Programming Platforms for Big Data Analysis

Jiannong Cao, Shailey Chawla, Yuqi Wang and Hanqing Wu

Abstract Big data analysis imposes new challenges and requirements on programming support. Programming platforms need to provide new abstractions and run time techniques with key features like scalability, fault tolerance, efficient task distribution, usability and processing speed. In this chapter, we first provide a comprehensive survey of the requirements, give an overview and classify existing big data programming platforms based on different dimensions. Then, we present details of the architecture, methodology and features of major programming platforms like MapReduce, Storm, Spark, Pregel, GraphLab, etc. Last, we compare existing big data platforms, discuss the need for a unifying framework, present our proposed framework MatrixMap, and give a vision about future work.

Keywords Big data analysis · Programming platforms · Unifying framework · Data parallel · Graph parallel · Task parallel · Stream processing

1 Introduction

The necessity of increased computing speed and capacity offered by big data programming platforms has led to constantly evolving system architectures, novel development environments, and multiple third-party software libraries and application packages. Now, we are in an era where businesses, government sectors, small and big organizations have all realized the potential of big data analysis. The great demand

J. Cao (✉) · S. Chawla · Y. Wang · H. Wu
Department of Computing, Hong Kong Polytechnic University, King's Park, Hong Kong
email: csjcao@comp.polyu.edu.hk
URL: http://www4.comp.polyu.edu.hk/~csjcao/

S. Chawla
e-mail: csschawla@comp.polyu.edu.hk

Y. Wang
e-mail: csyqwang@comp.polyu.edu.hk

H. Wu
e-mail: cshwu@comp.polyu.edu.hk

A.Y. Zomaya and S. Sakr (eds.), *Handbook of Big Data Technologies*,
DOI 10.1007/978-3-319-49340-4_3

for big data analysis systems is giving a thrust to the research and development in this area. Large amounts of data have to be handled in a parallel and distributed way wherein, and the computations have to be distributed across many machines in order to be finished in a reasonable amount of time. The issue of how the computation can be parallelized, how data is distributed and how failures are handled in such a wide distribution are compelling, and call for special programming platforms for big data analysis.

In recent years, a lot of programming platforms have emerged for big data analysis. Figure 1 shows the time line of systems that handle large scale data. The timeline clearly indicates the increasing amount of interest in these systems recently.

Big data processing can be done on either distributed clusters or high performance computing machines like Graphical Processing Units [10].

In this section of the chapter, we provide an overview of existing programming platforms for big data analysis, which gives the readers a brief impression on existing big data platforms. The remaining part of the chapter is organized as follows. First, we discuss the special requirements and features of programming platforms for large scale data analysis in the next section. We then present in Sect. 3, a classification schema for big data programming platforms based on different dimensions, which would give insights on types of existing systems and their suitability to different kinds of applications. In Sect. 4, we will introduce the details of major existing programming platforms. The programming platforms are described with respect to their specific purpose, programming model, implementation details and important features. We discuss our unifying framework and our proposed framework called MatrixMap [15], as well as summarize the big data programming platforms according to the essential requirements in Sect. 5. Finally, we conclude this chapter by giving our understanding and vision on programming platforms. The chapter is intended to benefit anyone who is new to big data analysis by presenting details and features of popular big data programming platforms, analysts to choose appropriate program-

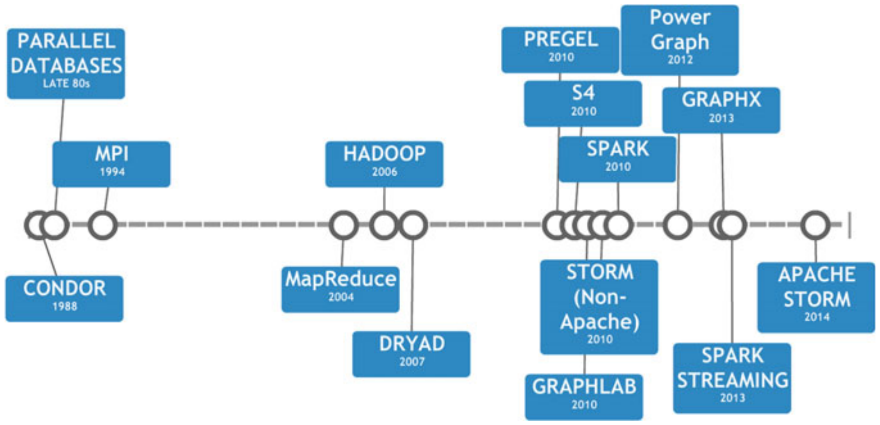


Fig. 1 Timeline of programming platforms for big data analysis

ming platforms for their specific applications by offering comparison across them, and also interested researchers by showing our current research work and vision on future direction.

2 Requirements of Big Data Programming Support

Programming platforms constitute of systems and language environments that can run on commodity, inexpensive hardware or software and can be programmed and operated by programmers and analysts with average, mainstream skills. Big data analysis need to have some essential requirements so as to deal with specific issues related to vast data and large scale computations, they also need to support distributed and local processing (data copies) and support ease of use, data abstraction, data flow and data transformations. In traditional programming platforms, the key feature is performance, but for systems with large scale data, there are many more features essential for smooth functioning of the system and being useful.

Scalability
Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [5]. Scaling can be done by either scaling up the system, which means adding additional resources on a single computer/node to improve the performance or scaling out the system, which refers to addition of more computers/nodes to the system in a distributed software system.

Support Multiple Data Types
Big data systems should be able to support multiple data types, e.g., record, graph or stream. Different common data types have been briefly explained in following text.

- *Record data* can be split into independent elements and thus data can be processed independently. Independent results can be summed up to get the final result.
- *Graph data* cannot be split into independent elements like in the case of record data. Elements may have relations with each other and thus the processing of one element depends on other elements. Graph data not only include real graphs but also other data which can be viewed as graph. The data can also be in form of stream which would require fast processing in memory.
- *Stream data* arrive at a rate that makes it infeasible to store everything in active storage. If it is not processed immediately or stored, then it is lost forever or we lose the opportunity to process them at all. Thus, stream-processing algorithm is executed in main memory, without or with only rare access to secondary storage.

Fault Tolerance
Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. In a distributed framework with large scale data, it is imperative that some nodes

carrying data can fail. For a fault tolerant system, when a server in the cluster fails, a stand-by server is automatically activated to take over the tasks, there are also check pointing and recovery to minimize state loss.

Efficiency
Massive computation capability is required for big data analysis and hence efficiency is very critical when programming platforms are scaled up or scaled out for handling large amounts of data. Efficiency means faster speed with respect to usage of certain resources like memory or number of nodes.

Data I/O Performance
Data I/O performance refers to the rate at which the data is transferred to/from a peripheral device. In the context of big data analytics, this can be viewed as the rate at which the data is read and written to the memory (or disk) or the data transfer rate between the nodes in a cluster. The systems should have low latency to minimize the time taken for reading and writing to the memory, and high throughput for data transmission.

Iterative Task Support
This is the ability of a system to efficiently support iterative tasks. Since many of the data analysis tasks and algorithms are iterative in nature, it is an important metric to compare different platforms, especially in the context of big data analytics. The systems must be suitable for iterative algorithms so that the result of one iteration can be easily used in the next iteration, and all the parameters can be stored locally. Processes can reside and can keep running as long as the machine is running.

The properties described above are very significant for description of programming platforms. In the next section, we propose a classification of programming system based on different dimensions. We have classified the programming platforms based on the processing techniques and data sources.

3 Classification of Programming Platforms

The existing programming platforms for big data analysis have numerous special features as discussed in the previous section. It is important to realize what kinds of systems encompass what features so that it is easier to make a choice of programming system with respect to the application. We classify the existing programming platforms based on different dimensions as that have been described in the following subsections.

3.1 Data Source

Data analysis is done for different kinds of source data. The data can arrive for processing either in batches or continuous stream. Hence based on how the data arrives, various systems can be classified into Batch Processing Systems and Stream Processing Systems. Many big data analysis applications work on batch-wise input, and there are many like twitter or stock markets dealing with multiple data streams. There have been much development in this regard, and specific programming platforms have been fostered to deal with streaming data.

Batch Processing Systems are the systems that execute a series of programs which take a set of data files as input, processes the data and produce a set of output data files. It is termed as batch processing because the data is collected in batches as sets of records and processed as a unit. Output is another batch that can be reused as input if required.

Batch processing systems have existed for very long and they have various advantages. These systems utilize computing resources in an optimum and efficient manner based on the priority of other jobs. Batch processing techniques are likely to avoid system overhead.

Many distributed programming platforms like MapReduce [8], Spark [32], GraphX [12], Pregel [22] and HTCondor [14] are batch processing systems. They analyze large scale data in batches in a distributed and parallel fashion.

Stream Processing Systems are the systems that process continuous input of data. These systems should have faster rate of processing than rate of incoming data. So an input dataset coming at time t needs to be processed before dataset arrives at time $t + 1$. The stream processing systems work under a very strict time constraint. They are important in applications, which need continuous output from incoming data like stock market, twitter etc. Big data programming platforms like Storm [30], Spark Streaming [32] and S4 [24] are used for processing stream data.

3.2 Processing Technique

Programming platforms can also be classified based on the processing techniques. Large scale processing can be done using different techniques like data parallel, task parallel or graph parallel techniques. We have classified the programming platforms according to the techniques they employ for processing data.

Data parallel programming platforms focus on distributed data across parallel computing nodes. In data parallelism, each node executes the same task on different pieces of distributed data. It emphasizes that data is distributed and executed in parallel on different computing nodes, and then the result from different nodes is consolidated and processed further. The data parallel systems tend to be very fault tolerant as they can have redundancy. Also, this kind of arrangement makes processing of large-scale data simpler by breaking down data into smaller units. MapReduce, Spark, Hadoop are data parallel systems and have been very popular in big data programming community. We also proposed MatrixMap to efficiently support matrix computations.

