PERFORMANCE COMPARISON BY RUNNING BENCHMARKS ON HADOOP, SPARK, AND HAMR

by

Lu Liu

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Fall 2015

PERFORMANCE COMPARISON BY RUNNING BENCHMARKS ON HADOOP, SPARK, AND HAMR

by

Lu Liu

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor Prof. Guang R. Gao for the guide of my study and master thesis, for giving me the opportunity to study in CAPSL, for his patience, motivation, enthusiasm, and immense knowledge.

I would thank Jose Monsalve, Long Zheng and Sergio Pino who were my mentors. Before I started the research, I had no experience on Linux and benchmarking. Long Zheng was my first mentor who gave me a training on how to install the distributed system on the cluster. I had a very struggling time at the beginning my research about setting up the Hadoop, Spark and HAMR clusters and the configurations of them. Sergio and Jose gave me so many supports. Sergio was always patient to answer my questions and gave me guidance about how to figure out these problems by myself. He also helped me to revise my thesis. Jose was in the same office with me and gave me hands-on help very timely. I could not accomplish my thesis without their help.

I would thank Haitao Wei who gave me so much help about writing and revising my thesis. He spent a lot of time to have meetings with me, and gave me suggestions about me paper. He encouraged me in my hard times. He is not only a mentor but also a friend of me.

I also want to thank Yao Wu. His previous work on HAMR gives me a deeper understanding for the system. Figure in Chapter 4 are drawn by him.

I want to thank all CAPSL members. To study and work with them is a great experience for me.

Finally, I want to thank my parents. They give me a great chance and supports to study in the U.S..

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Today, Big Data is a hot topic both in industrial and academic fields. Hadoop is developed as a solution to Big Data. It provides reliable, scalable, fault-tolerance and efficient service for large scale data processing based on HDFS and MapReduce. HDFS stands for the Hadoop distributed file system and provides the distributed storage for the system. MapReduce provides the distributed processing for Hadoop. However, MapReduce is not suitable for all classes of applications. An alternative to overcome the limitation of Hadoop is new in-memory runtime systems such as Spark, that is designed to support applications reuse a working set of data across multiple parallel operations [31]. The weakness of Spark is that the performance is restricted by the memory. HAMR is a new technology that runs faster than Hadoop and Spark with less memory and CPU consumptions.

At the time I started this thesis, CAPSL didn't have a platform to provide students an environment to test big data applications. The purpose of the thesis is not to perform an extensive research but to construct a main eco-system that Hadoop and Spark can be in a same working condition. In additional, HAMR has also been installed as a test platform in the research eco-system. I also engaged the work of a selected of big data benchmarks, and took a preliminary test in all three eco-systems.

To stress the different aspects of three big data runtimes, we selected and ran PageRank, WordCount, Sort, TeraSort, K-means and Naive Bayes benchmarks on Hadoop and Spark runtime systems, and ran PageRank and WordCount on HAMR runtime system. We measured the running time, maximum and average memory and CPU usage, the throughput to compare the performances difference among these platforms for the six benchmarks. As result, we found Spark has a outstanding performance on machine learning applications including K-means and Naive Bayes. For PageRank, Spark runs faster with small input size. Spark is faster on WordCount. For Sort and TeraSort, Spark runs faster with large input. However, Spark consumes more memory capacity and the performance for Spark is restricted by the memory. HAMR is faster than Hadoop for both two benchmarks with improvements on CPU and memory usage.

Chapter 1

INTRODUCTION

Data now is generated from everywhere: from smart phones; from social medias; from the ecommerce and credit cards; from transportation; from wireless sensor monitoring systems; from industrial productions; and from scientific and engineering computing. The following data was posted by DOMO in the Data Never Sleeps 3.0 [20] in 2015 that in every minute: Facebook users posted over 4,166,667 likes; Instagram users liked 1,736,111 photos; Twitter users sent 347,222 tweets; Skype users made 110,040 calls; Apple users downloaded 51,000 apps. All these big numbers have referred people to a hot topic today - the Big Data.

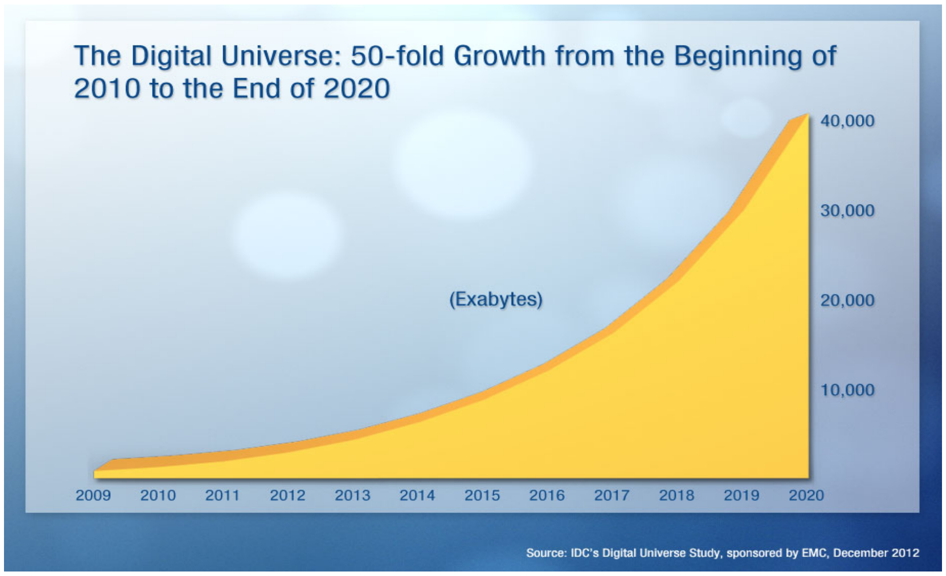


Figure 1.1: Data Growth from 2009 to 2020 [11]

Data is growing explosively every minute and it is not going to slow down. In 2012, Google received over 2,000,000 [18] search queries every minute but in 2014 that number was doubled to 4,000,000 [19]. As shown in Figure 1.1, published by IDC in 2012 [11], data was exponential growing and expected to reach around 40 ZB in 2020. The size is doubled every two years. That huge amount of data exceeds the capacity of current storage systems and processing systems, and brings technology challenges in developing big data computing models. Those challenges were defined by the U.S. Department of Energy [21] and summarized in paper "Exascale Computing and Big Data: The Next Frontier" in following fields [22]:

- Energy efficient circuit, power and cooling technologies
- High performance interconnect technologies
- Advanced memory technologies to improve capacity and bandwidth
- Scalable system software that is resilience ware
- Data management software that can handle the volume, velocity and diversity of data
- Programming environments to express massive parallelism, data locality and resilience
- Reformulation of science problems and refactoring solution algorithms
- Ensuring correctness in face of faults, reproducibility and algorithm verification.
- Mathematical optimization and uncertainty quantification for discover, design, and decision
- Software engineering and supporting structures to enable scientific productivity.

1.1 Background

The idea of solving big data problem firstly came from Google. Google is the most widely used search engine, that it collects huge amount of data everyday. Google figured out two key technologies to handle the amount data they want to store and analyze.

First, Google developed a distributed storage model called Google File System, which became the underlining storage architecture for the massive amount of data they need to store. GFS runs on a large array of cheap hardware. And it is designed to accepted and accommodate frequent hardware failure. Google published the paper in 2003, releasing the GFS concept to the public [15]. Then in 2004, they published another paper on their distributed computing system called MapReduce [14]. MapReduce vastly simplifies the distributed programming but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library [14].

Based on these two fundamental technologies from Google, Hadoop [13] was created as an open source version of the google technologies. It is a framework for running applications on large clusters of commodity hardware that can deal with complexities of high volume, velocity and variety of data. Hadoop provides reliable, scalable, fault-tolerance and efficient service for distributed computing. The cluster is designed to extended to thousands of nodes that each nodes can provides computation and storage. Hadoop consists of two components: the HDFS (Hadoop Distributed File System) which provides the distributed storage for the system and the MapReduce which is the processing model for Hadoop. The Hadoop project includes another module called YARN, which is a resource negotiator. It can also provide services for other big data runtimes such as Spark and HAMR. Hadoop is used by a wide variety of companies and organizations. Its users include Amazon, EBay, Facebook, Google, IBM, LinkedIn, Twitter and Yahoo! [7].

