Big Data Landscape

Ge Song, Nicolas Narbais, ActiveEon

Executive Summary:

Introduction

Nowadays, we are awash in a flood of data. With a rapid growth of applications like social network analysis, semantic web analysis, bio-information analysis, AI, IoT analysis, a large amount of data needs to be processed to improve accuracy and create a meaning. For example: Google deals with more than 24 PB of data every day (a thousand times the volume of all the publications contained by the US National Library); Facebook updates more than 10 million photos and 3 billion "likes" per day; YouTube accommodates up to 800 million visitors monthly (a video more than an hour uploaded every second); and Twitter is almost doubling its size every year, etc.

In this environment flooded with data, or as it is called Big Data, the systems have four features:

- Volume: The amount of data is very large, usually in an order (such as TB and PB) that can not be stored and processed by a single machine. Therefore, big data processing and cloud computing technologies are closely linked.
- Variety: The data types are variety. There are low-dimensional data (such as geographic coordinates), and also high-dimensional data (such as audio, video, pictures, etc.) as well. The formats of data could also be different. So when processing big data, the model needs to adapt to different kinds of data, or transform them into a unified format.
- Value: Information is everywhere, but the density of data value is relatively low. So before embarking on a high-density computing, a preprocessing step needs to firstly be used to clean and extract information from the original data.
- Velocity: The timeliness is high, so the processing speed should also be high. Generally, recent data is more valuable, so old data should be withdrawn when new data arrives. At the same time, the system needs to respond in a real-time or quasi real-time manner.

Consequently Big Data processing represents diverse challenges at different time of the data journey. The data will go through a first set of analysis primarily focusing on real time information, a second set of application will then store, clean and sort the data. Finally, advanced algorithms will extract deeper and more valuable analysis. To organize all of these analysis and optimize parallelization, an orchestrator or meta-scheduler is required.

Parallel Data Processing

Parallel computing is a computation model which uses two or more processors (cores or computers) in combination to perform multiple operations concurrently. The basic condition for parallel computing is that in general, a large problem can be divided into a limited number of smaller problems, and these smaller problems can be handled simultaneously.

Unlike the traditional serial computation, parallel computing uses multiple resources simultaneously to solve a problem. The problem should first be cut into

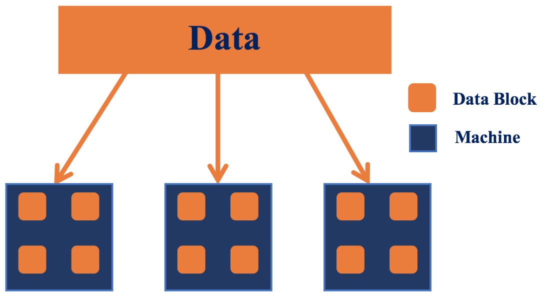

Figure 1: *Data Parallelism Schematic*

a series of instructions, which will later be executed simultaneously on different processors. This model is much more suitable for explaining, modeling and solving complex real world phenomena. It does not only speed up the time spent to perform a large task, but also makes it possible to process large-scale data sets or complex problems which cannot be handled by a single machine.

In parallel computing, there are mainly two forms of parallelism:

- Data Parallelism
- Task Parallelism

We will introduce them separately in the coming section.

Two types of parallelisms

Data parallelism focuses on distributing data across different parallel computing resources, in which the same computation is applied to multiple pieces of data. This is usually used for data-intensive tasks. Fig. 1 shows the schematic of data parallelism.

In a data parallelism process, the data set is first divided into partitions, which will later be processed by different processors using the same task. This simple idea makes the storing and handling of big data possible. For example, Facebook has several million photos uploaded each day. But these photos are too large to be stored in a single machine. Then a data parallelism strategy is suitable for this problem.

However, because each machine only has a subset of data, gathering the results together is a problem that this model needs to address. At the same time, the main factor affecting the performance of this model is the transmission of intermediate data, hence reducing the amount of data to be transferred is another problem to face. Since data parallelism emphasizes the parallel and distributed nature of data, when the size of data is growing, it is inevitable to use this model in parallel computing. Examples of Big Data frameworks that uses data parallelism are: Hadoop MapReduce [1], Apache Spark[2], YARN[3], and Apache Storm[4].

Task parallelism focuses on distributing tasks concretely performed by processors across different parallel computing resources, in which the same data (or maybe different data in a hybrid system) is processed by different tasks. This is usually used for computation-intensive tasks.

