This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

A Survey on Geographically Distributed Big-Data Processing using MapReduce

Shlomi Dolev, *Senior Member, IEEE,* Patricia Florissi, Ehud Gudes, *Member, IEEE Computer Society,* Shantanu Sharma, *Member, IEEE,* and Ido Singer

Abstract—Hadoop and Spark are widely used distributed processing frameworks for large-scale data processing in an efficient and fault-tolerant manner on private or public clouds. These big-data processing systems are extensively used by many industries, *e.g.*, Google, Facebook, and Amazon, for solving a large class of problems, *e.g.*, search, clustering, log analysis, different types of join operations, matrix multiplication, pattern matching, and social network analysis. However, all these popular systems have a major drawback in terms of *locally distributed* computations, which prevent them in implementing geographically distributed data processing. The increasing amount of geographically distributed massive data is pushing industries and academia to rethink the current big-data processing systems. The novel frameworks, which will be beyond state-of-the-art architectures and technologies involved in the current system, are expected to process geographically distributed data at their locations without moving entire *raw datasets* to a single location. In this paper, we investigate and discuss challenges and requirements in designing geographically distributed data processing frameworks and protocols. We classify and study batch processing (MapReduce-based systems), stream processing (Spark-based systems), and SQL-style processing geo-distributed frameworks, models, and algorithms with their overhead issues.

Index Terms—MapReduce, geographically distributed data, cloud computing, Hadoop, HDFS Federation, Spark, and YARN.

1 Introduction

Currently, several cloud computing platforms, *e.g.*, Amazon Web Services, Google App Engine, IBM's Blue Cloud, and Microsoft Azure, provide an easy *locally distributed*, scalable, and on-demand big-data processing. However, these platforms do not regard geo(graphically) data locality, *i.e.*, geo-distributed data [1], and hence, necessitate data movement to a single location before the computation.

In contrast, in the present time, data is generated geo-distributively at a much higher speed as compared to the existing data transfer speed [2], [3]; for example, data from modern satellites [4]. There are two common reasons for having geo-distributed data, as follows: (*i*) Many organizations operate in different countries and hold datacenters (DCs) across the globe. Moreover, the data can be distributed across different systems and locations even in the same country, for instance, branches of a bank in the same country. (*ii*) Organizations may prefer to use multiple public and/or private clouds to increase reliability, security, and processing [5], [6], [7]. In addition, there are several applications and computations that process and analyze a huge amount of massively geo-distributed data to provide the final output. For example, a bioinformatic application that analyzes existing genomes in different labs and countries to track the sources of a potential epidemic. The following are few examples of applications that process geo-distributed datasets: climate science [8], [9], data generated by multinational companies [8], [10], [11], sensor networks [9], [12], stock exchanges [9], web crawling [13], [14], social networking applications [13], [14], biological data processing [8], [12], [15] such as DNA sequencing and human microbiome investigations, protein structure prediction, and molecular simulations, stream analysis [9], video feeds from distributed cameras, log files from distributed servers [12], geographical information systems (GIS) [4], and scientific applications [8], [9], [13], [16].

It should be noted down here that all the above-mentioned applications generate a high volume of *raw data* across the globe; however, most analysis tasks require only a small amount of the original raw data for producing the final outputs or summaries [12].

Geo-distributed big-data processing vs. the state-of-the-art big-data processing frameworks. Geo-distributed databases and systems have been in existence for a long time [17]. However, these systems are not highly fault-tolerant, scalable, flexible, good enough for massively parallel processing, simple to program, able to process a large-scale (and/or real-time) data, and fast in answering a query.

On a positive side, several *big-data processing programming models and frameworks* such as MapReduce [18], Hadoop [19], Spark [20], Dryad [21], Pregel [22], and Giraph [23] have been designed to overcome the disadvantages (*e.g.*, fault-tolerance, unstructured/massive data processing, or slow processing time) of parallel computing, distributed databases, and cluster computing. Thus, this survey paper focuses on the MapReduce, Hadoop, and Spark based systems. On a negative side, these frameworks do not regard geo-distributed data locations, and hence, they follow a trivial solution for geo-distributed data processing:

- *S. Dolev is with the Department of Computer Science, Ben-Gurion University of the Negev, Israel. E-mail:* dolev@cs.bgu.ac.il
- *P. Florissi is with Dell EMC, USA. E-mail:* patricia.florissi@dell.com
- *E. Gudes is with the Department of Computer Science, Ben-Gurion University of the Negev, Israel. E-mail:* ehud@cs.bgu.ac.il
- *S. harma is with the Department of Computer Science, University of California, Irvine, USA. E-mail:* shantanu.sharma@uci.edu
- *I. Singer is with Dell EMC, Israel. E-mail:* ido.singer@dell.com

Manuscript received ABCD

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

copy all *raw data* to one location before executing a *locally distributed* computation.

The trivial solution has a bottleneck in terms of data transfer, since it is not always possible to copy the whole *raw data* from different locations to a single location due to security, privacy, legal restrictions, cost, and network utilization. Moreover, if the output of the computation at each site is smaller than the input data, it is completely undesirable to move the raw input data to a single location [13], [24], [25]. In a widely distributed environment with network heterogeneity, Hadoop does not work well because of heavy dependency between MapReduce phases, highly coupled data placement, and task execution [26]. In addition, HDFS Federation [27] cannot support geo-distributed data processing, because DataNodes at a location are not allowed to register themselves at a NameNode of another location, which is governed by another organization/country. Thus, the current systems cannot process data at multiple-clusters.

It is also important to mention that the network bandwidth is also a crucial factor in geo-distributed data movement. For example, the demand for bandwidth increased from 60Tbps to 290Tbps between the years 2011 and 2015 while the network capacity growth was not proportional. In the year 2015, the network capacity growth was only 40%, which was the lowest during the years 2011 and 2014 [28].

Fig. 1 shows an abstract view of desirable geo-distributed data processing, where different locations hold data and a local computation is executed on the site. Each site executes an assigned computation locally distributed and transfers (partial) outputs to the closest site or the user site. Eventually, all the (partial) outputs are collected at a single site (or the user site) that executes another job to obtain the final outputs. Different sites are connected with different speeds (the bandwidth consideration in the context of geo-distributed data processing is given in [10]). The thick lines show high bandwidth networks, and the thinner lines are lower bandwidth networks.

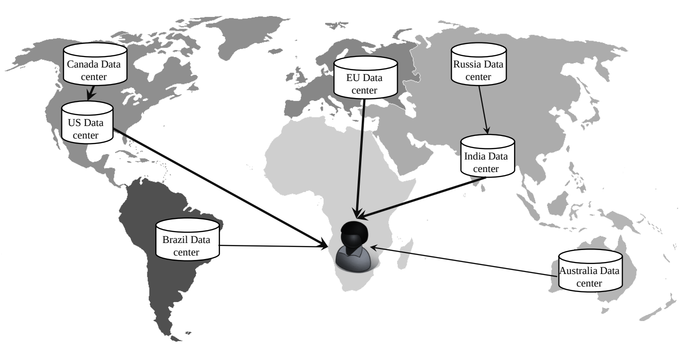


Fig. 1: A scenario for geographically distributed data processing.

Currently, several researchers are focusing on the following important questions: what can be done to process geo-distributed big-data using Hadoop, Spark, and similar frameworks, and how? Can we process data at their local sites and send only the outputs to a single location for producing the final output? The on-site processing solution requires us to rethink, redesign, and revisualize the current implementations of Hadoop, Spark, and similar frameworks. In this work, we will review several models, frameworks, and resource allocation algorithms for geo-distributed big-data processing that try to solve the above-mentioned problems. In a nutshell, geo-distributed big-data processing frameworks have the following properties:

- *Ubiquitous computing*: The new system should regard different data locations, and it should process data at different locations, transparent to users. In other words, new geo-distributed systems will execute a geo-computation like a locally distributed computation on geo-locations and support any type of big-data processing frameworks, languages, and storage media at different locations [10], [16], [24], [29].
- *Data transfer among multiple DCs*: The new system should allow moving only the *desired data*, which eventually participate in the final output,[1] in a secure and privacy-preserving manner among DCs, thereby reducing the need for high bandwidth [10], [29], [31].
- *High level of fault-tolerance*: Storing and processing data in a single DC may not be fault-tolerant when the DC crashes. The new system should also allow data replication from one DC to different trusted DCs, resulting in a higher level of fault-tolerance [32]. (Note that this property is somewhat in conflict with the privacy issues. These types of systems will be reviewed under the category of *frameworks for user-located geo-distributed big-data* in §4.2.)

Advantages of geo-distributed data processing. The main advantages of geo-distributed big-data processing are given in [33] and listed below:

- A geo-distributed Hadoop/Spark-based system can perform data processing across nodes of multiple clusters while the standard Hadoop/Spark and their variants cannot process data at multiple clusters [33].
- More flexible services, *e.g.*, resource sharing, load balancing, fault-tolerance, performance isolation, data isolation, and version isolation, can be achieved when a cluster is a part of a geo-distributed cluster [11], [16].
- A cluster can be scaled dynamically during the execution of a geo-distributed computation [33].
- The computation cost can be optimized by selecting different types of virtual nodes in clouds according to the user requirement and transferring a job to multiple clouds [34].

1.1 Scope of the Review

Today, big-data is a reality yielded by the distributed internet of things that constantly collect and process sensing information from remote locations. Communication and processing across different geographic areas are major resources that should be optimized. Other aspects such as regulations and privacy-preserving are also important criteria. Our paper is also motivated by these important emerging developments of big-data.

A schematic map of the paper is given in Fig. 2. In this paper, we discuss design requirements, challenges, proposed frameworks, and algorithms to Hadoop-based geo-distributed data processing. It is important to emphasize that this work is not only limited to MapReduce-based batch processing geo-distributed frameworks; we will discuss

1. We are not explaining the method of finding only desired data before the computation starts. Interested readers may refer to [30].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

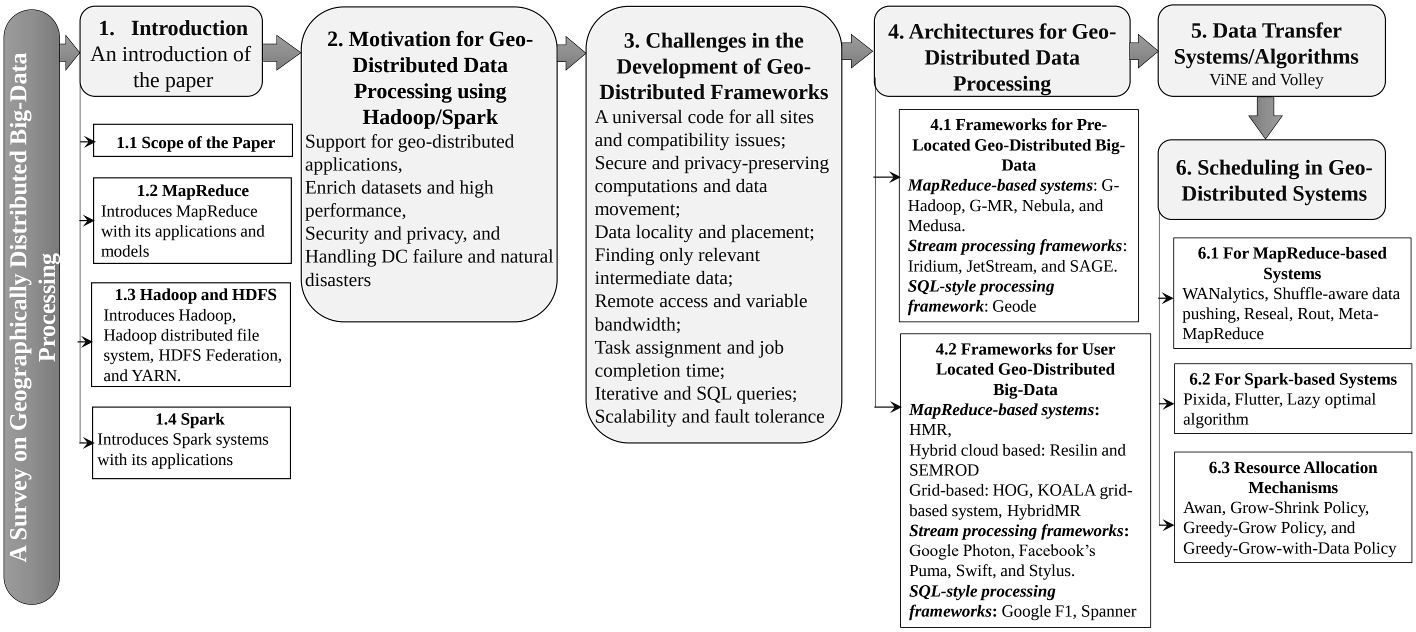


Fig. 2: Schematic map of the paper.

architectures designed for geo-distributed streaming data (SAGE [9] and JetStream [35]), Spark-based systems (Iridium [10]), and SQL-style processing frameworks (Geode [36] and Google's Spanner [37]). Open issues to be considered in the future are given at the end of the paper in §7.

In this survey, we do not study techniques for multi-cloud deployment, management, and migration of virtual machines, leasing cost models, security issues in the cloud, API design, scheduling strategies for non-MapReduce jobs, and multi-cloud database systems.

1.2 MapReduce

MapReduce [18], introduced by Google 2004, provides parallel processing of large-scale data in a timely, failure-free, scalable, and load balance manner. MapReduce (see Fig. 3) has two phases, the *map phase* and the *reduce phase*. The given input data is processed by the map phase that applies a user-defined map function to produce intermediate data (of the form $\langle key, value\rangle$). This intermediate data is, then, processed by the reduce phase that applies a user-defined reduce function to keys and their associated values. The final output is provided by the reduce phase. A detailed description of MapReduce can be found in Chapter 2 of [38].

