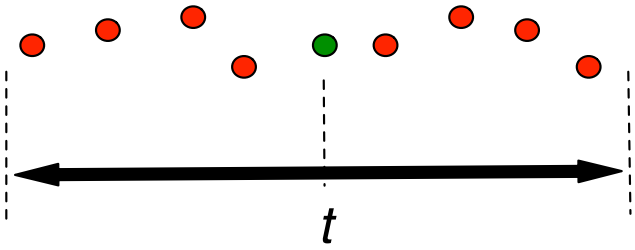

Fig. 3. Sampling by taking the mobility trace closest to the middle of the time window as the representative one.

MapReduce implementation of sampling. Both sampling techniques have been implemented as MapReduce applications consisting only of map phases. The reduce phase is not necessary as sampling represents a computationally cheap operation and can be performed in a single pass. Each map task reads its input chunk and processes each line of the chunk, corresponding to a mobility trace. In GeoLife, each trace is composed of the location and the timestamp, plus additional information about the speed associated to this trace. All the mapper tasks process in parallel their respective chunks by executing the same code. In a nutshell, for each time window, the mapper artificially generates a "reference trace", which is either the trace located at the end or at the middle of the time window depending on the used sampling technique. Afterwards, the current mobility trace read from the chunk is compared against the reference trace. Then, the result of this comparison determines if this trace is kept or not, as only the trace closest to the reference trace is outputted by the mapper.

Running sampling with Hadoop. The initial GeoLife dataset of 1,61 GB is split into 64 MB-size chunks, leading to a total of 26 mapper tasks. The experimental testbed consisted of 7 nodes on the Parapluie cluster, thus each node executes approximately 4 mapper tasks. The user can specify as input parameters: the size of the considered time window and the desired sampling technique as well as the input and output folders.

For a time window of size 10 seconds, the completion of the sampling process on the overall dataset takes 1 minute and 4 seconds. Table I summarizes how the size of the dataset changes after sampling at different rates (namely 1 minute, 5 minutes and 10 minutes). It can be observed that the number of traces decreases drastically even when downsampling with a rate of 1 minute, which is not really surprising due to the dense nature of the GeoLife dataset in which the GPS logs were collected every 1 to 5 seconds.

Initial dataset	1min sampling	5min sampling	10min sampling
2033686	155260	41263	23596

TABLE I
NUMBER OF TRACES IN THE GEOLIFE DATASET UNDER DIFFERENT SAMPLING CONDITIONS: NO SAMPLING, SAMPLING RATES OF 1, 5 AND 10 MINUTES.

Finally, Figure 4 displays on the map the difference in the number of traces generated after applying sampling at a time window of 1 minute and 10 minutes.

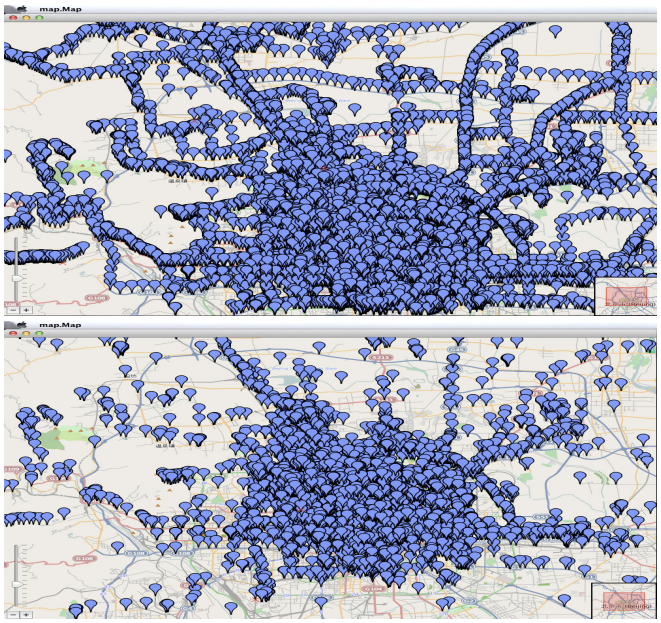

Fig. 4. Sampling rate of 1 minute versus 10 minutes.