In a task parallelism process, the parallelism is organized around the functions to be run rather than around the data. It depends on task decomposition. This idea makes it possible to handle a complex problem.

The difficulties of this type of process lie first on the decomposition of the work, specifically the decomposition of queries in a join process. Also task parallelism processes usually suffer from bad load balancing, since it is not easy to divide tasks with equal complexity. The communication among tasks is another problem. Synchronization is the most important communication approach in task parallelism processes, and can be divided into thread synchronization and data synchronization. Thread synchronization focuses on determining the order of execution in order to avoid Data Race Condition problems. Data synchronization is mainly used to ensure the consistency among multiple copies of data.

Recently, the most popular task parallelism example is deep learning. Deep learning is a branch of machine learning which is based on a set of algorithms which attempt to model high-level abstractions in data by using multiple processing layers. The difference between deep learning and traditional machine learning is that in deep learning instead of having one model to train all the data, we separate the model into layers, and each layer can be considered as a sub-task of the whole model.

The most common combination of data parallelism and task parallelism is pipelining. Suppose you have multiple tasks, task I, task II and task III, instead of having each one operating on the data independently, pipelining takes the data and first give it to task I, then task II and finally task III. Image processing often chooses to use a pipeline. Images are coming in a stream, some of the processing starts with the first task, and applies a certain filter on the images, then passes on to the second task, and so on. An example of pipeline processing of images is shown in Fig. 2.

Task parallelism and data parallelism complement each other, and they are often used together to tackle large-scale data processing problems. The Big Data

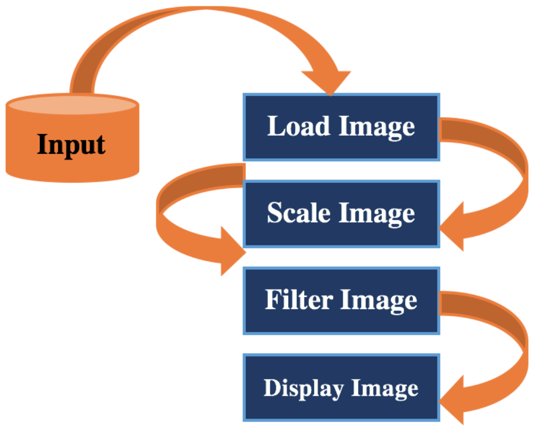

Figure 2: *Images Pipeline Processing*

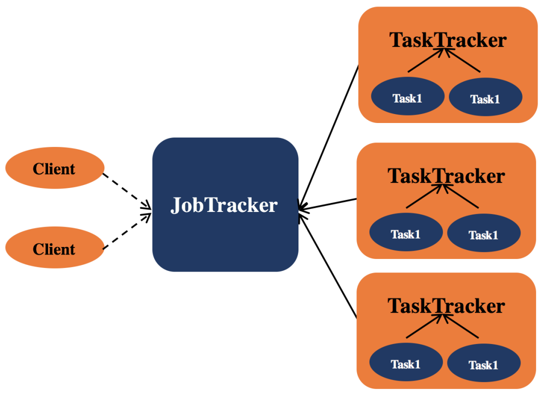

Figure 3: *Physical Structure of Hadoop System*

frameworks that use task parallelism are: Apache YARN [3] and Apache Storm [4]. They are hybrid systems which support both task and data parallelism.

Big Data processing systems

MapReduce is a flexible and scalable parallel and distributed programming paradigm which is specially designed for data-intensive processing. It was initially introduced by Google [5] and popularized by the Hadoop framework.

The concept of MapReduce has been widely known since 1995 with the message passing Interface (MPI) [6] standard, having *reduce* and *scatter* operations. The MapReduce programming model is composed of a Map procedure and a Reduce procedure. The Map task is usually used for performing some preliminary and cleaning work such as filtering and sorting. For example we can use a Map task to sort the students by alphabetical order of their surname, and then filter the students whose score is below a certain level. The Reduce task is used to perform a summary operation such as count or aggregation. For example we can use a Reduce task to count the number of students whose score is above a given level.