Task parallel platforms are systems that process data in a parallel manner across multiple processors. Task parallelism focuses on distributing execution processes

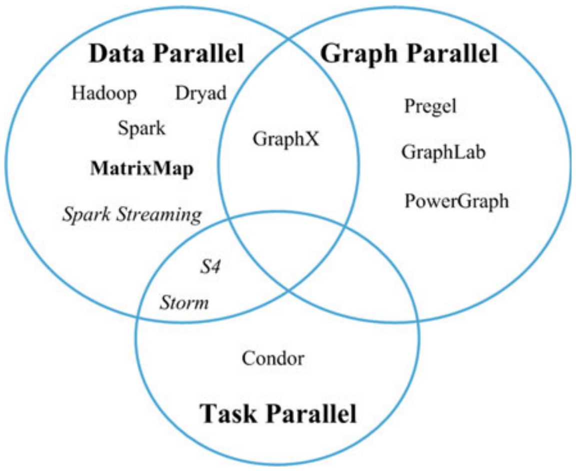


Fig. 2 Classification schema of big data programming platforms (stream processing platforms are mentioned in italics)

across different parallel computing nodes. HTCondor programming system is an example of task parallel system.

Graph parallel platforms are systems that encode computation as vertex programs which run in parallel and interact along edges in the graph. Graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism, and effective partitioning to minimize communication. Formally, a graph-parallel abstraction consists of a sparse graph $G = \{V, E\}$, and a vertex-program Q which is executed in parallel on each vertex v belongs to set V, and can interact (e.g., through shared-state in GraphLab, or messages in Pregel) with neighboring instances $Q\ (u)$ where (u, v) belongs to E. In contrast to more general message passing models, graph-parallel abstractions constrain the interaction of vertex-program to a graph structure enabling the optimization of data-layout and communication [11]. Pregel, Graphlab, GraphX are graph parallel systems popular for social network analysis.

Figure 2 depicts the classification schema of various programming platforms for big data analysis. The Figure presents the classification in the form of a Venn diagram, and the programming platforms are placed according to their matching criterion. The systems in italics are stream processing systems, while the remaining are batch processing systems. In the next section we describe the major existing programming platforms in detail.

4 Major Existing Programming Platforms

In this section we describe in detail some major programming platforms that are prominent in big data analysis. The programming platforms have been described according to the prominent processing techniques used in their programming models.

4.1 Data Parallel Programming Platforms

Data parallel programming platforms are the systems that distribute data over parallel computing nodes [6]. In distributed systems, data parallelism is achieved by dividing the data into a smaller size and each parallel computing node performing the same task over small sized data. The intermediate result is then integrated to achieve the final outcome of processing.

4.1.1 Hadoop

Hadoop is based on MapReduce programming model [8] which is the most popular paradigm for big data analysis till date, and brought a breakthrough in big data programming. In this model, data-parallel computations are executed on clusters of unreliable machines by systems, that automatically provide locality-aware scheduling, fault tolerance, and load balancing. Hadoop MapReduce is an open source form of Google MapReduce.

MapReduce is useful in a wide range of applications, including distributed pattern-based searching, distributed sorting, web link-graph reversal, web access log stats, inverted index construction, document clustering, machine learning, and statistical machine translation. At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web. It has replaced the old ad hoc programs that updated the index and ran various analyses.

The MapReduce abstraction allows expressing simple computations without revealing the complicated details of parallelization. There are two main primitives in this abstraction called the *Map* and *Reduce* operations. The computation is expressed in form of these two functions, wherein it takes a set of input *key/value* pairs and produces a set of output *key/value* pairs.

Map, written by the user, takes an input pair and produces a set of intermediate *key/value* pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the *Reduce* function.

Reduce function, written by user too, accepts intermediate key and a set of values for that key. It merges these values together to form a possibly smaller set of values. This is done in an iterative fashion, so that list of values that are too large can fit in memory. This is the key concept of the MapReduce paradigm that enables it to handle large scale data in an efficient way.

The MapReduce framework transforms a list of (key, value) pairs into a list of values. This behavior is different from the typical functional programming, *Map* and *Reduce* combination, which accepts a list of arbitrary values and returns one single value that combines all the values returned by map. Figure 3 [4] depicts the architecture of MapReduce programming model.

MapReduce framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a

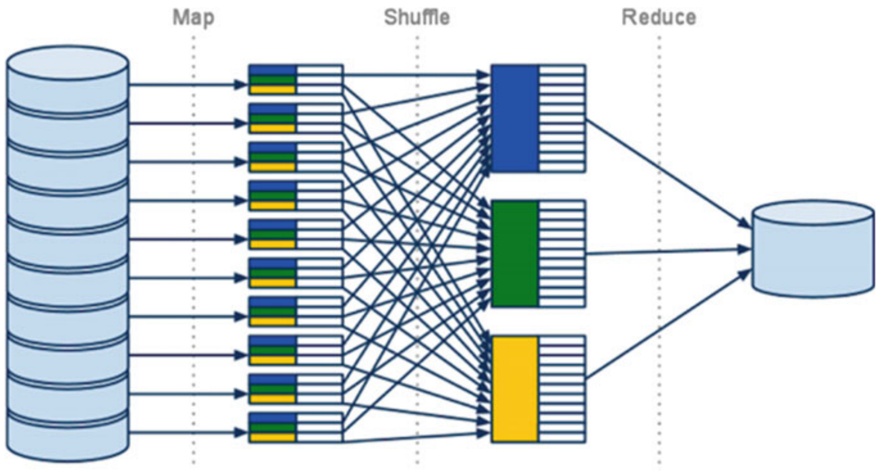


Fig. 3 Architecture of MapReduce model

grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware). Processing can occur on data stored either in a file system (unstructured) or in a database (structured). MapReduce can take advantage of locality of data, processing it on or near the storage assets in order to reduce the distance over which it must be transmitted.

"Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

"Shuffle" step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key are located on the same worker node.

"Reduce" step: Worker nodes now process each group of output data, per key, in parallel.

The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled - assuming the input data is still available.

Hadoop has following important features:

Scalability: Hadoop is highly scalable and it can be scaled out instead of scaling up. The main feature of Hadoop is that the machines with normal functioning capacity can also be used for big data analysis. Multi-node clusters of Hadoop system can be set up in distributed master slave architecture and scalability can be achieved for thousands of nodes.

Fault tolerance: Fault tolerance is the most significant feature of MapReduce programming model that makes it a robust and reliable programming system for large scale data processing. Fault tolerance is achieved in MapReduce by redundancy of data. Each dataset is duplicated in 3–4 places. Even when a node fails, the same dataset can be retrieved from other nodes.

Performance: MapReduce programming model is very efficient for large amounts of data. However, the performance is not good when the dataset is small. The time lag of Hadoop model is compromised because of its efficient fault tolerance and high scalability.

4.1.2 Spark

Spark is an efficient and iterative processing model for big data processing. At its core, Spark provides a general programming model that enables developers to write applications by composing arbitrary operators, such as *mappers*, *reducers*, *joins*, *group-bys*, and *filters*. This composition makes it easy to express a wide array of computations, including iterative machine learning, streaming, complex queries, and batch processing.

Spark programming model focuses on applications that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. It keeps track of the data that each of the operators produces, and enables applications to reliably store this data in memory. This feature enables efficient iterative algorithms and low latency computations.

Spark provides two main abstractions for parallel programming: resilient distributed datasets and parallel operations on these datasets. Spark programming model is shown in Fig. 4 [27]. It describes two kinds of computations, iterative and

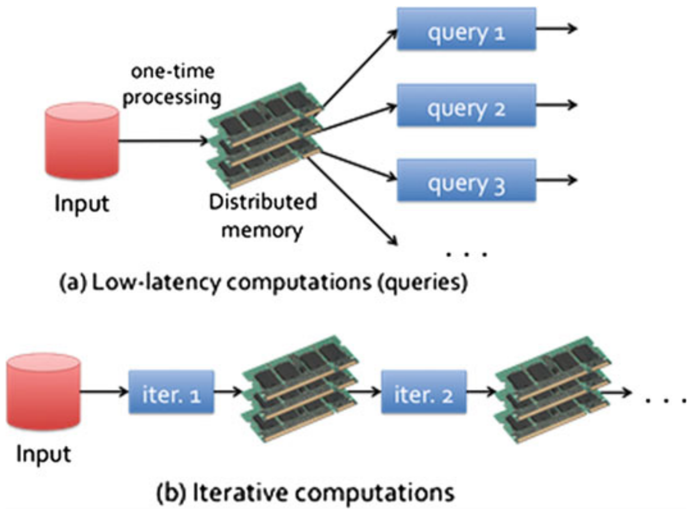


Fig. 4 Spark programming model

non-iterative. The main abstraction in Spark is that of a Resilient Distributed Dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail. In Spark, each RDD is represented by a Scala [25] object. Spark lets programmers construct RDDs in various ways like from a file in a shared file system, by "parallelizing" a Scala collection (e.g., an array) in the driver program, by transforming an existing RDD and by changing the persistence of an existing RDD. Several parallel operations like *reduce*, *collect*, *foreach* etc. can be performed on RDDs.

Spark also lets programmers create two restricted types of shared variables to support two simple but common usage patterns. Programmer can create a "broadcast variable" object that wraps the value and ensures that it is only copied to each worker once. Also, Accumulators can be defined for any type that has an "add" operation and a "zero" value. Due to their "add-only" semantics, they are easy to make fault-tolerant.

Spark is built on top of Mesos [13], a "cluster operating system" that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. This allows Spark to run alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI [26], and share data with them. In addition, building on Mesos greatly reduced the programming effort that had to go into Spark.

The two types of shared variables in Spark, broadcast variables and accumulators, are implemented using classes with custom serialization formats. Spark is implemented in Scala (Scala programming language.), a statically typed high-level programming language for the Java Virtual Machine, and exposes a functional programming interface similar to DryadLINQ [31]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster.

Spark has following important features:

Scalability: It is based on MapReduce architecture so it provides scalability feature.

Fault tolerant: Spark retains the fault tolerant feature of map reduce. Also, its novel feature is the use of Resilient Distributed Datasets (RDD). The main property of RDD is the capability to store its lineage or the series of transformations required for creating it as well as other actions on it. This lineage provides fault tolerance to RDDs.