VII. CLUSTERING WITH k-MEANS

The k-means algorithm [21] is a classical clustering algorithm partitioning a set of objects (*e.g.*, datapoints) into k clusters of objects that are similar, by trying to minimize the average distance between the objects in a cluster and its centroid, hence the name k-means. An object is generally represented as a vector of d attributes, thus corresponding to a datapoint in a d-dimensional space. The algorithm takes as input a dataset composed of n points and k the number of clusters returned by k-means. The output of k-means is the k clusters as well as their respective centroids. The parameter k

has to be specified by the user or inferred by cross-validation. The user also defines a distance metric that quantifies how close or far are two points relative to each other. Typical examples of distance include the Euclidean distance and the Manhattan distance (*i.e.*, L_1 norm).

The generic sketch of k-means is the following:

1) Randomly choose k points from the input dataset as initial centroids of the k clusters (*initialization* phase).
2) For each point, assign it to the cluster corresponding to the closest centroid (*assignment* step).
3) For each cluster, compute the new centroid by averaging the points assigned to this cluster (*update* step).
4) Repeat from step 2) until convergence (*i.e.*, clusters are stable or after a bounded number of iterations).

In general, averaging the points assigned to a cluster is done directly by computing the arithmetic mean of all the points dimension by dimension (*i.e.*, attribute by attribute). The algorithm has been proven to converge after a finite number of iterations. The clustering generated by k-means is influenced by parameters such as the distance used, the method for choosing the initial centers of the clusters as well as the choice of the parameter k itself.

While simple, the k-means algorithm can have a high time complexity when applied on large datasets. Other limitations of the algorithm include the possibility of being trapped in a local minimum (*i.e.*, finding the optimal k-means clustering is NP-Hard) and its sensitivity to changes in the input conditions. For instance, for two different random initializations of the clusters' centers, it is highly probable that the two associated clustering outputted by the algorithm will be significantly different, which makes k-means non-deterministic. In addition, if it is manually tuned, k the number of clusters must be known before the clustering begins, which may be difficult for users to specify in advance for some type of data. Finally, another drawback of using the mean as the center of the cluster instead of the median, is that outliers can have a sensible impact on a generated center. This can be a significant problem when the input is the geolocated dataset, as a single noisy and uninteresting mobility trace may pull the center of the cluster towards it much more than it should.

MapReducing k-means. Consider the situation in which the input dataset is a geolocated one. MapReducing the k-means algorithm amounts to MapReducing each iteration of the algorithm, thus implementing each k-means iteration as a MapReduce job. The initialization phase of the algorithm randomly picks k mobility traces as initial centroids of the clusters. This phase requires no distribution because it is computationally cheap and can be performed by a single node. In contrast, the two steps of an iteration, the assignment step and the update step, are perfect candidates for being MapReduced. More precisely, the map phase is in charge of assigning each mobility trace to the closest centroid while the reduce phase computes the new centroid of each cluster previously built by the mappers. The program iterates over the input points and clusters, outputting a new directory "clusters-i" containing clusters files for the i^{th} iteration. This process uses a mapper/reducer/main as follows:

- kMeansMapper (Algorithm 3, *cf.* Appendix)
  1) Read the input clusters during the setup() method from the clusters file.
  2) Assign each input trace to the nearest cluster by using the distance specified by the user.
  3) Output: (intermediate key, intermediate value) $\leftarrow$ (centroid identifier, assigned trace).
- kMeansReducer (Algorithm 4, *cf.* Appendix)
  1) Receive all traces assigned to a cluster (as specified by the intermediate key of this reducer).
  2) Compute the new centroid of the cluster by averaging all the traces assigned to the cluster.
  3) Verify if the updated centroid differs from the previous one to detect convergence.
  4) Write the cluster and its new centroid to the clusters file.
  5) Output the new centroid: (output key, output value) $\leftarrow$ (centroid identifier, new centroid).
- kMeans Main (Algorithm 5, *cf.* Appendix)
  1) Randomly select k traces as initial centroids and write them to the clusters file.
  2) Iterate until all output clusters have converged or until a predefined maximum number of iterations has been reached.
  3) For each iteration, a new clusters directory "clusters-i" is produced and the clusters outputted by the current iteration are used as input for the next one.

All these steps as well as the way centroids are updated during each iteration are summarized in Figure 5.