Applications and models of MapReduce. Many MapReduce applications in different areas exist. Among them: matrix multiplication [39], similarity join [40], [41], detection of near-duplicates [42], interval join [43], [44], spatial join [45], [46], graph processing [47], [48], pattern matching [49], data cube processing [50], [51], skyline queries [52], k-nearest-neighbors finding [53], [54], star-join [55], theta-join [56], [57], and image-audio-video-graph processing [58], are a few applications of MapReduce in the real world. Some efficient MapReduce computation models for a single cloud are presented by Karloff et al. [59], Goodrich [60], Lattanzi et al. [61], Pietracaprina et al. [62], Goel and Munagala [63], Ullman [64], Afrati et al. [65], [66], [67], and Fish et al. [68].

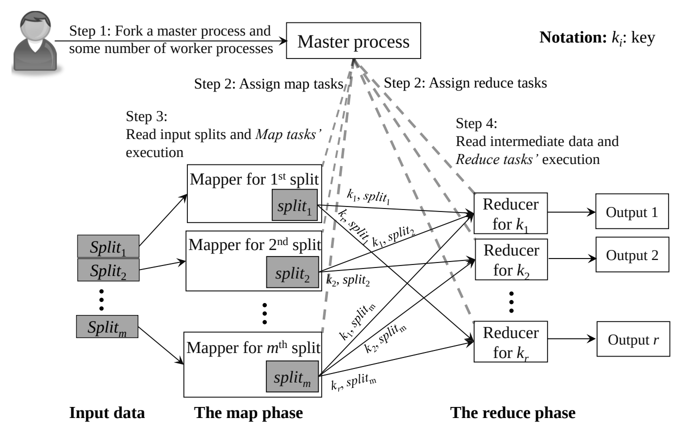


Fig. 3: A general execution of a MapReduce algorithm.

1.3 Hadoop, HDFS, HDFS Federation, and YARN

Hadoop. Apache Hadoop [19] is a well-known and widely used open-source software implementation of MapReduce for distributed storage and distributed processing of large-scale data on clusters of nodes. Hadoop includes three major components, as follows: (*i*) Hadoop Distributed File System (HDFS) [69]: a scalable and fault-tolerant distributed storage system, (*ii*) Hadoop MapReduce, and (*iii*) Hadoop Common, the common utilities, which support the other Hadoop modules.

Hadoop cluster consists of two types of nodes, as; (*i*) a master node that executes a JobTracker and a NameNode and (*ii*) several slave nodes, each slave node executes a TaskTracker and a DataNode; see Fig. 4. The computing environment for a MapReduce job is provided by the JobTracker (that accepts a MapReduce job from a user and executes the job on free TaskTrackers) and TaskTrackers (produces the final output). An environment for distributed file system, called HDFS is supported by the NameNode (manages the cluster metadata and DataNodes) and DataNodes (stores data). HDFS supports `read`, `write`, and `delete` operations

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

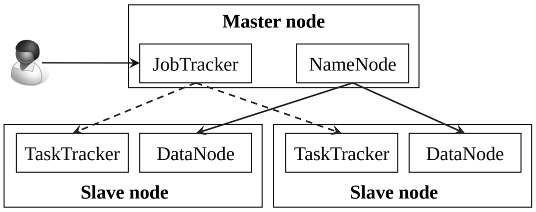


Fig. 4: Structure of a Hadoop cluster with one master node and two slave nodes.

on files, and `create` and `delete` operations on directories. In HDFS, data is divided into small splits, called *blocks*, (64MB and 128MB are most commonly used sizes). Each block is independently replicated to multiple DataNodes, and block replicas are processed by mappers and reducers. More details about Hadoop and HDFS may be found in Chapter 2 of [70].

HDFS Federation. In the standard HDFS, there is only one NameNode, which is a single point of failure. HDFS Federation [27] overcomes this limitation of HDFS by adding multiple NameNodes that are independent and do not require coordination with each other. DataNodes store data, and in addition, each DataNode registers with all the NameNodes. In this manner, HDFS Federation creates a large virtual cluster that increases performance, turns NameNode to be fault-tolerant, and provides multiple isolated jobs' execution framework.

YARN Architecture. YARN [71] is the latest version of Hadoop-2.7.1 and partitions the two major functionalities of the JobTracker of the previous Hadoop, *i.e.*, resource management and job scheduling monitoring, into separate daemons, called a global ResourceManager daemon and a per-application ApplicationMaster daemon. Details about YARN may be found in [72].

The ResourceManager is responsible for dividing the cluster's resources among all the applications running in the system. The ApplicationMaster is an application-specific entity, negotiates resources from the ResourceManager. The NodeManager is a per-node daemon, which is responsible for launching the application's containers, monitoring their resource usage (CPU, memory etc.), and reporting back to the ResourceManager. A container represents a collection of physical resources.

1.4 Spark

Apache Spark is a cluster computing platform that extends MapReduce-style processing for efficiently supporting more types of fast and real-time computations, interactive queries, and stream processing. The major difference between Spark and Hadoop lies in the processing style, where MapReduce stores outputs of each iteration in the disk while Spark stores data in the main memory, and hence, supports fast processing. Spark also supports Hadoop, and hence, it can access any Hadoop data sources. Spark Core contains task scheduling, memory management, fault recovery, interacting with storage systems, and defines *resilient distributed datasets* (RDDs). RDDs are main programming abstraction and represent a collection of distributed items across many computing nodes that can execute a computation. Spark supports several programming languages such as Python, Java, and Scala. Details about Spark may be found in [73].

Matrix computations [74], machine learning [75], graph processing [76], iterative queries [73], and stream processing [77] are a few popular examples of computational fields where Spark is commonly used. Apache Flink [78], Apache Ignite [79], Apache Storm [80], and Twitter Heron [81] are other stream processing frameworks.

2 Motivations and Needs for Geo-Distributed Big-Data Processing using Hadoop or Spark

We list four major motivational points behind the design of a geo-distributed big-data processing framework, as follows:

Support for geo-distributed applications. As we mentioned in §1, a lot of applications generate data at geo-distributed locations or DCs. On the one hand, genomic and biological data, activity, session and server logs, and performance counters are expanding geographically much faster than inter-DC bandwidth; hence, such entire datasets cannot be efficiently transferred to a single location [2], [10], [82], [83]. On the other hand, analysis and manipulation operations do not require an entire dataset from each location in providing the final output. Thus, there is a need of on-site big-data processing frameworks that can send only the desired inputs (after processing data at each site) to a single location for providing the final outputs under legal constraints of an organization.

Enrich datasets and high performance. Currently, a data-intensive application produces, manipulates, or analyzes data of size Petabytes or more. Sharing such a huge amount of data across the globe enriches datasets and helps several communities in finding recent trends, new types of laws, regulations, and networking constraints [83]. In contrast, the data processing using MapReduce is dominated by the communication between the map phase and the reduce phase, where several replicas of an identical data are transferred to the reduce phase for obtaining final outputs [30], [65], [67]. However, sometimes none of the replicated partial outputs provides the final outputs.

For example, we want to execute a top-k query on n locations that have their own data. In this case, a trivial solution is to send the whole data from all the n locations to a single location. In contrast, we can execute a top-k query on each n location and send only top-k results from each n locations to a single location that can find the final top-k answers. One more example, in case of equijoin of two relations, say $X(A, B)$ and $Y(B, C)$, where X and Y are located in different sites, if there is no tuple containing a joining value, say b_1, in the relation Y, then it is worthless to transfer all the tuples of the relation X having b_1 to the location of the relation Y.

Therefore, it is clear and trivial that one can solve geo-applications by moving an entire dataset to a single location; however, it will increase the network load, job completion time, space requirement at the single site, and decrease the performance of the system. Thus, the development of efficient geo-distributed Hadoop or Spark-based frameworks transparent to users is needed [10].

Providing geo-security and geo-privacy mechanisms. The development of a geo-distributed framework, inherently, requires a secure and privacy-preserving mechanism for data transfer and computation execution. The design of geo-security and geo-privacy mechanisms will help in geo-frameworks, also in a single cloud computing platform, where data and computation locations are identical.

Here, we explain why do the current security and privacy mechanisms fail in a geo-distributed framework. Data may be classified as public, sensitive, or confidential with special handling [84]. Executing a geo-application on public data of an organization may breach the security and privacy of sensitive or confidential data, since a geo-application may attempt to scan the entire dataset. For example, executing a geo-application on a health data of a country allows data movement within the country; however, the same data may not be allowed to be accessed by a geo-application that is moving the data outside the country. Another example of privacy breaking that can occur when the public data of the geo-locations of a person is associated with the disease data which is sensitive [84]. In a similar manner, the data confidentiality is vulnerable in a geo-computation.

Thus, the security and privacy in a geo-computation depend on several factors, *e.g.*, organizations that are asking for data, the organizations' location, and the scope of the computation. In addition, if on-site data processing is allowed, an organization wishes to ensure the security and privacy of their output data according to their policies, during the transmission and computations at other sites, resulting in no malicious activities on the data [2]. Therefore, the design of geo-secure and geo-privacy mechanisms is required, thus, maintaining data security and privacy during a geo-distributed computation. However, creating a secure and privacy-preserving geo-distributed framework raises several challenges, which we discuss in the next section.

Handling DC failure and natural disasters. The classical Hadoop was designed to prevent a job failure due to disk, node, or rack failure by replicating the data along with the job to multiple nodes and racks within a DC. A single DC failure/outage is not very common; however, if it does happen, then it leads to severe obstacles. For example, in the month of May 2017, due to the power outage, British Airways DC was crashed, and that leads to catastrophic impacts.[2] In order to handle DC failure, replication of the data with the jobs to different DCs (possibly outside the country) is expected to be a trivial solution. However, such a geo-replication requires us to redesign new Hadoop/Spark-based systems that can work over different DCs transparent to any failure. We will study some frameworks supporting geo-replication in §4.2.

3 Challenges in the Development of Geo-Distributed Hadoop or Spark based Frameworks

The existence of big-data, on one hand, requires the design of a fault-tolerant and computation efficient framework, and Hadoop, Spark, or similar frameworks satisfy these requirements. On the other hand, globally distributed big-databases — as opposed to traditional parallel databases, cluster computations, and file-sharing within a DC — introduce new research challenges in different domains, as follows: (*i*) the database domain has new challenges such as query planning, data locality, replication, query execution, cost estimation, and the final output generation; (*ii*) the wide area network domain has new challenges such as bandwidth constraints and data movement [36]. In addition, geo-distributed data processing using the current frameworks inherits some old challenges such as location transparency, (*i.e.*, a user will receive a correct output regardless of the data location), and local autonomy, (*i.e.*, the capability to administer a local database and to operate independently when connections to other nodes have failed) [11], [36].

In this section, we describe new challenges in the context of geo-distributed big-data processing using Hadoop or Spark-based systems. After each challenge, we give references to solutions to the challenge, which are described later in this paper.

A universal code for all sites and compatibility issues. In the present time, several big-data processing frameworks, languages, and mechanisms are proposed; for example, Hadoop, Yarn, Spark, Hive [85], [86], Pig Latin [87], Dryad, Spark SQL [88], etc. In addition, different big-data and metadata storages, like HDFS, Gfarm file system [89], GridDB [90], MongoDB [91], HBase [92], etc., are available. These databases have non-identical data formats, APIs, storage policies, privacy concerns for storing and retrieving data, network dynamics, and access control [93], [94], [95].

Moreover, different sites may have different types of regulations for exporting data, operating systems, availability of data, services, security checks, resources, cost, and software implementations. Sometimes, simultaneous usages of multiple frameworks improve utilization and allow applications to share access to large datasets [96]. However, the existence of different frameworks at different locations poses additional challenges such as different scheduling needs, programming models, communication patterns, task dependencies, data placement, and different APIs.

Hence, according to a client perspective, it is not desirable to write code for different frameworks and different data formats. For example, if there are two sites with HDFS and Gfarm file system, then the data retrieval code is not identical and the user has to write two different codes for retrieving data. In this scenario, it is required that a universal code will work at all the locations without modifying their data format and processing frameworks. It may require an interpreter that converts a user code according to the requirement of different frameworks. However, the use of an interpreter puts some additional challenges such as how does a system follow the inherent properties, *e.g.*, massive parallelism and fault-tolerance. It should be noted that user-defined compatibility tasks may slow down the overall system performance [4], [82]. Unfortunately, there is not a single geo-distributed system that can solve this challenge, to the best of our knowledge.

Solutions: Mesos [96] provides a solution to the above-mentioned requirements to some extents. Twitter's Summingbird [97] integrates batch and stream processing within a single DC. Recently, BigDAWG [98] and Rheem [99] are two new systems that are focusing on compatibility issues

2. https://www.theguardian.com/business/2017/may/30/british-airways-it-failure-experts-doubt-power-surge-claim

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

in a single DC. In short, BigDAWG [98] and Rheem [99] are trying to achieve platform-independent processing, multi-platform task execution, exploit complete processing capabilities of underlying systems, and data processing abstraction (in a single DC). Since Mesos [96], Summingbird [97], BigDAWG [98], and Rheem [99] deals with processing in a single DC, we do not study these systems in this survey.

Awan [1] (§6.3) is a system that allocates resources to different geo-distributed frameworks. However, Awan does not provide universal compatibilities to the existing systems. Also, during our investigation, we did not find any system that can handle above-mentioned compatibility issues in the geo-distributed settings.

Secure and privacy-preserving computations and data movement. Geo-distributed applications are increasing day-by-day, resulting in an increasing number of challenges in maintaining fine and coarse grain security and privacy of data or computations. The classical MapReduce does not support the security and privacy of data or computations within a single public cloud. But even if the security and privacy in a single cloud is preserved, it is still a major challenge in ensuring the security and privacy of data or computations in geo-distributed big-data processing. A survey on security and privacy in the standard MapReduce may be found in [100].