The idea of the MapReduce paradigm comes from high-order functional programming, where Map and Reduce are two primitives. In this paradigm every record is represented by a $< key, value >$ pair. The Map function processes a fragment of $< key, value >$ pairs in order to generate a list of intermediate $< key, value >$ pairs. Each $< key, value >$ pair is processed by the same map function on different machines without depending on other pairs. The output keys of the Map tasks could be either the same as the input keys or different from them. The output $< key, value >$ pairs have an information of partition which indicates to which Reduce task this pair needs to be sent. The partition information makes sure that all pairs with the same key can be later sent to the same Reduce task. The Reduce function gathers the outputs of the same partition from all map tasks together through a Shuffle phase and merges all the values associated with the same key, then produces a list of $< key, value >$ pairs as output.

Hadoop [2] is an open-source framework written in Java for distributed storing and processing large scale data sets. The core of Hadoop contains a distributed storage named Hadoop Distributed File System (HDFS), and the MapReduce programming paradigm. HDFS is a distributed, scalable, and portable file-system written in Java. It stores large files across multiple machines on a cluster. Its reliability is achieved by replicating the data among multiple nodes. The default number of replications is set to 3, which means the same piece of data is stored on three nodes. It is very popular not only among the academic institution but also in many companies such as web search, social network, economic computation and so on. A lot of research work focuses on optimizing Hadoop performance and its efficiency in many different aspects.

The whole system of Hadoop works in a master-slave manner, with JobTracker as the master, and the other nodes as slaves. A TaskTracker daemon runs on each slave node. The JobTracker daemon is responsible for resource allocation (e.g. managing the worker nodes), tracking (e.g. resource consumption or resource availability) and management (e.g. scheduling). The TaskTracker has much more simple responsibilities. It is in charge of launching tasks with an order decided by the JobTracker, and sending the task status information back to JobTracker periodically. The schematic of this process is shown in Fig. 3.

When running a Hadoop job, input data will first be divided into some splits (64M by default). Then each split will be processed by a user-defined map task.

So the whole process of a Hadoop job as shown in Fig. 4 can be summarized as follow:

- **Step 1:** Split data into blocks (64M by default)
- **Step 2:** Map Phase: Extract information from data (filter, sort)
- **Step 3:** Shuffle Phase: Exchange data through network from Map Phase to Reduce Phase
- **Step 4:** Reduce Phase: Summary operation (count, aggregation)

The previous introduction on Hadoop shows that a Hadoop MapReduce job has some special characteristics as shown below,

- **Execution Similarity:** According to the programming model, users only have to provide a map function and a reduce function. And the execution for each Map task (or Reduce task) is very similar to others. In other words, all data will be processed by these functions repeatedly. Thanks to this design, we only need to study how each $< key, value >$ pairs are processed for a particular job, as reading, sorting, transferring and writing data are independent of these two functions.
- **Data Similarity:** MapReduce is well suited for off-line batches processing. And it is usually used to do repeated work in which the input data has very similar format, such as log analysis, inverted index and so on. We can just take a look at a small sample and then we can estimate the whole dataset.

Hadoop is now a very mature system, with specific application and user groups. However, due to the limitation of the MapReduce paradigm and the Hadoop implementation, it has performance limitations in some application scenarios. In order to better integrate Hadoop in the applications, many works have been done from the very beginning to extend Hadoop and to improve its performance. We discuss a limited number of them.

Apache Spark [2] is another popular parallel computing framework after Hadoop MapReduce. Spark provides an application programming interface in Java, Scala, Python and R on a data structure called the resilient distributed dataset (RDD). Spark also uses the MapReduce paradigm but it overcomes the limitations in MapReduce. Hadoop MapReduce forces a particular linear data flow, it reads input data from disk, maps a function across the data, reduces the results of the map, and stores reduction results on disk. The resilient distributed dataset structure works as a working set for distributed programs, it offers a restricted form of distributed shared memory. Unlike Hadoop jobs, the intermediate data of Spark can be saved in memory, which avoids the unnecessary reading and writing from HDFS. Therefore Spark is better for data mining and machine learning algorithms, which require iterations. RDD facilitates the implementation of iterative algorithms which need to visit the dataset multiple times in a loop. It also makes it easy to do interactive or exploratory data analysis, like repeated database-style querying of data. Spark requires a manager which is in charge of the cluster, and a distributed file system. Spark also supports a pseudo distributed mode (local mode), which is usually needed for testing.