MapReduce has been successful in processing big data applications. While there are a class of applications that can not implemented efficiently by Hadoop [31]. Another framework called Spark [1] is designed to support applications that resue a working set of data across multiple parallel operations [31]. To achieve that goal, Spark introduces a new technology called resilient distributed datasets, or RDD, that enables data reuse by allowing persisting intermediate results in memory. In the event of failure, a lost RDD can be recovered by lineage information instead of checkpoints. It provides a friendly programming interface and a stack of libraries including SQL, MLib, GraphX and Spark Streaming. Spark SQL is Spark's module for working with structured data; MLib is Spark's scalable machine learning library; GraphX is an API for graphs and graph-parallel computation; and Spark Streaming allows Spark to build streaming applications. It was first developed in the AMPLab in 2009. Now it has been widely used by lots of companies including Amazon, Baidu, EBay, Tencent and Yahoo! [8].

Spark can provide a faster speed than Hadoop for many applications. However, it also requires more memory resource to support the in memory computing. HAMR [3] is a new cluster computing framework for processing and analyzing large scales of data. It is developed and first released by ET International Inc. HAMR is even faster than Hadoop and Spark with less memory consumptions.

1.2 Main Work

At the time I started this thesis, CAPSL didn't have a platform to provide students an environment to test big data applications. The purpose of the thesis is not to perform an extensive research but to construct a main eco-system that Hadoop and Spark can be in a same working condition. In additional, HAMR has also been installed as a test platform in the research eco-system. I also engaged the work of a selected of big data benchmarks, and took a preliminary test in all three eco-systems. Our experiment is based on these platforms: Hadoop 2.7.1, Spark 1.3.1 and HAMR 0.4.1. The detail can be found in the first section of chapter 5. The main work during this research includes:

- We set up Hadoop, Spark and HAMR clusters to run six benchmarks including PageRank, WordCount, Sort, TeraSort, Naive Bayes and K-means. These benchmarks are classified into three categories that are micro benchmark, web search and machine learning. And the three distributed systems are built on a four nodes cluster.

- We measured the running time, speedup, throughput, maximum and average memory and CPU usage for all the benchmarks on the three platforms of Hadoop, Spark and HAMR. And we compared the performance differences among these three platforms based on the characteristics of the benchmarks. Also, the experimental results are shown in following chapters and analyzed separately for different benchmarks.

- We compared the performances between Hadoop and Spark on PageRank, WordCount, Sort, TeraSort, Naive Bayes and K-means. The experimental results showed that Spark is faster than Hadoop. Specifically, Spark has a outstanding performance on machine learning applications including K-means and Naive

Bayes since these applications apply a function repeatedly to the same dataset. For PageRank, Spark runs faster with small input size. It begins to fail behind Hadoop with 16 million pages input since the memory resource is not enough. Spark is faster on WordCount. For Sort and TeraSort, Spark runs faster with large input. However, Spark consumes more memory capacity and the performance for Spark is restricted by the memory.

- We compared the performance between HAMR and Hadoop on PageRank and WordCount. The experimental results showed that HAMR runs faster than Hadoop for these two benchmarks. HAMR requires less memory usage compared with Spark. Compared with Spark, HAMR is faster when runs PageRank but is slower when runs WordCount.

This paper is organized as follows. Chapter 2 describes more details on Hadoop including HDFS, MapReduce and YARN. Chapter 3 describes more details on Spark including RDD. Chapter 4 discusses HARM and its Flowlet technology. Chapter 5 presents our experiments design and results. We give the conclusion in Chapter 6.

Chapter 2
HADOOP

Apache Hadoop is an open source software project. It is designed as a big data solution. In this chapter, we introduce HDFS, MapReduce and YARN which are three main technologies of Hadoop. In this chapter, we introduce Hadoop based on its official document [13].

2.1 HDFS

The Hadoop Distributed File System (HDFS) is a scalable file system for large distributed data-intensive applications. Unlike other distributed file system, HDFS is designed to be built from low-cost commodity component which requires it to be highly fault-tolerant.

A HDFS cluster consists a master server called NameNode and several slave servers called DataNodes. The architecture of HDFS is illustrated in Figure 2.1. Figure 2.1 was drawn based on the Hadoop website [13].The NameNode manages the meta-data including file names, locations, replications and the client's access to files. The DataNode manages the storage. Data in Hadoop cluster is split into smaller pieces called blocks and will be stored in DataNodes throughout the cluster. The block size can be set by users for optimization. These blocks are automatically replicated for fault tolerance. The replication factor is configurable and the default value is set to 3. The first two replications are put on the same rack but different DataNodes, and the third replication is put on different rack. The placements of replication is key to HDFS reliability and performance.

When a client wants to read from HDFS or write to HDFS, it first talks to the NameNode to get the locations of files and the access permit. Then it contacts

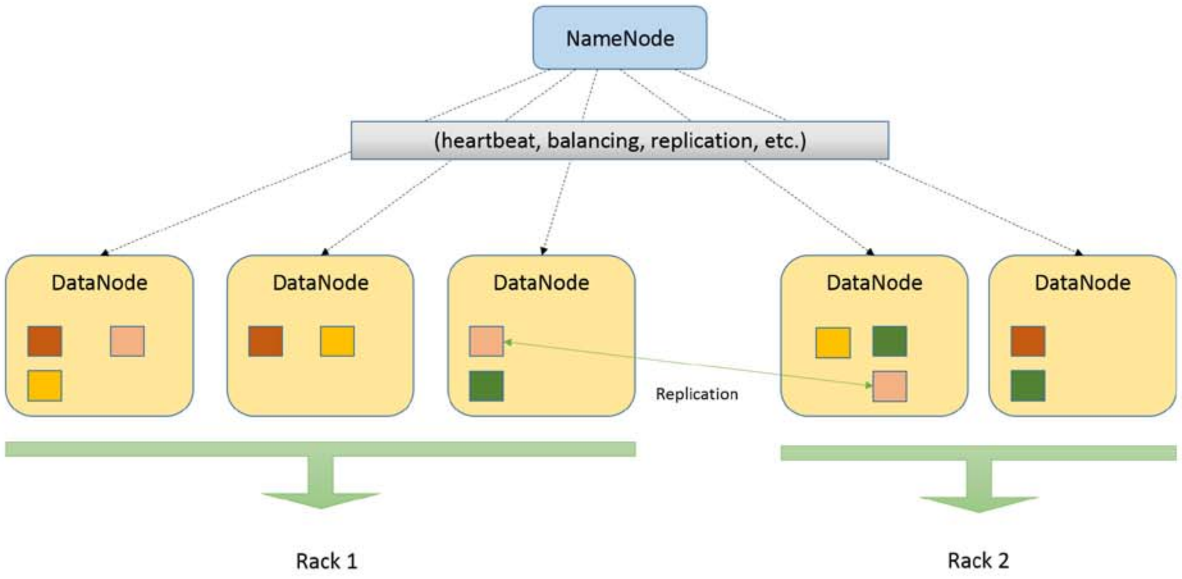


Figure 2.1: HDFS Architecture [13]

directly to these DataNodes. It is necessary to mention that data never flow directly through the NameNode, only the information stored there. The NameNode is also responsibly for detecting the condition of DataNodes. Each DataNode sends messages to the NameNode periodically, which are called heartbeats. The NameNode marks a DataNode as dead if it cannot detect heart-beat from that node and will stop sending I/O requests to that node. Data stored on dead nodes is not available to access anymore and will be re-replicated by the NameNode.

2.2 Classic MapReduce (MapReduce 1.0)

MapReduce was first developed by Google. It is a massively scalable, parallel processing programming model and software framework for generating large data sets. MapReduce is the processing part of Hadoop. Based on the technology of MapReduce and HDFS, processing can be executed at the location of data which reduces the cost of data transferring. The term MapReduce stands for two functions: Map and Reduce. Both of Map tasks and Reduce tasks work on key-value pairs. The main idea is to map the input data into key-value pairs and group together values with the same keys, then the reduce function merges together these values with the same keys.

