RESEARCH

Programming Big Data Analysis: Principles and Solutions

Loris Belcastro[1,2], Riccardo Cantini[1], Fabrizio Marozzo[1,2*], Alessio Orsino[1], Domenico Talia[1,2] and Paolo Trunfio[1,2]

*Correspondence:
fmarozzo@dimes.unical.it
[1]University of Calabria, Rende, Italy
Full list of author information is available at the end of the article

Abstract

In the age of the Internet of Things and social media platforms, huge amounts of digital data are generated by and collected from many sources, including sensors, mobile devices, wearable trackers and security cameras. This data, commonly referred to as Big Data, is challenging current storage, processing, and analysis capabilities. New models, languages, systems and algorithms continue to be developed to effectively collect, store, analyze and learn from Big Data. Most of the recent surveys provide a global analysis of the tools that are used in the main phases of Big Data management (generation, acquisition, storage, querying and visualization of data). Differently, this work analyzes and reviews parallel and distributed paradigms, languages and systems used today to analyze and learn from Big Data on scalable computers. In particular, we provide an in-depth analysis of the properties of the main parallel programming paradigms (MapReduce, workflow, BSP, message passing, and SQL-like) and, through programming examples, we describe the most used systems for Big Data analysis (e.g., Hadoop, Spark, and Storm). Furthermore, we discuss and compare the different systems by highlighting the main features of each of them, their diffusion (community of developers and users) and the main advantages and disadvantages of using them to implement Big Data analysis applications. The final goal of this work is to help designers and developers in identifying and selecting the best/appropriate programming solution based on their skills, hardware availability, application domains and purposes, and also considering the support provided by the developer community.

Keywords: Parallel Programming models; Programming systems; Big Data analysis; MapReduce; Workflow; Message Passing; Bulk Synchronous Parallel; SQL-like

1 Introduction

Over the last years, with the development of the Internet of Things, the growth of social networks and the widespread diffusion of mobile devices, enormous amounts of digital data are being generated by and gathered from several sources. For instance, data from sensors, webcams, in-vehicle infotainment, mobile devices, GPS devices, wearable trackers, social networks and web services is drastically rising. This huge amount of data, commonly referred to as Big Data, is characterized by the complexity, by the variety in terms of format [1], and is produced at a speed that is challenging the current storage, processing and analysis capabilities. In fact, if on the one hand it opens up to several opportunities to extract useful information and

produce valuable knowledge for science [2], economy [3], health [4], and society [5], on the other hand, its volume and speed are overwhelming the ability to use it.

To extract valuable information from the analysis of such data, novel architectures, programming models and systems have been developed in the last years that address their complexity and/or high velocity [6, 7]. In this scenario, data mining and machine learning have grown over the past decades as two research and technology fields that provided several different techniques and algorithms to automatically extract hidden, unknown, but potential value from massive repositories [8]. However, sequential data analysis algorithms are not feasible for extracting useful models and patterns from huge volumes of data in a reasonable time. For this reason, high performance computers, such as many and multi-core systems, Clouds, and multi-clusters, along with parallel and distributed algorithms and systems are required by data scientists to tackle Big Data issues [9].

This work provides a structured overview of programming models and systems for Big Data analysis, which is the final and most important phase of the Big Data life cycle management (data generation, acquisition, storage, and analysis) [10]. Taking into account the most popular parallel programming models for Big Data analysis (MapReduce, workflow, Bulk Synchronous Parallel, message passing, and SQL-like), here we analyze the features of the main frameworks implementing them. For each framework, through code snippets and schemes, we show how data analysis applications can be implemented. The different frameworks have been compared according to three main aspects: programming features, diffusion and advantages/disadvantages. The programming feature comparison is based on three main criteria that assess the suitability of each framework in supporting parallel and distributed programming: *i*) *type of parallelism* that describes how a system allows for expressing parallel operations; *ii*) *level of abstraction* that refers to their programming capabilities for hiding low-level details; and *iii*) *class of applications* that describes the most common application domain of a system. To analyze the use and popularity of each framework, the diffusion analysis is based on four aspects: *i*) the *main companies* that use it, which describes its current status and its potential on an industrial scale; *ii*) the *API support* that describes the available programming languages to develop applications by using it; *iii*) the *community size*, in terms of the number of questions posed on Stack Overflow[1], one of the most popular Q&A sites for programming problems; and *iv*) *number of commits* and *stars* in its official GitHub[2] repository. Finally, the last comparison analyzes the *main technical advantages* and *disadvantages* of using each framework.

Through the analysis, comparison and programming examples featured in this manuscript, developers can find a useful way to identify and select the best solution based on their skills, hardware availability, application domains and purposes, and the support provided by the developer community. This manuscript extends the work presented in [6] in the following main aspects: *i*) we focused our attention on systems that are widespread and used by a large number of users around the world, by analyzing their characteristics and describing their peculiarities; *ii*) we showed

[1] https://stackoverflow.com/
[2] https://github.com/

data analysis examples supported by diagrams and code snippets, for all the frameworks we considered; and *iii*) we provided a broad comparison of frameworks based on different principles, such as programming aspects, size of the developer community and diffusion in the IT world, strengths and weaknesses of each framework.

The reminder of this paper is organized as follows. Section 2 presents related work and the main contribution of our study. Section 3 describes the most widespread programming models for Big Data analysis and introduces the main software frameworks implementing them. Section 4 discusses the most used framework for each programming model, also providing an application example for each of them. Section 5 presents an in-depth comparison of the described systems and Section 6 concludes the paper.

2 Related work

Big Data analysis has been discussed in several surveys and review papers, as well as books and research reports. Among those, some papers review the main challenges and state-of-the-art of Big Data. For example, Chen et al. [10] provided an extensive analysis of technologies related to Big Data, such as Cloud Computing, Internet of Things, and Data centers. The authors addressed the challenges of Big Data concerning data representation and reduction, life cycle, energy management, and scalability. They focused on the whole value chain of Big Data, from data generation to data acquisition (i.e., data collection, transportation, and preprocessing) and data storage in either distributed systems or NoSQL databases, right up to data analysis. Concerning data analysis systems, authors analyzed only MPI, Hadoop MapReduce, and Dryad [11], with a focus on Big Data mining tools such as R[3], Excel, RapidMiner[4], Weka[5], and Pentaho. Finally, they discussed different fields where Big Data can be applied, such as text, web data and network data analysis, but without discussing programming examples.

The paper by Oussous et al. [12] deals with Big Data challenges (management, cleaning, aggregation, and analysis) and mostly focused on the Hadoop ecosystem and distributions. In particular, the authors discussed the different layers and their main software: data storage (HDFS, HBase), data processing (MapReduce, YARN) and querying (Pig, Hive), data ingestion (Sqoop[6], Flume[7]) and streaming (Storm, Spark), and the management and deployment layer (Zookeeper, Oozie, Ambari).

Hu et al. [13] provided a technology-oriented tutorial on Big Data analytics tools. In particular, they presented a systematic framework to decompose Big Data systems in a value chain involving data generation, acquisition, storage, and analytics. They discussed approaches and mechanisms both from research and industry communities, presenting a few programming models with associated frameworks: MapReduce with Hadoop, Directed Acyclic Graph with Dryad, Storm and Apache S4 (Simple Scalable Streaming System)[8], and Directed Graph with Pregel and GraphLab for parallel machine learning [14].

[3] https://www.r-project.org/
[4] https://rapidminer.com/
[5] https://www.cs.waikato.ac.nz/ml/weka/
[6] https://sqoop.apache.org/
[7] https://flume.apache.org/
[8] http://incubator.apache.org/projects/s4.html

Yaqoob et al. [15] analyzed the state-of-the-art for Big Data technologies based on batch and stream data processing with related strengths and weaknesses. Big Data analysis techniques (data mining, web mining, machine learning, social network analysis) are discussed with case studies. Also emerging technologies are presented (granular computing, bio-inspired computing, quantum computing, semantic web, etc.) in the work.

Singh and Reddy [16] presented a survey on Big Data analytics by distinguishing between horizontal scaling and vertical scaling platforms. For the former class P2P networks, Hadoop, and Spark have been discussed, while for the latter one high performance computing clusters, multi-core systems, GPU, and FPGA have been reviewed. They provided a comparison of different platforms based on parameters such as scalability, data I/O performance, fault tolerance, real-time processing, data size supported and iterative tasks support. Finally, the development of a K-Means clustering example is analyzed on the different platforms.

Wang et al. [17] analyzed the infrastructure of Big Data service architectures for collection and storage of massive data. They analyzed different types of NoSQL databases and discussed data processing frameworks such as MapReduce, Dryad, Storm, Spark, Flink[9], and Pregel. The characteristics of these frameworks are analyzed in terms of scalability, real-time processing, reliability, data persistence, multi-language programming, memory programming, stream processing, batch computing, interactive query, and more. Finally, they focused on Big Data-based Cloud Computing service systems and presented some application scenarios such as recommendation systems, smart grid, and emotional analysis, but without presentation and discussion of code snippets.

Rao et al. [18] provided a generalized view of Big Data systems and models like MapReduce, Bulk Synchronous Parallel (BSP) and in-memory models. Distributed file systems and distributed machine learning tools are also discussed (Mahout, Spark MLlib, and FlinkML). Authors investigated Hadoop, Spark and Flink, providing a comparison and highlighting their advantages and limitations. Finally, they discussed interactive analytical processing tools (Hive, Impala[10], and Tez[11]), data ingestion tools (Flume, Sqoop, Chukwa[12]), and large-scale graph processing tools (GraphX for Spark and Gelly for Flink).

Also Saggi et al. [19] discussed Big Data analytics and decision-making frameworks based on machine learning. In particular, tools such as Spark, Hadoop, Mahout, R, and Giraph[13] were analyzed. Similarly, Tsai et al. [20] focused on Big Data analytics frameworks and platforms for data mining and machine learning algorithms (i.e., Hadoop paired with HDFS for storage and Mahout[14] for analytics).

As mentioned, most of the existing surveys on Big Data provided a global analysis of frameworks that are used in all phases of Big Data management (data generation, acquisition, storage, and analytics). However, only a few of them addressed the data

[9] https://flink.apache.org/
[10] https://impala.apache.org/
[11] https://tez.apache.org/
[12] http://chukwa.apache.org/
[13] https://giraph.apache.org/
[14] https://mahout.apache.org/

analysis and machine learning frameworks in detail. Those last ones did it with some significant differences from our work:

- We classified the main framework models into five categories and for each of them we provide an extensive analysis of the most widespread systems;
- For each system we discuss a real application scenario also using diagrams and code snippets, which can help programmers/developers to better understand how to structure a data analysis application;
- We provide a broad comparison of systems based on different principles and features including programming aspects, size of the developer community and diffusion in the IT world. Also strengths and weaknesses of each system are identified.

3 Programming models and systems

This section presents and discusses the most popular programming models for Big Data analysis and their major associated software frameworks. They are MapReduce, workflow, Bulk Synchronous Parallel (BSP), message passing, and SQL-like. The goal of the section is to highlight features, issues and benefits of each programming model and the software systems based on it.

3.1 MapReduce

MapReduce [21] is a programming model inspired by functional programming. It is based on the parallel execution of *map* and *reduce* functions for designing large-scale data-intensive applications. Specifically, the distributed execution of a MapReduce application is delegated to a set of mapper and reducer processes [22]. Each *mapper* executes the *map* function by reading a chunk of the input data and generating a list of intermediate key/value pairs. Those pairs are then shuffled and sorted on the basis of their keys, so as all pairs with the same keys are assigned to the same reducer. Hence, each *reducer* executes the reduce function which merges all the values associated with the same key to generate a possibly smaller set of values. The results of each reducer are then collected so as to generate the final output data. Figure 1 shows how input data is partitioned into a set of n chunks $c_1, c_2, \ldots, c_n$ and processed on horizontally-scaled units.

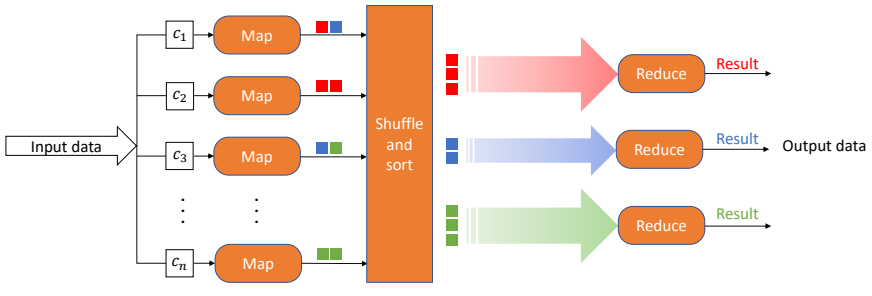

Figure 1 MapReduce data partitioning and processing on horizontally scaled units.

The MapReduce model is specially designed for data-intensive applications, such as social media analysis, image retrieval, scientific simulation, and website crawling.

In such applications, whose complexity is mainly linked to the large volume of data to be processed, MapReduce tries to move computation as close as possible to the data, which can avoid bottlenecks in data access. In particular, it allows a full exploitation of data-parallelism, enabling the efficient execution in distributed environments.

In addition, it can be adapted to several computing environments including multi-core, many-core and multi-cluster systems, dynamic cloud platforms, and high-performance computing systems [22]. For these reasons, nowadays it is considered one of the most important parallel programming models for distributed processing, which is supported by the major IT companies, including Google, Amazon[15], Microsoft[16] and IBM[17].

The most used open source framework based on the MapReduce programming model is *Apache Hadoop*[18], a general-purpose framework designed to process very large amounts of data in infrastructures with up to tens of thousands of distributed machines. It enables the development of distributed and parallel applications using many programming languages, relieving developers from having to deal with classical distributed computing issues, such as load balancing, fault tolerance, data locality, and network bandwidth saving.

Over the years, several minor implementations of the MapReduce model have been proposed and implemented, such as *Phoenix++* [23] and *Sailfish* [24], but none of these have ever achieved the same success as Hadoop. In particular, Phoenix++ is a C++ implementation that leverages multi-core chips and shared memory multi-processors. Its runtime automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance. Sailfish is a MapReduce framework for large-scale data processing, which facilitates batch transmission from mappers to reducers to improve performance. An abstraction called *I-files* is used for supporting data aggregation, adapting the original model to efficiently batch data written and read by multiple nodes.

3.2 Workflow

Workflows provide another important paradigm adopted by several frameworks for Big Data processing. A workflow is a well defined, and possibly repeatable, pattern designed to achieve a certain transformation of data [25], usually programmed as a graph. A workflow is developed as a graph composed of a finite set of directed edges and vertices, which can be used to model complex data analysis scenarios, such as distributed data mining, machine learning and stream analysis applications. Workflow tasks can be composed together following different patterns (e.g., loops, pipelines, parallel constructs), which enable the efficient modeling and execution of a wide range of applications where input, output, and tasks may depend on other tasks. A comprehensive collection of workflow patterns focusing on the description of control flow dependencies among tasks has been described in [26]. When a workflow does not include cycles, it can be referred to as Directed Acyclic Graph (DAG),

[15] https://aws.amazon.com/elasticmapreduce/
[16] https://azure.microsoft.com/services/hdinsight/
[17] https://www.ibm.com/analytics/us/en/technology/hadoop/
[18] https://hadoop.apache.org/

which is the most common programming structure used in workflow management systems [27] and adopted by the most famous systems.