Easy to use: Spark's parallel programs look very much like sequential programs, which make them easier to develop and reason about. Spark allows users to easily combine batch, interactive, and streaming jobs in the same application. As a result, a Spark job can be up to 100 times faster and requires writing 2–10 times less code than an equivalent Hadoop job. One of Spark's most useful features is the interactive shell, bringing Spark's capabilities to the user immediately - no Integrated Development Environment (IDE) and code compilation required. The shell can be used as the primary tool for exploring data interactively, or as means to test portions of an application you're developing. Spark can read and write data from and to Hadoop Distributed File System (HDFS).

Better Performance: Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

4.1.3 Dryad

Dryad [17] was a research project at Microsoft Research for writing parallel and distributed programs to scale from a small cluster to a large data-center. From 2007, Microsoft made several preview releases of this programming model technology available as add-ons to Windows HPC Server 2008 R2. However, Microsoft dropped Dryad processing work and focused on Apache Hadoop in October 2011. Dryad allows a programmer to use the resources of a computer cluster or a data center for running data-parallel programs. A Dryad programmer can use thousands of machines, each of them with multiple processors or cores, without knowing anything about concurrent programming.

A Dryad programmer writes several sequential programs and connects those using one-way channels. The computation of an application written for Dryad is structured as a Directed Acyclic Graph (DAG). The DAG defines the dataflow of the application, and the vertices of the graph define the operations that are to be performed on the data. The "computational vertices" are written using sequential constructs, devoid of any concurrency or mutual exclusion semantics. A Dryad job is a graph generator which can synthesize any directed acyclic graph. The structure of Dryad jobs is shown in Fig. 5 [28]. These graphs can even change during execution, in response

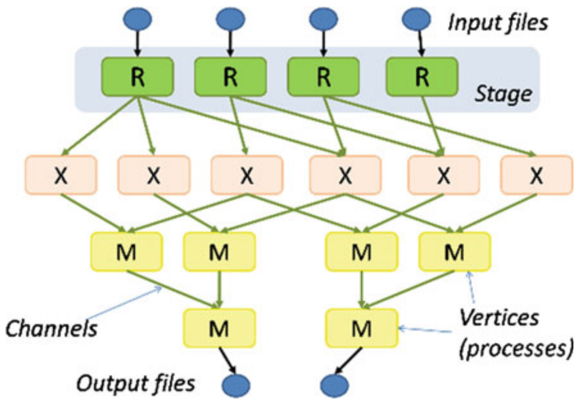


Fig. 5 The structure of Dryad jobs

to important events in the computation. The Dryad runtime parallelizes the dataflow graph by distributing the computational vertices across various execution engines. Scheduling of the computational vertices on the available hardware is handled by the Dryad runtime, without any explicit intervention by the developer of the application or administrator of the network.

The flow of data between one computational vertex to another is implemented by using communication "channels" between the vertices, which in physical implementation is realized by TCP/IP streams, shared memory or temporary files. A stream is used at runtime to transport a finite number of structured items.

Dryad defines a domain-specific language, implemented via a C++ library, that is used to create and model a Dryad execution graph. Computational vertices are written using standard C++ constructs. To make them accessible to the Dryad runtime, they must be encapsulated in a class that inherits from the GraphNode base class. The graph is defined by adding edges; edges are added by using a composition operator that connects two graphs with an edge. A lot of operators are defined to help building a graph, including Cloning, Composition, Merge and Encapsulation. Managed code wrappers for the Dryad API can also be written.

Dryad's architecture includes components that do resource management as well as the job management. A Dryad job is coordinated by a component called the Job Manager. Tasks of a job are executed on cluster machines by a Daemon process. Communication with the tasks from the job manager happens through the Daemon, which acts like a proxy. In Dryad, the scheduling decisions are local to an instance of the Dryad Job Manager C i.e., it is decentralized. The logical plan for a Dryad DAG results in each vertex being placed in a "Stage". The stages are managed by a "Stage manager" component that is part of the job manager. The Stage manager is used to detect state transitions and implement optimizations like Hadoop's speculative execution.

Overall, Dryad is quite expressive. It completely subsumes other computation frameworks, such as Google's MapReduce, or the relational algebra. Moreover, Dryad handles job creation and management, resource management, job monitoring and visualization, fault tolerance, re-execution, scheduling, and accounting.

Dryad has following special features:

Scalability: Dryad is designed to scale to much larger implementations, up to thousands of computers.

Fault tolerance: The fault tolerance model in the Dryad comes from the assumption that vertices are deterministic. Since the communication graph is acyclic, it is relatively straightforward to ensure that every terminating execution of a job with immutable inputs will compute the same result, regardless of the sequence of computer or disk failures over the course of the execution.

Performance: The Dryad system can execute jobs containing hundreds of thousands of vertices, processing many terabytes of input data in minutes. Microsoft routinely uses Dryad applications to analyze petabytes of data on clusters of thousands of computers.

Flexibility: Programmers can easily use thousands of machines and create large-scale distributed applications, without requiring them to master any concurrency programming beyond being able to draw a graph of the data dependencies of their algorithms.

4.2 Graph Parallel Programming Platforms

Graph parallel systems are systems that encode computation as vertex programs which run in parallel and interact along edges in the graph. Graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism and effective partitioning to minimize communication.

4.2.1 Pregel

Pregel [22] is a programming model for processing large graphs in distributed environment. It is a vertex-centric model, which defines serials of actions on an angle of a single vertex, and then the program will run such vertices through a graph and finally get the result.

Pregel has been created for solving large scale graph computations that is required in modern systems like social networks and web graphs. Many graph computing problems like shortest path, clustering, page rank, connected components etc. need to be implemented for big graphs hence the requirement of the system.

Vertices iteratively process data and send messages to neighboring vertices. Edges do not have any associated computation in this programming model. The computations consist of a sequence of iterations, called *supersteps*. Within each *superstep*, the vertices compute in parallel, each executing the same user defined function that expresses the logic of a given algorithm. A vertex can modify its state or that of its outgoing edges, receive messages sent to it in the previous *superstep*, send messages to other vertices (to be received in the next *superstep*), or even mutate the topology of the graph. The state machine of vertex is shown in Fig. 6 [22].

The input to a Pregel computation is a directed graph in which each vertex is uniquely identified by a string vertex identifier. Each vertex is associated with a modifiable, user defined value. The directed edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier. A typical Pregel computation consists of input, when the graph is

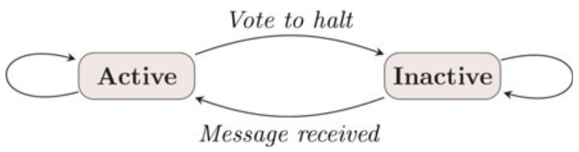


Fig. 6 State machine for a vertex

initialized, followed by a sequence of *supersteps* separated by global synchronization points until the algorithm terminates, and finishing with output. Algorithm termination is based on every vertex voting to halt. The output of a Pregel program is the set of values explicitly output by the vertices. It is often a directed graph isomorphic to the input, but this is not a necessary property of the system because vertices and edges can be added and removed during computation. A clustering algorithm, for example, might generate a small set of disconnected vertices selected from a large graph.

The Pregel library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' outgoing edges. Assignment of a vertex to a partition depends solely on the vertex ID, which implies it is possible to know which partition a given vertex belongs to even if the vertex is owned by a different machine, or even if the vertex does not yet exist. The default partitioning function is just hash (ID) mod *N*, where *N* is the number of partitions, but users can replace it. The execution of Pregel is depicted in Fig. 7 [16]. In the absence of faults, the execution of a Pregel program consists of several stages. First, many copies of the user program begin executing on a cluster of machines. One of these copies acts as the master. It is not assigned any portion of the graph, but is responsible for coordinating worker activity. The workers use the cluster management system's name service to discover the master's location, and send registration messages to the master. Then, the master determines how many partitions the graph will have, and assigns one or more partitions to each worker machine. Having more than one partition per worker allows parallelism among the partitions and better load balancing, and will usually improve performance. Each worker is given the complete set of assignments for all workers.

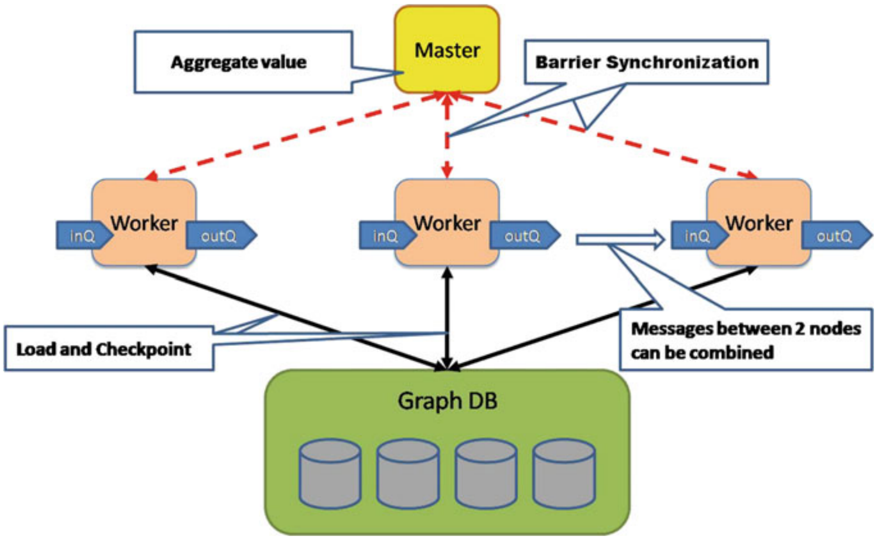


Fig. 7 Implementation of Pregel

After this stage, the master assigns a portion of the user's input to each worker. The input is treated as a set of records, each of which contains an arbitrary number of vertices and edges. The division of inputs is orthogonal to the partitioning of the graph itself, and is typically based on file boundaries. If a worker loads a vertex that belongs to that worker's section of the graph, the appropriate data structures are immediately updated. Otherwise the worker enqueuers a message to the remote peer that owns the vertex. After the input has finished loading, all vertices are marked as active.

Later, the master instructs each worker to perform a *superstep*. The worker loops through its active vertices, using one thread for each partition. The worker calls *Compute()* for each active vertex, delivering messages that were sent in the previous *superstep*. When the worker is finished it responds to the master, telling the master how many vertices will be active in the next *superstep*. This step is repeated as long as any vertices are active, or any messages are in transit. After the computation halts, the master may instruct each worker to save its portion of the graph.