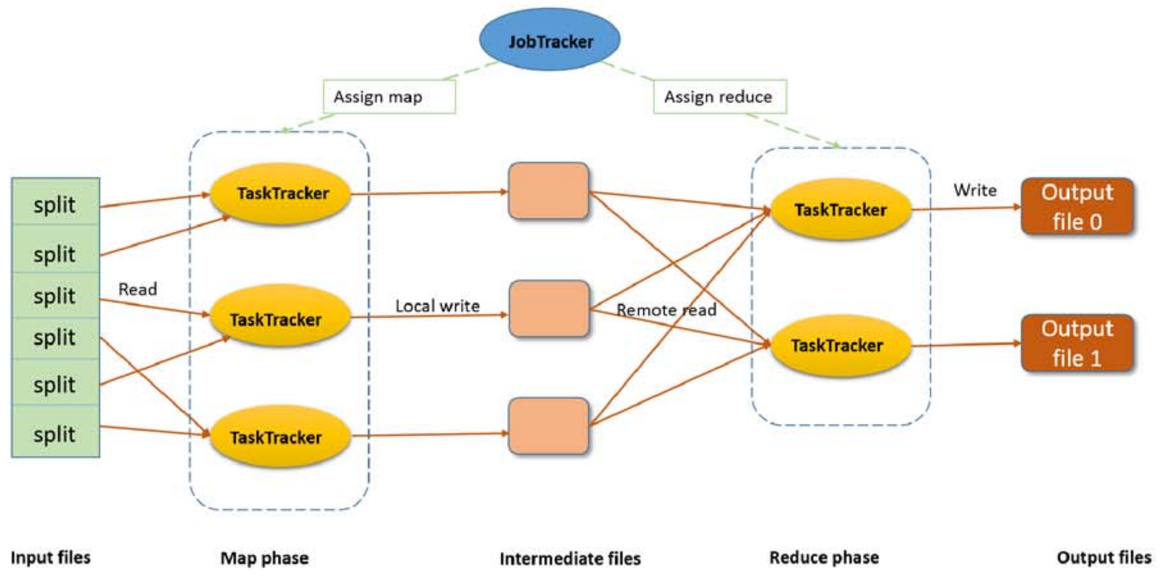


Figure 2.2: MapReduce 1.0 [14]

Figure 2.2 is the MapReduce architecture. It was drawn based on the paper: MapReduce: Simplified Data Processing on Large Clusters [14]. Just like the HDFS system, MapReduce cluster consists a master server called JobTracker and a number of slave servers called TaskTracker. The JobTracker is responsible for assigning task and the TaskTracker is responsible for computing. When a job comes into the MapReduce, it is split into small pieces of tasks. And JobTracker assigns these tasks to each TaskTracker. There are three steps: map, shuffle and reduce. The first step is the Map process. Data is computing in this step. The JobTracker will always try to pick nodes with local data for a Map task to reduce data transmission. If nodes with local data already have enough tasks running, the JobTracker will assign the task to a node in the same rack. The second step is shuffle. Results of Map process are sorted in this step and assigned to TaskTrackers which are running the reduce task. The intermediate data is combined here and distilled to have the final output.

TaskTrackers also send heartbeats to the JobTracker periodically. When a TaskTracker is marked as dead, the JobTracker will automatically assign tasks on that node to others, which makes MapReduce to be a highly fault-tolerant system.

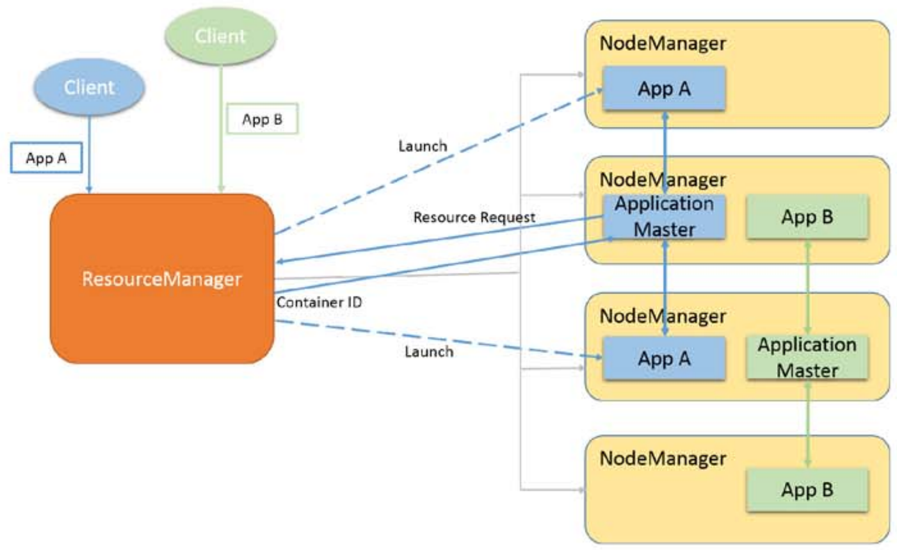


Figure 2.3: YARN (MapReduce 2.0). The cluster including one master and four slave nodes.[5] Application B submitted by the green client has ran on the cluster. The blue client just submitted Application A. The ResourceManager launched an ApplicationMaster for it on the second NodeManager. AM requested resources from RM and two containers were launched by RM. The container IDs would be sent back to AM. Application A began to execute.

2.3 YARN (MapRecuce 2.0)

YARN (Yet Another Resource Negotiator) or MapReduce 2.0 (MRv2) is the cluster resource manager. It is a key feature in Hadoop 2 version that significantly improves the scalability of the cluster and supports MapReduce and non-MapReduce applications, such as Spark, MPI, Giraph and Hamr, running on the same cluster. The core idea of YARN is to split the two functions of JobTracker into separate processing engine – the ResourceManager and the ApplicationMaster.

Figure 2.3 simply illustrates the architecture for YARN. It was drawn based on the slices published by Cloudera [5]. There is a global ResourceManager for entire cluster deployed on the master node which assigns CPU, memory and storage to applications running on the cluster. It also tracks heartbeats from NodeManagers and ApplicationMasters. The NodeManager is deployed on each slave node which monitors the resource usage including CPU, memory, disk and network, and communicates with

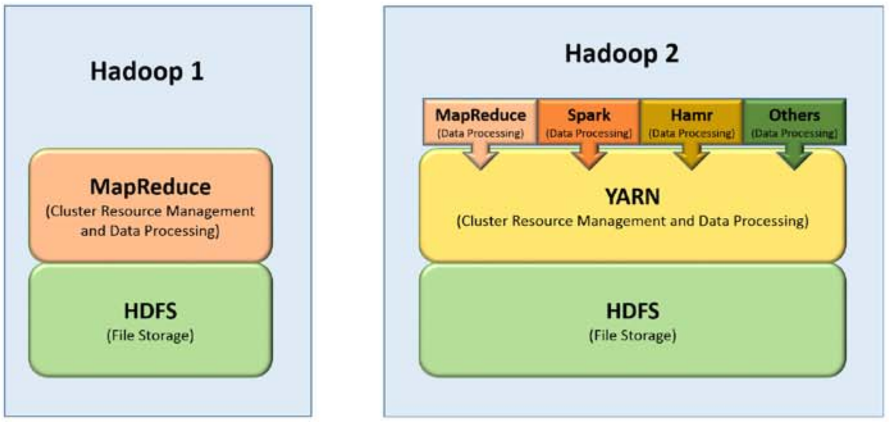


Figure 2.4: Differences between the Components of Hadoop1 and Hadoop 2

the ResourceManager. The ApplicationMaster is launched every time a job is submitted into the cluster. It negotiates resources from the ResourceManager and works with the NodeManager to execute the task. The container is created by ResourceManager upon request. A certain amount of resources are allocated on slave nodes for executing applications and will be De-allocated when the application completes.

The process of running jobs on MapReduce 2.0 is illustrated in Figure 2.3. Application B submitted by the green client has ran on the cluster. The blue client just submitted the Application A to the ResourceManager. The ResourceManager launched an ApplicationMaster for the job on the second NodeManager. Then the ApplicationMaster requested resources from the ResourceManager. Two containers were launched by the ResourceManager on two different slave nodes. The container IDs would be sent back to the ApplicationMaster. And the Application A began to execute. Map tasks and reduce tasks were run in containers. The progress was updated to the ApplicationMaster.

Figure 2.4 compares the architecture of Hadoop 1 and Hadoop 2. Since most functions of JobTracker are moved to ApplicationMasters running on the slave nodes, YARN significant reduces the responsibility of master node and improves the scalability of the cluster. According to the paper "Apache Hadoop YARN: Yet Another Resource Negotiator" [23], the limitation of Hadoop cluster is increased from 4000 nodes to 7000 nodes.