Compared with Hadoop, Spark has the following advantages:

- Store intermediate data into memory, providing a higher efficiency for iterative opera- tions. So Spark is more suitable for Data Mining and Machine Learning algorithms containing a lot of iterations.
- Spark is more flexible than Hadoop. It provides many operators like: map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, mapValues, sort, partionBy, while Hadoop only provides Map and Reduce. However, due to the char- acteristics of RDD, Spark does not perform well on the fine-grained asynchronous update applications or the applications with incremental changes, such as the web crawlers with updates.
- By providing a wealth of Scala, Java, Python APIs and interactive Shell API, Spark has a higher availability with different programming languages and different modes to use.

According to the characteristics of Spark, its applicable scenarios are:

- Iterative calculations requiring multiple operations
- Applications that require multiple operations on a specific data set

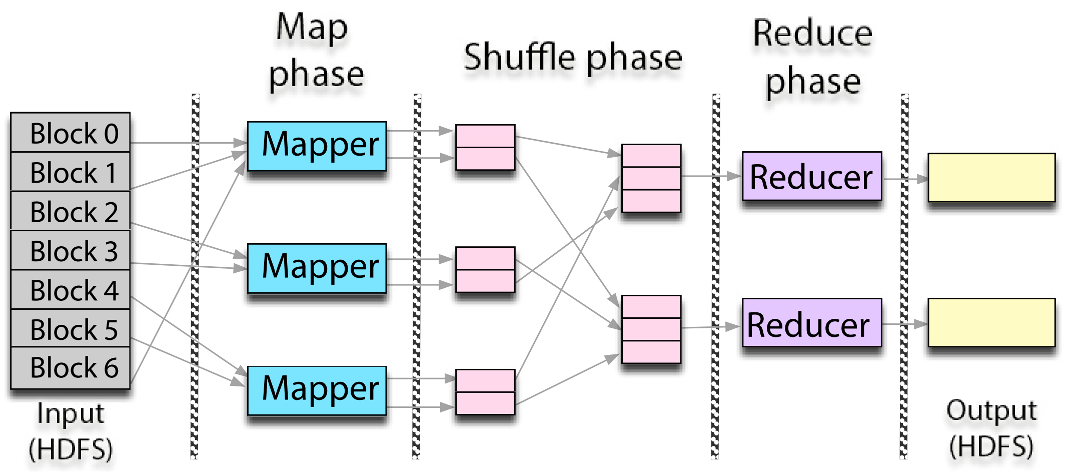

Figure 4: *Logical View of Hadoop Framework*

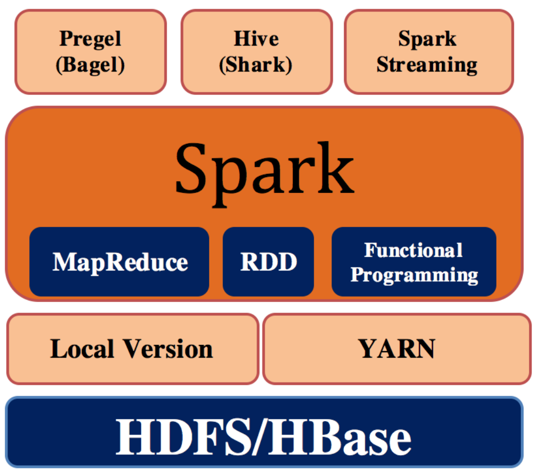

Figure 5: *Structure of Spark System.*

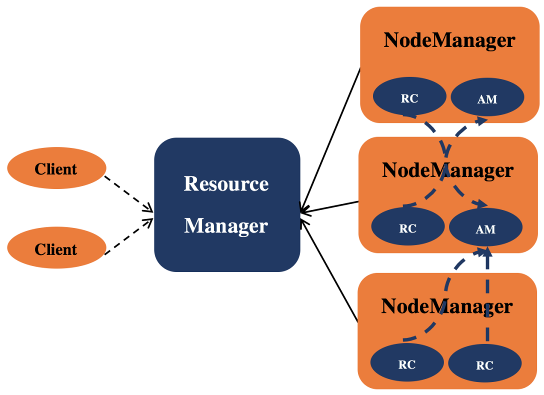

Figure 6: *Architectural View of YARN*

And the benefit increases with the amount of data and the number of operations. But the benefit is smaller in applications with a small amount of data and intensive computations. The structure of a Spark system is shown in Fig. 5.