Apache Spark[19] is one of the most popular frameworks based on the workflow paradigm. It relies on the DAG structure and is commonly used to develop applications that exploit in-memory computation (e.g., iterative machine learning algorithms), by caching data in RAM memory so as to speed up the execution compared to Hadoop [28]. Furthermore, many powerful and robust libraries are built on top of it for dealing with a wide range of applications involving machine learning, SQL analytics and graph computation.

While Spark has been designed as a general-purpose distributed computing engine for large-scale data processing, there exist many other frameworks that have been specifically developed to be used in a specific application domain. For instance, *Apache Storm*[20] is an open source and distributed system for real-time stream processing, capable of coping with huge amounts of unbounded data in large-scale infrastructures. Storm is designed to be highly scalable, fault-tolerant, and to ensure high-speed data processing (million tuples processed per second per node) with low-latency response time. A Storm application is outlined as a topology in the shape of a DAG, where spouts and bolts act as the graph vertices. *Apache Flink*[21] is another open source real-time stream processing system designed to deal with large volumes of data. Flink provides a DAG-based streaming dataflow paradigm for processing both finite and infinite datasets. Dataflow operations can simply look at one individual event at a time, or remember information across different events (i.e., stateful processing). The core of Flink is a distributed streaming dataflow runtime, which is an alternative to Hadoop MapReduce, and a rich set of APIs. Thanks to its user-friendly features, Flink can be used by small companies for business purposes.

Since workflows are used in a wide range of application domains, including scientific simulations, data analysis and machine learning, different parallel/distributed frameworks have been proposed for easing application design and execution, as well as exploiting distributed computational and/or storage resources efficiently. Among the workflow-based frameworks, *COMPSs* [29] includes a programming system and an execution runtime, designed to ease the development of scientific data analytics workflows for distributed environments. Users can create sequential applications in Java or Python and select which methods will be executed remotely. COMPSs runtime manages the parallel execution of an application, relieving users from dealing with the low-level infrastructure or classical distributed computing issues (e.g., synchronization, data transfer). The *Data Mining Cloud Framework (DMCF)* [30] is another software system for designing and executing distributed data analysis workflows. It integrates a visual workflow language, which allows users to model complex workflows without worrying about low-level aspects, and a parallel runtime based on a Software-as-a-Service (SaaS) model for executing them on Clouds. Its runtime is able to parallelize the execution of workflow tasks by exploiting the maximal concurrency according to data dependencies [31]. Other visual workflow management

[19] https://spark.apache.org
[20] https://storm.apache.org
[21] https://flink.apache.org

systems are *Kepler* [32], *YAWL* (Yet Another Workflow Language) [33], and *Pegasus* [34]. Kepler provides a graphical user interface for designing scientific workflows, where users to create a workflow can select and then connect analytical components and data sources. Kepler helps scientists and analysts create, execute, and share models and analyses. Its built-in components focus on statistical analysis and support task parallelism by using multiple threads on a single machine. YAWL, instead, provides a modeling language for workflows based on the Petri Nets formalism, enriched with constructs to deal with multiple instance patterns. It is based on a concise modeling language, which handles complex data transformations and integration with resources and external applications. The language is supported by a framework that includes an execution engine, a graphical editor and a worklist handler. Finally, Pegasus includes a set of technologies to execute workflow-based applications in different environments. The system can manage the execution of a complex application modeled as a visual workflow by mapping it onto available distributed resources, enabling users to express workflows at an abstract level.

Other systems that exploit distributed computing for executing complex workflows are *Swift* [35] and *Taverna* [36]. Specifically, Swift is a parallel scripting language designed to run scientific data analytics workflows across different distributed systems. It provides a functional language based on a C-like syntax and uses an implicit data-driven task parallelism [37]. A workflow is modeled as a set of program invocations with associated input and output files. The runtime allows the parallel execution of Swift scripts taking into account data dependencies and the availability of external resources. Regarding Taverna, it is a workflow management system mostly used in the scientific community, for example for evidence gathering methods involving text or data mining. It is designed to combine distributed Web Services and/or local tools into complex analysis applications, by exploiting a pipeline parallelism. These pipelines can be executed on local desktop machines or through larger infrastructure, such as supercomputers, multi-clusters and Cloud environments.

3.3 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) [38] is a parallel computation model designed as a bridging model between parallel hardware and software. It is defined on the BSP computer, an abstract computing model composed of: *i*) a group of p processors for communication and local asynchronous computation; *ii*) a network for allowing their communication; and *iii*) a synchronization mechanism. In a BSP application, parallel computation is divided into a sequence of supersteps, in each of which processors perform local computation, exchange data and synchronize to the same barrier. Nowadays, the BSP model is one of the most adopted models for executing massive computational tasks on graphs and matrices, deep learning, machine learning, and network algorithms.

In this context, *Apache Hama*[22] is a BSP-based open source framework, designed to solve complex tasks involving matrix- and graph-based computation in small infrastructures. It is used mainly for developing highly iterative graph processing applications (e.g., graph analysis, deep learning, machine learning) exploiting

[22]https://hama.apache.org

the BSP model. Other open source processing frameworks based on the BSP programming model are *BSPLib* [39] and *Apache Giraph*[23]. BSPLib is an easy to use C++ implementation of the BSP threading model. Giraph provides iterative graph computation for developing high scalable applications, relying on Hadoop as resource manager and Netty[24] for communication. Since Giraph runs map-only jobs, supporting data parallelism, it improves performance by eliminating the reduce operations. Giraph is mainly used by academia and small industry to run graph processing applications in small infrastructures.

3.4 Message Passing

There is a large number of applications whose computational structure does not fit the discussed paradigms (i.e., MapReduce, workflow, or BSP), which makes it difficult to express general-structure computations and efficiently exploit the underlying distributed resources [40]. To achieve high performance while ensuring flexibility in expressing the computations required by several applications, the message passing model provides the basic mechanisms for process-to-process communication in distributed computing systems. It is a well-known paradigm used in many programming languages, operating systems, and libraries for supporting data communication in distributed memory systems. In message passing data is moved from the private memory address space of one process to that of another process through basically two operations: *Send(destination, message)* and *Receive(source, message)*. Although the message passing paradigm has been and still is widely used as a general parallel programming model, recently it has been exploited for implementing scalable Big Data applications [40, 41, 42].

MPI (Message Passing Interface) [43] is the de-facto standard message passing interface. It is a general-purpose distributed memory paradigm for parallel programming, which is commonly used for developing iterative parallel applications where nodes require data exchange and synchronization. There are several MPI implementations that are used in many application fields related to high performance computing, such as bioinformatics, biology, physics and weather modeling [44]. MPI provides hundreds of primitives for point-to-point communication, broadcasting, barrier, reduce, and it supports the ability to collect processes in groups and communicate according to a specific tag. Its basic implementation has been extended by researchers to deal with emerging challenges in Big Data analysis. For example, *BDMPI* [40] (Big Data MPI) is a runtime system that enables the efficient out-of-core execution of distributed-memory parallel programs. It leverages the MPI semantic to orchestrate the execution of a large number of processes on distributed computing systems, enabling the development of efficient out-of-core applications avoiding complexities associated with coding multiple levels of blocking. However, traditional MPI all-to-all communication does not scale well in Exascale systems (i.e., highly parallel computing systems capable of at least one exaFLOPS). Hence to solve this issue new MPI releases (like MPI+X) have been proposed to support neighbor collectives for providing sparse "all-to-some" communication patterns that

[23] https://giraph.apache.org
[24] https://netty.io

reduce the data exchange on limited regions of processors [45, 46]. Many alternatives to MPI have been proposed in the literature, aimed at including data locality, raising the level of abstraction, as well as leveraging modern language design features. These alternatives may consist in both parallel programming languages (e.g., UPC [47], Julia [48]), frameworks for large-scale data processing (e.g., Tensorflow[25], Dask[26]), and extensions of existing languages (e.g., COMPSs [49], UPC++ [50]).

3.5 SQL-like

In the last few years, NoSQL (Not Only SQL) databases addressed several issues about storing and managing Big Data compared to relational databases, ensuring horizontal scaling of continuous read/write operations distributed over many servers. In particular, instead of the ACID model (Atomicity, Consistency, Isolation, Durability), NoSQL databases typically follow another alternative model namely BASE (Basic Availability, Soft-state and Eventual consistency), which releases the requirement of consistency after every transaction for supporting the processing of several instances on different servers simultaneously.

Although NoSQL solutions enable to effectively process large volumes of fast-moving data, there are many applications that still need to be ACID-compliant for user security and privacy, such as managing financial transactions or personal data (e.g., health information). Consequently, relational databases including Oracle, MySQL, Microsoft SQL Server, and PostgreSQL are still more widespread than the most popular NoSQL solutions, such as MongoDB, Redis and Cassandra[27]. Moreover, NoSQL databases are often not suitable for data analytics, which led to the development of different MapReduce solutions to query and analyze data in a more productive manner. SQL-like systems try to combine the effectiveness and query capabilities of Hadoop with the ease of use of the SQL-like language, in order to allow the development of simple and efficient data analysis applications. They are widely used to overcome the complexity of writing MapReduce applications in Hadoop, also for simple tasks (e.g., row aggregations, selections, or counts) while maintaining its performance in terms of querying times and scalability. Their main application domains are data manipulation (ETL operations), data querying and reporting on large repositories.

In this context, one of the most popular system is *Apache Hive*[28], a data warehouse software built on top of Hadoop for reading, writing, and managing data in large-scale infrastructures. It allows the scalable and fault-tolerant management of a huge amount of data through a declarative SQL-like language, namely *Hive Query Language* (HiveQL). In Hive, each data manipulation query is automatically translated into a MapReduce job, which allows to easily process Big Data without the need of writing complex MapReduce programs.

Apache Pig[29] is another Hadoop-based framework that exploits a SQL-like language for executing data flow applications in large-scale infrastructures. It was originally developed for easing the development of Big Data analysis applications, allowing programmers to develop a data analysis application through a scripting and

[25] https://www.tensorflow.org/
[26] https://dask.org/
[27] https://db-engines.com/en/ranking (accessed December 2021)
[28] https://hive.apache.org
[29] https://pig.apache.org

procedural data flow language, called *Pig Latin*. A script contains a sequence of operations, each of one is defined in a SQL-like syntax, for describing how data must be manipulated and processed. Although Hive and Pig can be used to develop the same type of applications, they have different goals. Pig is meant for programmers with a good SQL background to process large amounts of unstructured data, while Hive is a data warehouse software used to read, write, and manage large amounts of structured data in a distributed manner.

Slightly different is *Apache Impala* [51], a massively-parallel query engine, which runs on small Hadoop data processing environments. Impala provides low latency and high concurrency for analytic queries on Hadoop, but circumvents the MapReduce model to directly access the data through the distributed query engine for offering an RDBMS-like experience. Indeed, it uses SQL as a query language, combining it with the performance of traditional databases and the scalability of Hadoop.

4 System features and programming examples

After we reviewed the main models and systems for Big Data analysis in parallel systems, in this section we discuss programming details and illustrate advanced examples of data analysis and mining applications for some of the frameworks introduced in the previous section. We made a selection of those systems keeping one or two of them for each programming model. In particular, we have chosen Hadoop for MapReduce, Spark and Storm for workflows, Hama for BSP, MPI for message passing, and Hive and Pig for the SQL-like model. These seven systems are quite representative and are widely used all over the world in data analysis both in the research and business sectors.

4.1 Apache Hadoop

Apache Hadoop is widely used to develop batch applications and over the years it has been adopted by most of the leading IT companies, such as Yahoo!, IBM and Amazon. For example, Yahoo! used it for developing ad systems, web search, and scaling tests. However, it is suitable for batch processing only, resulting in inefficiency with highly iterative applications that repeatedly perform operations on the same set of data. This is due to the disk-based processing on the distributed file system when computing intermediate results with the MapReduce model [52]. Nevertheless, the project is supported by a large user community and its diffusion is linked to high support for different programming languages and constant updates and bug fixes by a massive open source community.

Hadoop provides a *low-level of abstraction* because programmers can define an application using APIs that are powerful but not easy to use. In fact, they are close to the computing infrastructure and require a low-level understanding of the system and the execution environment for dealing with issues related to distributed file systems, networked computers and distributed programming [53]. Developing an application by Hadoop requires more lines of code and development effort if compared to systems providing a higher level of abstraction (e.g., Spark, Pig, or Hive), but the code is generally more efficient because it can be fully tuned.

Hadoop is designed for exploiting *data parallelism* during map/reduce steps. In fact, input data is partitioned into chunks and processed by different machines in

parallel. Data chunks are replicated on different nodes, ensuring high fault tolerance along with checkpoint and recovery. However, the partitioning strategy does not guarantee efficiency when it is needed to access a large amount of small files.

In addition to the MapReduce programming model, the Hadoop project includes many other modules, such as:

- Hadoop Distributed File System (HDFS), a distributed file system providing fault tolerance with automatic recovery, portability across heterogeneous and low-cost commodity hardware and operating systems, high-throughput access and data reliability.
- YARN, a framework for cluster resource management and job scheduling.
- Hadoop Common, common utilities that support the other Hadoop modules.

In particular, thanks to the introduction of YARN (Yet Another Resource Negotiator) in 2013, Hadoop turns from a batch processing solution into a reference platform for several other programming systems, such as: Storm for streaming data analysis; Hama for graph analysis; Hive for querying large datasets; HBase[30] for random and real-time read/write access to data in a non-relational model; Oozie[31], for managing Hadoop jobs; Ambari[32] for provisioning, managing, and monitoring Hadoop clusters; ZooKeeper[33] for maintaining configuration information, naming, and providing distributed synchronization and group services; and more.

4.1.1 Programming example

The application example we discuss here shows how Hadoop MapReduce can be exploited for creating an inverted index for a large set of Web documents [54]. An inverted index contains a set of words (index terms), and for each word it specifies the IDs of all the documents that contain it and the number of occurrences in each document. The inverted index data structure is a central component of a search engine indexing system. Figure 2 shows the dataflow and main components (Mapper, Combiner, Reducer) of the proposed application.

