CONTRIBUTED
PAPER

Parallel Processing Systems for Big Data: A Survey

This survey categorizes the existing parallel data processing systems for big data and introduces representative projects in each category.

By Yunquan Zhang, *Member IEEE*, Ting Cao, *Member IEEE*, Shigang Li, Xinhui Tian, Liang Yuan, Haipeng Jia, and Athanasios V. Vasilakos, *Senior Member IEEE*

ABSTRACT | The volume, variety, and velocity properties of *big data* and the valuable information it contains have motivated the investigation of many new parallel data processing systems in addition to the approaches using traditional database management systems (DBMSs). MapReduce pioneered this paradigm change and rapidly became the primary big data processing system for its simplicity, scalability, and fine-grain fault tolerance. However, compared with DBMSs, MapReduce also arouses controversy in processing efficiency, low-level abstraction, and rigid dataflow. Inspired by MapReduce, nowadays the big data systems are blooming. Some of them follow MapReduce's idea, but with more flexible models for general-purpose usage. Some absorb the advantages of DBMSs with higher abstraction. There are also specific systems for certain applications, such as machine learning and stream data processing. To explore new research opportunities and assist users in selecting suitable processing systems for specific applications, this survey paper will give a high-level overview of the existing parallel data processing systems categorized by the data input as batch processing, stream processing, graph processing, and machine learning processing and introduce representative projects in each category. As the pioneer, the original MapReduce system, as well as its active variants and extensions on dataflow, data access, parameter tuning, communication, and energy optimizations will be discussed at first. System benchmarks and open issues for big data processing will also be studied in this survey.

KEYWORDS | Big data; machine learning; MapReduce; parallel processing; SQL; survey

I. INTRODUCTION

The scale of petabyte data flood daily from web services, social media, astronomy, and biology science, for example, have driven the shift of data-processing paradigm. Big data refers to a collection of large datasets that may not be processed using traditional database management tools [72]. The storage, manipulation, and especially information retrieval of big data have been widely researched and engineered by academia and industry.

Google's MapReduce [30] leads this shift. It has inspired new ways of thinking about the design and programming of large distributed systems [69]. Compared with traditional database management systems (DBMSs), MapReduce is outstanding for better simplicity, scalability, and fault-tolerance, but controversial in efficiency and programming complexity because of the low abstraction. Since the publication of MapReduce in 2004, there are numerous works targeting the limitations of MapReduce. It is now the most actively researched and solid big data-processing system. Hadoop [7], an open-source implementation of MapReduce, has been extensively used outside Google.

Manuscript received May 27, 2015; revised October 29, 2015 and May 19, 2016; accepted July 5, 2016. Date of publication August 19, 2016; date of current version October 18, 2016. This work was supported by the National Key Research and Development Program of China under Grant 2016YFB0200803, by the National Natural Science Foundation of China under Grant 61432018, Grant 61133005, Grant 61272136, Grant 61521092, Grant 61502450, and Grant 61402441, by the National High Technology Research and Development Program of China under Grant 2015AA01A303 and Grant 2015AA011505, by the China Postdoctoral Science Foundation under Grant 2015T80139, by the Key Technology Research and Development Programs of Guangdong Province under Grant 2015B010108006, and by the CAS Interdisciplinary Innovation Team of Efficient Space Weather Forecast Models.

Y. Zhang, **T. Cao**, **S. Li**, **L. Yuan**, and **H. Jia** are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zyq@ict.ac.cn; caoting@ict.ac.cn; lishigang@ict.ac.cn; yuanliang@ict.ac.cn; jiahaipeng@ict.ac.cn).
X. Tian is with the Advanced Computer Systems Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: tianxinhui@ict.ac.cn).
A. V. Vasilakos is with the Department of Computer Science, Electrical and Space Engineering, Lulea University of Technology, 97187 Lulea, Sweden (e-mail: vasilako@ath.forthnet.gr).

Digital Object Identifier: 10.1109/JPROC.2016.2591592

After the success of MapReduce, many other big data-processing systems also aiming at horizontal scalability, simple API, and schema-free data have emerged. There are three major developing trends. One follows the data parallel idea of MapReduce, which employs low-level programming with user-defined methods for general-purpose usage, but with more flexible programming models, as well as improved performance. For example, Spark [83] supports iterative and interactive computations. Dryad [50] gives fine control over communication graph and user-defined operations rather than requisite Map/Reduce. Another trend takes advantage of the long-time experience of DBMSs using high-level abstractions. In the data storage layer, NoSQL (Not only SQL) databases, such as MongoDB [67] and Cassandra [53], implement the characteristics of scalability, schema-free, and consistency compared with traditional relational databases for big data applications. In the data-processing layer, systems of this trend either only develop SQL-like languages on top of general execution engines such as Hive [75] or build systems from scratch, including storage, execution engine, and programming model, such as Dremel [66]. The remaining trend focuses on domain-specific problems, e.g. machine learning (ML) and stream data processing. Recently, large-scale ML systems have been actively researched and developed, since scalability is one of the bottlenecks now for ML applications. Representative ML systems include the graph-centric GraphLab [62], and the ML-centric Petuum [81]. S4 [68] and Storm [76] are proposed to process stream/real-time data, e.g., queries in search engines and Tweets in Twitter. The following subsection will categorize and paint an overall picture of the current non-relational big data systems.

A. Categorization of Big Data Systems

Big data ecosystems are layered software stacks, including a low-level database and a data-processing engine based on that.

The low-level databases are used for data maintenance and low-latency random queries. Compared with traditional ones, the new generation databases for big data applications are featured with high scalability, schema-free, consistency, high availability, and simple API. Based on the physical data layout on disk, the systems can be categorized as column store, document store, graph store, and key-value store [85]. Fig. 1(a) shows representative systems in each category. A column-oriented database stores attribute values belonging to the same column contiguously, instead of rows. A document-oriented database stores data in documents, with each document assigned a unique key used to retrieve the document. A graph-oriented database stores graph structures representing collections of entities (nodes) and their relationships (edges) with each other. A key-value database stores data as a set of key-value pairs, also known as an associative array, organized into rows. It is designed to scale to a large size.

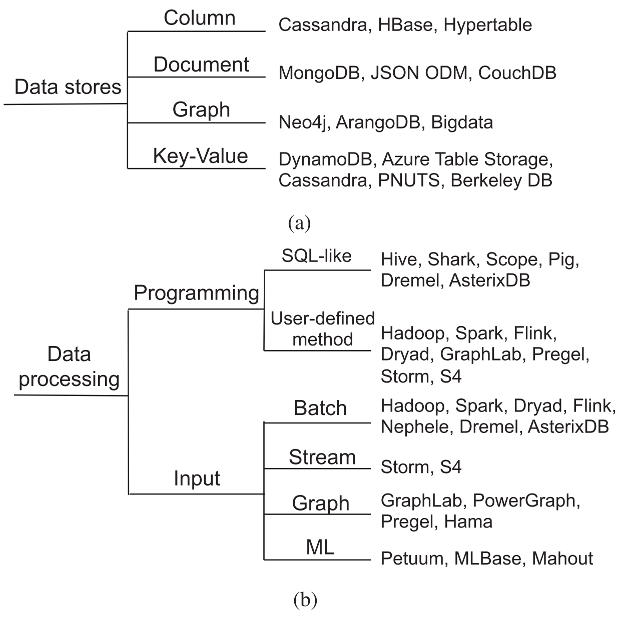

Data stores
Column
Cassandra, HBase, Hypertable
Document
MongoDB, JSON ODM, CouchDB
Graph
Neo4j, ArangoDB, Bigdata
Key-Value
DynamoDB, Azure Table Storage, Cassandra, PNUTS, Berkeley DB
(a)
Data processing
Programming
SQL-like
Hive, Shark, Scope, Pig, Dremel, AsterixDB
User-defined method
Hadoop, Spark, Flink, Dryad, GraphLab, Pregel, Storm, S4
Input
Batch
Hadoop, Spark, Dryad, Flink, Nephele, Dremel, AsterixDB
Stream
Storm, S4
Graph
GraphLab, PowerGraph, Pregel, Hama
ML
Petuum, MLBase, Mahout
(b)

Fig. 1. ***Categorization of (a) data storage systems and (b) data-processing systems.***

Another dimension that can categorize databases is data format, including structured, semi-structured, and unstructured. It decides how data is interpreted. Unstructured data, such as text messages and videos, is data that has not been organized into a structure enabling easy access to elements of the data. Structured data is the opposite, organized so each element can be accessed in various combinations. Semi-structured data lies between the two, although not organized into a structure, it does have additional information associate with the data to allow elements of that data to be addressed. Traditional relational databases just support structured data, but new generation databases such as MongoDB and Cassandra can support structured and semi-structured data, as well as unstructured data to some degree.

Atop a database is a data-processing layer to enable data scientists, developers, and business users to explore and analyze data. As Fig. 1(b) shows, systems in this layer can be categorized along two dimensions. According to the programming abstraction, as mentioned earlier, systems like Dremel and AsterixDB use high-level SQL-like declarative languages while others use MapReduce-like user-defined functions. Depending on the input, we categorize current big data processing as batch, stream, graph and machine learning processing. Batch processing is efficient for processing large datasets, where data are collected, distributed and processed in batches. Stream processing emphasizes on the velocity of the continual input, such as user request and click streams on web pages. Data must be processed in a certain time period disallowing batching data together for efficiency. Graph processing operates on graph data structures and conducts model parallelism besides the data parallelism of MapReduce. It normally operates iteratively on the same input in each iteration followed by some synchronisation. Graph processing can also support partial solutions for ML problems. While, systems like Petuum and Mahout are developed specifically for ML algorithms, which are formalized as iterative-convergent programs [81]. The goal of the systems is fast and efficient convergence to get the optimality of an objective function. Thus, fine-grained fault tolerance and strong consistency for other processing systems may not be necessary for ML problems.

This paper will focus on the survey of data processing layer. It will first introduce MapReduce, compare it with DBMSs, and discuss the optimization works for MapReduce. More detailed surveys for MapReduce and Hadoop can be seen in [55], [57], [72], and [73]. This paper will then overview the other systems in the categories discussed above: batch, stream, graph, and machine learning processing. The section on batch processing will cover the two programming abstractions: user-defined and SQL-like. The survey will be more from a research point of view, focusing on distinctive ideas and model abstractions of different systems, even though some of them may not be popularly used right now.

A fair quantitative and qualitative comparison of all those available big data processing systems is important. However, no standard benchmark suite is available yet. Because of this benchmarking and evaluating pressures of big data systems, this survey will also study current work on big data benchmark building, which is challenging and has not attracted enough attention. Lastly, some lessons and future research direction will be discussed.

The remainder of the paper is organized as follows. Section II will first introduce the basic framework of MapReduce and then compare it with DBMSs. Extension works on MapReduce will be discussed in Section III, including support for different dataflows, optimization for data access and communication, auto parameter tuning, as well as energy efficiency. Section IV will introduce other batch processing systems, including the ones with user-defined methods and SQL-like languages as programming models. Stream, graph, and ML processing systems will be discussed in Section V. Section VI is about big data benchmarks. The remaining sections are for lessons and future works.

II. BAISC MapReduce FRAMEWORK

MapReduce is a programming model for processing big data on large-scale distributed data center systems, introduced by Dean and Ghemawat [30]–[32]. It is simple and abstracts the details of running a distributed program, such as parallelization, fault-tolerance, data distribution and load balancing. It is now widely used for a variety of algorithms, including large-scale graph processing, text processing, data mining, machine learning, statistical machine translation, and many other areas [31]. There are numerous open source and commercial implementations of MapReduce, out of which the most popular one is Hadoop developed by Apache [7]. This section will first talk about the basic framework of MapReduce and then give a comparison with DBMS.

A. MapReduce Framework

The MapReduce model has two phases—*Map* and *Reduce* working on key/value pairs. The Map phase maps the user written functions and input pairs to distributed machines generating intermediate key/value pairs, and the Reduce phase reduces the intermediate pairs to a single result. The workflow of MapReduce execution is as follows

1) The distributed file system (e.g., Google File System [42]) will first partition the input data into a set of *M* splits (e.g., 64 MB in size) and store several copies of each split on different machines for fault tolerance.
2) The MapReduce library will generate a master and many worker copies of the user program (*M* Map tasks and *R* Reduce tasks). The master assigns work to worker copies and coordinates all the tasks running. For locality, the master will attempt to schedule a Map task on a machine that contains a replica of the corresponding input data.
3) A Map worker will scan its local input partition and generate intermediate key pairs by using user's Map function. The results are stored on local disk and are divided into *R* partitions for each Reduce task. The addresses of those intermediate results will be informed to the master. The master will forward those locations to the Reduce workers.
4) A Reduce worker will read the intermediate results from the local disk of each Map task remotely and then sort the results by their key values. The user's Reduce function will aggregate values with the same key and generate final results stored in a global file system.
5) The master will ping every worker periodically. If any Map or Reduce worker fails, its task will be scheduled to another available worker.
6) After all the tasks finish, the user program will be waked up.

