Spark vs. Hadoop MapReduce

Siyu Nan
University of Rochester
Rochester,NY
snan@u.rochester.edu

Zhongda Su
University of Rochester
Rochester,NY
zsu4@ur.rochester.edu

ABSTRACT
Spark, an open-source cluster-computing framework came out in 2014, provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

Being hype in recent distributed computing field, Spark is being taken as an improvement of MapReduce cluster computing paradigm. We will focus on the Apache Spark cluster computing framework, an important contender of Hadoop MapReduce in the Big Data Arena. Spark provides great performance advantages over Hadoop MapReduce, especially for iterative algorithms, thanks to in-memory caching. Also, gives Data Scientists an easier way to write their analysis pipeline in Python and Scala, even providing interactive shells to play live with data.

This research intends to compare these two distributed computing framework. What are their strengths and weaknesses? What are their unique characteristics respectively? Can Spark potentially replace Hadoop?

1. INTRODUCTION
The Big data is becoming more and more popular in recent days. Big data has created countless opportunities lots of fields including business, medical, insurance and other fields. Technically, big data refers to the datasets with a size ranges from terabytes to exabytes. It also has three characters: large volume, high dimensions and dramatically large varieties [1].

The technology about solving big data problems in processing data is urged to develop. In this paper, we will trace the MapReduce, Hadoop and Spark revolution and understand the differences between them.

2. MapReduce and Hadoop
MapReduce is a programming model used for processing large data sets, which can be automatically parallelized and implemented on a large cluster of machines. It is also easy to use even for programmers without professional experience in parallel and distributed systems [2].

2.1 Programming Model
The basic idea of programming model is inspired by the map and reduce in functional languages. Map can produce a set of intermediate key or value pairs by taking an input pair. The MapReduce library groups values by keys. The reduce function accepts an intermediate key I and values for that key, and it can merge these values to form a possibly smaller set of values. Here is an example of counting the number of occurrences of each word in a large collection of documents. User can define map and reduce functions similar to the following pseudo-code:

```
map (String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
     EmitIntermediate (w, "1")
Reduce (String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
        result += ParseInt(v) ;
   Emit (AsString (result) );    [2]
```

2.2 Implementation
First, the user split inputs into M chunks by splitter function and create M map jobs and R reduce jobs. One of these jobs is special and we call it the master. Map task is about reading the contents of the corresponding input split and do the Map function. Periodically, the map workers will write the data to local disk. The reduce worker will do reduce function when the master call it. Finally, when all tasks are finished, the master will call the user. Figure 1 is an overview of the execution flow [2].

There are also some interesting problems which can be better solved by using MapReduce. For example:

Distributed Sort: We can use map function to emit a (key, record) pair by extracting the key. After that we can use reduce function to emits all pairs unchanged.

Inverted Index: The map function can emit a sequence of (word, document) by parsing each document. The reduce function can combine all the sequences and sort the corresponding document IDs, then emit a (word, list (document ID)) pair [2].

So far, we only talked about the basic idea of how MapReduce works. To implement MapReduce in real problems, we also need to consider failure of each part of the whole task. We need to

consider worker failure, master failure and semantics in the presence of failure. Furthermore, we will introduce an open source software for distributed computing.

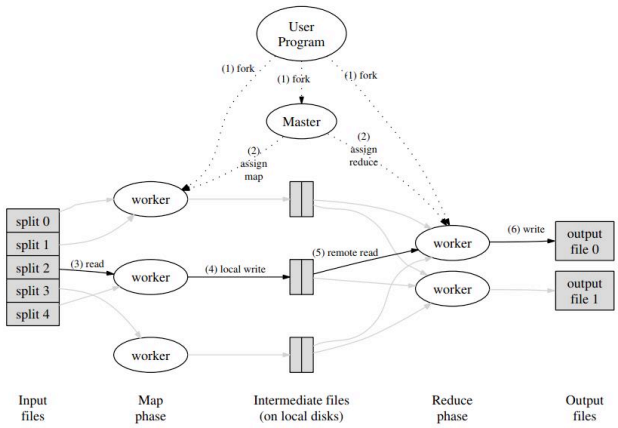

Figure 1: Execution overview [2]

2.3 Apache Hadoop

Apache Hadoop is an open-source software alternative to Google's MapReduce system and they are very similar in many respects. [3]

It is built on YARN system for job allocation and resource management. It uses Java as its default language. All the files are kept in HDFS (Hadoop Distributed File System)

Above are the key features of Hadoop but it also has several other modules. These include Pig, Ambari, Avro, Cassandra, Hive, Pig, Oozie, Flueme, and Sqoop. Pig is most use for data extraction, transformation, loading (ETL). Apache Sqoop are one tool designed for efficiently transferring bulk data between Apache Hadoop and structured databases. Apache Hive is similar to SQL language used for querying and management in HDFS. Oozie is a workflow scheduler system to manage jobs. Zookeeper can provide operational services for a Hadoop cluster group services. Flume is used for collecting, aggregating and moving large amounts of log data. Tez is a generalized data-flow programming framework, which can provide flexible functions to implement different tasks [4].

