applied sciences

MDPI

Review

A Comparative Analysis of Big Data Frameworks: An Adoption Perspective

Madiha Khalid * and Muhammad Murtaza Yousaf

Department of Software Engineering, Faculty of Computing and Information Technology, University of the Punjab, Lahore 54000, Pakistan; murtaza@pucit.edu.pk
* Correspondence: madiha.khalid@pucit.edu.pk; Tel.: +92-42-111-923-923 (ext. 554)

Abstract: The emergence of social media, the worldwide web, electronic transactions, and next-generation sequencing not only opens new horizons of opportunities but also leads to the accumulation of a massive amount of data. The rapid growth of digital data generated from diverse sources makes it inapt to use traditional storage, processing, and analysis methods. These limitations have led to the development of new technologies to process and store very large datasets. As a result, several execution frameworks emerged for big data processing. Hadoop MapReduce, the pioneering framework, set the ground for forthcoming frameworks that improve the processing and development of large-scale data in many ways. This research focuses on comparing the most prominent and widely used frameworks in the open-source landscape. We identify key requirements of a big framework and review each of these frameworks in the perspective of those requirements. To enhance the clarity of comparison and analysis, we group the logically related features, forming a feature vector. We design seven feature vectors and present a comparative analysis of frameworks with respect to those feature vectors. We identify use cases and highlight the strengths and weaknesses of each framework. Moreover, we present a detailed discussion that can serve as a decision-making guide to select the appropriate framework for an application.

Keywords: big data frameworks; fault tolerance; stream processing systems; distributed frameworks; Spark; Hadoop; Storm; Samza; Flink; comparative analysis; a survey; data science

Citation: Khalid, M.; Yousaf, M.M. A Comparative Analysis of Big Data Frameworks: An Adoption Perspective. *Appl. Sci.* **2021**, *11*, 11033. https://doi.org/10.3390/app112211033

Academic Editor: Kamran Shaukat

Received: 25 October 2021
Accepted: 12 November 2021
Published: 22 November 2021

1. Introduction

Over the years, data has been generated from millions of data sources at an astonishing rate. Perhaps, the most substantial byproduct of the digital revolution is the generation of an explosive amount of data. This incredible data growth is predicted in a report generated by The Internet Data Center (IDC). In 2012, the IDC estimated that the digital data would be grown by 300 times between 2005 and 2020, which means it will increase from 130 exabytes to 20,000 exabytes [1]. More recently, IDC has predicted in its white paper that by 2025, the digital data will grow by 175 zettabytes [2]. This increasingly growing amount of data is often referred to as *'big data'* [3]. Data is a valuable asset in our information society. Extracting meaningful information from these high volume datasets has become a fundamental activity for industrial and academic communities. Business organizations use big data analytics to analyze customer behaviors and trends that can be capitalized. The high availability of data is also greatly changing the trends in scientific communities. Science has moved towards a data-oriented quest, where the data-intensive computations yield discoveries in science.

Analyzing large scale data may also result in unexpected outcomes that can help to improve the quality of life in many ways. For example, by analyzing the number of flu-related queries, Google Flu Trend can detect regional flu outbreaks faster than the Center for Disease Control and Prevention [4]. Similarly, Google uses large amounts of data to solve complex problems. The data generated by Global Positing System (GPS) is

used to avoid traffic jams, define routes, and determine optimal paths between locations. IBM uses real-time traffic data to accurately predict the arrival times of buses in London [5]. All these data-oriented advancements are achieved mainly due to the emergence of new technologies that provide powerful functions and controls to programmers, so that they can focus on data processing and information extraction instead of managing resources and low-level parallelism details.

Since datasets have grown to an order of magnitude which is difficult to perceive, to put this into perspective, Figure 1 shows some real-world examples to illustrate large data scales. Such voluminous and incremental datasets make it impossible to store and process them in a reasonable amount of time using traditional computing techniques. To Address this challenge, parallel computing environments and technologies have emerged as a prime solution. These parallel computing environments offer increasingly powerful ways to analyze and process large scale data for real-time analysis. However, the complexity of such parallel computing environments and their characteristics hinders the maximum productive utilization of these platforms. Some of the concerns include resolving dependencies, load balancing, scheduling and scalability. The problem poses an additional challenge when we add an almost certain possibility of machine failure and fluctuation of workloads that may be caused by temporary suspension or activation of computing nodes. These challenges resulted in the development and evolution of several big data processing frameworks.

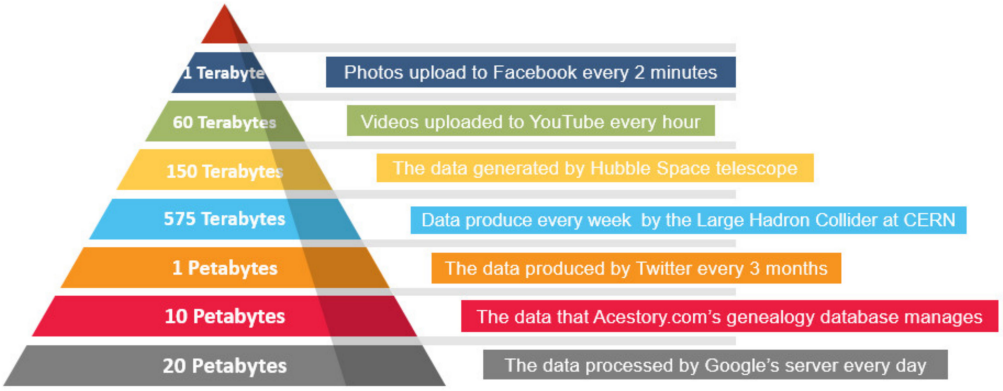

Figure 1. Understanding the data deluge.

Hadoop MapReduce is one of the earliest frameworks that empowers the use of inexpensive commodity machines for big data analytics in place of expensive high-end systems. The early adoption of Hadoop MapReduce by research communities caused the rapid evolution of the system, which laid the foundation for the development of several other frameworks. Among those frameworks, some are based on Hadoop distributions, while others are self-developed.

Big data analytics is a rewarding tech trend in business communities and enterprise software development. Consequently, big data processing is fundamental to the business models of many companies. The availability of a number of big data frameworks makes it difficult for practitioners and researchers to decide which framework will better serve the needs of a particular application. Therefore, selecting an appropriate big data processing framework for a particular application is a non-trivial task. In order to choose the right framework for a particular application, one must consider the architectural features, the data processing model, semantics of fault tolerance and performance guarantees of the framework. These considerations will help determine whether or not a framework will best meet the needs of an application. Therefore, it is important to have a study that discusses and compares the most prominent and widely used big data frameworks.

This article provides an in-depth review of the five most popular and widely adopted big data frameworks in the open-source landscape, i.e., Hadoop, Spark, Storm, Samza and Flink. We identify some key features and group the logically related features together

to form a feature vector. The frameworks discussed in this study are then compared based on seven feature vectors. We also identify the key strengths and limitations of these frameworks. We discover the use cases where each of these frameworks ideally fits in. The main contributions of this work are as follows:

- A comprehensive overview of the most widely used open-source frameworks with an architectural perspective. We discussed the different fault-tolerance mechanisms, the scheduling schemes, and the data flow models used by each of these frameworks.
- Comparative analysis of the frameworks with respect to identified feature vectors.
- Identification of key strengths and weaknesses of each of the frameworks under consideration. We presented a detailed discussion that serves as a guide for selecting the appropriate framework for an application.
- We pointed out the application use cases where each of the frameworks ideally fits in.

For convenient referencing, Table 1 provides a list of acronyms used in this article. The rest of the paper is organized as follows: Section 2 surveys existing literature studies and highlights the differences of our work from existing studies. Section 3 discusses the requirements of a distributed framework. Section 4 presents the architectural details along with the identified requirements of the presented frameworks. Section 5 discusses feature vectors and comparative analysis of the frameworks with respect to those feature vectors. Section 6 presents a comprehensive discussion and point out the findings of the comparison. Finally, in Section 7, we present our conclusion. Figure 2 outlines the overall organization of the paper.

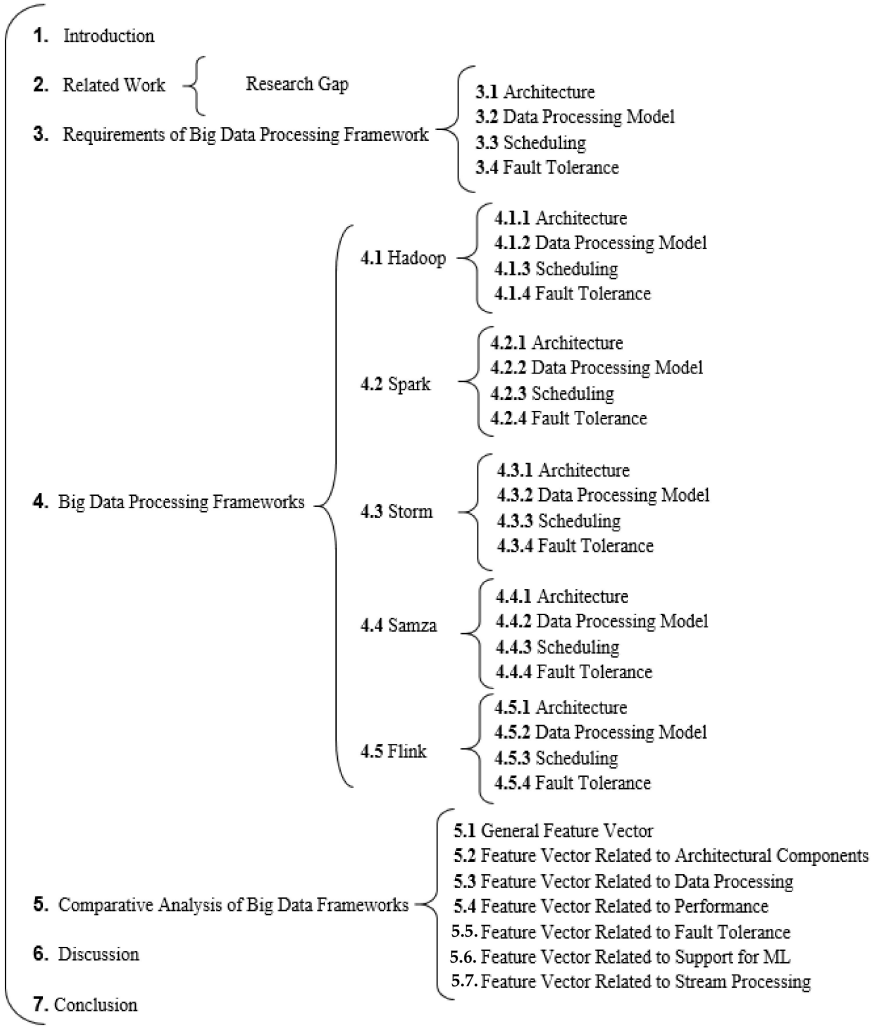

Figure 2. Organization of the paper.

Table 1. List of acronyms used in this article.