Chapter 3
SPARK

Spark is an open source cluster computing framework for solving big data problems. It was first developed in the AMPLab at UCBerkeley in 2009. Now it has become one of the most widely used programs to solve big data problems. Spark is designed to support applications which resue a working set of data across multiple parallel operations while also providing the same scalability and fault tolerance properties as MapReduce [31]. These applications are mainly classified in two types: iterative jobs and interactive analysis.

According to its website, Spark runs programs up to 100x faster than Hadoop MapReduce if it is used in-memory computing only. It can run 10x faster if it is used with combination of memory and disks. RDD, or resilient distributed dataset, is a distributed shared memory system and support reuse of data and intermediate results in memory. Spark is also designed to be used easily. It provides APIs in Java, Scala, Python and R shells. It can run on YARN and accessing data from HDFS. In this chapter we introduce Spark according its official document [1].

3.1 Spark Components

According to Spark document, Figure 3.1 drawn based on Spark website [1], gives a short overview about how Spark runs on clusters. The driver program is the main program that contains the SparkContext and coordinates the applications. The SparkContext needs to connect to a cluster manager to get memory and processing resource to execute the programs. Spark can run on three cluster managers:

- Standalone mode. It is provided by Spark distribution. The cluster can be launched manually or by launch scripts. It can be simply used on a single node for testing.

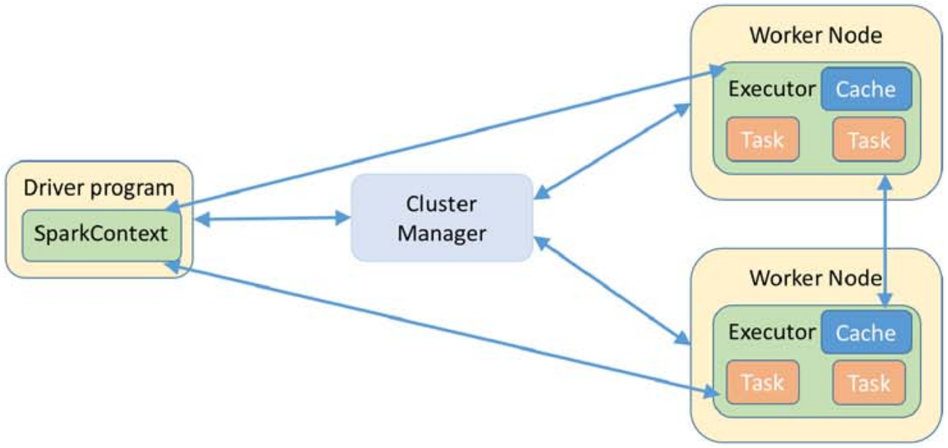


Figure 3.1: Spark Components [1]

- Mesos mode. Mesos is a cluster manager provided by Apache.
- YARN mode. That requires the Hadoop 2.0 intalled on the cluster.

The cluster manager allocates resource on executors. Different applications run on different executor processes which means they can not share data in memory. The cluster manager also sends the executor information back to driver program that connection is set up between driver and workers. Tasks are sent to executors by SparkContext.

3.2 RDD

A resilient distributed dataset is a read-only and fault-tolerant collection of objects that support parallel processing [1]. To achieve the fault-tolerant goal, Spark use the lineage. That means if a partition is lost, Spark can rebuild that partition by the lineage information instead of going back to the checkpoint. That also helps the system to save the cluster network resource by avoiding replicating data for checkpoint [30]. As discussed in paper "Spark: Cluster Computing with Working Sets" [31], RDDs can only be created in four ways:

- From a file in HDFS or other shared file systems.
- By parallelizing a Scala collection in the driver program.

- By transforming an existing RDD from one type to another type using the flatMap operation.
- By chaning the persistence of an exiting RDD.

When a job is submitted to Spark, workers read data from HDFS or other distributed file systems and cache it in memory. Data can be reused for each iteration. RDD helps to reduce reading from and writing to disk which contributes to speed up the processing. RDD is read-only which means any changes to a RDD partition will cause a creation of new partition [17].

3.3 Spark Streaming

Stream programming has been widely used in High Performance Computing community [28, 29, 27, 25, 26, 24, 29].Spark supports both batch and streaming data processing. Spark Streaming is an extension of Spark engine. As described by its document, it provides the scalable, high-throughput and fault-tolerant stream processing of live data streams. The main abstraction of Spark streaming is called discretized stream or DStream. It is a continuous stream of data represented by RDDs. The working process of Spark Streaming is shown in Figure 3.2. When a live input data stream comes into the system, Spark Streaming divides it into DStreams. Then the streaming computations can be changed to Spark batch jobs that execute RDD transforms. The result are batches of processed data.

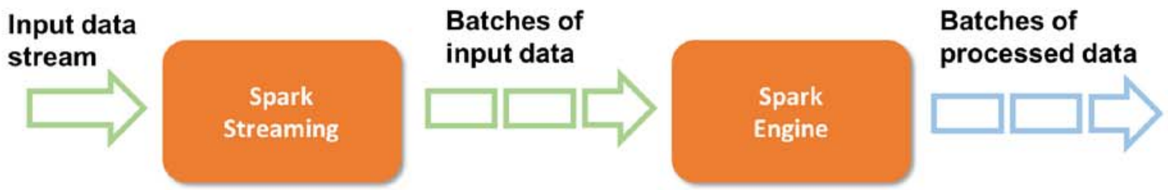


Figure 3.2: Spark Streaming [10]

Spark Streaming can receive data from Kafka, Flume, HDFS/S3, Kinesis or Twitter. The processed data can be written to both HDFS and Databases.

Chapter 4
HAMR

HAMR is a new cluster computing framework for processing and analyzing large scales of data. It is developed and first released by ET International Inc. [12]. HAMR is a new solution to BigData that can provide up to 30 times faster processing speed than Hadoop and 10 times faster speed than Spark. Unlike Hadoop and Spark, HAMR is a streaming engine that drive streaming and real-time analytics by the its Flowlet technology. To achieve the high performance goal, HAMR aims to reduce and push data out of system early. That is not only reducing the memory usage of the program but also decreasing the CPU utilization. In this chapter, we introduce HAMR according to its official document [3].

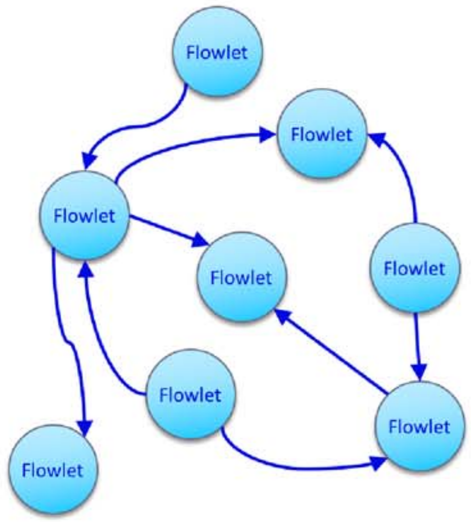


Figure 4.1: Directed Acyclic Graph

A HAMR cluster also contains two types of nodes: a single master nodes and several slave nodes. The main routine is executed on the master node and then the job will be assign to slave nodes by the master. Nodes included in one cluster are interconnected with each other. A HAMR workload can be illustrated as a structure called directed acyclic graph or DAG as shown in Figure 4.1. A DAG is a directed graph consisting of nodes and edges with no cycles. In HAMR workload, the node is called Flowlet and the directed edge connecting nodes represents the transformation of data.

HAMR can read data from a HDFS and a network file system. Data will be converted to key-value pairs which is the required data format by HAMR and transferred between Flowlets. HAMR supports both batch and stream models.

To manage the allocation of memory and processor resources, HAMR supports two resource negotiators:

- HORN: HAMR's Other Resource Negotiator
- YARN: Yet Another Resource Negotiator

HORN is provided by HAMR for convenient testing. It is easy to use and works with small clusters. However, HORN only supports a single user. In our experiment, we ran HAMR with HORN.

