A Survey on Spark Ecosystem: Big Data Processing Infrastructure, Machine Learning, and Applications

Shanjiang Tang, Bingsheng He, Ce Yu, Yusen Li, and Kun Li

Abstract—With the explosive increase of big data in industry and academic fields, it is important to apply large-scale data processing systems to analyze Big Data. Arguably, Spark is the state-of-the-art in large-scale data computing systems nowadays, due to its good properties including generality, fault tolerance, high performance of in-memory data processing, and scalability. Spark adopts a flexible Resident Distributed Dataset (RDD) programming model with a set of provided transformation and action operators whose operating functions can be customized by users according to their applications. It is originally positioned as a *fast* and *general* data processing system. A large body of research efforts have been made to make it more efficient (faster) and general by considering various circumstances since its introduction. In this survey, we aim to have a thorough review of various kinds of optimization techniques on the generality and performance improvement of Spark. We introduce Spark programming model and computing system, discuss the pros and cons of Spark, and have an investigation and classification of various solving techniques in the literature. Moreover, we also introduce various data management and processing systems, machine learning algorithms and applications supported by Spark. Finally, we make a discussion on the open issues and challenges for large-scale in-memory data processing with Spark.

Index Terms—Spark, shark, RDD, in-memory data processing

1 Introduction

In the current era of 'big data', the data is collected at unprecedented scale in many application domains, including e-commerce [112], social network [140], and computational biology [146]. Given the characteristics of the unprecedented amount of data, the speed of data production, and the multiple of the structure of data, large-scale data processing is essential to analyzing and mining such big data timely. A number of large-scale data processing frameworks have thereby been developed, such as MapReduce [87], Storm [14], Flink [1], Dryad [102], Caffe [103], Tensorflow [64]. Specifically, MapReduce is a batch processing framework, while Storm is streaming processing system. Flink is a big data computing system for batch and streaming processing. Dryad is a graph processing framework for graph applications. Caffe and Tensorflow are deep learning frameworks used for model training and inference in computer vision, speech recognition and natural language processing.

However, all of the aforementioned frameworks are not *general* computing systems since each of them can only work for a certain data computation. In comparison, Spark [160] is a *general* and *fast* large-scale data processing system widely used in both industry and academia with many merits. For example, Spark is much faster than MapReduce in performance, benefiting from its in-memory data processing. Moreover, as a general system, it can support batch, interactive, iterative, and streaming computations in the same runtime, which is useful for complex applications that have different computation modes.

Despite its popularity, there are still many limitations for Spark. For example, it requires considerable amount of learning and programming efforts under its RDD programming model. It does not support new emerging heterogenous computing platforms such as GPU and FPGA by default. Being as a general computing system, it still does not support certain types of applications such as deep learning-based applications [25].

To make Spark more *general* and *fast*, there have been a lot of work made to address the limitations of Spark [63], [94], [115], [121] mentioned above, and it remains an active research area. A number of efforts have been made on performance optimization for Spark framework. There have been proposals for more complex scheduling strategies [137], [150] and efficient memory I/O support (e.g., RDMA support) to improve the performance of Spark. There have also been a number of studies to extend Spark for more sophisticated algorithms and applications (e.g., deep learning algorithm, genomes, and Astronomy). To improve the ease of use, several high-level declarative [23], [129], [156] and procedural languages [49], [54] have also been proposed and supported by Spark.

- S.J. Tang, C. Yu, and K. Li are with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China. E-mail: {tashj, yuce, likun30901}@tju.edu.cn.
- B.S. He is with the School of Computing, National University of Singapore, Singapore 119077. E-mail: hebs@comp.nus.edu.sg.
- Y. Li is with the School of Computing, Nankai University, Tianjin 300071, China. E-mail: liyusen@nbjl.nankai.edu.cn.

Manuscript received 3 Dec. 2018; revised 26 Nov. 2019; accepted 14 Feb. 2020. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding authors: Shanjiang Tang and Ce Yu.)
Recommended for acceptance by L. Chen.
Digital Object Identifier no. 10.1109/TKDE.2020.2975652

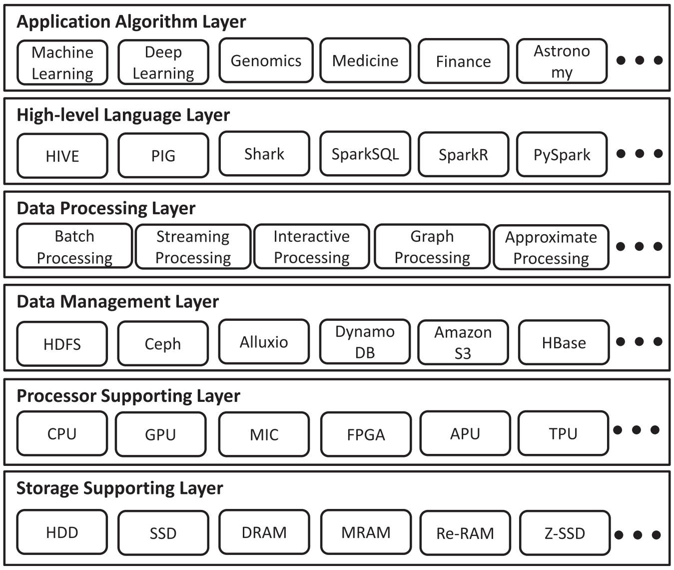


Fig. 1. Overview of Spark ecosystem from the bottom up. We classify it into six layers for improved generality and performance efficiency.

Still, with the emergence of new hardware, software and application demands, it brings new opportunities as well as challenges to extend Spark for improved generality and performance efficiency. In this survey, for the sake of better understanding these potential demands and opportunities systematically, we classify the study of Spark ecosystem into six support layers as illustrated in Fig. 1, namely, Storage Supporting Layer, Processor Supporting Layer, Data Management Layer, Data Processing Layer, High-level Language Layer and Application Algorithm Layer. The aim of this paper is two-fold. We first seek to have an investigation of the latest studies on Spark ecosystem. We review related work on Spark and classify them according to their optimization strategies in order to serve as a guidebook for users on the problems and addressing techniques in data processing with Spark. It summarizes existing techniques systematically as a dictionary for expert researchers to look up. Second, we show and discuss the development trend, new demands and challenges at each support layer of Spark ecosystem as illustrated in Fig. 1. It provides researchers with insights and potential study directions on Spark.

The rest part of this survey is structured as follows. Section 2 introduces Spark system, including its programming model, runtime computing engine, pros and cons, and various optimization techniques. Section 3 describes new caching devices for Spark in-memory computation. Section 4 discusses the extensions of Spark for performance improvement by using new accelerators. Section 5 presents distributed data management, followed by processing systems supported by Spark in Section 6. Section 7 shows the languages that are supported by Spark. Section 8 reviews the Spark-based machine learning libraries and systems, Spark-based deep learning systems, and the major applications that the Spark system is applied to. Section 9 makes some open discussion on the challenging issues. Finally, we conclude this survey in Section 10.

2 Core Techniques of Spark

This section first describes the RDD programming model, followed by the overall architecture of Spark framework. Next it shows the pros and cons of Spark, and various optimization techniques for Spark.

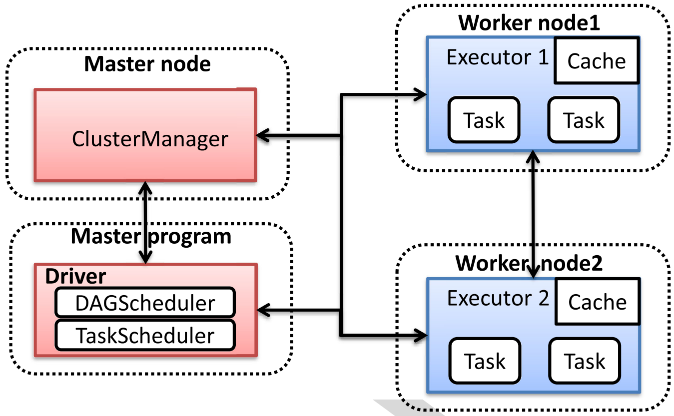


Fig. 2. Architecture overview of Spark.

2.1 Programming Model

Spark is based on Resilient Distributed Dataset (RDD) [159] abstraction model, which is an immutable collection of records partitioned across a number of computers. Each RDD is generated from data in external robust storage systems such as HDFS, or other RDDs through coarse-grained *transformations* including *map*, *filter* and *groupByKey* that use identical processing to numerous data records. To provide fault tolerance, each RDD's transformation information is logged to construct a lineage dataset. When a data partition of a RDD is lost due to the node failure, the RDD can recompute that partition with the full information on how it was generated from other RDDs. It is worthy mentioning that the transformation is a *lazy* operation that only defines a new RDD instead of calculating it immediately. In order to launch the computation of RDD, Spark offers another group of *action* operations such as *count*, *collect*, *save* and *reduce*, which either return a data result to an application program or store the RDD's data to an external storage system. Moreover, for the data of a RDD, they can be persisted either in memory or in disk, controlled by users.

2.2 Spark Architecture

Fig. 2 overviews the architecture of Spark on a cluster. For each Spark application, it spawns one master process called *driver*, which is responsible for task scheduling. It follows a hierarchical scheduling process with jobs, stages and tasks, where *stages* refer to as smaller sets of tasks divided from interdependent jobs, which resemble map and reduce phases of a MapReduce job. There are two schedulers inside it, namely, *DAGScheduler* and *TaskScheduler*. The DAGScheduler figures out a DAG of stages for a job and keeps track of the materialized RDDs as well as stage outputs, whereas TaskScheduler is a low-level scheduler that is responsible for getting and submitting tasks from each stage to the cluster for execution.

Spark provides users with three different cluster modes (i.e., Mesos [97], YARN [149], and standalone mode) to run their Spark applications by allowing driver process to connect to one of existing popular cluster managers including Mesos, YARN and its own independent cluster manager. In each worker node, there is a slave process called *executor* created for each application, which is responsible for running the tasks and caching the data in memory or disk.

2.3 Pros and Cons of Spark

MapReduce and Flink are two powerful large-scale data processing systems widely used for many data-intensive applications. In this section, we take MapReduce and Flink as baselines to discuss the pros and cons of Spark.

2.3.1 Spark versus MapReduce

Compared to MapReduce, Spark has the following merits:

Easy to Use. Spark provides users with more than 80 high-level simple operators (e.g., *map*, *reduce*, *reduceByKey*, *filter*) that allow users to write parallel applications at the application level with no need to consider the underlying complex parallel computing problems like data partitioning, task scheduling and load balancing. Moreover, Spark allows users to write their user-defined functions with different programming languages like Java, Scala, Python by offering corresponding APIs.

Faster Than MapReduce. Due to its in-memory computing, Spark has shown to be $10\times \sim 100\times$ faster than MapReduce in batch processing [13].

General Computation Support. First, from the aspect of processing mode, Spark is an integrated system that supports batch, interactive, iterative, and streaming processing. Second, Spark has an advanced DAG execution engine for complex DAG applications, and a stack of high-level APIs and tools including Shark [156], Spark SQL [129], MLlib and Graphx [94] for a wide range of applications.

Flexible Running Support. Spark can run in a standalone mode or share the cluster with other computing systems like MapReduce by running on YARN or Mesos. It also provides APIs for users to deploy and run on the cloud (e.g., Amazon EC2). Moreover, it can support the access of various data sources including HDFS, Tachyon [115], HBase, Cassandra [111], and Amazon S3 [21].

Albeit many benefits, there are still some weakness for Spark, compared with MapReduce as follows:

Heavy Consumption of Storage Resources. As an in-memory data processing framework, Spark is superior to MapReduce in performance, achieved by reducing the redundant computations at the expense of storage resources, especially memory resource. Similar to existing popular in-memory caching systems like Memcached [134], [163] and Redis [78], it saves RDD data in memory and keeps it there for data sharing across different computation stages. More memory resources are needed when there are a large volume of RDD data to be cached in computation.

Poor Security. Currently, Spark supports authentication through a shared secret [12]. In comparison, Hadoop has more security considerations and solutions, including Knox [10], Sentry [16], Ranger [11], etc. For example, Knox provides the secure REST API gateway for Hadoop with authorization and authentication. In contrast, Sentry and Ranger offer access control and authorization over Hadoop data and metadata.

Learning Curve. Although Spark is faster and more general than MapReduce, the programming model of Spark is much more complex than MapReduce. It requires users to take time to learn the model and be familiar with provided APIs before they can program their applications with Spark.

2.3.2 Spark versus Flink

As the biggest competitor of Spark, Flink [1] is a stateful in-memory big data computing system for batch, streaming and interactive data processing. The two frameworks learn from each other and have many similarities in their functions, which are compared and summarized as follows:

Data Abstraction Model and Performance. The two frameworks are based on different programming models for batch and streaming applications. For Spark, it is based on RDD abstraction model for batch computation and DStream model for streaming computation. Since DStream is internally RDD itself, the streaming computation of Spark is indeed a *near* realtime streaming processing system achieved by emulating the streaming process through a serial of micro-batch computations. In contrast, Flink leverages Dataset abstraction for batch applications and DataStream for streaming applications, which is the real event-based streaming system.

Compared to MapReduce, Spark and Flink can achieve higher performance efficiency for batch and streaming applications due to their in-memory computation. Particularly, for iterative batch applications and streaming applications, Flink is faster than Spark due to its incrementally iterative computation and streaming architecture that only handle portion of data that have actually changed [126].