MapReduce has several outstanding advantages:

- *Simplicity:* It requires programmers have no parallel and/or distributed system experience. The system configuration and installation are relatively straightforward.

Table 1 Comparison of Parallel DBMSs and MapReduce

	Parallel DBMS	**MapReduce**
Schema	✓	Not naturally
Index	✓	Not naturally
Programming	Declarative (SQL)	Imperative (C++, Java, ...)
Optimization	Compression, Column storage, ...	Not naturally
Pre-parsing	✓	Not naturally
Flexibility	Not naturally	✓
Fault tolerance	Transaction-level	Task-level

- *Fault tolerance:* Failures are common for a computer cluster with thousands of nodes. MapReduce can deal with fine-grain failures, minimize the amount of work lost, and does not need restart the job from scratch.
- *Flexibility:* The input data of MapReduce can have any format instead of a specific schema.
- *Independency:* MapReduce is also storage system-independent. The storage systems supported include files stored in distributed file system, database query results, data stored in Bigtable [24] and structured input files [32].
- *Scalability:* MapReduce can scale to thousands of processors.

B. Comparison Between MapReduce and DBMS

Before MapReduce, parallel DBMSs are used as the approaches for large-scale data analysis. Basically, all MapReduce tasks can be written as equivalent DBMS tasks [74]. During the early term of MapReduce, it provoked strong doubts from Database societies [29]. Comparisons and debates between DBMS and MapReduce have been shown in a series of articles [32], [33], [55], [70], [74].

The debates were toned down until Stonebraker *et al.* [74] concluded the relationship between DBMSs and MapReduce. They noted that MapReduce is complementary to DBMSs, not a competing technology. The goal of DBMS is efficiency while MapReduce aims at scalability and fault tolerance. The two systems are clearly improving themselves through drawing the counterpart's strength. Works like SCOPE [23] and Dryad [50] all point this way. Those systems will be introduced later in this paper.

The prominent differences between DBMS and MapReduce are listed in Table 1. The advantages of MapReduce have been discussed in the previous section. The comparison also exposes some controversial deficiencies of MapReduce. Here, we itemize two major ones.

Inefficiency: Pavlo *et al.* [70] compare the performance of the two systems. Results show that the loading time of MapReduce is faster than DBMS, but slower in task execution time. Longer execution time of MapReduce is partly because some implementation specific problems of MapReduce, such as the start-up cost of Hadoop. There are some model-related reasons too. As Table 1 shows, DBMS does the parsing at loading time and may also reorganize the input data for certain optimizations, while MapReduce won't change the layout of the data and it has to do all the parsing at run time. DBMS also has advanced technologies developed for decades, such as compression, column storage, or sophisticated parallel algorithms. In addition, MapReduce needs to send many control messages to synchronize the processing which increases overhead.

Reusability: MapReduce does not use schema or index. Programmers have to parse the structure of the input files or implement indexes for speedup in the Map and Reduce programs. Besides, users have to provide implementations for simple and common operations, such as selection and projection. Those custom code will be hard to be reused or shared by others and it can be error-prone and suboptimal. However, DBMS has built-in schema and index that can be queried by programmers. DBMS also supports many operators with higher level of abstractions. Users can simply specify what they want from the system instead of how to get it.

To conclude, according to the strength of MapReduce, the situations where MapReduce is preferable to DBMS are [74]:

- Once-analysis data sets. Those data sets are not worth the effort of indexing and reorganization in a DBMS. By comparison, DBMS is more suitable for tasks needing repetitive parsing.
- Complex analysis. Dean and Ghemawat [32] point out some situations where the Map functions are too complicated to be expressed easily in a SQL query, such as extracting the outgoing links from a collections of HTML documents and aggregating by target document.
- Quick start analysis. MapReduce implementations are easy to configure, to program and to run.
- Limited-budget task. Most MapReduce implementations are open-source, while rare open-source parallel DBMSs exist.

III. VARIANTS AND EXTENSIONS OF MapReduce

Here, we will summarize variants and extensions based on basic MapReduce framework, including support for different dataflow, as MapReduce's rigid steps are unfit for flexible dataflow, optimizations for data access, as MapReduce can be inefficient at data parsing, extensions for auto parameter tuning, as many parameters need to be turned in MapReduce system, and communication optimizations, as the current implementations of MapReduce normally use inefficient communication techniques. Research on energy efficiency for MapReduce clusters will be introduced in this section as well.

A. Support for Different Dataflow

MapReduce requires the problem composition to be strict Map and Reduce steps in a batch processing way. This subsection will discuss the extension works on iterative, online and streaming dataflow support based on MapReduce framework.

1) Iterative Dataflow: MapReduce does not support iterative or recursive directly. However, many data analysis applications, such as data mining, graph analysis and social network analysis, require iterative computations. Programmers can manually issue multiple MapReduce jobs to implement iterative programs. However, this method causes three main performance penalties. First, unchanged data from iteration to iteration will be reloaded and reprocessed at each iteration, wasting I/O, network bandwidth and CPU. Second, some termination conditions, like no output change between two consecutive iterations, may itself invoke a MapReduce job. Third, the MapReduce jobs of different iterations have to finish serially. To solve those problems, there are a number of works on extending MapReduce for iterative processing [22], [37], [38], [77], [84].

To support iterative processing, those works normally need to implement three main extensions of MapReduce: adding an iterative programming interface, especially the interface for termination conditions, caching invariant/static data in local disk or memory, and modifying task scheduler to make sure data reuse across iterations. We will next introduce some representative works as examples.

HaLoop: A programmer specifies the loop body and optionally specifies a termination condition and loop-invariant data to write a HaLoop [22] program. The master node of HaLoop maintains a mapping from each slave node to the data partitions that this node processed in the previous iteration. The master node will then try to assign to this node a task using its cached data. HaLoop maintains three types of caches. First, reducer input cache, caches the user specified loop-invariant intermediate tables. So in later iterations, the reducer can search for the key in the local reducer input cache to find associated values and pass together with the shuffled key to the user-defined Reduce function. Second, reducer output cache, stores and indexes the most recent local output on each reducer node to reduce the cost of evaluating termination conditions. Third, mapper input cache, caches the input data split that a mapper performs a nonlocal read on the first iteration. Each of them fits different application scenarios.

iMapReduce: Compared with HaLoop, the advancement of iMapReduce [84] is that it allows asynchronous execution of each iteration. The Map tasks of this iteration can be overlapped with the Reduce tasks of the last iteration. This work assigns a same data subset to a Map task and a Reduce task. Therefore, there is a one-to-one correspondence between the Map and the Reduce tasks. After the Reduce task produces certain records, the results will be sent back to the corresponding Map task. The task scheduler always assigns a Map task and its corresponding Reduce task to the same worker to reduce the network resources needed. The Map can start processing the data without waiting for other Map tasks so the process can be accelerated. iMapReduce proposes the concept of *persistent* Map and Reduce. For a persistent Map task, when all the input data are parsed and processed, the task will wait for the results from the reduce tasks and be activated again. To implement this, the granularity of data split has to make sure all the persistent tasks start at the beginning according to the available task slots. This makes load balancing challenging.

iHadoop: iHadoop [38] also supports asynchronous Map/Reduce tasks as iMapReduce. However, it uses dynamic scheduling instead of static task and node mapping. It will not persist tasks for the next iteration. Instead, the paper reckons that with the large scale data sets, the runtime can optimize the task granularity so that the time to create, destroy, and schedule tasks is insignificant to the time to process the input data. Thus, iHadoop could support wider kinds of iterative applications.

Twister: Similar to iMapReduce's persistent Map/Reduce, Twister [37] also uses long running Map/Reduce tasks and does not initiate new Map and Reduce tasks for every iteration. It is an in-memory runtime using publish/subscribe messaging based communication and data transfer instead of a distributed file system. However, Twister is based on the assumption that data sets can fit into the distributed memory, which is not always the case.

MapIterativeReduce: MapIterativeReduce [77] supports iterative Reduce for reduce-intensive applications such as linear regression and dimension reduction for Microsoft Azure cloud platform. It eliminates the barrier between Map and Reduce by starting reducers process the data as soon as it becomes available from some mappers. The results from the last iteration reducer will be fed back to successive reducers until a reducer combines all the input data and produces the final result.

2) Online and Streaming Dataflow: Computations on MapReduce are executed in a batch-oriented pattern, namely the entire input and output of each Map or

Reduce task has to be materialized to a local file before it can be consumed by the next stage. This step allows for the implementation of a simple and elegant task-based fault-tolerance mechanism. However, these blocking operations can cause performance degradation making it difficult to support online and stream processing.

MapReduce Online: MapReduce Online approach [25], [26] is proposed to extend MapReduce beyond batch processing, in which intermediate data is pipelined between operators. The approach can reduce task completion time and improve system utilization while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. Hadoop Online Prototype (HOP) is implemented based on the MapReduce Online approach.

The MapReduce Online approach provides important advantages to the MapReduce framework. First, since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution. Secondly, pipelining widens the domain of problems to which MapReduce can be applied, such as event monitoring and stream processing. Finally, pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time.

However, pipelining raises several design challenges. First, Google's attractively simple MapReduce fault tolerance mechanism is predicated on the materialization of intermediate state. The paper fixes this by allowing producers to periodically ship data to consumers in parallel with their materialization. Second, pipelines have implicit greedy communication, which is at odds with batch-oriented optimizations, such as pre-aggregation before communication at the map side. This is addressed by modifying the in-memory buffer design. The buffer content is sent to reducers until the buffer grows to a threshold size. The mapper then applies the *combiner* function, sorts the output by *partition* and *reduce* key, and writes the buffer to disk using a spill file format. Finally, pipelining requires that producers and consumers are coscheduled intelligently. The paper uses the client interface to annotate the dependencies of the jobs, so the scheduler can look for the annotation and coschedule the jobs with their dependencies.

Cloud MapReduce [28] also uses a pipeline model for stream data processing. Compared with MapReudce Online which uses pairwise socket connections and buffer to transit intermediate key-value pairs between Map and Reduce phases, Cloud MapReduce uses queues. It also provides added flexibility and scalability by utilizing the features of cloud computing.

DEDUCE: While the increasing of the volume of stored data, the streaming data is also growing rapidly. This requires the data processing system not only scalably analyze the large amount of data but also process very fast moving data streams. The DEDUCE system [52] has been presented as a middleware that attempts to combine real-time stream processing with the capabilities of a large-scale data analysis framework like MapReduce. DEDUCE extends SPADES dataflow composition language [41] to enable the specification and use of MapReduce jobs as dataflow elements, and provides the capability to describe reusable modules for implementing offline MapReduce tasks aimed at calibrated analytic models. It also provides configuration parameters associated with the MapReduce jobs that can be tweaked to perform end-to-end system optimizations and shared resource management. Besides, DEDUCE augments the IBM's System S stream processing runtime infrastructure to support the execution and optimized deployment of Map and Reduce tasks.

B. Optimizations for Data Access

As discussed in Section II-B, compared with DBMS, the inefficiency of MapReduce is partially because it has to do all the parsing at run time while DBMS does it while data loading. Besides, DBMS has developed advanced technologies such as column storage, while MapReduce hasn't. This subsection will introduce the works using index structure, column storage, and other better data placement policies to fasten the MapReduce data access.

1) Index Support: Hadoop++ [34] adopts a non-intrusive method to improve the performance of Hadoop. It boosts task performance without changing the Hadoop framework at all. To achieve this, Hadoop++ injects its code through User-Defined Functions (UDFs) to Hadoop. Results show that Hadoop++ can speed up tasks related to index and join processing. In principle, Hadoop++ has the following three features:

Trojan Index: The original Hadoop implementation does not provide index access due to the lack of *a priori* knowledge of the schema and the MapReduce jobs being executed. Hadoop++ provides Trojan index to integrate indexing capability into Hadoop. The core idea of the Trojan index is to organize the data set as indexed splits, which consist data, index, header and footer. Footer is the delimiter of the split. Trojan indexes are created during the data loading time and thus have no penalty at query time, and these indexes are sorted during creating. Each Trojan index provides an optional index access path which can be used for selective MapReduce jobs.

Trojan Join: Hadoop++ assumes that the schema and the expected workload are known previously, then the input data can be copartitioned during the loading time. In detail, multiple input relations apply the same partitioning function according to the join attributes at data loading time, and place the cogroup pairs, having the same join key from the these relations, on the same split. As a result, *join* operations can be then processed locally within each node at query time, without using *map*, *shuffle* and *reduce* operations.

Trojan Layout: For the internal data of each split, the attributes frequently accessed together are grouped and aggregated. The optimal data layout is calculated according to the workload.