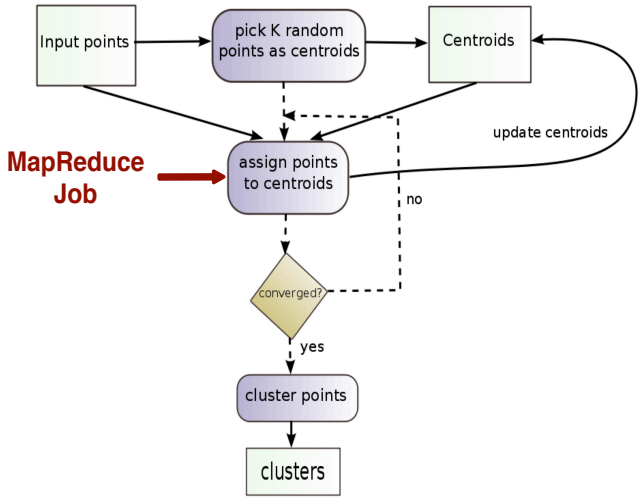

Fig. 5. Workflow for the MapReduced version of k-means.

Running k-means on Hadoop. In the following, we report on the performance obtained when running our MapReduced implementation of k-means on the GeoLife dataset. We designed

several testing scenarios in order to measure how varying the input parameters impacts the performance of k-means. For each scenario, we tuned parameters such as the dataset size and the distance used. For measuring the distance between points, we considered two metrics: the *squared Euclidean distance* and the *Haversine distance*. The squared Euclidean distance uses the same formula as the standard Euclidean distance, but without computing the square root part. As a consequence, clustering with the squared Euclidean distance is faster than clustering with the regular one while preserving the order relationship between different points. The Haversine formula [24] computes the distance between two points over the earth's surface by taking into account the shape of the earth.

The metric used to assess the performance of the implementation is the time required to complete one iteration. Note that the number of iterations required by k-means to converge depends on the initial selection of centroids. In our experiments, the number of iterations reported corresponds to a rounded average of 3 to 5 trials. The runtime arguments for k-means in our implementation are the following:

argument	role
input path	path to the directory containing the input files
output path	path to the directory to which the output will be written
input file	the input file from which the initial centroids will be generated
clusters path	path to the directory storing the current centroids
k	number of clusters outputted by the algorithm
distanceMeasure	name of the metric used for measuring distance between points
$convergencedelta$	value used for determining the convergence after each iteration
$maxIter$	maximum number of iterations

TABLE II
RUNTIME ARGUMENTS FOR k-MEANS.

The k-means algorithm was executed with the following parameters:

argument	value
k	11
maxIter	150
convergenceDelta	0.5
distanceMeasure	Squared Euclidean or Haversine

TABLE III
RUNTIME ARGUMENTS FOR k-MEANS.

The k-means algorithm was run on two datasets that are actually subsets of the original GeoLife dataset; the first dataset is composed of 90 users and has a size of 66 MB while the bigger one is composed of 178 users and is of size 128 MB. The experiments were carried out on the Parapluie cluster of the Grid'5000 platform. For Hadoop, the namenode was deployed on a dedicated machine, the jobtracker on another machine, and finally the datanodes and the tasktrackers on the remaining nodes (with one entity per machine), leading to a total of 7 nodes overall. Table IV summarizes our results obtained under different testing scenarios, corresponding to different values for the runtime arguments. The fifth column of the table contains the time (measured in seconds) required to run a k-means iteration with our MapReduce implementation.

We observe that a crucial parameter having a big influence on the computational time is the chunk size. Usually, in Hadoop the chunk size can be set to two values: 32 MB and 64 MB. A smaller chunk size leads to a larger number of chunks, which in turn generates more map tasks. Obviously, a higher number of mappers working in parallel will improve the computational time. Moreover as expected, the Haversine distance increases the execution time of an iteration compared to the squared Euclidean distance due to the more complex computations that the Haversine formula requires.

Data (MB)	Nb of traces	Distance	Chunk (MB)	Iter. time (sec)	Nb of iter. to converge
66	1.050.000	Haversine	64	57	73
66	1.050.000	Squared Euclidean	64	48	72
66	1.050.000	Squared Euclidean	32	41	70
66	1.050.000	Haversine	32	45	73
128	2.033.686	Squared Euclidean	64	51	85
128	2.033.686	Squared Euclidean	32	45	83
128	2.033.686	Haversine	32	48	89
128	2.033.686	Haversine	64	60	93

TABLE IV
RESULTS OF THE MAPREDUCED k-MEANS EXPERIMENTATIONS.

VIII. DJCLUSTER

Density-Joinable Cluster (DJ-Cluster) [32] is a density-based clustering algorithm that looks for dense neighborhoods of traces. The density of the neighborhood is defined mainly by two parameters: the radius r of a circle defining the neighborhood and the number of points contained within this circle, which must be greater than a predefined lower bound of $MinPts$. The rationale of the algorithm is that clusters correspond to areas with a higher density of points than areas outside the clusters. At the end of the clustering process, the datapoints that do not belong to a cluster will be considered as noise.