It is also worth to note what Hadoop is not. First, Hadoop is not a substitute for a database. It stores data but not index them. If you want to find something you need to first to generate index and it is very expensive to regenerating indexes. Second Hadoop is not a good place to learn Java programming since MapReduce is very complex and you should learn from simple [4].

3. Spark

3.1 Spark Introduction

Apache Spark, a fast and general engine for large-scale data processing, was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project. [5] In Hadoop, Spark is taken as a module. With this being said, Spark is smaller compared with Hadoop system. Other modules in Hadoop, as mentioned above, includes Hadoop Distributed File System, Hadoop MapReduce, Hadoop YARN and Hadoop Common etc.

3.2 Spark Components

Spark have six main components, Spark core, SparkSQL, Spark Streaming, MLlib (for machine learning), GraphX and Standalone Scheduler. SparkSQL, Spark Streaming, MLlib and GraphX act as high-level libraries complementing Spark Core. Standalone Scheduler acts as Spark's own cluster manager. Spark also supports Hadoop YARN and Mesos cluster to act as cluster manager for applications to run on.

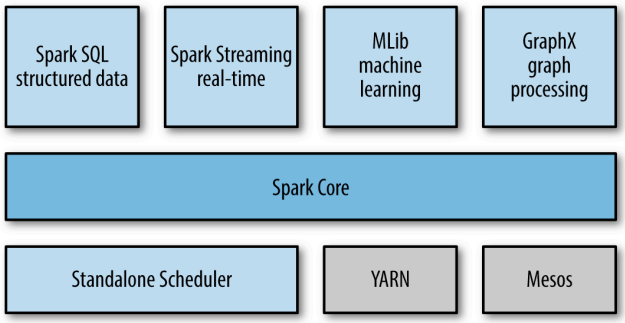

Figure 2: Component of Spark [6]

3.2.1 Spark Core

Spark acts as the base engine for large-scale parallel and distributed data processing. Its functions include:

- memory management and fault recovery
- scheduling, distributing and monitoring jobs on a cluster
- interacting with storage systems

API that defines resilient distributed datasets (RDDs), which are Spark's main programming abstraction is also located in Spark Core. More about RDD is discussed in session 3.3.

3.2.2 Spark SQL

SparkSQL as a main component, supports querying data either via SQL or via the Hive Query Language. It is now integrated with the Spark stack which used to be as the Apache Hive port to run on top of Spark. Apart from providing support for different data sources, it makes it possible to weave SQL queries with code transformations.

3.2.3 Spark Streaming

Spark Streaming supports real time processing of streaming data, such as social media like Twitter and various messaging queues. The Processes of Spark Streaming include receives the input data streams and divides the data into batches followed by get processed by the Spark engine and generate final stream of results in batches.

3.2.4 MLib

MLlib is the library in Spark containing common machine learning (ML) functionality. MLlib includes all basic types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering. Moreover, it supports model evaluation and data import. It also provides some lower-level ML primitives, including optimization algorithm like basic

gradient descent and stochastic gradient descent. All of these methods are designed to scale horizontally across a cluster.

3.2.5 GraphX

GraphX is a library for manipulating graphs, for example, an American domestic flight graph, and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX expands the Spark RDD API, letting us to create a directed graph with properties assigned by programmer, attached to each vertex and edge. GraphX also provides various operators for graphs manipulation like subgraph and a library of common graph algorithms like triangle counting.

3.3 RDD

RDD is Spark's main programming abstraction. It is an immutable fault-tolerant, distributed collection of objects that can be manipulated in parallel. An RDD can have any type of object and is formed by loading an external dataset or distributing a collection from the driver program.

RDDs support two types of operations:

- Transformations: operations such as map, filter, join, union etc. which performed on an RDD and yield a new RDD containing the result.
- Actions: operations such as reduce, count, first etc. that return a value after running a computation on an RDD.