Generality. Like Spark, Flink is also a general computing system that 1) supports a variety of computations including batch, streaming, iterative, interactive computation as well as graph, machine learning computation, etc, and 2) has a number of programming language supports such as SQL, Java, Scala, Python, R, etc. Moreover, both Spark and Flink are fully compatible to Hadoop Ecosystem, which can run in YARN and process data in HDFS, HBase, Cassandra, Hive, etc. All of these make Spark and Flink become flexible and easy-to-use in practice.

Fault Tolerance. Spark and Flink are both fault tolerant but on the basis of different mechanisms. Spark achieves fault tolerance based on the lineage recovery mechanism, which is an efficient fault tolerance mechanism that only needs to recompute the lost data through lineage information with no extra storage cost. In constrat, Flink is based on Chandy-Lamport distributed snapshots [76] acting as consistent checkpoints, which is a lightweight fault tolerance mechanism that can achieve high throughput while offer strong consistency guarantees at the same time.

Maturity and Popularity. Spark is relatively more mature and popular than Flink in the big data community. First, the documents of Spark are well written and maintained by Spark community whereas for Flink it is still under documenting. Because of this, the number of active users for Spark is much larger than Flink. Second, like Spark, the security of Flink is poor and not mature. It only supports user-level authentication via Hadoop/Kerberos authentication.

Summary. For the sake of better understanding Spark's characteristics, we make a summary of Spark, Flink and MapReduce in Table 1 with respect to different metrics. First, the three frameworks have a good usability, flexibility,

TABLE 1
The Comparison of Spark, Flink and MapReduce

Metrics	Spark	Flink	MapReduce
Usability	Easy-to-use	Easy-to-use	Easy-to-use
Performance	High Efficiency	High Efficiency	Low Efficiency
Generality	Yes	Yes	No
Flexibility	Yes	Yes	Yes
Scalability	Yes	Yes	Yes
Fault Tolerance	Yes	Yes	Yes
Memory Consumption	Heavy	Heavy	Heavy
Security	Poor	Poor	Strong
Learning	hard-to-learn	hard-to-learn	easy-to-learn
Popularity	Yes	No	No

scalability, and fault tolerance properties. All of complex details of distributed computation are encapsulated and well considered by frameworks and are transparent to users. Second, both Spark and Flink outperform MapReduce in performance and generality, attributing to Spark and Flink's in-memory computation and their flexible programming models. Reversely, MapReduce has a stronger security and easy-to-learn property than Spark and Flink. Compared to Spark and Flink, the programming model of MapReduce is more simple and mature. Moreover, the three frameworks have the problem of high memory consumption, due to the heavy memory usage of JVMs. Finally, due to the strong merits and well-written documentation of Spark, it has become the most popular project among the three frameworks.

2.4 Spark System Optimization

Performance is the most important concern for Spark system. Many optimizations are studied on top of Spark in order to accelerate the speed of data handling. We mainly describe the major optimizations proposed on the Spark system in this section.

2.4.1 Scheduler Optimization

The current Spark has a centralized scheduler which allocates the available resources to the pending tasks according to some policies (e.g., FIFO or Fair). The design of these scheduling policies can not satisfy the requirements of current data analytics. In this section, we describe different kinds of schedulers that are especially optimized for large-scale distributed scheduling, approximate query processing, transient resource allocation and Geo-distributed setting, respectively.

Decentralized Task Scheduling. Nowadays, more and more Big Data analytics frameworks are with larger degrees of parallelism and shorter task durations in order to provide low latency. With the increase of tasks, the throughput and availability of current centralized scheduler can not offer low-latency requirement and high availability. A decentralized design without centralized state is needed to provide attractive scalability and availability. Sparrow [137] is the-state-of-art distributed scheduler on top of Spark. It provides the power of two choices load balancing technique for Spark task scheduling. The power probes two random servers and places tasks on the server with less load. Sparrow adapts the power of two choices technique to Spark so that it can effectively run parallel jobs running on a cluster with the help of three techniques, namely, Batch Sampling, Late Binding and Policies and Constraints. Batch Sampling reduces the time of tasks response which is decided by the finishing time of the last task by placing tasks of one job in a batch way instead of sampling for each task individually. For the power of two choices, the length of server queue is a poor norm of latency time and the parallel sampling may cause competition. Late binding prevents two issues happening by delaying allocation of tasks to worker nodes before workers get ready to execute these tasks. Sparrow also enforces global policies using multiple queues on worker machines and supports placement constraints of each job and task.

Data-Aware Task Scheduling. For machine learning algorithms and sampling-based approximate query processing systems, the results can be computed using any subset of the data without compromising application correctness. Currently, schedulers require applications to statically choose a subset of the data that the scheduler runs the task which aviods the scheduler leveraging the combinatorial choices of the dataset at runtime. The data-aware scheduling called KMN [150] is proposed in Spark to take advantage of the available choices. KMN applies the "late binding" technique which can dynamically select the subset of input data on the basis of the current cluster's state. It significantly increases the data locality even when the utilization of the cluster is high. KMN also optimizes for the intermediate stages which have no choice in picking their input because they need all the outputs produced by the upstream tasks. KMN launches a few additional jobs in the previous stage and pick choices that best avoid congested links.

Transient Task Scheduling. For cloud servers, due to various reasons, the utilization tends to be low and raising the utilization rate is facing huge competitive pressure. One addressing solution is to run insensitive batch job workloads secondary background tasks if there are under-utilized resources and evicted them when servers's primary tasks requires more resources (i.e., *transit resources*). Due to excessive cost of cascading re-computations, Spark works badly in this case. Transient Resource Spark (TR-Spark) [157] is proposed to resolve this problem. It is a new framework for large-scale data analytic on transient resources which follows two rules: data scale reduction-aware scheduling and lineage-aware checkpointing. TR-Spark is implemented by modifying Spark's Task Scheduler and Shuffle Manager, and adding two new modules Checkpointing Scheduler and Checkpoint Manager.

Scheduling in a Geo-Distributed Environment. Geo-distributed data centers are deployed globally to offer their users access to services with low-latency. In Geo-distributed setting, the bandwidth of WAN links is relatively low and heterogeneous compared with the intra-DC networks. The query response time over the current intra-DC analytics frameworks becomes extreme high in Geo-distributed setting. Iridium [139] is a system designed for Geo-distributed data analytics on top of Spark. It reduces the query response time by leveraging WAN bandwidth-aware data and task placement approaches. By observing that network bottlenecks mainly occur in the network connecting the data centers rather than in the up/down links of VMs as assumed by Iridium, Hu *et al.* [98] designed and implemented a new task scheduling algorithm called Flutter on top of Spark. which reduces both the completion time and network costs by formulating the optimization issue as a lexicographical min-max integer linear programming (ILP) problem.

2.4.2 Memory Optimization

Efficient memory usage is important for the current in-memory computing systems. Many of these data processing frameworks are designed by garbage-collected languages like C#, Go, Java or Scala. Unfortunately, these garbage-collected languages are known to cause performance overhead due to GC-induced pause. To address the problem, current studies either improvement the GC performance of these garbage-collected language or leverage application semantics to manage memory explicitly and annihilate the GC overhead of these garbage-collected languages [2], [4], [122], [123]. In this section, we introduce these optimizations from these two aspects.

Spark run multiple work processes on different nodes and the Garbage Collection (GC) is performed independently in each node at run. Works communicate data between different nodes (e.g, shuffle operation). In this case, no node can continue until all data are received from all the other nodes. GC pauses can lead to unacceptable long waiting time for latency-critical applications without the central coordination. If even a single node is stuck in GC, then all the other nodes need wait. In order to coordinate the GC from the central view, Holistic Runtime System [122], [123] is proposed to collectively manages runtime GC across multiple nodes. Instead of making decisions about GC independently, such Holistic GC system allows the runtime to make globally coordinated consensus decision through three approaches. First, it let applications choose the most suitable GC policy to match the requirement of different applications (e.g., throughput versus pause times). Second, Holistic system performs GC by considering the application-level optimizations. Third, the GC system is dynamically reconfigured at runtime to adapt to system changes.

Instead of replying the memory management of such managed languages. Spark also tries to manage the memory by itself to leverage the application semantic and eliminate the GC overhead of these garbaged-collected languages. Tungsten [4] improves the memory and CPU efficiency of spark applications to make the performance of Spark reach the limits of modern hardware. This work consists of three proposes. First, it leverages the off-heap memory, a feature provided by JAVA to allocate/deallocate memory like c and c++, to manage memory by itself which can take advantage of the application semantics and annihilate the overhead of JVM and GC. Second, it proposes cache-obvious algorithms and data structures to develop memory hierarchical structure. Third, it uses the code generation to avoid the overhead the expression evaluation on JVM (e.g., too many virtual functions calls, extensive memory access and can not take advantage modern CPU features such as SIMD, pipeline and prefetching). Recently, Spark further optimizes its performance by integrating the techniques proposed in Modern parallel database area [132]. Spark 2.0 leverages whole process code generation and vectorization to further ameliorate the code generation at runtime [2].

2.4.3 I/O Optimization

For large-scale data-intensive computation in Spark, the massive data loading (or writing) from (or to) disk, and transmission between tasks at different machines are often unavoidable. A number of approaches are thereby proposed to alleviate it by having a new storage manner, using data compression, or importing new hardware.

Data Compression and Sharing. One limitation for Spark is that it can only support the in-memory data sharing for tasks within an application, whereas not for tasks from multiple applications. To overcome this limitation, Tachyon [115], [116] is proposed as a distributed in-memory file system that achieves reliable data sharing at memory speedup for tasks from different processes. The Spark applications can then share their data with each other by writing (or reading) their data to (or from) Tachyon at memory speedup, which is faster than disk-based HDFS file system. Moreover, to enable more data saved in memory for efficient computation, Agarwal *et al.* [65] proposed and implemented a distributed data store system called Succinct in Tachyon that compresses the input data and queries can be executed directly on the compressed representation of input data, avoiding decompression.

Data Shuffling. Besides the performance degradation from the disk I/O, the network I/O may also be a serious bottleneck for many Spark applications. Particularly, *shuffle*, a many-to-many data transfer for tasks across machines, is an important consumer of network bandwidth for Spark. Zhang *et al.* [164] observed that the bottleneck for shuffle phase is due to large disk I/O operations. To address it, a framework called Riffle is proposed to improve I/O efficiency through combining fragmented intermediate shuffle files into larger block files and converting small and random disk I/O operations into large and sequential ones. Davidson *et al.* [63] proposed two approaches to optimize the performance in data shuffling. One is to apply the Columnar compression technique to Spark's shuffle phase in view of its success in a column-oriented DBMS called C-Store [144], so as to offload some burden from the network and disk to CPU. Moreover, they observe that Spark generates a huge number of small-size shuffle files on both the map and reduce phase, which introduces a heavy burden on operating system in file management. A shuffle file consolidation approach is thereby proposed to reduce the number of shuffle files on each machine.

Moreover, prefetching is an effective technique to hide shuffling cost by overlapping data transfers and the shuffling phase. Current state-of-the-art solutions take simple mechanisms to determine where and how much data to acquire from, resulting in the performance of sub-optimal and the excessive use of supplemental memory. To address it, Bogdan *et al.* [133] proposed an original adaptive shuffle data transfer strategy by dynamically adapting the prefetching to the calculation. It is achieved by taking into account load balancing for request extraction using executor-level coordination, prioritization according to locality and responsiveness, shuffle block aggregation, elastic adjustment of in-flight restrictions, static circular allocation of initial requests, and dispersal using in-flight increment.

There are also some work focusing on optimizing shuffling under a certain circumstance. Kim *et al.* [107] considered the I/O optimization for Spark under large memory servers. It can achieve better data shuffling and intermediate storage by replacing the existing TCP/IP-based shuffle with a large shared memory approach. The communication cost of map and reduce tasks can be reduced significantly through referencing to the global shared memory compared with data transferring over the network. Liu *et al.* [120] studied the data shuffling in a wide-area network, where data transfers occur between geographically distributed datacenters. It designed and implemented a data aggregation spark-based system by aggregating the output of map tasks to a subset of worker datacenters strategically and proactively, which replaces the original passive fetch mechanisms used in Spark across datacenters. It can avoid repetitive data transfers, which can thereby improve the utilization of inter-datacenter links.

RDMA-Based Data Transfer. Lu *et al*. [121] accelerated the network communication of Spark in big data processing using Remote Direct Memory Access (RDMA) technique. They proposed a RDMA-based data shuffle engine for Spark over InfiniBand. With RDMA, the latency of network message communication is dramatically reduced, which improves the performance of Spark significantly.

2.4.4 Provence Support

Data-intensive scalable computing (DISC) systems such as Hadoop and Spark, provide a programming model for users to authorize data processing logic, which is converted to a Directed Acyclic Graph (DAG) of parallel computing [101]. Debugging data processing logic in DISC systems is difficult and time consuming. A library, *Titian* [101], provides data provenance support at the velocity of interactive based on Apache Spark. The contributions of Titian are summarized as follow: A data lineage capture and query support system while minimally impacting Spark job performance. Interactive data provenance query support the expansion of a conversant programming model Spark RDD with less overhead. Titian extends the native Spark RDD interface with tracing capabilities and returns a LineageRDD, traveling by dataflow transformations at stage boundaries. The user is able to retrospect to the intermediate data of the program execution from the given RDD, then leverage local RDD transformations to reprocess the referenced data.