Different from Hadoop++, HAIL [35] (Hadoop Aggressive Indexing Library) changes the upload pipeline of HDFS in order to create different clustered indexes on each data block replica. Thus, each replica can create indexes on different attributes and support multi-attribute querying. In terms of data upload, HAIL improves over HDFS by up to 60% with the default replication factor of three. In terms of query execution, HAIL runs up to 68× faster than Hadoop and even outperforms Hadoop++.

2) Column Storage Support: One of the main limitations of Hadoop-based warehouse systems is that they cannot directly control storage disks in clusters. To store a huge amount of table data, they have to utilize the cluster-level distributed file system, such as HDFS (Hadoop Distributed File System). Thus, the data placement structure that determine the organization of the table data in distributed file system is the key to system efficiency.

RCFile: He *et al.* [44] point out that the data placement structure in a MapReduce environment should have the following four critical requirements:

- Fast data loading.
- Fast query processing.
- Highly efficient storage space utilization.
- Strong adaptivity to highly dynamic workload patterns.

To satisfy the four requirements above, they propose RCFile (Record Columnar File), which is implemented in Hadoop. In principle, RCFile has the following features: Firstly, a table stored in RCFile is first horizontally partitioned into multiple row groups. Then, each row group is vertically partitioned so that each column is stored independently. Secondly, RCFile utilizes a column-wise data compression within each row group, and provides a lazy decompression technique to avoid unnecessary column decompression during query execution. Finally, RCFile allows a flexible row group size. A default size is given considering both data compression performance and query execution performance. RCFile also allows users to select the row group size for a given table.

According to the HDFS structure, a table can have multiple HDFS blocks. In each HDFS block, RCFile organizes records with the basic unit of a row group. That is to say, all the records stored in an HDFS block are partitioned into row groups. For a table, all row groups have the same size. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups. A row group contains three sections. The first section is a sync marker that is placed in the beginning of the row group. The second section is a metadata header for the row group. The third section is the table data section that is actually a column-store.

Llama: To support column storage for the MapReduce framework, Lin *et al.* [60] present new cluster-based data warehouse system, Llama, which is a hybrid data management system which combines the features of row-wise and column-wise database systems. To provide the basis for the vertical partitioning of tables, each imported table is transformed into column groups, where each group contains a set of files representing one or more columns.

A column-wise file format, CFile, is proposed in Llama. In CFile, each file can contain multiple data blocks and each block of the file contains a fixed number of records. Each file includes a block index, which is used to locate a specific block. In order to achieve storage efficiency, block-level compression is used. Furthermore, *concurrent join* is proposed in Llama, which is a multiway join approach in the MapReduce framework. The main objective is to push most operations, such as *join*, to the map phase, which can effectively reduce the number of MapReduce jobs with less I/O costs and network transfer.

3) Related Data Placement Support: In the basic implementation of the Hadoop project, the objective of the data placement policy is to achieve good load balance by distributing the data evenly across the data servers. This simple data placement policy works well with most Hadoop applications that access just a single file, but applications that process data from different files hardly get ideal performance. This is caused by the lack of ability to collocate related data on the same set of nodes.

CoHadoop: Eltabakh *et al.* [39] present CoHadoop to overcome this bottleneck. CoHadoop improves the data placement policy of Hadoop by colocating copies of the related files in the same server, which is done by introducing a new file-level property called a locator. Each locator is represented by a unique value. Each file in HDFS is assigned to at most one locator and many files can be assigned to the same locator. Files with the same locator are placed on the same set of data nodes, whereas files with no locator are placed via Hadoop's default strategy. This colocation process involves all data blocks, including replicas.

CoHadoop introduces a new data structure, the locator table, to manage the locator information and keep track of collocated files. The locator table stores a mapping of locators to the list of files that share this locator. The locator table is not synchronized to disk. Instead, it is maintained dynamically in memory while the namenode is running and reconstructed from the file-system image when the namenode is restarted. To facilitate reconstruction, the locator of each file is stored in its INode, which is a disk-resident data structure that keeps track of all file properties. In practice, for the two use cases of log processing, i.e., join and sessionization, copartitioning and colocating can speed up query processing significantly. However, the data has to be classified according to the attributes before being committed to HDFS. Thus, CoHadoop brings the overhead of preprocessing.

C. Auto Parameter Tuning

In order to get good performance when using data processing system such as MapReduce, a large number of configuration parameters need to be tuned, which control various aspects of job behavior, such as concurrency, I/O behaviors, memory allocation and usage, and network bandwidth usage. The practitioners of big data analytics usually lack the expertise to tune system internals, who can easily run into performance problems. This subsection will discuss the auto parameter tuning work for MapReduce.

1) System Tuning: To automate the process of finding the best setting of the massive configuration parameters, Herodotou and Babu [46] introduce a *Profiler* which can collect profiles online while MapReduce jobs are executed on the production cluster. Novel opportunities arise from storing these job profiles over time, e.g., tuning the execution of MapReduce jobs adaptively within a job execution and across multiple executions. A What-if Engine is used for fine-grained cost estimation, which is needed by the Cost-based Optimizer.

Starfish: Based on the Profiler and What-if engine, Herodotou and Babu present Starfish [47], which is a self-tuning system for big data analytics based on MapReduce. Starfish adapts to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning tricks in Hadoop. Basically, Starfish uses data manager to manage metadata, intermediate data, data layout and storage. Cost-based optimizations, base on the What-if engine, are carried out including job-level tuning, workflow-level tuning, and workload-level tuning. To find a good configuration setting, the cost-based optimizer enumerates and searches efficiently through the space of high-dimensional configuration parameter settings, by making appropriate calls to the What-if engine.

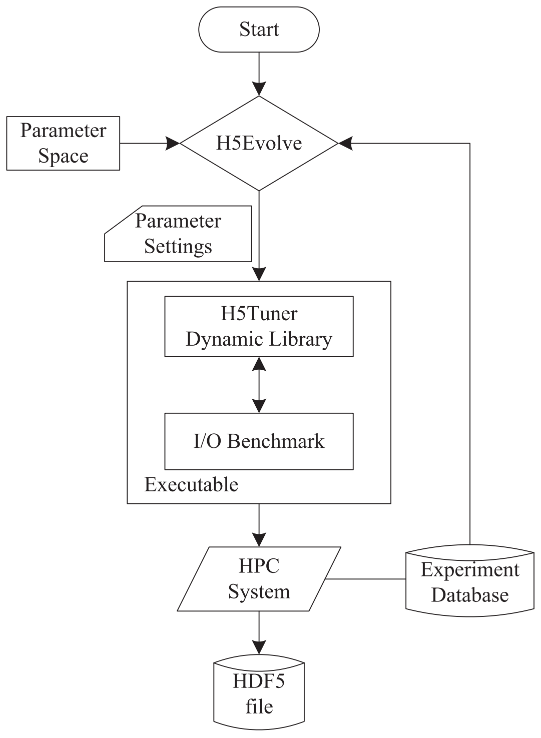


Fig. 2. ***Autotuning framework for HDF5. Adaptation with the permission of Behzad et al.* [18].**

It clusters parameters into low-dimensional subspaces such that the globally optimal parameter setting in the high-dimensional space can be generated by composing the optimal settings found for the subspaces.

Stubby: Stubby [59] is presented as a cost-based optimizer for MapReduce. Stubby is able to consider the correct subspace of the full plan space based on the information available. Furthermore, Stubby is designed to be a general-purpose system for workflow optimization where workflows can be optimized regardless of the interfaces used and availability of information.

2) I/O Tuning: Big data processing is usually data-intensive. Thus I/O can easily become the system bottleneck. The parallel file system and I/O middleware layers all offer optimization parameters that can result in better I/O performance. However, the optimal combination of parameters is dependent on applications. Choosing the best one from parameter combinations brings great burden on algorithm developers.

Behzad *et al.* [18] present an auto-tuning system for optimizing I/O performance of applications in HDF5 data model. The auto-tuning framework is presented in Fig. 2. H5Evolve is designed to search the I/O parameter space using a genetic algorithm. H5Evolve samples the parameter space by testing a set of parameter combinations and then, based on I/O performance, adjusts the combination of tunable parameters for further testing. In order not to modify the source code during autotunning, H5Tuner is designed to dynamically intercept HDF5 calls and inject optimization parameters into parallel I/O calls. To further reduce the searching effort, Behzad *et al.* [19] develop an approach to construct automatically an I/O performance model, and use the model to reduce the search space for good I/O configurations.

D. Communication Optimization

Communication in Hadoop system is implemented by inefficient techniques, such as HTTP/RPC. Lu *et al.* [63] presented DataMPI, which is an efficient communication library for big data computing that features the processing and communication of large numbers of key-value pairs. To bridge the two fields of high performance computing and big data computing and extend MPI to support Hadoop-like big data computing jobs, they abstract the requirements of a 4D (Dichotomic, Dynamic, Data-centric, and Diversified) bipartite communication model.

- Dichotomic. The MapReduce and other big data systems show that communications happen between two communicators. The underlying communication is a bipartite graph, i.e., the processes are dichotomic and belong to either the O communicator or the A communicator.
- Dynamic. Big Data communication has a dynamic characteristic, which means the number of concurrent running tasks in each communicator often changes dynamically because of task finish and launch.
- Data-centric. Jim Gray's Laws [48] tell that computations should be moved to the data, rather than data to the computations in big data computing. Such principle can be found in the most popular big data computing models and systems.
- Diversified. Although many similar communication behaviors among different big data computing systems can be found, there still exist diversities.

They also abstract key-value pair based communication, which capture the essential communication characteristics of Hadoop-like big data computing. From the perspective of programming, many big data computing systems (e.g., Hadoop MapReduce, S4, or HBase) choose key-value pair as the core data representation structure, which is simple but has a strong ability to carry rich information. Therefore, it is a good idea to provide key-value pair based communication interfaces for big data computing systems and applications but not the traditional buffer-to-buffer interface signature. Such high level interface abstraction can reduce much programming complexity in parallel Big Data applications.

Fig. 3 presents the two-layer architecture of DataMPI. In the JVM layer, DataMPI extends the mpiJava design

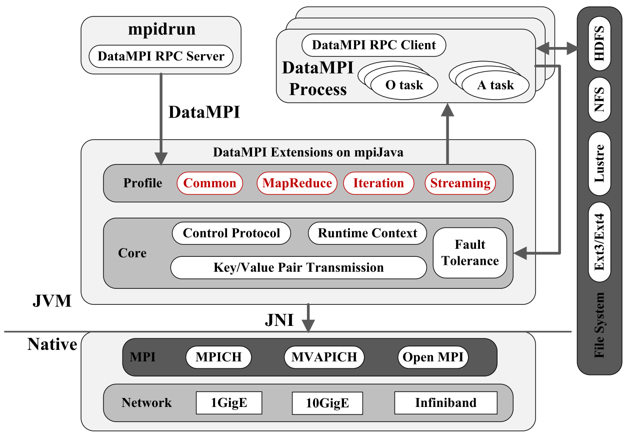


Fig. 3. ***Architecture of DataMPI. Adaptation with the permission of Lu et al.* [63].**

by dynamic process management in the Java layer, optimized buffer management by native direct IO, the implementation of DataMPI specification and so on. The lower layer is the native layer, in which JNI is utilized to connect upper Java-based routines to native MPI libraries. Compared with Hadoop, DataMPI provides a more light-weight and flexible library to users.

Liang *et al.* [58] use BigDataBench, a big data benchmark suite we will introduce in Section VI, to do comprehensive studies on performance and resource utilization characterizations of Hadoop, Spark and DataMPI. From the experiments, authors observe that DataMPI outperforms Hadoop and Spark significantly. Most of the benefits come from the high-efficiency communication mechanisms in DataMPI.

E. Energy Efficiency Optimization

Energy efficiency is an important topic for data centers. Globally, data centers are estimated to consume about US$30 billion worth of electricity per year [71]. Power and cooling costs more than the IT equipment it supports and make up about 42% of the data centers operating costs [65]. There are many publications for data center energy management. However, energy for MapReduce clusters has not attracted enough attention. It will be a promising future research direction.

There are now broadly two ways to reduce the energy cost of MapReduce clusters: one is to power down the low utilization nodes [54], [56]; the other is to match the hardware resources for the workload characteristics, such as the CPU computing ability [65], [79]. Both ways are a tradeoff between performance and energy.

1) Power Down Nodes: The motivation of [54] and [56] is that the average CPU utilization for clusters is low. These two papers go to two directions. Reference [54] uses a subset of the nodes for the MapReduce job and powers down the others, while [56] uses all the nodes to finish the job first, then powers down all the nodes after the job finishes. Lang and Patel [54] define a covering subset (CS), and change the HDFS (Hadoops file system) to make sure that at least one replica of each data block is in the covering subset. Then the nodes not in the set can be all disabled without affecting the availability of the data and the execution of the job. The results show that disabling nodes in all cases reduces energy cost and the reduction amount depends on the application. The results also show a running time and energy tradeoff tendency while the number of disabled nodes increases. Leverich and Kozyrakis [56] tries to avoid pitfalls of [54], namely HDFS modification, running time increase and storage overprovision. It proposes a technique called All-In Strategy (AIS) that starts all the nodes to finish the job as fast as possible and then powers down the entire system when there is no job. So the response time degradation of a job is predictable, based on the time to power up the hardware and OS. Results show that for long and computation complex MapReduce jobs, AIS outperforms than CS in response time and energy saving.