4.1 Flowlet

Each Flowlet stands for a stage of the processing in a HAMR workload. The directed edges between them are provided by binding ports and slots together. There are two models to transmit key-value pairs between flowlets: push mode which is normally used and pull mode which is used to retrieve data and provide random access to storage sink. A Flowlet consists of one or more partitions. And the number of partitions is a parameter that determines the level of concurrency in each Flowlet. These partitions are distributed on slave nodes as shown in figure 4.2. Each Flowlet has a partitioner for determining the mapping of income key-value pairs to partitions. In figure, Flowlet A and Flowlet B both contain four partitions and distributed on two slave nodes. When a key-value pair is pushed into Flowlet B from A, the network layer

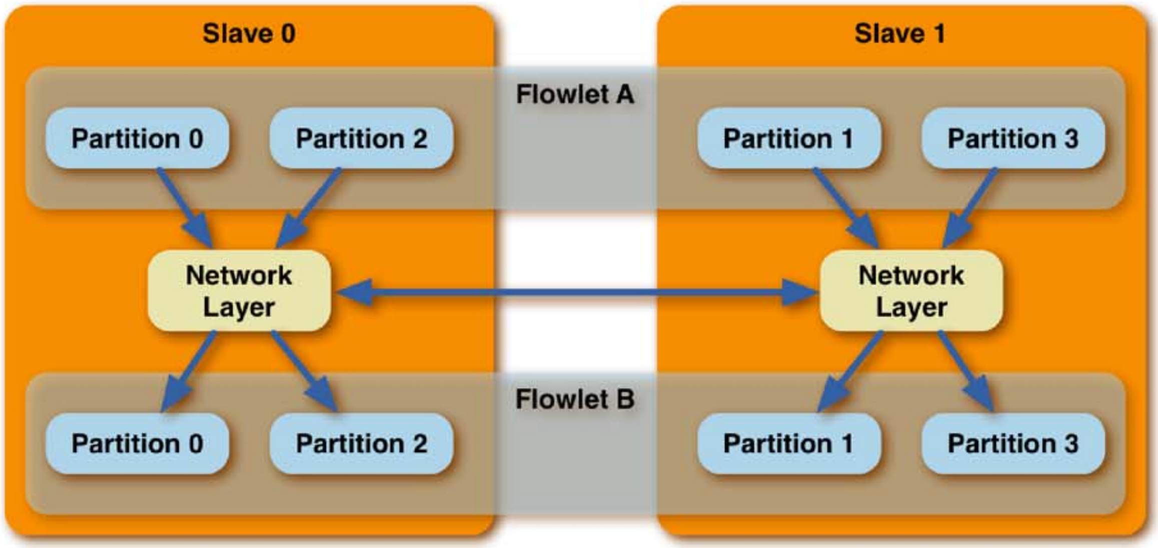


Figure 4.2: Flowlets and Partitions

decides which partition the data will be mapped to based on the partitioner of Flowlet B.

HAMR uses the aggregator to store pushed key-value pairs. An aggregator is a buffer locating on the same thread with the partition. A pushed data will first be serialized and then stored in the buffer. Key-value pairs are sorted and flushed to network in two situations: the buffer is full or a default time has passed. Based on this way of transferring data, HAMR support batch and stream models.

4.2 Runtime

Hamr runtime includes two types of thread that are worker threads and network threads that are represented by yellow blocks as shown in Figure 4.3. The worker thread is responsible for retrieving tasks from a work queue and processing them. It is a Java thread and non-interruptable. The network thread receives bins and schedules them to destination partitions. The small blocks with number represent tasks. Each task is a unit of non-interruptable serial work. When a task is sent to a worker thread for processing, it can not be blocked. Only one task of contained in a same partition can be executed at most one thread at any given time. That makes partition tasks thread-safe.

Tasks in different partitions are independently and can be executed asynchronously.

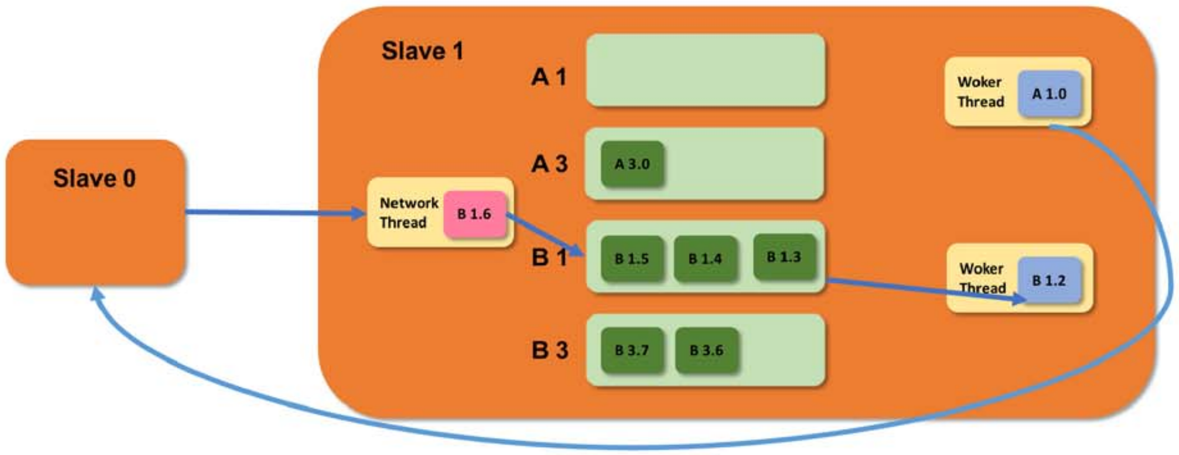


Figure 4.3: Hamr Runtime

In Figure 4.3, there are two Flowlets, A and B, on Slave 1 and each of them contains two partitions. Therefore Slave 1 has four work queues in total. The network thread receives a bin from slave 0. A new task B1.6 is created and scheduled to partition 1 of Flowlet B by the network thread. Partition B1 has three tasks representing by the green blocks and them are processed from B1.3 to B1.5. Task A1.0 and Task B1.2 are being executed on two worker threads. Task A1.0 is sending bins to Flowlet B on slave 0.

4.3 Software dependency

Unlike Hadoop and Spark, HAMR only runs on Linux and requires software dependencies before deploying the cluster. Before installing HAMR, users should have ZooKeeper [2] and RabbitMQ [9] on the cluster. ZooKeeper is an open source framework released by Apache that provides centralized service for holding and communicating configuration, control and job state information between all nodes. RabbitMQ is an open source queuing system that serves as an intermediary for messaging. In HAMR cluster, it provides the network layer between nodes. Nodes having request to transfer data need to make a connection to RabbitMQ and then RabbitMQ delivers data to destination nodes [6]. At least one installation of ZooKeeper is requested for one cluster. However, RabbitMQ needs to be placed on every node.

Chapter 5
EXPERIMENT RESULTS AND ANALYSIS

5.1 Cluster Setup

The experimental cluster we used consists of four computers. One of them is designed to serve as both a master and a slave node. The other three are designed to be slave nodes. The hardware information for the cluster is shown as following:

- 4 nodes interconnected by Each node 10G-Ethernet.
- Each node has 2 Intel Xeon CPU E5-2670 running at 2.60GHz
- Each CPU has 8 cores, each core has 2 threads (hyper-threading).
- Each node has 64GB memory.
- The configured capacity for HDFS is 6.49 TB with 1.62 TB per node.

We use the CentOS 6.7 operating system and JAVA 1.7.0 version for all the nodes. We installed Hadoop 2.7.1, Spark 1.3.1 and HAMR 0.4.1 for running all the benchmarks described below. The version for Hadoop and Spark are stable released and the version for HAMR is the latest release. We use the Hadoop and Spark benchmarks and data generators provided in HiBench Benchmark Suite 4.0 version [4]. For Hadoop Naive Bayes and K-means benchmarks, we use Mahout 0.10.1 distribution instead of Mahout 0.9 which is included in HiBench. The HiBench Benchmark Suite can be found and downloaded from the following web page:`https://github.com/intel-hadoop/HiBench`. The benchmarks for HAMR are included in HAMR 0.4.1 distributed. Table 5.1 shows the paths for all the benchmarks.

