Big Data Landscape

Ge Song, Nicolas Narbais, *ActiveEon*

Executive Summary:

Introduction

Challenges of Big Data processing

Real-time processing for data streams

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 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.

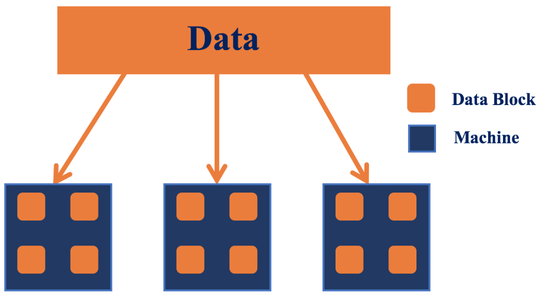

Figure 1: *Data Parallelism Schematic*

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

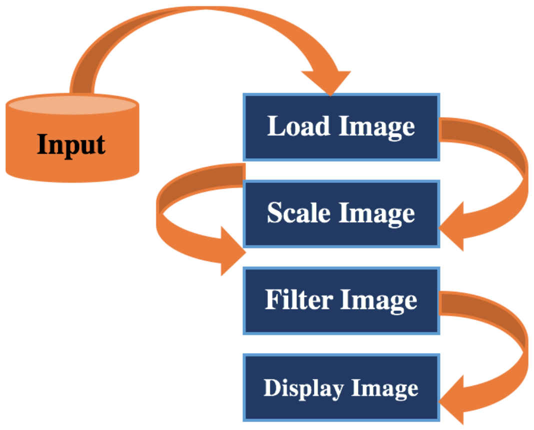

Figure 2: *Images Pipeline Processing*

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

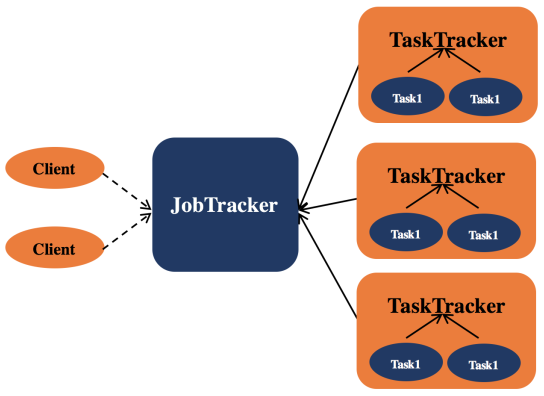

Figure 3: *Physical Structure of Hadoop System*

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.

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

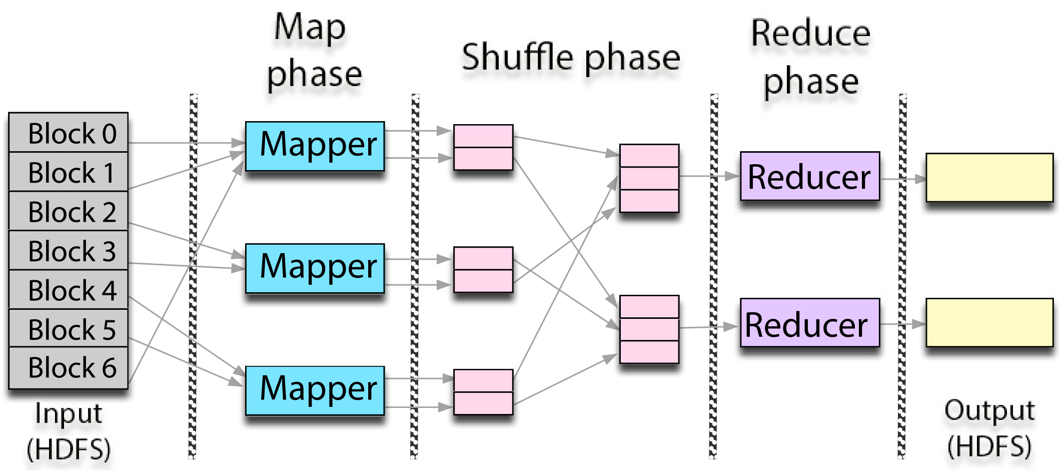

Figure 4: *Logical View of Hadoop Framework*

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. 2.3. 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. 2.4 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 3:** 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, mainly through the following three ways:

- Improve the performance of Hadoop by predicting its performance and tuning the parameters.
- Extend Hadoop to have database-like operations.
- Combine with other programming language or model.

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

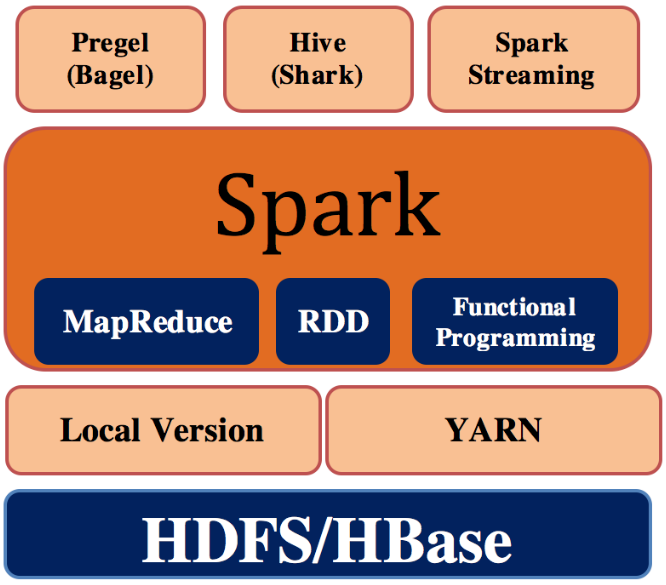

Figure 5: *Structure of Spark System.*

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.

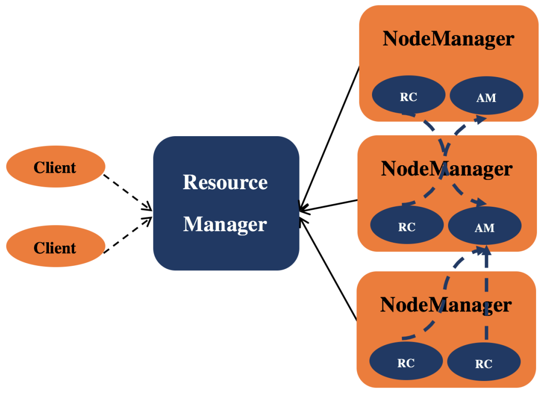

Figure 6: *Architectural View of YARN*

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

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

ETL for Big Data

Real-time Data Processing

Rules for Data Stream processing

Data Stream management systems

Introduction to Apache Storm

ProActive

Introduction to ProActive

Role of ProActive in Big Data area

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] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, 2004.

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