The security analysis also requires risk management. Applying complex security mechanisms on a geo-computation without considering risk management may harm the computation and system performance, while it is not required to implement rigorous security systems. Hence, there is a need to consider risk when designing security and privacy mechanisms for geo-computations [101], [102]. The privacy of an identical type of data (*e.g.*, health-care data) may not be treated the same when implemented in different countries. Hence, there are several issues to be addressed while designing a secure geo-distributed framework, as follows: how to trust the data received from a site, how to ensure that the data is be transferred in a secure and private manner, how to build trust among sites, how to execute computations in each site in a privacy-preserving manner, how to pre-inspect programs utilizing the data, how to ensure usage of the data, and how to allow a computation execution while maintaining fine-grained security features such as authentication, authorization, and access control [103].

Solutions: G-Hadoop [16] (§4.1.1) provides an authentication mechanism, and ViNe [103] (§5) provides an end-to-end data security. However, currently, we are not aware of any complete solution for security and privacy in the context of geo-distributed Hadoop/Spark jobs.

Data locality and placement. In the context of geo-data processing, data locality refers to data processing at the same site or nearby sites where the data is located [12], [102]. However, the current Hadoop/Spark/SQL-style based geo-distributed data processing systems are designed on the principle of data pulling from all the locations to a single location, and hence, they do not regard data locality [36]. In addition, due to a huge amount of raw data generated at different sites, it is challenging to send the whole dataset to a single location; hence, the design and development of systems that take the data locality into account are crucial for optimizing the system performance.

In contrast, sometimes a framework regarding the data locality does not work well in terms of performance and cost [32], [104], [105] due to the limited number of resources or slow inter-DC connections. Hence, it may be required to access/process data in nearby DCs, which may be faster than the local access. Thus, we find a challenge in designing a system for accessing local or remote (nearby) data, leading to optimized job performance.

Solutions: G-Hadoop [16] (§4.1.1), GMR [24] (§4.1.1), Nebula [106] (§4.1.1), Iridium [10] (§4.1.2), and JetStream [35] (§4.1.2).

Finding only relevant intermediate data. A system built on the "data locality" principle processes data at their sites and provides intermediate data. However, sometimes, the complete intermediate data at a site do not participate in the final output, and hence, it necessitates to find only relevant intermediate data. We emphasize that the concepts of the data locality and relevant intermediate data finding are different.

A computation can be characterized by two parameters: (*i*) the amount of input data (in bits) at a data source, and (*ii*) the expansion factor, e, which shows a ratio of the output size to the input size [10], [11], [13], [14]. Based on the expansion factor, a computation can be of three types, as follows: (*i*) $e \gg 1$: the output size is much larger than the input size, *e.g.*, join of relations; (*ii*) $e = 1$: the output size is of the same size as the input size, *e.g.* outputs of a sorting algorithm; (*iii*) $e \ll 1$: the output size is less than the input size, *e.g.*, word count. When dealing with a geo-distributed computation, it is not efficient to move all the data when $e \gg 1$ and only some parts of that data participate in the final outputs [30], [82]. For example, in the first, second, and third types of computations, if the computation performs a joining of relations while most of the tuples of a relation do not join with any other relations at the other sites, a global sorting only on selected intermediate sorted outputs, and a frequency-count of some words, respectively, then we need to find only relevant intermediate data.

These challenges motivate us to find only relevant intermediate data at different locations before obtaining the final output. In addition, it is also required to prioritize data considering dynamic requirements, resources, and usefulness of data before moving data [12].

Solutions: Iridium [10] (§4.1.2), Geode [36] (using difference finding, §4.1.3), and Meta-MapReduce [30] (§6.1).

Remote access and variable bandwidth. The cost of a geo-distributed data processing is dependent on remote accesses and the network bandwidth, and as the amount of inter-DC data movement increases, the job cost also increases [102], [107]. In addition, we may connect DCs with a low-latency and high-bandwidth interconnects for fine-grain data exchanges and that results in a very high cost. Hence, an intelligent remote data access mechanism is required for fetching only the desired data. Moreover, frequently accessed data during the execution of similar types of jobs can be placed in some specific DCs to reduce the communication [2].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

The limited bandwidth constraint, thus, motivates to design efficient distributed and reliable systems/algorithms for collecting and moving data among DCs while minimizing remote access, resulting in lower job cost and latency [14], [108]. In addition, the system must adjust the network bandwidth dynamically; hence, the system can drop some data without affecting data quality significantly [35]. In other words, there is a need of an algorithm that will know the global view of the system (consisting of the bandwidth, data at each DC, and distance to other DCs).

Serving data transfer as best-effort or realtime is also a challenge in geo-computations. On the one hand, if one serves best-effort data transfer, then the final output will be delayed, and it requires a significant amount of the network bandwidth at a time. On the other hand, if we transfer data in real-time, then the user will get real-time results at the cost of endless use of the network bandwidth [109].

Solutions: Iridium [10] (§4.1.2), Pixida [5] (§6.2), Lazy optimal algorithm [110] (§6.2), JetStream [35] (§4.1.2), Volley [111] (§6.1), Rout [82] (§6.1), and Meta-MapReduce [30] (§6.1).

Task assignment and job completion time. The job completion time of a geo-distributed computation is dependent on several factors, as follows: (*i*) data locality, (*ii*) the amount of intermediate data, (*iii*) selection of a DC for the final task — it is required to assign the final task to a DC that has a major portion of data that participate in the final outputs, resulting in fast job completion and reduced data transfer time [8], [25], and (*iv*) the inter-DC and intra-DC bandwidth. The authors [112] showed that an execution of a MapReduce job on a network-aware vs. network-unaware scheme significantly impacts the job completion time.

The challenge comes in finding a straggler process. In a locally distributed computation, we can find straggler processes and perform a speculative execution for fast job completion time. However, these strategies do not help in a geo-distributed data processing, because of different amount of data in DCs and different bandwidth [12].

In addition, the problem of straggler processes cannot be removed by applying offline task optimization placement algorithms, since they rely on a priori knowledge of task execution time and inter-DC transfer time, which both are unknown in geo-distributed data processing [102].

Solutions: Iridium [10], Joint optimization of task assignment, data placement, and routing in geo-distributed DCs [113], Reseal [114] (§6.1), Tudoran et al. [8] (§5) and Gadre et al. [115] (§5), and Flutter [102] (§6.2).

Iterative and SQL queries. The standard MapReduce was not developed for supporting iterative and a wide range of SQL queries. Later, Hive, Pig Latin, and Spark SQL were developed for supporting SQL-style queries. However, all these languages are designed for processing in-home/local data. Since relational algebra is a basis of several different operations, it is required to develop a geo-distributed query language regarding data locality and the network bandwidth. The new type of query language must also deal with some additional challenges such as geo-distributed query optimization, geo-distributed query execution plan, geo-distributed indexing, and geo-distributed caching. The problem of joining of multiple tables that are located at different locations is also not trivial. In this case, moving an entire table from one location to the location of the other table is naive yet cumbersome, because of network bandwidth, time, and cost. Hence, we see the joining operation in geo-distributed settings is a major challenge. The joining operation gets more complicated in the case of streaming of tables where a window-based join does not work [116] because the joining values of multiple tables may not *synchronously* arrive at an identical time window, thereby leading to missing outputs.

Processing iterative queries on the classical Hadoop was a cumbersome task due to disk-based storage after each iteration (as we mentioned the difference between Hadoop and Spark in §1.4). However, Spark can efficiently process iterative queries due to *in-memory* processing. Processing iterative queries in a geo-computation requires us to find solutions to store intermediate results in the context of an iterative query.

Solutions: Geode [36] (§4.1.3) provides a solution to execute geo-distributed SQL queries. Google's F1 [117] and Spanner [37] (§4.2.3) are two SQL processing systems. There are some other systems [118], [119] for machine learning based on iterative Hadoop/Spark processing. However, in this paper, we are not covering any paper regarding machine learning using Hadoop/Spark.

Scalability and fault-tolerance. Hadoop, Yarn, Spark, and similar big-data processing frameworks are scalable and fault-tolerant as compared to parallel computing, cluster computing, and distributed databases. Because of these two features, several organizations and researchers use these systems daily for big-data processing. Hence, it is an inherent challenge to design a new fault-tolerant geo-distributed framework so that the failure of the whole/partial DC does not lead to the failure of other DCs and also scalable in terms of adding or removing different DCs, computing nodes, resources, and software implementations [29], [82], [120].

Solutions: Medusa [121] (§4.1.1), Resilin [34] (§4.2), HOG [29] (§4.2), and KOALA-grid-based system [33] (§4.2).

The above-mentioned issues will naturally impact on the design and division of functionality of different components of a framework, which are located at non-identical locations.

4 Architectures for Geo-Distributed Big-Data Processing

In this section, we review several geo-distributed big-data processing frameworks and algorithms under two categories, as follows:

Pre-located geo-distributed big-data. This category deals with data that is *already* geo-distributed before the computation. For example, if there are n locations, then all the n locations have their data.

User-located geo-distributed big-data. This category deals with frameworks that *explicitly* distribute data to geo-locations before the computation begins. For example, if there are n locations, then the user distributes data to the n locations.

Note that there is a clear distinction between the above-mentioned two categories, as follows: The first category requires the distribution of jobs (*not data*) over different clouds by the user site and then aggregation of outputs of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

Frameworks/Protocols	Data distribution	Data processing	Security and privacy	Secure data movement	Optimized paths among DCs	Resource management	Data locality	Relevant intermediate data finding	Bandwidth consideration	Scalability	SQL-support	Result caching
Geo-distributed batch processing MapReduce-based systems for pre-located geo-distributed data (Section 4.1.1)												
G-Hadoop [16]	P & U	✓	$✓^p$				✓	✓				
G-MR [24]	P	✓			✓	✓	✓	✓				
Nebula [106]	P	✓					✓	✓				
Medusa [121]	P & U	✓							✓			
Geo-distributed stream processing frameworks for pre-located geo-distributed data (Section 4.1.2)												
Iridium [10]	P	✓			✓		✓	✓	✓			
JetStream [35]	P						✓		✓			
SAGE [9]	P	✓				✓						
SQL-style processing framework for pre-located geo-distributed data (Section 4.1.3)												
Geode [36]	P	✓					✓	✓	✓		✓	✓
Geo-distributed batch processing MapReduce-based systems for user-located geo-distributed data (Section 4.2.1)												
HMR [31]	U	✓										
Resilin [34]	U	✓				✓				✓		
SEMROD [122]	U	✓	✓				✓			✓		
HOG [29]	U	✓				✓				✓		
KOALA grid-based system [33]	U	✓				✓				✓		
HybridMR [15]	U	✓								✓		
Geo-distributed stream processing frameworks for user-located geo-distributed data (Section 4.2.2)												
Photon [116]	P & U	✓				✓	✓	✓		✓		
SQL-style processing framework for user-located geo-distributed data (Section 4.2.3)												
Spanner [37]	P & U	✓				✓	✓	✓		✓	✓	
Data transfer systems/algorithms (Section 5)												
Tudoran et al. [8], Gadre et al. [115]	P & U					✓						
Volley [111]	U					✓			✓			
Scheduling for geo-distributed MapReduce-based systems (Section 6.1)												
WANalytics [2]	P & U				✓	✓						✓
Shuffle-aware data pushing [14]	U					✓						
ViNe [103]	P & U		$✓^p$	✓								
Reseal [114]	P											
Rout [82]	P & U				✓	✓			✓			
Meta-MapReduce [30]	P & U					✓		✓	✓			
Zhang el al. [11]	P					✓						
Scheduling for geo-distributed Spark-based systems (Section 6.2)												
Pixida [5]	P				✓				✓			
Flutter [102]	P				✓	✓						
Lazy optimal algorithm [110]	P					✓			✓			
Resource allocation mechanisms for geo-distributed systems (Section 6.3)												
Awan [1]	P					✓						
Gadre et al. [25]	P & U				✓	✓						
Ghit et al. [33]	P & U					✓						
Notations. P: Pre-located geo-distributed big-data. U: User-located geo-distributed big-data. $✓^p$: Systems provide only partial security, not a complete secure and private solution (*e.g.*, G-Hadoop and ViNE allow authentication and end-to-end security, respectively, while SEMROD allows sensitive data security in the context of a hybrid cloud).												

TABLE 1: Summary of geo-distributed big-data processing frameworks and algorithms.

all the sites at a specified site. The second category requires the distribution and/or partitioning of *both* the data as well as jobs over different clouds by the user site. Here, an aggregation of outputs of all the sites is not must and depends on the job, if the job is partitioning the data over the clouds.

In the first category, we see MapReduce-based frameworks (*e.g.*, G-Hadoop, GMR, Nebula, Medusa), Spark-based system (*e.g.*, Iridium), a system for processing SQL-queries. As we mentioned, all these systems require to distribute a job over multiple clouds and then aggregation of outputs. In the second category, we see frameworks that do user-defined data and computation partitioning for achieving (*i*) a higher level of fault-tolerance (by executing a job on multiple clouds, *e.g.*, HMR, Spanner, and F1), (*ii*) a secure computation by using public and private clouds (*e.g.*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

SEMROD), and (*iii*) the lower job cost by accessing grid-resources in an opportunistic manner (*e.g.*, HOG, KOALA grid-based system, and HybridMR).

A comparison of frameworks and algorithms for geo-distributed big-data processing based on several parameters such as security and privacy of data, data locality, selection of an optimal path for data transfer, and resource management is given in Table 1.

4.1 Frameworks for Pre-Located Geo-Distributed Big-Data

4.1.1 Geo-distributed batch processing MapReduce-based systems for pre-located geo-distributed data

G-Hadoop. Wang et al. provided G-Hadoop [16] framework for processing geo-distributed data across multiple cluster nodes, without changing existing cluster architectures. On the one hand, G-Hadoop processes data stored in a geo-distributed file system, known as Gfarm file system. On the other hand, G-Hadoop may increase fault-tolerance by executing an identical task in multiple clusters.

G-Hadoop consists of a G-Hadoop master node at a central location (for accessing G-Hadoop framework) and G-Hadoop slave nodes (for executing MapReduce jobs). The G-Hadoop master node accepts jobs from users, splits jobs into several sub-jobs and distributes them across slave nodes, and manages metadata of all files in the system. The G-Hadoop master node contains a metadata server and a global JobTracker, which is a modified version of Hadoop's original JobTracker. A G-Hadoop slave node contains a TaskTracker, a local Job Tracker, and an I/O server.