ACLs	Access Control Lists
AMP	Advanced Materials Processing
API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
FIFO	First In First Out
GPS	Global Positioning System
HDFS	Hadoop Distributed File System
I/O	Input/Output
IDC	Internet Data Center
ML	Machine Learning
NM	Node Manager
RDDs	Resilient Distributed Datasets
RM	Resource Manager
SSL	Secure Sockets Layer
TLS	Transport Layer Security
YARN	Yet Another Resource Negotiator

2. Related Work

There are various studies published in the literature that compares different big data frameworks. In this section, we will discuss some of the significant and recent research contributions on the topic. Figure 3 depicts a timeline view of the research works discussed.

Chen and Zhang [6] focused on the problems and challenges of big data and the techniques to address these problems. The authors discussed a number of methodologies to handle large scale data such as cloud computing, granular computing, quantum computing and bio-inspired computing. Singh and Reddy [7] presented a comprehensive analysis of big data platforms, including HPC clusters, Hadoop ecosystem, peer-to-peer networks, GPUs, multicore CPUs and field programmable gate arrays (FPGAs). Morais [8] provided a theoretical survey of Hadoop, Spark and Storm. The author compared two frameworks Spark and Storm, based on five parameters, processing model, latency, fault tolerance, batch framework integration and supported languages. Hesse and Lorenz [9] carried out a conceptual survey of stream processing systems. The discussion is generally focused on some fundamental differences related to stream processing systems. Landset et al. [10] discussed open-source tools for machine learning with big data in the Hadoop ecosystem. The authors discussed machine learning tools and their compatibility with MapReduce, Spark, Flink, Storm and H_2O. They evaluated a number of machine learning frameworks based on scalability, ease of use, and extensibility. Authors in [11] compared big data frameworks on a number of performance parameters based on some empirical evidence from the available literature. Bajaber et al. [12] presented a taxonomy and investigation of open challenges in big data systems. The authors discussed big data systems, their SQL processing support and graph processing support. The research focuses on identifying research problems and opportunities for innovations in future research.

A survey of the state-of-the-art stream processing systems is presented in [13]. Authors discussed mechanisms for resource elasticity to adapt to the needs of streaming services in cloud computing. The research also explores the problems and existing solutions associated with effective resource management. The research indicators are limited to the elasticity aspect of stream processing systems. Inoubli [14] compared big data frameworks on the basis of experimental evaluation. Another study [15] empirically compared the performance of popular big data frameworks on a number of applications that include WordCount, Kmeans, PageRank, Grep, TeraSort, and connected components. Authors demonstrated that Spark outperformed Flink and Hadoop for WordCount and k-means, while Flink performed well for PageRank. Both Spark and Flink yield similar results on rest of the applications. In [16], researchers measured the performance of Spark and

Hadoop using WordCount and a logistic regression program. The results showed that Spark outperformed Hadoop.

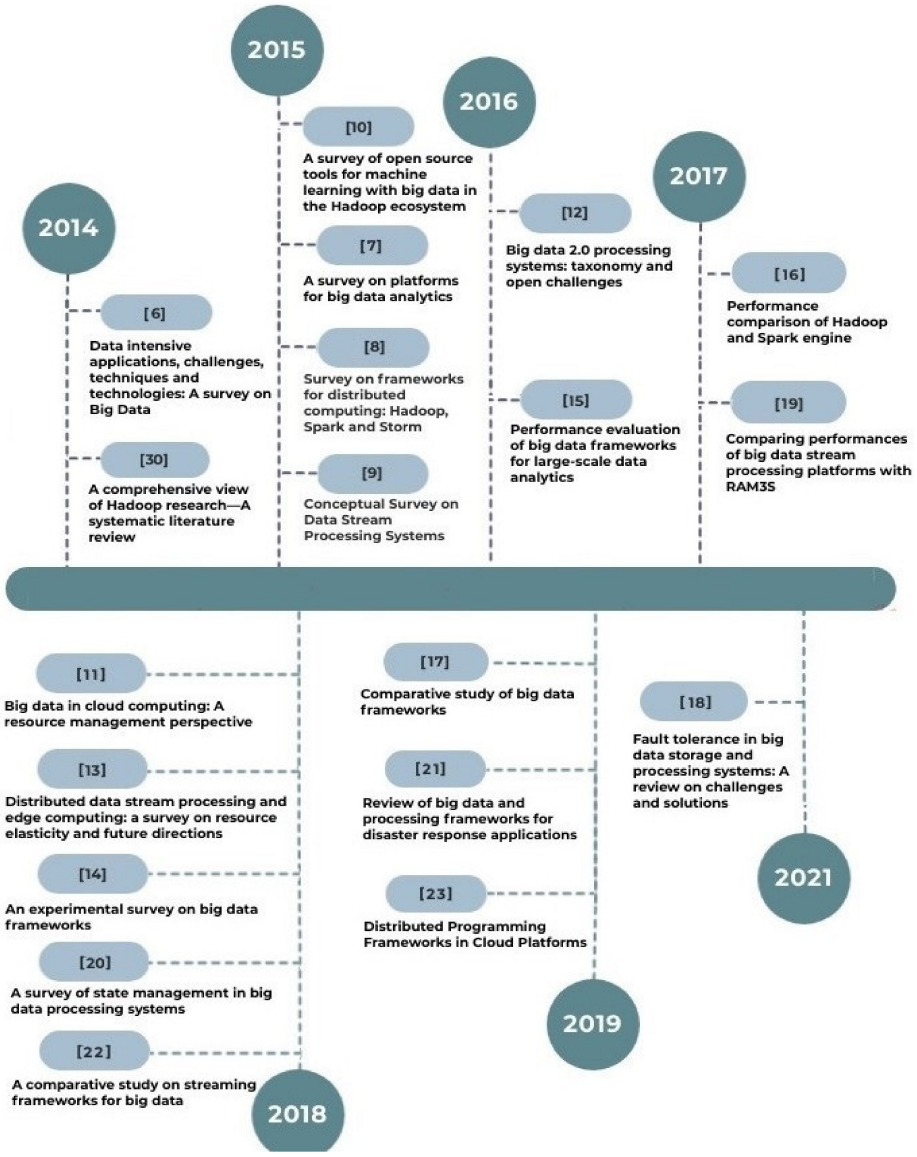

Figure 3. A timeline view of the research works reviewed.

Gupta and Parveen [17] surveyed popular big data frameworks on the basis of a few primitive parameters. Saadoon et al. [18] highlighted common problems and potential solutions related to fault tolerance in big data systems. The study presented a thorough discussion based on the findings derived from existing studies. Bartolini and Patella [19] conducted a study that used sustainable input rate as a measure of efficiency to compare big data frameworks. Authors empirically proved that Storm outperformed both Flink and Spark in local cluster setting as well as in cloud environment. In [20] authors focused state management techniques in big data systems such as Flink, Heron, Samza, Spark, and Storm. Cumbane [21] reviewed big data frameworks for disaster response applications. This research also discussed the similarities and differences of big data frameworks. However, the focus of the study revolves around the application of big data systems in the response phase of a disaster. Inoubli et al. [22] surveyed popular stream processing frameworks and an experimental evaluation of resource consumption. Patil [23] discussed big data frameworks and empirically compared Spark and Flink with TeraSort benchmark using execution time, network usage and throughput as performance measures.

Research Gap

There are some research gaps in the aforementioned literature works. The architectural aspects of big data frameworks are not thoroughly explored, which could have significantly assisted in suitability assessment. Several researchers [14–17,19,22,23] focused on the performance comparison of the big data frameworks. A number of studies are dedicated to reviewing big data frameworks concerning a specific aspect or in the context of a specific application domain [10,12,20–22]. The studies that made architectural comparisons considered only a limited number of features. Although the architectural and related aspects of some big data frameworks have been explored recently, no guidance is provided to help developers and practitioners select a suitable framework for their application. Furthermore, the existing studies do not elaborate on the comparative analysis that can help to build meaningful preference about the frameworks for a particular application. There is a need to cover all architectural aspects related to the core functionality of a big data framework. A detailed comparative analysis based on the comprehensive feature set will help choose a suitable big data framework for an application. This clearly gives a motivation to explore the most widely used big data frameworks and the important architectural features in qualitative assessment.

3. Requirements of a Big Data Processing Framework

This section specifies the four important requirements of a big data processing framework that we chose to focus on.

3.1. Architecture

Almost all big data processing frameworks distribute workloads across multiple processors, which requires partitioning and distribution of data files, managing data storage in a distributed file system, and monitoring actual processing performed by multiple processing nodes in parallel. The architecture of a framework is designed to handle all these responsibilities. It typically serves as a reference blueprint for available infrastructure that defines various logical roles and describes how the system will work, the components used, and how the data will flow. Most of the big data framework architectures include four major modules: resource management, scheduling, execution, and storage [24]. The architecture should outline the entire data life cycle starting from the data source till the generation and storage of results for future use [25]. This architecture must ensure fault tolerance, scalability, and high availability.

3.2. Data Processing Model

The data processing model defines how the data is processed and how the computations are represented in the system. Generally, the data is processed either in batch mode or in the form of a continuous stream. In a batch processing system, the processing is done on a block of data that is stored over a period of time. Whereas, in a stream processing system, the processing is done on a continuous stream of data as it arrives. Computations are also represented in different forms, for example, Hadoop processes computations in map-reduce functions. Alternatively, data and computations can also be realized as a logical directed acyclic graph (DAG).

3.3. Scheduling

In a big data processing system, a lot of CPU cycles, network bandwidth, and disk I/O are required. Therefore, it is important to schedule tasks efficiently such that it minimizes the overall execution time and maximizes resource utilization. The primary goal of task scheduling is to schedule independent and dependent tasks and the optimal reduction of the number of task migrations, thereby reducing computation time while increasing resource utilization.

3.4. Fault Tolerance

In distributed computing, faults can occur at various levels, such as node failures, process failures, network failures, and resource constraints. Although the failure probability of a single component is relatively low, when a large number of such components work together, the probability of component failure at any given time cannot be ignored. In fact, in large scale computing, failure is not an exception rather a norm. Fault tolerance is an attribute that enables the system to continue working if one or more components fail.

4. Big Data Processing Frameworks

Parallel and distributed computing has emerged as a prime solution for the processing of very large datasets. Yet, their complexity and some of their features may prevent its maximum productive utilization to common users. Data partitioning and distribution, scalability, load balancing, fault tolerance, and high availability are among the major concerns. Various frameworks have been released to abstract these functions and provide users with high-level solutions. These frameworks are typically classified according to their data processing approach, i.e., batch processing and stream processing (as shown in Figure 4). MapReduce has emerged as one of the earliest programming models for batch processing. Today, it is recognized as a pioneer system in big data analytics. In 2004, under the influence of an ever-increasing amount of data on the web, Google developed Google File System [26] and MapReduce [27]. In a MapReduce program, the user specifies the computation in the form of two functions map and reduce, which can be executed in parallel on multiple computing nodes and easily scaled to large clusters. The Map function takes in a key/value pair and generates a collection of intermediate key/value pairs. The Reduce function combines all the intermediate values assigned to the same intermediate key. The runtime system is responsible for data partitioning, distribution of data and computation across the cluster, handling machine failures, and managing necessary communications among computing nodes.