We configure the YARN and MapReduce memory allocation according to the convention set by Hortonworks (`http://docs.hortonworks.com/HDPDocuments/HDP2/`

Benchmark	**Platform**	**Source**
PageRank	Hadoop	HIBENCH_HOME/src/pegasus/target/ pegasus-2.0-SNAPSHOT.jar
PageRank	Spark	org.apache.spark.examples.SparkPageRank
PageRank	HAMR	com.etinternational.hamr.benchmark.pagerank
WordCount	Hadoop	HADOOP_HOME/share/hadoop/mapreduce/ hadoop-mapreduce-examples-2.7.1.jar
WordCount	Spark	com.intel.sparkbench.wordcount.ScalaWordCount
WordCount	HAMR	com.etinternational.hamr.examples.wordcount.WordCount
Sort	Hadoop	HADOOP_HOME/share/hadoop/mapreduce/ hadoop-mapreduce-examples-2.7.1.jar
Sort	Spark	com.intel.sparkbench.sort.ScalaSort
TeraSort	Hadoop	HADOOP_HOME/share/hadoop/mapreduce/ hadoop-mapreduce-examples-2.7.1.jar
TeraSort	Spark	com.intel.sparkbench.terasort.ScalaTeraSort
Naive Bayes	Hadoop	MAHOUT_HOME/bin/mahout seq2sparse MAHOUT_HOME/bin/mahout trainnb
Naive Bayes	Spark	org.apache.spark.examples.mllib.SparseNaiveBayes
K-means	Hadoop	MAHOUT_HOME/bin/mahout kmeans
K-means	Sparks	org.apache.spark.examples.mllib.DenseKMeans

Table 5.1: Benchmarks

HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html). We set the HDFS block replication to be 3 and the block size to be 128MB.

We run Spark on YARN with yarn-client mode. We changed each parameter to tune the Spark. As a result, we configured the Spark with the following parameters to get the best performance:

- 20G executor memory
- 10G driver memory
- 8 executors
- 8 executor cores

We run HAMR on HORN. For the software dependencies requirement, we use ZooKeeper 3.4.6 and RabbitMQ 3.5.4. Both of them are the least release version.

5.2 Benchmarks

In this section we describe the six benchmarks we used to compare the Hadoop, Spark and Hamr performances. These benchmarks including Sort, WordCount, TeraSort, PageRank, Naive Bayes, K-means are classified into three categories as shown in table 5.2. One thing we need to mention is that only two workloads were ran on Hamr. These Hadoop and Spark workloads are contained in HiBench suite which is provided by Intel. The Hamr programs are included in Hamr distributed system. All the input files are created by the data generator provided in HiBench suite.

Category	Benchmark	Hadoop	Spark	Hamr
Micro Benchmark	Sort	yes	yes	
	WordCount	yes	yes	yes
	TeraSort	yes	yes	
Web Search	PageRank	yes	yes	yes
Machine Learning	Naive Bayes	yes	yes	
	K-means	yes	yes	

Table 5.2: Benchmarks Used in the experiment and their categories

5.2.1 Micro Benchmark

The Sort, WordCount and TeraSort are provided as examples in Hadoop library files and the Spark versions are provided by HiBench. The function of Sort is to sort the input text file by key. The WordCount program reads a text input file and count how many times each word occurs. TeraSort is a well-know benchmark on Hadoop. It is writen by Owen O'Malley at Yahoo Inc. and won the annual general purpose terabyte sort benchmark in 2008 and 2009. The Terasort package includes three applications: Teragen which is a MapReduce program and can be used to generate input data, TeraSort which can be used to sorts the input data, and TeraValidata which can be used to check the output.

5.2.2 Web Search

PageRank was named by Larry Page at Google. It wass first used by Google to rank the website pages by their importance. Each page is assigned a numerical value called PageRank which can represent the importance. When a page links to another, it will cast votes to another one. Also, a page with higher PageRank has more weight to vote. That means a page gets more important if it is linked by more pages with higher PageRank values. The Hadoop version benchmark is provided by Intel and the Spark version is contained in Spark distribution.

5.2.3 Machine Learning

Machine learning is an branch of artificial intelligence that allows the computer builds models based on data rather than being decided by the developer. The Naive Bayes Classifier algorithm is based on Bayes' theorem with independence assumptions between the features to categorize text. There are two steps in Naive Bayes: training and testing. In the training step, the classifier is trained by the sample text file and get a model. In the testing step, the classifier processes the input data based on the model [16].

K-Means algorithm is used to group items into k clusters. And the value of k can be decided by users. The algorithm first randomly select k data points as centroids for each cluster. Then the input data points are assigned to clusters depending on which centroid is closest to them. After that the algorithm computes new centroids for all clusters by the average of all the points and assigns all points to clusters with new centroids. This processing will be repeated until it converges [16].

5.3 Results and Analysis

In this section, we present the experiment results and analysis for benchmarks described above. We ran all benchmarks on Hadoop and Spark distributed systems and ran PageRank and WordCount on HAMR cluster. To compare their performance, we record the running time, the maximum and average CPU usage, the maximum and average memory usage. The CPU and memory usage are measured for 32 threads on one node. Also we calculate the speedup, the throughput and the throughput per node. For each benchmark, we choose five data sets and run it three times with each data sets to record the average running time. And the running time is record in seconds.

5.3.1 Comparison between Hadoop and Spark

5.3.1.1 PageRank

The following graph 5.1 represents a running time performance comparison for PageRank running on Hadoop and Spark. And Table 5.3 summarizes Spark's speedup over Hadoop on the same data sets.

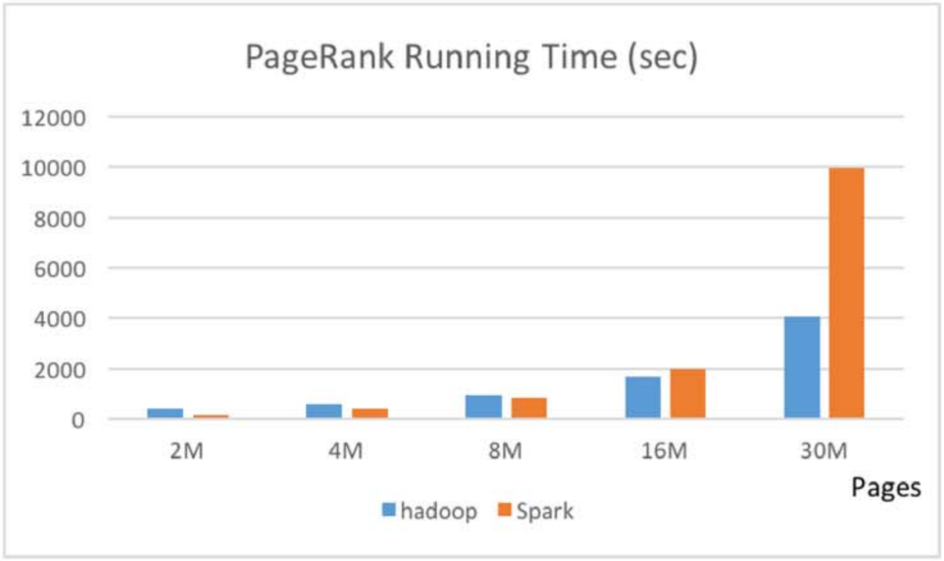


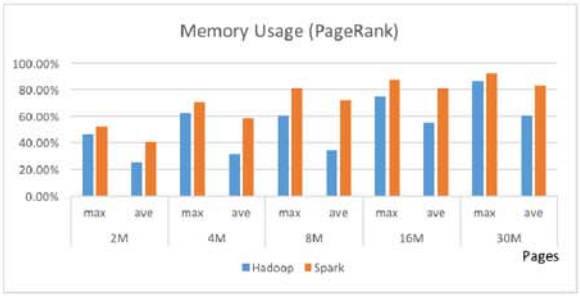
Figure 5.1: Running Time Comparison for PageRank running on Hadoop and Spark

Input	**2M**	**4M**	**8M**	**16M**	**30M**
Hadoop	405	575	926	1691	4078
Spark	154	404	834	1965	9953
Speedup	2.63	1.42	1.11	0.86	0.41

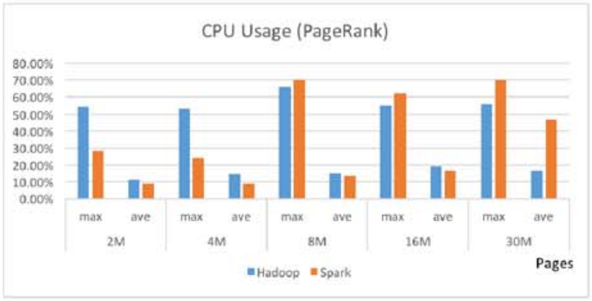
Table 5.3: Spark's Speedup over Hadoop on Running PageRank

The input data sets are range from 2 million pages to 30 million pages and the input sizes are range from 1 GB to 19.9 GB. The workload is ran with 3 iterations.