YARN [3] is an attempt to take Apache Hadoop beyond MapReduce for data-processing. As we explained above, in Hadoop, the two major responsibilities of the JobTracker are resource management and job scheduling or monitoring. As there is only one JobTracker in the whole system, it becomes a bottleneck. The fundamental idea of YARN is to split these functions into two separate daemons — a global ResourceManager (RM) and a per-application ApplicationMaster (AM).

The ResourceManager together with the per-node slave daemon NodeManger forms a new generic system for managing tasks in a distributed manner. Moreover, the ResourceManager is the ultimate authority that arbitrates resources among all applications in the system, while the per-application ApplicationMaster is a framework specific entity and is used to negotiate resources from the master ResourceManager and the slaves NodeManagers to execute and monitor the tasks. A pluggable Scheduler is used in the ResourceManager to allocate resources to jobs. The Scheduler works using an abstract concept of Resource Container (RC) which incorporates resource elements such as CPU, Memory, Disk, Network etc. The NodeManager is a per-node slave daemon, and its responsibility is to launch the tasks and to monitor the resources (CPU, Memory, Disk, Network). From the system perspective, the ApplicationMaster runs as a normal container. An architectural view of YARN is shown in Fig. 6.

Hadoop, **Spark** and **YARN** all use the **MapReduce** paradigm as their abstract computational concept. The ecosystem of MapReduce and its derivative methods are mature and very good for parallel processing of big data. But most of them are still an ‘offline’ processing platform, which means that they can not handle dynamic data streams.

Parallel data warehouse

Pivotal Greenplum Database [7] is an Open Source parallel database. It is used for fast parallel data loading or unloading and fast internal data transfer of operations such as CTAS (Create Table as Select). It is a MPP (Massively Parallel Processing) database built on PostgreSQL (which is an open souce object oriented database). Green plum supports ACID principles of transaction management for concurrent data access and modification.

The system of Greenplum consists of a master node, standby master node, and segement nodes. The master nodes are used to store the catalog information. And all of the data are stored on the segments nodes. One or more segments can be run on the segment nodes, which is modified PostgreSQL database instances and are assigned a content identifier. For each table, the data is divided among the segment nodes based on the distribution column keys sepcified by the user in the DDL statement. For each segment content identifier, there are a primary segment and a mirror segment, and they are not run on the same physical host. The master node is used to parse, optimize and dispatch the SQL queries to all of the segments. The segments are used to execute the query plan and either return the requested data or insert the result of the query into a table of the database.

The segment nodes support both Bulk loading and unload. It can read and write data from external of the Greenplum cluster such as ETL nodes, flat files or HDFS.

ETL for Big Data

As a reminder, ETL objective is to ease the process of moving and transforming data. It is split into three functions: data extraction from one or multiple sources, data transformation (cleaning, sorting, standardizing, aggregating, etc.), data loading in targeted systems following the right format. These process are used regularly so that the targeted systems are up to date and analysis can be performed. However, with the growth of data produced by businesses, data latency, the mix of data movement and transformation, these systems become companies bottleneck. As a close comparison, tools such as Hadoop focus on real time processing and making data available to users as soon as possible. The data never leaves Hadoop and this real time objective has been included in the development from the beginning. However, with Hadoop, the management of the entire process can become difficult since it relies on heavy interactions with the whole IT infrastructure. Workload automation becomes required to meet business expectations. However, ETL makes sense in the Big Data world to pre-compute data and enable end-users to easily perform analysis. For instance, in the NoSQL world, the same event can be stored in multiple places due to different level of granularity and different information required. ETL will then be in charge of creating additional views for end users (people or software).

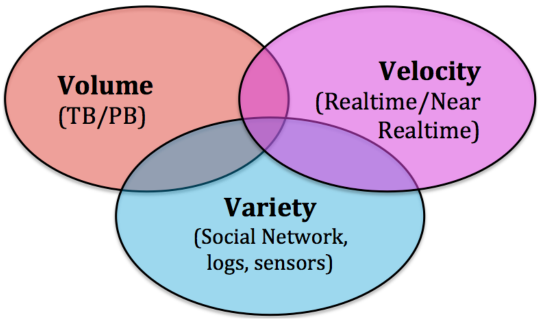

Figure 7: *The 3 Vs of Big Data*

Real-time Data Processing

As we introduced in the Introduction Section, the characteristics of Big Data contains 4 Vs, among which Volume, Variety and Velocity are the most important. The relations among these 3 Vs are shown in Fig. 7.