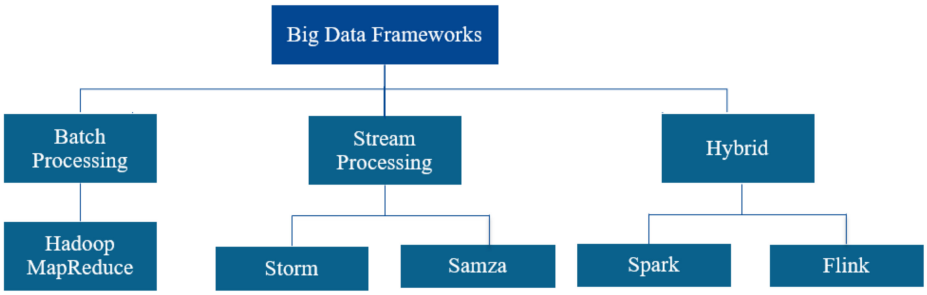

Figure 4. Classification of big data processing frameworks.

Efforts have been made to convert these technologies into open-source software, which resulted in the development of Apache Hadoop [28] and Hadoop file system [29], that laid the foundation for other big companies like Yahoo!, IBM, Twitter, LinkedIn, Oracle and HP to invest in building solutions for large scale data processing. MapReduce and its related distributions had established them as a sound solution to batch workloads. However, a well-defined processing model for distributed stream processing was still lacking. Consequently, a number of reliable and popular open-source frameworks were developed, such as Sparks, Storm, Samza and Flink. This section sheds light on the five most prominent and widely adopted big data processing frameworks.

4.1. Hadoop

Apache Hadoop [28] is the most widely used implementation of the map reduce programming model. Being an open-source implementation, Hadoop is equally popular in the researchers and industrial community. However, the major reason for its increasing adoption is its inherent characteristics such as load balancing, fault tolerance, and scalability.

The framework was first developed by a yahoo employee Doug Cutting and a professor of the University of Michigan, Mike Cafarella [30]. Later, it evolved over several years to reach its current stable version. Since its initial release, Hadoop gained the increasing attention of researchers and was adopted by several industry giants such as Yahoo!, Amazon, Facebook, eBay, and Adobe, which caused the rapid evolution of the framework [30].

4.1.1. Architecture

The overall architecture of Hadoop consists of two major components Hadoop Distributed File System (HDFS) [29] and Yet Another Resource Negotiator (YARN) [31]. HDFS is Hadoop's very own distributed file system which provides high throughput access to application data across thousands of machines in a fault tolerant manner. While YARN is responsible for resource management and scheduling. Figure 5 illustrates the architecture of Apache Hadoop.

In HDFS, data is stored in the form of files that are divided into data blocks. Each block is stored in one of the nodes in the Hadoop cluster. Thus, each node in the cluster stores a part of a file. Data blocks are replicated to ensure high availability and fault tolerance. HDFS is primarily based on two daemons called NameNode and DataNode. The HDFS cluster consists of one master node and several slave nodes. The master runs the NameNode daemon that is responsible for managing the file system and regulating file access to users. While slave runs DataNode daemon that stores data blocks and serves read/write requests from a user.

YARN was introduced by Yahoo! and Hortonworks in 2012 [31], and it became part of Hadoop as Hadoop 2.x [30]. In Hadoop 2.x, YARN takes the role of distributed application manager and MapReduce stays as pure computational framework. It has two major components named Resource Manager (RM) and Node Manager (NM). Resource Manager is a master daemon that is responsible for scheduling and resource management, while, Node Manager is a slave daemon that is responsible for managing containers and monitoring resource usage on a single node. It periodically monitors the health status of the host node and reports the same to RM.

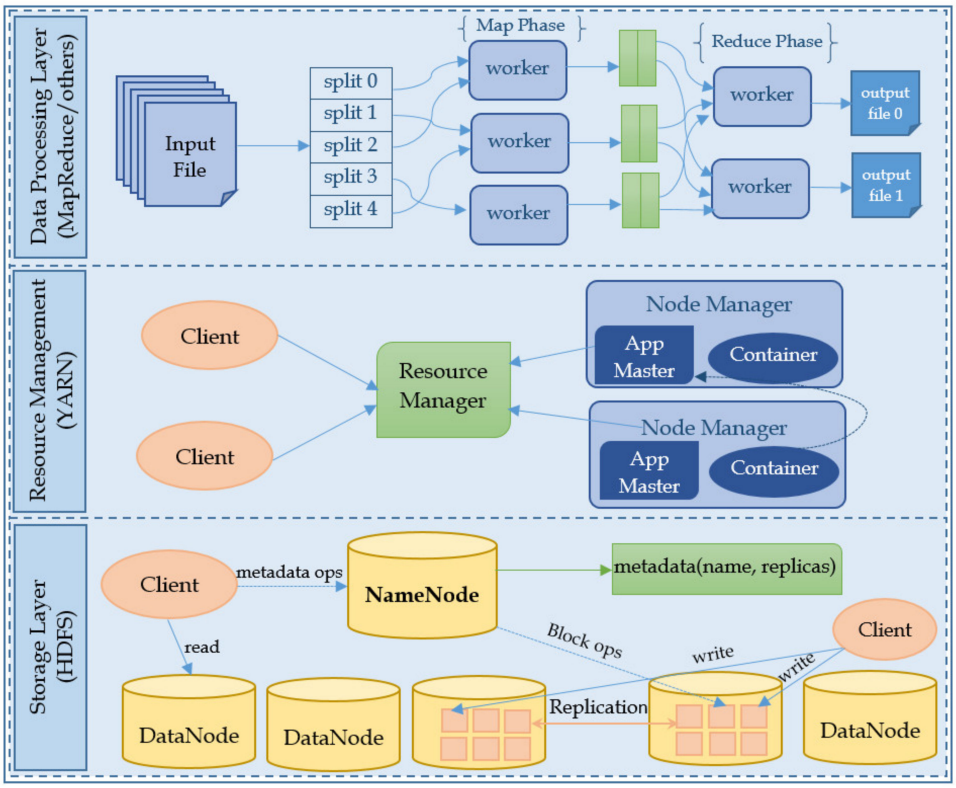

Figure 5. Apache Hadoop architecture.

4.1.2. Data Processing Model

Apache Hadoop is well suited for batch processing of unstructured data. Prior to Hadoop 2.x, there were two layers in the Hadoop, MapReduce layer for data processing and cluster management while HDFS layer for data storage. Therefore, MapReduce was the only data processing framework for Hadoop 1.x. Later, with the emergence of Hadoop 2.x, data processing and resource management are separated into two layers that enable Hadoop to work with other data processing platforms such as Crunch (Apache Crunch: http://crunch.apache.org) (accessed on 12 June 2021), Tez [32], Pig [33] and Cascading (Cascading: https://www.cascading.org) (accessed on 12 June 2021). However, majorly the data processing is done using the MapReduce paradigm.

4.1.3. Scheduling

Scheduler in Hadoop is part of YARN resource manager and is purely responsible for scheduling resources to run applications. YARN scheduler offers three pluggable scheduling policies, FIFO, Capacity and Fair. FIFO is the default scheduling policy that serves resource requests on a first come, first serve basis. This scheduling policy is not suitable for shared clusters as larger applications will occupy all the resources resulting longer waiting times for other applications in the queue. The Capacity Scheduler, originally developed by Yahoo!, allows large clusters to be shared with multiple tenants while providing each tenant with a minimum capacity guarantee. On the contrary, the Fair scheduler, developed by Facebook, does not reserve resources as per capacity rather, it dynamically balances resources among all outstanding jobs. The idea is to share the available resources among submitted jobs such that each job will get, on average, an equal share of resources.

4.1.4. Fault Tolerance

Apache Hadoop is highly fault tolerant. HDFS layer ensures fault tolerance using data replication. Data blocks stored in DataNodes are replicated and distributed across the cluster to provide reliability and high availability. Prior to the 2.x, the NameNode was a single point of failure in an HDFS cluster, with the release of Hadoop 2, HDFS high availability enables multiple standby NameNodes to run on a single HDFS cluster in an active/passive configuration. A failure can arise at three levels: task level, slave node level, master node level. If a task fails at the slave node, it reports back to the resource manager before exiting which in return allocates the failed task to another machine in the cluster and marks up the free slot on slave for another task. If a task fails at the master node, that task can be restarted in the same node from the last check point state since the master takes periodic check points of all the master data structures. When a slave node fails, the RM at the master node will detect this failure by timing out its heartbeat response. The RM will assign failed node tasks to some other idle node in the cluster and remove the failed node from a pool of resources. If the fault is transient, the NM at slave cleans up its local state, resynchronize itself with RM and redo its work done during the fault. RM failure was a single point of failure before Hadoop 2.x. However, the high availability feature enables running a redundant resource manager in standby mode that takes up in case of active resource manager failure.

4.2. Spark

Apache Spark is a cluster computing framework originally developed by Matei Zaharia at UC Barkley in 2009 and later donated to Apache Foundation in 2013 [34]. Previously, Hadoop was deficient for iterative working sets that are reused across multiple parallel operations. Hence, the primary design goal of Spark is to extend the MapReduce model to support interactive queries and iterative jobs efficiently. Spark introduced Resilient Distributed Datasets [35] for in-memory processing to speed up computations. Since its inception, several communities have contributed to building a sound echo system around it, especially a rich set of APIs initially available in Scala and later available in Java, Python,

R and SQL. Spark is one of the first frameworks to support distributed batch processing and stream processing along with iterative queries.

4.2.1. Architecture

Apache Spark has a layered architecture in which all Spark components and layers are loosely interconnected, as shown in Figure 6. Spark has a rich set of high-level libraries, which includes SparkSQL [36] for the processing of structured data, Spark streaming [37] for real-time stream processing, MLlib [38] for machine learning algorithms, GraphX [39] for graph processing and SparkR [40] for big data analysis from R shell. Spark core is responsible for task scheduling, fault recovery, interacting with storage systems and memory management. The spark core engine is based on the master slave architecture. The master node has the driver program that executes the main function of the user application. The driver program creates the spark context. The spark context connects with the cluster manager and acquires the executors on worker nodes and distributes tasks to the executors. The worker nodes are salve nodes that actually executes the tasks and return the result to spark context. Every worker node launches a process known as executor to run tasks and provide in-memory storage for Resilient distributed datasets (RDDs). RDDs are in-memory partitioned sets of data that are distributed over multiple nodes across the cluster. Spark can be deployed as a Standalone server or run on top of a cluster manager like Mesos [41] or YARN. Spark does not have its storage mechanism rather. It can work with any Hadoop compatible data sources such as HBase, HDFS, Casandra and Hive etc. [34].