The Gfarm file system is a master-slave based distributed file system designed to share a vast amount of data among globally distributed clusters connected via a wide-area network. The master node called a Metadata Server (MDS) is responsible for managing the file system's metadata such as file names, locations, and access credentials. The MDS is also responsible for coordinating access to the files stored in the cluster. The multiple slave nodes, referred to as Data Nodes (DN), are responsible for storing raw data on local hard disks using local file systems. A DN runs a daemon that coordinates the access to the files on the local file system.

Job execution in G-Hadoop. Now, we discuss the job flow in G-Hadoop, which will help readers to understand a job execution in the geo-distributed environment. The job flow consists of three steps, as follows:

1) *Job submission and initialization*. The user submits a job to the G-Hadoop master node that creates a unique ID for the new job, and then, the user copies the map and reduce functions, job configuration files, and input files to a designated working directory at the master node of Gfarm file system. The global JobTracker initializes and splits the job.
2) *Sub-job assignment*. TaskTrackers of the G-Hadoop slaves request the global JobTracker for new tasks, periodically. The task assignment problem also considers the data locations. When a TaskTracker receives tasks, it copies executables and resources from the working directory of Gfarm file system.
3) *Sub-job execution*. Now, a computing node executes an assigned MapReduce task, as follows: (*i*) a map task: it processes input data and writes outputs to a shared directory in the cluster; (*ii*) reduce task: it contacts TaskTrackers that have executed the corresponding map tasks and fetches their outputs. If the TaskTracker is located in an identical cluster where data and reduce tasks are assigned, the data is read from the common shared directory of the cluster. Otherwise, the data is fetched using an HTTP request. The results of a reduce task are written to Gfarm file system.

Pros. G-Hadoop provides an efficient geo-distributed data processing, regards data locality, and hence, performs the map phase at the local site. G-Hadoop has a security mechanism, thereby an authenticated user can get access to only authorized data.

Cons. G-Hadoop randomly places reducers in involved DCs [123]. Also, it does not support iterative queries and HDFS, which is a common data storage, instead keeps the data in a new type of file system, Gfarm file system.

G-MR. G-MR [24], see Fig. 5, is a Hadoop-based framework that executes MapReduce jobs on a geo-distributed dataset across multiple DCs. Unlike G-Hadoop [16], G-MR does not place reducers randomly and uses a single directional weighted graph for data movement using the shortest path algorithm. G-MR deploys a GroupManager at a single DC and a JobManager at each DC. The GroupManager distributes the code of mappers and reducers to all the DCs and executes a data transformation graph (DTG) algorithm. Each JobManager manages and executes assigned local MapReduce jobs using a Hadoop cluster. Each JobManager has two components, namely a CopyManager for copying outputs of the job of one DC to other DCs and an AggregationManager for aggregating results from DCs.

<table>
<tr><td colspan="3">GroupManager</td></tr>
<tr><td colspan="3">JobManager</td></tr>
<tr><td>Hadoop</td><td>CopyManager</td><td>AggregationManager</td></tr>
<tr><td colspan="3">Geo-distributed databases at datacenters</td></tr>
</table>

Fig. 5: G-MR.

The DTG algorithm finds an optimized path based on characteristics of the dataset, MapReduce jobs, and the DC infrastructure, for executing MapReduce jobs. The DTG algorithm constructs a graph by taking all the possible execution paths for executing the job. A node of the graph shows the number of MapReduce phases that have applied to input data and the data locations, and a weighted edge shows the computation flow. After constructing the graph, the problem of finding an optimized path for executing the job is reduced in finding a minimum weight path, which can be solved using the shortest path algorithm for the graph.

Execution steps. A user submits G-MR codes to one of the DCs that executes the GroupManager. The GroupManager executes the DTG algorithm and determines the best path for collecting outputs from all the DCs. The GroupManager informs a JobManager of a DC about (*i*) MapReduce jobs, (*ii*) the local data that should be accessed by the job, and (*iii*) where to copy the outputs of the job. The JobManagers execute the job accordingly using Hadoop, and then, use their local AggregationManager and CopyManager components for executing the respective tasks. Eventually, the GroupManager holds outputs of all the remaining DCs and performs the final computation to provide the final output.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

Pros. G-MR is a fully-functional and geo-distributed Hadoop-based framework.
Cons. GMR is a non-secure framework and can only be used when the data is associative, *i.e.*, the iterative and hierarchical reduce will not change the final result [123]. Also, G-MR, like G-Hadoop, do not handle arbitrary and malicious faults, and cloud outages. These systems are unable to handle crash faults, similar to the standard MapReduce [121].

Nebula. Nebula [106] is a system that selects the *best node* for minimizing overall job completion time. Nebula consists of four centralized components, as follows: Nebula central, compute pool master, data-store master, and Nebula monitor. The Nebula central accepts jobs from the user who also provides the location of geo-distributed input data to the data-store master. The cmpute nodes, which are geo-distributed, periodically contact with the compute pool master, which is aware of all computing nodes in the system, and ask for jobs. A scheduler assigns tasks to the computing nodes based on the scheduling policy with the help of the compute pool master. Then, the computing nodes download the tasks and the input data from the data nodes according to specified locations by the data store master. When the computation finishes, the output is uploaded to data nodes, and the data-store master is informed of the location of the final outputs.

Medusa. Medusa [121] system handles three new types of faults: processing corruption that leads to wrong outputs, malicious attacks, and cloud outages that may lead to the unavailability of MapReduce instances and their data. In order to handle such faults, a job is executed on $2f + 1$ clouds, where f faults are tolerable. In addition, a cloud is selected based on parameters such as available resources and bandwidth so that the job completion time is decreased.
Pros. Medusa handles new types of faults.
Cons. Except handling new types of faults, Medusa does not provide any new concept, and the fault handling systems can be included in a system that considers resource allocation and WAN traffic movement. The authors claim that they are not modifying the standard Hadoop; however, this claim is doubtful in the case of obtaining final outputs. The standard Hadoop system cannot produce the final outputs from partial outputs, *e.g.*, equijoin of relations or finding maximum salaries of a person working in more than one department.

4.1.2 Geo-distributed stream processing frameworks for pre-located geo-distributed data

Iridium. Iridium [10] is designed on the top of Apache Spark and consists of two managers, as follows: (*i*) a *global manager* is located in only one site for coordinating the query execution across sites, keeping track of data locations, and maintaining durability and consistency of data; and (*ii*) a *local manager* is located at each site for controlling local resources, periodically updating the global manager, and executing assigned jobs. Iridium considers heterogeneous bandwidths among different sites and optimizes data and task placement, which results in the minimal data transfer time among the sites. The task placement problem is described as a linear program that considers site bandwidths and query characteristics. An iterative greedy heuristic is used to move small chunks of datasets to sites having more bandwidth, resulting in efficient data transfer, without affecting the job completion time. Iridium speeds up processing by 64% to 92% as compared to Conviva [124], Bing Edge, TPC-DS [125] and Berkeley Big Data Benchmark [126], when deployed across eight Amazon Elastic Compute Cloud (EC2) regions in five continents. Iridium saves WAN bandwidth usage by 15% to 64%.
Pros. While minimizing the job completion time, Iridium considers data and task placement regarding different bandwidth among DCs.
Cons. Iridium considers the network congestion within a DC only, not among DCs. Also, Iridium minimizes only latency and does not consider the network bandwidth optimally [5], [102].

The following frameworks, which are not based on Spark, are also designed for geo-distributed stream data processing where the data already exist at multiple locations.

JetStream. JetStream [35] system processes geo-distributed streams and regards the network bandwidth and data quality. JetStream has three main components, as follows: geo-distributed workers, a centralized coordinator, and a client. The data is stored in a structured database of the form of a datacube. A client program creates a dataflow graph and submits it for the execution to the centralized coordinator. The coordinator selects linked-dataflow operators for each worker and then sends a relevant subset of the graph to each worker. Then, a worker creates necessary network connections with other workers and starts the operators. The execution terminates when the centralized coordinator sends a stop message or all the sources send a stop marker indicating that there will be no more data.
Pros. JetStream, like Iridium, minimizes the amount of inter-DC traffic, but the approach is different from Iridium. JetStream uses data aggregation and adaptive filtering that support efficient OLAP queries as compared to Iridium.
Cons. JetStream provides some degree of inaccuracy in the final results because of dropping and sampling results, hence, it is also good for small sensor networks. Unlike Iridium, JetStream does not support arbitrary SQL queries and does not optimize data and task placement [10]. However, both, Iridium and JetStream do not deal with the network traffic and user-perceived delay simultaneously [110], [127].

SAGE. SAGE [9] is a *general-purpose* cloud-based architecture for processing geo-distributed *stream* data. SAGE consists of two types of services, as follows: (*i*) *Processing services* process incoming streaming data by applying the users' processing functions and provide outputs. Several queues at each geo-location handle stream data, where each processing service has one or more incoming queues. In addition, data is transformed into the required system format; and extract, transform, and load (ETL) software, *e.g.*, IBM's InfoSphere DataStage [128], are used for transforming data into the required format. (*ii*) A *global aggregator service* computes the final result by aggregating the outputs of the processing services. This process is executed in a DC nearby the user-location.
Pros. Sage is independent of a data format, unlike JetStream, and also performs aggregation operation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

Cons. Sage is designed to work with a limited number of DCs. The above-mentioned three stream processing frameworks perform data analytics over multiple geo-distributed sites, and the final computation is carried out at a single site. In the context of a large-scale IoT system, many sensors are widely distributed and send their expected results very often, *e.g.*, location tracking systems. However, the current stream processing systems are not capable of handling streaming from such a huge number of devices [129], [130].

G-cut. G-cut [107] proposed a way for allocating tasks in stream processing systems, specifically, for graph processing. Unlike Iridium that focuses on a general problem on a particular implementation (Spark), G-cut focuses on graph partitioning over multiple-DCs while minimizing inter-DC bandwidth usages and achieving user-defined WAN usages constraints. The algorithm consists of two phases: in the first phase, a stream graph processing algorithm does graph partitioning while satisfying the criteria of minimum inter-DC traffic and regarding heterogeneous bandwidth among DCs, and the second phase is used to refine the graph partitioning obtained in the first phase.

4.1.3 SQL-style processing framework for pre-located geo-distributed data

Geode. Geode [36] consists of three centralized components, as follows: a central command layer, pseudo-distributed measurement of data transfer, and a workload optimizer. The main component of Geode is the central command layer that receives SQL analytical queries from the user, partitions queries to create a distributed query execution plan, executes this plan over involving DCs, coordinates data transfers between DCs, and collates the final output. At each DC, the command layer interacts with a thin proxy layer that facilitates data transfers between DCs and manages a local cache of intermediate query results used for data transfer optimization. The workload optimizer estimates the current query plan or the data replication strategy against periodically obtained measurements from the command layer. These measurements are collected using the pseudo-distributed execution technique. Geode is built on top of Hive and uses less bandwidth than centralized analytics in a Microsoft production workload, TPC-CH [131], and Berkeley Big Data Benchmark [126].

Pros. Geode performs analytical queries locally at the data site. Also, Geode provides a caching mechanism for storing intermediate results and computing differences for avoiding redundant transfers. The caching mechanism reduces the data transfer for the given queries by 3.5 times.

Cons. Geode does not focus on the job completion time and iterative machine learning workflows.

4.2 Frameworks for User-Located Geo-Distributed Big-Data

In many cases, a single cluster is not able to process an entire dataset, and hence, the input data is partitioned over several clusters (possibly at different locations), having different configurations. In addition, geo-replication becomes necessary for achieving a higher level of fault-tolerance, because services of a DC may be disrupted for a while [32], [132]. In this section, we review frameworks that distribute data to geo-distributed clusters of different configurations, and hence, a user can select machines based on CPU speed, memory size, network bandwidth, and disk I/O speed from different locations. Geo-replication also ensures that a single DC will not be overloaded [111], [120]. A system that distributes data to several locations must address the following questions at the time of design:

1) How to store the input data, the intermediate data, and the final results?
2) How to address shared data, data inter-dependencies, and application issues [15], [111]?
3) How to schedule a task and where to place data? Answers to these questions impact job completion time and the cost significantly.
4) How to deal with task failures caused by using different clouds of non-identical configurations?

In addition, these systems must address inherent questions, *i.e.*, how to efficiently aggregate the outputs of all the locations, how to deal with variable network bandwidth, and how to achieve strong consistency? Further details about geo-replication may be found in [133].

4.2.1 Geo-distributed batch processing MapReduce-based systems for user-located geo-distributed data

HMR. Hierarchical MapReduce (HMR) [31] is a two-layered programming model, where the top layer is the global controller layer and the bottom layer consists of multiple clusters that execute a MapReduce job; see Fig. 6. A MapReduce job and data are submitted to the global controller, and the job is executed by the clusters of the bottom layer. Specifically, the global controller layer has three components, as follows: (*i*) a job scheduler: partitions a MapReduce job and data into several sub-jobs and subsets of the dataset, respectively, and assigns each sub-job and a data subset to a local cluster; (*ii*) a data manager: transfers map and reduce functions, job configuration files, and a data subset to local clusters; and (*iii*) a workload controller: does load balancing. In the bottom layer, a job manager in a local cluster executes an HMR daemon and a local MapReduce sub-job. When the local sub-job is finished in a local cluster, the local cluster moves the final outputs to one of the local clusters that executes a global reducer for providing the final output.