Currently, researchers use cloud computing platforms to analyse Big Data in parallel, but debugging massive parallel computations is time consuming and infeasible for users. To meet the low overhead, scalability and fine-grained demands of big data processing in Apache Spark, a group of interactive and real-time debugging primitives were developed. BIGDEBUG [95] provides simulated breakpoints and guarded watchpoints with the trifling influence of performance, which indicates less than 19 percent overhead for crash monitoring, 24 percent overhead for record-level tracing, and 9 percent overhead for watchpoint on average. BIGDEBUG supports a real-time rapid repair and recovery to prevent re-running the job from the beginning. Besides, BIGDEBUG offers the provenance of the culprit and fine-grained tracking of records in distributed pipes to track intermediate results back and forth.

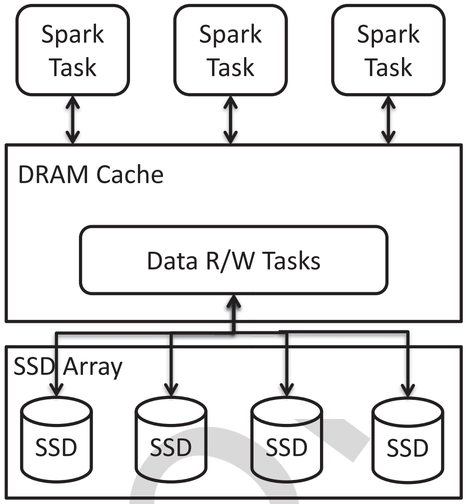


Fig. 3. Multi-tier storage system consisting of DRAM and SSD.

An improved version of the original Titian system is designed to reduce the lineage query time [100]. The two key features of Titian are crash culprit determination and automated fault localization. The culprit information is packaged and dispatch to users with other run-time records. The delta debugging technique diagnose whether mistakes in code and data. To promote the performance of lineage queries, they extend Spark with an available way to retrieve lineage records more pragmatically. For large-scale data, small tracing queries generate remarkable overhead from jobs that make little contribution to the result. Therefore, it proposes Hyperdrive, a customized Spark scheduler, which utilizes partition statistics to exclude the situation. Moreover, Hyperdrive decouples task operations from partitions and dispenses multiple partitions to one task.

3 Storage Supporting Layer

Spark takes DRAM as caches in its in-memory computation. Although DRAM has a much higher bandwidth and lower latency compared with HDD in data communication, its capacity is often limited due to the high cost of DRAM as well as its high power consumption [70]. It can significantly constrain large-scale data applications from gaining high in-memory hit-rates that is essential for high-performance on Spark. The new emerging storage devices in recent years give us a chance to alleviate it in the following ways:

SSD-Based In-Memory Computing. Solid-State Disk (SSD) is a new storage device that provides much higher access speed than traditional HDD. Instead of using HDD, one approach is to adopt SSD as persistent storage by setting up a multi-tier storage system as illustrated in Fig. 3. In comparison to HDD, the data movement between memory and SSD is much faster. We can improve Spark performance by spilling RDDs to SSD when the memory cache is full. By using SSDs, there can be up to $10\times$ performance improvement over HDD-based caching approach for Spark [59].

NVM-Based In-Memory Computing. Compared to DRAM, the latency of SSD is still very large (i.e., about $500\times$ slower than DRAM) although it is much faster than HDD [81]. Emerging Non-Volatile Memory (NVM), such as PCM, STT-RAM and ReRAM, is considered as an alternative to

SSD [119] due to its much lower latency and higher bandwidth than SSD. We can integrate DRAM, NVM and SSD to establish a multi-tier caching system by first caching the data in DRAM, or putting into NVM when DRAM is full, or in the SSD when both DRAM and SSD are full.

4 Processor Supporting Layer

Since the limited performance and energy efficiency of general-purpose CPUs have impeded the performance scaling of conventional data centers, it becomes more and more popular to deploy accelerators in data centers, such as GPU and FPGA. Therefore, accelerator-based heterogeneous machine has become a promising basic block of modern data center to achieve further performance and efficiency. In this section, we first provide a summary of Spark systems integrating with GPU to accelerate the computing task. Second, we make a survey of Spark systems with FPGA.

4.1 General Purpose Computation on Graphics Processors (GPGPU)

While Graphics Processing Units (GPU) is originally designed for graphics computation, it now has been widely evolved as an accelerator to deal with general computing operations traditionally handled by CPU, which is referred to as GPGPU [138]. GPU has been widely integrated into modern datacenter for its better performance and higher energy efficiency over CPU. However, the modern computing framework like Spark cannot directly leverage GPU to accelerate its computing task. Several related projects reach out to fill the gap.

1) *HeteroSpark*. Li *et al.* [118] present an novel GPU-enabled Spark *HeteroSpark* which leverages the compute power of GPUs and CPUs to accelerate machine learning applications. The proposed GPU-enabled Spark provides a plug-n-play design so that the current Spark programmer can leverage GPU computing power without needing any knowledge about GPU.
2) *Vispark*. Choi *et al*. [82] propose an extension of Spark called *Vispark*, which leverages GPUs to accelerate array-based scientific computing and image processing applications. In particular, Vispark introduces Vispark Resilient Distributed Dataset (VRDD) for handling the array data on the GPU so that GPU computing abilities can be fully utilized.
3) *Exploring GPU Acceleration of Apache Spark*. Manzi *et al*. [125] explore the possibilities and benefits of offloading the computing task of Spark to GPUs. In particular, the non-shuffling computing tasks can be computed on GPU and then the computation time is significantly reduced. The experimental result shows that the performance of K-Means clustering application was optimized by 17X. Its implementation is publicly available (https://github.com/adobe-research/spark-gpu).
4) *Columnar RDD*. Ishizaki [43] proposes one prototype which saves the inner data in a columnar RDD, compared with the conventional row-major RDD, since the columnar layout is much easier to benefit from using GPU and SIMD-enabled CPU. Therefore, the performance of the applicatin logistic regression is improved by 3.15X.

4.2 FPGA

FPGA is integrated into the computing framework Spark to accelerate inner computing task. In particular, there are two related projects: FPGA-enabled Spark and Blaze.

1) *FPGA-enabled Spark [80]*. It explores how to efficiently integrate FPGAs into big-data computing framework Spark. In particular, it designs and deploys an FPGA-enabled Spark cluster, where one representative application next-generation DNA sequencing is accelerated with two key technologies. The first one is that they design one efficient mechanism to efficiently harness FPGA in JVM so that the JVM-FPGA communication (via PCIe) overhead is alleviated. The other one is that one FPGA-as-a-Service (FaaS) framework is proposed where FPGAs are shared among multiple CPU threads. Therefore, the computing abilities of FPGAs can be fully utilized and then the total execution time is significantly reduced.
2) *Blaze [83]*. It provides a high-level programming interface (e.g., Java) to Spark and automatically leverages the accelerators (e.g., FPGA and GPU) in the heterogeneous cluster to speedup the computing task without the interference of programmer. In other words, each accelerator is abstracted as the subroutine for Spark task, which can be executed on local accelerator when it is available. Therefore, the computation time can be significantly reduced. Otherwise, the task will be executed on CPU.

5 Data Management Layer

In the age of Big Data, data is generally saved and managed in distributed filesystems or databases. This sections gives a survey of widely used data storage and management systems for Spark.

5.1 Distributed File Systems

1) *Hadoop Distributed File System (HDFS)*. Hadoop Distributed File System is proposed to be deployed on low-cost commodity hardware. It is highly scalable and fault-tolerant, enabling it to run on a cluster includes hundreds or thousands of nodes where the hardware failure is normal. It takes a master-slave architecture, which contains a master called *NameNode* to manage the file system namespace and regulating access to files by users, and a number of slaves called *DataNodes* each located at a machine for storing the data. Data uploaded into HDFS are partitioned into plenty of blocks with fixed size (e.g., 64 MB per data block) and the NameNode dispatched the data blocks to different DataNodes that save and manage the data assigned to them. To improve data reliability, it replicates each data block three times (the replicator is 3 by default and users can change it) and saves each replica in a different rack. HDFS data access has been originally supported by Spark with its provided native interface,[1]

1. Spark provides users the *'spark-submit'* script to launch applications, which supports hdfs.

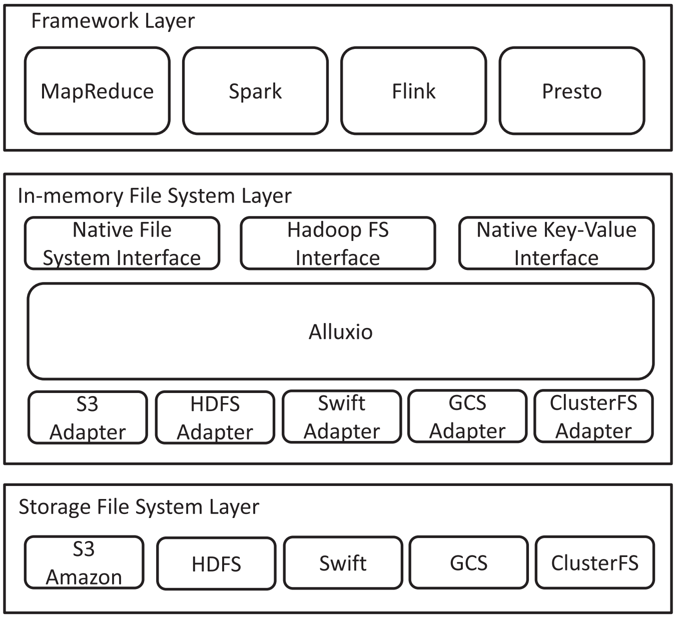


Fig. 4. The Alluxio architecture.

which enables Spark applications to read/write data from/to HDFS directly.

2) *Ceph*. The centralized nature inherent in the client/server model has testified a important barrier to scalable performance. Ceph [153] is a distributed file system which offers high performance and dependability while promising unprecedented expansibility. Ceph uses generating functions replacing file allocation tables to decouple the operations of data and metadata. Ceph is allowed to distribute the complexity around data access, update sequence, duplication and dependability, fault detection, and resume by using the intelligence in OSDs. Ceph uses a highly adaptive distributed metadata cluster architecture that greatly enhances the scalability of metadata access and the scalability of the whole system.

3) *Alluxio*. With the rapid growth of today's big data, storage and networking pose the most challenging bottlenecks since data writes can become network or disk binding, especially when duplication is responsible for fault-tolerance. Alluxio [19], used to be considered as Tachyon, is a fault-tolerant, memory-centric virtual distributed file system that can address the bottleneck. It enables reliable operation of memory speed and data sharing between different applications and cluster computing frameworks. To obtain high throughput writes without impairing fault-tolerance, Alluxio leverages the notion of lineage [74] to recover the lost output by re-implementing output tasks, without the need of replicating the data. With Alluxio, users can do transformations and explorations on large datasets in memory for high performance while enjoying its high data reliability.

Fig. 4 illustrates the memory-centric architecture of Alluxio. It manages data access and fast storage for user applications and computing frameworks by unifying the computing frameworks (e.g., MapReduce, Spark and Flink), and traditional storage systems (e.g., Amazon S3, Apache HDFS and OpenStack Swift), which facilitates data sharing and locality between jobs no matter whether they are running on the same computing system. It serves as a unifying platform for various data sources and computing systems. There are two key functional layers for Aullxio: lineage and persistence. The lineage layer offers high throughput I/O and tracks the information for tasks which produced a specific output. In contrast, the persistent layer materializes data into storage, which is mainly used for checkpoints. Aullxio employs a stand master-slave architecture. That master mainly manages the global metadata of the entire system, tracks lineage information and interacts with a cluster resource manager to distribute resources for recalculation. The slaves manage local storage resources allocated to Alluxio, and storing data and serving requests from users.

5.2 Cloud Data Storage Services

Cloud storage system is able to be typically viewed as a network of distributed data centers that provides storage service to users for storing data by using cloud computing techniques such as virtualization. It often saves the same data redundantly at different locations for high data availability, which is transparent to users. The cloud storage service can be accessed by a co-located cloud computer service, an application programming interfaces (API) or by applications that use the API [27]. There are two popular cloud storage services: Amazon S3 and Microsft Azure.

1). Amazon Simple Storage Service (S3). Amazon S3 is a web-based storage service that allows the user to save and fetch data at any time and any place through web services interfaces such as REST-style HTTP interface, SOSP interface and BitTorrent protocol [21]. It charges users for on-demand storage, requests and data transfers.

The data in Amazon S3 is managed as objects with an object storage architecture, which is opposed to file systems that manage data as a file hierarchy. Objects are organized into *buckets*, each of which is owned by an AWS account. Users can identify objects within each bucket by a unique, user-assigned key.

Spark's file interface can allow users to access data in Amazon S3 by specifying a path in S3 as input through the same URI formats[2] that are supported for Hadoop [40]. However, the storage of Spark dataframe in Amazon S3 is not natively supported by Spark. Regarding this, users can utilize a spark s3 connector library [50] for uploading dataframes to Amazon S3.