Pregel has following special features:

Scalability: Pregel has very good scalability. It can work for large sized graphs with millions of vertices.

Fault tolerance: Fault tolerance is achieved through check pointing. At the beginning of a *superstep*, the master instructs the workers to save the state of their partitions to persistent storage, including vertex values, edge values, and incoming messages; the master separately saves the aggregator values. Worker failures are detected using regular "ping" messages that the master issues to workers. If a worker does not receive a ping message after a specified interval, the worker process terminates. When one or more workers fail, the current state of the partitions assigned to these workers is lost. The master reassigns graph partitions to the currently available set of workers, and they all reload their partition state from the most recent available checkpoint at the beginning of a *superstep S*.

Performance: Pregel is very fast compared to non-graph based frameworks. But during implementation it waits for the slow workers that decrease its speed.

Flexibility: Pregel provides flexibility to implement different algorithms. The Pregel implementation is easy to understand and implementation of varied algorithms can be done on it. Programming complexity is simplified by using the *supersteps*.

4.2.2 GraphX

GraphX [12] is an efficient, resilient, and distributed graph processing framework that provides graph-parallel abstractions and supports a wide range of iterative graph algorithms. Existing specialized graph processing systems, such as Pregel and GraphLab, are sufficient to process only graph data. Thus, using specialized graph processing systems in large-scale graph analytics pipeline, requires extensive data movement and duplication across file system, and even network. Moreover, users have to learn and manage multiple systems, such as Hadoop, Spark, Pregel and GraphLab. Overall, having separate systems in entire graph analytics pipeline is difficult to use and inefficient.

GraphX addresses the above challenges by providing both table view and graph view on the same physical data. On one hand, GraphX views physical data as graphs so that it can naturally express and efficiently execute iterative graph algorithms. On the other hand, graphs in GraphX are distributed as tabular data-structures so that GraphX also provides table operations on physical data. By exploiting this unified data representation, GraphX enables users to easily and efficiently express the entire graph analytics pipeline. Since graph can be composed by tables in GraphX, tabular data preprocessing and transformation between table and graph are directly realized within one system. Meanwhile, GraphX provides APIs similar to specialized graph processing systems for naturally expressing and efficiently executing iterative graph algorithms. Moreover, GraphX can leverage in-memory computation and fault-tolerance by being embedded in Spark, a general-purpose distributed dataflow framework.

Programmers can implement iterative graph algorithms without caring much about the iterations and only need to define a vertex program. However, the foundation of GraphX' graph-parallel abstractions is different from the common one that is iterative local transformation [12]. GraphX further decomposes iterative local transformation into specific dataflow operators, which are a sequence of join stages and group-by stages punctuated by map operations. The join operation and group-by operation are in the context of relational database, and the map operation is to perform update. GraphX realizes the partitioning of graphs in its representation of physical data, called distributed graph representation. Figure 8 [12] illustrates how a graph is represented by horizontally partitioned vertex and edge collections and their indices. The edges are divided into three edge partitions by applying a partition function (e.g., 2D Partitioning), and the vertices are partitioned by vertex id. Partitioned with the vertices, GraphX maintains a routing table encoding the edge partitions for each vertex.

GraphX is built as a library on top of Spark [32], which is a general-purpose distributed dataflow framework. The architecture of Spark with GraphX is illustrated by Fig. 9 [12]. As seen from the architecture, there is one more layer called Gather Apply Scatter (GAS) Pregel API between GraphX and some graph algorithms. The GAS Pregel API is implementation of Pregel abstraction of graph-parallel using GraphX dataflow operations. It is claimed that GraphX can implement Pregel abstractions in less than 20 lines of codes. Data structure of GraphX, the distributed graph representation, is built on Spark RDD abstraction, and GraphX API is expressed on top of Spark standard dataflow operators. GraphX can also exploited Scala foundation of Spark, which enables GraphX to interactively load, transform, and compute on massive graphs. GraphX requires no modifications to Spark. As a result, GraphX can also be seen as a general method to embed graph computation within distributed dataflow frameworks and distill graph computation to a specific join-map-group-by dataflow pattern. Being embedded in Spark allows GraphX to inherit many good

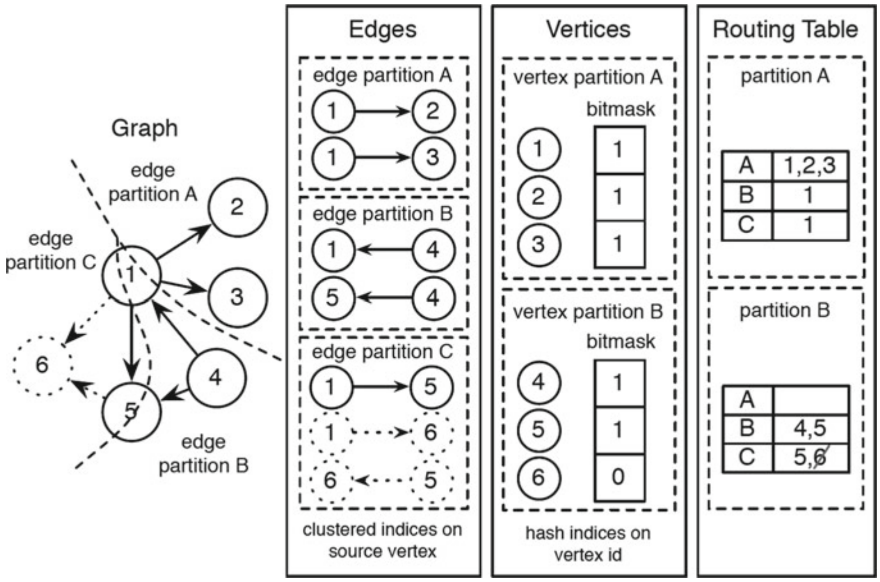


Fig. 8 Distributed graph representations

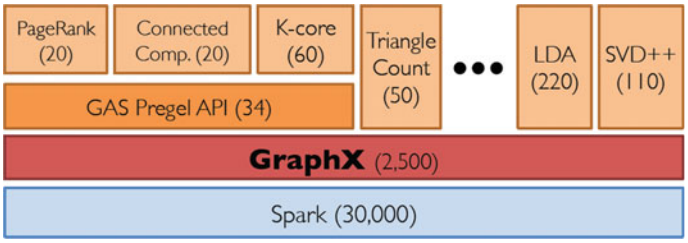


Fig. 9 Spark with GraphX

features of Spark, such as in-memory computation and fault-tolerance. Compared with Pregel and GraphLab, GraphX can achieve these features with a smaller cost.

GraphX has following important features:

Scalability: Being embedded in Spark allows GraphX to inherit Spark scalable property.

Fault tolerance: Being embedded in Spark allows GraphX to inherit Spark fault tolerance. Different from checkpoint-based fault tolerance, which is adopted by other graph systems, fault tolerance of GraphX is based on lineage. Compared with checkpoint fault tolerance, lineage-based fault tolerance produces smaller performance overhead and optimal dataset replication.

Efficient for graph analytics pipeline: Similar to specialized graph processing systems, such as Pregel and GraphLab, GraphX enables users to naturally express and efficiently execute iterative graph algorithms. Moreover, GraphX provides operations for tabular data preprocessing, and transformation between graph and tabular data so that there is no data movement and duplication across the network and file system.

Support for SQL: Being embedded in Spark allows GraphX to inherit Spark SQL.

4.2.3 GraphLab

GraphLab is an efficient and parallel processing model for big data processing especially for large graph processing. As its core, GraphLab supports the representation of structured data dependencies, iterative computation, and flexible scheduling. By targeting common patterns in machine learning algorithms and tasks, GraphLab achieves notable usability, expressiveness and performance.

GraphLab programming model focuses on applications that share a coherent computational pattern: *asynchronous iterative and parallel computation on graphs with a sequential model of computation*. This pattern encodes a broad range of machine learning algorithms, and facilitates efficient parallel implementations.

GraphLab exploits the *sparse structure* and common *computational patterns* of machine learning algorithms, and by composing problem specific computation, data-dependencies, and scheduling, it enables users to easily design and implement efficient parallel algorithms.

GraphLab's ease-of-use comes from its abstraction which consists of the following parts: the data graph, the update function, scheduling primitives, the data consistency model, and the sync operation. The data graph represents user modifiable program state, stores the user-defined data and encodes the sparse computational dependencies, an example is shown in Fig. 10 [21]. The update function represents the operation and computation on the data graph by transforming data in small overlapping contexts called scopes. Scheduling primitives determine the computation order. The data consistency model expresses how much computation can overlap. Last, the sync operation concurrently keeps track of global states.

The GraphLab is implemented in the shared memory setting [20] and distributed in-memory setting [21]. In the shared memory setting, the GraphLab abstraction uses *PThreads* for parallelism. The data consistency models have been implemented using race-free and deadlock-free ordered locking protocols. To attain maximum performance, issues related to parallel memory allocation, concurrent random number generation, and cache efficiency are addressed in [20]. The shared memory setting is extended to the distributed setting by refining the execution model, relaxing the scheduling requirements, and introducing a new distributed data-graph, execution engines, and fault-tolerance systems [21].

The GraphLab API serves as an interface between the machine learning and systems communities. Parallel machine learning algorithms built on the GraphLab API benefit from developments in parallel data structures. As new locking protocols and parallel scheduling primitives are incorporated into the GraphLab API, they become

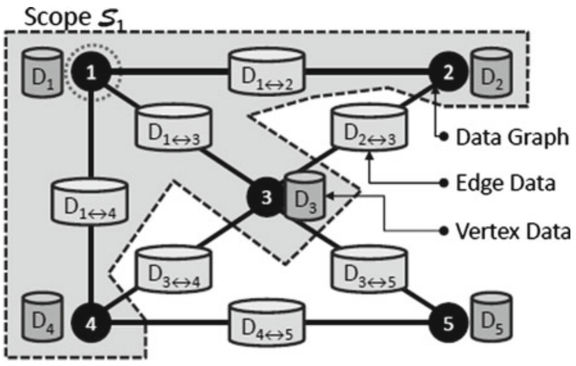


Fig. 10 The GraphLab data graph and scope S_1 of vertex 1 are illustrated in this figure. Each *gray cylinder* represents a block of user defined data and is associated with a vertex or edge. The scope of vertex 1 is illustrated by the region containing vertices $\{1, 2, 3, 4\}$. An update function applied to vertex 1 is able to read and modify all the data in S_1 (vertex data D_1, D_2, D_3, D_4 and edge data $D_{1\to 2}$, $D_{1\to 3}$, and $D_{1\to 4}$)

immediately available to the machine learning community. On the other hand, Systems experts can use machine learning algorithms to new parallel hardware more easily by porting the GraphLab API. Actually, on top of GraphLab, several implemented libraries of algorithms in various application domains are already provided including topic modeling, graph analytics, clustering, collaborative filtering, computer vision etc.