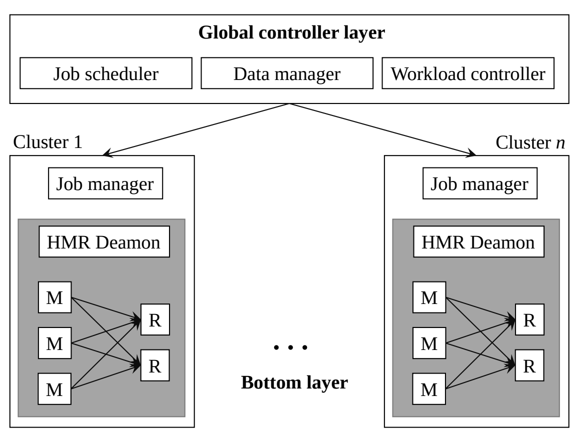


Fig. 6: Hierarchical MapReduce programming model.

Pros. HMR is a trivial framework for geo-distributed MapReduce map-intensive jobs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

Cons. HMR requires a full MapReduce job using the identity mapper to be executed before the global reducer. HMR is not efficient if intermediate data at different sites is huge and needs to be transferred to the global reducer, which is at a single site, resulting in the network bottleneck. In HMR, we also need to explicitly install a daemon on each one of the DCs.

A simple extension to HMR is proposed in [134], where the authors suggested to consider the amount of data to be moved and the resources required to produce the final output at the global reducer. However, like HMR, this extension does not consider heterogeneous inter-DC bandwidth and available resources at the clusters. Another extension to both the systems is provided in [135], where the authors included clusters' resources and different network link capacity into consideration.

Resilin. Resilin [34] provides a hybrid cloud-based MapReduce computation framework. Resilin; see Fig. 7, implements Amazon Elastic MapReduce (EMR) [136] interface and uses the existing Amazon EMR tools for interacting with the system. In particular, Resilin allows a user to process data stored in a cloud with the help of other clouds' resources. In other words, Resilin partitions data as per the number of available clouds and moves data to those sites, which perform the computation and send the partial outputs to the source site. Resilin implements four services, as follows: (*i*) a *provision service* for starting or stopping virtual machines (VM) for Hadoop; (*ii*) a *configuration service* for configuring VMs; (*iii*) an *application service* for handling job flow; and (*iv*) a *frontend service* for implementing Amazon EMR API and processing users' requests.

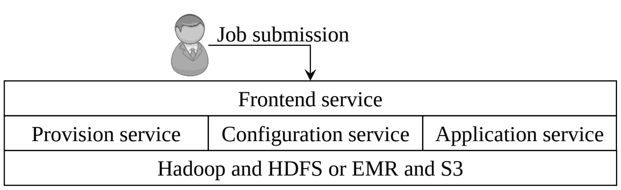


Fig. 7: Resilin architecture.

Pros. Resilin provides a way for exploiting the best available public resources. The major advantage of Resilin over EMR is that users can dynamically handle VMs as per needs, select different types of VMs, operating systems, and Hadoop versions.
Cons. Resilin cannot be directly implemented to process geo-distributed data and requires further enhancements, which are not presented in [34]. Resilin does not provide data security by dealing with data sensitivity that can be explored in a hybrid setting. The next hybrid cloud based system handles data sensitivity and provides a secure solution.

SEMROD. SEMROD [122] first finds sensitive and non-sensitive data and sends non-sensitive data to public clouds. Private and public clouds execute the map phase. In order to hide, some keys that are required by the private cloud, the public cloud sends all the outputs of the map phase to the private cloud only in the first iteration. The private cloud executes the reduce phase only on sensitive key records and ignores non-sensitive keys. For example, let k_1 and k_2 are two keys at the public cloud, and k_2 also exists at the private cloud. The public cloud will send $\langle key, value\rangle$ pairs of k_1 and k_2 to the private cloud that will perform the reduce phase only on k_1. Public clouds, also, execute the reduce phase on all the outputs of the map phase. At the end, a filtering step removes duplicate entries, created by the transmission of the public mappers' outputs.
Pros. By storing sensitive data in the private cloud, SEMROD provides a secure execution and performs efficiently if the non-sensitive data is smaller than the sensitive data. Note that SEMROD is not the first hybrid cloud solution for MapReduce computations based on data sensitivity. HybrEx [137], Sedic [138], and Tagged-MapReduce [139] are also based on data sensitivity. However, they are not secure because during the computation they may leak information by transmitting *some* non-sensitive data between the private and the public cloud, and this is the reason we do not include HybrEx, Sedic, and Tagged-MapReduce in this survey.
Cons. The transmission of the whole outputs of the map phase to the private cloud is the main drawback of SEMROD. If only a few keys are required at the private cloud, then it is useless to send entire public side outputs to the private cloud.

HOG. Hadoop on the Grid (HOG) [29] is a geo-distributed and dynamic Hadoop framework on the grid. HOG accesses the grid's resources in an opportunistic manner, *i.e.*, if users do not own resources, then they can opportunistically execute their jobs, which can be preempted at any time when the resource owner wants to execute a job. HOG is executed on the top of Open Science Grid (OSG) [140], which spans over 109 sites in the United States and consists of approximately 60,000 CPU cores. HOG has the following components:

1) *Grid submission and execution component*. This component handles users' requests, allocation and deallocation of the nodes on the grid, which is done by transferring a small-sized Hadoop executables package, and the execution of a MapReduce job. Further, this component dynamically adds or deletes nodes according to an assigned job requirement.
2) *HDFS*. HDFS is deployed across the grid. Also, due to preemption of tasks, which results in a higher node failure, the replication factor is set to 10. A user submits data to a dedicated node using the grid submission component that distributes the data in the grid.
3) *MapReduce-based framework*. This component executes MapReduce computations across the grid.

Pros. HOG (and the following two grid-based systems) consider that a single DC is distributed across multiple DCs while the previously reviewed frameworks support multiple DCs collaborating for a task.
Cons. The multi-cluster HOG processing is only for fault-tolerance; however, there is *no* real multi-cluster processing in HOG so that no parallel processing, and hence, no necessity for aggregating the site outputs.

KOALA grid-based system. Ghit et al. [33] provided a way to execute a MapReduce computation on KOALA grid [141]. The system has three components, as shown in Fig. 8, MapReduce-Runner, MapReduce-Launcher, and MapReduce-Cluster-Manager.
MapReduce-Runner. MapReduce-Runner interacts with a user, KOALA resource manager, and the grid's physical re-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

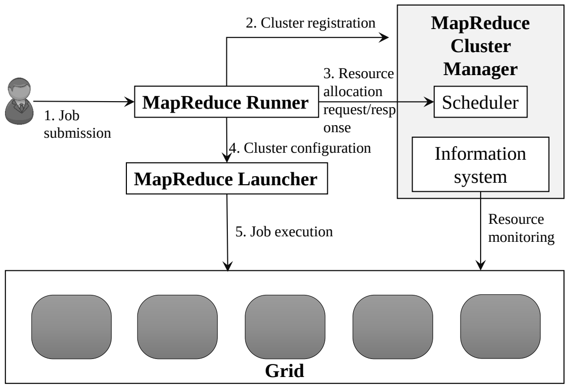


Fig. 8: A multi-cluster MapReduce architecture based on KOALA grid scheduler.

sources via MapReduce-Launcher. It deploys a MapReduce cluster on the grid with the help of MapReduce-Launcher and monitors parameters such as the total number of (real) MapReduce jobs, the status of each job, and the total number of map and reduce tasks. MapReduce-Runner designates one node as the master node and all the other nodes as slave nodes.

MapReduce-Launcher. MapReduce-Launcher is responsible for configuring a distributed file system on the grid resources and a compute framework. In addition, MapReduce-Launcher executes an assigned job on the grid and turns off the cluster after the execution.

MapReduce-Cluster-Manager. MapReduce-Cluster-Manager is a central entity, which stays in the scheduler site for maintaining the grid, metadata of each MapReduce cluster. MapReduce-Cluster-Manager is also responsible for growing or shrinking the nodes in a cluster, with the help of the KOALA Information System module.

Pros. KOALA grid-based system provides scheduling of multiple jobs as a part of single MapReduce instances [142]. Also, it provides a way for performance, data, failure and version isolation in the grid settings.

Cons. KOALA grid-based system and the previous systems, HOG, are dependent on a special grid architecture. Also, the practicality of these systems in the context of public clouds is not known.

HybridMR. HybridMR [15] allows a MapReduce job on a desktop grid and cloud infrastructures simultaneously. HybridMR consists of two layers, as follows: (*i*) a *service layer* that allows data scheduling, job scheduling, metadata storage, and database storage to a new distributed file system, called HybridDFS; (*ii*) a *resource layer* that contains reliable cluster nodes and many unreliable volunteer/desktop nodes. A user uploads MapReduce jobs and data into HybridMR. The data scheduler assigns data to cloud nodes and desktop nodes on which jobs are scheduled by the job scheduler.

Pros. Unlike KOALA grid-based system and HOG, HybridMR provides a way to execute a job on the cloud as well as on the grid.

Cons. Resilin, KOALA grid-based system, HOG, and HybridMR execute a MapReduce job using a modified version of the standard Hadoop on grid systems in an opportunistic manner, which a challenging task, because resource seizing may delay the entire job. In addition, all the grid-based systems, such as mentioned above, suffer from inherited limitations of a grid, *e.g.*, accounting and administration of the grid, security, pricing, and a prior knowledge of resources. Moreover, the grid-based systems include new nodes during computations. However, adding nodes without data locality during the execution may reduce the job performance, resulting in no gain from inter-DC scaling [143].

4.2.2 Geo-distributed stream processing frameworks for user-located geo-distributed data

In order to give a flavor of stream processing in user-located geo-distributed data, we include Photon [116] that is not built on top of Apache Spark.

Google's Photon. Google's Photon [116] is a highly scalable and very low latency system, helping Google Advertising System. Photon works on exactly-once semantics (that is only one joined tuple is produced) and handles automatic DC-level fault-tolerance.

Photon performs equijoin between a primary table (namely, the *query event* that contains `query id`, `ads id`, and `ads text`) and a foreign table (namely, the *click event* that contains `click id`, `query id`, and `user clicked log information`). Both the tables are copied to multiple DCs. Any existing streaming-based equijoin algorithm cannot join these two tables, because a click can only be joined if the corresponding query is available. In reality, the query needs to occur before the corresponding click, and that fact is not always true in the practical settings with Google, because the servers generating clicks and queries are not located at a single DC.

An identical Photon pipeline is deployed in multiple DCs, and that works independently without directly communicating with other pipelines. Each pipeline processes all the clicks present in the closest DCs and tries to join the clicks with the query based on the `query id`. Each pipeline keeps retrying until the click and query are joined and written to an *IdRegistry*, which guarantees that each output tuple is produced exactly once.

Google's Mesa [94], Facebook's Puma, Swift, and Stylus [144] are other industry deployed stream processing distributed frameworks. A brief survey of general approaches for building high availability stream processing systems with challenges and solutions (Photon, F1, and Mesa) is presented in [145].

4.2.3 SQL-style processing framework for user-located geo-distributed data

Google's Spanner. Google's Spanner [146] is a globally-distributed data management system. In [37], database aspects, *e.g.*, distributed query execution in the presence of sharding/resharding, query restarts upon transient failures, and range/index extraction, of Spanner are discussed.

In Spanner, *table interleaving* is used to keep tables in the database, *i.e.*, rows of two tables that will join based on a joining attribute are kept co-located, and then, tables are partitioned based on the key. Each partition is called a *shard* that is replicated to multiple locations.

A new type of operation is introduced, called *Distributed Union* that fetches results from all the shard according to a query. However, performing the distributed union before executing any other operations, *e.g.*, scan, filter, group by,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

join, and top-k, will cause to read multiple shards, which may not participate in the final output. Hence, all such operators are pushed to the table before the distributed union, which takes place at the end to provide the final answer. Three different mechanisms of index or range retrieval are given, as follows: distribution range extraction, seek range extraction, and lock range extraction.

A recent paper [147] carries the same flavor of the hybrid cloud computation, discussed in §4.2.1, and suggests a general framework for executing SQL queries, specifically, select, project, join, aggregation, maximum, and minimum, while not revealing any sensitive data to the public cloud during the computation.

5 Data Transfer Systems/Algorithms

We reviewed G-MR (§4.1.1) that finds the best way only for inter-cloud data transfer, but not based on real-time parameters. Tudoran et al. [8] and Gadre et al. [115] proposed a data management framework for efficient data transfer among the clouds, where each cloud holds monitoring, data transfers, and decision management agents. The monitoring agent monitors the cloud environment such as available bandwidth, throughput, CPU load, I/O speed, and memory consumption. The decision agent receives the monitored parameters and generates a real-time status of the cloud network and resources. Based on the status, the decision agent finds a directed/multi-hop path for data transfer from the source to the destination. The transfer agent performs the data transfers and exploits the network parallelism. ViNe [103] is the only system that offers end-to-end secure connectivity among the clusters executing MapReduce jobs.

Volley. Volley [111] is a 3-phase iterative algorithm that places data across geo-distributed DCs. A cloud submits its logs to Volley that analyzes the logs using SCOPE [148], a scalable MapReduce-based platform, for efficient data transfer. Volley also includes real-time parameters, such as capacity and cost of all the DCs, latency among DCs, and the current data item location. The current data item location helps in identifying whether the data item requires movement or not. In phase 1, data is placed according to users' IP addresses at locations as closest as possible to the user. However, the data locations as a consequence of phase 1 are not the best in terms of closeness to the actual user's location. Hence, in phase 2, data is moved to the closest and best locations to the user via a MapReduce-based computation. Phase 3 is used to satisfy the DC capacity constraint, and if a DC is overloaded, then some data items that are not frequently accessed by the user are moved to another closest DC.

Pros. Volley can be used in optimizing automatic data placement before a computation execution using any above-mentioned frameworks in §4.2.

Cons. Volley does not consider bandwidth usage [149], unlike JetStream [35].