Most of the limitations of k-means are overcome by density-based clustering algorithms. In particular, clusters of arbitrary shapes can be discovered with most of the density-based clustering algorithms while traditional ones are usually limited to spherical ones. Moreover, the resulting clusters usually contain less outliers and noise. Finally, most of the density-based clustering algorithms are deterministic and therefore, their output is stable and not affected by the order in which points are processed.

The DJ-Cluster algorithm proceeds in three phases:

1) The *preprocessing phase* discards all the traces corresponding to a movement (*i.e.*, whose speed is above ϵ, a small predefined value) and replaces all sequences of repeated stationary traces with a single trace.
2) The *neighborhood identification phase* computes the neighborhood of each trace, which corresponds to points that are within distance r from the point currently considered with the additional constraint that at least *MinPts* should be contained within this neighborhood. If no such neighborhood exists, the current trace is labeled as *outlier* (or *noise*).
3) The *merging phase* joins all the clusters sharing at least one common trace into a single cluster.

MapReducing DJ-Cluster. Each of the three phases of DJ-Cluster can be expressed in the MapReduce programming model. Thereafter, we describe how each phase can be implemented as one or several MapReduce jobs.

A. Preprocessing phase

The first phase of DJ-Cluster preprocesses the mobility traces by applying two filtering techniques that remove the mobility traces that are not interesting or might even perturb the clustering process. These two filtering techniques have been implemented in the form of two MapReduce jobs executed in pipeline. More precisely, the output of the first job constitutes the input of the second one. During the first MapReduce job, stationary traces are kept while moving ones are discarded.

Identifying the moving traces amounts to measuring the speed of each trace and then removing the traces whose speed is higher than a predefined threshold ϵ (for ϵ a small value). The speed of a trace is computed as the distance traveled between the previous and the next traces divided by the corresponding time difference. The computation of the speed for each trace can be performed only by map tasks. Each mapper reads its corresponding data chunk and outputs only the traces whose speed is less than ϵ. As no aggregation or additional filtering is required, the implementation of this part does not include a reduce phase.

The second filtering technique removes redundant consecutive traces, which correspond to mobility traces that have (almost) the same spatial coordinate but different timestamps. Similarly to the first filtering technique, only the map phase is needed. The role of the mapper is simply to output the first trace from a sequence of traces that are redundant.

Figure 6 shows the two pipelined MapReduce jobs implementing the preprocessing phase of DJ-Cluster.

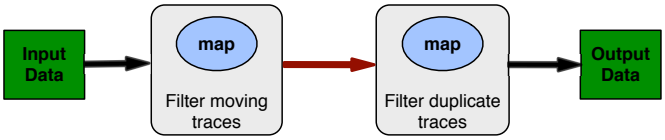

Fig. 6. First Phase of DJ-Cluster with MapReduce

These two filtering techniques reduce considerably the amount of data that needs to be processed by the clustering algorithm. In order to measure this reduction of the dataset size, we ran a set of experiments that apply the preprocessing phase on the sampled datasets at different sampling rates of respectively 1, 5 and 10 minutes (see Section VI, Table I). Table V shows the number of traces remaining after the preprocessing phase. The value of ϵ, the threshold speed, was set to 0.72 km/h (which is equivalent to 2 $m/10$ s).

Sampling rate	**Unfiltered**	**Filter moving traces**	**Remove duplicates**
1 min	155260	86416	85743
5 min	41263	23996	23894
10 min	23596	14207	14174

TABLE V
NUMBER OF TRACES IN THE SAMPLED DATASETS AFTER THE PREPROCESSING PHASE.

B. Neighborhood computation and merging of clusters

The main challenge of the second phase of DJ-Cluster is to design an efficient method for discovering the neighbors of a datapoint. We chose to rely on a data structure known as *R-Tree* [15], as computing the neighborhood of a point with such a data structure can be done in $O(n \log n)$, for n the size of the dataset. The construction of an R-Tree is described in subsection VIII-C, but for now we assume that an R-Tree indexing all the mobility traces in the dataset is stored in the distributed cache of Hadoop and can be read by any tasktracker.