The Mapper (or MapTask) parses text lines coming from some input documents and emits a pair $\langle word{:}documentID, numberOfOccurrences \rangle$ for each word they contain, where *documentID* is the identifier of the document and *numberOfOccurences* is set to 1 (see Listing 1). Each word is processed with common steps of Natural Language Processing (NLP), such as punctuation removal, lemmatization, and stemming. In order to handle Objects' serialization in a lighter way, programmers have to use specific types for keys and values. As an example, Hadoop uses *Text* and *IntWritable* instead of String and Integer, respectively, which contain the same information by using a much easier abstraction on top of byte arrays.

```
public class MapTask extends Mapper<Object, Text, Text, IntWritable> {
    private final Text keyContent = new Text();
    private final static IntWritable one = new IntWritable(1);

    @Override
    protected void map(Object key, Text value, Context context)
```

[30] https://hbase.apache.org/
[31] https://oozie.apache.org/
[32] https://ambari.apache.org/
[33] https://zookeeper.apache.org/

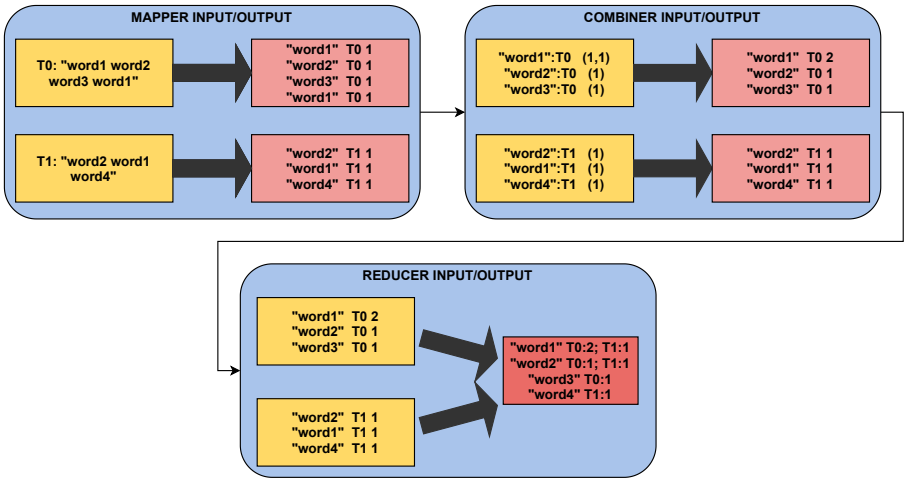

Figure 2 Architecture of the proposed Hadoop application.

```
    throws IOException, InterruptedException {
        // Extract the filename from the current input split
        FileSplit fileSplit = (FileSplit)context.getInputSplit();
        String filename = fileSplit.getPath().getName();
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            // Remove punctuation, apply lemmatization and stemming
            String word = process(itr.nextToken())
            keyContent.set(word + ":" + filename);
            context.write(keyContent, one);
        }
    }
}
```

Listing 1 Inverted Index Mapper.

After word mapping, a combine function is exploited to aggregate intermediate data produced by mappers, before passing them to reducers. As shown in Listing 2, the combiner sums all the occurrences of each word that appear multiple times in a document, and emits a pair $(documentID, sumNumberOfOccurrences)$.

```
public class CombineTask extends Reducer<Text, IntWritable, Text, Text> {
    private final Text sumContent = new Text();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
        context)
    throws IOException, InterruptedException {
        int sum = 0;
        // Sum all the occurrences of a word in the document
        for (Text value : values)
            sum += value.get();
        int split = key.toString().indexOf(":");
        sumContent.set(key.toString().substring(split + 1) + ":" + sum);
        key.set(key.toString().substring(0, split));
        context.write(key, sumContent);
    }
}
```

Listing 2 Inverted Index Combiner.

For each word, the Reducer produces the list of all the documents containing that word and the number of occurrences in each document. Specifically, as shown in Listing 3, a $\langle word, documentID{:}numberOfOccurrences \rangle$ pair is emitted for each word. The set of all output pairs generated by the reduce function forms the inverted index for the input documents.

```
public class ReduceTask extends Reducer<Text, Text, Text, Text> {
    private final Text result = new Text();

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
        StringBuilder documentList = new StringBuilder();
        for (Text value : values)
            // Concatenate all the documents of a word
            documentList.append(value.toString()).append(";");
        result.set(documentList.toString());
        context.write(key, result);
    }
}
```

Listing 3 Inverted Index Reducer.

Finally, Listing 4 shows the main class used to set up and run the application. A programmer must specify the classes to be used as mapper, combiner, and reducer, the input/output format for such classes, and the data input/output paths.

```
public class InvertedIndexJob extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(this.getClass());
        job.setMapperClass(Map.class);
        job.setCombinerClass(Combine.class);
        job.setReducerClass(Reduce.class);
        // Set the output class of key and value for the mapper
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        // Set the output class of key and value for the reducer
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        // Specify input and output paths
        FileInputFormat.addInputPaths(job, "webPage1,webPage2,...");
        FileOutputFormat.setOutputPath(job, new Path(args[0]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Listing 4 Inverted Index Job.

4.2 Apache Spark

Apache Spark is commonly used to develop in-memory applications, such as interactive query and batch processing. Many powerful and robust libraries are built on top of Spark making it a flexible system for a wide range of applications, such as Spark SQL[34] for dealing with SQL queries, MLlib[35] for scalable machine learning

[34] http://spark.apache.org/sql/
[35] https://spark.apache.org/mllib/

applications, GraphX[36] for graph-parallel computation, and Spark Streaming[37] for streaming analysis. The execution of a generic Spark application on a cluster is driven by a central coordinator (i.e., the main process of the application), which can connect with different cluster managers, such as Apache Mesos[38], YARN, or Spark Standalone (i.e., a cluster manager embedded into the Spark distribution). Ambari can be used for provisioning, managing, and monitoring Spark clusters. Spark does not provide its own distributed storage system, but it has been designed to run on top of several data sources, such as distributed file systems (e.g., HDFS), Cloud object storages (e.g., Amazon S3, OpenStack Swift) and NoSQL databases (e.g., Cassandra). A comprehensive software stack is shown in Figure 3.

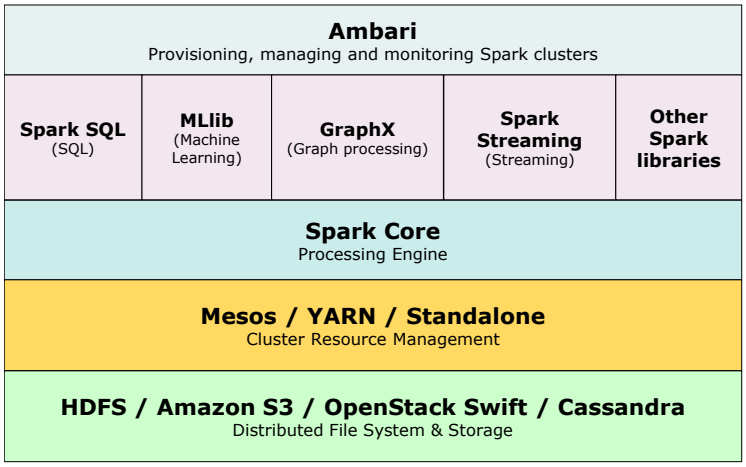

Figure 3 The Spark software stack.

Several big companies use Spark in production to quickly extract insights from data for analysis purposes, such as eBay, Amazon, and Alibaba. For example, eBay uses Big Data and machine learning solutions based on Spark for log aggregation and to provide targeted offers enhancing customer experience. Its user community is very large and its development is constantly expanding. In particular, many efforts are oriented towards the MLlib library, which provides advanced data analytics with parallel machine learning algorithms.

Spark provides a *low-level of abstraction*, in fact programmers must define applications using APIs that are powerful but require advanced programming skills. Compared to Hadoop, developing an application using Spark results in a smaller number of lines of code. In fact, Spark provides some built-in operators (e.g., *filter*, *map*, *reduceByKey*, *groupByKey*) that make easier to code a parallel application exploiting transformations and actions on distributed datasets. Moreover, Spark results up to $100x$ faster than Hadoop [52], thanks to in-memory computing, and easier to use specially when used with the Scala programming language, which provides an object-oriented and functional programming high-level interface. On the other hand, it is more costly compared to Hadoop and presents the same limits when

[36] http://spark.apache.org/graphx/
[37] https://spark.apache.org/streaming/
[38] http://mesos.apache.org/

dealing with large numbers of small files. Even though Spark can be considered a better alternative to Hadoop, in some classes of applications it has limitations that make it complementary to Hadoop. The main one is that to reduce execution time datasets must fit in main memory. In fact, RAM is a critical resource and Spark can suffer from the lack of automatic optimization processes aimed at maximizing in-memory computing while minimizing the probability of data spilling, which is a major cause of performance degradation [55]. A Spark application is defined as a set of independent stages running on a pool of worker nodes and connected in a DAG. A stage is a set of tasks executing the same code on different partitions of input data, thus providing *data parallelism*, as input data is divided into chunks and processed in parallel by different computing nodes. Spark supports *task parallelism* as well when independent stages of the same application are executed in parallel.

Section 4.2.1 proposes an example in which Spark is used for designing a batch application based on the MLlib library. Additionally, despite being a general-purpose framework, Spark also provides a set of libraries specially designed for several tasks, such as structured data analytics, stream processing, and graph computation. For this reason, we provide a pointwise comparison between task-specific libraries of Spark and the corresponding special purpose frameworks. In particular:

- Section 4.3.2 shows how the streaming application implemented in Storm can be expressed with the Spark Streaming library;
- Section 4.4.2 discusses how the BSP-based application proposed in Hama can be modeled with the Spark GraphX library;
- Section 4.7.2 shows how Pig queries can be written with the Spark SQL library.

4.2.1 Programming example

The proposed application shows how to exploit Spark for implementing a customer churn prediction system (i.e., customer switch from a company to another) [56]. In order to identify potential churners and re-engage them, machine learning algorithms can help to mine shared behavioral patterns of those already churned customers and promptly detect current customers at risk of churn. Due to the volume of historical data of both churned customers and existing ones, and the need to periodically analyze new customers and retrain the model, distributed and parallel frameworks such as Spark can be employed with benefits. Figure 4 shows the architecture of the proposed application.

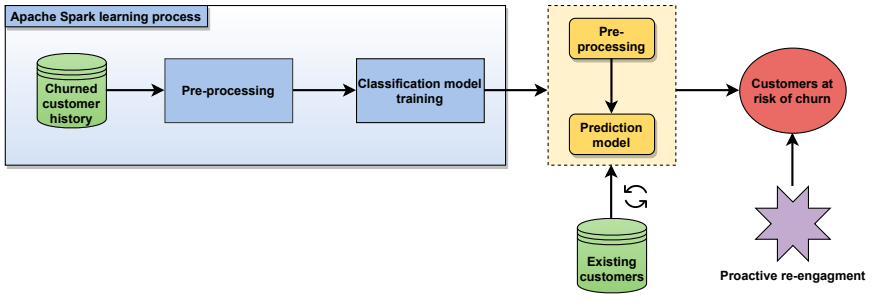

Figure 4 Architecture of the proposed Spark application.

Spark is used for preprocessing historical data and training a prediction model, which will be used to forecast whether a customer will change to another company. Specifically, the MLlib package is employed to handle data organized in RDDs (Resilient Distributed Datasets) and build the classification model. The dataset used for training the prediction model consists of telecommunication customer activity data (e.g., total day minutes, total day calls, customer service calls, etc.), along with a churn label specifying whether a customer has cancelled the subscription. Overall, a generic tuple in the dataset is composed of 20 features. Firstly, clients need to connect to the master node of the Spark cluster via the *spark session*. Spark currently supports authentication for RPC (Remote Procedure Call) channels using a shared secret. In particular, master should be configured to require authentication via the *spark.authenticate* property. However, according to the CVE (Common Vulnerabilities and Exposures) database[39], some versions of Spark (2.4.5 version and earlier) present some security issues when a master in standalone deploy mode is accessed remotely. In particular, even without the shared key, a RPC to the master can be specially-crafted for starting an application's resources on the Spark cluster, thus allowing to execute shell commands on the host machine.

Once connected to the master node, data is retrieved from a batch file and uploaded into a RDD, as shown by the Scala code in Listing 5. The objects representing the different users are defined by parsing the RDD. Then, data is cached for performance purposes.

```
// Build and configure the SparkSession
val spark: SparkSession = SparkSession.builder()
                          .config("spark.app.name", "...")
                          .config("spark.master", "spark://master_ip:port")
                          .config("spark.authenticate.secret", "SecretKey")
                          .getOrCreate()
// Read the training set
val textRDD = spark.sparkContext.textFile(filepath)
val header = textRDD.first()
// Filter out the header row and load data into RDD
val data: RDD[User] = textRDD.filter(line => line != header).map(
                                    line => val col = line.split(",")
                                    User(col(0),..., col(19)))

// Cache data for improving performance
data.cache()
```

Listing 5 Loading data into a RDD for model training. The *config* method allows refining the session behaviour using key-value pairs. For instance, it sets up the application name, the master URL for the cluster and the shared key for authentication.

Afterwards, data is processed to be properly used by the machine learning algorithm. A tuple is converted to a *LabeledPoint*, which represents the features (as a local dense vector) and the label of a data point. Categorical features are encoded as numerical to be standardized by removing the mean and scaling to unit variance. The preprocessing phase is shown in Listing 6. The *numericalFeature* function exploits a map to assign a numerical value (1.0 or 0.0) to categorical features.

```
// Define an encoding function from categorical features to numerical by
    exploiting an internal map
```

[39] https://cve.mitre.org/index.html (accessed December 2021)

```
def numericalFeature: String => Double = (categoricalFeature: String) => {
    val encodeMap: Map[String, Double] = Map("Yes"->1.0, "No"->0.0,
                                             "True"->1.0, "False"->0.0)
    encodeMap(categoricalFeature)}

// Convert tuples to labeled points
val points: RDD[LabeledPoint] = data.map(x =>
                                LabeledPoint(numericalFeature(x.Churn),
                                Vectors.dense(
                                Array(x.AreaCode, x.CustomerServiceCalls,
                                x.TotalIntlCalls, x.TotalDayCharge,
                                x.TotalDayMinutes, x.NumberMessages,
                                x.TotalNightCharge, x.TotalDayCalls,
                                x.TotalNightCalls, x.TotalNightMinutes,
                                numericalFeature(x.IntPlan), ...))))

// Numerical features are scaled using a standard scaler
val scaler = new StandardScaler(withMean = true, withStd = true).fit(points
             .map(x => x.features))

// Preprocessed data is given in output
val preprocessedData = points.map(x => LabeledPoint(x.label, scaler.
                       transform(Vectors.dense(x.features.toArray))))
```

Listing 6 Preprocessing data.

After preprocessing, data is partitioned into two sets used for training a decision tree model and evaluating its performance. The decision tree is configured with three main hyper-parameters: *i*) the impurity measure used for computing the split information gain; *ii*) the maximum depth for terminating the algorithm; and *iii*) the maximum number of bins used when discretizing continuous features. The training phase is carried out by invoking the *trainClassifier* method that outputs the trained model, saved to disk for the classification of unclassified customers. Subsequently, the test set is used to evaluate the model against unseen examples by computing the test error and binary metrics such as precision and recall. Empty *categoricalFeatures* indicates that all features are continuous after the scaling. The complete code is described in Listing 7.