Apache Flume. Apache Flume [150] is a distributed, reliable, scalable and available service for efficiently collecting, aggregating, and moving a large amount of log data from various sources to a centralized data store — HDFS or HBase. However, it should be noted down that Flume is *a general-purpose data collection service*, which can be used in geo-distributed settings. An *event* is the basic unit of the data transported inside Flume. Events and log data are generated at different log servers that have Flume *agents*, see Fig. 9. Flume agents transfer data to intermediate nodes, called *collectors*. The collector aggregates data and pushes this data to a centralized data store. Flume provides guaranteed data delivery and stores data in a buffer when the rate of incoming data exceeds the rate at which data can be written to the destination [151].

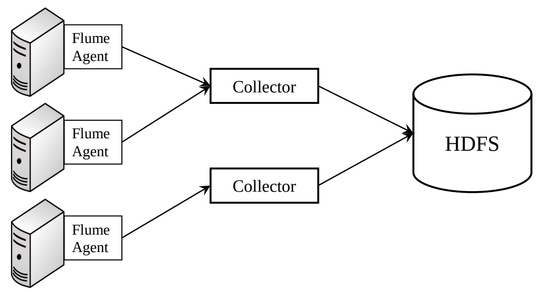


Fig. 9: Apache Flume.

6 Scheduling in Geo-Distributed Systems

In this section, we present some methods/architectures that preprocess a job before deploying it over distributed locations to find the best way for data distribution and/or the best node for the computation. The main idea of the following methods is in reducing the total amount of data transfer among DCs. Note that the following methods work offline and do not provide a way for executing a geo-distributed job on top of Hadoop/Spark, unlike systems in §4.2 that execute a job and may handle such offline tasks too.

6.1 Scheduling for Geo-distributed MapReduce-based Systems

WANalytics. WANalytics [2] preprocesses a MapReduce job before its real implementation and consists of two main components, as follows:

Runtime analyzer: executes user's job directed acyclic graph (DAG), which is about job execution flow, in a distributed way across DCs. The runtime analyzer finds a physical plan that specifies where does each stage of the job to be executed and how will data be transferred across DCs. The runtime layer consists of a centralized coordinator, only with one DC that interacts with all the other DCs. Users submit a DAG of jobs to the coordinator that asks the workload analyzer to provide a physical distributed execution plan for the DAG.

Workload analyzer: continuously monitors and optimizes the user's DAG and finds a distributed physical plan according to the DAG. The plan is determined in a manner that minimizes the total bandwidth usage by considering DC locations and data replication factor. *Cons*. Unlike Iridium, WANalytics does not consider the network bandwidth and job latency, and only focuses on the amount of data transfer among DCs. In addition, WANalytics is not designed to handle iterative machine learning workflows [152].

Shuffle-aware data pushing. Heintz et al. [14] suggested shuffle-aware data pushing at the map phase. It finds all those mappers that affect the job completion in a DC, and hence, rejects those mappers for a new job. In other words,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

the algorithm selects only mappers that can execute a job and shuffle the intermediate data under a time constraint. Mappers are selected based on monitoring the most recent jobs. The algorithm is presented for a single DC and can be extended to geo-distributed settings.
Cons. It is assumed that the same mappers have appeared in previous jobs; otherwise, it is hard to have a prior knowledge of mappers.

Reseal. Reseal [114] considers a bi-objective scheduling problem for scheduling response-critical (RC) tasks and best effort (BE) tasks. Each task is associated with a utility function that provides a value, which is a function of the task's slow down. A task's value is initially set to be high, and then, decreases over time if the task is delayed. Two approaches are suggested for allocating RC tasks, as follows: (*i*) *Instant-RC*: refers to the scheduling of a RC task over many BE tasks on the arrival the RC task. In other words, a RC task is allocated to have an identical throughput as it would achieve in the absence of any BE task in the system; (*ii*) *Threshold-RC*: refers to the scheduling of a RC task according to its utility function. In other words, a RC task is not allocated on its arrival, but scheduled in a manner that it finishes with a slowdown according to its utility function.
Pros. Reseal is the first approach for dealing with realtime scheduling and regarding the network bandwidth in terms of BE transfers in the context of geo-distributed computations.

Error-bound vs staleness-bound algorithms. In [153], the authors considered the problem of adaptive data movement satisfying the timeliness vs accuracy under bandwidth limitations and presented two online algorithms: error-bound and staleness-bound algorithms. These algorithms are based on 2-levels of caching. The error-bound algorithm allows the insertion of new values to the second-level cache, and the values from the second-level cache are moved to the first-level cache when their aggregate values exceed the error constraint. In contrast, the staleness-bound algorithm dynamically finds the ranking of the second-level cache values by their (estimated) initial prefix error, and then, defines the first-level cache to comprise the top values from the second-level cache. Both the cache-based algorithms does not answer the following questions: (*i*) how to define the size of the cache, is it application dependent or not? (*ii*) how do the cache-based algorithms handle a huge amount of streaming data in an IoT environment, do the algorithms sustain any progressive computation on data or not.

Rout. Jayalath and Eugster [82] extended Pig Latin [87], called Rout, by introducing geo-distributed data structures and geo-distributed operations. The authors suggest that before executing a geo-distributed job, it is beneficial to analyze the job, thereby the data transfer among DCs is reduced. A Rout program maximizes job parallelization by generating a set of MapReduce jobs and determines optimal points in the execution for performing inter-DC copy operations. A Rout program generates a MapReduce dataflow graph, like Pig Latin, and analyzes it for finding points, *i.e.*, which DCs will perform inter-DC data transfer and to where. Based on the points, the dataflow graph is annotated, and then, an execution plan is generated to consider dynamic runtime information and transfer data to not overloaded DCs.
Pros. Rout reduces the job completion time down to half, when compared to a straightforward schedule.

Meta-MapReduce. Meta-MapReduce [30] reduces the amount of data required to transfer between different locations, by transferring essential data for obtaining the result. Meta-MapReduce regards the locality of data and mappers-reducers and avoids the movement of data that does not participate in the final output. Particularly, Meta-MapReduce provides an algorithmic way for computing the desired output using metadata (which is exponentially smaller than the original input data) and avoids uploading the whole data. Thus, Meta-MapReduce enhances the standard MapReduce and can be implemented into the state-of-the-art MapReduce systems, such as Spark, Pregel [22], or modern Hadoop.
Pros. A MapReduce job can be enhanced by sampling local data, which cannot be used for future analysis. However, designing good sampling algorithms is hard. Meta-MapReduce does not need any sampling, and hence, has a wide applicability.

Zhang el al. [11] provided prediction-based MapReduce job localization and task scheduling approaches. The authors perform a sub-cluster-aware scheduling of jobs and tasks. The sub-cluster-aware scheduling finds sub-clusters that can finish a MapReduce job efficiently. The decision is based on several parameters such as the execution time of the map phase, the execution time of a DC remote map task, percentage of remote input data, number of map tasks in the job, and number of map slots in a sub-cluster.

Li et al. [154] provided an algorithm for minimizing the shuffle phase inter-DC traffic by considering both data and task allocation problems in the context of MapReduce. The algorithm finds DCs having higher output to input ratio and poor network bandwidth, and hence, move their data to a *good* DC. Note that the difference between this algorithm and Iridium [10] is in considering an underlying framework. Chen et al. [105] also provided a similar algorithm and showed that the data local computations are not always best in a geo-distributed MapReduce job.

6.2 Scheduling for Geo-distributed Spark-based Systems

Pixida. Pixida [5] is a scheduler that minimizes data movement across resource constrained inter-DC links. Silos are introduced as the main topology. Silo considers each node of a single location as a super-node in a task-level graph. The edges between the super-nodes show the bandwidth between them. Hence, Pixida considers that sending data to a node within an identical silo is preferable than sending data to nodes in remote silos. Further, a variation of the min-k cut problem is used to assign tasks in a silo graph.

Flutter. The authors suggested a scheduling algorithm, Flutter [102], for MapReduce and Spark. This algorithm is network-aware and finds on-the-fly job completion time based on available compute resources, inter-DC bandwidth, and the amount of data in different DCs. At the time of the final computation assignment, Flutter finds a DC that results

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

in the least amount of data transfer and having most of the inputs that participate in the final output.

Lazy optimal algorithm. Lazy optimal algorithm [110] considers a tradeoff between the amount of inter-DCs data and staleness. Lazy optimal algorithm is based on two algorithms, as follows: (*i*) traffic optimality algorithm: transfers exactly one update to the final computational site for each distinct key that arrived in a specified time window, and (*ii*) eager optimal algorithm: transfers exactly one update for each distinct key immediately after the last arrival for that key within a specified time window. The lazy optimal algorithm makes a balance between the two algorithms and transfers updates at the last possible time that would still provide the optimal value of staleness. As a major advantage, the lazy optimal algorithm considers several factors such as the network bandwidth usage, data aggregation, query execution, and response latency, and extends Apache Storm [80] for supporting efficient geo-distributed stream analytics [129].

6.3 Resource Allocation Mechanisms for Geo-Distributed Systems

Awan. Awan [1] provides a resource lease abstraction for allocating resources to individual frameworks. In other words, Awan is a system that does not consider underlying big-data processing frameworks when allocating resources. Awan consists of four centralized components, as follows: (*i*) file master, (*ii*) node monitor, (*iii*) resource manager, which provides the states of all resources for different frameworks, and (*iv*) framework scheduler, which acquires available resources using a resource lease mechanism. The resource lease mechanism provides a lease time to each resource in which resources are only used by the framework scheduler during the lease only, and after the lease time, the resource must be vacated by the framework scheduler.

Ghit et al. [33] provided three policies for dynamically resizing a distributed MapReduce cluster. As advantages, these policies result in less reconfiguration costs and handle data distribution in reliable and fault-tolerant manners.

- *Grow-Shrink Policy*. It is a very simple policy that maintains a ratio of the number of running tasks (map and reduce tasks) and the number of available slots (map and reduce slots). Based on the ratio, the system adds (or removes) nodes to (or from) the cluster.
- *Greedy-Grow Policy*. This policy suggests adding a node to a cluster in a greedy manner. However, all the resources are added regardless of the cluster utilization.
- *Greedy-Grow-with-Data Policy*. This policy adds core nodes, unlike the previous policy that adds only transient nodes. Hence, on resource availability, the node is configured for executing TaskTracker. However, the policy does not consider cluster shrink requests.

Ghit et al. [155] extended the above-mentioned policies by accounting dynamic demand (job, data, and task), dynamic usage (processor, disk, and memory), and actual performance (job slowdown, job throughput, and task throughput) analysis when resizing a MapReduce cluster.

Gadre et al. [25] provided an algorithm for assigning the global reduce task, thereby the data transfer is minimal. The algorithm finds the answer to the questions such as when to start the reduce phase for a job, where to schedule the global reduce task, which DC is holding a major part of partial outputs that participate in the final output, and how much time is required to copy outputs of DCs to a single (or multiple) location for providing the final outputs? During a MapReduce job execution, one of the DCs (working as a master DC) monitors all the remaining DCs and keeps the total size of outputs in each DC. Monitoring helps in identifying the most prominent DC while scheduling the global reduce phase.

Cons. The Awan and the above-mentioned three policies do not answer a question: what will happen to a job if resources are taken during the execution? Also, these mechanisms do not provide a way for end-to-end overall improvement of the MapReduce dataflow, load balancing, and cost-efficient data movement [156]. Gadre et al. [25] optimizes the reduce data placement according to map's output location, which might slow down the job due to the low bandwidth [26].

7 Concluding Remarks and Open Issues

The classical parallel computing systems cannot efficiently process a huge amount of massive data, because of less resiliency to faults and limited scalability of systems. MapReduce, developed by Google in 2004, provides efficient, fault-tolerant, and scalable large-scale data processing at a single site. Hadoop and Spark were not designed for on-site geographically distributed data processing; hence, all the sites send their *raw data* to a single site before a computation proceeds. In this survey, we discussed requirements and challenges in designing geo-distributed data processing using MapReduce and Spark. We also discussed critical limitations of using Hadoop and Spark in geo-distributed data processing. We investigated systems under their advantages and limitations. However, we did not find a system that can provide a solution to all the mentioned challenges in §3.

Open issues. Based on this survey, we identified the following important issues and challenges that require further research:

- *Security and privacy*. Most of the frameworks do not deal with security and privacy of data, computation, data transfer, or a deadline-constraint job. Hence, a major challenge for a geo-computation is: how to transfer data and computations to different locations in a secure and privacy-preserving manner, how to trust the requested computations, how to ensure security and privacy within a cluster, and how to meet real-time challenges (recall that we found that G-Hadoop [16], ViNE [103], and SEMROD [122] provide an authentication mechanism, end-to-end data transfer security, and sensitive data security in the hybrid cloud, respectively).
- *Fine-grain solutions*. Most of the frameworks do not provide fine-grain solutions to different types of compatibilities. In reality, different clusters have different versions of software, hence, how will be a job executed on different sites having non-identical implementations of MapReduce, operating systems, data storage systems, and security-privacy solutions.
- *Global reducer*. There are some solutions (*e.g.*, G-Hadoop [16] and HMR [31]) that require a global reducer at a pre-defined location. However, the selection of a

global reducer has been considered separately while it directly affects the job completion time. Hence, a global reducer may be selected dynamically while respecting several real-time parameters [25]. Though not each site sends its complete datasets, there still exists open questions to deal with, *e.g.*, should all the DCs send their outputs to a single DC or to multiple DCs that eventually converge, should a DC send its complete output to a single DC or partition its outputs and send them to multiple DCs, and what are the parameters to select a DC to send outputs.