2). Microsft Azure Blob Storage (WASB). Azure Blob storage (WASB) [35] is a cloud service for users to save and fetch any amount of unstructured data like text and binary data, in the form of Binary Large Objects (BLOBs). Three types of blobs are supported, namely, block blobs, append blobs and page blobs. Block blobs are suitable for storing and streaming cloud objects. Append blobs are optimized for append operations. In contrast, page blobs are improved to represent IaaS disks and support random writes. Multiple Blobs are grouped into a container and a user storage account can have any number of containers. The saved data can be accessed via HTTP, HTTPS, or REST API.

2. The form of URI is: *s3n://<bucket>/path*.

Spark is compatible with WASB, enabling the data saved in WASB to be directly accessed and processed by Spark via specifying an URI of the format *'wasb://path'* that represents the path where the data is located.

5.3 Distributed Database Systems

1). Hbase. Apache Hbase [9] is an open-source implementation of Google's BigTable [79], which is a distributed key-value database with the features of data compression, in-memory operation and bloom filters on a per-column basis. It runs on top of Hadoop that leverages the high scalability of HDFS and strong batch processing capabilities of MapReduce to enable massive data analysis, and provides real-time data access with the speed of a key/value store for individual record query.

It is a column-oriented key-value database that each table is saved as a multidimensional sparse map, having a timestamp for each cell tagged by column family and column name. A cell value can be identified and retrieved by specifying (Table Id, Row Key, Column-Family:Column, Timestamp). A Hbase table consists of regions, each of which is defined by a startKey and endKey. Except for parent column families being fixed in a schema, users can add columns to tables on-the-fly. All table accesses are achieved by the primary key through the Java API, REST, Avro or Thrift gateway APIs.

There are a number of libraries and tools emerged that enable Spark to interact with HBase. *Spark-HBase Connector* [44] is such a library that provides a simple and elegant API for users' Spark applications to connect to HBase for reading and writing data. To enable native and optimized SQL access to HBase data via SparkSQL/Dataframe interfaces, a tool called *Spark-SQL-on-HBase* [51] is developed by Huawei. Moreover, for efficient scanning, joining and mutating HBase tables to and from RDDs in a spark environment, there is a generic extension of spark module called *spark-on-hbase* [46] developed.

2). Dynamo. Amazon Dynamo [88] is a decentralized distributed key-value storage system with high scalability and availability for Amazon's applications. It has characteristics of both databases and distributed hash tables (DHTs) [28]. It is built to control the state of Amazon's application programs which require high reliability over the trade-offs between availability, consistency, cost-effectiveness and performance. Several Amazon e-commerce services only need primary-key access to a data store, such as shopping carts, customer preferences and sales rank. For these services, it caused inefficiencies and limited size and availability by using relational databases. In comparison, Dynamo is able to fulfill these requirements by providing a simple primary-key only interface.

Dynamo leverages a number of efficient optimization techniques to achieve high performance. It first uses a variant of consistent hashing to divide and replicate data across machines for overcoming the inhomogeneous data and workload distribution problem. Second, the technology is similar to arbitration and decentralized replication synchronization protocols to ensure data consistency during the update. Third, it employs a gossip-style membership protocol that enables each machine to learn about the arrival (or departure) of other machine for the decentralized failure detection.

3). DynamoDB. Amazon DynamoDB [20] is a new fast, high reliability, cost-effective NoSQL database service designed for Internet applications. It is based on strong distributed systems principles and data models of Dynamo. In contrast to Dynamo that requires users to run and manage the system by themselves, DynamoDB is a fully managed service that frees users from the headaches of complex installation and configuration operations. It is built on Solid State Drives which offers fast and foreseeable performance with very low latency at any scale. It enables users to create a database table that can store and fetch any amount of data through the ability to disperse data and traffic to a sufficient number of machines to automatically process requests for any level of demand.

Medium company [36] creates a library called *Spark-DynamoDB* [30] that provides DynamoDB data access for Spark. It enables to read an DynamoDB table as a Spark DataFrame, and allows users to run SQL quries against DynamoDB tables directly with SparkSQL.

4). Cassandra. Apache Cassandra [111] is a highly scalable, distributed structured key-value storage system designed to deal with large-scale data on top of hundreds or thousands of commodity servers. It is open sourced by Facebook in 2008 and has been widely deployed by many famous companies.

Cassandra integrates together the data model from Google's BigTable [79] and distributed architectures of Amazon's Dynamo [88], making it eventually consistent like Dynamo and having a columnFamily-based data model like BigTable. Three basic database operations are supported with APIs: *insert(table, key, rowMutation)*, *get(table, key, columnName)* and *delete(table, key, columnName)*. There are four main characteristics [22] for Cassandra. First, it is decentralized so that every node in the cluster plays the same role without introducing a single fault point of the master. Second, it is highly scalable that read/write throughput both increase linearly as the increasement of new machines and there is no downtime to applications. Third, each data is replicated automatically on multiple machines for fault tolerance and the failure is addressed without shutdown time. Finally, it offers a adjustable level of consistency, allowing the user to balance the tradeoff between read and write for different circumstances.

To enable the connection of Spark applicaitons to Cassandra, a *Spark Cassandra Connector* [42] is developed and released openly by DataStax company. It exposes Cassandra tables as Spark RDDs and can save RDDs back to Cassandra with an implicit *saveToCassandra* call. Moreover, to provide the python support of pySpark [49], there is a module called *pyspark-cassandra* [38] built on top of *Spark Cassandra Connector*.

5.4 Comparison

Table 2 shows the comparison of different storage systems supported by Spark. We summarize them in different ways, including the type of storage systems they belong to, the storage places where it supports to store the data, the data storing model, the data accessing interface and the licence. Similar to Hadoop, Spark has a wide range support for various typed storage systems via its provided low-level APIs or SparkSQL, which is crucial to keep the generality of

TABLE 2
The Comparison of Different Storage Systems

Storage	System Type	Supported Layer	Data Model	Spark Query Interface	License
HDFS	Distributed File System	In Memory, In Disk	Document-Oriented Store	Low-Level API	Open source- Apache
Ceph	Distributed File System	In Disk	Document-Oriented Store	Low-Level API	Open source- LGPL
Alluxio	Distributed File System	In Memory, In Disk	Document-Oriented Store	Low-Level API	Open source- Apache
Amazon S3	Cloud Storage System	In Disk	Object Store	Low-Level API	Commercial
Microsoft WASB	Cloud Storage System	In Disk	Object Store	Low-Level API	Commercial
Hbase	Distributed Database	In Disk	Key-Value Store	SparkSQL, Low-Level API	Open source- Apache
DynamoDB	Distributed Database	In Disk	Key-Value Store	SparkSQL, Low-Level API	Commercial
Cassandra	Distributed Database	In Memory, In Disk	Key-Value Store	SparkSQL, Low-Level API	Open source- Apache

Spark from the data storage perspective. Like Spark's in-memory computation, the in-memory data caching/storing is also very important for achieving high performance. HDFS, Alluxio and Cassandra can support in-memory and in-disk data storage manners, making them become most popular and widely used for many big data applications.

6 Data Processing Layer

As a general-purpose framework, Spark supports a variety of data computation, including Streaming Processing, Graph Processing, OLTP and OLAP Queries Processing, and Approximate Processing. This section discusses about research efforts on them.

6.1 Streaming Processing

Spark Streaming provides users to deal with real-time data from different sources such as Kafka, Flume, and Amazon Kinesis. Spark is built upon the data parallel computing model and offers reliable real-time streaming data processing. Spark streaming converts the processing into a series of deterministic micro-batch calculations, and then utilizes distributed processing framework of Spark to implement. The key abstraction is a Discretized Stream [161] which distributes data stream into tiny batches. The Spark Streaming works as follows, it partitions the live data stream into batches (called microbatches) of a pre-defined interval (N seconds). Next it takes each batch of data as Resilient Distributed Datasets (RDDs) [159]. Spark Streaming can incorporate with any other Spark components such as MLlib and Spark SQL seamlessly. Due to the popularity of spark streaming, research efforts are devoted on further improving it. Das *et al.* [85] study the relationships among batch size, system throughput and end-to-end latency.

There are also efforts to extend spark streaming framework.

1) *Complex Event Processing*. Complex event processing (CEP) is a type of event stream processing that assembles various sources data to find patterns and complex relationships among various events. By analyzing many data sources, CEP system can help identify opportunities and threats for providing real-time alerts to act on them. Over the last decades, CEP systems have been successfully utilized in diffident fields such as recommendation, stock market monitoring, and health-care. There are two open-source projects on building CEP system on Spark. Decision CEP engine [3] is a Complex Event Processing platform which combines Spark Streaming framework with Siddhi CEP engine. Spark-cep [5] is another stream processing engine built on top of Spark supporting continuous query language. Comparing to the existing Spark Streaming query engines, it supports more efficient windowed aggregation and "Insert Into" query.
2) *Streaming Data Mining*. In this big data era, the growing of streaming data motivates the fields of streaming data mining. There are typically two reasons behind the need of evolving from traditional data mining approach. First, streaming data has, in principle, no volume limit, and hence it is often impossible to fit the entire training dataset into main memory. Second, the statistics or characteristics of incoming data are continuously evolving, which requires a continuously re-training and evolving. Those challenges make the traditional offline model approach no longer fit. To this end, open-sourced distributed streaming data mining platforms, such as SOMOA [130] and StreamDM [6] are proposed and have attracted many attentions. Typically, StreamDM [6], [73] uses Spark Streaming as the provider of streaming data. A list of data mining libraries are supported such as SGD Learner and Perception.

6.2 Graph Processing

For graph processing, it can be easily out of the computation and memory capacities of machines when it become larger in scale and more ambitious in their complexity for graph problems. To this end, distributed graph processing frameworks like GraphX [94] are proposed. GraphX is a library atop of Spark, which encodes graphs as collections and expresses the GraphX APIs using standard dataflow operators. In GraphX, a number of optimization strategies are developed, and we briefly mention a few here.

- GraphX contains a series of built-in partitioning functions suach as the vertex collection and edge collection. A routing table is co-divided with the vertex collection which is hash-partitioned by vertex ids. The edge collection can be split horizontally by users and offers vertex-cut partition.
- To maximize index reuse, the subgraph operation generates subgraphs thatwhich share all graph indexes, and utilizes a bitmask to represent which items are contained.
- In order to reduce join operation, GraphX resolves which attributes a function accesses by analysising JVM bytecode. Using triple unrealized views that are

not yet implemented, only one attribute accessed GraphX will involve a two-way join. In the absence of attribute access, Gracx can completely eliminate the join.

In contrast to many specialized graph processing system such as Pregel [124], PowerGraph [93], GraphX is closely integrated into modern general-purpose distributed dataflow system (i.e., Spark). This approach avoids the need of composing multiple systems which increases complexity for a integrated analytics pipelines, and reduces unnecessary data movement and duplication. Furthermore, it naturally inherited the efficient fault tolerant feature from Spark, which is usually overlooked in specialized graph processing framework. The experimental evaluation also shows that GraphX is close to or faster than specialized graph processing systems.

6.3 OLTP and OLAP Queries Processing

Hybrid Transaction/Analytical Processing (HTAP) systems respond to OLTP and OLAP queries by keeping data in dual formats and it provides streaming processing by the utilization of a streaming engine. SnappyData [141] enable streaming, transactions and interactive analytics in a unitary system. It exploits AQP techniques and multiple data summaries at true interactive speeds. SnappyData include a deep integration of Spark and GemFire. An operation of in-memory data storage is combined with the model of Spark computation. It will make all available CPU kernels busy when tasks are implmneted in partition mode. Spark's API are extended to uniform API for OLAP, OLTP and streaming.

6.4 Approximate Processing

Modern data analytics applications demand near real-time response rates. However, getting exact answer from extreme large size of data takes long response time, which is sometimes unacceptable to the end users. Besides utilizing extra resources (i.e., memory and CPU) to reduce data processing time, approximate processing provides faster query response by reducing the amount of work need to perform through techniques such as sampling or online aggregation. It has been widely observed that users can accept some inaccurate answers which come quickly, especially for exploratory queries.

1). Approximate Query Processing. In practice, having a low response time is crucial for many applications such as web-based interactive query workloads. To achieve that, Sameer *et al.* [67] proposed a approximate query processing system called BlinkDB atop of Shark and Spark, based on the distributed sampling. It can return the query result for a large queries of 17 full data terabytes within 2 seconds while keeping substantial error bounds bound to results with 90–98 percent. The strength of BlinkDB comes from two meaningful ideas: (1) an adaptive optimization framework which keeps a series of multi-dimensional samples from raw data based on time (2) a dynamic sample selection strategy based on the accuracy and response time of queries. Moreover, to evaluate the accuracy of BlinkDB, Agarwal *et al.* [66] proposed an effective error estimation approach by extending the prior diagnostic algorithm [108] to check when bootstrap-based error estimates are not reliable.

Considering that the join operation is a key building block for any database system, Quoc *et al.* [114] proposed a new join operator called APPOXJOIN that approximates distributed join computations on top of Spark by interweaving Bloom filter sketching and stratified sampling. It first uses a Bloom filter to prevent non-joinable data shuffling and then uses a stratified sampling approach to get a representative sample of the joined output.

2). Approximate Streaming Processing. Unlike the batch analysis method in which the input data keeps unchanged during the sampling process, the data for streaming analytics is changing over time. Quoc *et al.* [113] shows that the traditional batch-oriented approximate computing are not well-suited for streaming analytics. To address it, they proposed a streaming analytics system called STREAMAPROX by designing an online stratified reservoir sampling methed to generate approximate output with tight margins of error. It implements STREAMAPROX on Apache Spark Streaming and experimental results show that there can be a accelerate rate of $1.1\times - 2.4\times$ while keeping the same accuracy over the baseline of Spark-based approximate calculation system utilizing the existing sampling modules in Apache Spark.