2) Match Hardware: Mashayekhy *et al.* [65] propose energy-aware MapReduce scheduling algorithms while satisfying the service level agreement instead of minimizing running time. It utilizes the truth that same Map or Reduce tasks consume variant energy and running time on different nodes of the underlying cluster. As many jobs need to run periodically, such as spam detection, the algorithms first profile the tasks' energy and running time cost on different kinds of nodes, then construct an optimized schedule for future execution. The scheduler will pick the nodes with low energy cost as well as satisfying the service level agreement to finish the jobs. Results show that the algorithms can obtain near optimal solutions with significant energy savings compared to schedulers aiming at minimizing the running time. Wirtz and Ge [79] adjust the processor frequency based on the jobs' computation needs. It compares three policies: 1) same frequency for all processors; 2) maximum processor frequency inside the map and Reduce functions and minimum frequency otherwise, so the computations will use fast cores and I/O will use slow ones; and 3) frequency set to bound the performance loss within a value user specified. This setting uses CPUMiser on each node. CPUMiser will collect performance counter information for CPU activity and then adjust the frequency accordingly. Results show that intelligent frequency setting can improve the energy efficiency, however, the level of improvements depend on the workload characteristic.

A recent work [40] evaluates the performance and power footprint of Hadoop on both physical and virtual clusters, considering the traditional Hadoop model collocating data and computing, and the alternative model separating the two. Results show performance and energy degradation when data and computation are separated.

Energy efficiency analysis of MapReduce is still in the early stages of investigation with many open issues remaining. The proposed algorithms are not mature enough to benefit both data and compute intensive workloads.

IV. BATCH PROCESSING SYSTEMS

Batch processing can group the operations on multiple data items for the concern of throughput. In addition to MapReduce, this section will discuss other popular batch processing systems in both programming models: user-defined methods and SQL-like languages.

A. General-Purpose Systems With User-Defined Methods

These systems follow MapReduce's idea for general-purpose usage, but with more flexible dataflow and programming models. We will introduce Dryad, Nephele/PACT, and Spark. They are popular systems based on which many other systems are built. For example, Apache Tez and Flink are basically open-source implementations of Dryad and Nephele/PACT, respectively. This subsection will end up with a discussion and comparison among the general-purpose batch processing paradigms.

1) Dryad and Apache Tez: Dryad [50] is a general-purpose data parallel programming model or distributed systems developed by Microsoft. Apache Tez is implemented based on Dryad's paradigm. Other projects employ Dryad as an execution engine include SCOPE, DryadLINQ, and Apache Hama.

Dryad models an application as a DAG (Directed Acyclic Graph), of which vertices and directed edges represent the computation and communication, respectively. Dryad automatically maps each vertex on an available computer and provides communication channels through TCP pipes, files or shard-memory FIFOs.

Compared with MapReduce, Dryad sacrifices some model simplicity but allows more flexible dataflow. It gives fine control over the communication graph and the subroutines in the vertices. Developers can specify arbitrary communication DAG rather than a requisite sequence of map/reduce operations. The data transport mechanisms can be selected too, which delivers substantial performance gains for suitable tasks. In particular, the Dryad vertices can have an arbitrary number of inputs and outputs while MapReduce only supports single input set and generates single output set.

A DAG in Dryad is represented as a graph, $G = \langle V_G, E_G, I_G, O_G \rangle$. V_G contains all of the computational vertices. E_G is the set of all edges. The vertices in I_G or O_G are the input or output nodes, namely, and there are no directed edges entering or leaving them. The basic

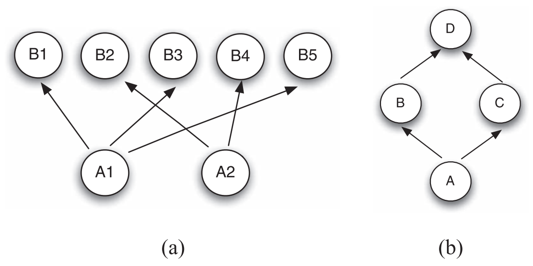


Fig. 4. ***Examples of graph merging operators. (a)*** $A >= B$, $|O_A| = 2$, $|I_B| = 5$. ***(b)*** $(A >= B >= D)\|(A >= C >= D)$.

graph, $G = \langle\{v\}, \emptyset, \{v\}, \{v\}\rangle$, called a singleton graph, contains one vertex.

Developers could define the communication pattern (DAG) by three graph-merging methods, which is more flexible than MapReduce and other models.

The first operator is $>=$, $A >= B = \langle V_A \oplus V_B, E_A \cup E_B \cup E_{\text{new}}, I_A, O_B\rangle$, $\oplus$ unites all of the vertices in A and B. E_{new} contains directed edges from O_A to I_B in a round-robin style. Fig. 4(a) shows an example where $|O_A| < |I_B|$.

The second operator is $\gg$. It's similar to $>=$ except that the E_{new} contains all of the directed edges from every vertex in O_A to every vertex in I_B. Thus, E_{new} is actually a complete bipartite graph.

The final operator is $\|$, $A\|B = \langle V_A \oplus^* V_B, E_A \cup E_B, I_A \cup^* I_B, O_A \cup^* O_B\rangle$. $\oplus^*$ is similar to $\oplus$ with duplicated vertices deleted from B, but $\cup^*$ means the directed edges are merged. Fig. 4(b) shows an example. The input vertex A and output vertex D in the second graph $(A >= C >= D)$ are deleted. However, the related directed edges are merged to the corresponding vertex in the first graph.

Dryad runtime can restructure the communication graph to avoid the data aggregation bottleneck. Fig. 5 shows an example. In the left subfigure, all input vertices labeled as A_i transfer data to a single receiver B. This scheme wastes the real network topology where there are hierarchical layers. Thus, Dryad can dynamicly refine the communication graph by adding one internal layer C, as shown in the right figure. Each vertex in the internal layer receives a subset of A, and B only processes the data transferred from C_i.

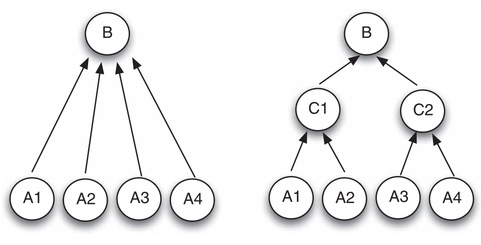


Fig. 5. ***Dynamic optimization.***

Dryad can be used as a general-purpose high-level programming model. Moreover, it can also be used as a platform on which some domain-specified languages develop to provide easier programming interfaces. For example, Nebula scripting language, designed on top of Dryad, allows developers to express a task as a number of execution stages. It transfers Dryad programs to a generalization of the UNIX pipe. Existing executable files can be assembled to any DAG.

DryadLINQ [82] is a system that allows the developers to code.Net programs on large scale distributed computing platforms. DryadLINQ expands the LINQ programming language with data-parallel operators and datatypes. It compiles.Net LINQ programs to distributed plans and employs the Dryad distributed runtime system to schedule these plans. SCOPE will be introduced in the next section. Apache Hadoop community has implemented Tez inspired by Dryad's DAG concept. Tez [11] is built atop YARN [13], the new generation of Hadoop with a complete overhaul of MapReduce.

2) Nephele/PACT and Apache Flink: Nephele/PACT system [1], [16] is a parallel data-processing system which is composed of two components: a programming model named Parallelization Contracts (PACTs) and a scalable parallel execution engine named Nephele. It is the most important part of Stratosphere [2], an open-source software stack for parallel data analysis. Apache Flink [8] also originates from Stratosphere project. It is an open-source implementation based on the PACT model.

Similar to Dryad, PACT also supports more flexible dataflows than MapReduce. Each mapper can have distinct inputs and user can specify complex operations besides Map and Reduce. Nephele/PACT is highlighted by the following properties:

- A richer programming model than MapReduce. PACTs is a generalization of MapReduce but based on Input and Output Contract. Input Contract is a second-order function which allows developers to express complex data analytical operations naturally and parallelize them independently. Output Contract is an optional component which annotates properties of first-order functions and enable certain optimizations.
- A separation of the programming model from the concrete execution strategy. Data-processing tasks implemented as PACT programs can be executed in several ways. A compiler determines the most efficient execution plan and translates it into a parallel data flow program modeled as directed acyclic graphs(DAGs).
- A flexible execution engine. Nephele executes DAGs which is generated by the complier in a parallel and fault-tolerant way. PACT compiler can influence the execution in a very fine-grained manner.

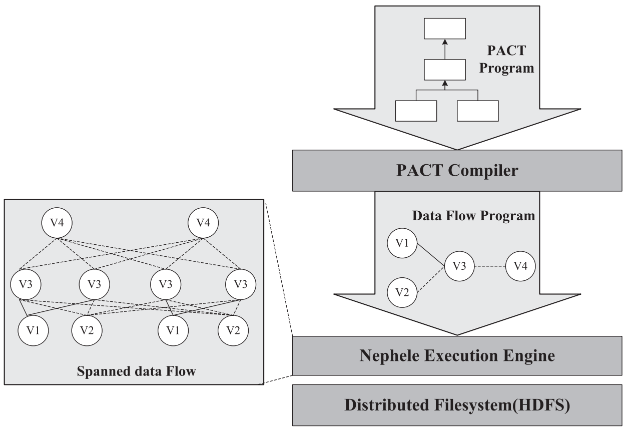


Fig. 6. ***Architecture of Nephele/PACT. Adaptation with the permission of Alexandrov et al.*** **[1].**

Fig. 6 shows the three-layer architecture of Nephele/PACT. When a PACT program is submitted to the *PACT Compiler*, the compiler will translate it to a dataflow program and hand it to the Nephele execution engine for parallel execution. The engine will generate the parallel dataflow graph by spanning the received DAG and then execute it. Input and output data are stored in the distributed file system HDFS. We will next introduce the PACT model, compiler, and execution engine of this system.

The PACT Programming Model The programming model is based on Parallelization Contracts (PACTs), which consists of one second-order function called *Input Contract* and an optional *Output Contract*. PACT programs are implemented by providing task-specific user code (the user functions, UFs) for selected PACTs and are assembled to a work flow. Input Contract defines how the UF can be evaluated in parallel. Output Contract is an optional component, which allows the optimizer to infer certain properties of the output data of a UF and hence to create a more efficient execution strategy for a program.

The Complier and Execution Engine PACT program can be compiled to a Nephele DAG by PACT compiler. For a single PACT program, PACT compiler can choose the most efficient execution plan from several execution plans with varying costs because of the declarative character of PACT.

After Nephele DAG is generated, the parallel execution engine will execute the compiled PACT program in a parallel fashion. One thing should be noted, that is the initial DAG cannot reflect parallel execution. Nephele will generate the parallel data flow graph before execution through spanning the received DAG. During spanning, vertices are multiplied to the desired degree of parallelism and connection patterns that are attached to channels define how the multiplied vertices are rewired. Currently, Nephele supports three types of communication channels: network, in-memory, and file channels. Network and in-memory channels allow the PACT compiler to construct low-latency execution pipelines. File channels collect the entire output of a task in a temporary file which can be considered as check points.

Nephele will take care of resource scheduling, task distribution, communication as well as synchronization issues. Nephele's fault-tolerance mechanisms will help to mitigate the impact of hardware outages. Unlike existing systems, Nephele offers to annotate jobs with a rich set of parameters, which will influence the physical execution. During task execution, Nephele will monitor the involved compute nodes and record statistics on the utilization of CPU and network. A graphical interface will display these statistics information. Hence, immediate visual feedback on the efficiency of the parallel execution strategies chosen by the PACT compiler is provided. Furthermore, performance bottlenecks resulting from network congestions or inappropriate degrees of parallelism can also be detected.

Stratosphere is an outcome of a research project whose goal is massive parallel execution of tasks on large data in a cloud environment. This project investigates "Information Management on the Cloud" and includes universities from the area of Berlin, namely, TU Berlin, Humboldt University and the Hasso Plattner Institute. Besides Nephele/PACT, the other part of Stratosphere is Sopremo [45], a semantically rich operator model. Programs for the Sopremo can be written in Meteor [45], an operator-oriented query language. Once a Meter script has been submitted to Stratosphere, the Sorpremo translates the script into an operator plan. Sorpremo plans will be compiled to PACT programs by a program assembler, as the input to the Nephele/PACT system.

3) Spark: As discussed in Section III-A, MapReduce does not support iterative data processing. That causes performance penalties for applications like data mining and graph analysis. Spark [83] is a cluster computing framework designed for iterative dataflow. Instead of designing specialized systems for different applications, one of Spark's purpose is unification. Spark is easy to be extended with different libraries. Current libraries include Streaming, GraphX, MLbase, etc. Spark is one of the most active projects in the big data field.