- *A wide variety of operations*. The existing work proposes frameworks that allow a limited set of operations. However, it is necessary to find answers to the following question: how to perform many operations like the standard MapReduce on a geographically distributed MapReduce-based framework. Also, we did not find a system that can process secure SQL-queries on geo-distributed data, except in [147], but they focus on the hybrid cloud and store a significant amount of non-sensitive data in the private cloud too.
- *Job completion time and inter-DC transfer*. Most reviewed frameworks do not deal with the job completion time. In a geo-distributed computation, the job completion time is affected by distance and the network bandwidth among DCs, the outputs at each DC, and the type of applications. Iridium [10] and JetStream [35] handle job completion time. However, there is no other framework that jointly optimizes job completion time and inter-DC transfer while regarding variable network bandwidth, which is considered in JetStream [35] and WANalytics [2]. Thus, there is a need to design a framework that optimizes several real-time parameters and focuses on the job completion time. In addition, the system must dynamically learn and decide whether the phase-to-phase or the end-to-end job completion time is crucial? Answering this question may also require us to find straggling mappers or reducers in the partial or entire computation [13], [14].
- *Consistency and performance*. A tradeoff is evident between consistency and performance, for example, if a job is distributed over different locations such as in bank transactions. It is required in a geo-distributed computation to have consistent outputs while maximizing the system performance. In order to ensure the output consistency, the distributed components must be in coordination or more appropriately the WAN links must be in coordination. However, achieving coordination is not a trivial task and would certainly incur significant performance overhead in return [157].
- *Geo-distributed IoT data processing*. We reviewed a sufficient number of stream processing systems. Evidently, there is a huge opportunity in developing real-time stream processing systems for IoT. In an IoT environment, data gathering and real-time data analysis are two prime concerns because of several data outsourcing (sensor) devices, which send small data (*e.g.*, GPS coordinates) vs large data (*e.g.*, surveillance videos) possibly at a very high speed. However, the current stream processing systems are not able to handle such a high-velocity data [158] and require explicit ingestion corresponding to an underlying system [159]. Hence, the existing systems in a geo-distributed IoT system cannot support multiple platforms and underlying databases. In such an environment, it would be interesting to find a way to implement existing popular stream processing systems such as Spark, Flink, and decide how and when to transmit data, which types of algorithms will work regarding small vs large data, how much resources are required at the cloud or edge servers, what would be data filtering criteria, how to maintain privacy of entities, and which DC should be selected for the next level processing.
- *Geo-distributed machine learning*. Machine learning (ML) provides an ability to analyze and build models from large-scale data. Specifically, ML helps in classification, recommender systems, clustering, frequent itemsets, pattern mining, collaborative filtering, topic models, graph analysis, etc. There are some famous ML systems/libraries, *e.g.*, Apache Mahout [160], MLlib [161], GraphLab [162], and Google's TensorFlow [163]. However, all these systems deal with only a single DC ML computations. To the best of our knowledge, there are two systems/algorithms, Gaia [164] and [152], for performing geo-distributed ML computations. These systems regard variable network bandwidth, and Gaia does not require to change an ML algorithm to be executed over geo-locations. However, we still need to explore a wide variety of geo-distributed ML algorithms in the context of security, privacy, extending MLlib and Mahout to be able to work on geo-distributed settings.

In short, we can conclude that geo-distributed big-data processing is highly dependent on the following five factors: task assignment, data locality, data movement, network bandwidth, and security and privacy. However, currently, we are not aware of any system that can jointly optimize all of these factors. In addition, while designing a geo-distributed system, one should memorize the lesson from the experience of Facebook's teams: the system should "*not just on the ease of writing applications, but also on the ease of testing, debugging, deploying, and finally monitoring hundreds of applications in production*" [144].

ACKNOWLEDGEMENTS

The authors of the paper are thankful to the anonymous reviewers for insightful comments. This research was supported by a grant from EMC Corp. We are also thankful to Faisal Nawab for suggesting BigDAWG [98] and Rheem [99], Yaron Gonen for discussing early versions of the paper, and Ancuta Iordache for discussing Resilin [34]. The first author's research was partially supported by the Rita Altura Trust Chair in Computer Sciences; the Lynne and William Frankel Center for Computer Science; the grant from the Ministry of Science, Technology and Space, Israel, and the National Science Council (NSC) of Taiwan; the Ministry of Foreign Affairs, Italy; the Ministry of Science, Technology and Space, Infrastructure Research in the Field of Advanced Computing and Cyber Security and the Israel National Cyber Bureau.

REFERENCES

[1] A. Jonathan and et al., "Awan: Locality-aware resource manager for geo-distributed data-intensive applications," in *IC2E*, 2016, pp. 32–41.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

[2] A. Vulimiri and et al., "WANalytics: analytics for a geo-distributed data-intensive world," in *CIDR*, 2015.
[3] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Commun. ACM*, vol. 58, no. 7, pp. 56–68, 2015.
[4] K. A. Hawick and et al., "Distributed frameworks and parallel algorithms for processing large-scale geographic data," *Parallel Computing*, vol. 29, no. 10, pp. 1297–1333, 2003.
[5] K. Kloudas and et al., "PIXIDA: optimizing data parallel jobs in wide-area data analytics," *PVLDB*, vol. 9, no. 2, pp. 72–83, 2015.
[6] http://www.computerworld.com/article/2834193/cloud-computing/5-tips-for-building-a-successful-hybrid-cloud.html.
[7] http://www.computerworld.com/article/2834193/cloud-computing/5-tips-for-building-a-successful-hybrid-cloud.html.
[8] R. Tudoran and et al., "Bridging data in the clouds: An environment-aware system for geographically distributed data transfers," in *CCGrid*, 2014, pp. 92–101.
[9] R. Tudoran, G. Antoniu, and L. Bougé, "SAGE: geo-distributed streaming data analysis in clouds," in *IPDPS Workshops*, 2013, pp. 2278–2281.
[10] Q. Pu and et al., "Low latency geo-distributed data analytics," in *SIGCOMM*, 2015, pp. 421–434.
[11] Q. Zhang and et al., "Improving Hadoop service provisioning in a geographically distributed cloud," in *IEEE Cloud*, 2014, pp. 432–439.
[12] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Making every bit count in wide-area analytics," in *HotOS*, 2013.
[13] M. Cardosa and et al., "Exploring MapReduce efficiency with highly-distributed data," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 2011, pp. 27–34.
[14] B. Heintz, A. Chandra, R. K. Sitaraman, and J. B. Weissman, "End-to-end optimization for geo-distributed MapReduce," *IEEE Trans. Cloud Computing*, vol. 4, no. 3, pp. 293–306, 2016.
[15] B. Tang, H. He, and G. Fedak, "HybridMR: a new approach for hybrid MapReduce combining desktop grid and cloud infrastructures," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4140–4155, 2015.
[16] L. Wang and et al., "G-Hadoop: MapReduce across distributed data centers for data-intensive computing," *FGCS*, vol. 29, no. 3, pp. 739–750, 2013.
[17] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, pp. 183–236, 1990.
[18] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[19] Apache Hadoop. Available at: http://hadoop.apache.org/.
[20] M. Zaharia and et al., "Spark: Cluster computing with working sets," in *HotCloud*, 2010.
[21] M. Isard and et al., "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.
[22] G. Malewicz and et al., "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.
[23] Apache Giraph. Available at: http://giraph.apache.org/.
[24] C. Jayalath, J. J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running MapReduce across data centers," *IEEE Trans. Computers*, vol. 63, no. 1, pp. 74–87, 2014.
[25] H. Gadre, I. Rodero, and M. Parashar, "Investigating MapReduce framework extensions for efficient processing of geographically scattered datasets," *SIGMETRICS Performance Evaluation Review*, vol. 39, no. 3, pp. 116–118, 2011.
[26] B. Heintz, C. Wang, A. Chandra, and J. B. Weissman, "Cross-phase optimization in MapReduce," in *IC2E*, 2013, pp. 338–347.
[27] HDFS Federation. Available at: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html.
[28] https://www.telegeography.com/research-services/global-bandwidth-research-service/.
[29] C. He, D. Weitzel, D. Swanson, and Y. Lu, "HOG: distributed Hadoop MapReduce on the grid," in *SC*, 2012, pp. 1276–1283.
[30] F. N. Afrati, S. Dolev, S. Sharma, and J. D. Ullman, "Meta-MapReduce: A technique for reducing communication in MapReduce computations," *CoRR*, vol. abs/1508.01171, 2015.
[31] Y. Luo and B. Plale, "Hierarchical MapReduce programming model and scheduling algorithms," in *CCGrid*, 2012, pp. 769–774.
[32] V. Zakhary and et al., "Db-risk: The game of global database placement," in *SIGMOD*, 2016, pp. 2185–2188.
[33] B. Ghit and et al., "Resource management for dynamic MapReduce clusters in multicluster systems," in *SC*, 2012, pp. 1252–1259.
[34] A. Iordache and et al., "Resilin: Elastic MapReduce over multiple clouds," in *CCGrid*, 2013, pp. 261–268.
[35] A. Rabkin and et al., "Aggregation and degradation in JetStream: Streaming analytics in the wide area," in *NSDI*, 2014, pp. 275–288.
[36] A. Vulimiri and et al., "Global analytics in the face of bandwidth and regulatory constraints," in *NSDI*, 2015, pp. 323–336.
[37] D. F. Bacon and et al., "Spanner: Becoming a SQL system," in *SIGMOD*, 2017, pp. 331–343.
[38] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press, 2014.
[39] F. N. Afrati and et al., "Vision paper: Towards an understanding of the limits of Map-Reduce computation," *CoRR*, vol. abs/1204.1754, 2012.
[40] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *SIGMOD*, 2010, pp. 495–506.
[41] F. N. Afrati and et al., "Fuzzy joins using MapReduce," in *ICDE*, 2012, pp. 498–509.
[42] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting near-duplicates for web crawling," in *WWW*, 2007, pp. 141–150.
[43] B. Chawda and et al., "Processing interval joins on Map-Reduce," in *EDBT*, 2014, pp. 463–474.
[44] F. N. Afrati and et al., "Bounds for overlapping interval join on MapReduce," in *EDBT/ICDT Workshops*, 2015, pp. 3–6.
[45] H. Gupta and B. Chawda, "ϵ-controlled-replicate: An improvedcontrolled-replicate algorithm for multi-way spatial join processing on Map-Reduce," in *WISE*, 2014, pp. 278–293.
[46] F. Tauheed and et al., "THERMAL-JOIN: A scalable spatial join for dynamic workloads," in *SIGMOD*, 2015, pp. 939–950.
[47] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using Map-Reduce," in *ICDE*, 2013, pp. 62–73.
[48] P. Malhotra and et al., "Graph-parallel entity resolution using LSH & IMM," in *EDBT/ICDT Workshops*, 2014, pp. 41–49.
[49] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang, "MapReduce-based pattern finding algorithm applied in motif detection for prescription compatibility network," in *APPT*, 2009, pp. 341–355.
[50] A. Nandi and et al., "Data cube materialization and mining over MapReduce," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 10, pp. 1747–1759, 2012.
[51] T. Milo and et al., "An efficient MapReduce cube algorithm for varied datadistributions," in *SIGMOD*, 2016, pp. 1151–1165.
[52] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman, "Parallel skyline queries," in *ICDT*, 2012, pp. 274–284.
[53] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *EDBT*, 2012, pp. 38–49.
[54] W. Lu and et al., "Efficient processing of k nearest neighbor joins using MapReduce," *PVLDB*, vol. 5, no. 10, pp. 1016–1027, 2012.
[55] G. Zhou, Y. Zhu, and G. Wang, "Cache conscious star-join in MapReduce environments," in *Cloud-I*, 2013, pp. 1:1–1:7.
[56] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *SIGMOD*, 2011, pp. 949–960.
[57] X. Zhang and et al., "Efficient multi-way theta-join processing using MapReduce," *PVLDB*, vol. 5, no. 11, pp. 1184–1195, 2012.
[58] Z. Yu and et al., "Multimedia applications and security in MapReduce: Opportunities and challenges," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 17, pp. 2083–2101, 2012.
[59] H. J. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for MapReduce," in *SODA*, 2010, pp. 938–948.
[60] M. T. Goodrich, "Simulating parallel algorithms in the MapReduce framework with applications to parallel computational geometry," *CoRR*, vol. abs/1004.4708, 2010.
[61] S. Lattanzi and et al., "Filtering: a method for solving graph problems in MapReduce," in *SPAA*, 2011, pp. 85–94.
[62] A. Pietracaprina and et al., "Space-round tradeoffs for MapReduce computations," in *ICS*, 2012, pp. 235–244.
[63] A. Goel and K. Munagala, "Complexity measures for Map-Reduce, and comparison to parallel computing," *CoRR*, vol. abs/1211.6526, 2012.
[64] J. D. Ullman, "Designing good MapReduce algorithms," *ACM Crossroads*, vol. 19, no. 1, pp. 30–34, 2012.
[65] F. N. Afrati and et al., "Upper and lower bounds on the cost of a Map-Reduce computation," *PVLDB*, vol. 6, no. 4, pp. 277–288, 2013.
[66] F. N. Afrati and J. D. Ullman, "Matching bounds for the all-pairs MapReduce problem," in *IDEAS*, 2013, pp. 3–4.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

[67] F. N. Afrati and et al., "Assignment problems of different-sized inputs in MapReduce," *TKDD*, vol. 11, no. 2, pp. 18:1–18:35, 2016.

[68] B. Fish and et al., "On the computational complexity of MapReduce," in *DISC*, 2015, pp. 1–15.

[69] K. Shvachko and et al., "The Hadoop distributed file system," in *MSST*, 2010, pp. 1–10.

[70] J. Lin and C. Dyer, "Data-intensive text processing with MapReduce," *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–177, 2010.

[71] Apache YARN. Available at: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html.

[72] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham, *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[73] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: lightning-fast big data analysis*. "O'Reilly Media, Inc.", 2015.

[74] R. B. Zadeh and et al., "Matrix computations and optimization in Apache Spark," in *SIGKDD*, 2016, pp. 31–38.

[75] X. Meng and et al., "Mllib: Machine learning in Apache Spark," *CoRR*, vol. abs/1505.06807, 2015.

[76] R. S. Xin and et al., "GraphX: a resilient distributed graph system on Spark," in *GRADES*, 2013, p. 2.

[77] M. Zaharia and et al., "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *HotCloud*, 2012.

[78] Apache Flink. Available at: https://flink.apache.org/.

[79] Apache Ignite. Available at: http://ignite.apache.org/.