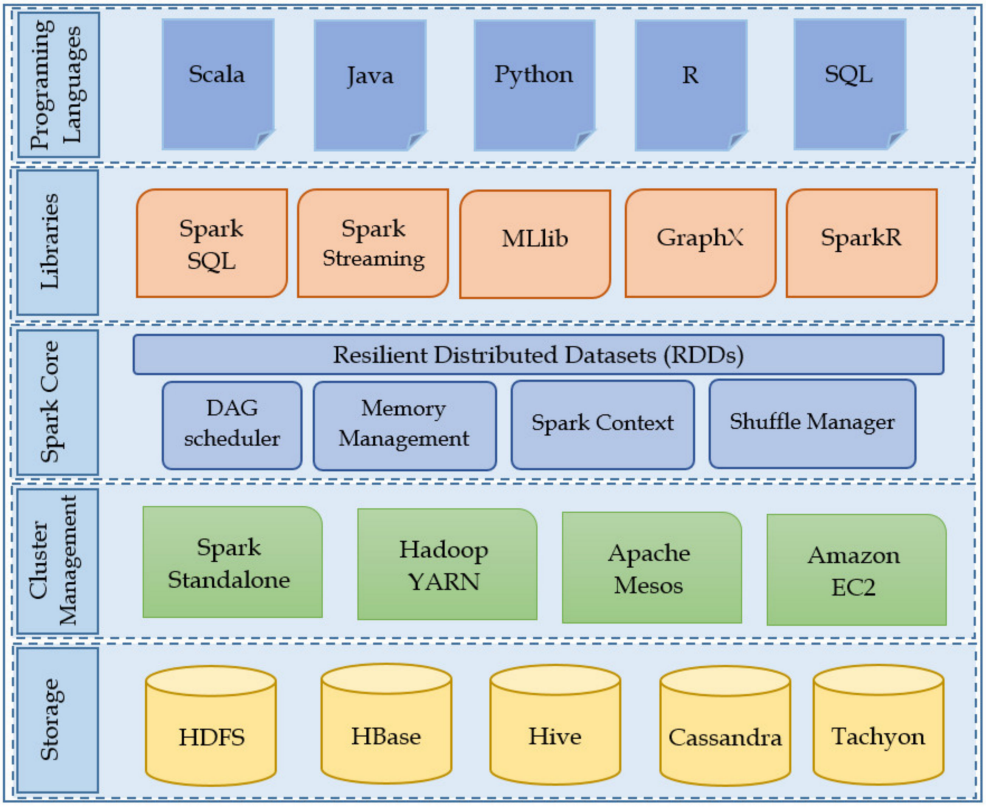

Figure 6. Layered architecture of Spark.

4.2.2. Data Processing Model

Unlike Hadoop, Spark is suitable for stream processing as well as batch processing. Spark works with RDDs and DAG to run operations. All Spark jobs are eventually converted into a Directed Acyclic Graph prior to execution. In contrast to MapReduce, where there are only two stages of computations map stage and reduce stage, Spark has multiple stages of computations that forms a DAG.

4.2.3. Scheduling

The internal scheduling of tasks within a Spark application is done either using a FIFO scheduler or a Fair scheduler. FIFO is the default scheduler that may cause significant delay to process later tasks if the earlier tasks are long running. Fair scheduler mitigates this problem by assigning tasks round robin fashion. FAIR Scheduler also supports grouping tasks into pools. In addition, different scheduling parameters (such as weights) can be configured for each pool. This is useful for creating higher priority pools for certain tasks. For scheduling across multiple applications over the cluster, Spark relies on the cluster managers that it runs on. For a multi-user environment in the cluster, Spark offers both static and dynamic scheduling. With static partitioning of resources, on startup each application is given its maximum required resources for its lifetime. Spark's standalone cluster mode, coarse-grained Mesos mode and YARN use static partitioning. In standalone mode, by default, applications run in FIFO order, and each application will try to use all available nodes. Spark also have a mechanism to dynamically allocate resources based on the demand of applications. This enables applications giving up on resources even during the execution if they are not in use and applications can request them back if they are needed.

4.2.4. Fault Tolerance

Since, spark processes data in a fault-tolerant file system (such as HDFS), so any RDD built from fault-tolerant data will be fault-tolerant. Also, it inherits the fault-tolerance of highly resilient cluster managers such as YARN and apache mesos, as it runs on top of them. Another main semantic of fault tolerance in Apache Spark is that all Spark RDDs are immutable and the dependencies between the RDDs are recorded through the lineage graph in the DAG. Hence, each RDD remembers that how it was created from previous RDD. In case of failure, RDDs can be recovered by using lineage information. If a worker node fails, the executors on that worker node will be killed along with the data in its memory. With the help of a lineage graph, these tasks can be re-executed on another worker node. If the master node fails, the cluster manager can restart the application.

4.3. Storm

Apache Storm is a distributed processing framework for real-time data streams. Storm was originally developed by Nathan Martz of BackType, which was acquired by Twitter in 2011. Storm became open-source in 2012 and became part of Apache projects later in 2014. Since its inception, Storm has been widely recognized and adopted by some of the big names in the industry, such as Twitter, Yahoo!, Alibaba, Groupon, WeatherChannel, Baidu and Rocket Fuel [42]. Apache Storm is designed to process and analyze large amounts of unbounded data streams that may come from various sources and publish real-time updates on the user interface or other locations without storing any real data. Storm is highly scalable in nature and provides low latency with an easy to use interface through which developers can program virtually in any programming language [43]. To achieve this language independence, Apache Storm uses Thrift definition to define and deploy topologies.

4.3.1. Architecture

The Apache Storm is based on the master/slave architecture. The Storm architecture allows only one master node. The physical architecture of Storm consists of three main components. Nimbus, Zookeeper and supervisor (as shown in Figure 7a). Nimbus is a master daemon that distributes work among all available workers. The key responsibilities of Nimbus include assignment of tasks to working nodes, tracking the progress of tasks, and rescheduling of the tasks to other working nodes in case of failure. Actual processing is performed by worker nodes. Each worker node can run one or more worker processes. At any given time, a single machine may be running multiple worker processes. Each worker node has a supervisor process running on it which communicates with Nimbus.

The supervisor coordinates the status of the currently running topology and announces any available slots to take up possibly more work. Nimbus monitors the topologies that need to be mapped and does the mapping between those topologies and supervisors when needed. Zookeeper [44] is used for all coordination between Nimbus and the Supervisors.

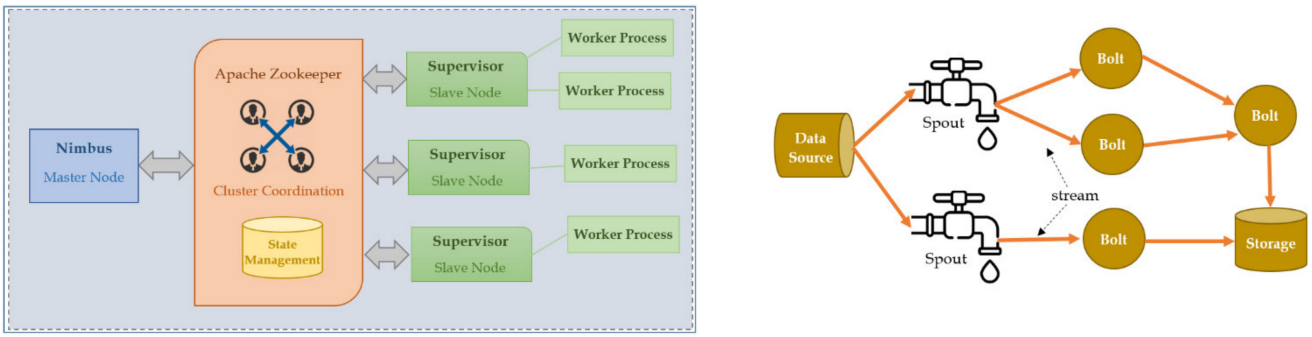

Figure 7. Apache Storm (**a**) Architecture of Apache Storm; (**b**) An illustration of Storm Topology.

4.3.2. Data Processing Model

The basic data processing model of Storm consists of four abstractions, topology, spouts, bolts, stream. In Storm, a stream is an unbounded sequence of tuples, where tuples are named lists of values, that can be of any type including strings, integers, floating-point numbers, etc. The logic of any real time storm application is presented in the form of a topology, which is a network of bolts and spouts. Figure 7b illustrates a storm topology. Spouts are source of stream that essentially connects with a data source such as Kafka [45] or Kestrel. It continuously receives data and converts it to stream of tuples and pass them to bolts. Bolts are the processing units of a storm application that can perform a variety of tasks.

4.3.3. Scheduling

Apache Storm has four built-in schedulers: Default, Isolation, Multitenant, and Resource Aware [46]. The default Storm scheduler is fair scheduler that takes into account each node when scheduling tasks. It implements a simple round-robin strategy with the goal of producing an even distribution of work among workers. Isolation scheduling enables safe sharing of a cluster among many topologies. In isolation scheduling, the user can specify which topologies should be isolated, which means topologies marked isolated are running on a dedicated set of nodes in the cluster, and other topologies will not be able to run on those nodes. These isolated topologies have priority in the cluster, when there is competition with the non-isolated topology, resources are allocated to the isolated topology. When the isolated topology requires resources, the resources are taken away from the non-isolated topologies. The multitenant scheduler is designed for a multitenant storm cluster. It allocates resources on per user basis. Whereas, the resource aware scheduler works in two phases. In the first phase, task selection is performed by obtaining a list of unassigned tasks. In the later phase, it selects a node for each task based on resource-aware considerations such as CPU, memory, physical distance and network bandwidth. Other than built-in schedulers, there are a number of scheduling techniques proposed in the literature [46–51] that focuses on improving latency, network traffic, and throughput.

4.3.4. Fault Tolerance

The Nimbus and Supervisors are designed to fail-fast, yet resilient and stateless daemons, with all of their data stored in Zookeeper or on the local discs, thus, preventing any catastrophic loss in case of processes or processor failure. This design artifact is the key to Storm's fault-tolerance. Even if the Nimbus process fails, the workers can still make progress. However, without Nimbus, workers will not be reassigned to other machines

when necessary. Furthermore, the Supervisors can restart the worker processes if they fail. If a supervisor fails and unable to send a heartbeat to Nimbus, or the task assigned to a supervisor is timed out, then the Nimbus will reschedule that task to another slave node. Additionally, Storm offers reliable spouts to replay the stream in case it is failed to be processed.

4.4. Samza

Samza [52], developed in-house at LinkedIn in 2013 and later donated to Apache Software Foundation. It is based on a unified design for the stateful processing of batched data and high volume real time data streams. Samza is designed to support high throughput (millions of messages per sec) for data streams while providing quick fault recovery and high reliability. To achieve these design goals, samza uses some key abstractions, such as partitioned streams, changelog capturing and local state management. Samza is now adopted by several big companies including LinkedIn, VMWare, Uber, Netflix, and TripAdvisor [53].

4.4.1. Architecture

Samza has a layered architecture that consists of three layers. Figure 8 depicts three layers of Samza architecture. A streaming layer which is responsible for providing replayable data source such as Apache Kafka, AWS Kinesis, or Azure EventHub. The execution layer which is responsible for scheduling and resource management, and processing layer which is responsible for data processing and flow management. The streaming layer and execution layer are pluggable components. Data streaming can be provided by any existing data sources. Similarly, cluster management and scheduling can be done by Apache YARN or Mesos. However, Samza has built-in support for Apache Kafka data streaming and Apache YARN for job execution. The execution model of a Samza Job is based on publish/subscribe task concept. It listens to a data stream from Kafka topic, processes the message when it arrives, and then sends its output to another stream. The data stream in Kafka consists of several partitions based on a key value pair. Samza tasks consume data streams and can run multiple tasks in parallel to consume all partitions of the stream in parallel. Samza tasks are executed in the YARN containers. YARN distributes containers on multiple nodes in the cluster and distributes tasks evenly among the containers. After execution the output can be sent to another stream for further processing. Unlike common stream processing systems, Samza is based on a decentralized model where there is no system level master to coordinate activities rather every job has a lightweight coordinator to manage it.