It is worthy to note that transformations in Spark do not compute their results right away. They just "remember" the operation to be performed and the its respective dataset. The transformations are only computed when an action is called and the output is then returned to the driver program. This design enables Spark to run more efficiently.

3.4 Features

It is important to note that Spark has no file management. Thus, it runs on top of existing Hadoop Distributed File System (HDFS) to provide advanced functionality.

Spark is outstanding because of its features such as provides a faster and more general data processing platform. As well as ease of use since makes it possible to write code more quickly with over 80 high-level operators. Some other important features include:

- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries helping the optimization of the overall data processing workflow.
- APIs in Scala, Java and Python.
- Provides interactive shell for Scala and Python.
- Works and Integrates well with the Hadoop ecosystem and data sources (HDFS, Amazon S3, Hive, HBase, Cassandra, etc.)

4. Comparison of Spark and Hadoop

Hadoop has been around longer than Spark as a big data processing technology. It has proven to be a great tool processing large data sets. However, MapReduce shows its advantage for one-pass computations whereas less efficient for multi-pass computations and algorithms. Map and Reduce are its only two phases which force each step in the data processing workflow into one Map phase and one Reduce phase. Users need to convert any use case into MapReduce pattern.

Moreover, all job output data from one step to the next, has to be stored in the distributed file system before the next step begin. Therefore, with replication and disk storage issues, Hadoop tends to be slow. Also, Hadoop solutions normally include clusters that are hard to set up and manage. It also requires the integration of several tools for different big data use such as machine learning.

If you wanted to do something complicated, you would have to arrange a series of MapReduce jobs and execute them in order. No later jobs could start until the previous job had finished completely.

In Spark, software engineer would be able to develop complicated, multi-step data pipelines using directed acyclic graph (DAG) pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data. This increases efficiency to a great extent.

Spark is famous for its speed and ease of use. Different from Hadoop's disk-based processing, Spark is an in-memory processing engine. It does allow use of disk but only for data doesn't fit into memory. This quality itself makes Spark extraordinary in its speed. Other than the in-memory quality, the structure itself also makes it performs better than Spark. Winer of Daytona GraySort Contest in 2014 did sorting on disk (HDFS), without using Spark's in-memory cache. The team sorted 100 TB of data on disk in 23 minutes, compared with the previous world record set by Hadoop MapReduce used 2100 machines and took 72 minutes.[7] That is to say, as shown in Figure 1, Spark sorted the same data 3 times faster using 10 times fewer machines. This credits mainly to Spark's RDD Resilient Distributed Dataset.

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Figure 3: Comparison of performance [7]

5. Conclusion

Spark performs better than Hadoop in many aspect as discussed above. First of all, with its in-memory quality, it requires less disk space, thus less expensive than Hadoop in general. Secondly, it's easy to use with its over 80 higher level operators comparing to Hadoop's only map-reduce phase. Last but not the least, Spark supports in-memory data sharing across DAGs, so that different

jobs can work together with the same data comparing with Hadoop's storing output data from one step to the next in the distributed file system before the next step begin. This also contributes to Spark's high speed. However, we can't say Spark is a potential replacement of Hadoop. They should be described as having a symbiotic relationship with each other. Hadoop has its own features that Spark does not possess, such as a distributed file system. Together, they contribute to better efficiency and greater user experience for big data computing.

6. ACKNOWLEDGMENTS

We appreciate Course Instructor Tamal Tanu Biswas for teaching us about database system design. We also want to thank course TA Vojtech Aschenbrenner and Md Kamrul Hasan for inspiring us this topic about Hadoop and Spark.

7. REFERENCES

[1] Ramez Elmasri, Shamkant B. Navathe, "7th Edition, Fundamentals of Database Systems", 2016, 911- 953. Print

[2] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, 6th Symposium on Operating Systems Design and Implementation, 2004

[3] Michael L. Scott, Parallel and Distributed Systems course notes, April 2017

[4] Welcome to Apache™ Hadoop Retrieved April 23, 2017, from http://hadoop.apache.org/

[5] Big Data Processing with Apache Spark – Part 1: Introduction. (n.d.). Retrieved April 23, 2017, from https://www.infoq.com/articles/apache-spark-introduction

[6] Tamal Tanu Biswas, University of Rochester, Database Systems, lecture note 14, Spring 2017

[7] Apache Spark officially sets a new record in large-scale sorting. (2016, October 27). Retrieved April 23, 2017, from https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html