[80] Apache Storm. Available at: http://storm.apache.org/.

[81] S. Kulkarni and et al., "Twitter Heron: Stream processing at scale," in *SIGMOD*, 2015, pp. 239–250.

[82] C. Jayalath and P. Eugster, "Efficient geo-distributed data processing with Rout," in *ICDCS*, 2013, pp. 470–480.

[83] "Best practices for sharing sensitive environmental geospatial data," 2010, available at: http://ftp.maps.canada.ca/pub/nrcan_rncan/publications/ess_sst/288/288863/cgdi_ip_15_e.pdf.

[84] J. Zerbe, "Geospatial data confidentiality guidelines," 2015.

[85] A. Thusoo and et al., "Hive - A warehousing solution over a Map-Reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[86] Y. Huai and et al., "Major technical advancements in Apache Hive," in *SIGMOD*, 2014, pp. 1235–1246.

[87] C. Olston and et al., "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008, pp. 1099–1110.

[88] M. Armbrust and et al., "Spark SQL: relational data processing in Spark," in *SIGMOD*, 2015, pp. 1383–1394.

[89] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm grid file system," *New Generation Computing*, vol. 28, no. 3, pp. 257–275, 2010.

[90] D. T. Liu, M. J. Franklin, and D. Parekh, "GridDB: A database interface to the grid," in *SIGMOD*, 2003, p. 660.

[91] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[92] L. George, *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[93] B. F. Cooper and et al., "PNUTS: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[94] A. Gupta and et al., "Mesa: Geo-replicated, near real-time, scalable data warehousing," *PVLDB*, vol. 7, no. 12, pp. 1259–1270, 2014.

[95] N. Bronson and et al., "TAO: Facebook's distributed data store for the social graph," in *USENIX Annual Technical Conference*, 2013, pp. 49–60.

[96] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.

[97] P. O. Boykin, S. Ritchie, I. O'Connell, and J. J. Lin, "Summingbird: A framework for integrating batch and online MapReduce computations," *PVLDB*, vol. 7, no. 13, pp. 1441–1451, 2014.

[98] V. Gadepally and et al., "The BigDAWG polystore system and architecture," in *HPEC*, 2016, pp. 1–6.

[99] D. Agrawal and et al., "Road to freedom in big data analytics," in *EDBT*, 2016, pp. 479–484.

[100] P. Derbeko and et al., "Security and privacy aspects in MapReduce on clouds: A survey," *Computer Science Review*, vol. 20, pp. 1–28, 2016.

[101] V. Kundra, "Federal cloud computing strategy," 2011, available at: https://www.dhs.gov/sites/default/files/publications/digital-strategy/federal-cloud-computing-strategy.pdf.

[102] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *INFOCOM*, 2016, pp. 1–9.

[103] K. Keahey and et al., *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[104] K. Oh and et al., "Redefining data locality for cross-data center storage," in *BigSystem*, 2015, pp. 15–22.

[105] W. Chen, I. Paik, and Z. Li, "Tology-aware optimal data placement algorithm for network traffic optimization," *IEEE Trans. Computers*, vol. 65, no. 8, pp. 2603–2617, 2016.

[106] M. Ryden and et al., "Nebula: Distributed edge cloud for data intensive computing," in *IC2E*, 2014, pp. 57–66.

[107] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenter," in *ICDCS*, 2017.

[108] P. Tandon and et al., "Minimizing remote accesses in MapReduce clusters," in *IPDPS Workshops*, 2013, pp. 1928–1936.

[109] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *SOSP*, 2013, pp. 69–84.

[110] B. Heintz and et al., "Optimizing grouped aggregation in geo-distributed streaming analytics," in *HPDC*, 2015, pp. 133–144.

[111] S. Agarwal and et al., "Volley: Automated data placement for geo-distributed cloud services," in *NSDI*, 2010, pp. 17–32.

[112] L. Yazdanov and et al., "EHadoop: Network I/O aware scheduler for Elastic MapReduce cluster," in *CLOUD*, 2015, pp. 821–828.

[113] L. Gu, D. Zeng, P. Li, and S. Guo, "Cost minimization for big data processing in geo-distributed data centers," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 3, pp. 314–323, 2014.

[114] R. Kettimuthu, G. Agrawal, P. Sadayappan, and I. T. Foster, "Differentiated scheduling of response-critical and best-effort wide-area data transfers," in *IPDPS*, 2016, pp. 1113–1122.

[115] H. Gadre, I. Rodero, J. D. Montes, and M. Parashar, "A case for MapReduce over the Internet," in *CAC*, 2013, pp. 8:1–8:10.

[116] R. Ananthanarayanan and et al., "Photon: fault-tolerant and scalable joining of continuous data streams," in *SIGMOD*, 2013, pp. 577–588.

[117] J. Shute and et al., "F1: A distributed SQL database that scales," *PVLDB*, vol. 6, no. 11, pp. 1068–1079, 2013.

[118] J. Rosen and et al., "Iterative MapReduce for large scale machine learning," *CoRR*, vol. abs/1303.3517, 2013.

[119] K. Zhang and X. Chen, "Large-scale deep belief nets with mapreduce," *IEEE Access*, vol. 2, pp. 395–403, 2014.

[120] R. Potharaju and N. Jain, "An empirical analysis of intra- and inter-datacenter network failures for geo-distributed services," in *SIGMETRICS*, 2013, pp. 335–336.

[121] P. A. R. S. Costa and el al., "Medusa: An efficient cloud fault-tolerant MapReduce," in *CCGrid*, 2016, pp. 443–452.

[122] K. Y. Oktay and et al., "SEMROD: secure and efficient MapReduce over hybrid clouds," in *SIGMOD*, 2015, pp. 153–166.

[123] J. Zhang and et al., "Key based data analytics across data centers considering bi-level resource provision in cloud computing," *FGCS*, vol. 62, pp. 40–50, 2016.

[124] Conviva. Available at: http://www.conviva.com/products/the-platform/.

[125] TPC Decision Support benchmark. Available at: http://www.tpc.org/tpcds/.

[126] Big Data Benchmark. Available at: https://amplab.cs.berkeley.edu/benchmark/.

[127] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *SoCC*, 2015, pp. 111–124.

[128] InfoSphere DataStage. Available at: http://www-03.ibm.com/software/products/en/ibminfodata.

[129] B. Cheng and et al., "Geelytics: Geo-distributed edge analytics for large scale IoT systems based on dynamic topology," in *WF-IoT*, 2015, pp. 565–570.

[130] H. Li, M. Dong, K. Ota, and M. Guo, "Pricing and repurchasing for big data processing in multi-clouds," *IEEE Trans. Emerging Topics Comput.*, vol. 4, no. 2, pp. 266–277.

[131] R. L. Cole and et al., "The mixed workload ch-benchmark," in *DBTest*, 2011, p. 8.

[132] S. K. Madria, "Security and risk assessment in the cloud," *IEEE Computer*, vol. 49, no. 9, pp. 110–113, 2016.

[133] Marcos K. Aguilera. Geo-replication in data center applications. Available at: http://boemund.dagstuhl.de/mat/Files/13/13081/13081.AguileraMarcos2.Slides.pdf.

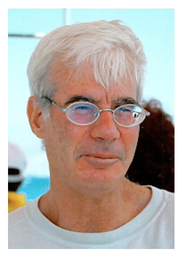
[134] M. Cavallo, G. D. Modica, C. Polito, and O. Tomarchio, "Application profiling in hierarchical Hadoop for geo-distributed computing environments," in *ISCC*, 2016, pp. 555–560.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2723473, IEEE Transactions on Big Data

[135] M. Cavallo, C. Polito, G. D. Modica, and O. Tomarchio, "H2F: a hierarchical Hadoop framework for big data processing in geo-distributed environments," in *BDCAT*, 2016, pp. 27–35.
[136] Amozon Elastic MapReduce. Available at: http://aws.amazon.com/elasticmapreduce/.
[137] S. Y. Ko and et al., "The HybrEx model for confidentiality and privacy in cloud computing," in *HotCloud*, 2011.
[138] K. Zhang and et al., "Sedic: privacy-aware data intensive computing on hybrid clouds," in *CCS*, 2011, pp. 515–526.
[139] C. Zhang, E. Chang, and R. H. C. Yap, "Tagged-MapReduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds," in *CCGrid*, 2014, pp. 31–40.
[140] R. Pordes and et al., "The open science grid," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012057.
[141] H. H. Mohamed and D. H. J. Epema, "KOALA: a co-allocating grid scheduler," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 16, pp. 1851–1876, 2008.
[142] A. Kuzmanovska and et al., "Dynamically scheduling a component-based framework in clusters," in *JSSPP*, 2014, pp. 129–146.
[143] Y. Guo, "Moving MapReduce into the cloud: Elasticity, efficiency and scalability," PhD thesis, University of Colorado, 2015.
[144] G. J. Chen and et al., "Realtime data processing at Facebook," in *SIGMOD*, 2016, pp. 1087–1098.
[145] A. Gupta and J. Shute, "High-availability at massive scale: Building Google's data infrastructure for ads."
[146] J. C. Corbett and et al., "Spanner: Google's globally-distributed database," in *OSDI*, 2012, pp. 261–264.
[147] K. Y. Oktay and et al., "Secure and efficient query processing over hybrid clouds," in *ICDE*, 2017, pp. 733–744.
[148] J. Zhou and et al., "SCOPE: parallel databases meet MapReduce," *VLDB J.*, vol. 21, no. 5, pp. 611–636, 2012.
[149] P. Bodík and et al., "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM*, 2012, pp. 431–442.
[150] Apache Flume. Available at: https://flume.apache.org/.
[151] http://hortonworks.com/apache/flume/.
[152] I. Cano and et al., "Towards geo-distributed machine learning," *CoRR*, vol. abs/1603.09035, 2016.
[153] B. Heintz and et al., "Trading timeliness and accuracy in geo-distributed streaming analytics," in *SoCC*, 2016, pp. 361–373.
[154] P. Li and et al., "Traffic-aware geo-distributed big data analytics with predictable job completion time," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1785–1796, 2017.
[155] B. Ghit and et al., "Balanced resource allocations across multiple dynamic MapReduce clusters," in *SIGMETRICS*, 2014, pp. 329–341.
[156] S. Saeed, "Sandooq: improving the communication cost and service latency for a multi-user erasure-coded geo-distributed cloud environment," Master thesis, University of Illinois at Urbana-Champaign, 2016.
[157] F. Nawab and et al., "The challenges of global-scale data management," in *SIGMOD*, 2016, pp. 2223–2227.
[158] P. Karunaratne and et al., "Distributed stream clustering using micro-clusters on Apache Storm," *JPDC*, 2016.
[159] V. Arora and et al., "Multi-representation based data processing architecture for IoT applications," in *ICDCS*, 2017.
[160] Apache Mahout. Available at: http://mahout.apache.org/.
[161] MLlib. Available at: https://spark.apache.org/mllib/#.
[162] Y. Low and et al., "Distributed GraphLab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.
[163] M. Abadi and et al., "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016, pp. 265–283.
[164] K. Hsieh and et al., "Gaia: Geo-distributed machine learning approaching LAN speeds," in *NSDI*, 2017, pp. 629–647.

Shlomi Dolev received his DSc in Computer Science in 1992 from the Technion. He is the founder and the first department head of the Computer Science Department at Ben-Gurion University, established in 2000. Shlomi is the author of a book entitled Self-Stabilization published by MIT Press in 2000. His publications includes more than three hundred publications. He served in more than a hundred program committees, chairing several including the two leading conferences in distributed computing, DISC 2006, and PODC 2014. Prof. Dolev is the head of the Frankel Center for Computer Science and holds the Ben-Gurion university Rita Altura trust chair in Computer Sciences. From 2011 to 2014, Prof. Dolev served as the Dean of the Natural Sciences Faculty at Ben-Gurion University of the Negev. From 2010 to 2016, he has served as Head of the Inter University Computation Center of Israel. Shlomi currentlty serves as the steering committee head of the Computer Science discipline of the Israeli ministry of education.

Patricia Florissi is VP and Global Chief Technology Officer for DellEMC Sales and holds the honorary title of EMC Distinguished Engineer. Patricia is a technology thought leader and innovator, with 20 patents issued and more than 20 patents pending. Patricia is an active keynote speaker on topics related to big-data, technology trends and innovation. Before joining EMC, Patricia was the Vice President of Advanced Solutions at Smarts in White Plains, New York. Patricia holds a PhD in Computer Science from Columbia University in New York.



Ehud Gudes received the BSc and MSc degrees from the Technion and the PhD degree in Computer and Information Science from the Ohio State University in 1976. Following his PhD, he worked both in academia (Pennsylvania State University and Ben-Gurion University (BGU)), where he did research in the areas of database systems and data security, and in industry (Wang Laboratories, National Semiconductors, Elron, and IBM Research), where he developed query languages, CAD software, and expert systems for planning and scheduling. He is currently a Professor in Computer Science at BGU, and his research interests are knowledge and databases, data security, and data mining, especially, graph mining. He is a member of the IEEE Computer Society.

Shantanu Sharma received his PhD in Computer Science in 2016 from Ben-Gurion University, Israel, and Master of Technology (M.Tech.) degree in Computer Science from National Institute of Technology, Kurukshetra, India, in 2011. He was awarded a gold medal for the first position in his M.Tech. degree. Currently, he is pursuing his Post Doc at the University of California, Irvine, USA, assisted by Prof. Sharad Mehrotra. His research interests include designing models for MapReduce computations, data security, distributed algorithms, mobile computing, and wireless communication.

Ido Singer is a senior software engineering manager in DellEMC Israel. For the past 4 years, he managed the RecoverPoint site in Beer-Sheva, mastering replication data-path algorithms. Ido recently led a big-data innovation collaboration with the Ben-Guryon University. Before joining EMC, Ido worked in the IDF as a system software analyst leading a software transportation algorithm team. Ido holds a PhD in Applied Mathematics from Tel-Aviv University, Israel.