```
// Setup the parameters of the decision tree for the training phase
val splits = preprocessedData.randomSplit(Array(0.7, 0.3))
val (trainingSet, testSet) = (splits(0), splits(1))
val numClasses = 2
val categoricalFeatures = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32
val model = DecisionTree.trainClassifier(trainingSet, numClasses, impurity,
                                         maxDepth, categoricalFeatures,
                                         maxBins)
model.save(sparkContext, "DTModel")
// Evaluate the model on test set
val predictions = testSet.map{case LabeledPoint(trueLabel, features)
                              => val prediction = model.predict(features)
                                 (prediction, trueLabel)}
val testErr = predictions.filter(r => r._1 != r._2).count()/testSet.count()
val metrics = new BinaryClassificationMetrics(predictions)
val precision = metrics.precisionByThreshold
val recall = metrics.recallByThreshold
```

Listing 7 Training and evaluation of the classification model.

It is worth noticing that in a real-world scenario, in which the set of customers is continuously evolving, it may be required to retrain the classification model in order to discover up-to-date churning patterns. For this purpose, streaming machine learning algorithms provided by Spark MLlib can be used, aimed at incrementally updating the model as new data arrives. However, not all the models currently support incremental learning, and in this case the only solution is to add the newly generated examples to the training set and retrain the model from scratch.

The classification model can be exploited to periodically monitor current customers to find potential churners and react appropriately. The system integrates structured data from a data warehouse, such as Apache Hive, a feature extraction module and the trained model to infer new churning customers, as shown in Listing 8. Queries to Hive warehouse are expressed in HiveQL based on the Spark SQL library (see Section 4.6).

```
// Load data of unclassified customers from a data warehouse
val warehouseLocation = new File("spark-warehouse").getAbsolutePath
val spark: SparkSession = SparkSession.builder()
                          .config("spark.app.name", "ChurnPrediction")
                          .config("spark.sql.warehouse.dir",
                                   warehouseLocation)
                          .enableHiveSupport().getOrCreate()
val unclassCustomers: RDD[User] = sql("SELECT * FROM Users").rdd
                                  .map(x => x(0).toString.split(","))
                                  .map(col => User(col(0), col(1), ...,
                                       col(19)))

// Load the trained model from disk
val model = DecisionTreeModel.load(spark.sparkContext, "DTModel")
// Process real data as for training
...
// Run model on real instances and find out if a customer will churn or not
val predictions = unclassCustomers.map(x => model.predict(x.features))
```

Listing 8 Classification of new customers from Hive data warehouse.

4.3 Apache Storm

Apache Storm is widely used for real-time analytics by big companies such as Twitter, Groupon, and Spotify. For example, Twitter developers use Storm for processing many terabytes of data flows a day, for filtering and aggregating contents or for applying machine learning algorithms on data streams. Its user community is relatively small but, thanks to its user-friendly features and flexibility, Storm can be adopted by medium companies for business purposes (e.g., real-time customer services, security analytics, and threat detection). Other typical use cases of Storm are online machine learning, continuous computation, and distributed RPC.

The programming paradigm offered by Storm is based on four abstractions:

- *Stream*: it represents an unbounded sequence of tuples, which is created or processed in parallel. Streams can be created using standard serializers (e.g., integer, doubles) or with custom ones;

- *Spout*: it is the data source of a stream. Data is read from different external sources, such as social network APIs, sensor network, queuing systems (e.g., JMS, Kafka[40], Redis[41]), and then it is feeded into the application;
- *Bolt*: it represents the processing entity. Specifically, it can execute any type of tasks or algorithms (e.g., data cleaning, functions, joins, queries);
- *Topology*: it represents a job. A generic topology is configured as a DAG, where spouts and bolts represent the graph vertices and streams act as their edges. It may run forever until it is stopped.

Storm adopts by default a stateless processing semantic "at least once", which ensures all the messages will be processed, but some of them may be processed more than once (e.g., in case of system failure). This does not guarantee a message ordering and if programmers need to implement a stateful operation they could use the *Trident* library, which provides a "only one" processing semantic.

Storm provides a *medium-level of abstraction* as programmers can easily define an application by using spouts, streams, bolts, and topologies. The Storm APIs allow developers to test an application in local-mode, without having to run it on a cluster. Storm is written mainly in Clojure, but different programming languages are supported through the Multi-Language Protocol that allows to implement bolts and spouts with other languages such as Java and Python. It supports *data parallelism* when many threads execute in parallel the same code on different chunks, *task parallelism* when different spouts and bolts run in parallel, and *pipeline parallelism* for data stream propagation in a topology.

4.3.1 Programming example

The proposed application shows how to leverage Storm for implementing a network intrusion detection system. Network intrusion detection is a critical part of network management for security and quality of service. These systems allow early detection of network intrusion and malicious activities through anomaly detection based techniques (AIDS, Anomaly-based Intrusion Detection System) [57].

A Storm application requires defining three entities: spouts, bolts and the topology. The proposed topology, composed of one spout and two bolts, is given below:

- *ConnectionSpout* is the only data source. This spout streams connections coming from a firewall or stored in a log file, and each record is forwarded as a tuple to the next bolt. In this example, a connection is described by 41 features, some of which are duration, protocol type, service, etc.;
- *DataPreprocessingBolt* receives the tuples from the spout and performs preprocessing. Specifically, it converts the categorical features to numerical and performs standardization for the machine learning model;
- *ModelBolt* performs the classification through a Support Vector Machine (SVM) model trained offline and stores the results to a file for further analysis.

The training phase is performed offline using the Python *scikit-learn* library, as Storm does not provide any native machine learning library. As with the churn prediction system discussed in the Spark programming example (see Section 4.2.1),

[40] https://kafka.apache.org/
[41] https://redis.io/

it may be required to periodically retrain the classification model to discover up-to-date patterns of malicious connections. However, the scikit-learn library leveraged in this example does not support online SVM training. To overcome this limitation, the linear SVM can be approximated to a Stochastic Gradient Descent (SGD) classifier, which supports the partial fit option.

All the trained models (i.e., standard scaler for numerical features, label encoder for categorical features and the SVM model) are dumped in files using the *pickle* module. Hence, the Storm Multi-Language protocol can be adopted to use the trained models in a topology implemented in a JVM language. Figure 5 shows the whole architecture of the proposed application.

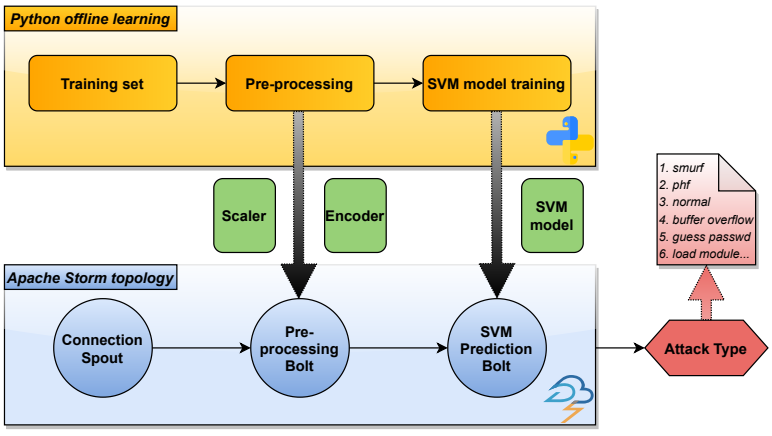

Figure 5 Architecture of the proposed Storm application.

The Storm topology Java code is shown in Listing 9. A topology can be submitted to a production cluster using the storm client, specifying the path of the jar file, the class-name to run, and any other arguments. The shuffle grouping ensures that tuples are randomly distributed so that each bolt receives an equal number of tuples.

```
public class IntrusionTopology {
    public static void main(String[] args) {
        // Build and submit the topology to a cluster
        Config conf = new Config();
        conf.setNumWorkers(20);
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new ConnectionSpout());
        builder.setBolt("process", new DataPreprocessingBolt())
               .shuffleGrouping("spout");
        builder.setBolt("model", new PredictionModelBolt())
               .shuffleGrouping("process");
        StormSubmitter.submitTopology("IntrusionDetection", conf, builder.
            createTopology());
    }
}
```

Listing 9 Storm topology.

The data model used by Storm is the tuple. Each spout node must specify the collector used to emit the tuples (method *open*), how to emit the next tuple (method *nextTuple*) and must declare the output fields for the tuples it emits (method *declareOutputFields*), as shown in Listing 10.

```
public class ConnectionSpout implements IRichSpout {
    private SpoutOutputCollector collector;
    // Define the name of each column in the training data
    private String[] field_names = new String[]{"duration", "protocol_type"
        , "service", ...};

    @Override
    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector) {
        // Define the collector to be used for emitting tuples
        this.collector = collector;
    }

    @Override
    public void nextTuple() {
        // Read from a log file
        while ((str = reader.readLine()) != null) {
            String[] fields = str.split(",");
            // Emit a tuple from input file
            this.collector.emit(new Values(fields[0], ... , fields[40]));
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer dec) {
        // Declare the name of each field of the tuples
        dec.declare(new Fields(field_names[0], ...,  field_names[40]));
    }
}
```

Listing 10 Storm ConnectionSpout.

Each tuple is emitted by the spout and will be processed by the subsequent bolts as declared in the topology. In this case, the class DataPreprocessingBolt (see Listing 11) is a proxy for the Python bolt defined in Listing 12, which processes the tuples by applying the transformations of a set of trained models loaded from the disk (e.g., the encoders for categorical features and the scalers for numerical features). The Multi-Language protocol only requires the bolt specifies the script to execute (see Listing 11), while all the application logic is contained in the Python script (see Listing 12).

```
public class DataPreprocessingBolt extends ShellBolt implements IRichBolt {
    public DataPreprocessingBolt() {
        // Use the Multi-Language protocol to run a Python script
        super("python3", "preprocessingBolt.py");
    }
}
```

Listing 11 Storm DataPreprocessingBolt.

```
class preprocessingBolt(storm.BasicBolt):
    # Load encoders for protocol type, service and flag
    label_enc_prot = pickle.load(open("label_enc_prot", 'rb'))
    label_enc_serv = pickle.load(open("label_enc_serv", 'rb'))
    label_enc_flag = pickle.load(open("label_enc_flag", 'rb'))

    # Load the standard scaler for numerical features
    standard_scaler = pickle.load(open("standard_scaler", 'rb'))
    # Mark nominal, binary and numerical features
    nominal_idx, binary_idx = [1, 2, 3], [6, 11, 13, 14, 20, 21]
```

```
    numerical_idx = list(set(range(41)).difference(nominal_idx).difference(
        binary_idx))

    def process(self, tuple):
        # Encode categorical features
        # protocol type = nominal_idx[0]
        prot_type = label_enc_prot.transform([tuple.values[nominal_idx
            [0]]])[0]
        # service = nominal_idx[1]
        serv = label_enc_serv.transform([tuple.values[nominal_idx[1]]])[0]
        # flag = nominal_idx[2]
        flag = label_enc_flag.transform([tuple.values[nominal_idx[2]]])[0]

        # Scale numerical features
        scaled_features = standard_scaler.transform(
            np.reshape([float(tuple.values[i]) for i in numerical_idx], (1,
                -1)))[0]
        # Emit the tuple after processing
        storm.emit([scaled_features[0], prot_type, serv, flag, ... ])

preprocessingBolt().run()
```

Listing 12 Python DataPreprocessingBolt.

Finally, the ModelBolt in Listing 13 acts similarly to the DataPreprocessingBolt, with the application of the SVM model trained offline using the scikit-learn library. The predicted connection type, from a set of 23 types (e.g., smurf, buffer overflow, guess password, etc.), allows the Network Security infrastructure to react and mitigate possible threats.

```
class modelBolt(storm.BasicBolt):
    # Load the SVM model from disk
    model = pickle.load(open("svm_model", 'rb'))

    def process(self, tuple):
        # Predict the connection type from a set of 23 types
        prediction = model.predict(np.reshape([tuple.values[0],
                     tuple.values[1],..., tuple.values[40]], (1, -1)))[0]

        # Emit the predicted connection type
        storm.emit([prediction])

modelBolt().run()
```

Listing 13 Python ModelBolt.

4.3.2 Comparison with Spark Streaming

Spark Streaming is a library provided in Apache Spark for scalable, high-throughput, and fault-tolerant stream processing. Data can be ingested from many stream services (e.g., Apache Kafka or Amazon Kinesis) or TCP sources, and processed using advanced algorithms (e.g., machine learning and graph processing). Spark Streaming comes up with a high-level abstraction for a continuous stream of data, named DStream (Discretized Stream), which are internally represented as a sequence of RDDs. Each RDD in a DStream collects data from a certain time window and all operations performed on a DStream are forwarded to the underlying RDDs.

Listing 14 shows how the real-time intrusion detection application discussed for Storm can be coded with Spark Streaming. The *StreamingContext* is the entry point for all streaming operations. Through it, a data stream can be ingested and collected into a DStream from text data received on a TCP socket, by specifying the source host name and port. It is worth noting that developers need to explicitly start data ingestion and processing, by invoking the *start()* operation on the streaming context, and wait for processing to stop (either manually or due to an error) via the *awaitTermination()* operation. Furthermore, it is assumed that the model has been trained previously using MLlib (as seen in 4.2.1) and loaded from disk to be used in inference mode during the processing of data streams.

```
val conf = new SparkConf().setMaster("...")
// StreamingContext with a batch interval of 1 second
val ssc = new StreamingContext(conf, Seconds(1))
// Create a DStream that will connect to hostname:port
val data: ReceiverInputDStream[String] = ssc.socketTextStream(hostname,
    port)
// Process data in window of windowLength and slideInterval
val dataInWindow = data.window(Seconds(30), Seconds(10)).map {line =>
        val col = line.split(",")
        ConnectionTest(col(0).toDouble, col(1).toInt, ..., col(40).toInt) }
// Load trained model and scaler
val model = SVMModel.load(ssc.sparkContext, "SVMModel")
val scaler = StandardScalerModel.load("Scaler")
// Apply transformations on each underlying RDD and get predictions
val predictions = dataInWindow.foreachRDD { rdd =>
    rdd.foreach { x =>
        val processed = scaler.transform(Vectors.dense(Array(x.duration, x.
            src_bytes, ..., convert_categorical(x.flag))))
        model.predict(processed)
    }
}