GraphLab has following important features:

Scalability: GraphLab scales very well in various machine learning and data mining tasks, and scaling performance improves with higher computation to communication ratio.

Expressivity: Unlike many high-level abstractions (i.e., MapReduce), GraphLab is able to express complex computational dependencies with the data graph and provides sophisticated scheduling primitives which can express iterative parallel algorithms with dynamic scheduling.

Better Performance: GraphLab can outperform Hadoop by 20–60x in iterative machine learning and data mining tasks, and is competitive with tailored MPI implementation. The C++ execution engine is optimized to leverage extensive multithreading and asynchronous IO.

Powerful Machine Learning Toolkits: GraphLab has a large selection of machine learning methods already implemented. Users can also implement their own algorithms on top of the GraphLab programming API.

4.3 Task Parallel Platforms

Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer codes across multiple processors in parallel computing environments. Task parallelism focuses on distributing execution processes (threads) across different parallel computing nodes. In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication usually takes place by passing data from one thread to the next as part of a workflow.

4.3.1 HTCondor

HTCondor has been derived from Condor that is a batch system for harnessing idle cycles on personal workstations [19]. Since then, it has matured to become a major player in the compute resource management area and renamed HTCondor in 2012. HTCondor (HTCondor) is a high throughput computing system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management.

HTCondor is able to transparently produce a checkpoint and migrate a job to a different machine which would otherwise be idle when it detects that a machine is no longer available. It does not require a shared file system across machines - if no shared file system is available, it can transfer the job's data files on behalf of the user, or it may be able to transparently direct all the job's I/O requests back to the submit machine. As a result, it can be used to seamlessly combine all of an organization's computational power into one resource.

HTCondor programming model has several logical entities, as shown in Fig. 11 [23]. The central manager acts as a repository of the queues and resources. A process called the "collector" acts as an information dashboard. A process called the "startd" manages the computes resources provided by the execution machines (worker nodes in the diagram). The *startd* gathers the characteristics of compute resources such as CPU, memory, system load, etc. and publishes it to the collector. A process called the "schedd" maintains a persistent job queue for jobs submitted by the users. A process called the "negotiator" is responsible for matching the computer resources to user jobs.

The communication flow in Condor is fully asynchronous. Each *startd* and each *schedd* advertise the information to the collector asynchronously. Similarly, the negotiator starts the matchmaking cycle using its own timing. The negotiator periodically queries the *schedd* to get the characteristics of the queued jobs and matches them to available resources. All the matches are then ordered based on user priority and communicated back to the *schedds* that in turn transfer the matched user jobs to

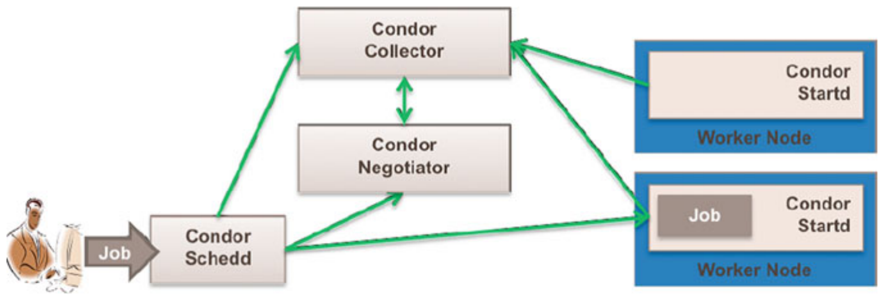


Fig. 11 Condor architecture overview

the selected *startds* for execution. To fairly distribute the resources among users, the negotiator tracks resource consumption by users and calculates user priorities accordingly.

Condor supports the transferring of input files to a worker node (startd) before a job is launched and of output files to the submit node (schedd) after the job is finished. Using a flexible plugin architecture, HTCondor can easily be extended to support domain specific protocols, such as GridFTP and Globus Online.

HTCondor has following important features:

Flexibility: The *ClassAd* mechanism in HTCondor provides an extremely flexible and expressive framework for matching resource requests (jobs) with resource offers (machines). Jobs can easily state both job requirements and job preferences. Likewise, machines can specify requirements and preferences about the jobs they are willing to run.

Efficiency: HTCondor is a high throughput computing system. Also, it utilizes the computing resources in a very efficient way.

4.4 Stream Processing Programming Platforms

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in minutes; a search site may wish to model which users visit a new page; and a service operator may wish to monitor program logs to detect failures in seconds. To enable these low-latency processing applications, there is need for streaming computation models that scale transparently to large clusters, in the same way that batch models like MapReduce simplified offline processing.

Designing such models is challenging, however, because the scale needed for the largest applications can be hundreds of nodes. At this scale, two major problems are faults and stragglers (slow nodes). Both problems are inevitable in large clusters, so streaming applications must recover from them quickly. Given below are some popular programming platforms for stream processing.

4.4.1 Storm

Apache Storm is a free and open source distributed real-time computation system. Storm is a complex event processing engine from Twitter [30]. Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing [29].

It has been used by various companies for many purposes like real time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. The fundamental concept in Storm is that of a stream, which can be defined as an unbounded sequence of tuples. Storm provides ways to transform the stream in various ways in decentralized and fault tolerant manner [1].

The storm topology lays down the architecture for processing of streams. The topology comprises of a spout, which is a reader or source of streams and a bolt, which is a processing entity and wiring together of spouts and bolts as shown in Fig. 12 [2].

Clients submit topologies to a master node, which is called the *Nimbus*. Nimbus is responsible for distributing and coordinating the execution of the topology. The actual work is done on worker nodes. Each worker node runs one or more worker processes. At any point in time a single machine may have more than one worker processes, but each worker process is mapped to a single topology. Note more than one worker process on the same machine may be executing different part of the same topology. The high level architecture of Storm is shown in Fig. 13 [22].

Each worker process runs a JVM, in which it runs one or more executors. Executors are made of one or more tasks. The actual work for a bolt or a spout is done in

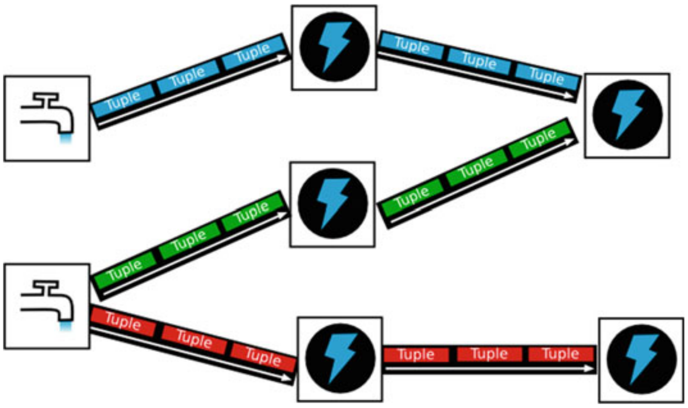


Fig. 12 Storm topology

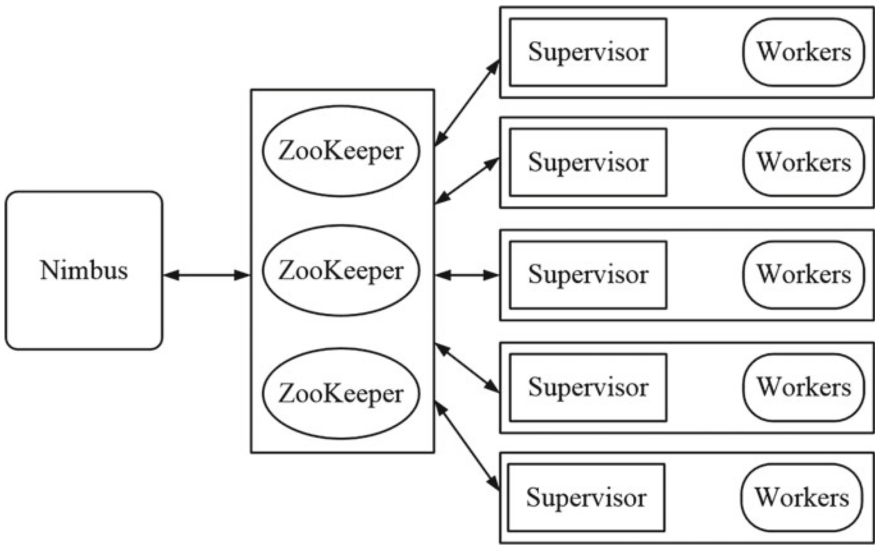


Fig. 13 High level architecture of Storm

the task. Thus, tasks provide intra-bolt/intra-spout parallelism, and the executors provide intra-topology parallelism. Worker processes serve as containers on the host machines to run Storm topologies. Spouts can read streams from Kafka (distributed publish-subscribe system from LinkedIn), Twitter, RDBMS etc.

Storm supports the following types of partitioning strategies. *Shuffle* grouping randomly partitions the tuples. *Fields* grouping hashes on a subset of the tuple attributes/fields. *All* grouping replicates the entire stream to all the consumer tasks. *Global* grouping sends the entire stream to a single bolt. *Local* grouping sends tuples to the consumer bolts in the same executor. The partitioning strategy is extensible and a topology can define and use its own partitioning strategy.

Each worker node runs a Supervisor that communicates with Nimbus. The cluster state is maintained in Zookeeper [3], and Nimbus is responsible for scheduling the topologies on the worker nodes and monitoring the progress of the tuples flowing through the topology.