3). Approximate Incremental Processing. Incremental processing refers to a data computation that is incrementally scheduled by involving the same application logic over the input data [96] so as to avoid recomputing everything from scratch. Like approximate computation, it works over a subset of data items but differ in their choosing means. Krishnan *et al.* [110] observe that the two paradigms are complementary and proposed a new paradigm called approximate incremental processing that leverages the approximation and incremental techniques in order for a low-latency execution. They proposed an online stratified sampling algorithm by leveraging adaptation calculation to generate an incremental updated approximation with bounded error and executed it in Apache Spark Streaming by proposing a system called INCAPPROX. The experimental evaluation shows that benefits of INCAPPROX equipping with incremental and approximate computing.

7 High-Level Language Layer

Spark is designed in Scala [41], which is an object-oriented, functional programming language running on a JVM that can call Java libraries directly in Scala code and vice versa. Thus, it natively supports the Spark programming with Scala and Java by default. However, some users might be unfamiliar with Scala and Java but are skilled in other alternative languages like Python and R. Moreover, Spark programming is still a complex and heavy work especially for users that are not familiar with Spark framework. Thereby, having a high-level language like SQL declarative language on top of Spark is crucial for users to denote tasks while leave all complicated implementing majorization details to the backend Spark engine, which alleviates users' programming burdens significantly. In next section, we indicate the research work which has been proposed to address problems.

7.1 R and Python High-Level Languages Support

1) *SparkR*. In the numeric analysis and machine learning domains, R [39] is a popular programming

language widely used by data scientists for statistical computing and data analysis. SparkR [53], [151] is a light-weight frontend system that incorporates R into Spark and enables R programmers to perform large amount of data analysis from the R shell. It extends the single machine implementation of R to the distributed data frame implementation on top of Spark for large datasets. The implementation of SparkR is on the basis of Spark's parallel DataFrame abstraction [129]. It supports all Spark DataFrame analytical operations and functions including aggregation, filtering, grouping, summary statistics, and mixing-in SQL queries.

2) *PySpark.* PySpark [48] is the Python API for Spark, which exposes the Spark programming model to Python. It allows users to write Spark applications in Python. There are a few differences between PySpark and Spark Scala APIs. First, Python is a dynamically typed language so that the RDDs of PySpark have the capability to save objects of multiple types. Second, the RDDs of PySpark support the same functions as that of Scala APIs but leverage Python functions and return Python collection types. Third, PySpark supports anonymous functions, which can be passed to the PySpark API by using Python's lambda functions.

7.2 SQL-Like Programming Language and System

1). Shark. Apache Shark [91], [156] is the first SQL-on-Spark effort. It is built on top of Hive codebase and uses Spark as the backend engine. It leverages the Hive query compiler (HiveQL Parser) to analysis a HiveQL query and produce an abstract syntax tree followed by turning it into the logical plan and optimization. Shark then generates a physical plan of RDD operations and finally executes them in Spark system. A number of performance optimizations are considered. To reduce the large memory overhead of JVM, it executes a columnar memory storage based on Spark's native memory store. A cost-based query optimizer is also implemented in Shark for choosing more efficient join order according to table and column statistics. To reduce the impact of garbage collection, Shark saves all columns of primitive types as JVM primitive arrays. Finally, Shark is completely compatible with Hive and HiveQL, but much faster than Hive, due to its inter-query caching of data that eliminates the need to read/write repeatedly on disk. It can support more complex queries through User Defined Functions (UDFs) that are referenced by a HiveQL query.

2). Spark SQL. Spark SQL [129] is an evolution of SQL-on-Spark and the state-of-art new module of Spark that has replaced Shark in providing SQL-like interfaces. It is proposed and developed from ground-up to overcome the difficulty of performance optimization and maintenance of Shark resulting from inheriting a large, complicated Hive codebase. Compared to Shark, it adds two main capabilities. First, Spark SQL provides much tighter hybrid of relational and procedural processing. Second, it becomes easy for users to do some extensions, including adding composable rules, controling code generation, and defining extension points. It is compatible with Shark/Hive that supports all existing Hive data formats, user-defined functions (UDF) and the Hive metastore, while providing the state-of-the-art SQL performance.

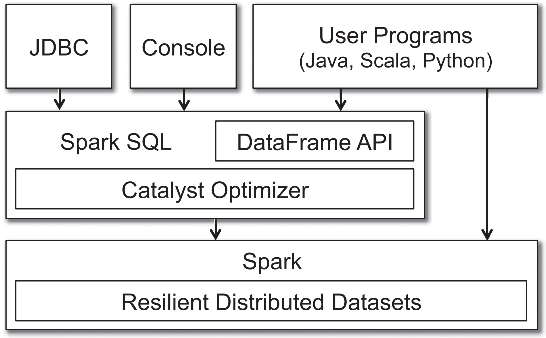


Fig. 5. Interfaces to Spark SQL [129].

Fig. 5 presents the programming interface to Spark SQL containing two main cores of DataFrame API and Catalyst Optimizer, and its interaction with Spark. It exposes SQL interfaces through a command line console such as JDBC or ODBC, and the DataFrame API implemented in Spark's procedural programming languages. The DataFrame is the main abstraction in Spark SQL's API. It is a distributed sets of records that enable to execute with Spark's supported API and new relational APIs. The Catalyst, in contrast, is a scalable query optimizer with functional programming constructs. It simplifies the addition of new optimization techniques and characteristics of Spark SQL and enables users to expand the optimizer for their application needs.

3). Hive/HiveQL. Apache Hive [147] is an open-source data warehousing method based on Hadoop by the Facebook Data Infrastructure Team. It aims to incorporate the classical relational database notion as well as high-level SQL language to the unstructured environment of Hadoop for those users who were not familiar with map-reduce. There is a mechanism inside Hive that can project the structure of table onto the data saved in HDFS and enable data queries using a SQL-like declarative language called HiveQL, which contains its own type system with support for tables, collections and nested compositions of the same and data definition language (DDL). Hive compiles the SQL-like query expressed in HiveQL into a directed acyclic graph of map-reduce jobs that are executed in Hadoop. There is a metastore component inside Hive that saves the metadata about underlying tables, which is particular during the creation and reused whenever the table is referenced in HiveQL. The DDL statements supported by HiveQL enable to create, drop and alter tables in Hive databases. Moreover, the data manipulation statements of HiveQL can be used to import data from external sources such as HBase and RCFile, and put query results into Hive tables.

Hive has been widely used by many organizations/users for their applications [8]. However, the default backend execution engine for Hive is MapReduce, which is less powerful than Spark. Adding Spark as an alternative backend execution engine to Hive is thus an important way for Hive users to migrate the execution to Spark. It has been realized in the latest version of Hive [23]. Users can now run Hive on top of Spark by configuring its backend engine to Spark.

4). Pig/Pig Latin. Apache Pig [24] is an open source dataflow processing system developed by Yahoo!, which serves