// Start the computation and wait for it to terminate
ssc.start()
ssc.awaitTermination()
```

Listing 14 Network intrusion detection using Spark Streaming.

As it emerges from the code, similarly to Storm, Spark Streaming provides windowed operations that allow to apply transformations on a sliding window of data. Each time the window slides over a source DStream, all associated RDDs that fit into that window are combined and processed to produce a windowed DStream. Any window operation must specify the duration of the window and the interval in which the window operation is performed. On the other hand, some of the main advantages offered by Spark Streaming over Storm are: *i*) the full integration with MLlib, which allows the easy use of a wide range of algorithms for offline learning; *ii*) the native support for streaming machine learning algorithms which can simultaneously learn and predict given a stream of data; *iii*) the support for the Scala programming language, an object-oriented language with scalable functional programming features, which leads to a more compact and readable code.

4.4 Apache Hama

Apache Hama is used mainly for developing iterative graph processing applications (e.g., graph analysis) based on the BSP model, with support for graphics processing units (GPGPU) acceleration. For example, Sogou, a well-known Chinese search

engine, adopts Hama to compute PageRank and determine the relevance or importance of a page [58]. Its user community is very small, so that in April 2020 it moved to Apache Attic, which collects Apache projects that have reached their end of life.

Hama is written in Java and built on the Hadoop Distributed File System (HDFS), thereby being fully compatible with Hadoop clusters. However, it is not limited to HDFS, but can be used with other distributed file systems. The architecture of Hama is based on the master-worker model and consists of three main components:

- *BSP Master*: it is responsible for job scheduling and assignment of tasks to Groom servers. It also has the role of monitoring the superstep sequence in a cluster, checking errors and maintaining the state of Groom servers using heartbeat messages.
- *Groom servers*: a Groom server acts as a worker component in the BSP architecture and is responsible for the execution of the tasks assigned by the BSP Master. It uses heartbeat messages to periodically report important information to the BSP Master, such as its current status and other metrics including available memory on the server and maximum task capacity.
- *ZooKeeper*: it is responsible for the efficient management of BSP peers synchronization, following a mechanism based on blocking barriers. It typically runs together with the BSP Master on the same node.

In the last few years, the growing use of social media platforms has drawn attention of the scientific community to graphs that are abstract data structures very suitable for representing social network contents and connections. In this context, many solutions have been proposed for the efficient computation of massive graphs. Following the BSP model, *Google Pregel* [59] was the first to provide a scalable and general-purpose computing system, aimed at supporting the implementation of algorithms on arbitrary graphs in distributed environments. Inspired by Google Pregel, Hama supports vertex-centric graph computation, allowing the user to program intensive iterative applications with Google Pregel style through a simple programming interface. Specifically, Hama outperforms MapReduce frameworks by avoiding the processing overhead of sorting, shuffling and reducing the vertices. This is due to the message passing interface it provides, thanks to which each BSP superstep is faster than a full MapReduce job execution, especially in the case of highly iterative applications like those involving graph computations [60].

Hama provides a *low-level of abstraction*, because programmers must define an application using low-level BSP primitives for computation and communication. It can be used with any distributed file system in addition to HDFS and it provides explicit support for the message passing interface (MPI). In particular, the BSP model enables Hama to avoid conflicts and deadlines during communication at the largest scale, due to the synchronization mechanism included in the BSP superstep. In contrast, Hama does not provide proper APIs (e.g., for input/output data, for data partitioning) or high-level operators that make it easier to build parallel applications, and the BSP Master represents a single point of failure. Hama supports *data parallelism* by executing the same code in parallel on different portions of data.

4.4.1 Programming example

The proposed application shows how to exploit Hama for addressing the influence maximization problem. Since social media platforms are increasingly used to con-

vey advertising campaigns for products or services, the goal is to identify a set of k users in a social network, namely seeds, that maximizes the spread (i.e., the number of influenced users). The application is based on a bio-inspired technique for numerical optimization, called the Artificial Bee Colony (ABC) [61], which belongs to the field of swarm intelligence. It focuses on the study of self-organized systems, such as ant and bee colonies, flocks of birds and schools of fish, in which a complex action derives from a collective intelligence [62]. The main techniques in this field include Genetic Algorithms, Ant Colony Optimization, Particle Swarm Optimization, Differential Evolution, Artificial Bee Colony, Glowworm Swarm Optimization, and Cuckoo Search Algorithm [63].

Differently from Ant Colony Optimization, which exploits a local search process, the ABC system relies on a global strategy and is composed of three main entities:

- *Food source*, characterized by its goodness in terms of quantity of nectar or distance from the hive;
- *Employer bees*, which collect the nectar and carry details about the source of food to the hive;
- *Unemployer bees*, which are not currently picking up nectar. They can be divided into two categories: *scout bees*, which search for new sources of food and *on-looker bees*, which choose a source of food according to the information brought to the hive by the employer bees.

The main goal of the system is to maximize the nectar collection and it can be adapted to the influence maximization problem as described in [64, 65]. Specifically, each user of a social network is considered as a source of food, employer bees identify the opinion leaders of the network (i.e., final seeds), scout bees are used for exploring the neighborhood of employer bees, and on-looker bees indicate the influenced users.

Figure 6 shows the execution flow of the proposed application, implemented using the Vertex-Centric model provided by Hama for graph computations.

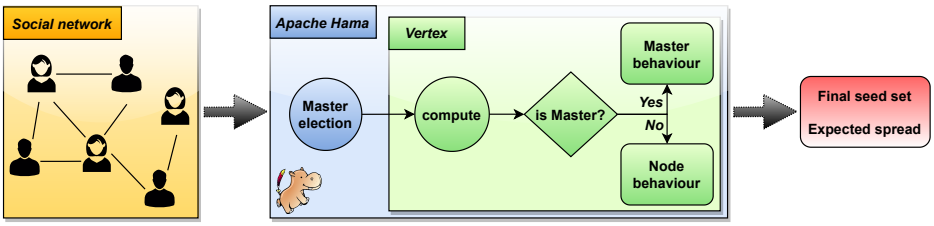

Figure 6 Architecture of the proposed Hama Vertex-Centric application.

The main step involved in writing a Hama graph application is to extend the predefined *Vertex* class, specifying the value types for vertices, edges and messages through its template arguments, as shown in the Java code in Listing 15. The user must encode, by overriding the *compute()* method, the behavior of a vertex, i.e. the set of operations that will be executed by each active node at each superstep. Furthermore, built-in methods such as *sendMessage(Edge* $\langle V, E \rangle$ *e, M msg)* and *getValue()* allow the current vertex to send messages to other vertices or to inspect its associated value.

```
public static class ABC extends Vertex<Text, TextPair, MapWritable> {
    @Override
    public void setup(HamaConfiguration conf) {
        initializeDataStructures();
    }

    @Override
    public void compute(Iterable<MapWritable> mess) throws IOException {
        if(this.getVertexID() != MASTER_ID)
            vertexbehavior(mess);
        else
            masterbehavior(mess);
    }
    ...
}
```

Listing 15 ABC Vertex subclass.

During the *setup* phase the nodes initialize their data structures and one of them is elected as master, taking on the role of coordinator. The compute method separates the behavior of the master from that of the other vertices, by simply checking the id associated with the current vertex. The behavior of the vertices, shown in Listing 16, depends on the type of message they receive. During the first phase each seed sends the rank of its neighbors to the master. Then, when notified by the master, the vertex sends a new message to its neighbors specifying the activation probability. Once the propagation phase is over, i.e. there are no more messages to be processed, the node with maximum influence probability is chosen, sending this value to an aggregator that evaluates the fitness of each seed. Finally, when a vertex receives the stop signal from the master, it votes to halt the execution and suspend itself.

```
private void vertexbehavior(Iterable<MapWritable> messages) throws
    IOException {
    for (MapWritable mex : messages) {
        int messageType = getMessageType(mex);
        if (messageType == UPDATE_DATA) //send data to master
            sendNeighToMaster();
        if (messageType == ACTIVATE) { //the node is activating
            this.getValue().put(new IntWritable(INFL_SOURCE), mex.get(
                INFL_SOURCE));
            this.getValue().put(new IntWritable(DIST), mex.get(DIST));
            MapWritable doneProp = ((MapWritable) this.getValue().get(
                PROP_DONE));
            IntWritable source = ((IntWritable) this.getValue().get(
                INFL_SOURCE));
            if (!doneProp.containsKey(source)) {
                this.getValue().put(new IntWritable(PROB_INDEX),
                                    ((DoubleWritable) mex.get(PROB_INDEX)));
                tryToInfluenceNodes();
                doneProp.put(this.getValue().get(INFL_SOURCE),
                                        new BooleanWritable(true));
                this.getValue().put(new IntWritable(PROP_DONE), doneProp);
            }
        }
    }
    // Vertices receive stop signal from the master and suspend themselves
    MapWritable aggValue = this.getAggregatedValue(0);
    if (this.getSuperstepCount() > 1)
        if (((IntWritable) aggValue.get(new IntWritable(MEX_TYPE))).get()
            == MASTER_STOP)
            voteToHalt();
}
```

Listing 16 Vertex behavior.

The behavior of the master, shown in Listing 17, is described as follows. Once the first iterative phase is over, it elects the scout bees with the highest ranking, notifying the beginning of the influence evaluation. Thus, each scout bee sends an influence message along the outgoing edges, in order to evaluate its fitness. Once the aggregator has completed the evaluation, the master determines whether to proceed with the role switch (*scout* $\rightarrow$ *employer*), communicating it to the other nodes. The process iterates until either the entire set of scout bees is evaluated or convergence is reached (i.e., the minimum percentage increment of the spread between two subsequent iterations is less than a threshold ω). At the end of the process the final result is stored, which consists of the final seed set (i.e., the selected influencers) and the expected spread of influence within the network.

```
private void masterbehavior(Iterable<MapWritable> messages) throws
    IOException {
    if (!((BooleanWritable) this.getValue().get(BOOT_DONE)).get()) {
        // Bootstrap, wait for all seed nodes
        masterBoot(messages);
        Writable[] seedList = ((IntArrayWritable) this.getValue().get(SEEDS
            )).get();
        boolean completed = true;
        for (Writable i : seedList) {
            if (((IntWritable) i).get() == UNSET)
                completed = false;
        }
        if (completed) {
            // Master notifies the beginning of the influence evaluation
            sendStartToSeed();
            this.getValue().put(new IntWritable(BOOT_DONE), new
                BooleanWritable(true));
        }
    } else {
        for (MapWritable mex : messages) {
            int mexType = ((IntWritable) mex.get(MEX_TYPE)).get();
            // Each vertex sends influence data to the master for fitness
                evaluation
            if (mexType == UPDATE_INFLUENCE) {
                this.getValue().put(new IntWritable(PROP_RUNNING), new
                    BooleanWritable(true));
                updatePropagation(mex);
            }
        }
        // Establish if the propagation is ended
        if (((BooleanWritable) this.getValue().get(PROP_RUNNING)).get()) {
            MapWritable agg = this.getAggregatedValue(0);
            // Aggregator has completed the fitness evaluation
            if (((IntWritable) agg.get(MEX_TYPE)).get() == EVAL_END) {
                this.getValue().put(new IntWritable(PROP_RUNNING), new
                    BooleanWritable(false));
                boolean doSwitch = ((BooleanWritable) this.getValue().get(
                    SWITCH)).get();
                if (doSwitch)
                    // Role switch from scout bee to employer bee
                    updateSeedNode();
                if (checkStopCondition()){
                    stopProcess();
                    storeResults();
                }
            }
        }
    }
}
```

Listing 17 Master behavior.

4.4.2 Comparison with Spark GraphX

GraphX is a high-level extension of Spark RDD APIs for graph-parallel computations. It is based on the *Graph* abstraction, which represents a directed multigraph with vertex and edge properties. In addition to basic graph-based queries and algorithms (e.g., subgraph sampling, connected components identification, PageRank, etc.) it provides an optimized version of the Google Pregel graph processing system. In particular, it is specially included for expressing graph-parallel iterative algorithms where the properties of vertices and edges are recomputed iteratively up to a stop condition. Internally, the Pregel operator is a BSP messaging abstraction that performs a series of supersteps in which a vertex receives an aggregation of its in-neighborhood messages from the previous superstep, computes a new value for its property, and finally sends messages to its neighborhood in the next superstep. Differently from Pregel, messages are computed in parallel using a user defined messaging function. Moreover, a vertex can only send messages to its neighborhood and, if it does not receive a message, it is skipped within that superstep.

Listing 18 shows how the ABC algorithm for influence maximization can be modeled using the Pregel API of GraphX. For the sake of brevity, we only reported the message-based communication among vertices, showing the master operations only for the fitness evaluation.

```
def rcvMsg(vertexId: VertexId, nodeStatus: Map[Int, Any], message: (Int,
    Map[Int, Any])): Map[Int, Any] = {
    if (vertexId != MASTER_ID) {
        val messageType = message._1 //an integer constant
        val messageContent = message._2 //a map of properties
        if (messageType == UPDATE_DATA)
            //a neighborhood information update is required to the master
            nodeStatus + (SEND_NEIGH_TO_MASTER -> true)
        if (messageType == ACTIVATE) { //the node is activating
            val infl_source = messageContent.get(INFL_SOURCE)
            val distance = messageContent.get(DIST)
            val doneProp = messageContent.get(PROP_DONE)
            if (!doneProp.contains(infl_source)) {
                val prob = messageContent.get(PROB_INDEX)
                tryToInfluenceNodes(prob, distance)
                doneProp + (infl_source -> true)
            }
            // update internal status of the node to be returned
            nodeStatus + (INFL_SOURCE -> infl_source)
            nodeStatus + (PROP_DONE -> doneProp)
            return nodeStatus
        }
        if (messageType == MASTER_STOP)
            voteToHalt()
    }
    else { // master behavior
        ...
    }
}

def sendMsg(triplet: EdgeTriplet[Map[Int, Any], Double]): Iterator[(
    VertexId, (Int, Map[Int, Any]))] = {
    val sourceNodeStatus = triplet.srcAttr
    if (triplet.srcId != MASTER_ID) {
        if (sourceNodeStatus.get(SEND_NEIGH_TO_MASTER)) {
            // get neighborhood info to be sent to the master
            sourceNodeStatus + (NEIGH_INFO -> getNeighs(triplet.srcId))
            sourceNodeStatus + (SEND_NEIGH_TO_MASTER -> false)
            //identify the role of the destination vertex
            if (triplet.dstId == MASTER_ID)
                sourceNodeStatus + (DEST_TYPE -> "master")
```

```
            else
                sourceNodeStatus + (DEST_TYPE -> "slave")
            return Iterator((triplet.dstId, (UPDATE_INFLUENCE,
                sourceNodeStatus)))
        }
    Iterator.empty
    }
    else { //master behavior
        ...
    }
}

def mergeMsg(message1: (Int, Map[Int, Any]), message2: (Int, Map[Int, Any])
    ): (Int, Map[Int, Any]) = {
    //master receives the propagation results and computes fitness
    if (message1._2.get(DEST_TYPE) == "master" && message2._2.get(DEST_TYPE
        ) == "master") {
        if (message1._2.get(PROP_DONE) && message2._2.get(PROP_DONE))
            compute_fitness(message1._2.get(NEIGH_INFO), message2._2.get(
                NEIGH_INFO))
    }
    ...
}