Storm currently runs on hundreds of servers (spread across multiple datacenters) at Twitter. Several hundreds of topologies run on these clusters, some of which run on more than a few hundred nodes. Many terabytes of data flows through the Storm clusters every day, generating several billions of output tuples. Storm topologies are used by a number of groups inside Twitter, including revenue, user services, search, and content discovery. These topologies are used to do simple things like filtering and aggregating the content of various streams at Twitter (e.g., computing counts), and also for more complex things like running simple machine learning algorithms (e.g., clustering) on stream data.

Storm has following important features:

Scalability: It is scalable. It is easy to add or remove nodes from storm cluster without disrupting existing data flows.

Fault tolerance: Storm guarantees that the data will be processed. Storm is very resilient in regards to fault tolerance.

Easy to use: Storm is very easy to set up and operate.

Extensibility: Storm topologies may call arbitrary external functions (e.g., Looking up a MySQL service for the social graph), and thus needs a framework that allows extensibility.

Efficiency: Storm uses a number of techniques, including keeping all its storage and computational data structures in memory. Storm is very fast in processing. A benchmark clocked it at over a million tuples processed per second per node.

4.4.2 S4

Simple Scalable Streaming System (shorted for S4) [24] was released for processing continuous, unbounded streams of data by Yahoo. S4 is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data.

S4 is designed to solve real-world problems in the context of search applications that use data mining and machine learning algorithms. Compared with current processing systems, S4, a low latency, scalable stream processing engine, is developed. The stream throughput is improved by 1000% (200 k + messages /s /stream) in S4 [18].

The design goal of S4 is developing a high performance computing platform that can hide the complexity inherent in a parallel processing system from the application programmer. Simple programming interfaces for processing data streams are provided in S4. A cluster with high availability is designed; the cluster can scale using commodity hardware. Latency is minimized by using local memory in each processing node, and the disk I/O bottlenecks are avoided as well. A symmetric and decentralized architecture is used in S4. Because all nodes in S4 share the same functionality and responsibilities, there is no central node with specialized responsibilities. Thus, the deployment and maintenance of S4 are greatly simplified. The design is friendly and easy to program and flexible by using a pluggable architecture. The gap between complex proprietary systems and batch-oriented open source computing platforms is filled in S4 [18].

S4 provides a runtime distributed platform that handles communication, scheduling and distribution across containers. The nodes are the distributed containers, which are deployed in S4 clusters. The size of clusters is fixed in S4, the size of an S4 cluster corresponds to the number of logical partitions (tasks). The key concepts are shown in Fig. 14 [18].

In S4, computation is executed by Processing Elements (PEs) and messages are transmitted between them in the form of data events. The stream is defined as a

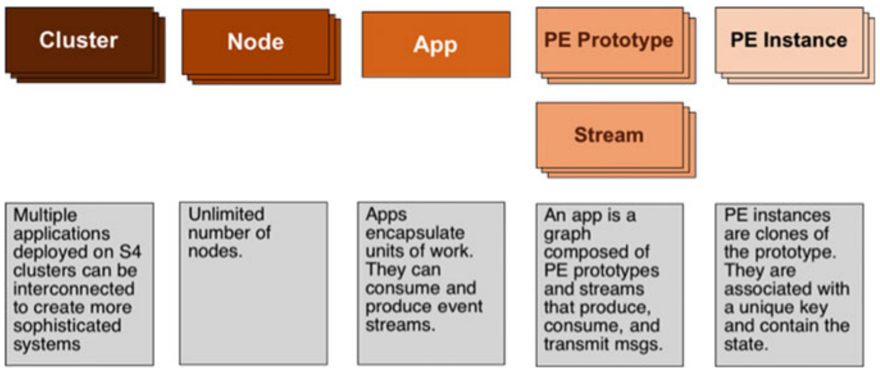

Cluster
Node
App
PE Prototype
PE Instance
Stream
Multiple applications deployed on S4 clusters can be interconnected to create more sophisticated systems
Unlimited number of nodes.
Apps encapsulate units of work. They can consume and produce event streams.
An app is a graph composed of PE prototypes and streams that produce, consume, and transmit msgs.
PE instances are clones of the prototype. They are associated with a unique key and contain the state.

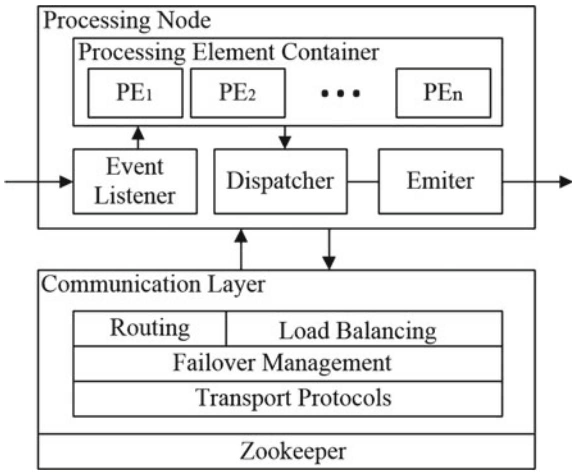
Fig. 14 Key concepts in S4 (Incubator)

sequence of elements (events). The only mode of interaction between PEs is event emission and consumption. PE cannot access to the state of other PEs. The framework provides the capability to route events to appropriate PEs and to create new instances of PEs [24].

PEs are assembled into applications using the Spring Framework. Processing Elements (PEs) are the basic computational units in S4. Each instance of a PE is uniquely identified by four components (the functionality, the types of events, the keyed attribute and the value of keyed attribute).

Processing nodes (PNs) are the logical hosts to PEs. They are responsible for listening to events, executing operations on the incoming events, dispatching events with the assistance of the communication layer, and emitting output events (Fig. 15 [24]). S4 routes each event to PNs based on a hash function of the values of all known

Fig. 15 Processing node



keyed attributes in that event. A single event may be routed to multiple PNs. The set of all possible keying attributes is known from the configuration of the S4 cluster. An event listener in the PN passes incoming events to the processing element container (PEC) which invokes the appropriate PEs in the appropriate order. There is a special type of PE object: the PE prototype. It has the first three components of its identity (functionality, event type, keyed attribute); the attribute value is unassigned.

The communication layer uses Zookeeper (an open source subproject of Hadoop maintained) (Apache ZooKeeper) to coordinate between nodes in a cluster. The communication layer can provide cluster management and automatic failover to standby nodes and maps physical nodes to logical nodes. The communication layer uses a pluggable architecture to select network protocol. Events may be sent with or without a guarantee.

The core platform is written in Java. The implementation is modular and pluggable, and S4 applications can be easily and dynamically combined for creating more sophisticated stream processing systems. Every PE consumes exactly those events which correspond to the value on which it is keyed. It may produce output events. Two primary handlers are implemented by developers: an input event handler *processEvent()* and an output mechanism *output()*. The *output()* method is optional and is set to be invoked in a variety of ways. The *output()* method implements the output mechanism for the PE, typically to publish internal state of the PE to some external system [24].

S4 has following important features:

Fault tolerance: When a server in the cluster fails, a stand-by server is automatically activated to take over the tasks. Check pointing and recovery mechanism are used to minimize state loss.

Flexible deployment: Application packages and platform modules are standard jar files (suffixed.s4r). The keys are homogeneously sparsed over the cluster, the flexible deployment can help balance the load, especially for fine grained partitioning.

Modular design: Both the platform and the applications are built by dependency injection, and configured through independent modules. The system is easy to be customized according to specific requirements.

Dynamic and loose coupling of applications: The subsystems are easy to be assembled into larger systems. The applications can be reused in S4, and pre-processing can be separated. The subsystems can be controlled and updated independently.

4.4.3 Spark Streaming

Spark Streaming system simplifies the construction of scalable fault-tolerant streaming applications. The authors propose a new processing model, discretized streams (D-Streams), that overcomes these challenges [33]. D-Streams enable a parallel recovery mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. D-Streams build applications through high-level operators and make efficient fault tolerance while combining streaming with batch and interactive queries.

Existing streaming models use replication or upstream backup for fault tolerance. This mechanism costs much time on fault tolerance and stragglers. Also their event driven programming interface does not directly support parallel processing. The purposes of Spark Streaming are to directly support parallel processing, fault tolerance and efficient stragglers.

Unlike stateful programming model, Spark Streaming use batch processing method to process continuous streaming and cut streaming into discretized intervals. It can take advantage of batch operations in Spark and also provide typical streaming operations. Spark Streaming uses short stateless, deterministic tasks instead of continues, stateful operators. The state stored in memory across tasks into RDD. Spark Streaming runs a streaming computation as a series of very small, deterministic batch jobs. When the streaming data is coming, Spark Streaming chops up the live stream into batches of 0.5–1 second. It treats each batch of data as RDDs and processes them using RDD operations. In this way, it has potential for combining batch processing and streaming processing in the same system.

For fault-tolerance, RDDs remember the operations that created them and replicated batches of input data in memory for fault-tolerance. So data lost due to worker failure can be recomputed from replicated input data via RDD. Therefore, all data is fault-tolerant. The lineage graph of RDD is shown in Fig. 16 [33].

Spark Streaming can easily be composed with batch and query model. It provides both batch operation in Spark and standard streaming systems (Das) [7]. Batch API in Spark includes *Map*, *Reduce*, *GroupBy*, *Join* operations. Streaming API in Spark supports *Windowing*, *Incremental Aggregation* operations.