Spark is written in Scala, which is a statically typed high-level programming language for the Java VM. The critical abstraction in Spark is the resilient distributed dataset (RDD). RDD contains read-only Scala objects which is stored in a distributed system. It is resilient since the objects can be rebuilt even the partition information is lost. RDD is based on coarse-grained updates rather than fine-grained. An RDD can be explicitly cached in memory across machines and be reused in multiple MapReduce-like parallel operations.

Developers can build RDDs in four approaches. The first way is to construct an RDD from a file in a shard

Table 2 Comparisons of General-Purpose Big Data-Processing Systems

	MapReduce	Dryad	Spark	Nephele/PACTs
Programming Abstraction	Map and Reduce Functions	User-defined Communication DAG	RDD	Parallelization Contracts
Operators	Map and Reduce Operators	Three Kinds of Graph Operators	Operators on RDD	UF on Contracts
Execution Dataflow	Map and Reduce Parses	DAG	DAG of RDD operations	Nephele DAG
Aware of Input Partitioner	No	No	Yes	Yes
Result Caching	No	No	Yes	No
Fault Tolerance Mechanism	Task-grained	Vertex-grained	RDD-grained	Task-grained

file system, such as HDFS (Hadoop Distributed File System). The second way parallelizes an existing Scala collection and distributes them to multiple nodes. The other two ways are to transform the type or change the persistence of an existing RDD. Spark provides the flatMap operation to change the RDD type. It can also use more flexible filters to select specified data. Two actions, *cache* and *save*, are provided to identify the persistence of an RDD. The *cache* means the data should be kept in memory since it will be reused. The *save* action writes the data in a file system such as HDFS.

To use Spark, users define a driver program as a manager. The driver program also customizes RDDs and launches the actions associated with them. To represent RDDs, developers use a set of information: partitions, which are the subsets of the dataset; dependencies, which indicate the relations to parent RDDs; a computational function; and information about partitioning scheme and data position. There are two kinds of dependencies: narrow and wide. For narrow dependency, each parent RDD is used by at most one child RDD. For wide dependency, each parent RDD can be used multiple times.

Spark supports several parallel operations: *reduce*, *collect*, and *foreach*. The *reduce* operation calculates a value by combining the data through an associative function, such as *add* and *minimum*. The *collect* operation sends data to the driver program. The *foreach* operation calls the user defined function with each element. Users can call other operates, such as *map*, *filter*, and *reduce* operations with specified functions. This is similar to functional programming.

The scheduling scheme is similar to Dryad, but it assigns tasks to machines based on the data locality information. Like the owner computing, the task is sent to the node on which its data is kept. The runtime system uses an LRU replacement policy to manage RDDs. The least recently used RDD is evicted to store a new generated one.

Based on the fundamental Spark components, there are many libraries for specific fields. Spark SQL supports the SQL queries in a Spark application. Spark Streaming provides APIs for streaming processing. MLlib and GraphX are frameworks for machine learning and graph processing, respectively.

4) Comparisons: Here, we will give a comparison of the above general-purpose big data-processing systems. The comparison is summarized in Table 2. We mainly compare these systems on six aspects from the design of the programming model to the specific execution strategy.

MapReduce introduces a simple programming model with only two functions for developers. This programming model hides many implementation details from application programmers. For complex workloads, such as data mining and online analytical processing (OLAP), however, developers need to implement specific operations by writing Map and Reduce functions, which may lead to bad performance and hardly reusable codes. The programming model of Dryad gives a higher level abstraction, which presents each application as a serial of operations on a DAG. Spark and PACTs introduce more flexible abstractions based on the original DAG concept. Such abstractions with flexible and user defined operators can provide developers with more convenient APIs and make them focus more on the design of workload procedures.

For execution strategy, we mainly demonstrate the differences on four aspects: execution dataflow, whether being aware of the input partition mechanism, result caching support and the fault tolerance mechanism. Except for MapReduce, all of the other systems build the dataflow based on DAG. DAG-based dataflows can provide more detailed information on the dependencies of operators, which can help the engine to better optimize the execution and make better scheduling choices. Systems aware of how the input data are partitioned can abandon unnecessary repartition operations when the partition mechanism of output data is the same as inputs. Caching the results can significantly improve computing performance when applications have iterative operations, thus Spark can perform much better on iterative machine learning workloads than other systems.

Fault tolerance has become an unavoidable issue when designing a big data processing system, as the scale of machines increases and faults become regular. MapReduce and Nephele provide a task-grained fault tolerance mechanism, which means they can be aware when a task fails, and recompute this task. Spark uses a linear mechanism to record the dependency of every RDD. If an RDD fails, Spark will recompute it from the parent RDD it depends, so its fault tolerance mechanism is RDD-grained. Similar to Spark, Dryad uses a fault tolerance mechanism on the level of vertex.

B. SQL-Like Processing Systems

The big data systems mentioned above always provide users with low-level operations which force users to implement even simple functions such as *selection* and *projection*. Moreover, the customized codes can be hardly reusable and optimized.

SQL is a declarative language widely used in database management systems. It is very useful for data analysis and data mining, and also easy to learn for data scientists. Many researches have been done to support SQL or SQL-like languages for massive data analysis, including building SQL parsers on available general big data processing systems or building new parallel database systems.

1) SQL on General Execution Engine

Hive: Hive [75] is a scalable data warehouse built by Facebook to support massive data analysis. Hive adopts a SQL-like language called HQL for users to write queries. The engine is Hadoop.

Hive consists of a metastore, a Hive driver, a query compiler, interface components (Command Line Interface, HiveServer and User defined function interface) and MapReduce engine. Hive stores data in tables, and the table is defined as a directory in HDFS. Tables are also separated into buckets. The metastore stores the system catalog and metadata about tables, columns and partitions. A statement is submitted from one interface component to the Hive driver, which manages the whole lifecycle of the statement. The statement is first translated into an abstract syntax tree (AST), and converted into an operator tree through type checking and semantic analysis. In Hive, an operator represents a specific data operation. For instance, a TableScanOperator is used for loading tables from HDFS, and a SelectOperator is used for projection operation. The operator tree is then optimized and turned into a physical plan composed of multiple MapReduce and HDFS tasks. This physical plan is submitted to the underlying MapReduce engine for execution.

Hive can reliably scale to petabytes, thus is widely used to build data store. However, Hive is not suitable for interactive query, due to the weakness of MapReduce framework on query processing. A Hive query request needs to be translated into multiple MapReduce jobs for execution. The intermediate data between each jobs are stored in the HDFS, which significantly increases the overhead of disk I/O.

Shark: Shark [80] is a data analysis system towards low latency and fine grained fault tolerance. It is built on top of Spark, thus to benefit from the speed of in-memory computing. Similar to Hive, Shark also implements a SQL parser on a scalable engine. To achieve lower latency, Shark also supports the features of caching tables in memory, dynamic query plan optimization and memory-based shuffle. Moreover, a SQL query can be executed as a series of operations on one RDD in Spark, so the intermediate data can be kept in memory during the execution.

SCOPE: Structured Computations Optimized for Parallel Execution (SCOPE) [23] is a SQL-like declarative and extensible scripting language developed by Microsoft, which aims at web-scale data analysis. SCOPE extends SQL with C♯ expressions to support more complex analysis applications and customized operations. To support extensibility, SCOPE also provides interfaces for users to define their own functions and new implementations of existing operators, including extractors, processors, reducers, and combiners.

SCOPE uses the Cosmos distributed computing platform as the storage system and low level execution engine. The Cosmos storage system is a distributed file system with all writes to be append-only, and concurrent writers to be serialized. The Cosmos execution environment provides low level primitives and higher-level DAG-like programming interface.

A SCOPE script consists of commands. The compiler first translates the script into a parse tree, and then generates a data flow DAG, where each vertex is a program and each edge represents a data channel. A vertex is a serial program consisting of SCOPE physical operators, which can be executed in a pipelined way in the Cosmos system. The compiler also includes optimizations to generate a physical plan with the lowest estimated cost to achieve good performance.

2) Parallel Database Systems

AsterixDB: AsterixDB [3], [17] is a research project involving researchers at UC Irvine as well as UC San Diego and UC Riverside, which aims to build a scalable information management system with support for the storage, querying, and analysis of huge collections of semi-structured nested data objects, with a new declarative query language (AQL).

The user model of AsterixDB consists of two core components: the Asterix Data Model (ADM) and the query language (AQL) targeting semi-structured data. ADM supports a wide variety of semi-structured data formats, and performs AsterixDB data storage and query processing. Each individual ADM data instance is typed, self-describing and stored in a dataset. The datasets of ADM can be indexed, partitioned over multiple hosts in a cluster, and replicated to achieve scalability or availability. Datasets may have associated schema information that describes the core content of their instances.

Asterix Query Language (AQL) is the query language for AsterixDB, which is used to access and manipulate Asterix Data. AQL borrows from XQuery and Jaql the programmer-friendly declarative syntax and is comparable to those languages in terms of expressive power. AQL is designed to cleanly match and handle the data structuring constructs of ADM. So it omits many XML-specific and document-specific features.

AsterixDB uses a scalable parallel engine called Hyracks [20] to process queries. AQL queries are compiled into Hyracks Jobs for execution, which are in the form of DAGs consisted of Operators and Connectors. Each operator presents a AQL operation and is responsible for loading partitions of input data and producing output data partitions, while each connector redistributes the output partitions and prepares input partitions for the next Operator.

Operators in Hyracks have a three-part specification, given here.

1) *Operator Descriptors*. Every operator is constructed as an implementation of the Operator Descriptor interface.
2) *Operator Activities*. Hyracks allows an operator to describe the various phases involved in its evaluation in a high level.
3) *Operator Tasks*. Each activity of an operator actually represents a set of parallel tasks to be scheduled on the machines in the cluster. Hyracks also comes with a library of pre-existing operators and connectors, for example, *File Readers/Writers* and *Mappers* operators and *M:N Hash-Partitioner* connectors.

Hyracks programming model has two broad classes of users: end users who use Hyracks as a job assembly layer to solve problems, and operator implementors who wish to implement new operators for use by end users. Correspondingly, Hyracks programming model is composed of end user model and operator implementor model.

For end user model, Hyracks native end user model not only supports Hyracks job but also supports AQL, as well as higher level programming languages such as Hive and Pig. The model expresses a Hyracks job as a DAG of operator descriptors connected to one another by connector descriptors. Hyracks is also compatible to the Hadoop software stack with a MapReduce user model, which enables MapReduce jobs developed for Hadoop run on the Hyracks platform without modification. This is mainly achieved through developing two extra operator descriptors that can wrap the Map and Reduce functionality, named *hadoop_mapper* and *hadoop_reducer*, respectively.

Hyracks includes two layers for job execution: Control Layer and Task Execution Layer. The control layer contains two controllers: Cluster Controller (CC) and Node Controller (NC). These two Controllers have their own different functions: Cluster Controller (CC) manages the Hyracks cluster through performing five functions: 1) accepting job execution requests from clients; 2) planning evaluation strategies of these requests; 3) scheduling jobs' tasks to run on the selected machines in the cluster; 4) monitoring the state of the cluster; and 5) re-planning and re-executing some or all of the tasks of a job in the event of a failure. During the task execution, each worker machine runs a Node Controller (NC) process which accepts task execution requests from the CC and reports on its health via a heartbeat mechanism.

Dremel: Most of the data generated today is non-relational. Moreover, most of the data structures are represented in a nested way, as they come from the structures of programming languages, messages of the distributed systems, and structured documents. Dremel [66] is a scalable and interactive ad-hoc query system for native read-only nested data analysis. Systems based on Dremel's paradigm include Cloudera Impala and Apache Drill. Dremel borrows the idea of serving trees from distributed search engines, and provides a high-level, SQL-like language to express *ad hoc* queries. The query engine executes queries natively instead of translating them into MapReduce jobs, which is different from MapReduce-based database systems.

The data model of Dremel is based on strongly-typed nested records, and originated from protocol buffers, which is widely used in the context of distributed systems at Google. The abstract syntax is as follows:

$$\tau = \mathbf{dom}|\left\langle A^1 : \tau[*|?], \ldots, A^n : \tau[*|?]\right\rangle$$

where τ is an atomic or a record type of dom. Atomic types include integers, floating-point numbers, strings, etc, while records contain one or multiple fields. A^i is the name of Field i. The repeated fields $(*)$ can appear multiple times, and the optional fields (?) may be missing from the record. Such nested data model has good cross-language interoperability and provides a platform-neutral way for structured data serializing.

Dremel uses a columnar storage format for nested data. To distinguish values of repeated fields, Dremel introduces the concepts of repetition and definition levels. The repetition level of a value records at what repeated field in the field's path the value has repeated, or the level up to which the nested record exists requires extra information. The definition level is used to specify the number of fields in the path of a NULL value that can be undefined are actually present. With such two levels, Dremel can decide which fields in the path of a value is required, and omit the optional or repeated fields, for effectively compressing.