SQL
```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

Pig Latin
```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
         category, AVG(good_urls.pagerank);
```

Fig. 6. A instance of SQL Query and its equivalent Pig Latin program. [24].

for experienced procedural programmers with the preference of map-reduce style programming over the pure declarative SQL-style programming in pursuit of more control over the execution plan. It consists of a execution engine and high-level data flow language called *Pig Latin* [136], which is not declarative but enables the expression of a user's task with high-level declarative queries in the SQL spirit and low-level procedural programming with MapReduce. Fig. 6 gives a instance of SQL query and the Pig Latin program which has the same function, which is a sequence of transformation steps each of which is carried out using SQL-like high-level primitives such as filtering, grouping and aggregation. Given a Pig Latin program, the Pig execution engine generates a logic query plan, compiles it into a DAG of MapReduce jobs, and finally submitted to Hadoop cluster for execution.

There are several important characteristics for Pig Latin in casual ad-hoc data analysis, including the support of a nested data model as well as a set of predefined and customizable UDFs, and the capability of operating over raw data without the schema. The basic data type is Atom (e.g., integer, double, and string) in Pig Latin. Multiple Automs can be integrate into several Tuples which can form a Bag. Map is a complex data type supported by Pig Latin, which contains a key and a set of items that can be searched with its associated key.

Like Hive, the default backend execution engine for Pig is MapReduce. To enable the execution of Pig jobs on Spark for performance improvement, there is a Pig-on-Spark project called Spork [54] that plugs in Spark as an execution engine for Pig. With Spork, users can choose Spark as the backend execution engine of the Pig framework optionally for their own applications.

7.3 Comparison

Table 3 illustrates the comparison of different programming language systems used in Spark. To be compatible, it supports Hive and Pig by allowing users to replace the backend execution engine of MapReduce with Spark. To make the query efficient, Shark is first developed and later evolves to SparkSQL. Moroever, SparkR and PySpark are provided in Spark in order to support R and Python languages which are widely used by scientific users. Among these languages, the major differences lie in their supported language types. SparkR and PySpark can support Dataflow and SQL-like programming. In contrast, Shark, SparkSQL and Hive are SQL-like only languages, while Pig is a dataflow language.

8 APPLICATION/ALGORITHM LAYER

As a general-purpose system, Spark has been widely used for various applications and algorithms. In this section, we first review the support of machine learning algorithms on Spark. Next we show the supported applications on Spark.

8.1 Machine Learning Support on Spark

Machine learning is a powerful technique used to develop personalizations, recommendations and predictive insights in order for more diverse and more user-focused data products and services. Many machine learning algorithms involve lots of iterative computation in execution. Spark is an efficient in-memory computing system for iterative processing. In recent years, it attracts many interests from both academia and industry to build machine learning packages or systems based on Spark. We will discuss about research efforts on it in this section.

8.1.1 Machine Learning Library

1). MLlib. The largest and most active distributed machine learning library for Spark is MLlib [17], [128]. It contains fast and scalable executions of common machine learning algorithms and a variety of basic analytical utilities, low-level optimization primitives and higher-level pipeline APIs. It is a general machine learning library that provides algorithms for most use cases and meanwhile allows users to expand it for Professional utilization.

There are several core features for MLlib as follows. First, it implements a number of classic machine learning algorithms, including various linear models (e.g., SVMs, logistic regression, linear regression), naive Bayes, and random forest for classification and regression problems; alternating least squares for collaborative filtering; and k-means for clustering and dimensionality reduction; FP-growth for frequent pattern mining. Second, MLlib provides many optimizations for supporting efficient distributed learning and prediction. Third, It supports practical machine learning pipelines natively by using a package called spark.ml inside MLlib, which simplifies the adjustment of multi-stage learning pipelines by offering unified high-level APIs. Lastly, there is a tight and seamless integration of MLlib with Spark's other components including Spark SQL, GraphX, Spark streaming and Spark core, bringing in high

TABLE 3
The Comparison of Different Programming Language Systems

System	Language Type	Data Model	UDF	Access Interface	MetaStore
SparkR	Dataflow, SQL-like	Nested	Supported	Command line, web, JDBC/ODBC server	Supported
PySpark	Dataflow, SQL-like	Nested	Supported	Command line, web, JDBC/ODBC server	Supported
Shark	SQL-like	Nested	Supported	Command line	Supported
SparkSQL	SQL-like	Nested	Supported	Command line, web, JDBC/ODBC server	Supported
Hive	SQL-like	Nested	Supported	Command line, web, JDBC/ODBC server	Supported
Pig	Dataflow	Nested	Supported	Command line	Not supported

performance improvement and various functionality support for MLlib.

MLlib has many advantages, including simplicity, scalability, streamlined end-to-end and compatibility with Spark's other modules. It has been widely used in many real applications like marketing, advertising and fraud detection.

2). KeystoneML. KeystoneML [143] is a framework for ML pipelines, from the UC Berkeley AMPLab aimed to simplify the architecture of machine learning pipelines with Apache Spark. It enables high-throughput training in a distributed environment with a high-level API [58] for the end-to-end large-scale machine learning applications. KeystoneML has several core features. First, users can specify machine learning pipelines in a system with high-level logical operators. Second, as the amount of data and the complexity of the problem change, it expands dynamically. Finally, it automatically improves these applications by a library of operators and users resources. KeystoneML is open source and being applied in scientific applications about solar physics [104] and genomics [31].

3). Thunder. Thunder [55] is an open-source library developed by Freeman Lab [32] for large-scale neural data analysis with Spark. It is desiged by PySpark APIs for robust numerical and scientific computing libraries (e.g., NumPy and SciPy), and offers the simplest front end for new users. Thunder provides a set of data structures and uses to load and storing data with a amount of input formats and classes for processing distributed data of spatial and temporal, and modular functions such as time series analysis, image processing, factorization and model fitting [92]. It can be used in many fileds involving medical imaging, neuroscience, video processing, and geospatial and climate analysis.

4). ADAM. ADAM [56] is a library and parallel framework that enables to work with both aligned and unaligned genomic data using Apache Spark across cluster/cloud computing environments. ADAM provides competitive performance to optimized multi-threaded tools on a single node, while enabling scale out to clusters with more than a thousand cores. ADAM is built as a modular stack where it supports a wide range of data formats and optimizes query patterns without changing data structures, which is different from traditional genomics tools that are not flexible and only targeted at a certain kind of applications or functions [61]. There are seven layers of the stack model from bottom to top: Physical Storage, Data Distribution, Materialized Data, Data Schema, Evidence Access, Presentation, Application [127]. A "narrow waisted" layering model is developed for building similar scientific analysis systems to enforce data independence. This stack model separates computational patterns from the data model, and the data model from the serialized representation of the data on disk. They exploit smaller and less expensive machines, resulting in a 63 percent cost improvement and a 28× improvement in read preprocessing pipeline latency [135].

8.1.2 Machine Learning System

In the current era of Artificial Intelligence (AI), there is a trend that data and AI should be unified together given that a large amount of constantly updated training data are often required to build state-of-the-art models for AI applications. Spark is the only unified analytics system that integrates large-scale data processing with sate-of-the-art machine learning and AI algorithms so far [62].

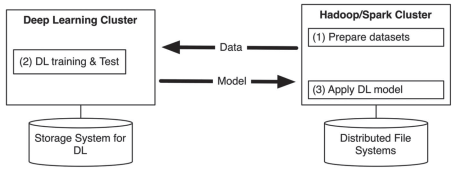
1). MLBase. The complexity of existing machine learning algorithms is so overwhelming that users often do not understand the trade off and difficults of parameterizing and picking up between different learning algorithms for achieving good performance. Moreover, existing distributed systems that support machine learning often require ML researchers to have a strong background in distributed systems and low-level primitives. All of these limits the wide use of machine learning technique for large scale data sets seriously. MLBase [109], [145] is then proposed to address it as a platform.

2). Sparkling Water. H2O [33] is a fast, scalable, open-source, commercial machine learning system produced by *H2O.ai Inc.* [34] with the implementation of many common machine learning algorithms including generalized linear modeling (e.g., linear regression, logistic regression), Naive Bayes, principal components analysis and k-means clustering, as well as advanced machine learning algorithms like deep learning, distributed random forest and gradient boosting. It provides familiar programming interfaces like R, Python and Scala, and a graphical-user interface for the ease of use. To utilize the capabilities of Spark, Sparkling Water [52] integrates H2O's machine learning engine with Spark transparently. It enables launching H2O on top of Spark and using H2O algorithms and H2O Flow UI inside the Spark cluster, providing an ideal machine learning platform for application developers.

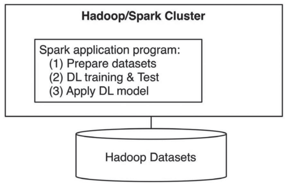
Sparking Water is designed as a regular Spark application and launched inside a Spark executor spawned after submitting the application. It offers a method to initialize H2O services on each node of the Spark cluster. It enables data sharing between Spark and H2O with the support of transformation between different types of Spark RDDs and H2O's H2OFrame, and vice versa.

3). Splash. It is efficient to address machine learning and optimization problems with Stochastic algorithms. Splash [165] is a framework for speeding up stochastic algorithms, which are efficient approaches to address machine learning and optimization problems, on distributed computing systems. It makes up of a programming interface and an execution engine. Users can develop sequential stochastic algorithms with programming interface and then the algorithm is automatically parallelized by a communication-efficient execution engine. It can call Splash framwork to construct parallel algorithms by execution engine of Splash in a distributed manner. With distributed versions of averaging and reweighting approach, Splash can parallelize the algorithm by converting a distributed processing task into a sequential processing task. Reweighting scheme ensures the total load handled by individual thread is same as the number of samples in full sequence. It indicates a single thread to produce a complete update of completely unbiased estimates. Splash automatically discerns the optimal parallelism for this algorithm by using the approach. The experiments show that Splash outperforms the prior art algorithms of single-thread stochastic and batch by an order of magnitude.

4). Velox. BDAS(Berkeley Data Analytics Stack) contained a data storage manager, a dataflow execution engine, a



(a) ML Pipeline with multiple programs on separated clusters.



(b) ML Pipeline with single program on one cluster.

Fig. 7. Distributed deep learning computing model. [26].

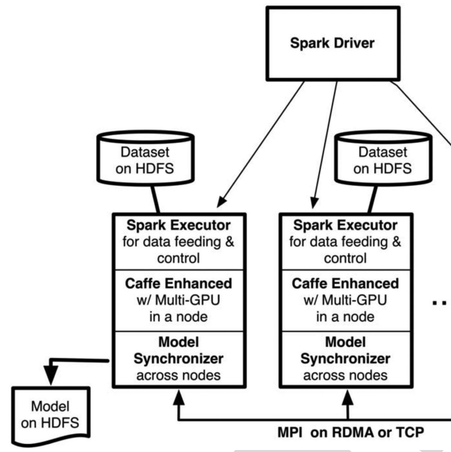


Fig. 8. CaffeOnSpark Architecture. [26].

stream processor, a sampling engine, and a set of advanced analytics packages. But BDAS has insufficiencies in the way to offer users actually data, and industrial users of the stack have come up with their solutions to model services and management. Velox [84] fills the gap which is a system for executing model services and model maintenance in proportion. It offers a low-latency, intuitive model interface for applications and services. Moreover, it transforms the original statistical model which is currently trained by offline computing frameworks into a complete end-to-end data recommending products such as target advertisements and web content. Velox consists of two key element of construction: Velox model predictor and manager. Velox model manager orchestrates the computation and maintenance of a set of pre-declared machine learning models, incorporating feedback, evaluating the capability of models and retraining models if necessary.

Deep Learning. As a class of machine learning algorithms, *Deep learning* has become very popular and been widely used in many fields like computer version, speech recognition, natural language processing and bioinformatics due to its many benefits: accuracy, efficiency and flexibility. There are a number of deep learning frameworks implemented on top of Spark, such as CaffeOnSpark [25], DeepLearning4j [37], and SparkNet [131].

5). CaffeOnSpark. In many existing distributed deep learning, the model training and model usage are often separated, as the computing model shown in Fig. 7a. There is a big data processing cluster (e.g., Hadoop/Spark cluster) for application computation and a separated deep learning cluster for model training. To integrate the model training and model usage as a united system, it requires a large amount of data and model transferred between two separated clusters by creating multiple programs for a typical machine learning pipeline, which increases the latency and system complexity for end-to-end learning. In contrast, an alternative computing model, as illustrated in Fig. 7b, is to conduct the deep learning and data processing in the same cluster.

Caffe [103] is a popular deep learning framework, which is developed in C++ with CUDA by Berkeley Vision and Learning Center (BVLC). According to the model of Fig. 7b, Yahoo extends Caffe to Spark framework by developing CaffeOnSpark [25], [26], which supports distributed deep learning on a cluster consisting of GPU and CPU machines. CaffeOnSpark is a Spark package for deep learning, as a complementary to non-deep learning libraries MLlib and Spark SQL.

The architecture of CaffeOnSpark is shown in Fig. 8. It can launch Caffe engines within the Spark executor on GPU or CPU devices by invoking a JNI layer with fine-grain memory management. Moreover, to achieve similar performance as dedicated deep learning clusters, CaffeOnSpark takes Spark+MPI architecture, which leverages MPI allreduce style interface for the network communication across CaffeOnSpark executors by TCP/Ethernet or RDMA/Infiniband.

6). Deeplearning4j/dl4j-spark-ml. Deeplearning4j [37] is the first commercial grade but open source, distributed deep learning library designed for Java and Scala, and a computing framework with the support and implementation of many deep learning algorithms, including restricted Boltzmann machine, deep belief net, deep autoencoder, stacked denoising autoencoder and recursive neural tensor network, word2vec, doc2vec and GloVe. It integrates with Spark via a Spark package called *dl4j-spark-ml* [47], which provides a set of Spark components including DataFrame Readers for MNIST, Labeled Faces in the Wild (LFW) and IRIS, and pipeline components for NeuralNetworkClassification and NeuralNetworkReconstruction. It supports heterogeneous architecture by using Spark CPU to drive GPU coprocessors in a distributed context.

7). SparkNet. SparkNet [29], [131] is an open-source, distributed system for training deep network in Spark released by the AMPLab at U.C. Berkley in Nov 2015. It is based on Spark and Caffe, where Spark works for distributed data processing and Caffe framework is responsible for the core learning process. SparkNet can read data from Spark RDDs through interfaces which is compatible to Caffe. It achieves a good scalability and tolerance of high-latency communication by utilizing a simple palatalization scheme for stochastic gradient descent. It also allows Spark users to construct deep networks using existing deep learning libraries or systems, such as TensorFlow [64] or Torch as a backend, instead of building a new deep learning library in Java or Scala. Such a new integrated model of combining existing model training frameworks with existing batch frameworks is beneficial in practice. For example, machine learning often involves a set of pipeline tasks such as data retrieving, cleaning and processing before model training as well as model deployment and model prediction after training. All of these can be well handled with the existing data-processing pipelines in today's distributed computational environments such as Spark. Moreover, the integrated model of SparkNet can inherit the in-memory computation from Spark that data can be cached in memory to complete for fast computation, instead of writing to disk between operations as a segmented approach does. It also allows machining learning algorithm easily to pipeline with Spark's other components such as Spark SQL and GraphX.