The computation of the neighborhood for each trace in the dataset is a good candidate for parallelization and distribution. Indeed, splitting the dataset into small chunks processed by different nodes is likely to speed up the computation process. The second and the third parts of DJ-Cluster can be modeled respectively as a map and as a reduce phases. Before processing its chunk, a mapper first loads the R-Tree from the distributed cache while executing its setup method. For each trace in the chunk, the mapper computes the set of neighbors within a distance r from the current trace. More precisely, the mapper searches for the nearest neighbors of a trace by relying on the R-tree. This searching process traverses mainly the branches of the R-Tree in which neighbors may be located. If the size of the computed neighborhood has less than $MinPts$ elements, the mapper marks the current trace as noise. The $(key, value)$ pair outputted by the mapper corresponds to a trace and its associated neighborhood. Algorithm 1 describes more precisely this process.

Algorithm 1 DJ-CLuster Mapper

```
setup(Configuration conf)
    rTree ← load from file in distributed cache
map(K key, V value)
    trace ← value
    neighborhood ← rTree.kNN(trace, MinPts, r)
    if neighborhood.size < MinPts
      trace.markAsNoise
    emitIntermediate(const, neighborhood)
```

A single reducer implements the last phase of the algorithm as the merging of joinable neighborhoods must be done by a centralized entity that obtains a knowledge about all current neighborhoods built by mappers. In the intermediate $(key, value)$ pair emitted by mappers, the key field is set to a constant value such that all pairs are collected by a single reducer (*i.e.*, all intermediate pairs that have the same key field are redirected towards the same reducer). This reducer collects all neighborhoods outputted by the mappers and then proceeds to building the clusters. The construction of clusters

is done by merging all joinable neighborhoods. By definition, two neighborhoods are joinable if there exists at least one trace such that both neighborhoods contain it. The reducer merges all joinable neighborhoods with existing clusters or creates new clusters if a neighborhood cannot be joined with any of the existing clusters. Thus, by the end of the clustering process, each trace is either assigned to a cluster or marked as noise. In addition, the clusters are assured to be non-overlapping and to contain at least $MinPts$ mobility traces. The output of this reduce phase, which is also the output of the algorithm, consists in the computed clusters. All the above steps are shown in Algorithm 2.

Algorithm 2 DJ-Cluster Reducer

```
setup(Configuration conf)
    clusters ← ∅
reduce(K key, V[] values)
    for neighborhood ∈ values
        for cluster ∈ clusters
            if neighborhood.intersects(cluster)
                cluster ← cluster.merge(neighborhood)
            else
                new_cluster ← neighborhood
    for cluster ∈ clusters
        emit (clusterId, cluster)
```

C. Constructing an R-tree with MapReduce

R-Trees [15] are data structures commonly used for indexing multidimensional data. In a nutshell, an R-Tree groups datapoints into clusters and represents them through their *minimum bounding rectangle* in their upper level in the tree. At the leaf level, each rectangle contains only a single datapoint (*i.e.*, each rectangle is a point) while higher levels aggregate an increasing number of datapoints. When querying an R-Tree, only the bounding rectangles intersecting the current query are traversed.

The indexing of large datasets into an R-Tree structure can be implemented as a MapReduce algorithm [6]. In this previous work, each point in the dataset is defined by two attributes: a location in some spatial domain used to guide the construction of the R-Tree and a unique identifier used to reference the object in the R-Tree. The construction of an R-Tree is a process that can be split into three phases:

1) The partition of datapoints into p clusters.
2) The indexing of each cluster into a small R-Tree.
3) The merging of the small R-Trees obtained from the previous phase into a final one indexing the whole dataset.

The first two phases are MapReduced while the last phase is executed sequentially by a single node due to its low computational complexity. Figure 7 illustrates how these three phases are executed sequentially.

More precisely, the first phase computes a *partitioning function* assigning each datapoint of the initial dataset to one of the p partitions. This partitioning function should yield equally-sized partitions and preserve at the same time data locality (*i.e.*, points that are close in the spatial domain should be assigned to the same partition). In order to satisfy these constraints, the partitioning function has to map multidimensional datapoints into an ordered sequence of unidimensional values. In practice, this transformation is performed with the help of *space-filling curves* that precisely map multidimensional data to one dimension while preserving data locality. In our R-Tree construction, we implemented and tested two types of space-filling curves: the *Z-order* curve and the *Hilbert* curve [20]. The MapReduce design for this phase consists of several mappers and one reducer. Each mapper (Algorithm 6, *cf.* Appendix) samples a predefined number of objects from its data chunk and outputs the corresponding single-dimensional values obtained after applying the space-filling curve. Afterwards, the reducer (Algorithm 7, *cf.* Appendix) collects a set of single dimensional values from all mappers, orders this set and then determines $(p-1)$ partitioning points in the sequence (partitioning points delimit the boundaries of each partition).