//create the graph from vertices and edges
val graph = Graph(vertices, edges)
graph.cache() //store to avoid recomputations
val finalGraph = graph.pregel(initialMessage,
                              Int.MaxValue,
                              EdgeDirection.Out)
                             (rcvMsg,
                              sendMsg,
                              mergeMsg)
```

Listing 18 ABC influence maximization using GraphX.

The Pregel operator takes six arguments partitioned in two lists: *i*) the initial message, which triggers the start of the application, the maximum number of iterations, and the edge direction in which to send messages; *ii*) three user defined functions for receiving, computing and merging messages. The output is the final graph where there are no remaining messages to be processed. In the proposed example, a vertex is characterized by a *vertexId*, that is a unique identifier in the whole network, and a *nodeStatus*, modeled as a map, which internally stores information about the vertex, such as its role (i.e., master or slave) and its neighborhood. Next, messages are modeled as the nodeStatus, since the internal properties of the vertices are updated through message passing. Similarly to Hama, this framework provides an efficient implementation of the Pregel model based on the BSP paradigm. However, unlike Hama, GraphX requires developers to provide three user defined functions for specifying how messages have to be exchanged and processed. On the other hand, GraphX provides some advantages, such as in-memory computation, the ability to express efficient queries on graph data using built-in operators, and the compactness of the code deriving from the use of the Scala programming language.

4.5 Message Passing Interface

The Message Passing Interface (MPI), defined since 1992 by a forum composed of many industrial and academic organizations, is widely used by academia and industry in medium-scale infrastructures for developing parallel and distributed

applications. Even if the MPI user community is medium in size, the project engages many contributors.

The first version MPI-1 provided a rich set of messaging primitives, based on a set of eight basic functions that enable it to fully express parallel programs, and other 129 advanced functions. A MPI-1 parallel program is composed of a set of similar processes running on different processors that use MPI functions for message passing. According to the SPMD (*Single Program Multiple Data*) model, MPI is designed for exploiting *data parallelism*, because all the MPI processes that compose a parallel program execute the same code on different data elements. Examples of MPI point-to-point communication primitives are:

- *MPI_Send(msg, leng, type, rank, tag, comm);*
- *MPI_Recv(msg, leng, type, source, tag, comm, status).*

Group communication is implemented by the primitives:

- *MPI_Bcast (inbuf, incnt, intype, root, comm);*
- *MPI_Gather (outbuf, outcnt, outype, inbuf, incnt, intype, root, comm);*
- *MPI_Reduce (inbuf, outbuf, count, type, op, root, comm).*

Other primitives include *MPI_Init* and *MPI_Finalize*, which are used to initialize and terminate a program respectively.

MPI provides a *low-level of abstraction* for developing efficient and portable iterative parallel applications, even if performance may be limited by the communication latency between processors. MPI programmers cannot exploit any high-level construct and must manually cope with complex distributed programming issues, such as data exchange, distribution of data across processors, synchronization, and deadlock. Those issues make it hard to debug an application and does not make the programming task easy on end-user parallel applications where higher level languages are required to simplify the developer task. However, thanks to its portability and efficiency coming from a low-level programming model, MPI is largely used and has been implemented on a very large set of parallel and sequential architectures, such as MPP systems, workstation networks, clusters, and Grids. It is worth mentioning that MPI-1 did not make any provision for process creation, which were introduced later in the MPI-2 [66] version. The current version, MPI-4, provides extensions to better support hybrid programming models and fault tolerance.

4.5.1 Programming example

In this section we present an application for parallel counting characters in a text file by using the OpenMPI[42] implementation. In particular, given an input file of M bytes and N processes, a worker process, with rank not equal to 0, reads a chunk of $\frac{M}{N}$ bytes and counts each character in a private data structure. The master process, with rank equal to 0, receives the partial counts from other $N-1$ processes within the group with the specified tag and aggregates them, as shown in Figure 7.

Listing 19 shows the application code and basic primitives of MPI: *i*) *Init* and *Finalize* for initializing and terminating the program; *ii*) *bcast* to broadcast messages from the master to the workers; and *iii*) *send* and *recv* for point-to-point communication between master and workers. To run the application, the source code must be compiled with *mpijavac* command and executed using the *mpirun -N* command, where N is the number of processes per node on all allocated nodes.

[42] https://www.open-mpi.org/

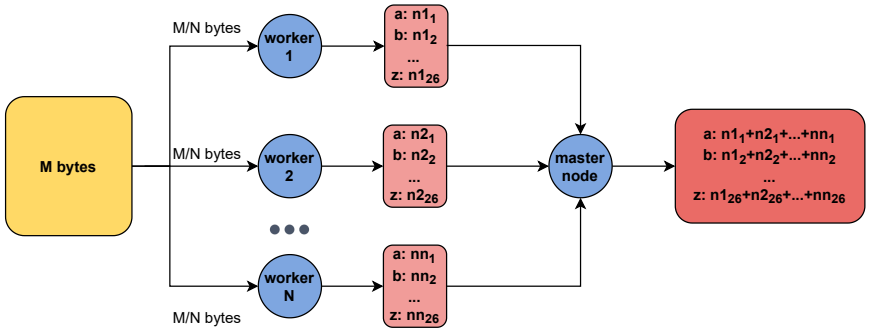

Figure 7 Architecture of the proposed MPI application.

```
static public void main(String[] args) throws MPIException, IOException {
    int tag = 42;
    // Initialize the message array for workers and result array for master
    int[] partial_counter = new int[26];
    int[] res = new int[26];
    int[] split_size = new int[1];
    String fileName = args[0];
    MPI.Init(args);
    Comm comm = MPI.COMM_WORLD;
    int rank = comm.getRank();
    int master_rank = 0;
    int ntasks = comm.getSize();
    if (rank == master_rank){ // The master computes the split size
        Path path = Paths.get(fileName);
        long bytes = Files.size(path);
        split_size[0] = (int) (bytes / ntasks);
    }
    // The split size is broadcast to all worker nodes
    comm.bcast(split_size, split_size.length, MPI.INT, 0);
    int size = split_size[0];
    byte[] readBytes = new byte[size];
    try (InputStream inputStream = new FileInputStream(fileName)) {
        // Each worker node determines the chunk in which to work
        int start = rank * size;
        inputStream.skip(start);
        inputStream.read(readBytes, 0, size);
        // Count and store all the characters of the chunk
        for (byte b : readBytes) {
            char c = (char) b;
            if (Character.isLetter(c)){
                int index = (int) Character.toLowerCase(c) - (int) 'a';
                partial_counter[index] += 1;
            }
        }
        // Each worker node communicates the partial counter to the master
        comm.send(partial_counter, partial_counter.length, MPI.INT,
            master_rank, tag);
    }
    if (rank == master_rank) { // The master aggregates the partial results
        for (int i = 0; i < ntasks; i++) {
            Status status = comm.recv(partial_counter, partial_counter.
                length, MPI.INT, MPI.ANY_SOURCE, tag);
            for (int j = 0; j < partial_counter.length; j++)
                res[j] += partial_counter[j];
        }
    }
    MPI.Finalize();
}
```

Listing 19 Character count.

When the program starts, only the master process is executed. After the *MPI_Init* primitive within the master process, $N-1$ additional processes (i.e., workers) are created to reach the number of parallel processes N indicated in the *mpirun* command. To identify a process, MPI uses an integer ID, called *rank*, for each process, which is 0 for the master and is incremented each time a new process is created. In this way, the master can check the condition $rank == master\_rank$ to perform two operations: *i*) establish the split size of a chunk for each worker; and *ii*) aggregate the partial character counts received by the workers. Communication is handled by the default communicator (i.e., *MPI.COMM_WORLD*), which groups all the processes to enable message exchange. Then, each process, including the master, continues to run distinct versions of the program. In particular, after receiving the split size broadcast by the master, the workers read the assigned data chunk, count the occurrences of each character, and store the result in a private structure (*partial_counter*). Finally, each worker sends the partial counter results to the master, in order to compute the final result.

4.6 Apache Hive

Apache Hive is commonly used by data analysts for *data querying* and *reporting* on large datasets and is adopted by several big companies such as Facebook, Netflix, Yahoo!, and Airbnb. For example, Netflix uses Hive for ad hoc queries and analytics.

The architecture of Hive is comprised of the following main components:

- *User interface*: allowing users to interact with HDFS via a web UI or a CLI;
- *Metastore*: it uses a relational database for storing the metadata of persistent relational entities and how they are mapped to HDFS;
- *HiveQL process engine*: it handles the communication with the Hive Metastore. HiveQL queries get converted into MapReduce jobs;
- *Execution engine*: it is the bridge component between the HiveQL process engine and MapReduce. It executes the MapReduce jobs resulting from the translation of HiveQL queries;
- *HDFS*: it is the underlying distributed file system used for data storage.

Hive provides a *high-level of abstraction*, because a programmer can develop a data processing application by using HiveQL, which relies on traditional concepts of relational databases (e.g., table, row, column). In addition, Hive provides many User Defined Functions (UDF) for data manipulation (e.g., *sum*, *average*, *explode*) and makes it really easy to write custom ones in different languages. For these reasons, Hive is supported by a large user community. However, it is designed only for Online Analytical Processing (OLAP) and not for Online Transaction Processing (OLTP), and does not provide real-time access to data unlike SQL Server.

Hive supports *data parallelism* which allows the execution of the same query on different portions of data. When many complex queries run in parallel, each query can be composed of several jobs, which could starve computational resources. To cope with this issue Hive is powered by Cost-Based Optimizer (CBO), which performs further optimizations by making a series of decisions based on the cost of queries (e.g., join order, type of join, number of parallel queries to run).

4.6.1 Programming example

The proposed application shows how to implement a RoI Mining application with Hive. The widespread use of social media and location-based services makes it possible to extract very useful information for understanding the behavior of large groups of people. Every day millions of people log into social media and share information about the places they visit. The analysis of geo-referenced data produced by users on social media is useful for determining whether users have visited interesting places (e.g., tourist attractions, shopping centres, squares, parks), often called Places-of-Interest (PoIs). Since a PoI is generally identified by the geographic coordinates of a single point, it is useful to define a Region-of-Interest (RoI), an area represented by the geographic boundaries of the PoI. RoI Mining techniques are aimed precisely at the discovery of regions of interest [67].

In this application data is collected from Flickr, a social network used for sharing photos. The initial goal is to assign a generic geo-localized Flickr post to the corresponding PoI, through an analysis of the textual content and metadata of the post (tags and description). After assigning each post to its PoI, the geographical coordinates ⟨*longitude*,*latitude*⟩ are aggregated through *DBSCAN* (Density-Based Spatial Clustering of Applications with Noise), a density-based clustering algorithm that exploits information on the density of points to identify clusters that are representative of a RoI. The cluster with the largest size represents the most significant subregion of the PoI to which it belongs since it is characterized by a greater density of the 2D points identified by the pair of geographical coordinates. Figure 8 shows the overall structure of the application.

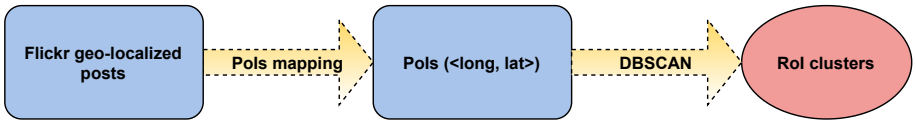

Figure 8 Structure of the proposed Hive application.

Flickr data is firstly filtered to select only a few fields for analysis: the *latitude* and *longitude* contain information related to geographical coordinates; *description* contains a description of the post; *dateposted* contains the date it was posted; *username* contains the id of the user who shared the post and *tags* is a string that contains the set of tags, separated by commas, which give additional information. Listing 20 shows how data is loaded into a Hive table.

```
LOAD DATA INPATH 'filepath' INTO TABLE data;
```

Listing 20 Hive table.

The first step is to map a Flickr tuple to the corresponding PoI. It can be done by defining an UDF that allows to run custom Java code within a Hive script; in particular, a Regular UDF works on a single row of a table and produces a single output row. The *assignRoI* method of Listing 21 checks if the tags and the description contain a keyword of the file loaded as input and returns the name of the RoI (e.g., the Colosseum is also called the Flavian Amphitheater, Coliseo, etc.).

```
public class GeoData extends UDF {
    HashMap<String, HashSet<String>> keyWords = null;
    // The file is stored on HDFS
    String fileName = "hdfs://hostname:port/KeywordFiles//";

    public GeoData() {
        super();
        keyWords = loadKeyWord(fileName);
    }

    public Text evaluate(String tags, String description) {
        // Assign the name of a RoI by comparing tags and description with
            keywords
        String roiName = assignRoI(tags, description);
        if (roiName != null)
            return new Text(roiName);
        return null;
    }
}
```

Listing 21 UDF GeoData.

The function can be called in a Hive query as shown in Listing 22, for determining a ranking of the most visited RoIs. In particular, the GeoData function in the *select* clause returns the name of a RoI for each point and the number of users who have visited that area. The rows selected from the *data* table are grouped by the RoI's name (*group by* clause) and sorted by the number of visitors (*order by* clause) in descending order.

```
-- Call the Java GeoData method and return the most visited RoIs
SELECT GeoData(tags, description) AS PoI, COUNT(username) AS tot
FROM data
WHERE tags IS NOT NULL AND description IS NOT NULL
GROUP BY GeoData(tags, description)
ORDER BY tot DESC;
```

Listing 22 Hive Query GeoData.

At this point data is ready for clustering analysis. To launch the DBSCAN algorithm it is required the implementation of an UDAF (User Defined Aggregate Function), which applies a function to multiple rows of a table by implementing five methods:

- *init()*: it initializes the *evaluator*, which actually implements the UDAF logic;
- *iterate()*: it is called whenever there is a new value to aggregate;
- *terminatePartial()*: it is called for the partial aggregation, and returns an object that encapsulates the state of the aggregation;
- *merge()*: it is called to combine a partial aggregation with another;
- *terminate()*: it is called when the final result of the aggregation is required.

In particular, DBSCAN is launched on the points belonging to the same RoI, where the name of a RoI is obtained using the previously defined GeoData function, as shown in Listing 23. Since DBSCAN should find more than one cluster, the one containing the highest number of points is chosen and returned as a KML (Keyhole Markup Language) string.

```
public class DbscanUDAF extends UDAF {
    // Define an UDAF to run the DBSCAN clustering algorithm
    public static class DBSCANUDAFEvaluator implements UDAFEvaluator {
        LinkedList<ClusterPoint> RoIPts = null;

        public DBSCANUDAFEvaluator() {
            super();
            init();
        }

        @Override
        public void init() {
            // The points of a RoI form a cluster
            RoIPts = new LinkedList<ClusterPoint>();
        }

        public boolean iterate(double latitude, double longitude) throws
            HiveException {
            // Build a point of a cluster using latitude and longitude
            ClusterPoint p = new ClusterPoint(longitude, latitude,
                SpatialContext.GEO);
            RoIPts.add(p);
            return true;
        }

        public LinkedList<ClusterPoint> terminatePartial() {
            return RoIPts;
        }

        public boolean merge(LinkedList<ClusterPoint> otherRoIPts) {
            // Merge intermediate results
            if (otherRoIPts == null) return true;
            RoIPts.addAll(otherRoIPts);
            return true;
        }

        public Text terminate() throws IOException {
            List<Double> dists = MainDBSCAN.calculateKNN(RoIPts, minPts);
            double eps = KDistanceCalculator.calculateEps(dists);
            DBSCANRoI<ClusterPoint> dbscan = new DBSCANRoI<>(eps, 2);
            List<Cluster<ClusterPoint>> clusters = dbscan.cluster(RoIPts);
            // Return the cluster with the highest support as a KML string
            String s = getMaxCluster(clusters);
            return new Text(s);
        }
    }
}
```

Listing 23 Java UDAF DBSCAN.

Finally, the UDAF can be called in the Hive script to get the final results. The HiveQL query is detailed in Listing 24.