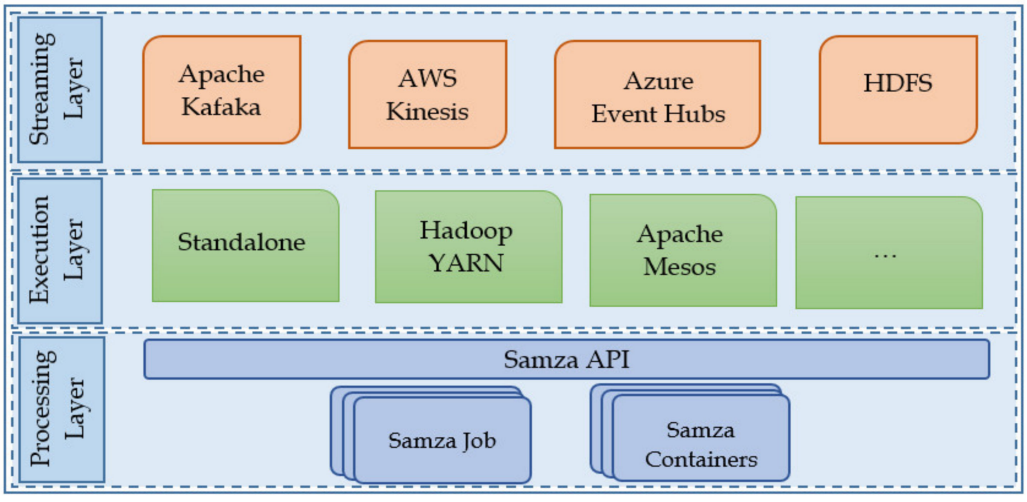

Figure 8. The layered architecture of Samza.

4.4.2. Data Processing Model

The data processing model of Samza builds upon two strong pillar: streams and jobs. Streams are primarily the inputs and outputs of the samza Jobs. A stream in Samza is a multi-part, ordered, multi-subscriber message sequence that is replay-able and lossless by design. User programs in Samza are processed in the form of Samza jobs. A Samza job is the code that consumes and processes a series of input streams and produce one or more output streams. These jobs are represented in the form of a directed graph of operators (vertices) connected by data streams (edges). The job and streams are further broken down into smaller execution units of parallelism, called tasks and partitions. Each task receives data from one or more partitions for each worker input stream. With this break down of stream to partitions and jobs to parallel tasks, samza achieves a linear scalability with number of containers [52].

4.4.3. Scheduling

For task scheduling and resource negotiation, Samza relies on pluggable cluster managers like YARN and Mesos.

4.4.4. Fault Tolerance

To ensure fault tolerance, each task in Samza runs a changelog-capturing service in the background that records incremental changes at a known place in the native file system. A failed task can be restarted by replaying the changelog. When a container restarts after a failure, it looks for the latest checkpoints and begins accepting messages from the latest checkpoint offset, this ensures at least once processing guarantees. This changelog mechanism is more efficient in comparison with full state check pointing, where a snapshot of the entire state is stored [52]. For further efficiency gains, the change log updates do not use the main computation hot path. Updates are stored in batches and sent to Kafka periodically in the background using spare network bandwidth. Re-scheduling a task after failure improves efficiency by Host Affinity mechanism that reduces the overhead of accessing changelog remotely, however, this overhead is unavoidable in case of permanent/long term machine failures.

4.5. Flink

Apache Flink is a distributed processing framework for stateful processing of unbounded and bounded streams of data. Flink is considered as next generation large scale data processing framework that is designed to work in all popular cluster environments with low latency and high throughput. Flink was initiated in 2009 at Technical University of Berlin with the name Stratosphere [54]. In 2014, as an Apache incubator project, Stratosphere became an open-source project with the name "Flink". Flink is known to process data hundred times faster than MapReduce [55]. Due to its highly flexible windowing mechanism, Flink programs can calculate early and approximate results, as well as delayed and accurate results through the same process, so there is no need to combine different systems for the two use cases [56].

4.5.1. Architecture

Flink has a layered architecture that consists of four layers (Figure 9). Primarily, Flink is a stream processing engine that doesn't offer a storage and resource management system of its own, instead it is designed to read data from various streaming and various storage systems. The core of Flink is a distributed data flow engine that receives user programs in the form of a DAG. The DAG in Flink is a parallel data flow graph which contains a series of tasks that produce and consume data streams. Flink has two main APIs: The DataSet API for processing bounded data streams i.e., batch processing and the DataStream API for processing potentially unbounded data streams. DataStream and DataSet APIs are interfaces that programmers can use to define jobs. When compiling the program, these APIs will generate a data flow graph.

The Flink runtime mainly consists of three processes: Job Manager, Task Manager and Client. The client receives the application code, converts it into a data flow graph, and then sends it to the JobManager. This conversion stage will also create serializers and other type-specific code. In addition, the program will go through a cost-based query optimization phase. JobManager is the master process that controls the execution of a single application. It is responsible for all activities that need to be centrally coordinated, such as distributed execution of data streams, monitoring the status and progress of each task, scheduling new tasks and coordinating checkpoints. Task Managers are the worker/slave processes that actually processes the data stream. Each application is managed by a different JobManager that receives the application from client. JobManager then requests the necessary resources from the ResouceManager. When it receives enough TaskManager, tasks will be distributed to the executing TaskManagers.

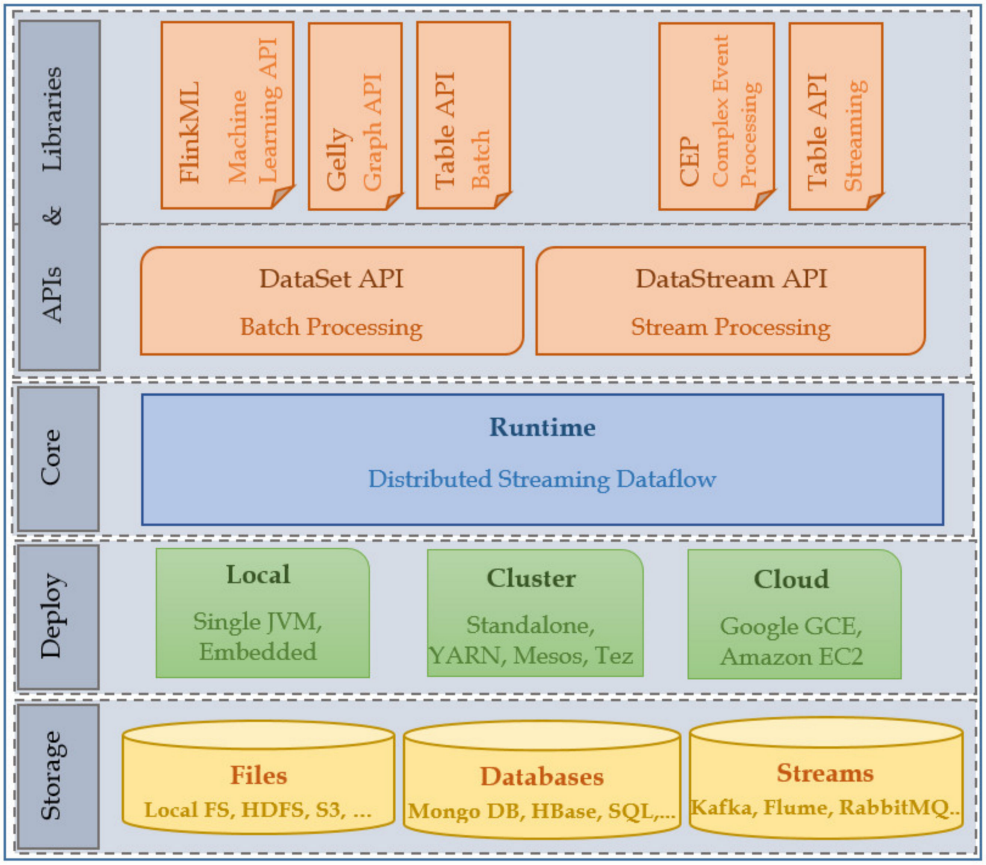

Figure 9. Apache Flink architecture.

4.5.2. Data Processing Model

All the Flink programs are compiled and converted into a common representation i.e., the data flow graph. The data flow graph is then executed by the Flink's execution engine, which is a common layer under the DataSet and DataStream APIs. The data flow graphs are executed in data-parallel manner. In a data flow graph, each edge represents a stream of data, and each vertex represents an operator that uses application defined logic to process data. Usually, there are two types of vertices, called source and sinks as depicted in Figure 10. The source consumes external data and injects it into the application, while the sink is meant to capture the results generated by the operators.

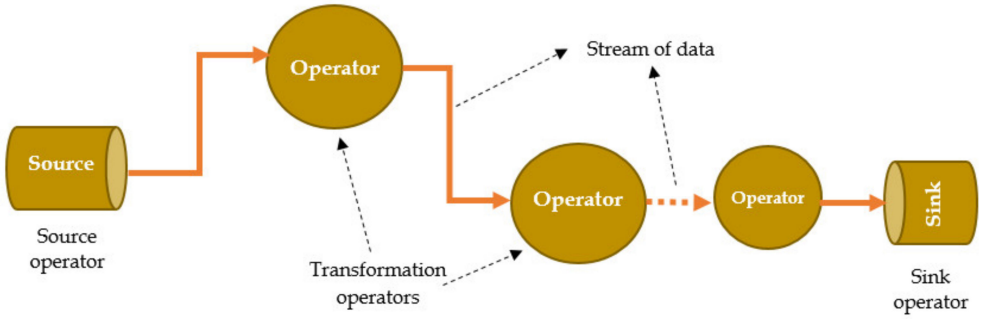

Figure 10. A data flow graph in Flink. Here, edges represent data streams and each vertices represent operators.

4.5.3. Scheduling

There are three scheduling strategies in Flink for resource allocation: all at once/eager scheduling, lazy from sources scheduling and pipelined region scheduling. All at once/eager scheduling tries to allocate required resources at once when the job starts. This strategy is primarily targeted for streaming jobs because in case of batch processing, acquiring all resources in advance will lead to resource underutilization as any resources allocated to subtasks that could not run at any time due to blocking results will be idle and therefore considered wasted. In order to solve the blocking effect and ensure that consumers are not deployed before completing the operations of the respective producers, Flink provides different scheduling strategy for batch processing. The lazy from source scheduling, starts from source and deploys subtasks according to their topological order, this ensures that a subtask can only be deployed if all of its inputs are ready. One limitation of this approach is that it operates on individual tasks and thus treating all tasks in a similar fashion, without considering the subtasks that are non-blocking and can be executed in parallel. To address this limitation, the pipelined region scheduling is introduced in Flink, that analyzes Job graph and identify pipelined regions before deploying tasks and subtasks. It schedules each region only when all of its predecessors are executed thereby making all of its inputs ready. If there are enough available resources, JobManager will try to execute as many pipeline regions in parallel as possible [57].