The second phase concurrently builds p individual R-Trees by indexing the partitions outputted by the first phase. The mappers (Algorithm 8, *cf.* Appendix) partition the dataset into p partitions using the space-filling curve computed during the first phase. Each mapper processes its chunk and assigns each object it reads to a partition identifier. The intermediate key is represented by the partition identifier such that all datapoints sharing the same key (*i.e.*, belonging to the same partition) will be collected by the same reducer. Then, each reducer (Algorithm 9, *cf.* Appendix) constructs the R-Tree associated with its partition, leading to a total of p reducers building p small R-Trees.

Finally, the last phase merges the small R-Trees into a global one that indexes all the datapoints of the initial dataset. For values of p not exceeding thousands, this process can be executed by a single node. For building the small R-Trees and then merging them into a global one, we use an existing off-the-shelf implementation. More precisely, the *Java Spatial Index* library (JSI) is an open-source implementation of an R-tree in Java whose main objective is to provide fast intersection query performance. JSI contains Java classes and structures defining nodes, rectangles and R-trees. It also provides procedures for adding and deleting a node to/from an R-tree and for finding the neighbors within a specific distance from a datapoint. JSI relies on the *Trove* library for defining high speed, regular and primitive collections for Java. Trove is a fast and lightweight implementation of the *java.uti.Collections API*. The main idea behind Trove is to use customized hashing strategies. In particular, it is possible to save time with a customized hashing function by skipping portions of the query that remain invariant.

IX. Conclusion

In this paper, we have proposed to adopt the MapReduce paradigm in order to be able to perform a privacy analysis on large scale geolocated datasets composed of millions of

mobility traces. More precisely, we have developed a complete MapReduce-based approach to GEPETO (for GEoPrivacy-Enhancing TOolkit) [9], a software that can be used to design, tune, experiment and evaluate various sanitization algorithms and inference attacks on location data as well as to visualize the resulting data. Most of the algorithms used to conduct an inference attack represent good candidates to be abstracted in the MapReduce formalism. For instance, we have designed MapReduced version of sampling as well as the k-means and the DJ-Cluster clustering algorithms and integrate them within the framework of GEPETO. These algorithms have been implemented with Hadoop and evaluate on a real dataset. Preliminary results show that the MapReduced versions of the algorithms can efficiently handle millions of mobility traces.

Currently, the clustering algorithms that we have implemented can be used primarily to extract the POIs of an individual from his trail of mobility traces, which correspond only to one possible type of inference attack. In the future, we aim at integrating other inference techniques within the MapReduced framework of GEPETO. In particular, we want to develop algorithms for learning a mobility model out of the mobility traces of an individual, such as Mobility Markov Chains (MMCs) [10]. In a nutshell, a MMC represent in a compact way the mobility behavior of an individual and can be used to predict his future locations or even to perform de-anonymization attacks, thus extending the range of inference attacks available within GEPETO. We also want to design MapReduced version of geo-sanitization mechanisms such as geographical masks that modify the spatial coordinate of a mobility trace by adding some random noise or aggregate several mobility traces into a single spatial coordinate. More sophisticated geo-sanitization methods will also be integrated at a later stage such as spatial cloaking techniques [14] and mix zones [4].

ACKNOWLEDGMENT

This work was funded by the "location privacy" activity of EIT ICT labs. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

[1] The Apache Hadoop Project. http://www.hadoop.org.

[2] The Hadoop MapReduce Framework. http://hadoop.apache.org/mapreduce/.

[3] L. O. Alvares, V. Bogorny, B. Kuijpers, J. A. F. de Macêdo, B. Moelans, and A. A. Vaisman. A model for enriching trajectories with semantic geographical information. In *Proceedings of the 15th ACM International Symposium on Geographic Information Systems*, page 22, 2007.

[4] A. R. Beresford and F. Stajano. Mix zones: User privacy in location-aware services. In *Proceedings of the Workshops of the 2nd IEEE Conference on Pervasive Computing and Communications*, pages 127–131, 2004.

[5] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, Seattle, Washington, USA, November 2005.