Dremel uses a multi-level serving tree architecture for query execution. The computing servers are divided into one root server, multiple leaf servers and intermediate servers. The root server is responsible for receiving queries, reading and storing metadata, and routing the queries to the next level of the tree. When a query comes in, the root server first determines all tables required, then rewrites the query with the partition information. The intermediate query is then sent to the next level of the serving tree for other rewriting, which decides the partitions that should be processed by this server. When the query reaches the leaf servers, which communicate with the storage layer, the required tables are scanned in parallel, and the results are sent to the upper level for further computation, until the root server gets the final result.

This execution model can be well-suited for interactive queries, since the aggregation queries often return small- or medium-sized results. The columnar storage format also helps Dremel to achieve a high throughput for scan operations, which leads to low latency query execution.

V. STREAM, GRAPH, AND MACHINE LEARNING PROCESSING SYSTEMS

This section is about the other three categories of big data processing systems. Although general-purpose systems can also be used for these applications, specific systems can leverage domain features more effectively, so efficiency, programmability, and correctness are normally improved. Storm and S4 will be discussed as representatives of stream processing. Graph systems will introduce GraphLab and Pregel, which can also be used for ML problems. Petuum and MLbase are specially designed for ML algorithms.

A. Stream Processing Systems

Data stream applications like the real-time search and social networks need scalable stream processing systems operating at high data rates instead of the long-latency batch processing of MapReduce-like systems. This section will introduce Storm and S4 systems specific for stream processing.

1) Storm: Storm [76] is an open-sourced distributed stream processing system developed by Twitter. It is easy to process unbounded streams of data, like a hundred million tweets per day. For real-time processing and continuous computation, Storm runs more efficiently than batch processing. More than 60 companies are using or experimenting with Storm.

The Storm model consists *streams of tuples* flowing through *topologies* defined by users as a Thrift [12] object. Thrift's cross language services make sure that any languages can be used to create a topology. The vertices of a topology represent computations and edges are data flow. There are two kinds of vertices, *spouts* and *bolts*. Spouts are the source of data stream, and bolts process the tuples and pass them to the downstream bolts. The topology can have cycles.

In the Storm system, there are three kinds of nodes: the master node called *Nimbus*, the *worker* nodes, and ZooKeeper [14]. Nimbus is responsible for distributing and coordinating the execution of topology.

The *worker* nodes execute the *spouts/bolts* and also run a *Supervisor* to communicate with the Nimbus. Supervisors contact Nimbus in a periodic heartbeat protocol informing the topologies running on the node or the vacancies available. The cluster state is maintained in Zookeeper which is an Apache open-source server for highly reliable distributed coordination. It provides services like maintaining configuration information and distributed synchronization.

Storm provides tuple processing guarantees. Each tuple will be generated a message id. After the tuple leaves the topology, a backflow mechanism will inform the spout that started this tuple to make sure every tuple is processed if needed.

2) S4: Different from Storm, S4 [68] uses a decentralized and symmetric architecture for simplicity. There is no central node in the S4 system. S4 design is derived from a combination of MapReduce and the Actors model.

A stream of S4 is defined as a sequence of events in the form of (K, A) where K is a tuple-valued key, and A is the attribute, for example, a (word, count) event in a word appearance counting problem. The events are transmitted through Processing Elements (PEs) which do the computations written by developers. A PE is distinguished from other PEs by four components: its functionality defined, the types of events it consumes, the keyed attribute in those events, and the value of the attribute. PE is defined by users and S4 system initiates one PE per each unique key in the stream. The state of a PE is not accessible by other PEs.

The PEs run in logical hosts, called Processing Nodes (PNs). PNs will be mapped to physical nodes by the communication layer. The PNs are responsible for listening the events, executing operations on the events, dispatching events, and emitting output events. The communication layer uses ZooKeeper too to coordinate between nodes. The main work of this layer is to manage cluster, recover failure, and map nodes.

S4's fault tolerance is through checkpoints. It does not support guaranteed processing for tuples like what Storm does.

B. Graph Processing Systems

To specifically support large-scale graph computations on clusters, parallel systems, such as GraphLab [62] and Pregel [64], are proposed. They execute graph models and data parallely. Some of ML algorithms also focus on the dependencies of data. It is natural to use graph structures to abstract those dependencies. So they are essentially graph-structured computations too.

1) GraphLab: GraphLab [62] adopts a vertex-centric model and computations will run on each vertex. The abstraction of GraphLab includes the data graph, update function and sync operation. The data graph manages user-defined data, including model parameters, algorithm state and statistical data. Update functions are user-defined computations modifying the data of a vertex and its adjacent vertices and edges in the data graph. Those functions will return the modified data and the vertices that need to be modified by the update functions in the future iterations. Sync operations are used to maintain global statistics describing data stored in the data graph, for example, global convergence estimators.

Take the popular PageRank problem [21], which recursively defines the rank of a webpage, as an example, each vertex of the data graph represents a webpage storing the rank and each edge represents a link storing the weight of the link. The update function will calculate the rank of a vertex according to the weighted links to its neighbor vertices. The neighbor vertices will be scheduled to the queue waiting for future update if the current vertex changes by more than a threshold.

For storage, the data graph will be partitioned according to domain specific knowledge or some graph partition heuristics. Each partition will be a separate file on a distributed storage system, such as HDFS. There will be a meta-graph storing the connectivity structure and file locations of those partitions. According to the meta-graph and the number of physical machines, a fast balanced distributed loading can be performed. Then Vertices will be executed simultaneously on clusters. The GraphLab execution engine supports fully asynchronous execution for vertices and also supports vertex priorities. It requires the entire graph and program state to reside in RAM. Each vertex is associated with a reader and a writer lock. Each machine only runs updates on local vertices after completing lock acquisition and data synchronization. The acquisition and synchronization are pipelined for different vertices on each machine to reduce latency.

The fault tolerance of GraphLab is implemented using distributed checkpoint mechanism. The mechanism is designed fully asynchronously based on the Chandy-Lamport snapshot. It incrementally constructs a snapshot without suspending execution. In the event of a failure, the system will be recovered from the last checkpoint.

GraphLab has nature support for several ML algorithm properties. It supports: 1) graph parallel for expressing data dependencies of ML algorithm; 2) asynchronous iterative computation for fast convergence; 3) dynamic computation for prioritizing computations on parameters requiring more iterations to converge; and 4) serializability for ensuring that all parallel executions have equivalent sequential execution to allow ML experts focus on algorithm design.

PowerGraph [43] is the subsequent version of GraphLab. It can efficiently process natural graphs or graphs with power-law distribution of connectivity. Those graphs can cause load imbalance problems due to that a few popular vertices can be with many edges, while a large number of unpopular vertices are actually with few edges.

2) Pregel: The purpose of Pregel [64] is to build a system which can implement arbitrary graph algorithms in a large-scale distributed environment with fault tolerance. We will introduce Pregel through comparisons with GraphLab.

Similar to GraphLab, Pregel also uses the vertex-centric approach. Each vertex is associated with a user-defined value. Each edge is associated with a value, a source, and a target vertex identifier. User-defined functions can change the state of vertices and edges.

There are two outstanding differences between GraphLab and Pregel. First, the update functions for GraphLab can access a vertex, its connected edges and its neighbor vertices. However, The Pregel functions can only access a vertex and its edges. The neighbors state will be sent to the vertex through messages. The function can receive messages sent to a vertex in the previous iteration, modify the vertex state and its outgoing edges, send messages to other vertices that will be received in next iteration and even mutate graph topology. Second, the graph and ML computation consist of a sequence of iterations. GraghLab supports the asynchronous iteration, while Pregel uses the barrier synchronization. Pregel is inspired by Valiant's Bulk Synchronous Parallel (BSP) model. During an iteration, the framework invoke the same user-defined function to each vertex and execute them in parallel, and then the computation will be synchronized after each iteration such as the increment of the global iteration index. Thus, the model does not need to consider data race or deadlock problems.

The fault tolerance of Pregel is also implemented by checkpointing. At the beginning of each iteration, the worker machines will save the state of their partitions to persistent storage and the master machine will save the aggregator values.

Apache Hama [9] is an open source project inspired by Pregel. Hama realizes BSP over the HDFS, as well as the Dryad engine from Microsoft.

C. Machine Learning Processing Systems

To explore the value of big data, and increase the accuracy of ML models, the scale of ML problems are increasing exponentially which is so called "Big Model on Big Data" [81]. Thus, scalability is one of the bottlenecks for ML research [36]. That is highly challenging because to design, implement, and debug distributed ML systems require ML experts to address cluster programming details, such as race condition and deadlock, while the developing of complex math models and algorithms. Most of traditional parallel processing systems, such as MapReduce, do not have natural support for processing ML problems on clusters. To ease the application of ML algorithms to industrial-scale problems, many systems specifically for ML have been developed.

While many of ML problems can be represented as graphs, there are also algorithms like topic modeling which are not easy or inefficient to be represented in that way. Besides, the execution correctness of asynchronous graph-based model is unclear [81]. This subsection will introduce ML-centric systems.

Petuum: Petuum [81] is the state-of-the-art distributed ML framework, built on an ML-centric principle. It formalizes ML algorithms as iterative-convergent programs. Compared to GraphLab, Spark and other platforms which support partial solutions for ML problems, it can support a wider spectrum of ML algorithms. Besides, Petuum considers data parallel and model parallel execution of an ML program, which can exploit the three unique properties of ML algorithm: error tolerance, dynamic structural dependency and non-uniform convergence. So it converges more efficiently than other platforms [51].

The Petuum system comprises three components: scheduler, workers, and parameter server. The three user-defined functions provided are *schedule*, *push*, and *pull*. The scheduler enables model parallelism by controlling which parameters to update by each worker according to the user-defined *schedule* function. This allows ML programs to analyze structural dependency dynamically and select independent parameters for parallel updates in case of parallelization error and non-convergence. The *schedule* function also allows to consider the non-uniform convergence problem of ML through prioritizing parameters that needs more iterations to converge. GraphLab system has also considered this problem. Petuum uses pipelining to overlap the *schedule* with worker execution. The scheduler is also responsible for central aggregation through *pull* function if needed.

The workers in Petuum are responsible to receive the parameters and run parallel update function *push*. The parameters will automatically synchronize with the parameter sever using a distributed shared memory API.

The parameter server provides global access to parameters via the shared memory API. Based on the theory that ML algorithms are often tolerant to minor errors, the parameter server implements Stale Synchronous Parallel (SSP) consistency model. So network communication and synchronization overheads are reduced significantly while maintaining the convergence guarantees.

Petuum does not specify a data abstraction, so any data storage system may be used. Fault tolerance is achieved by checkpoint-and-restart, but it is currently suitable for up to hundreds of machines.

Another popular distributed ML platforms is MLbase [5]. MLbase consists of three components: MLlib, MLI, and ML Optimizer. MLlib has been mentioned in Section IV-A3. It is a distributed low-level ML library written against Spark runtime. It was part of the MLbase project and is now supported by the Spark community. The library includes common algorithms for classification, regression, and so on. MLI is atop MLlib, which introduces high-level ML programming abstractions. The highest layer is ML Optimizer. It solves a search problem over the algorithms included in MLI and MLlib for a best applicable model. Users specify ML problems by MLbase declarative task language.

Mahout [10] is an open-sourced scalable ML library. It includes an environment for building scalable algorithms, many ML algorithm implementations on Hadoop, and new algorithms on Spark too. Besides these open-sourced systems, companies like Amazon, Facebook, and Yahoo! all have their own in-house distributed ML systems.

VI. BIG DATA BENCHMARK

While more and more big data processing systems are proposed and deployed, it is important to fairly evaluate those systems both quantitatively and qualitatively. However, the development of standard big data benchmarks lags behind because of the complexity, diversity, and variability of workloads. Besides, the software stacks workloads built on cover a broad spectrum. Most internet companies tend to keep their data and applications confidential, preventing the benchmark building [78]. This section will first introduce the most recent and diverse benchmark suite BigDataBench and then discuss other suites targeting specific applications.

Wang *et al.* [78] state that big data benchmarking should have the following requirements:

- Measuring and comparing big data systems and architecture.
- Being data-centric.
- Diverse and representative workloads.
- Covering representative software stacks.
- State-of-the-art techniques.
- Usability.