When the input is from 2 million to 8 million pages, Spark has better performance than Hadoop with maximum speedup up to 2.63 times. However, the advantage decreases as the input size increases. Spark begins to fail behind Hadoop with 16 million pages input and takes more than 2 times running time with 30 million pages input. PageRank runs with iterations which requires more memory resource. As shown in Figure 5.2a, Spark has a higher memory resource consumption up to 2x of Hadoop. When the input data size is too large, Spark does not have enough memory for new created RDDs and has to start its replacement police which effects Spark's performance and also increases the CPU consumption as shown in Figure 5.2b. The CPU usage for Hadoop is roughly same with different input size. Compared with Hadoop, Spark saves CPU resources with small inputs. Figure 5.3 compares the throughput between Hadoop and Spark. The throughput for Hadoop increases as the input size increases from 2 MB to 10 MB, and decreases with 30 MB. While the throughput for Spark decreases as the input size increases.



(a) Memory Usage Percentage

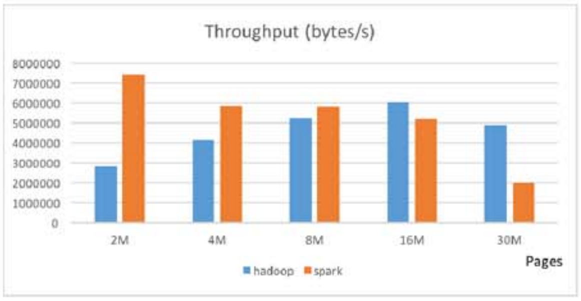


(b) CPU Usage Percentage

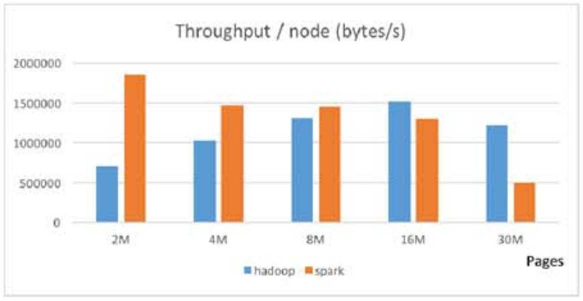
Figure 5.2: Memory and CPU Comparison for PageRank Running on Hadoop and Spark

5.3.1.2 WordCount

The following graph 5.4 represents a running time performance comparison for Running WordCount on Hadoop and Spark. And Table 5.4 summarizes Spark's speedup over Hadoop on the same data sets.



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.3: Throughput Comparison for PageRank Running on Hadoop and Spark

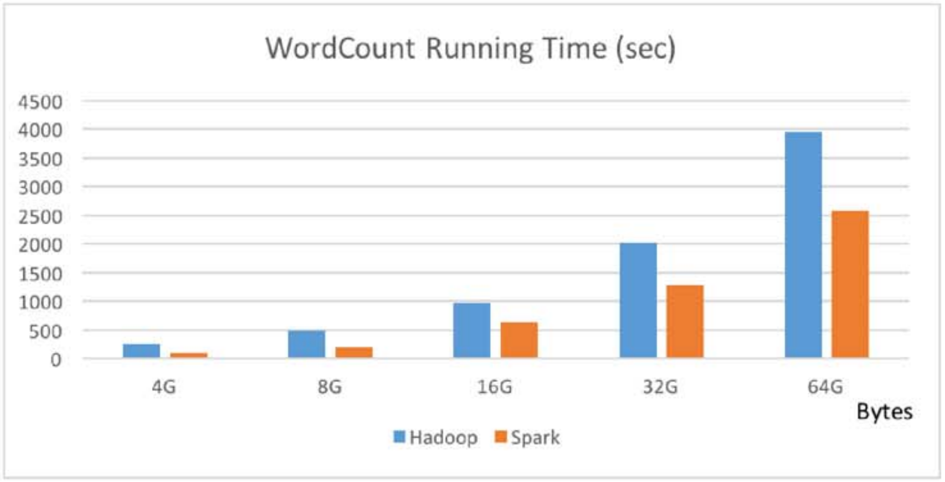


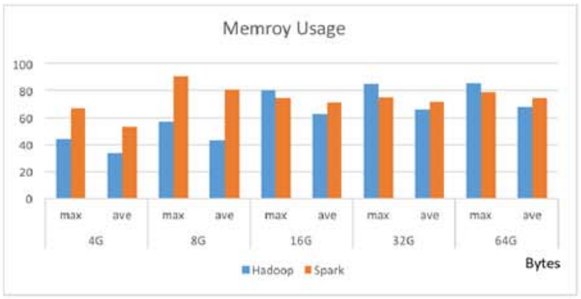
Figure 5.4: Running Time Comparison for WordCount Running on Hadoop and Spark

The input data sizes are range from 2 GB to 64 GB. As the input size increases, the running time of both Hadoop and Spark increases. The WordCount benchmark runs without iterations. Each word in the input file will be assigned a key. And all the words with the same keys will be counted together. Spark gives a better performance on running WordCount with the speedup up to 2.76 times. The speedup is influenced by the input data size. Tt decreases when the input size becomes larger. For WordCount benchmark, both Hadoop and Spark require high memory consumptions as shown in Figure 5.5a. However, Spark has outstanding performance on saving CPU resource. Hadoop has up to 7x more average CPU consumptions according to Figure 5.5b. For Hadoop, these key-vale pairs are written back to the disk. While for Spark, they are stored in memory. The throughput for Hadoop does not change with input. Even the

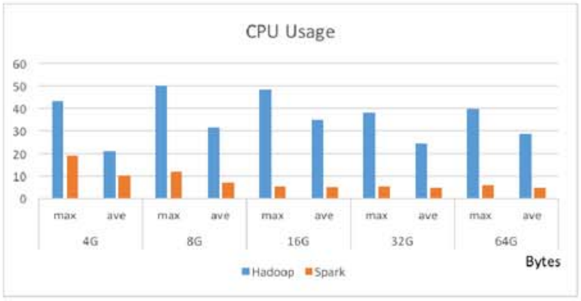
Input	4G	8G	16G	32G	64G
Hadoop	265	506	971	2023	3935
Spark	96	207	631	1273	2565
Speedup	2.76	2.44	1.54	1.59	1.53

Table 5.4: Spark's Speedup over Hadoop on Running WordCount

throughput for Spark get lower as input size goes larger, it still have up to 63.8 % improvement than Hadoop as shown in Figure 5.6.

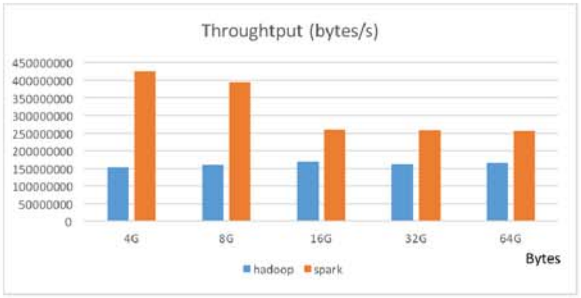


(a) Memory Usage Percentage

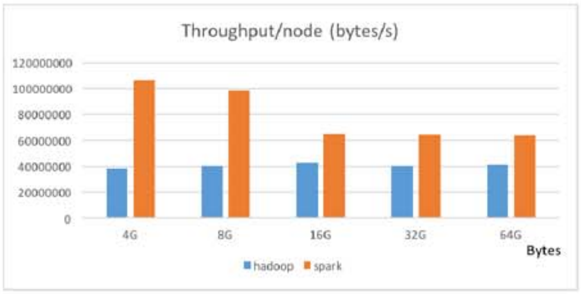


(b) CPU Usage Percentage

Figure 5.5: Memory and CPU Comparison for Running WordCount on Hadoop and Spark



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.6: Throughput Comparison for Running WordCount on Hadoop and Spark

5.3.1.3 Sort

The following graph 5.7 represents a running time performance comparison for running Sort between Hadoop and Spark. And Table 5.5 summarizes Spark's speedup over Hadoop on the same data sets.