[6] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, pages 302–319, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] G. De Francisci Morales, A. Gionis, and M. Sozio. Social content matching in MapReduce. *Proceedings of Very Large Data Base (VLDB) Endowment*, 4:460–469, April 2011.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] S. Gambs, M.-O. Killijian, and M. N. del Prado. GEPETO: a GEoPrivacy Enhancing Toolkit. In *Proceedings of the International Workshop on Advances in Mobile Computing and Applications: Security, Privacy and Trust, held in conjunction with the 24th IEEE AINA conference, Perth, Australia*, pages 1071–1076, April 2010.

[10] S. Gambs, M.-O. Killijian, and M. N. del Prado Cortez. Show me how you move and I will tell you who you are. *Transactions on Data Privacy*, 4(2):103–126, 2011.

[11] Y. Ganjisaffar, T. Debeauvais, S. Javanmardi, R. Caruana, and C. V. Lopes. Distributed tuning of machine learning algorithms using MapReduce clusters. In *Proceedings of the 3rd Workshop on Large Scale Data Mining: Theory and Applications*, LDMTA '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM.

[12] P. Golle and K. Partridge. On the anonymity of home/work location pairs. *Proceedings of the 7th International Conference on Pervasive Computing*, pages 390–397, May 2009.

[13] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, June 2008.

[14] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services*, 2003.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *Proceedings of Annual Meeting of ACM SIGMOD*, pages 47–57. ACM Press, 1984.

[16] L. Jedrzejczyk, B. A. Price, A. K. Bandara, and B. Nuseibeh. I know what you did last summer: Risks of location data leakage in mobile and social computing. *Department of Computing Faculty of Mathematics, Computing and Technology The Open University*, November 2009.

[17] Y. Jégou, S. Lantéri, J. Leduc, M. N., G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E. Talbi, and T. Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[18] J. H. Kang, B. Stewarta, G. Borriello, and W. Welbourne. Extracting places from traces of locations. In *Proceedings of the 2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, pages 110–118, 2004.

[19] J. Krumm. Inference attacks on location tracks. *Pervasive Computing*, pages 127–143, 2007.

[20] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of the 17th British National Conference on Databases*, pages 20–35, London, UK, 2000. Springer-Verlag.

[21] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[22] A. Y. Ng, G. Bradski, C.-T. Chu, K. Olukotun, S. K. Kim, Y.-A. Lin, and Y. Yu. MapReduce for machine learning on multicore. In *Proceedings of the 20th Annual Conference on Neural Information Processing Systems*, December 2006. Selected for Oral Presentation.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[24] R. W. Sinnott. Virtues of the haversine. In *Sky and Telescope*, volume 68, page 159, 1984.

[25] C. Song, Z. Qu, N. Blumm, and A.-L. Barabasi. Limits of predictability in human mobility. *Science*, 327(5968):1018–1021, 2010.
[26] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. F. de Macêdo, F. Porto, and C. Vangenot. A conceptual view on trajectories. *Data Knowl. Eng.*, 65(1):126–146, 2008.
[27] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.
[28] K. Wiley, A. Connolly, J. P. Gardner, S. Krughof, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu. Astronomy in the cloud: Using MapReduce for image coaddition. *CoRR*, abs/1010.1015, 2010.
[29] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on gps data. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, UbiComp '08, pages 312–321, New York, NY, USA, 2008. ACM.
[30] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IIEEE Data(base) Engineering Bulletin*, 33(2):32–39, 2010.
[31] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 791–800, New York, NY, USA, 2009. ACM.
[32] C. Zhou, D. Frankowski, P. Ludford, S. Shekhar, and L. Terveen. Discovering personal gazetteers: An interactive clustering approach. In *Proceedings of the 12th ACM International Workshop on Geographic Information Systems*, pages 266–273. ACM Press, 2004.

X. Appendix

Algorithm 3 kMeans Mapper

```
setup(Configuration conf)
    centroids ← load from file
map(K key, V value)
    trace ← value
    min ← MAX_VALUE
    for center ∈ centroids
        if distance(trace, center) < min
            min ← distance(trace,center)
            assign ← center
    emitIntermediate(assign, trace)
```

Algorithm 4 kMeans Reducer

```
setup(Configuration conf)
    centroids ← load from file
reduce(K key, V[] values)
    for v ∈ values
        new_centroid ← average(v)
    emit (key, new_centroid)
```

Algorithm 5 kMeans Main

```
randomCenters(Configuration conf)
    centers ← randomly choose k centroids
    write to file
main()
    i ← 0
    while true
        submit MapReduce job for iteration i
        i ← i + 1
        if hasConverged or i = maxIter
            stop
```