The need of velocity requires to process the data fast, so that the system can react to the changing conditions in real time. This requirement of processing high-volume data streams with low-latency becomes increasingly important in the areas of trading, fraud detection, network monitoring, and many other aspects, thus increasing the demand for stream processing. Under this requirement, the capacity of processing big volumes of data is not enough, we also need to react as fast as possible to the update of data.

Stream processing is a programming paradigm, which is also called dataflow programming or reactive programming. A stream is a sequence of data and a series of operations will be applied to each element in the stream. Data items in streams are volatile, they are discarded after some time. Since stream processing often involves large amount of data, and requires the results in real-time, the stream processing platforms (e.g. Apache Storm) often work in parallel. Besides, this paradigm is a good complement of parallel processing, and allows applications to more easily exploit a limited form of parallel processing.

Traditional popular big data processing frameworks like Hadoop and Spark assume that they are processing data from a data storage that all data is available when it is needed. And it may require several passes over a static, archived data. But when data arrives in a stream or streams, data will be lost if it is not processed immediately. Moreover, in the streaming scenarios, usually the data arrives so rapidly that it is not feasible to store it all in a conventional database, to process be processed when needed. So stream processing algorithms often rely on concise, approximate synopses of the input streams which can be computed with a simple pass over the streaming data.

In the coming sections of this part, we will first introduce the rules for processing Data Streams, then we will present several Data Stream management systems, and a detailed introduction about Apache Storm can be found in the end.

Rules in Data Stream processing

A **data stream** is a real-time, continuous, ordered (explicitly by timestamp or implicitly by arrival time) sequence of items.

In general, we need to follow some rules for processing low-latency and high-volume data streams. The most important rules are:

- **Rule 1: Keep the data moving**
  The first requirement for a real-time high-volume data stream processing framework is to process data "on the fly", without storing everything.
- **Rule 2: Handle Stream Imperfections**
  The second requirement is to provide resilience results against "imperfections" in streams, including delay, missing and out-of-order of data.
- **Rule 3: Generate Predictable Outcomes**
  A stream processing engine must guarantee predictable and repeatable outcomes.
- **Rule 4: Integrate Stored and Streaming Data**
  A stream processing system also needs to efficiently store, access, and modify state information, and combine it with new coming streaming data.
- **Rule 5: Guarantee Data Safety and Availability**
  Fault-tolerance is another important point for such a system.
- **Rule 6: Partition and Scale Applications Automatically**
  In order to meet the real-time requirement for high-volume and fast-changing data streams, the capability to distribute processings across multiple machines to achieve incremental scalability is also important. Ideally, the system should automatically and continuously distribute the data and queries.
- **Rule 7: Process and Respond Instantaneously**
  The last but most important requirement is to have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

When designing a stream processing algorithm, we need to keep two things in mind:

- It can be more efficient to get an approximate answer than an exact solution.
- A variety of techniques related to hashing turn out to be very useful. Because, these techniques introduce useful randomness, which produces an approximate answer that is very close to the exact one.

The approximation of a streaming algorithm comes from two aspects: (1) from limiting the size of states maintained for the process; (2) from reducing the precision of the result.

The algorithms for processing streams usually involve summarization of the stream in some ways. Summary data structures such as : wavelets, sketches, histograms and samples have been widely used especially for streaming aggregation. These algorithms always begin by considering how to make a useful sample or how to filter out most of the undesirable elements. Another important approach to summarize a stream is to process within a fixed-length window, then query the window as if it were a relation in a database. This model is called "Sliding Window Model" [8].

A lot of prior work on stream processing focused on developing space-efficient, one-pass algorithms for performing a wide range of centralized, one-shot computations over massive streams. These applications involve: (1) computing quantiles; (2) estimating distinct values; (3) counting frequent elements; (4) estimating join sizes and stream norms.

As the size of data is getting larger, some recent efforts have concentrated on distributed stream processing and proposing communication efficient streaming frameworks to handle a number of query tasks such as aggregation, quantiles and join (such as Apache Storm, Spark Streaming, Yahoo! S4 etc.) which will be introduced in the next Section.

Data Stream management systems

There are many efforts towards building a Data Stream processing system. Some works have extended the MapReduce paradigm to process dynamic data. But these frameworks are not suitable for processing data streams and returning the results in real-time because of the nature of MapReduce paradigm.