Moreover, there are some other Spark-based deep learning libraries and frameworks, including OpenDL [18], DeepDist [15], dllib [57], MMLSpark [60], and DeepSpark [106]. OpenDL [18] is a deep learning training library based on Spark by applying the similar idea used by DistBelief [86]. It executes the distributed training by splitting the training data into different data shards and synchronizes the replicate model using a centralized parameter server. DeepDist [15] accelerates model training by offering asynchronous stochastic gradient descent for data saved on HDFS. Dllib [57] is a distributed deep learning framework based on Apache Spark. It offers a simple interface for users to write and run deep learning algorithms on spark. For MMLSpark [60], it provides users with a set of deep learning tools for Spark, For example, it enables seamless integration of Spark Machine Learning pipelines with Microsoft Cognitive Toolkit (CNTK) and OpenCV as well as the creation of powerful, highly-scalable predictive and analytical models for large image and text datasets quickly. DeepSpark [106] is an alternative deep learning framework similar to SparkNet. It integrates three components including Spark, asynchronous parameter updates, and GPU-based Caffe seamlessly for enhanced large-scale data processing pipeline and accelerated DNN training.

8.2 Spark Applications

As an efficient data processing system, Spark has been widely used in many application domains, including Genomics, Medicine&Healthcare, Finance, and Astronomy, etc.

8.2.1 Genomics

Due to its computational efficiency and good adaptive capability for simple and complex phenotypes, the effective scoring statistical method is widely applied for the inference of high-throughput genomic data. To solve the problem of resulting calculation for resampling based inference, it is need a scalable distributed computing approach. Cloud computing platforms are appropriate, because they allow users to analyze data at a modest cost without access to mainframe computer infrastructure. SparkScore [71] is a series of distributed computing algorithms executed in Spark. It uses the awkward parallel nature of genomic resampling inference based on effective score statistics. This calculation takes advantage of Spark's fault-tolerant features and can be easily expanded to analyze DNA and RNA sequencing data such as expression of quantitative feature loci (eQTL) and phenotypic association studies. Experiments with synthetic datasets show the efficiency and scalability of SparkScore, including large-capacity resampling of Big Data, under Amazon Elastic MapReduce (EMR) cluster. To study the utility of Spark in the genomic context, SparkSeq [155] was proposed, which executes in-memory computings on the Cloud via Apache Spark. It is a versatile tool for RNA and DNA sequencing analysis for processing in the cloud. Several operations on generic alignment format (e.g., Binary Alignment/Map (BAM) format and Sequence Alignment/Map (SAM) format [117]) are provided, including filtering of reads, summarizing genomic characteristics and basic statistical analyses operations. Moreover, SparkSeq makes it possible to customize secondary analyses and iterate the algorithms of machine learning. Spark-DNAligning [68] is an acceleration system for DNA short reads alignment problem by exploiting Spark's performance optimizations, including caching, broadcast variable, join after partitioning, and in-memory computations. SPARK-MSNA [152] is a multiple sequence alignment (MSA) system for massive number of large sequences, which is promised to achieve a better alignment accuracy and comparable execution time than state-of-the-art algorithms (e.g., HAlign II).

8.2.2 Medicine & Healthcare

In a modern society with great pressure, more and more people trapped in health issues. In order to reduce the cost of medical treatments, many organizations were devoted to adopting big data analytics into practice so as to avoid cost. Large amount of healthcare data is produced in healthcare industry but the utilization of those data is low without processing this data interactively in real-time [69]. Now it is possible to process real time healthcare data with Spark given that Spark supports automated analytics by iterative processing on large data set. But in some circumstances the quality of data is poor, which brings a big problem. To generate an accurate data mart, a spark-based data processing and probability record linkage method is proposed [72]. This approach is specifically designed to support data quality assessment and database connectivity by the Brazilan Ministry of Health and the Ministry of Social Development and Hunger Reduction. Moreover, to study the sensitivity of drug, Hussain *et al.* [99] make a prediction analysis of the drug targets in the base of cancer cell line using various machine learning algorithms such as support vector machine, logistic regression, random forest from MLlib of Spark.

8.2.3 Finance

Big data analytic technique is an effective way to provide good financial services for users in financial domain. For stock market, to have an accurate prediction and decision on the market trend, there are many factors such as politics and social events needed to be considered. Mohamed *et al.* [142] propose a real-time prediction model of stock market trends by analyzing big data of news, tweets, and historical price with Apache Spark. The model supports the offline mode that works on historical data, and real-time mode that works on real-time data during the stock market session. Li *et al.* [45] builds a quantitative investing tool based on Spark that can be used for macro timing and portifolio rebalancing in the market.

To protect user's account during the digital payment and online transactions, fraud detection is a very important issue in financial service. Rajeshwari *et al.* [148] study the credit card fraud detection. It takes Spark streaming data processing to provide real-time fraud detection based on Hidden Markov Model (HMM) during the credit card transaction by analyzing its log data and new generated data. Carcillo *et al.* [77] propose a realistic and scalable fraud detection system called Real-time Fraud Finder (SCARFF). It uses a machine learning approach to integrate Big Data softwares including Kafka, Spark and Cassandra by dealing with class imbalance, nonstationarity and verification latency.

Moreover, there are some other financial applications such as financial risk analysis [7], financial trading [90].

8.2.4 Astronomy

Considering the technological advancement of telescopes and the number of ongoing sky survey projects, it is safe to say that astronomical research is moving into the Big Data era. Sky surveys provide a huge data set that can be used simultaneously for various scientific researches. Kira [166], a flexible distributed astronomy image processing toolkit based on Spark, is proposed to execute a Source Extractor application and the extraction accuracy can be improved. To support the task of querying and analyzing arbitrarily large astronomical catalogs, AXS [162] is proposed. It first enables efficient online positional cross-matching in Spark. Second, it provide a Python library for commonly-used operations on astronomical data. Third, it implements ZONES algorithm for scalable cross-matching. Moreover, there are some other work on Astronomy such as spatial data analysis [154], [158].

9 Challenges and Open Issues

In this section, we discuss research issues and opportunities for Spark ecosystem.

Memory Resource Management. As an in-memory processing platform built with Scala, Spark's performance is sensitive to its memory configuration and usage of JVMs. The memory resource is divided into two parts. One is for RDD caching. The other is used for tasks' working memory to store objects created during the task execution. The proper configuration of such memory allocation is non-trivial for performance improvement. Moreover, the overhead of JVM garbage collection can be a challenge when there are a amount of "churn" for cached RDDs, or due to serious interference between the cached RDDs and tasks' working memory. For this, Maas *et al.* [122] have a detailed study for GC's impact on Spark in distributed environment. The proper tuning of GC thus plays an important role in performance optimization. Currently, it is still at early stage and there are not good solutions for Spark. It opens an important issue on the memory resource management and GC tuning for Spark. Regarding this, recently, Spark community starts a new project for Spark called Tungsten [4] that places Spark's memory management as its first concern.

New Emerging Processor Support. In addition to GPU and FPGA, the recent advancement on computing hardware make some new processors emerged, such as APU [75] and TPU [105], etc. These can bring new opportunities to enhance the performance of Spark system. For example, APU is a coupled CPU-GPU device that incorporates the CPU and the GPU into a single chip so that the CPU and the GPU can communicate with each other by the shared physical memory via featuring shared memory space between them [75]. It can improve the performance of existing discrete CPU-GPU architecture where CPU and GPU communicate via PCI-e bus. TPU is a domain-specific processor for deep neural network. It can give us a chance to speedup Spark for deep learning applications by migrating Spark to TPU platform.

Heterogenous Accelerators Support. Besides emerging processors, it could be possible in practice that a Spark computing system consists of a number of diverse processors such as CPU, GPU, FPGA and MIC as illustrated in Spark ecosystem of Fig. 1. Rather than supporting a single processor only, it is crucial to have a upgraded Spark that can utilize all of the computing devices simultaneously for maximum performance. Due to the fact that different accelerators are based on different programming models (e.g., CUDA for GPU, OpenCL for FPGA), it open us a new challenge on how to support such different types of accelerators for Spark at the same time.

RDD Operation and Sharing. There are several open issues for current Spark's RDD. First, it allows only coarse-grained operations (i.e., one operation for all data) on RDDs, whereas the fine-grained operations (e.g., partial read) are supported. One work is to design some fine-grained operations on partial data of RDD. Second, current RDDs are immutable. Instead of modifying on existing RDD, any update operation would generate new RDD, some data of which can be redundant and thus results in a wast of storage resource. Third, for a RDD, its data partitions can be skewed, i.e., there are many small partitions coupled with a few number of large-size partitions. Moreover, a Spark task computation generally involves a series of pipelined RDDs. Thus, the skewed RDD partitions can easily incur the chained unbalanced problem for tasks, which causes some workers much busier than others. Fourth, Spark itself does not support RDD sharing across applications. For some applications that have the same input data or redundant task computation, enabling RDD sharing can be an approach to improve the performance of the whole applications.

Failure Recovery. In contrast to MapReduce that provides fault tolerance through replication or checkpoint, Spark achieves failure recovery via lineage re-computation, which is much more cost efficient since it saves costs caused by data replication between network and disk storage. The lineage information (e.g., input data, computing function) for each RDD partition is recorded. Any lost data of RDDs can be recovered through re-computation based on its lineage information. However, there is a key assumption that all RDD lineage information is kept and always available, and the driver does not fail. It means that Spark is not 100 percent fault tolerance without overcoming this assumption. It thus remains us an open issue on how to enhance fault tolerance for Spark.

5G Network. The upcoming of 5G is supposed to significantly improve the bandwidth and reduce the latency of communication network, bringing new opportunities for many research area and applications including Internet of Things (IoT), autonomous driving, augmented and virtual reality (AR/VR) services [89]. The high speed of 5G enables the application data from mobile devices to be transfered to remote servers directly for (realtime) computation. It implies that there can be more opportunities for Spark to handle streaming computation applications. In this situation, one open issue is about the security enhancement of 5G data during the Spark computation given the existing poor security mechanism of Spark. Another opportunity driven by 5G can be that we can establish a mobile Spark cluster for data computation using mobile devices such as smart phones and smart tablets under the 5G network. In this case, one open issue can be that the communication network would be no longer a bottleneck. Instead, the electricity power of mobile devices can then be the major concern.

10 Conclusion

Spark has gained significant interests and contributions both from industry and academia because of its simplicity, generality, fault tolerance, and high performance. However, there is a lack of work to summarize and classify them comprehensively. In view of this, it motives us to investigate the related work on Spark. We first overview the Spark framework, and present the pros and cons of Spark. We then provide a comprehensive review of the current status of Spark studies and related work in the literature that aim at improving and enhancing the Spark framework, and give the open issues and challenges regarding the current Spark finally. In summary, we hopefully expect to see that this work can be a useful resource for users who are interested in Spark and want to have further study on Spark.

Acknowledgments

This work is sponsored by the National Natural Science Foundation of China (61972277) and Tianjin Natural Science Foundation (18JCZDJC30800). Ce Yu was supported by the Joint Research Fund in Astronomy (U1731243, U1931130) under cooperative agreement between the National Natural Science Foundation of China (NSFC) and Chinese Academy of Sciences (CAS). Bingsheng was supported by a MoE AcRF Tier 1 grant (T1 251RES1610) and an NUS startup grant in Singapore.

References

[1] Apache flink, 2019. [Online]. Available: https://flink.apache.org/
[2] Apache spark as a compiler: Joining a billion rows per second on a laptop, 2016. [Online]. Available: https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a- billion-rows-per-second-on-a-laptop.html
[3] Decision cep, 2016. [Online]. Available: http//github.com/stratio/decision
[4] Project tungsten: Bringing apache spark closer to bare metal, 2015. [Online]. Available: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-close r-to-bare-metal.html
[5] Spark cep, 2017. [Online]. Available: https://github.com/samsung/spark-cep
[6] Streamdm, 2018. [Online]. Available: http: http//huawei-noah.github.io/streamdm/
[7] Estimating financial risk with apache spark, 2014. [Online]. Available: https://blog.cloudera.com/blog/2014/07/estimating-financial-risk-with-apache-spark/
[8] Shark, spark SQL, hive on spark, and the future of SQL on apache spark, 2014. [Online]. Available: https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html
[9] Apache hbase, 2015. [Online]. Available: http://hbase.apache.org/
[10] Apache knox gateway, 2015. [Online]. Available: http://hortonworks.com/hadoop/knox-gateway/
[11] Apache ranger, 2015. [Online]. Available: http://hortonworks.com/hadoop/ranger/
[12] Apache security, 2015. [Online]. Available: https://spark.apache.org/docs/latest/security.html
[13] Apache spark, 2015. [Online]. Available: https://spark.apache.org/
[14] Apache storm, 2015. [Online]. Available: https://storm.apache.org/
[15] Deepdist: Lightning-fast deep learning on spark via parallel stochastic gradient updates, 2015. [Online]. Available: http://deepdist.com/
[16] Introducing sentry, 2015. [Online]. Available: http://www.cloudera.com/content/cloudera/en/campaign/introducing-sentry.html
[17] Machine learning library (MLlib) guide, 2015. [Online]. Available: https://spark.apache.org/docs/latest/mllib-guide.html
[18] OpenDL: The deep learning training framework on spark, 2015. [Online]. Available: https://github.com/guoding83128/OpenDL/
[19] Alluxio, formerly known as tachyon, is a memory speed virtual distributed storage system, 2016. [Online]. Available: http://www.alluxio.org/
[20] Amazon DynamoDB, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Amazon_DynamoDB
[21] Amazon S3, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Amazon_S3
[22] Apache cassandra, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Apache_Cassandra
[23] Apache hive, 2016. [Online]. Available: https://github.com/apache/hive
[24] Apache pig, 2016. [Online]. Available: https://pig.apache.org/
[25] Caffeonspark, 2016. [Online]. Available: https://github.com/yahoo/CaffeOnSpark
[26] Caffeonspark open sourced for distributed deep learning on big data clusters, 2016. [Online]. Available: http://yahoohadoop.tumblr.com/post/139916563586/caffeonspark-open-sourced-for-distributed-deep
[27] Cloud storage, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Cloud_storage
[28] Distributed hash table, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_hash_table
[29] Distributed neural networks for spark, 2016. [Online]. Available: https://github.com/amplab/SparkNet
[30] Dynamodb data source for apache spark, 2016. [Online]. Available: https://github.com/traviscrawford/spark-dynamodb
[31] Encode-dream in-vivo transcription factor binding site prediction challenge, 2016. [Online]. Available: https://www.synapse.org/#!Synapse:syn6131484
[32] Freeman lab, 2016. [Online]. Available: https://www.janelia.org/lab/freeman-lab
[33] H2O, 2016. [Online]. Available: https://github.com/h2oai/h2o-3
[34] H2O.ai, 2016. [Online]. Available: http://www.h2o.ai/
[35] Introduction to microsoft azure storage, 2016. [Online]. Available: https://azure.microsoft.com/en-us/documentation/articles/storage-introduction/
[36] Medium, 2016. [Online]. Available: https://medium.com/
[37] Open-source, distributed, deep-learning library for the JVM, 2016. [Online]. Available: http://deeplearning4j.org/