```
-- Invoke the Java DbscanUDAF and group rows based on the name of RoIs
SELECT DbscanUDAF(latitude, longitude) AS RoI
FROM data
WHERE latitude IS NOT NULL AND longitude IS NOT NULL
GROUP BY GeoData(tags, description);
```

Listing 24 Hive query DBSCAN.

4.7 Apache Pig

Apache Pig is commonly used for developing *data querying*, simple *data analysis* and ETL (Extract, Transform, Load) applications, gathering data from several sources

such as streams, HDFS, or files. Companies and organizations using Pig in production include LinkedIn, PayPal, and Mendeley. For example LinkedIn, the largest professional online social network, uses the Hadoop ecosystem with Pig, though native MapReduce is sometimes used for performance reasons. Thanks to the Pig Latin scripting language, Pig provides a *medium-level of abstraction*. Compared to other systems, such as Hadoop, Pig developers are not required to write complex and lengthy codes. For this reason, Pig is adopted to ease the development process for several goals, such as link prediction, ad targeting and job recommendations [68].

Pig Latin statements are based on relations, which are similar to tables in a relational database and can be defined as follows: a relation is a bag; a bag is a collection of tuples; a tuple is an ordered set of fields; a field is a piece of data.

Tuples in a bag correspond to the rows in a table, although unlike a relational table Pig relations do not require that each tuple contains the same number of fields. A Pig script can use a large set of operators to ease the development of common tasks on data, such as load, filter, join, and sort. Pig scripts can be invoked by applications written in many other programming languages (actually Java, Python, and JavaScript) and it can exploit custom User Defined Functions (UDFs) for advanced analytics. Each Pig script is translated into a set of MapReduce jobs that are automatically optimized by the Pig engine by using several built-in optimization rules, such as reducing unused statements or applying filters during data loading. In addition, it exploits a multi-query execution system to process an entire script or a batch of statements at once.

Pig supports both *data parallelism*, which is exploited by partitioning data in chunks and processing them in parallel, and *task parallelism* when multiple queries run in parallel on the same data.

4.7.1 Programming example

The Pig application discussed here shows how to use Pig for implementing a dictionary-based sentiment analyzer. Since Pig does not provide a built-in library for sentiment analysis, the system exploits external dictionaries to associate words to their sentiments and determine the semantic orientation of the opinion words [69]. Given a dictionary of words associated with positive or negative sentiment, the sentiment of a text (e.g., sentence, review, tweet, comment) is calculated by summing the scores of positive and negative words in the text and then by calculating the average rating, as shown in Figure 9.

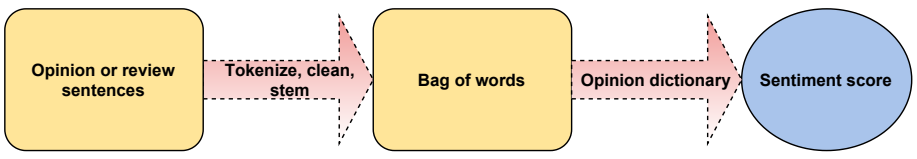

Figure 9 Structure of the proposed Pig application.

As in Hive, developers can include advanced analytics in a script by defining UDFs. For example, the *PROCESS* UDF in Listing 25 is aimed at processing a tuple by removing punctuation as a preprocessing step. Other functionalities, if required, can be added to the *exec* method, which is implemented in Java.

```
public class Processing extends EvalFunc<String> {
    @Override
    public String exec(Tuple tuple) throws IOException {
        if (tuple == null || tuple.size() == 0 || tuple.get(0) == null)
            return null;
        String str = (String) tuple.get(0);
        // Remove punctuation and, if required, apply lemmatization,
            stemming and other
        String clean = str.toLowerCase().replaceAll("\\p{Punct}", "");
        ...
        return clean;
    }
}
```

Listing 25 Java UDF processing.

Listing 26 shows the code in PigLatin of the sentiment analyzer. Once registering the UDF defined in Java, the dataset containing the reviews is loaded from HDFS. Each review is tokenized and processed according to the function defined in Listing 25. Then, a score from a sentiment dictionary is assigned to each token and the final rating of a review is calculated as the average of the scores of its tokens.

```
REGISTER PigUDF.jar;
DEFINE PROCESS main.Processing;
-- Load data from HDFS
reviews = LOAD 'hdfs://hostname:port/pigdata/reviews.csv' USING PigStorage
          ('\t') AS (id:int, text:chararray);
-- Load the dictionary of word sentiment
dictionary = LOAD 'hdfs://hostname:port/pigdata/dictionary.txt' USING
             PigStorage('\t') AS (word:chararray,rating:int);
-- Tokenize and process the text of each review
words = FOREACH reviews GENERATE id,text, FLATTEN(TOKENIZE(PROCESS(text)))
        AS word;
-- Join each word with the dictionary and assign a score sentiment
matching = JOIN words BY word LEFT OUTER, dictionary BY word;
matches_rating = FOREACH matching GENERATE words::id AS id, words::text AS
                 text, dictionary::rating AS rate;
-- Group and compute the average rating for a review
group_rating = GROUP matches_rating BY(id,text);
avg_ratings = FOREACH group_rating GENERATE group,AVG($1.$2) AS rate;
-- Store the results
STORE avg_ratings INTO 'ratings' USING PigStorage(',','-schema');
```

Listing 26 Pig sentiment analyzer.

4.7.2 Comparison with Spark SQL

Spark SQL is a module for structured data management and processing. It differs from the RDD API as it extends Spark with optimizations based on information about the structure of data and the computations performed. Developers can interact with Spark SQL via SQL statements and the Dataset API, using the same execution engine. In addition to SQL query execution, Spark SQL can also be used to read data from an existing Hive environment, as discussed in the Spark programming example (see Section 4.2.1). Spark SQL provides two high-level abstractions, namely *Dataset* and *DataFrame*. A Dataset is a distributed collection of data, which offers the benefits of RDDs, such as resilience and support for lambda

functions, along with a set of optimizations performed by the Spark SQL execution engine. A DataFrame is a Dataset structured into named columns, such as tables in a relational database, but with further optimizations. DataFrames can be created from structured files, Hive tables, databases, or existing RDDs. The data types are automatically inferred, but developers can provide an explicit schema via a *StructType*, matching the structure of the DataFrame in named columns. Spark SQL supports the vast majority of Hive features, such as UDFs and DataFrame operations. Listing 27 shows how the sentiment analysis application discussed in the previous section can be implemented with the Spark SQL Dataset API.

```
// Define the schema for input data
val reviewsSchema = StructType(Array(
    StructField("id", IntegerType, true),
    StructField("text", StringType, true)))

val dictSchema = StructType(Array(
    StructField("word", StringType, true),
    StructField("rating", IntegerType, true)))

val spark = SparkSession.builder().config("spark.app.name", "...")
                                  .config("spark.master", "spark://
                                      master_ip:port").getOrCreate()
// Load data into DataFrames
val reviewsDF: DataFrame = spark.read.schema(reviewsSchema).csv(inputFile)
val dictDF: DataFrame = spark.read.schema(dictSchema).csv(sentimentDict)

// Define user defined functions for text processing
val process = udf((text : String) => text.toLowerCase.replaceAll("[^\\w\\s]
    ", ""))
val tokenize = udf((text : String) => text.split("\\s+"))

// Add a column with processed data
val words = reviewsDF.withColumn("word",explode(tokenize(process(col("text"
    )))))

val matching = words.join(dictDF, words("word") === dictDF("word"),
                          "left_outer").select("id", "text", "rating")

val group_rating: RelationalGroupedDataset = matching.groupBy("id", "text")
val avg_ratings = group_rating.avg("rating")
```

Listing 27 Sentiment analyzer with Spark SQL.

It is worth noting that Spark SQL always uses the same engine when executing a query, regardless of which API is adopted (i.e., SQL or Dataset API). This kind of uniformity brings a major benefit to the developers, which can easily switch between the two APIs, based on which is the most suitable to express a certain transformation. As an example, the first query, dedicated to the tokenization of the processed text, can be easily expressed using SQL statements, as shown in Listing 28. This requires registering the UDFs in the *sqlContext* and creating a temporary or global view of the DataFrames.