To address the limitation, some stream processing engines have been proposed, some of them are centralized which are not able to process a huge volume of data streams. Others are parallel. Among all the parallel frameworks, **Spark Streaming**, **Yahoo! S4** and **Twitter Storm** are the most widely used. We will introduce them separately.

Spark Streaming [9] is an extension of the Spark framework. It enables scalable, high-throughput and fault-tolerant stream processing for dynamic data streams. Spark Streaming receives input data streams and divides them into batches, which will later be processed by the Spark engine to generate final stream in batches. It has APIs for Scala, Java and Python. Spark Streaming is a data parallelism framework. It follows the same ideas as MapReduce batch processing paradigm. It is not a real "real-time processing" framework: the incoming events are cached and processed as a batch, resulting in a larger delay than the real streaming processing frameworks such as Twitter Storm.

Yahoo! S4 [10] is a general-purpose, distributed, scalable, partially faut-tolerant, pluggable platform for data stream processing that allows users to easily develop applications for processing continuous unbounded streams of data. It is a java based solution which relies on the user defined classes to process and produce stream data. It uses Apache Zookeeper [11] to maintain the state of a distributed job. It allows a parallel execution of data streams. But it does not provide a dynamic load balancing protocol, which is left for users to define. S4 uses "Plain Old Java Objects" (POJO) mode as its communication protocol and User Datagram Protocol (UDP) as its underlying protocol, which has an impact on reliability. Besides, it does not support dynamic deployment of the cluster or add or delete nodes during run time.

Twitter Storm [4], is a distributed, parallel and fault tolerance framework for data streams processing. Queries can be expressed in using the boxes and arrows model. A Storm job, which is called a Topology, consists in two components: Spout and Bolt . Spout nodes are responsible for generating the system input streams. Bolt nodes are in charge of processing those streams and generate output results. It relies on Zookeeper servers to maintain the state of distributed setups. Storm supports task parallelism. In a Storm cluster, the data is flowing while the tasks do not. Storm is a real "flow processing" framework: every incoming data will be handled as an event, which has a smaller delay than the mini-batch processing frameworks such as Spark Streaming. It also supports dynamic deployment of the cluster, and add or delete nodes during run time.

Table. 1 shows a comparison of these three frameworks.

Through the above comparison, we think Storm is the best choice for a real-time and low latency data stream processing problem. The reasons are:

- Storm processes data in form of streams. And it processes intensive queries in parallel. (Latency)
- It is scalable. As the amount of data increases, we can simply increase the number of nodes for processing. (Dynamic Deployment)
- It has a good reliability, which can ensure that each event can be at least processed once. (Fault Tolerance)
- It provides good fault tolerance. Once a node fails, the task on this node is re-assigned to the other nodes. (Fault Tolerance)
- It provides a simple programming model, which reduces the complexity of real-time processing. (Distributing Work)
- It supports multiple programming languages such as Clojure, Java, Ruby and Python. (APIs)

We will give a detailed introduction about Apache Storm in the coming section.

Introduction to Apache Storm

Storm is a distributed, reliable, fault-tolerant framework for data streams processing. It was firstly developed in Twitter, and is now an Apache open source project.

A Storm job is called a **Topology**. A Topology is made of different types of components. Each component is responsible for a simple specific task. The

Table 1: *A comparison of Storm, Spark Streaming and Yahoo! S4*

	Storm	**Spark Streaming**	**S4**
Original Design	Twitter	UC Berkeley	Yahoo!
Implemented In	Clojure	Scala	Java
APIs	Java (and others)	Scala, Java	Java
Processing Model	Record at a time	Mini Batches	Record at a time
Latency	Sub-Second	Few Seconds	Sub-Second
Data Units	Tuple	Java Object	Java Object
Hadoop Distribution	Hortonworks HDP	Cloudera, MapR	None
Resource Manager	Mesos/Zookeeper	YARN/Mesos	Zookeeper
Distributing Work	User Specifies	MapReduce	Evenly
Fault Tolerance (Ever Record Processed:)	At Least Once	Exactly Once	No Guarentee
Dynamic Deployment	Yes	No	No