Algorithm 6 R-Tree First Phase Mapper

```
map(K key, V value)
    sample ← randomly sample objects in chunk
    scalar ← space_filling_curve(sample)
    emitIntermediate(const, scalar)
```

Algorithm 7 R-Tree First Phase Reducer

```
reduce(K key, V[] values)
    order(values)
    step ← total_samples ÷ R
    count ← 0
    for i ∈ values.size
        if i mod step = 0
            emit (count, values[i[)
            count ← count + 1
```

Algorithm 8 R-Tree Second Phase Mapper

```
setup(Configuration conf)
    partitions ← load ouput of first phase
map(K key, V value)
    scalar ← space_filling_curve(value)
    for i ∈ [0 : R]
        if partitions[i] < scalar < partitions[i + 1]
            partitionId ← i
            break
    emitIntermediate(partitionId, value)
```

Algorithm 9 R-Tree Second Phase Reducer

```
reduce(K key, V[] values)
    partitionId ← key
    tree ← build_RTree(values)
    emit (tree, treeRoot)
```

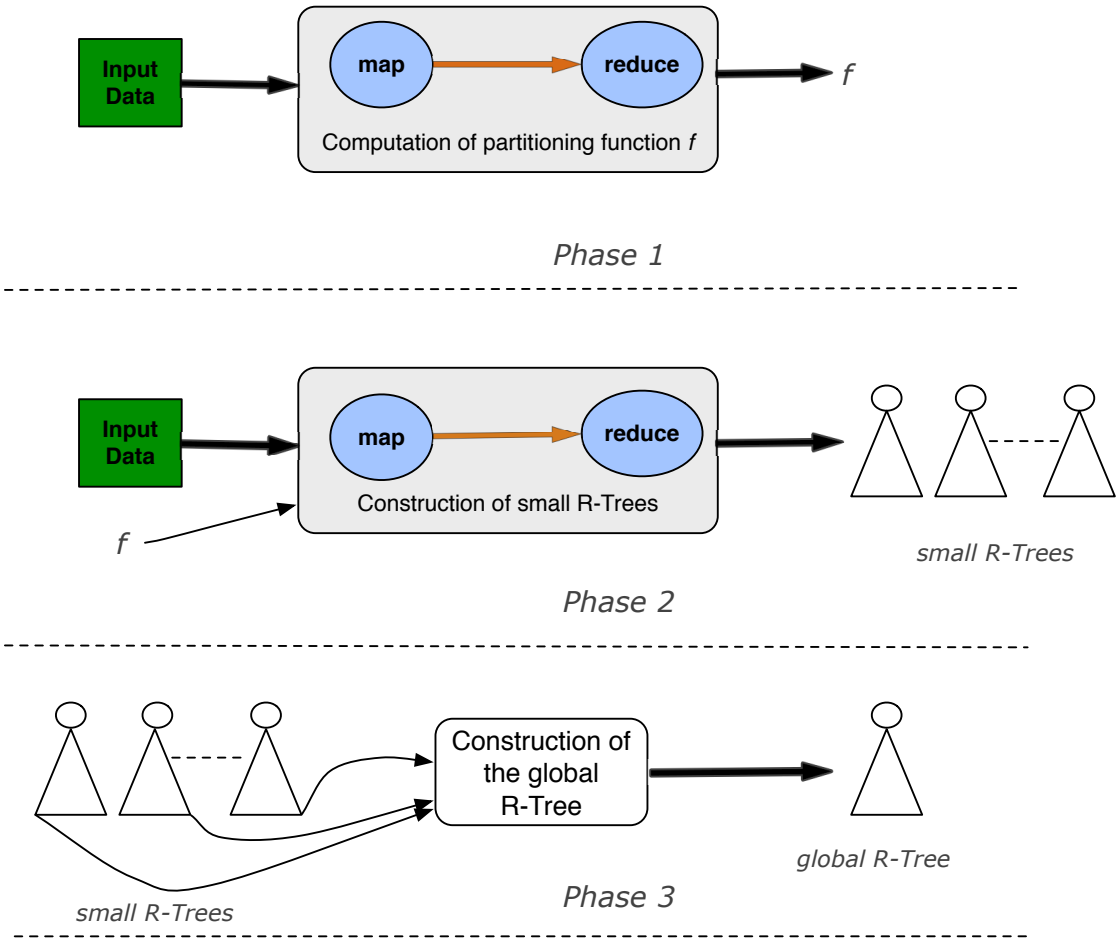

Fig. 7. Building an R-Tree with MapReduce