[38] Pyspark cassandra, 2016. [Online]. Available: https://github.com/TargetHolding/pyspark-cassandra
[39] The R project for statistical computing, 2016. [Online]. Available: https://www.r-project.org/
[40] S3 support in apache hadoop, 2016. [Online]. Available: http://wiki.apache.org/hadoop/AmazonS3
[41] Scala language, 2016. [Online]. Available: https://spark.apache.org/docs/latest/api/python/index.html

[42] Spark cassandra connector, 2016. [Online]. Available: https://github.com/datastax/spark-cassandra-connector

[43] Spark-gpu wiki, 2016. [Online]. Available: https://github.com/kiszk/spark-gpu

[44] Spark-hbase connector, 2016. [Online]. Available: https://github.com/nerdammer/spark-hbase-connector

[45] Spark-in-finance-quantitative-investing, 2016. [Online]. Available: https://github.com/litaotao/Spark-in-Finance-Quantitative-Investing

[46] spark-on-hbase, 2016. [Online]. Available: https://github.com/michal-harish/spark-on-hbase

[47] Spark package - dl4j-spark-ml, 2016. [Online]. Available: https://github.com/deeplearning4j/dl4j-spark-ml

[48] Spark python API, 2016. [Online]. Available: http://spark.apache.org/docs/latest/api/python/index.html

[49] Spark python API docs, 2016. [Online]. Available: http://www.scala-lang.org/

[50] spark-S3, 2016. [Online]. Available: https://github.com/knoldus/spark-s3

[51] Spark-SQL-on-hbase, 2016. [Online]. Available: https://github.com/Huawei-Spark/Spark-SQL-on-HBase

[52] Sparkling water, 2016. [Online]. Available: https://github.com/h2oai/sparkling-water

[53] Sparkr (R on spark), 2016. [Online]. Available: https://spark.apache.org/docs/latest/sparkr.html

[54] Spork: Pig on apache spark, 2016. [Online]. Available: https://github.com/sigmoidanalytics/spork

[55] Thunder: Large-scale analysis of neural data, 2016. [Online]. Available: http://thunder-project.org/

[56] Adam, 2017. [Online]. Available: https://adam.readthedocs.io/en/latest/

[57] dllib, 2017. [Online]. Available: https://github.com/Lewuathe/dllib

[58] Keystoneml API docs, 2017. [Online]. Available: http://keystone-ml.org/

[59] Databricks cache boosts apache spark performance-why NVMe SSDs improve caching performance by 10x, 2018. [Online]. Available: https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html

[60] Mmlspark: Microsoft machine learning for apache spark, 2018. [Online]. Available: https://github.com/Azure/mmlspark

[61] Bioinformatics tools for genomics, 2019. [Online]. Available: https://omictools.com/genomics2-category

[62] Spark+ai summit 2020, 2020. [Online]. Available: https://databricks.com/sparkaisummit/north-america

[63] A. Davidson and A. Or, "Optimizing shuffle performance in spark," Univ. California, Berkeley - Dept. Elect. Eng. Comput. Sci., Tech. Rep., 2013.

[64] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.

[65] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.

[66] S. Agarwal *et al.*, "Knowing when you're wrong: Building fast and reliable approximate query processing systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, pp. 481–492, 2014.

[67] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with bounded errors and bounded response times on very large data," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 29–42.

[68] M. AlJame and I. Ahmad, "DNA short read alignment on apache spark," *Appl. Comput. Informat.*, 2019.

[69] J. Archenaa and E. A. M. Anita, *Interactive Big Data Management in Healthcare Using Spark*. Berlin, Germany: Springer, 2016.

[70] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 211–224.

[71] A. Bahmani, A. B. Sibley, M. Parsian, K. Owzar, and F. Mueller, "SparkScore: Leveraging apache spark for distributed genomic inference," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 435–442.

[72] M. Barreto, R. Pita, C. Pinto, M. Silva, P. Melo, and D. Rasella, "A spark-based workflow for probabilistic record linkage of healthcare data," in *Proc. The Workshop Algorithms Syst. MapReduce Beyond*, 2015, pp. 17–26.

[73] A. Bifet, S. Maniu, J. Qian, G. Tian, C. He, and W. Fan, "StreamDM: Advanced data mining in spark streaming," in *Proc. IEEE Int. Conf. Data Mining Workshop*, 2015, pp. 1608–1611.

[74] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: A survey," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 1–28, Mar. 2005.

[75] A. Branover, D. Foley, and M. Steinman, "AMD fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, Mar. 2012.

[76] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," 2015, *arXiv:1506.08603*.

[77] F. Carcillo, A. D. Pozzolo, Y. L. Borgne, O. Caelen, Y. Mazzer, and G. Bontempi, "SCARFF: A scalable framework for streaming credit card fraud detection with spark," *Inf. Fusion*, vol. 41, pp. 182–194, 2018.

[78] J. L. Carlson, *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013.

[79] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," in *Proc. 7th USENIX Symp. Operating Syst. Des. Implementation*, 2006, pp. 15–15.

[80] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When spark meets FPGAs: A case study for next-generation DNA sequencing acceleration," in *Proc. 8th USENIX Conf. Hot Topics Cloud Comput.*, 2016.

[81] W. Cheong *et al.*, "A flash memory controller for 15us ultra-low-latency SSD using high-speed 3D NAND flash with 3us read time," in *Proc. IEEE Int. Solid - State Circuits Conf.*, 2018, pp. 338–340.

[82] W. Choi and W. K. Jeong, "Vispark: GPU-accelerated distributed visual computing using spark," in *Proc. IEEE 5th Symp. Large Data Anal. Vis.*, 2015, pp. 125–126.

[83] J. Cong, M. Huang, D. Wu, and C. H. Yu, "Invited - Heterogeneous datacenters: Options and opportunities," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, pp. 16:1–16:6.

[84] D. Crankshaw *et al.*, "The missing piece in complex analytics: Low latency, scalable model management and serving with velox," *Eur. J. Obstetrics Gynecol. Reproductive Biol.*, vol. 185, pp. 181–182, 2014.

[85] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[86] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1232–1240.

[87] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, pp. 10–10.

[88] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[89] A. Y. Ding and M. Janssen, "Opportunities for applications using 5G networks: Requirements, challenges, and outlook," in *Proc. 7th Int. Conf. Telecommun. Remote Sens.*, 2018, pp. 27–34.

[90] K. Dutta and M. Jayapal, "Big data analytics for real time systems," *Big Data Analytics Seminar*, pp. 1–13, 2015.

[91] C. Engle *et al.*, "Shark: Fast data analysis using coarse-grained distributed memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 689–692.

[92] J. Freeman *et al.*, "Mapping brain activity at scale with cluster computing," *Nature Methods*, vol. 11, no. 9, pp. 941–950, 2014.

[93] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[94] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[95] M. A. Gulzar *et al.*, "BigDebug: Debugging primitives for interactive big data processing in spark," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2016, pp. 784–795.

[96] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 75–88.

[97] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.

[98] Z. Hu, B. Li, and J. Luo, "Time- and cost- efficient task scheduling across geo-distributed data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 705–718, Mar. 2018.

[99] S. Hussain, J. Ferzund, and R. Ul-Haq, "Prediction of drug target sensitivity in cancer cell lines using apache spark," *J. Comput. Biol.*, vol. 26, no. 8, pp. 882–889, 2019.

[100] M. Interlandi *et al.*, "Adding data provenance support to apache spark," *The VLDB J.*, vol. 27, no. 5, pp. 595–615, 2018.

[101] M. Interlandi *et al.*, "Titian: Data provenance support in spark," *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 216–227, 2015.

[102] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[103] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[104] E. Jonas, M. Bobra, V. Shankar, J. T. Hoeksema, and B. Recht, "Flare prediction using photospheric and coronal image data," *Sol. Phys.*, vol. 293, no. 3, p. 48, 2018.

[105] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and G. Agrawal, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 1–12.

[106] H. Kim, J. Park, J. Jang, and S. Yoon, "DeepSpark: Spark-based deep learning supporting asynchronous updates and caffe compatibility," 2016, *arXiv:1602.08191*.

[107] M. Kim *et al.*, "Sparkle: Optimizing spark for large memory machines and analytics," 2017, *arXiv:1708.05746*.

[108] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I. Jordan, "A general bootstrap performance diagnostic," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2013, pp. 419–427.

[109] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "*MLbase*: A distributed machine-learning system," in *Proc. 6th Biennial Conf. Innovative Data Syst. Res.*, 2013, vol. 1, pp. 2–1.

[110] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "IncApprox: A data analytics system for incremental approximate computing," in *Proc. 25th Int. Conf. World Wide Web*, 2016, pp. 1133–1144.

[111] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[112] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style processing of fast data," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, Aug. 2012.

[113] D. Le Quoc, R. Chen, P. Bhatotia, C. Fetze, V. Hilt, and T. Strufe, "Approximate stream analytics in apache flink and apache spark streaming," 2017, *arXiv:1709.02946*.

[114] D. Le Quoc *et al.*, "Approximate distributed joins in apache spark," 2018, *arXiv:1805.05874*.

[115] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Memory throughput I/O for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–15.

[116] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 6:1–6:15.

[117] H. Li *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[118] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU spark platform for machine learning algorithms," in *Proc. IEEE Int. Conf. Netw. Architecture Storage*, 2015, pp. 347–348.

[119] H. Liu *et al.*, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 26:1–26:10.

[120] S. Liu, H. Wang, and B. Li, "Optimizing shuffle in wide-area data analytics," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 560–571.

[121] X. Lu, Md. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with RDMA for big data processing: Early experiences," in *Proc. IEEE 22nd Annu. Symp. High-Perform. Interconnects*, 2014, pp. 9–16.

[122] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 457–471.

[123] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *Proc. 15th USENIX Conf. Hot Topics Operating Syst.*, 2015, Art. no. 1.

[124] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[125] D. Manzi and D. Tompkins, "Exploring GPU acceleration of apache spark," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 222–223.

[126] O. Marcu, A. Costan, G. Antoniu, and M. S. Perez-Hernandez, "Spark versus flink: Understanding performance in big data analytics frameworks," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2016, pp. 433–442.

[127] M. Massie *et al.*, "Adam: Genomics formats and processing patterns for cloud scale computing," University of California at Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2013–207, 2013.

[128] X. Meng *et al.*, "MLlib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.

[129] A. Michael *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.

[130] G. D. F. Morales and A. Bifet, "SAMOA: Scalable advanced massive online analysis," *J. Mach. Learn. Res.*, vol. 16, pp. 149–153, 2015.

[131] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training deep networks in spark," 2015, *arXiv:1511.06051*.

[132] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[133] B. Nicolae, C. H. A. Costa, C. Misale, K. Katrinis, and Y. Park, "Leveraging adaptive I/O to optimize collective data shuffling patterns for big data analytics," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1663–1674, Jun. 2017.

[134] R. Nishtala *et al.*, "Scaling memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[135] F. A. Nothaft *et al.*, "Rethinking data-intensive science using scalable analytics systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 631–646.

[136] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.

[137] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 69–84.

[138] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, pp. 80–113, 2007.

[139] Q. Pu *et al.*, "Low latency geo-distributed data analytics," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 421–434.

[140] J. M. Pujol *et al.*, "The little engine(s) that could: Scaling online social networks," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 375–386.

[141] J. Ramnarayan *et al.*, "SnappyData: A hybrid transactional analytical store built on spark," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2153–2156.

[142] M. M. Seif, E. M. R. Hamed, and A. El Fatah Abdel Ghfar Hegazy, "Stock market real time recommender model using apache spark framework," in *Proc. Int. Conf. Adv. Mach. Learn. Technol. Appl.*, 2018, pp. 671–683.

[143] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "KeystoneML: Optimizing pipelines for large-scale advanced analytics," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 535–546.

[144] M. Stonebraker *et al.*, "C-store: A column-oriented dbms," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 553–564.

[145] A. Talwalkar *et al.*, "MLbase: A distributed machine learning wrapper," in *Proc. NIPS Big Learn. Workshop*, 2012, pp. 35–42.

[146] S. Tang *et al.*, "EasyPDP: An efficient parallel dynamic programming runtime system for computational biology," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 5, pp. 862–872, May 2012.

[147] A. Thusoo *et al.*, "Hive - A petabyte scale data warehouse using hadoop," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 996–1005.

[148] R. U and B. S. Babu, "Real-time credit card fraud detection using streaming analytics," in *Proc. 2nd Int. Conf. Appl. Theor. Comput. Commun. Technol.*, 2016, pp. 439–444.

[149] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu, Symp. Cloud Comput.*, 2013, pp. 5:1–5:16.

[150] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 301–316.

[151] S. Venkataraman *et al.*, "SparkR: Scaling R programs with spark," in *Proc. Int. Conf. Manag. Data*, 2016, pp. 1099–1104.
[152] V. Vineetha, C. L. Biji, and A. S. Nair, "SPARK-MSNA: Efficient algorithm on apache spark for aligning multiple similar DNA/RNA sequences with supervised learning," *Sci. Rep.*, vol. 9, no. 1, 2019, Art. no. 6631.
[153] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[154] R. T. Whitman, B. G. Marsh, M. B. Park, and E. G. Hoel, "Distributed spatial and spatio-temporal join on apache spark," *ACM Trans. Spatial Algorithms Syst.*, vol. 5, no. 1, pp. 6:1–6:28, Jun. 2019.
[155] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, vol. 30, no. 18, pp. 2652–2653, 2014.
[156] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 13–24.
[157] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-Spark: Transient computing for big data analytics," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 484–496.
[158] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: The geospark perspective and beyond," *GeoInformatica*, vol. 23, no. 1, pp. 37–78, Jan. 2019.
[159] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.
[160] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[161] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.
[162] P. Zečević *et al.*, "AXS: A framework for fast astronomical data processing based on apache spark," *The Astronomical J.*, vol. 158, no. 1, Jul. 2019, Art. no. 37.
[163] H. Zhang, B. M. Tudor, G. Chen, and B. C. Ooi, "Efficient in-memory data management: An analysis," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 833–836, Jun. 2014.
[164] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: Optimized shuffle service for large-scale data analytics," in *Proc. 13th EuroSys Conf.*, 2018, pp. 43:1–43:15.
[165] Y. Zhang and M. I. Jordan, "Splash: User-friendly programming interface for parallelizing stochastic algorithms," 2015, *arXiv:1506.07552*.
[166] Z. Zhang *et al.*, "Scientific computing meets big data technology: An astronomy use case," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 918–927.

Shanjiang Tang received the BS and MS degrees from Tianjin University (TJU), China, in July 2008 and January 2011, respectively, and the PhD degree from the School of Computer Engineering, Nanyang Technological University, Singapore, in 2015. He is currently an associate professor with the College of Intelligence and Computing, Tianjin University, China. His research interests include parallel computing, cloud computing, big data analysis, and machine learning.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University, China, in 1999 to 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2003 to 2008. He is an associate professor with the School of Computing, National University of Singapore, Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems.

Ce Yu received the BS and MS degrees from Tianjin University, China, in 2002 and 2005, respectively, and the PhD degree of computer science from Tianjin University(TJU), China, in 2009. He is currently an associate professor and the director of High Performance Computing Lab (HPCL) of Computer Science and Technology in Tianjin University, China. His main research interests include parallel computing, astronomy computing, cluster technology, cell BE, multicore, grid computing.

Yusen Li received the PhD degree from Nanyang Technological University, Singapore, in 2014. He is currently an associate professor with the Department of Computer Science and Security, Nankai University, China. His research interests include scheduling, load balancing, and other resource management issues in distributed systems and cloud computing

Kun Li received the BS and master's degrees from Tianjin University, China, in 2016 and 2019, respectively. He is currently working toward the PhD degree at Tianjin University, China. His main research interests include parallel computing and astronomy computing.