To satisfy the requirements above, they present BigDataBench. It is currently the most diverse and representative big data benchmark suite compared to other proposed suites which are normally for specific applications or software stacks. They pay attention to investigating workloads in three most important application domains according to widely acceptable metrics—the number of page views and daily visitors, including search engine, e-commerce, and social network. To consider workload candidates, they make a tradeoff between choosing different types of applications: including online services, offline analytics, and real-time analytics. The

Table 3 The Summary of BigDataBench. Adaptation With the Permission of Wang *et al.* [78]

Application Scenarios	Application Type	Workloads	Data Types	Data Source	Software Stacks
Micro Benchmarks	Offline Analytics	Sort	Unstructured	Text	Hadoop, Spark, MPI
		Grep			
		WordCount			
		BFS		Graph	
Basic Datastore Operations	Online Service	Read	Semi-structured	Table	Hbase, Cassandra, MongoDB, MySQL
		Write			
		Scan			
Relational Query	Realtime Analytics	Select Query	Structured	Table	Impala, MySQL, Hive, Shark
		Aggregate Query			
		Join Query			
Search Engine	Online Services	Nutch Server	Un-structured	Text	Hadoop
	Offline Analytics	Index			
		PageRank		Graph	Hadoop, Spark, MPI
Social Network	Online Services	Olio Server	Un-structured	Graph	Apache+MySQL
	Offline Analytics	Kmeans			Hadoop, Spark, MPI
		Connected Components			
E-commerce	Online Services	Rubis Server	Structured	Table	Apache+JBoss+MySQL
	Offline Analytics	Collaborative Filtering	Semi-structured	Text	Hadoop, Spark, MPI
		Naive Bayes			

data types include structured, semi-structured, and unstructured data. The benchmarks are oriented to different analytics systems, such as Hadoop, Spark, MPI, Hive, Hbase, and MySQL. The summary of BigDataBench is presented in Table 3.

HiBench [49] is a benchmark suite for Hadoop MapReduce developed by researchers from Intel. This benchmark contains four categories of workloads: Micro Benchmarks (the Sort, WordCount and TeraSort workloads), Web Search (the Nutch Indexing and PageRank workloads), Machine Learning (the Bayesian Classification and K-means Clustering workloads), and HDFS Benchmark. The inputs of these workloads are either data sets of fixed size or scalable and synthetic data sets. The fixed-size data sets can either be directly extracted from real data sets (e.g., the Wikipedia page-to-page link database used by the PageRank workload, and a subset of the Wikipedia dump data set used by the Bayesian classification workload) or generated based on real data sets (e.g., in the Nutch Indexing workload, the input are the 2.4 million web pages generated based on an in-house Wikipedia). The synthetic data sets can either be randomly generated (e.g., the inputs used by the three workloads of Micro Benchmarks are randomly generated using the programs in the Hadoop distribution) or created using some statistic distributions (e.g., the input of the K-means Clustering workload).

There are other open-sourced big data benchmark suites. However, like HiBench, they are mainly for specific applications. Yahoo! Cloud Serving Benchmark (YCSB) [27] and LinkBench [15] include online service workloads for cloud computing and social network applications respectively. The big data benchmark from AMPLab [6] is for real-time analysis system. All of those benchmarks are not as diverse as BigDataBench.

VII. OPEN RESEARCH ISSUES

This paper has given a survey of current big data processing systems. Some possible future work directions will be discussed in this section.

Novel Data Model: Although there are mature big data storage systems now, as the scale of data keeps growing rapidly, and the raw data comes from more different sources, it is still a great challenge for data storage. The systems need to process data of different structures, and the cost of normalizing or formatting massive data will be very expensive and unacceptable. Therefore, a flexible data model should be very essential.

Novel Processing Model: The number of classes of big data workloads also keeps increasing. More and more parallel algorithms are developed to extract valuable information from big data. Such a situation introduces a great challenge for current general purpose big data systems, as they only consider the existing workloads when they are designed. Moreover, different classes of applications always have different requirement of resources, how to build a scalable system model with effective resource management should be a very important problem. Besides, the appearance of many big data processing systems aiming at specific types of applications makes it difficult for companies with different kinds of workloads to consolidate them on one cluster.

Big Data Benchmark: There are already some benchmark suites developed for big data, in which BigDataBench discussed in this paper is the state-of-the-art one. Benchmarks need to be developed according to the area of application too, in case of biasing specific systems. Researchers should expand the benchmark suite with more popular used applications, and promote the suite to be publicly used, which will benefit the research work of big data system optimization and new processing frameworks development.

High Performance Computing: Supercomputers offer high performance in terms of computation, memory access and communication. Classic parallel programming tools, such as MPI, OpenMP, CUDA, and OpenCL, have performance advantages over the traditional big data processing tools, such as MapReduce. Big data communities have already resorted to HPC for higher data processing efficiency and real-time performance, such as DataMPI. The conjuncture of HPC and big data will draw more attentions.

Energy Efficiency: Since energy is more and more an important concern for large computing clusters, minimizing the energy consumption for each big data job is a critical problem which hasn't been widely researched. This paper talked about some related work on energy efficiency optimizations based on MapReduce. Big data processing systems with natively supported energy saving features will be an interesting future direction.

Large-Scale Machine Learning: Distributed ML systems on clusters are critical to applying advanced ML algorithms at industrial scales for data analysis. The big ML models now can reach billions of parameters with massive amount of data. ML models have distinct features, such as computation dependencies and uneven convergence. We have discussed several representative systems for large-scale ML. However, this research direction is still in its early stage. More features of ML need to be discovered and supported. More importantly, common formalism for data/model parallelism should be established as a guidance for future system development.

Other directions like large-scale computation debugging, domain-specific data processing, and public available evaluation data, are all promising research areas, deserving further exploration and development.

VIII. CONCLUSION

Big data-processing systems have been widely researched by academia and industry. This paper gave a survey of those systems. Based on the processing paradigm, we categorize those systems into batch, stream, graph, and machine learning processing. The paper first introduced the basic framework of MapReduce and its outstanding features as well as deficiencies compared to DBMSs. According to the deficiencies, we discussed the extensions and optimizations for MapReduce platform, including support for flexible dataflows, efficient data access and communication, parameter tuning, as well as energy. We then surveyed other batch-processing systems, including general-purpose systems Dryad, Nephele/PACT and Spark. SQL-like systems involved in this paper are Hive, Shark, SCOPE, AsterixDB, and Dremel. For stream processing systems, Storm and S4 are introduced as representatives. Scalability is one of the ML algorithms bottlenecks. We then discussed how graph-centric systems like Pregel and GraphLab, and ML-centric systems like Petuum, parallelize the graph and ML model, as well as their distinctive characteristics. Future research opportunities are discussed at the end of the survey. ■

REFERENCES

[1] A. Alexandrov *et al.*, "Massively parallel data analysis with PACTs on Nephele," *PVLDB*, vol. 3, no. 2 pp. 1625–1628, 2010. doi: 10.14778/1920841.1921056.

[2] A. Alexandrov *et al.*, "The Stratosphere platform for big data analytics," *Int. J. Very Large Data Bases*, vol. 23, no. 6, pp. 939–964, 2014.

[3] S. Alsubaiee1 *et al.*, "Asterixdb: A scalable, open source bdms," *Proc. VLDB Endowment*, vol. 7, no. 14, 2014.

[4] P. Alvaro *et al.*, "Boom analytics: Exploring data-centric, declarative programming for the cloud," in *Proc. 5th Eur. Conf. Comput. Syst.*, Paris, France, Apr. 13–16, 2010, pp. 223–236, doi: 10.1145/1755913.1755937.

[5] AMPLab at UC Berkeley. MLbase, 2013. [Online]. Available: http://www.mlbase.org

[6] AMPLab, UC Berkeley. Big Data Benchmark, 2014. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[7] Apache. Hadoop, 2014. [Online]. Available: http://hadoop.apache.org

[8] Apache. Flink, 2015. [Online]. Available: http://flink.apache.org

[9] Apache. Hama, 2015. [Online]. Available: https://hama.apache.org

[10] Apache. Mahout, 2015. [Online]. Available: http://mahout.apache.org

[11] Apache. Tez, 2015. [Online]. Available: http://tez.apache.org

[12] Apache. Thrift, 2015. [Online]. Available: https://thrift.apache.org

[13] Apache. Yarn, 2015. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[14] Apache. Zookeeper, 2015. [Online]. Available: https://zookeeper.apache.org

[15] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," K. A. Ross, D. Srivastava, and D. Papadias, eds., in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 22–27, 2013, pp. 1185–1196, doi: 10.1145/2463676.2465296.

[16] D. Battré *et al.*, "Nephele/PACTs: A programming model and execution framework for web-scale analytical processing," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 10–11, 2010, Indianapolis, IN, USA, pp. 119–130, doi: 10.1145/1807128.1807148.

[17] A. Behm *et al.*, "ASTERIX: Towards a scalable, semistructured data platform for evolving-world models," *Distrib. Parallel Databases*, vol. 29, no. 3, pp. 185–216, 2011, doi: 10.1007/s10619-011-7082-y.

[18] B. Behzad *et al.*, "Taming parallel I/O complexity with auto-tuning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Denver, CO, USA, Nov. 17–21, 2013, p. 68, doi: 10.1145/2503210.2503278.

[19] B. Behzad, S. Byna, S. M. Wild, Prabhat, and M. Snir, "Improving parallel I/O autotuning with performance modeling," in *Proc. 23rd Int. Symp. HPDC*, Vancouver, BC, Canada, Jun. 23–27, 2014, pp. 253–256, doi: 10.1145/2600212.2600708.

[20] V. R. Borkar *et al.*, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. 27th Int. Conf. Data Eng.*, Hannover, Germany, Apr. 11–16, 2011, pp. 1151–1162, doi: 10.1109/ICDE.2011.5767921.

[21] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Networks*, vol. 30 no. 1–7, pp. 107–117, 1998, doi: 10.1016/S0169-7552(98)00110-X.

[22] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, pp. 285–296, 2010, doi: 10.14778/1920841.1920881.

[23] R. Chaiken *et al.*, "Scope: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008, doi: 10.14778/1454159.1454166.

[24] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in B. N. Bershad and J. C. Mogul, eds., *Proc. 7th Symp. Operating Syst. Design Implementation*, Seattle, WA, USA, Nov. 6–8, 2006, pp. 205–218, USENIX Association. [Online]. Available: http://www.usenix.org/events/osdi06/tech/chang.html

[25] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. 7th USENIX Symp. Netw. Syst. Design Implementation*, San Jose, CA, USA, Apr. 28–30, 2010, pp. 313–328. [Online]. Available: http://www.usenix.org/events/nsdi10/tech/full_papers/condie.pdf

[26] T. Condie *et al.*, "Online aggregation and continuous query support in MapReduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Indianapolis, IN, USA, Jun. 6–10, 2010, pp. 1115–1118, doi: 10.1145/1807167.1807295.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, eds., *Proc. 1st ACM Symp. Cloud Comput.*, Indianapolis, IN, USA, Jun. 10–11, 2010, pp. 143–154, doi: 10.1145/1807128.1807152.

[28] D. Dahiphale *et al.*, "An advanced mapreduce: Cloud mapreduce, enhancements and applications," *IEEE Trans. Netw. Serv. Manage.*, vol. 11, no. 1, pp. 101–115, 2014, doi: 10.1109/TNSM.2014.031714.130407.

[29] D. J. DeWitt and M. Stonebraker, MapReduce: A Major Step Backwards, 2008. [Online]. Available: https://homes.cs.washington.edu/billhowe/mapreduce_a_major_step_backwards.html

[30] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Design Implementation*, San Francisco, CA, USA, Dec. 6–8, 2004, pp. 137–150. *USENIX Association*. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[31] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008, doi: 10.1145/1327452.1327492.

[32] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010, doi: 10.1145/1629175.1629198.

[33] D. DeWitt, MapReduce: A Major Step Backwards, 2008. [Online]. Available: http://homes.cs.washington.edu/billhowe/mapreduce_a_major_step_backwards.html

[34] J. Dittrich *et al.*, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010, doi: 10.14778/1920841.1920908.

[35] J. Dittrich *et al.*, "Only aggressive elephants are fast elephants," *PVLDB*, vol. 5, no. 11, pp. 1591–1602, 2012.

[36] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 2012, doi: 10.1145/2347736.2347755.

[37] J. Ekanayake *et al.*, "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, Chicago, IL, USA, Jun. 21–25, 2010, pp. 810–818, doi: 10.1145/1851476.1851593.

[38] E. Elnikety, T. Elsayed, and H. E. Ramadan, "iHadoop: Asynchronous iterations for MapReduce," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, Athens, Greece, Nov. 29–Dec. 1, 2011, pp. 81–90, doi: 10.1109/CloudCom.2011.21.

[39] M. Y. Eltabakh *et al.*, "CoHadoop: Flexible data placement and its exploitation in Hadoop," *PVLDB*, vol. 4, no. 9, pp. 575–585, 2011, doi: 10.14778/2002938.2002943.

[40] E. Feller, L. Ramakrishnan, and C. Morin, "Performance and energy efficiency of big data applications in cloud environments: A Hadoop case study," *J. Parallel Distrib. Comput.*, 2015, doi: 10.1016/j.jpdc.2015.01.001.

[41] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The systems declarative stream processing engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Vancouver, BC, Canada, Jun. 10–12, 2008, pp. 1123–1134, doi: 10.1145/1376616.1376729.