4.5.4. Fault Tolerance

Flink provides reliable execution with exactly-once consistency guarantees. It takes regular distributed snapshots of data stream and operator states. These snapshots serve as consistent checkpoints, in case of failure, the system can fall back to these checkpoints. Flink's mechanism for taking these snapshots is introduced in [58] which is inspired by the standard Chandy-Lamport's asynchronous distributed snapshot algorithm [59]. A general assumption made by the Flink's fault tolerance mechanism is that the data source is replay able. In addition to checkpoints recovery mechanism that is automatically triggered upon failure, Flink also provides SavePoint mechanism which is manually triggered and managed by the user. SavePoints support pause-and-resume jobs and scheduled backup. If a TaskManager fails, JobManager requests ResourceManager for more processing slots, restores the state of failed process using checkpoints, and re-executes it on another processing slot. However, a failure at JobManager is critical, since JobManager monitors the execution of streaming applications and manages related metadata. Flink supports a high-availability mode, which is based on Apache ZooKeeper that stores all necessary metadata on a reliable remote storage system. When the JobManager fails, a new or a standby JobManager takes over the work of the failed JobManager by requesting the Zookeeper to yield JobGraph, metadata and state handles of last successful check point of the application from the remote storage.

5. Comparative Analysis of Big Data Frameworks

This section presents a comparative analysis of the big data frameworks discussed above. We identified various parameters and features to compare big data systems that include scalability, fault tolerance, data processing model, support for programming languages, data storage, resource management, throughput, latency, scheduling, and maturity. To provide a clear representation of the related concepts, the related features are grouped to form a feature vector. To this end, seven distinct feature vectors are defined: general features, performance related features, fault tolerance, data processing, architectural features, machine learning support, and stream processing. These feature vectors are summarized in Table 2 and discussed in the following subsections.

Table 2. Description of feature vectors.

Feature Vectors	Components of Feature Vector
General features	Main backers, maturity, implementation language, support for programming languages.
Architectural features	Architecture model, data storage, resource management, scheduling, security.
Data processing	Data processing model, execution model, processing mode, support for in-memory processing.
Performance	Latency, throughput, scalability.
Fault tolerance	Failure identification, failure handling, high availability.
Machine learning	Native support, compatibility, supported ML algorithms
Stream processing	Processing guarantees, state management, data source, processing format, stream primitives

5.1. General Feature Vector

This feature vector holds general characteristics of frameworks such as, main backers, maturity, implementation language, support for programming languages. Table 3 summarizes the comparison of the frameworks against general features. Every framework has some strong backing of a well-established business company, for example Hadoop is primarily backed by Google and Yahoo, Spark is backed by AMP Lab, whereas Storm's main backers are BackType and Twitter, Samza and Flink are backed by LinkedIn and dataArtisans respectively. As regards language support, it can be seen that each framework supports a number of programming languages with Storm topping the list with the support of widest range of languages because Storm uses Thrift [60] definition to define and deploy topologies. Since Thrift has code generation support for any high level programming language so Storm topologies can also be defined using any programming language, thus making Storm more developer friendly than its competitors. Maturity is a significant parameter from adoption perspective, it is generally preferred that the framework is mature and well tested. Currently, Hadoop has overcome its unstable stage and it has now developed more reliable and stable among its less mature counter parts. On the other hand, as of today, Flink is still in its early stages of evolution, many features are continuously being modified making Flink a little difficult to understand as a beginner, because there may be less community support and limited active forums to discuss Flink-related queries.

Table 3. Comparison of big data frameworks with respect to general features.

	Hadoop	Spark	Storm	Samza	Flink
Main backers	Google, Yahoo!	AMP Lab	Backtype, Twitter	LinkedIn	dataArtisans
Implementation language	Java	scala	clojure	scala, java	java, Scala
Programming language support	most of the high level languages	java, Scala, python, R	any programming language	JVM languages	java, Scala, python, R
Maturity	very high	high	high	medium	low

5.2. Feature Vector Related to Architectural Components

Since, big data frameworks distribute workloads across multiple processors, that requires splitting and distributing data files, managing the storage of data in a distributed

file system, scheduling tasks across the available computing and ensuring security. In this feature set we grouped together architecture related features such as architecture model, resource manager, storage, scheduling and security. Table 4 summarizes the comparison of the frameworks against architecture related features. It can be seen that four of most popular frameworks are based on master/slave architecture, whereas, Samza has no system wide master, instead Samza's architecture is based on publish/subscribe model. Most of the frameworks under discussion support resource management through Hadoop YARN and Apache Mesos. All the frameworks provide reliable and fault tolerant computing through different abstractions. Many big data applications are migrating from in-house storage and preferred to be deployed in cloud environment where different users can easily access or maintain privacy-sensitive data that leads to privacy and security risks. To provide authorization and authentication for computing nodes, Hadoop and Storm use the Kerberos authentication protocol [61]. Spark uses a password-based configuration as well as Access Control Lists (ACLs) to ensure security. Flink uses Kerberos and TLS/SSL authentication and Samza has no built-in support for security.

Table 4. Comparison of big data frameworks with respect to architectural features.

	Hadoop	Spark	Storm	Samza	Flink
Architecture Model	master-slave	master-slave	master-slave	publish-subscribe	master-slave
Resource Manager	YARN	stand alone, YARN, Mesos	YARN, Mesos	stand alone, YARN, Mesos	stand alone, YARN, Mesos
Storage	HDFS	HDFS, HBase, Hive, Casandra	HDFS	HDFS	HDFS, streams databases,
Scheduling	Fair, FIFO, Capacity	FIFO, Fair	default, isolation, multitenant, resource aware	YARN scheduler	all at once, lazy from source, pipelined region
Security	Kerberos authentication protocol	Password based shared secret configuration, Access Control Lists (ACLs)	Kerberos authentication protocol	no built-in security	Kerberos and TLS/SSL authentication

5.3. Feature Vector Related to Data Processing

Data processing in big data frameworks define how the data is processed and how the computations are represented in the system. Hadoop is a batch processing system that is best suited for large scale data processing. Storm and Samza are stream processing systems. Spark and Flink are hybrid systems that can handle batch as well as streaming data. Stream processing frameworks can handle virtually unlimited data volumes. However, they generally process streaming data in one of the two ways, native streaming that processes data items as they arrive or micro-batching which processes very small batches of incoming data. Spark and Storm uses micro-batching while Samza and Flink uses native streaming. Table 5 compares big data frameworks on the basis of data processing features.

Table 5. Comparison of big data frameworks with respect to feature vector related to data processing.

	Hadoop	Spark	Storm	Samza	Flink
Execution Format	batch only	batch and stream	stream only	stream only	batch and stream
Data Processing Model	MapReduce	DAG	Topology	DAG of operators	data flow graph
Processing Mode	batch processing	micro-batching	micro-batching	native streaming	native streaming
In-memory processing	No	Yes	Yes	yes	yes

5.4. Feature Vector Related to Performance

This feature vector includes performance metrics and those parameters that have a significant impact on the performance of a big data framework. This includes latency

that defines how quickly a data item can be processed, throughput and scalability. The desired value of latency is as low as possible and the desired value for throughput is as high as possible. Scalability defines the ability of a system to adapt to change in workloads by involving additional resources. Scalability can be achieved either by making existing resources stronger and faster so that they can efficiently handle increased workload (referred as scale-up) or adding more resources in parallel to spread out the increased load (referred as scale-out). Comparison of big data frameworks with respect to performance related features is shown in Table 6.

Table 6. Comparison of big data frameworks with respect to performance related features.

	Hadoop	Spark	Storm	Samza	Flink
Latency	high (seconds)	low (few seconds)	very low (sub seconds)	very low (sub seconds)	very low (sub seconds)
Throughput	high	High	medium	high	high
Scalability	high	Moderate	moderate	low	high

5.5. Feature Vector Related to Fault Tolerance

To improve the reliability and performance of big data systems, the adoption of fault tolerant systems has grown over the years. The need for highly fault tolerant systems arise due to the frequent failures caused by the complexity, scale and heterogeneity of underlying systems. Fault detection is the starting point of any fault-tolerant mechanism, that enables faults to be detected as soon as they appear within the system. In large-scale systems, stable fault detection methods such as heartbeat mechanism and fault prediction are used. Most big data frameworks are based on the heartbeat detection approach [18]. After fault detection, fault recovery is used to restore the faulty component's normal behavior. Data replication is typically used to ensure the reliability of storage systems. Google File System and HDFS both storage systems employed this method for fault tolerance and high availability. Redundant processing and redundant storage of in-processing metadata is used to recover faults that occur during the data processing. Checkpointing is another approach to ensure fault tolerance in big data systems which is used by stream processing engines or real time transactions where low latency is desired. Table 7 presents a comparison of frameworks on the basis of feature vector related to fault tolerance.

Table 7. Comparison of big data frameworks with respect to features related to fault tolerance.

	Hadoop	Spark	Storm	Samza	Flink
Fault detection	heart beat mechanism	heart beat mechanism	heart beat mechanism	Keeping in-memory record of all emitted tuples and tracks them within a configured timeout	heart beat mechanism
Fault Recovery	Data Replication	RDD lineage	Tuples acknowledgement, Zookeeper	Change log capturing	Light weight distributed snapshots
High availability	Redundant standby NameNodes and ResourceManagers	Multiple masters with Zookeeper	Running multiple standby Nimbus servers	Relies on YARN's high availability mode	Stores JobGraph and all necessary metadata on a reliable remote storage system

5.6. Feature Vector Related to Support for Machine Learning

Machine learning is a powerful tool that enables the use of data to make predictions and help to make decisions. The core of machine learning is data that empowers the underlying models. The emerging technologies of big data processing heavily incorporate machine learning tools to help make smarter and more informed decisions based on data.

Since machine learning concepts and algorithms are increasingly used in big data analytics, thus almost all of the frameworks are complemented with machine learning libraries and toolkits. Though, no special platform or library is required to execute machine learning tasks on Hadoop clusters, yet, a set of machine learning packages that can be run on Hadoop includes Mahout, H_2O, Distributed Weka and Oryx [10]. Spark and Flink have their native machine learning libraries. Spark's MLlib aims to simplify machine learning tasks for Spark. Its core functions include clustering, classification, collaborative filtering and regression. Additionally, it also includes algorithms for dimension reduction, transformation, optimization and feature extraction [10]. FlinkML is Flink's native machine learning library which was released in April 2015. It aims at providing extensible machine learning tools and algorithms for the development of complex machine learning applications. Storm and Samza both do not offer native machine learning library. However, both have compatibility with SAMOA [62]. As can be seen from Table 8, that all five frameworks have support for either native or in compatibility mode for machine learning tools that implements clustering, classification and regression algorithms. However, H_2O is the only machine learning platform considered in this paper that implements deep learning algorithms, consequently, those frameworks that can be integrated with H_2O offers deep learning algorithms.