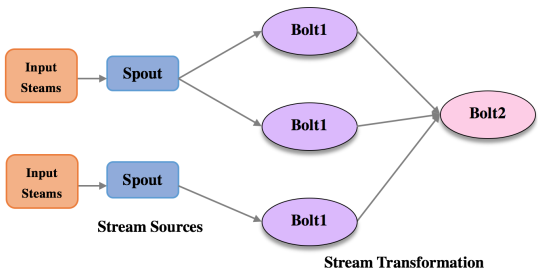

Figure 8: *An example of a Topology*

component for handling and distributing the input streams is called a **Spout**. The component for processing the streams is called a **Bolt**. A Spout passes data to Bolt(s), which transforms it in some ways. A Bolt is a user defined task, it can either persist the data into different sort of storages, or pass it to some other Bolts, or process it through some user specified functions. A Storm cluster can be seen as a chain of a Spout followed by several Bolts, where each Bolt makes some kind of transformation on the data. An example of a Topology is shown in Fig. 8.

A Storm cluster works in a master slave manner. A daemon called Nimbus runs on the Master node. It is responsible for distributing the user specified code to the cluster: assigning tasks to worker nodes, and monitoring the cluster for failures. A daemon called Supervisor runs on the Worker nodes. It executes the task on this node and is part of the Topology. Usually, a Topology runs across many worker nodes. Storm keeps the states of the cluster in Zookeeper

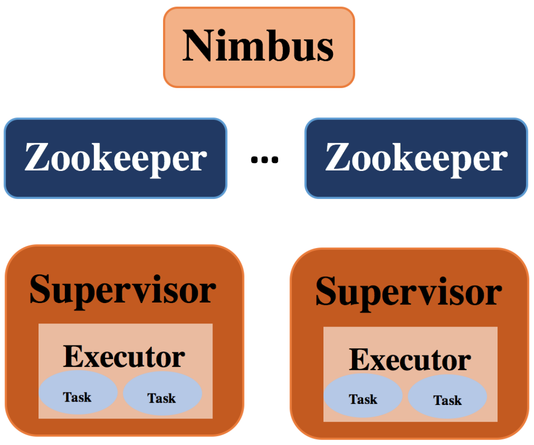

Figure 9: *A physical view of a Storm cluster*

or on local disk. So the daemons are stateless. They can fail or restart without affecting the health of the whole system. The physical view of a Storm cluster is shown in Fig. 9

ProActive

ProActive Parallel Suite is an open-source software for enterprise workload orchestration. A workflow model allows to define the set of executables and scripts written in any scripting language along with their dependencies, so ProActive Parallel Suite can schedule and orchestrate executions while optimising the use of computational resources.

Introduction to ProActive

Activeeon [12] offers IT automation, scalability and digital transformation solutions. Our fields of expertise include cloud migration projects, IT transformation, Big Data & IoT. We translate business processes into expressive computing workflows thanks to a flexible open source core. Job parallelization and distribution allows us to accelerate business processes and reduce infrastructure costs.

Role of ProActive in Big Data area

In this Big Data ecosystem, ActiveEon through its ProActive [13] open source solution fits into two main areas.

ProActive has proven records in algorithm optimization through distribution and parallelization which makes it suitable for long and complex analysis. By managing closely the diversity of resources available for a company and by understanding algorithm dependencies and requirements, businesses are getting insights into their data faster, at a cheaper cost while keeping control of the execution. R and Python which are the most common languages used to extract deeper information from the data are supported by the solution.

ProActive has also been used as a meta-scheduler and orchestrator for complex architecture which have to balance security rules, fast processing, access of information, governance, and need to interact with third party software. Moreover, the open source approach followed by ActiveEon could be seen in the flexibility offered by the solution which enables easy integration with existing IT architectures.

Advantages of using ProActive

References

[1] Hadoop: http://hadoop.apache.org.

[2] Spark: http://spark.apache.org.

[3] YARN: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn.html.

[4] Storm: http://storm.apache.org.

[5] MapReduce: https://en.wikipedia.org/wiki/mapreduce.

[6] Open MPI: https://en.wikipedia.org/wiki/open_mpi.

[7] Greenplum: http://greenplum.org/.

[8] Sliding Window Protocol: https://en.wikipedia.org/wiki/sliding_window_protocol.

[9] Spark Streaming: https://www.cloudera.com/products/apache-hadoop/impala.html.

[10] Yahoo S4: http://incubator.apache.org/s4/.

[11] Apache Zookeeper: https://zookeeper.apache.org/.

[12] Activeeon: http://www.activeeon.com/.

[13] ProActive: http://proactive.activeeon.com/.