[42] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in M. L. Scott and L. L. Peterson, eds., in *Proc. 19th ACM Symp. Operating Syst. Principles*, Bolton Landing, NY, USA, Oct. 19–22, 2003, pp. 29–43, doi: 10.1145/945445.945450.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Design Implementation*, Hollywood, CA, USA, Oct. 8–10, 2012, pp. 17–30.

[44] Y. He *et al.*, "Rcfile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," in *Proc. 27th Int. Conf. Data Eng.*, Hannover, Germany, Apr. 11–16, 2011, pp. 1199–1208, doi: 10.1109/ICDE.2011.5767933.

[45] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, "Meteor/sopremo: An extensible query language and operator model," in *Proc. Workshop End-to-end Manage. Big Data*, Istanbul, Turkey, 2012.

[46] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *PVLDB*, vol. 4, no. 11, pp. 1111–1122, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p1111-herodotou.pdf

[47] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *Proc. 5th Biennial Conf. Innovative Data Syst. Research*, Asilomar, CA, USA, Jan. 9–12, 2011, pp. 261–272. [Online]. Available: www.cidrdb.org; http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper36.pdf

[48] T. Hey, S. Tansley, and K. M. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Res., 2009. [Online]. Available: http://research.microsoft.com/en-us/collaboration/fourthparadigm/

[49] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. IEEE 26th Int. Conf.*, 2010, pp. 41–51.

[50] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, pp. 59–72, New York, NY, USA, 2007, doi: 10.1145/1272996.1273005.

[51] I. King M. R. Lyu, J. Zeng, and H. Yang, "A comparison of lasso-type algorithms on distributed parallel machine learning platforms," in *Proc. Workshop Distrib. Mach. Learning Matrix Comput.*, Montreal, QC, Canada, 2014.

[52] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, "DEDUCE: At the intersection of MapReduce and stream processing," in *Proc. 13th Int. Conf. Extending Database Technol.*, Lausanne, Switzerland, Mar. 22–26, 2010, vol. 426, pp. 657–662, doi: 10.1145/1739041.1739120.

[53] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010, doi: 10.1145/1773912.1773922.

[54] W. Lang and J. M. Patel, "Energy management for MapReduce clusters," *PVLDB*, vol. 3, no. 1, pp. 129–139, 2010. [Online]. Available: http://www.comp.nus.edu.sg/vldb2010/proceedings/files/papers/R11.pdf

[55] K.-H. Lee *et al.*, "Parallel data processing with MapReduce: A survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, 2011, doi: 10.1145/2094114.2094118.

[56] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," *Operating Syst. Rev.*, vol. 44, no. 1, pp. 61–65, 2010, doi: 10.1145/1740390.1740405.

[57] F. Li, B. C. Ooi, M. Tamer Özsu, and S. Wu, "Distributed data management using mapreduce," *ACM Comput. Surv.*, vol. 46, no. 3, p. 31, 2014, doi: 10.1145/2503009.

[58] F. Liang, C. Feng, X. Lu, and Z. Xu, *Performance Benefits of DataMPI: A Case Study with BigDataBench*, Lecture Notes in Computer Science, vol. 8807. Springer Int. Publishing, 2014, doi: 10.1007/978-3-319-13021-7_9.

[59] H. Lim, H. Herodotou, and S. Babu, "Stubby: A transformation-based optimizer for MapReduce workflows," *PVLDB*, vol. 5, no. 11, pp. 1196–1207, 2012, doi: 10.14778/2350229.2350239.

[60] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu, "Llama: Leveraging columnar storage for scalable join processing in the MapReduce framework," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Athens, Greece, Jun. 12–16, 2011, pp. 961–972, doi: 10.1145/1989323.1989424.

[61] B. T. Loo *et al.*, "Implementing declarative overlays," in *Proc. 20th ACM Symp. Operating Syst. Principles*, Brighton, U.K., Oct. 23–26, 2005, pp. 75–90, doi: 10.1145/1095810.1095818.

[62] Y. Low *et al.*, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p716_yuchenglow_vldb2012.pdf

[63] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to Hadoop-like big data computing," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, Phoenix, AZ, USA, May 19–23, 2014, pp. 829–838, doi: 10.1109/IPDPS.2014.90.

[64] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in A. K. Elmagarmid and D. Agrawal, Eds., in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Indianapolis, IN, USA, Jun. 6–10, 2010, pp. 135–146, doi: 10.1145/1807167.1807184.

[65] L. Mashayekhy, M. M. Nejad, D. Grosu, D. Lu, and W. Shi, "Energy-aware scheduling of MapReduce jobs," in *Proc. IEEE Int. Congress Big Data*, Anchorage, AK, USA, Jun. 27–Jul. 2, 2014, pp. 32–39, doi: 10.1109/BigData.Congress.2014.15.

[66] S. Melnik *et al.*, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 330–339, 2010.

[67] MongoDB Inc. Mongodb, 2016. [Online]. Available: https://www.mongodb.com

[68] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in W. Fan, W. Hsu, G. I. Webb, B. Liu, C. Zhang, D. Gunopulos, and X. Wu, eds., in *Proc. 10th IEEE Int. Conf. Data Mining Workshops*, Sydney, Australia, Dec. 13, 2010, pp. 170–177, doi: 10.1109/ICDMW.2010.172.

[69] D. A. Patterson, "Technical perspective: The data center is the computer," *Commun. ACM*, vol. 51, no. 1, p. 105, 2008, doi: 10.1145/1327452.1327491.

[70] A. Pavlo *et al.*, "A comparison of approaches to large-scale data analysis," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Providence, RI, USA, Jun. 29–Jul. 2, 2009, pp. 165–178, 2009, doi: 10.1145/1559845.1559865.

[71] M. Piszczalski, Locating Data Centers in an Energy-Constrained World, May 2012. [Online]. Available: http://pro.gigaom.com/2012/05/locating-data-centers-in-an-energy-constrained-world/

[72] I. Polato, R. Ré, A. Goldman, and F. Kon, "A comprehensive view of hadoop research—A systematic literature review," *J. Netw. Comput. Appl.*, vol. 46, pp. 1–25, 2014, doi: 10.1016/j.jnca.2014.07.022.

[73] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of MapReduce and large-scale data processing systems," *ACM Comput. Surv.*, vol. 46, no. 1, p. 11, 2013, doi: 10.1145/2522968.2522979.

[74] M. Stonebraker *et al.*, "MapReduce and parallel DBMSs: Friends or foes?" *Commun. ACM*, vol. 53, no. 1, pp. 64–71, 2010, doi: 10.1145/1629175.1629197.

[75] A. Thusoo *et al.*, "Hive—A petabyte scale data warehouse using Hadoop," in *Proc. 26th Int. Conf. Data Eng.*, Long Beach, CA, USA, Mar. 1–6, 2010, pp. 996–1005, doi: 10.1109/ICDE.2010.5447738.

[76] A. Toshniwal *et al.*, "Storm@twitter," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156, doi: 10.1145/2588555.2595641.

[77] R. Tudoran, A. Costan, and G. Antoniu, "Mapiterativereduce: A framework for reduction-intensive data processing on azure clouds," in *Proc. 3rd Int. Workshop MapReduce Appl. Date*, 2012, New York, NY, USA, pp. 9–16, doi: 10.1145/2287016.2287019.

[78] L. Wang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *Proc. 20th IEEE Int. Symp. High Perform. Comput. Architecture*, Orlando, FL, USA, Feb. 15–19, 2014, pp. 488–499, doi: 10.1109/HPCA.2014.6835958.

[79] T. Wirtz and R. Ge, "Improving MapReduce energy efficiency for computation intensive workloads," in *Proc. Int. Green Comput. Conf. Workshops*, Orlando, FL, USA, Jul. 25–28, 2011, pp. 1–8, doi:10.1109/IGCC.2011.6008564.

[80] R. S. Xin *et al.*, "Shark: Sql and rich analytics at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Fata*, 2013, pp. 13–24.

[81] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," in *Proc. 21th ACM SIGKDD Int. Conf. Knowledge Discovery Data Mining*, Sydney, NSW, Australia, Aug. 10–13, 2015, pp. 1335–1344, doi: 10.1145/2783258.2783323.

[82] Y. Yu *et al.*, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. 8th USENIX Conf. Operating Syst. Design Implementation*, 2008, Berkeley, CA, USA, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id = 1855741.1855742

[83] M. Zaharia, N. M. Mosharaf Chowdhury, M. Franklin, S. Shenker, and I. Stoica, Spark: Cluster Computing With Working Sets, Tech. Rep. UCB/EECS-2010-53, Dept. Electr. Eng. Comput. Syst., Univ. California, Berkeley, May 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html

[84] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A distributed computing framework for iterative computation," *J. Grid Comput.*, 10 vol. 1, pp. 47–68, 2012, doi: 10.1007/s10723-012-9204-9.

[85] I. A. T. Hashema, I. Yaqooba, N. B. Anuara, S. Mokhtara, A. Gania, and S. U. Khanb, "The rise of "big data" on cloud computing: Review and open research issues," *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015, doi: 10.1016/j.is.2014.07.006.

ABOUT THE AUTHORS

Yunquan Zhang (Member, IEEE) received the Ph.D. degree in computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2000.

He is a Full Professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current appointments include the Expert Review Group member of Information Sciences Division of the National Science Foundation, China, Secretary-General of China's High Performance Computing Expert Committee, Technical Committee of National 863 High-Performance Computer Evaluation Center, and General Secretary of the Specialty Association of Mathematical & Scientific Software of China. His research interests are in the areas of high performance parallel computing, with particular emphasis on large scale parallel computation and programming models, high-performance parallel numerical algorithms, and performance modeling and evaluation for parallel programs. He has published over 100 papers in international journals and conferences proceedings.

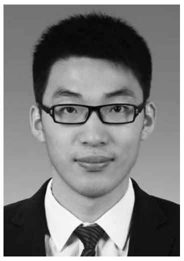
Prof. Zhang served as Chair of Program Committee of IEEE CSE 2010, Vice-Chair of Program Committee of High-Performance Computing China (2008–2012), member of Steering Committee of International Supercomputing Conference 2012, member of Program Committee of IEEE ICPADS 2008, ACM ICS 2010, IEEE IPDPS 2012, and IEEE CCGRid 2012.

Ting Cao (Member, IEEE) received the Ph.D. degree from Research School of Computer Science, Australia National University, Canberra, Australia, in 2014.

She is currently a Postdoctoral Fellow with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Her research areas include efficient implementation of high-level languages, novel computer architectures, and high-performance computing.

Shigang Li received the B.Eng. degree from School of Computer and Communication Engineering, University of Science and Technology Beijing, China, in 2009, and the Ph.D. degree from the Department of Computer Science and Technology, University of Science and Technology, Beijing, China, in 2014.

He was a Visiting Ph.D. student with the University of Illinois at Urbana-Champaign from 2011 to 2013. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include large-scale parallel computing based on MPI, heterogeneous computing, and scalable deep learning algorithms on multi-/many-core clusters.

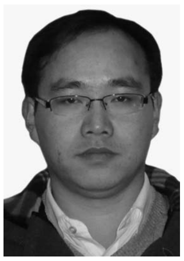


Xinhui Tian received the B.S. degree from the School of Electronics Engineering and Computer Science, Peking University, Peking, China, in 2011. He is currently working toward the Ph.D. degree at the Advanced Computer Systems Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

His research interests include big data and distributed system.

Liang Yuan received the B.S. degree from the School of Computer and Communication Engineering, University of Technology Tianjing, China, in 2005, and the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2013.

He was a Visiting Ph.D. student with the University of Rochester in 2011. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include large-scale parallel computing and heterogeneous computing.



Haipeng Jia received the B.Eng. and Ph.D. degrees from Ocean University of China, Qingdao, China, in 2007 and 2012, respectively.

He was a Visiting Ph.D. student with the Institute of Software, Chinese Academy Of Sciences, Beijing, China, from 2010 to 2012. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include heterogeneous computing, many-core parallel programming method and computer version algorithms on multi-/many-core processors.

Athanasios V. Vasilakos (Senior Member, IEEE) received the B.S. degree in electrical and computer engineering from the University of Thrace, Orestiada, Greece, in 1984, the M.S. degree in computer engineering from the University of Massachusetts Amherst, Amherst, MA, USA, in 1986, and the Ph.D. degree in computer engineering from the University of Patras, Patras, Greece, in 1988.

He is a Professor with the Lulea University of Technology, Sweden. He served or is serving as an Editor for many technical journals, such as the IEEE Transactions on Network and Service Management, IEEE Transactions on Cloud Computing, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Cybernetics, IEEE Transactions on Nanobioscience, IEEE Transactions on Information Technology in Biomedicine, *ACM Transactions on Autonomous and Adaptive Systems*, and the IEEE Journal on Selected Areas in Communications. He is also the General Chair of the European Alliances for Innovation.