```
spark.sqlContext.udf.register("process", process)
spark.sqlContext.udf.register("tokenize", tokenize)
reviewsDF.createOrReplaceTempView("reviews")
spark.sql("SELECT id, text, explode(tokenize(process(reviews.text)))
          as word FROM reviews")
```

Listing 28 Example of a query with SQL statement instead of the Dataset API.

5 Comparative analysis

This section summarizes and classifies the main features of all discussed programming frameworks, their diffusion and the advantages and disadvantages. Some of these systems share features and, in some cases, for programmers choosing one rather than another is an hard choice that can depend on several factors, such as budget (e.g., often high-level services are easy-to-use but more expensive than low-level solutions), type of parallelism, data format, data source, amount of data, performance, and so on. Indeed, given a specific Big Data analysis task, it can be implemented using different programming models and systems.

The comparative analysis carried out in this section may help scholars and developers to choose the best system based on their programming skills, parallel model, budget, application domain, and support provided by the community of both users and developers.

5.1 System features

Table 1 summarizes the features of the frameworks according to their programming model, the type of parallelism, the level of abstraction, the verbosity in writing code and the main classes of applications.

System	Programming model	Type of parallelism	Level of abstraction	Verbosity	Class of applications
Hadoop	MapReduce	Data	Low	High	General-purpose (batch processing)
Spark	Workflow	Data/Task	Low	Low	General-purpose (batch and stream processing, machine learning, graph analysis, structured data analysis)
Storm	Workflow	Data/Task/ Pipeline	Medium	Medium	Stream processing (real-time)
Hama	BSP	Data	Low	Medium	Massive scientific computations (matrix computation, graph analysis, machine and deep learning)
MPI	Message passing	Data	Low	Low	General-purpose (iterative parallel applications)
Hive	SQL-Like	Data	High	Low	Data querying and reporting
Pig	SQL-Like	Data/Task	Medium	Low	Data querying and analysis

Table 1 Features of the systems.

As for the *level of abstraction*, systems have been classified in three categories:

- *Low*: this category includes Hadoop, Spark, MPI, and Hama. Those systems provide powerful APIs and primitives that require distributed programming skills and make the development effort high, but code efficiency is high because it can be fully tuned;
- *Medium*: it includes Storm and Pig. Such systems allow developers to implement parallel and distributed applications using few constructs. They require some programming skills, but the development effort is lower than systems with a low-level of abstraction;
- *High*: this class includes Hive. Those systems require limited programming skills, allowing developers to rapidly build data analytics applications through simple visual interfaces or simple scripts.

About the *type of parallelism*, systems have been classified as follows:

- *Data parallelism*: here we have Hadoop, Spark, Storm, MPI, Hama, Hive, and Pig. Such systems are designed to automatically manage large input data, which is split in chunks and processed in parallel on different computing nodes.
- *Task parallelism*: this form of parallelism is exploited in Spark, Storm and Pig. Such systems allow to run in parallel independent tasks without any data dependency.
- *Pipeline parallelism*: it includes Storm, which allows sending the partial output of a task to the next tasks to be processed in parallel during the stages. Also workflow-based frameworks may exploit some form of pipeline parallelism.

As regards *verbosity*, systems have been classified as follows in the view of the discussed programming examples:

- *High*: Hadoop is included in this category. Those systems require a large number of lines of code and the use of many instructions/calls to build even a simple application. For example, a MapReduce application in Hadoop requires the definition of the mapper, reducer, and job. Writing applications with these systems is complex and lengthy [52];
- *Medium*: it includes Storm and Hama. Such systems require implementing specific interfaces and methods to codify an application. For example, Storm requires to implement the interfaces for spouts and bolts, and to override methods like *nextTuple* and *declareOutputFields*;
- *Low*: it includes frameworks like Spark, MPI, Hive, and Pig. Writing code in those systems is always compact, because programmers are not forced to use specific constructs or when needed it requires a few lines of code. They usually provide an easy to use style of programming (e.g., HiveQL or Pig Latin).

With regard to the *class of applications*, programmers can decide to exploit some *general-purpose systems* like Hadoop, Spark, and MPI or systems that have been developed to be used in specific application domains. For example, Hama has been used for developing *graph processing* applications, Hive and Pig for data querying, Storm for *real-time stream processing* and so on.

5.2 System diffusion

Table 2 summarizes the diffusion and popularity of each system from the user and developer perspectives. Data from Stack Overflow and GitHub has been accessed in December 2021.

As for the *diffusion*, we classified the systems considering the following parameters:

- *Main companies*, which considers the major companies in the IT world that use a given system.
- *User community*, which refers to the diffusion of a system among people who use it. We compared the systems in terms of the number of questions asked on Stack Overflow. In particular, we used the total number of questions as the main indicator of user interest towards the system, and the average number of questions per week for better capturing the latest trends in user adoption. Spark showed to be the most diffused system with a very large user community, followed by Hadoop and Hive, MPI, Pig and Storm. Lastly, Hama registered the fewest number of questions per week.

System	Main companies using it	User community size questions (weekly)	GitHub stars	API support	GitHub commits
Hadoop	Yahoo!, IBM, Amazon	Large - 43.3k (36)	11.8k	Java, C, C++, Ruby, Groovy, Perl, Python	25.1k
Spark	eBay, Amazon, Alibaba	Very Large - 69.5k (193)	30.4k	Scala, Python, Java, R	30.8k
Storm	Twitter, Groupon, Spotify	Small - 2.5k (2)	6.3k	Clojure, Java, Python, Ruby, JavaScript	10.4k
Hama	Samsung Electronics, Korea Telecom, Sogou	Very Small - 22 (<1)	129	Java, Python, C, C++	1.6k
MPI	Amazon WS, AMD, Cisco, Facebook	Medium - 6.3k (13)	1.3k	Java, Fortran, C, C++, Perl, Python	31.8k
Hive	Facebook, Netflix, Yahoo!, AirBnB	Large - 20.2k (44)	3.8k	HiveQL	15.6k
Pig	LinkedIn, PayPal, Mendeley	Small - 5.2k (<2)	631	PigLatin	3.7k

Table 2 Diffusion and popularity of the systems.

- *GitHub commits*, as an indicator of the size of the developer community who contribute to the development and maintenance of code. We referred to the number of commits on official GitHub repositories to grasp the interest of developers in fixing errors/bugs and introducing new features. As seen for the diffusion among users, Spark showed to be one of the frameworks involving most contributors, followed by Hadoop but preceded by MPI that gets the interest of programmers coming from a wide range of applications. Hive, Storm, Pig and Hama follow in order.
- *Github stars*, as an indicator of *popularity* and reusability of the systems. In fact, there exists a strong relationship between system popularity and user-perceived quality of that system and therefore its reuse. In particular, Papamichail et al. [70] showed a strong positive correlation between the number of stars and the number of forks, by analyzing the 100 most popular Java repositories on GitHub. The most popular and reused system is Spark, followed by Hadoop, Storm, Hive, MPI, Pig and Hama.
- *API support*, which indicates the set of programming languages available to develop applications with that system. Systems such as Hadoop, Spark, Storm, and MPI can embrace developers with different programming skills and backgrounds thanks to the wide set of languages and the provision of APIs in popular languages (specifically, Python and Java according to the PYPL index available at `https://pypl.github.io/PYPL.html`, accessed December 2021). Conversely, languages such as Pig or Hive may require learning new languages, such as PigLatin or HiveQL.

5.3 Advantages and disadvantages

Table 3 summarizes the main advantages and disadvantages of the described systems when they are used for programming Big Data analysis applications. The pros and cons are outlined starting from features emerged during the description of each

system and are discussed in the light of the applications and code snippets proposed in Section 4.

System	Advantages	Disadvantages
Hadoop	Fault tolerance, low cost, very large open source community	Verbosity, batch processing only, small files issues, inefficiency with iterative applications
Spark	In-memory computing, ease-of-use, flexibility, libraries for advanced analytics, scalable machine learning support	No automatic optimization process, small files issues, high memory consumption
Storm	Multi-language support, low-latency response time	Message ordering not guaranteed
Hama	Many Distributed FS supported, general-purpose computing on GPUs, conflicts and deadlines avoidance	Single point of failure (BSP Master), low flexibility of partitioning policies, small community
MPI	Efficiency, portability, shared or distributed memory	Hard to debug, bottleneck in network communication
Hive	Large distributed datasets querying, SQL-like language, UDFs for advanced data analysis	Support only for OLAP, real-time data access not supported
Pig	High-level procedural language, UDFs for advanced data analysis, easy learning and development	Small community, hard to tune performance

Table 3 Advantages and disadvantages of the systems.

The advantages of each system are related to the specific features it offers compared to related systems, in terms of functionality, support for different libraries and integration with other frameworks. For example, Spark is the only system that leverages an in-memory computing model, enabling the design of efficient data-intensive applications. Furthermore, as emerged from the comparison with special-purpose frameworks like Hama, Storm and Pig, Spark is highly flexible and can be leveraged in a wide range of application domains, as it provides libraries for stream and graph processing, machine learning, and structured data analysis.

On the other hand, system disadvantages are mainly related to lacks, weaknesses, costs and limitations in the use of a given system. For example, Hadoop suffers when it is used for iterative applications, whereas it is suitable for batch processing. The main disadvantage of using Storm in real-time data stream computations concerns the lack of message ordering. The Hama framework offers a simple programming model also for GPU-based systems, however the BSP Master failure is a critical issue for that system. MPI is generally efficient but it can be hard to debug due to its low-level programming model. Hive is well suited for large distributed data querying, however it does not support OLTP operations. Finally, Pig offers an easy-to-use programming interface for data analysis applications but debugging is complex.

6 Final remarks

In the age of Internet of Things and social media platforms, novel programming models and systems were proposed for collecting and analyzing huge amounts of data in a reasonable time, by leveraging high performance computers and parallel and distributed algorithms. However, the ability to generate and gather data is increasing in a constant and drastic way, which poses a series of challenges to the current solutions aimed at processing, storing and analyzing Big Data. Due to this, current frameworks are expected to be constantly improved for coping with such challenges, allowing the effective extraction of useful knowledge in several application domains. Furthermore, the novel Exascale systems pose new requirements for addressing architectures composed of a very large number of cores. In particular, in the near future, existing frameworks will have to address a wide range of issues related to energy consumption, scheduling, data distribution and access, communication and synchronization, in order to enable the scalable implementation of real Big Data analysis applications [71].

This paper presented a structured analysis and comparison of the most widespread programming models for Big Data analysis and the features of the main software frameworks implementing them. In particular, such systems have been compared according to several criteria concerning three main aspects: their features, diffusion and the advantages/disadvantages of using them. Furthermore, the analysis of each system is carried out with the discussion of a programming example and code snippets to better show the potential and limitations of each one.

The final aim of this work is to support users, designers and developers in identifying and selecting the best solution according to: *i*) their skills in terms of programming capabilities and knowledge of languages; *ii*) hardware availability; *iii*) application domain and purposes; and *iv*) support provided by the software community, concerning both the availability of multi-language APIs, project maintenance on GitHub repository and the availability of solutions for problems in Q&A platforms such as, for example, Stack Overflow or Memory Exceptions.

Ethics approval and consent to participate
Not applicable.

Availability of data and materials
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Funding
Not applicable.

Authors' contributions
All the authors contributed to the structuring of this paper, providing critical feedback and helping shape the research, analysis and manuscript. L. Belcastro. and F. Marozzo conceived of the presented idea and organized the manuscript. A. Orsino and R. Cantini wrote the manuscript with input from all authors and implemented the programming examples. D. Talia and P. Trunfio were involved in planning the work and supervised and reviewed the structure and contents of the paper.

Authors' information
L. Belcastro is a research fellow of computer engineering at the University of Calabria. R. Cantini is PhD student of computer engineering at the University of Calabria. F. Marozzo is an assistant professor of computer engineering at the University of Calabria. A. Orsino is PhD student of computer engineering at the University of Calabria. D. Talia is a professor of computer engineering at the University of Calabria and an adjunct professor at Fuzhou University. P. Trunfio is an associate professor of computer engineering at the University of Calabria.

Acknowledgements
This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Consent for publication
Not applicable.

Author details
[1]University of Calabria, Rende, Italy. [2]Dtok Lab, Rende, Italy.

References

1. Belcastro L, Marozzo F, Talia D, Trunfio P. Big data analysis on clouds. In: Handbook of big data technologies. Springer; 2017. p. 101–142.
2. Marx V. Biology: The big challenges of big data. Nature. 2013;498(7453):255–260.
3. Belcastro L, Marozzo F, Talia D, Trunfio P. Using Scalable Data Mining for Predicting Flight Delays. ACM Transactions on Intelligent Systems and Technology. 2016 October;8(1).
4. Murdoch TB, Detsky AS. The inevitable application of big data to health care. Jama. 2013;309(13):1351–1352.
5. Walker SJ. Big Data: A Revolution That Will Transform How We Live, Work, and Think. International Journal of Advertising. 2014;33(1):181–183.
6. Belcastro L, Marozzo F, Talia D. Programming models and systems for big data analysis. International Journal of Parallel, Emergent and Distributed Systems. 2019;34(6):632–652.
7. Jin X, Wah BW, Cheng X, Wang Y. Significance and challenges of big data research. Big Data Research. 2015;2(2):59–64.
8. Athmaja S, Hanumanthappa M, Kavitha V. A survey of machine learning algorithms for big data analytics. In: 2017 International conference on innovations in information, embedded and communication systems (ICIIECS). IEEE; 2017. p. 1–4.
9. Talia D, Trunfio P, Marozzo F. Data Analysis in the Cloud. Elsevier; 2015. ISBN 978-0-12-802881-0.
10. Chen M, Mao S, Liu Y. Big data: A survey. Mobile networks and applications. 2014;19(2):171–209.
11. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007; 2007. p. 59–72.
12. Oussous A, Benjelloun FZ, Ait Lahcen A, Belfkih S. Big Data technologies: A survey. Journal of King Saud University - Computer and Information Sciences. 2018;30(4):431–448.
13. Hu H, Wen Y, Chua TS, Li X. Toward scalable systems for big data analytics: A technology tutorial. IEEE access. 2014;2:652–687.
14. Low Y, Gonzalez JE, Kyrola A, Bickson D, Guestrin CE, Hellerstein J. Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:14082041. 2014;.
15. Yaqoob I, Hashem IAT, Gani A, Mokhtar S, Ahmed E, Anuar NB, et al. Big data: From beginning to future. International Journal of Information Management. 2016;36(6):1231–1247.
16. Singh D, Reddy CK. A survey on platforms for big data analytics. Journal of big data. 2015;2(1):1–20.
17. Wang J, Yang Y, Wang T, Sherratt RS, Zhang J. Big data service architecture: a survey. Journal of Internet Technology. 2020;21(2):393–405.
18. Rao TR, Mitra P, Bhatt R, Goswami A. The big data system, components, tools, and technologies: a survey. Knowledge and Information Systems. 2019;60(3):1165–1245.
19. Saggi MK, Jain S. A survey towards an integration of big data analytics to big insights for value-creation. Information Processing & Management. 2018;54(5):758–790.
20. Tsai CW, Lai CF, Chao HC, Vasilakos AV. Big data analytics: a survey. Journal of Big data. 2015;2(1):1–32.
21. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM. 2008;51(1):107–113.
22. Marozzo F, Talia D, Trunfio P. P2P-MapReduce: Parallel data processing in dynamic Cloud environments. Journal of Computer and System Sciences. 2012 September;78(5):1382–1402.
23. Talbot J, Yoo RM, Kozyrakis C. Phoenix++ modular mapreduce for shared-memory systems. In: Proceedings of the second international workshop on MapReduce and its applications; 2011. p. 9–16.
24. Rao S, Ramakrishnan R, Silberstein A, Ovsiannikov M, Reeves D. Sailfish: A framework for large scale data processing. In: Proceedings of the Third ACM Symposium on Cloud Computing; 2012. p. 1–14.
25. Talia D, Trunfio P. Service-oriented distributed knowledge discovery. Chapman and Hall/CRC; 2012.
26. Van Der Aalst WMP, Ter Hofstede AHM, Kiepuszewski B, Barros AP. Workflow Patterns. Distrib Parallel Databases. 2003 Jul;14(1):5–51.
27. Talia D. Workflow systems for science: Concepts and tools. International Scholarly Research Notices. 2013;2013.
28. Xin RS, Rosen J, Zaharia M, Franklin MJ, Shenker S, Stoica I. Shark: SQL and Rich Analytics at Scale. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. SIGMOD '13. New York, NY, USA: ACM; 2013. p. 13–24.
29. Marozzo F, Lordan F, Rafanell R, Lezzi D, Talia D, Badia RM. Enabling cloud interoperability with compss. In: European Conference on Parallel Processing. Springer, Berlin, Heidelberg; 2012. p. 16–27.
30. Marozzo F, Talia D, Trunfio P. A Workflow Management System for Scalable Data Mining on Clouds. IEEE Transactions On Services Computing. 2016;.
31. Marozzo F, Talia D, Trunfio P. Scalable script-based data analysis workflows on clouds. In: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science; 2013. p. 124–133.
32. Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, et al. Scientific workflow management and the Kepler system. Concurrency and computation: Practice and experience. 2006;18(10):1039–1065.

33. Van Der Aalst WM, Ter Hofstede AH. YAWL: yet another workflow language. Information systems. 2005;30(4):245–275.
34. Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming. 2005;13(3):219–237.
35. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, Foster I. Swift: A language for distributed parallel scripting. Parallel Computing. 2011;37(9):633–652.
36. Wolstencroft K, Haines R, Fellows D, Williams A, Withers D, Owen S, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic acids research. 2013;41(W1):W557–W561.
37. Wozniak JM, Wilde M, Foster IT. Language features for scalable distributed-memory dataflow computing. In: Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on. IEEE; 2014. p. 50–53.
38. Valiant LG. A Bridging Model for Parallel Computation. Commun ACM. 1990 aug;33(8):103–111.
39. van Duijn M, Visscher K, Visscher P. BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model.;.
40. LaSalle D, Karypis G. Mpi for big data: New tricks for an old dog. Parallel Computing. 2014;40(10):754–767.
41. Reyes-Ortiz JL, Oneto L, Anguita D. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. Procedia Computer Science. 2015;53:121–130.
42. Liang F, Lu X. Accelerating iterative big data computing through MPI. Journal of Computer Science and Technology. 2015;30(2):283.
43. Gropp W, Lusk E, Skjellum A. Using MPI: portable parallel programming with the message-passing interface. vol. 1. MIT press; 1999.
44. Laguna I, Marshall R, Mohror K, Ruefenacht M, Skjellum A, Sultana N. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19. New York, NY, USA: Association for Computing Machinery; 2019. .
45. Talia D, Trunfio P, Marozzo F, Belcastro L, Garcia Blas J, Del Rio D, et al. A Novel Data-Centric Programming Model for Large-Scale Parallel Systems. In: Euro-Par 2019: Parallel Processing Workshops. Lecture Notes in Computer Science. Gottingen, Germany; 2020. p. 452–463. ISBN: 978-3-030-48339-5.
46. Bader DA. Evolving mpi+ x toward exascale. Computer. 2016;49(08):10–10.
47. Consortium U, Bonachea D, Funck G. UPC Language and Library Specifications, Version 1.3. 2013 11;.
48. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM review. 2017;59(1):65–98.
49. Lordan F, Tejedor E, Ejarque J, Rafanell R, Álvarez J, Marozzo F, et al. ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing. 2014;12(1):67–91.
50. Zheng Y, Kamil A, Driscoll MB, Shan H, Yelick K. UPC++: A PGAS Extension for C++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium; 2014. p. 1105–1114.
51. Kornacker M, Behm A, Bittorf V, Bobrovytsky T, Ching C, Choi A, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In: Cidr. vol. 1; 2015. p. 9.
52. Verma A, Mansuri AH, Jain N. Big data management processing with Hadoop MapReduce and spark technology: A comparison. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). IEEE; 2016. p. 1–4.
53. Wadkar S, Siddalingaiah M, Venner J. Pro Apache Hadoop. Apress; 2014.
54. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6. OSDI'04. USA: USENIX Association; 2004. p. 10.
55. Cantini R, Marozzo F, Orsino A, Talia D, Trunfio P. Exploiting Machine Learning for Improving In-Memory Execution of Data-Intensive Workflows on Parallel Machines. Future Internet. 2021;13(5).
56. Balle B, Casas B, Catarineu A, Gavaldà R, Manzano-Macho D. The Architecture of a Churn Prediction System Based on Stream Mining. In: Frontiers in Artificial Intelligence and Applications. vol. 256; 2013. p. 157–166.
57. Khraisat A, Gondal I, Vamplew P, Kamruzzaman J. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecurity. 2019;2(1):1–22.
58. Siddique K, Akhtar Z, Kim Y, Jeong YS, Yoon EJ. Investigating Apache Hama: a bulk synchronous parallel computing framework. The Journal of Supercomputing. 2017;73(9):4190–4205.
59. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, et al. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM; 2010. p. 135–146.
60. Siddique K, Akhtar Z, Yoon EJ, Jeong YS, Dasgupta D, Kim Y. Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications. IEEE Access. 2016;4:8879–8887.
61. Karaboga D. An idea based on honey bee swarm for numerical optimization. Technical report-tr06, Erciyes university, engineering faculty, computer engineering department; 2005.
62. Wang D, Tan D, Liu L. Particle swarm optimization algorithm: an overview. Soft Computing. 2018;22(2):387–408.
63. Ab Wahab MN, Nefti-Meziani S, Atyabi A. A comprehensive review of swarm optimization algorithms. PloS one. 2015;10(5):e0122827.
64. Sankar CP, Kumar KS. Learning from bees: An approach for influence maximization on viral campaigns. PloS one. 2016;11(12):e0168125.
65. Cantini R, Marozzo F, Mazza S, Talia D, Trunfio P. A Weighted Artificial Bee Colony algorithm for influence maximization. Online Social Networks and Media. 2021;26:100167.
66. Geist A, Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Saphir W, et al. MPI-2: Extending the message-passing interface. In: Euro-Par'96 Parallel Processing. Springer; 1996. p. 128–135.
67. Belcastro L, Marozzo F, Talia D, Trunfio P. G-RoI: Automatic Region-of-Interest Detection Driven by

Geotagged Social Media Data. ACM Trans Knowl Discov Data. 2018;12(3).
68. Sumbaly R, Kreps J, Shah S. The big data ecosystem at linkedin. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data; 2013. p. 1125–1134.
69. Kumar A, Sebastian TM. Sentiment analysis on twitter. International Journal of Computer Science Issues (IJCSI). 2012;9(4):372.
70. Papamichail M, Diamantopoulos T, Symeonidis A. User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2016. p. 100–107.
71. Talia D. A view of programming scalable data analysis: from clouds to exascale. Journal of Cloud Computing. 2019;8(1):1–16.