Spark Streaming consists of three components, shown in Fig. 17 [7]. A master, that tracks the D-Stream lineage graph and schedules tasks to compute new RDD

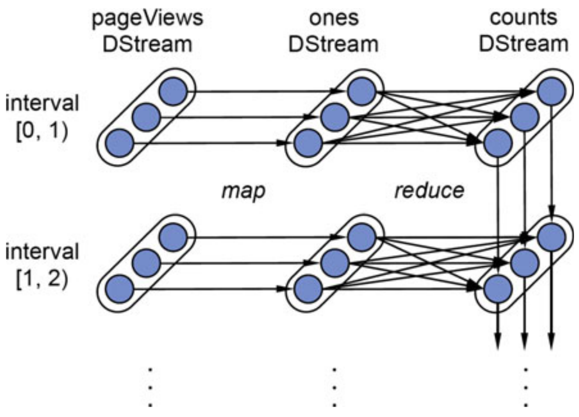


Fig. 16 Lineage graph for RDDs

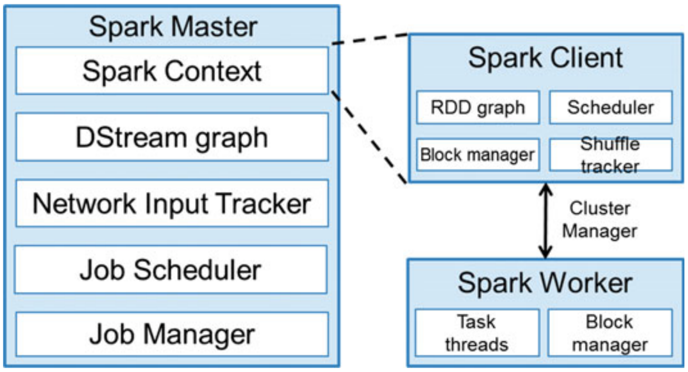


Fig. 17 Components of Spark Streaming architecture

partitions. Worker nodes that receive data, store the partitions of input and computed RDDs, and execute tasks. A client library used to send data into the system.

In the Spark Master, network Input Tracker keeps track of the data received by each network receiver and maps them to the corresponding input *DStreams*. Job Scheduler periodically queries the *DStream* graph to generate Spark jobs from received data, and hands them to Job Manager for execution. Job Manager maintains a job queue and executes the jobs in Spark.

Spark Streaming has many important features that make it a desirable programming platform. It scales to 100 s of nodes and achieves second scale latencies. It enables efficient and fault-tolerant stateful stream processing while integrating with Spark's batch and interactive processing. Spark provides a simple batch-like API for implementing complex algorithms.

5 A Unifying Framework

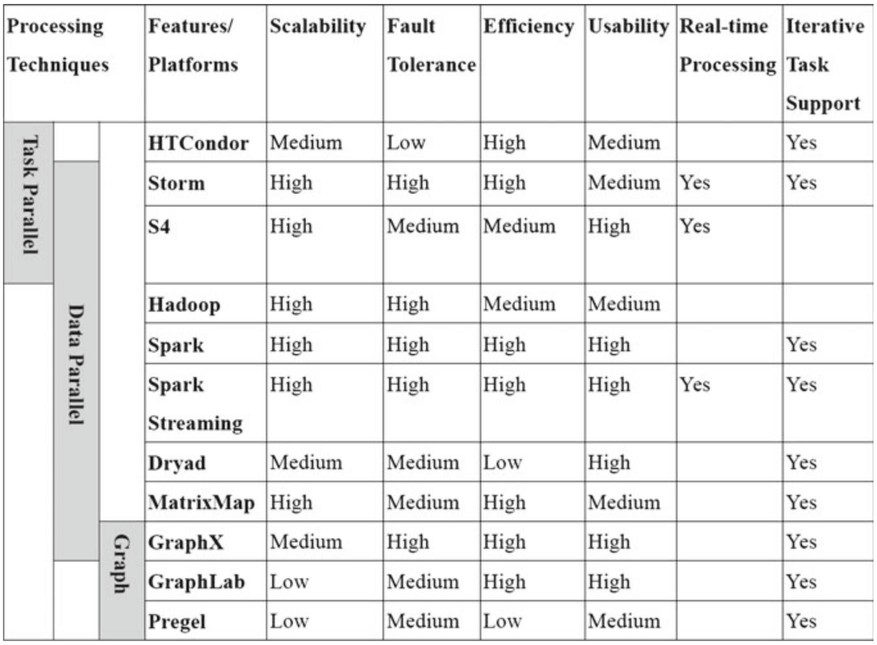
The existing programming platforms have various features that are relevant for particular kinds of applications. While some systems like traditional systems, MapReduce, Hadoop, Pregel are generic systems with limited abilities, and other systems are very specific for certain kind of applications like streaming data or graph based data. In this section, we compare different programming platforms against features that are important for big data analysis as mentioned in Sect. 2. We then discuss the existing challenges, and describe the need for a unifying framework that allows a generic abstraction over the underlying models and any new upcoming models. Then we present our framework MatrixMap that overcomes the challenges of supporting matrix computations in an efficient manner.

5.1 *Comparison of Existing Programming Platforms*

The comparison of various programming platforms with respect to some important features as discussed in the corresponding sections is summarized in Table 1. Most of the data parallel programming platforms have high scalability. Hadoop derivatives like Spark and Spark Streaming inherit similar characteristics for high scalability with distributed processing. Real time processing is supported by Storm, S4 and Spark Streaming in an efficient manner. Fault tolerance in big data analytics is a critical feature because of dependency on multiple systems and size of application. It is observed that Hadoop, Spark, Spark Streaming, GraphX and Storm are highly fault tolerant as they use redundancy and special data structures called RDDs. GraphLab, Pregel and S4 use checkpointing for fault tolerance.

The newer programming platforms like Storm, Spark Streaming have most attributes required for efficient big data analysis. Much research is being carried out to develop all machine learning algorithms for newer systems. For MapReduce based systems, not all the machine learning algorithms can be formulated as map and reduce problems. For interactive analysis, Storm, S4 and Spark Streaming can be used as programming platforms.

Table 1 Comparison of Programming platforms for big data analysis



Processing Techniques			Features/ Platforms	Scalability	Fault Tolerance	Efficiency	Usability	Real-time Processing	Iterative Task Support
Task Parallel			HTCondor	Medium	Low	High	Medium		Yes
Task Parallel	Data Parallel		Storm	High	High	High	Medium	Yes	Yes
Task Parallel	Data Parallel		S4	High	Medium	Medium	High	Yes	
	Data Parallel		Hadoop	High	High	Medium	Medium		
	Data Parallel		Spark	High	High	High	High		Yes
	Data Parallel		Spark Streaming	High	High	High	High	Yes	Yes
	Data Parallel		Dryad	Medium	Medium	Low	High		Yes
	Data Parallel		MatrixMap	High	Medium	High	Medium		Yes
	Data Parallel	Graph	GraphX	Medium	High	High	High		Yes
		Graph	GraphLab	Low	Medium	High	High		Yes
		Graph	Pregel	Low	Medium	Low	Medium		Yes

5.2 Need for Unifying Framework

One of the existing challenges in big data programming is that no single programming model or framework can excel at every problem. Different big data programming platforms address different requirements, e.g., some platforms support graph based processing and some systems are specifically designed for streaming data. Programmers need to spend much time learning individual models and their corresponding language, and there are always tradeoffs between simplicity, expressivity, fault tolerance, performance etc.

Therefore, there is need of a unifying framework that allows for a generic abstraction on top of the underlying models and upcoming new models like MatrixMap as shown in Fig. 18. Such an abstraction would integrate different programming platforms so that the programmers only need to learn a single language and techniques for diverse big data applications. Integration of big data platforms would require unifying the interface so that data and operations supported by different models can be abstracted, and mapping each data processing stage to underlying models. In addition, both inter-model and intra-model tasks need to be scheduled on processing units for better efficiency. The cloud resources also have to be allocated dynamically after analyzing the different computation requirements. Integrating data storage systems such as file systems and special databases are another issue. There are various open challenges in it calling for future research efforts.

Besides this, there are still many problems for the existing platforms when performing big data analysis in different application scenarios. Thus, designing new programming platform is another challenge that attracts much attention in the research communities. We will present our proposed platform MatrixMap in the next section.

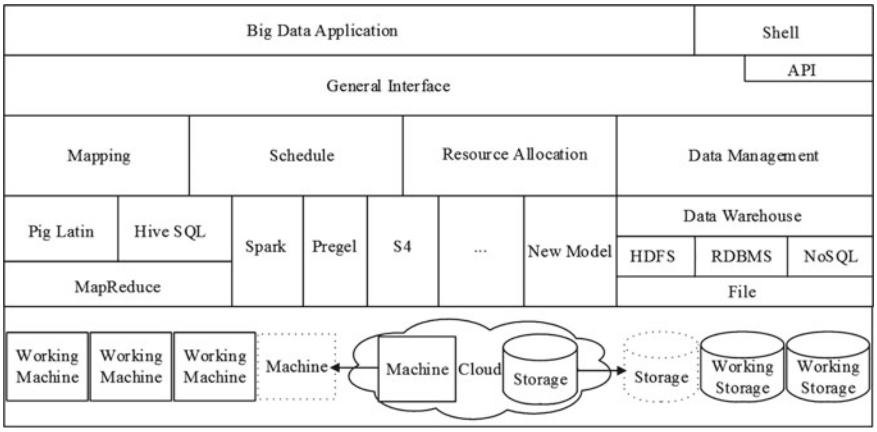


Fig. 18 Integration of diverse big data programming platforms

5.3 *MatrixMap Framework*

Machine learning and graph algorithms play vital important roles in big data analytics. Most algorithms are formulated based into matrix computations. That is they apply matrix operations on values and perform various manipulations of values according to their labels. However, existing big data programming platforms do not provide efficient support for matrix computations.

Most programming platforms provide separate models for machine learning and graph algorithms, e.g., in Spark has different interfaces: GraphX for graph algorithms and Spark for machine learning. The existing systems do not have direct support for important matrix operations, e.g., in MapReduce, matrix multiplication must be formulated into a series of map and reduce operations. The support is mostly limited to matrix multiplication, but not other popular machine learning and graph algorithms, e.g., Presto. Systems besides Spark save temporal data in secondary storage, slow to load data for operations. The cache memory uses LRU algorithm (e.g., Spark), which may not be efficient for all operations. These challenges have led us to develop a model and framework for handling matrix based computations for big data analysis.

MatrixMap [15] is a new model and framework to support data mining and graph algorithms. It provides matrix as language-level construct. The data is loaded into key matrices and then powerful and simple matrix patterns are provided that support basic operations for machine learning and graph algorithms. This model unifies data-parallel and graph-parallel models by abstracting matrix computations into graph patterns.