Table 8. Support for machine learning tools and algorithms in big data frameworks.

Frameworks	**Machine Learning Tools**		**Machine Learning Algorithms**				
	Native ML Support	Compatibility	Deep Learning	Clustering	Collaborative Filtering	Classification	Regression
Hadoop	Mahout	Distributed Weka, Oryx, H_2O, SAMOA	✔	✔	✔	✔	✔
Spark	MLlib	Distributed Weka, H_2O, Mahout, Oryx	✔	✔	✔	✔	✔
Storm	-	SAMOA, H_2O	✔	✔	×	✔	✔
Samza	-	SAMOA	×	✔	×	✔	✔
Flink	FlinkML	SAMOA	×	✔	✔	✔	✔

5.7. Feature Vector Related to Stream Processing

The streaming frameworks apply computations on the data as it enters the system. This requires a different processing model from batch processing. Rather than defining the operations that are applied to the entire dataset, stream processors define the operations that will be applied to individual data items or extremely small batches of data. These operations usually maintain no or minimal state in between record. However, some frameworks provide some mechanism to maintain state. There are some important features that revolve around stream processing frameworks, such as state management, processing guarantees, stream primitives etc. Table 9 compares the stream processing frameworks on the basis of these features.

Table 9. Comparison of big data frameworks with respect to features related to stream processing.

	Spark	**Storm**	**Samza**	**Flink**
Processing Guarantees	exactly once	at least once	at least once	exactly once
Data Source	HDFS, DBMS, Kafka	Spout	Kafka	HDFS, DBMS, Kafka
Processing Format	micro-batches	micro-batches	continuous flow streaming	continuous flow streaming, batched, micro-batched
Stream Primitives	Dstream	Tuple	message	datastream
State Management	stateful	stateless	stateful operators	stateful operators

6. Discussion

Hadoop MapReduce leverages disk storage. As each task needs to access the disk to read and write multiple times, so it is usually very slow. However, since disk space is typically one of the copious resources on a server, which means, MapReduce can process huge datasets. This also means that MapReduce jobs can usually be executed on less expensive hardware than some alternatives because it doesn't use in-memory computations. This makes Hadoop best suited for very large datasets where execution time is not a constraint. Hadoop MapReduce is highly scalable, it can scale up to thousands of nodes (Hadoop cluster at Yahoo! is reported to have 42,000 nodes [11]). Hadoop not only has a strong ecosystem but is also often used as a building block for other frameworks. Several frameworks and execution engines integrate Hadoop to use the capabilities of HDFS and YARN.

Spark is significantly faster than Hadoop due to its in-memory data structure RDDs and DAG scheduling. It supports both batch and stream processing models, which gives the advantage of managing multiple processing workloads from a single cluster. In addition to its core capabilities, Spark has a sound set of libraries for machine learning, interactive queries, and iterative jobs, etc. It is largely acknowledged in developer communities that Spark tasks are easier to write than MapReduce, which has a significant impact on performance [63]. Spark uses micro-batches for processing which means it buffers data as it enters the system. Though, the buffer is capable of keeping a large amount of data but waiting for the buffer to be flushed can cause a significant increase in latency, thereby making Spark streaming a less suitable choice for the applications where low latency is required. However, Spark is a good fit for applications where high throughput is more desired than latency. Also, because Spark uses in-memory computations and memory is often more expensive than disk, Spark may be more expensive than disk-based frameworks. However, faster execution means tasks can be finished early, which can completely offset the cost of working in a paid work environment. Yet, in a multitenant environment, Spark may be a less considerate neighbor as compared to Hadoop due to its extensive resource usage.

Storm is one of the most trusted solutions for the near real-time workloads that need to be processed with minimal latency. Storm is usually a good choice when processing time has a direct impact on the user experience, for example during interactive web sessions. Storm provides at least once processing guarantee, which means that every message is guaranteed to be processed, but some messages may be processed more than once. However, after the release of Trident, it supports exactly once processing guarantee. Storm does not support batch processing, though, Storm with Trident offers the flexibility of using micro-batches as an alternative to pure streaming. For interoperability, Storm integrates with the YARN and can be easily connected to existing Hadoop implementations. Strom is polyglot and supports more programming languages than any other framework.

Samza is heavily reliant on Kafka to ensures some unique features to the system. For example, Kafka provides replicated storage with notably low latency and a low-cost multi-subscription model for each data partition. Samza's fault tolerance mechanism has a strong contribution coming from Kafka. Results are also written back to Kafka that can be ingested by later stages. Writing the results directly to Kafka also helps eliminating backpressure problem [63]. Backpressure occurs when the peak load causes data to flow faster than the real-time processing speed of the components, resulting in processing downtime and possible data loss. Kafka is capable of holding data for longer time periods that enable components to read and process data at their convenience. The strong coupling between samza and Kafka leads to the loosely coupled processing stages. Samza is suitable for organizations in which multiple teams may need to access data streams at different stages of processing because several subscribers can consume the output of each stage. Samza can store state with the help of a fault-tolerant check pointing system and offers at-least once processing guarantees. This makes it provide inaccurate recovery of the aggregated state such as counts because, in case of failure, data can be delivered multiple times.

Like Spark, Flink is also a hybrid framework that provides low-latency streaming and supports traditional batch processing tasks. However, Spark is not preferred for streaming in many use cases due to its micro-batch processing style. While Flink provides real stream processing with low latency and high throughput. Flink treats batch processing as a special case of streaming i.e., bounded data stream. This is contrary to other frameworks approach, where batch-processing is the primary processing model and streaming is used as a subset of it. Unlike other frameworks that relies on Java garbage collector, Flink has its own memory management mechanism that handles garbage collection, partitioning and caching. While running in the Hadoop stack, it is designed to be a fair neighbor and only consumes the resources it needs at any given time. Flink's language and rich API support is limited to Java and Scala thereby, making it not a good fit if more support for high level APIs is needed.

To summarize the discussion above, there is no best fit for all solution, rather every framework has its own strengths and weaknesses. Which framework is most appropriate for a project is dependent largely on the nature of data being processed, the time constraints, and the type of processing that is intended in that particular project. There is a trade-off between implementing the best fit for all system and dealing with highly focused projects, and similar considerations apply when comparing innovative but new systems with well-tested and mature solutions. Table 10 presents the application use cases where each of the discussed frameworks ideally fits in.

Table 10. Application use cases where each of the frameworks ideally fits in.

Big Data Frameworks	Best Fit Application Use-Cases
Hadoop	Applications that require batch processing of very large datasets where execution time is not a hard constraint
Spark	Applications with batch or streaming workloads where high throughput is more desired than latency
Storm	Streaming applications where extremely low latency is desired with at least once processing guarantees
Samza	Streaming applications that require multiple teams to access same data streams at different stages of processing
Flink	Applications with batch or streaming workloads where extremely low latency is desired with exactly once processing guarantees
Spark	Applications with batch or streaming workloads where high throughput is more desired than latency
Storm	Streaming applications where extremely low latency is desired with at least once processing guarantees

7. Conclusions

In the presence of a number of available big data processing frameworks selecting most appropriate framework according to application context is non-trivial. To choose the appropriate framework for an application, one must consider a number of features and characteristics of the system. In the literature, several studies have performed comparisons of big data frameworks but these lack detailed comparison of architecture with respect to the core functionality of its components. No specific guidance is provided to help developers and practitioners in the selection of a suitable framework for their application. Furthermore, the classification of features that are used for comparative analysis is lacking. To fill this research gap, our work aims to provide a comprehensive review of most popular big data frameworks in an attempt to highlight the strengths and weaknesses of each framework. The features used for comparative analysis are logically classified the into seven feature vectors. The frameworks are thoroughly compared with respect to identified feature vectors. Furthermore, we pointed out the application use cases where each of the frameworks ideally fit in and a detailed discussion is presented that can serve as a decision making guide to select the appropriate framework for an application.

This work differs from existing studies in number of ways. First, a detailed-oriented review of big data frameworks is presented with respect to four significant aspects: architecture, fault tolerance, data processing model and scheduling. Second, this paper presents a purposeful discussion related to the core characteristics of big data frameworks. The

findings can help in selecting the most competent and preferable system according to particular scenario or application requirement.

We believe that our work will benefit the researchers and practitioners in the following ways:

- In the literature, the qualitative comparisons were performed in bits and pieces and architecture of the frameworks were discussed and analyzed briefly. Furthermore, the classification of features that are used for comparative analysis is lacking. In this paper, we have logically classified the features into seven feature vectors. We thoroughly compared the frameworks in terms of identified feature vectors.
- Although, the official documentation of the frameworks contains information about the architecture of the framework, but this information cannot be used to compare the frameworks because there is no consistent view under which the information is presented for each framework. The documentation varies from framework to framework on the basis of the format on which the information is presented and the characteristics are considered. Hence, there is no consistent view available through which one can compare these frameworks and conclude some frameworks selection guidelines. We have analyzed the popular big data frameworks in detail under a consistent view of feature vectors. That is, all the frameworks are compared and discussed on the basis of the four significant aspects and seven feature vectors. Thus, we have presented the information in systematic way. The readers can easily assess the pros and cons of each framework under a consistent view. Since, the information has been structured in a systematic way therefore, it is easy to perform a cross-comparison of the systems.

Author Contributions: M.K. was responsible for conceptualization and methodology of the work; M.K. and M.M.Y. participated in writing and editing the manuscript and analyzing data. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Reinsel, J.G. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Internet Data Cent, IDC: iView: IDC Analyze the Future 2007. 2012, pp. 1–16. Available online: http://www.emc.com/collateral/analyst-reports/idc-the-digital-uni (accessed on 22 June 2021).
2. Reinsel, D.; Gantz, J.; Rydning, J. The Digitization of the World—From Edge to Core. Internet Data Cent. 2018, pp. 1–28. Available online: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (accessed on 22 June 2021).
3. Chebbi, I.; Boulila, W.; Farah, I.R. Big Data: Concepts, challenges and applications. In *Computational Collective Intelligence, Lecture Notes in Computer Science*; Springer: Cham, Switzerland, 2015; Volume 9330, pp. 638–647.
4. Dugas, A.F.; Jalalpour, M.; Gel, Y.; Levin, S.; Torcaso, F.; Igusa, T.; Rothman, R.E. Influenza forecasting with Google Flu Trends. *PLoS ONE* **2013**, *8*, 2. [CrossRef]
5. Maier, M. Towards a Big Data Reference Architecture. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2013.
6. Chen, P.C.L.; Zhang, C.Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.* **2014**, *275*, 314–347. [CrossRef]
7. Singh, D.; Reddy, C.K. A survey on platforms for big data analytics. *J. Big Data* **2015**, *2*, 8. [CrossRef] [PubMed]
8. Morais, T. Survey on Frameworks for Distributed Computing: Hadoop, Spark and Storm. In Proceedings of the 10th Doctoral Symposium in Informatics Engineering—DSIE'15, Porto, Portugal, 29–30 January 2015.
9. Hesse, G.; Lorenz, M. Conceptual Survey on Data Stream Processing Systems. In Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, VIC, Australia, 14–17 December 2015; pp. 797–802. [CrossRef]