The framework implements parallel processing of matrix operations and data manipulations invoked by user defined functions. MatrixMap supports high-volume data with pattern-specific fetching and caching across memory and secondary storage.

Algorithms are formulated as a series of matrix patterns, which define sequences of operations on each element. Unary Operator: Map, Reduce; Binary Operator: Plus, Multiply; Mathematical matrix operations are special cases of matrix patterns filled with specific pre-defined lambda functions; User defined lambda functions according to matrix patterns to support various algorithms.

The data is loaded into Bulk Key Matrix (BKM) which is suitable for large volume data. BKM is a shared distributed data structure which spreads data into whole clusters. It can keep data across matrix patterns. It is constant and cannot be changed, after initiation. BKM is row-oriented or column-oriented. It cannot slice concrete matrix element. BKM use key (string or digit) to index row or column. MatrixMap adopts BSP model, while supporting asynchronous pipeline of IO and processing with data partitioning as shown in Fig. 19 [15].

There are many applications of matrix patterns like logistic regression, alternating least squares, all pairs shortest path, Pagerank among other applications. When compared with Spark, it achieved 20% improvement on execution time - the more iterations, the better as shown in Fig. 20.

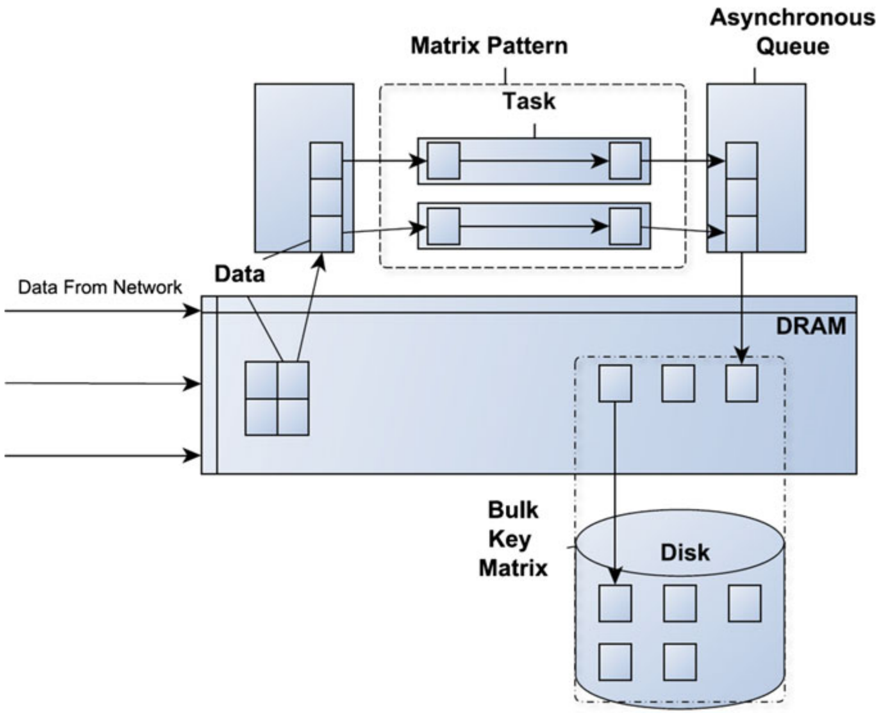


Fig. 19 Implementation of MatrixMap

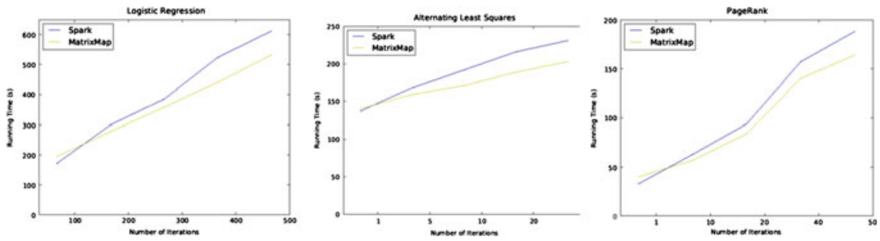


Fig. 20 MatrixMap performance w.r.t. Spark

MatrixMap provides powerful yet simple abstraction, consisting of a distributed data structure called bulk key matrix and a computation interface defined by matrix patterns. Users can easily load data into bulk key matrices and program algorithms into parallel matrix patterns. MatrixMap outperforms current state-of-the-art systems by employing three key techniques: matrix patterns with lambda functions for irregular and linear algebra matrix operations, asynchronous computation pipeline with optimized data shuffling strategies for specific matrix patterns and in-memory data structure reusing data in iterations. Moreover, it can automatically handle the parallelization and distribute execution of programs on a large cluster.

6 Conclusion and Future Directions

The purpose of this chapter is to survey various existing programming platforms for big data analysis. We have enumerated various essential features that a programming environment should possess for big data analysis. The prominent programming platforms have been discussed in brief to give an insight into their purpose, programming model, implementation and features. The comparisons of existing programming platforms against various features have been summarized as well as the need for a unifying framework and our proposed MatrixMap framework that implements machine learning and graph based algorithms using matrices as language constructs, which can handle large data in an efficient manner. In future, we would investigate more in unifying framework for different big data platforms, and improve the MatrixMap framework so that multiple machine learning algorithms can be implemented for different kinds of data. In summary, we can say that research and development of big data programming platforms are driven by real world applications and key industrial stakeholders and it's a challenging but compelling task. Programming platforms for handling big data specially streaming data are still evolving. Samza [9] is a recent addition to programming platforms for streaming data. The concept of "Lambda Architecture" that integrates batch processing and real time processing together in a harmonious way in terms of batch, speed and serving is also an area of interest for the researchers. The integration of different big data programming platforms is an open challenge with various issues related to task scheduling, resource allocation and model mapping to be resolved; while designing new platforms to better perform big data analysis in different application scenarios is another one. Developing a higher-level programming support on top of multiple models can help ease and shorten the development of big data applications.

Acknowledgements This work was partially supported by the funding for Project of Strategic Importance provided by The Hong Kong Polytechnic University (1-ZE26) and HK RGC under GRF Grant (PolyU 5104/13E).

References

1. V. Agneeswaran, *Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives*, 1st edn. (Pearson FT Press, USA, 2014)
2. Apache storm documentation, https://storm.apache.org/documentation/Home.html
3. Apache zookeeper, http://zookeeper.apache.org
4. Architecture of mapreduce model, https://cloud.google.com/appengine/docs/-python/images/mapreduce_mapshuffle.png
5. A.B. Bondi, Characteristics of scalability and their impact on performance, in *Workshop on Software and Performance* (2000), pp. 195C203
6. W. Daniel Hillis, G.L. Steele, Jr., Data parallel algorithms. Commun. ACM, **29**(12), 1170C1183 (1986)
7. T. Das, Deep dive into spark streaming. http://spark.apache.org/-documentation.html (2013)

8. J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1):107C113 (2008)
9. T. Feng, Z. Zhuang, Y. Pan, H. Ramachandra, A memory capacity model for high performing data-filtering applications in samza framework, in *2015 IEEE International Conference on Big Data, Big Data 2015*, Santa Clara, CA, USA, October 29 - November 1, 2015, p. 2600C2605
10. A. Fernández, S. del Ró, V. López, A. Bawakid, M. José del Jesús, J. Manuel Bentez, F. Herrera, Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks. Wiley Interdisc. Rew.: Data Min. Knowl. Discov. **4**(5), 380C409 (2014)
11. J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, Hollywood, CA, USA, October 8-10, 2012, p. 17C30
12. J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, Graphx: graph processing in a distributed dataflow framework, in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 14*, Broomfield, CO, USA, October 6–8, 2014, p. 599C613
13. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011*, Boston, MA, USA (2011)
14. Htcondor, http://research.cs.wisc.edu/htcondor/description.html
15. Y. Huangfu, J. Cao, H. Lu, G. Liang, Matrixmap: programming abstraction and implementation of matrix computation for big data applications, in *21st IEEE International Conference on Parallel and Distributed Systems, ICPADS 2015*, Melbourne, Australia (2015), p. 19C28
16. Implementation of pregel, http://people.apache.org/~edwardyoon/documents/-pregel.pdf
17. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in *Proceedings of the 2007 EuroSys Conference*, Lisbon, Portugal, March 21–23, 2007, p. 59C72
18. Key concepts in s4 (incubator), https://incubator.apache.org/s4/doc/0.6.0/-overview
19. M. J. Litzkow, M. Livny, M.W. Mutka, Condor - a hunter of idle workstations, in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, USA, June 13–17, 1988, p. 104C111
20. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: a new framework for parallel machine learning, in *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, Catalina Island, CA, USA, July 8–11, 2010, p. 340C349
21. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716C727 (2012)
22. G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*, Indianapolis, Indiana, USA (2010), p. 135C146
23. P. Mhashilkar, Z. Miller, R. Kettimuthu, G. Garzoglio, B. Holzman, C. Weiss, X. Duan, L. Lacinski, End-to-end solution for integrated workload and data management using glideinwms and globus online. J. Phys. Conf. Ser. **396**(3), 032076 (2012)
24. L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: distributed stream computing platform, in *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops*, Sydney, Australia, 13 Dec 2010, p. 170C177
25. Scala programming language, http://www.scala-lang.org
26. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI-The Complete Reference, vol. 1: The MPI Core, 2nd (revised) edn. (MIT Press, Cambridge 1998)
27. Spark programming model, http://blog.cloudera.com/blog/2013/11/-putting-spark-to-use-fast-in-memory-computing-for-your-big-data-applications
28. The structure of dryad jobs, http://research.microsoft.com/en-us/projects/dryad

29. M. Tim Jones, Process real-time big data with twitter storm. Technical Report pp. 1-9, IBM Developer Works (2013)
30. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D.V. Ryaboy, Storm@twitter, in *International Conference on Management of Data, SIGMOD 2014*, Snowbird, UT, USA, June 22–27, 2014, p. 147C156
31. Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Kumar Gunda, J. Currey, Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language, in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, San Diego, California, USA, Proceedings (2008), p. 1C14
32. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in *2nd USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud10, Boston, MA, USA (2010)
33. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: fault-tolerant streaming computation at scale, in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP 13*, Farmington, PA, USA (2013), p. 423C438