10. Landset, S.; Khoshgoftaar, T.M.; Richter, A.N.; Hasanin, T. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *J. Big Data* **2015**, *2*, 1–36. [CrossRef]
11. Ullah, S.; Awan, M.D.; Khiya, M.S.H. Big data in cloud computing: A resource management perspective. *Sci. Program.* **2018**, *8*, 1–7. [CrossRef]
12. Bajaber, F.; Elshawi, R.; Batarfi, O.; Altalhi, A.; Barnawi, A.; Sakr, S. Big data 2.0 processing systems: Taxonomy and open challenges. *J. Grid Comput.* **2016**, *14*, 379–405. [CrossRef]
13. Assunção, M.D.d.; Veith, A.d.S.; Buyya, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Netw. Comput. Appl.* **2018**, *103*, 1–17. [CrossRef]
14. Inoubli, W.; Aridhi, S.; Mezni, H.; Maddouri, M.; Nguifo, E.M. An experimental survey on big data frameworks. *Future Gener. Comput. Syst.* **2018**, *86*, 546–564. [CrossRef]
15. Veiga, J.; Expósito, R.R.; Pardo, X.C.; Taboada, G.L.; Tourifio, J. Performance evaluation of big data frameworks for large-scale data analytics. In Proceedings of the IEEE International Conference on Big Data, Washington, DC, USA, 5–8 December 2016; pp. 424–431.
16. Hazarika, A.V.; Ram, G.J.S.R.; Jain, E. Performance comparison of Hadoop and spark engine. In Proceedings of the I-SMAC (IoT in Social, Mobile, Nalytics and Cloud), Palladam, India, 10–11 February 2017; pp. 671–674.
17. Gupta, H.K.; Parveen, D.R. Comparative study of big data frameworks. In Proceedings of the International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), Ghaziabad, India, 27–28 September 2019; pp. 1–4. [CrossRef]
18. Saadoon, M.; Hamid, S.H.A.; Sofian, H.; Altarturi, H.H.M.; Azizul, Z.H.; Nasuha, N. Fault tolerance in big data storage and processing systems: A review on challenges and solutions. *Ain Shams Eng. J.* **2021**, in press. [CrossRef]
19. Bartolini, I.; Patella, M. Comparing performances of big data stream processing platforms with RAM3S. In Proceedings of the 25th Italian Symposium on Advanced Database Systems (SEBD), Squillace Lido, Italy, 25–29 June 2017; pp. 145–152.
20. To, Q.C.; Soto, J.; Markl, V. A survey of state management in big data processing systems. *VLDB J.* **2018**, *27*, 847–872. [CrossRef]
21. Cumbane, S.P.; Gidófalvi, G. Review of big data and processing frameworks for disaster response applications. *ISPRS Int. J. Geo-Inf.* **2019**, *8*, 387. [CrossRef]
22. Inoubli, W.; Aridhi, S.; Mezni, H.; Maddouri, M.; Nguifo, E. A comparative study on streaming frameworks for big data. In Proceedings of the 44th International Conference on Very Large Databases: Workshop LADaS-Latin American Data Science, Rio De Janeiro, Brazil, 27–31 August 2018; pp. 1–8.
23. Patil, A. Distributed Programming Frameworks in Cloud Platforms. *Int. J. Recent Technol. Eng.* **2019**, *7*, 1–9.
24. Demchenko, Y.; de Laat, C.; Membrey, P. Defining Architectural Components of the Big Data Ecosystem. In Proceedings of the International Conference on Collaboration Technologies and Systems (CTS), Minneapolis, MN, USA, 19–23 May 2014; pp. 104–112. [CrossRef]
25. Park, E.; Sugumaran, V.; Park, S. A Reference Model for Big Data Analytics. In Proceedings of the 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), New York, NY, USA, 8–10 November 2018; pp. 382–391. [CrossRef]
26. Ghemawat, S.; Gobioff, H.; Leung, S. The Google file system. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing, NY, USA, 19–22 October 2003; pp. 29–43.
27. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
28. White, T. *Hadoop: The Definitive Guide*; O'Reilly Media: Newton, MA, USA, 2009.
29. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The Hadoop distributed file system. In Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010; pp. 1–10.
30. Polato, I.; Goldman, R.R.A.; Kon, F. A comprehensive view of Hadoop research—A systematic literature review. *J. Netw. Comput. Appl.* **2014**, *46*, 1–25. [CrossRef]
31. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
32. Saha, B.; Shah, H.; Seth, S.; Vijayaraghavan, G.; Murthy, A.; Curino, C. Apache Tez: A unifying framework for modeling and building data processing applications. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 1357–1369.
33. Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In Proceedings of the International Conference on Management of Data (SIGMOD '08), Vancouver, BC, Canada, 9–12 June 2008; pp. 1099–1110.
34. Salloum, S.; Dautov, R.; Chen, X.; Peng, P.X.; Huang, J. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* **2016**, *1*, 145–164. [CrossRef]
35. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX NSDI'12 USENIX Association, San Jose, CA, USA, 25–27 April 2012.
36. Armbrust, M.; Xin, R.S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J.K.; Meng, X.; Kaftan, T.; Franklin, M.J.; Ghodsi, A.; et al. Spark SQL: Relational data processing in spark. In Proceedings of the International Conference on Management of Data (SIGMOD '15), Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 1383–1394.

37. Zaharia, M.; Das, T.; Li, H.; Hunter, T.; Shenker, S.; Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13), Farminton, PA, USA, 3–6 November 2013; pp. 423–438.
38. Meng, X.; Bradley, J.; Yavuz, B.; Sparks, E.; Venkataraman, S.; Liu, D.; Freeman, J.; Tsai, D.; Amde, M.; Owen, S.; et al. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.* **2016**, *17*, 1–7.
39. Xin, R.S.; Gonzalez, J.E.; Franklin, M.J.; Stoica, I. GraphX: A resilient distributed graph system on Spark. In Proceedings of the 1st Int. Workshop on Graph Data Management Experiences and Systems (GRADES '13), New York, NY, USA, 23 June 2013; Volume 2, pp. 1–6.
40. Venkataraman, S.; Yang, Z.; Liu, D.; Liang, E.; Falaki, H.; Meng, X.; Xin, R.; Ghodsi, A.; Franklin, M.; Stoica, I.; et al. SparkR: Scaling R programs with Spark. In Proceedings of the International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016; pp. 1099–1104.
41. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.; Shenker, S.; Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11), Boston, MA, USA, 30 March–1 April 2011; pp. 295–308.
42. Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J.M.; Kulkarni, S.; Jackson, J.; Gade, K.; Maosong, F.; Donham, J.; et al. Storm@twitter. In Proceedings of the ACM International Conference on Management of Data (SIGMOD '14), Snowbird, UT, USA, 22–27 June 2014; pp. 147–156.
43. Iqbal, M.H.; Soomro, T.R. Big data analysis: Apache Storm perspective. *Int. J. Comput. Trends Technol.* **2015**, *19*, 9–14. [CrossRef]
44. Hunt, P.; Konar, M.; Junqueira, F.; Reed, B. ZooKeeper: Wait-free coordination for internet-scale systems. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 23–25 June 2010; pp. 1–11.
45. Kreps, J.; Narkhede, N.; Rao, J. Kafka: A distributed messaging system for log processing. In Proceedings of the SIGMOD Workshop on Networking Meets Databases, Athens, Greece, 12 June 2011.
46. Muhammad, A.; Aleem, M. A3-Storm: Topology, traffic, and resource-aware storm scheduler for heterogeneous clusters. *J. Supercomput.* **2021**, *77*, 1059–1093. [CrossRef]
47. Cardellini, V.; Grassi, V.; Presti, F.L.; Nardelli, M. Optimal operator placement for distributed stream processing applications. In Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16), Irvine, CA, USA, 20–24 June 2016; pp. 69–80.
48. Aniello, L.; Baldoni, R.; Querzoni, L. Adaptive online scheduling in Storm. In Proceedings of the 7th ACM international conference on conference on Distributed event-based systems, Arlington, TX, USA, 29 June–3 July 2013; pp. 207–218.
49. Peng, B.; Hosseini, M.; Hong, Z.; Farivar, R.; Campbell, R. R-Storm: Resource-aware scheduling in Storm. In Proceedings of the 16th Annual Middleware Conference (Middleware '15), Vancouver, BC, Canada, 7–11 December 2015; pp. 149–161.
50. Xu, J.; Chen, Z.; Tang, J.; Su, S. T-Storm: Traffic-aware online scheduling in Storm. In Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS 13), Madrid, Spain, 30 June–3 July 2014; pp. 535–544.
51. Jian, T.; Xu, J. A predictive scheduling framework for fast and distributed stream data processing. In Proceedings of the IEEE International Conference on Big Data, Santa Clara, CA, USA, 29 October–1 November 2015; pp. 333–338.
52. Noghabi, S.A.; Paramasivam, K.; Pan, Y.; Ramesh, N.; Bringhurst, J.; Gupta, I.; Campbell, R.H. Samza: Stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.* **2017**, *10*, 1634–1645. [CrossRef]
53. Apache Samza. Available online: http://samza.apache.org/powered-by/ (accessed on 22 June 2021).
54. Alexandrov, A.; Bergmann, R.; Ewen, S.; Freytag, J.C.; Hueske, F.; Heise, A.; Kao, O.; Leich, M.; Leser, U.; Markl, V.; et al. The stratosphere platform for big data analytics. *VLDB J.* **2014**, *23*, 939–964. [CrossRef]
55. Armoogum, S.; Li, X. Big data analytics and deep learning in bioinformatics with Hadoop. In *Deep Learning and Parallel Computing Environment for Bioengineering Systems*; Academic Press: Cambridge, MA, USA, 2018; pp. 17–36.
56. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* **2015**, *36*, 28–38.
57. Zagrebin, A. Improvements in Task Scheduling for Batch Workloads in Apache Flink. Available online: https://flink.apache.org/2020/12/15/pipelined-region-sheduling.html#the-new-Pipelined-region-scheduling (accessed on 2 June 2020).
58. Carbone, P.; Fora, G.; Ewen, S.; Haridi, S.; Tzoumas, K. Lightweight asynchronous snapshots for distributed dataflows. *arXiv* **2015**, arXiv:1506 08603.
59. Chandy, K.M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **1985**, *3*, 63–75. [CrossRef]
60. Apache Thrift. Available online: https://thrift.apache.org/ (accessed on 22 June 2021).
61. Zhang, X.; Liu, C.; Nepal, S.; Dou, W.; Chen, J. Privacy-preserving layer over MapReduce on cloud. In Proceedings of the 2nd International Conference on Cloud and Green Computing, CGC, Xiangtan, China, 1–3 November 2012; pp. 304–310.
62. Morales, G.D.F.; Bifet, A. SAMOA: Scalable advanced massive online analysis. *J. Mach. Learn. Res.* **2015**, *16*, 149–153.
63. Ellingwood, J. Hadoop, Storm, Samza, Spark, and Flink: Big Data Frameworks Compared. Digital Ocean. 2016. Available online: https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared (accessed on 12 August 2021).