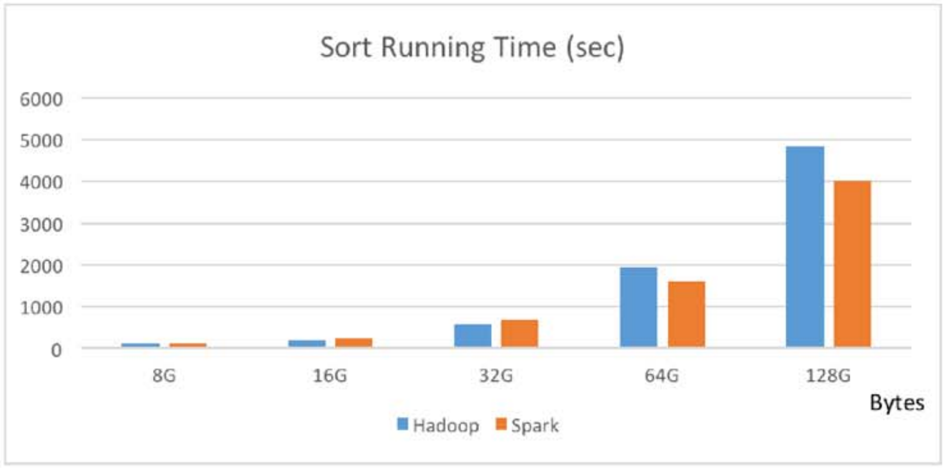


Figure 5.7: Running Time Comparison for Sort Running on Hadoop and Spark

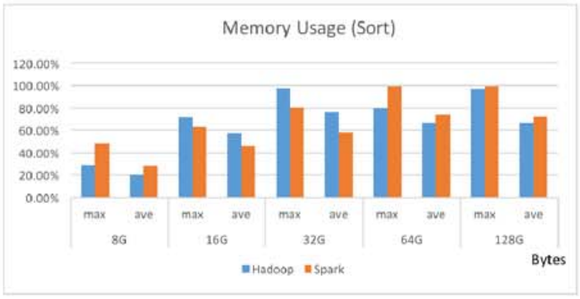
Input	**8G**	**16G**	**32G**	**64G**	**128G**
Hadoop	109	194	599	1947	4834
Spark	128	253	680	1603	4010
Speedup	0.85	0.77	0.88	1.21	1.21

Table 5.5: Spark's Speedup over Hadoop on Running Sort

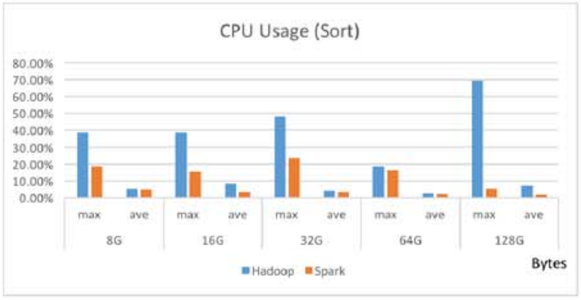
The input sizes are range from 8 GB to 128 GB. When the input size is small, Hadoop has better performance than Spark. Hadoop has the best performance when the input is 16 GB with 1.30x speedup over Spark. However, Spark beats Hadoop when the input is greater than 32 GB. The maximum speedup is 1.21 times. Figure 5.8 shows the memory and CPU usage for Hadoop and Spark. To implement Sort workload, Spark requires more memory resources but cost less CPU resources than Hadoop. Hadoop has extremely high maximum CPU usage which is up to 13x than Spark. The throughput decreases as the input size increases for both Hadoop and Spark as shown in Figure 5.9. Hadoop gives the maximum throughput which is around 80M per second when the input size is 16 GB.

5.3.1.4 TeraSort

The following graph 5.10 represents a running time performance comparison for TeraSort benchmark between Hadoop and Spark. And Table 5.6 summarizes Spark's

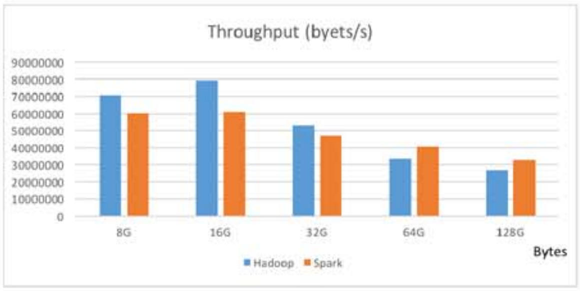


(a) Memory Usage Percentage

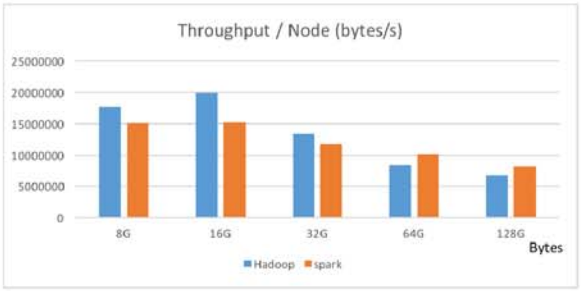


(b) CPU Usage Percentage

Figure 5.8: Memory and CPU Comparison for Sort Running on Hadoop and Spark



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.9: Throughput Comparison for Sort Running on Hadoop and Spark

speedup over Hadoop on the same data sets.

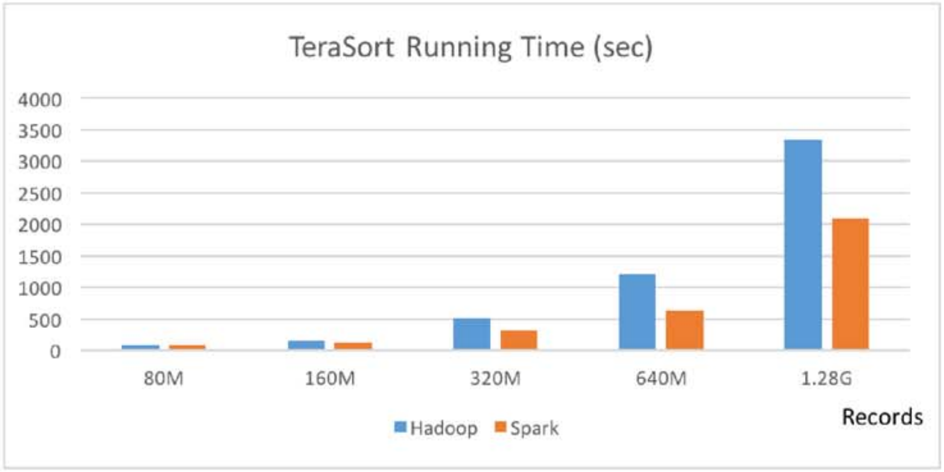


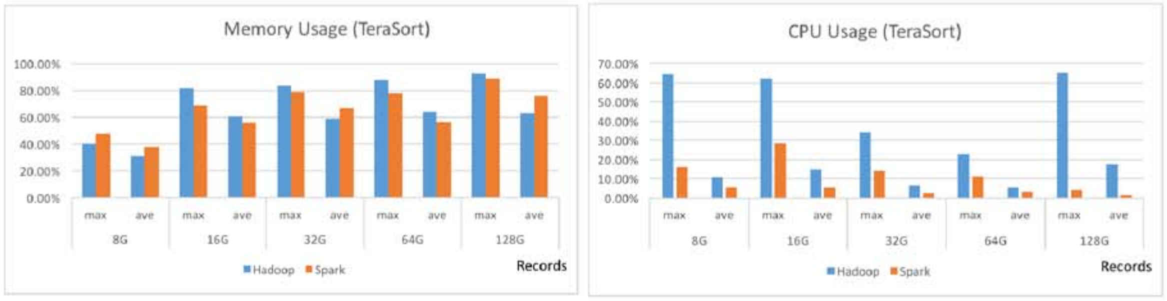
Figure 5.10: Running Time Comparison for TeraSort Running on Hadoop and Spark

The input sets are range from 80 million to 1280 million records and the sizes are range from 8 GB to 128 GB with 100 bytes for each record. Spark and Hadoop

Input	80M	160M	320M	640M	1.28G
Hadoop	85	157	507	1215	3334
Spark	88	129	323	641	2090
Speedup	0.97	1.22	1.57	1.90	1.60

Table 5.6: Spark's Speedup over Hadoop on Running TeraSort

have same performance when the input is small. However Spark executes faster than Hadoop when the input is larger than 160 million records. The advantage is more obvious with large input size. The maximum speedup occurs when the input is 320 million records which is 1.90x faster than Hadoop. As shown in Figure 5.11, both two system have roughly same memory utilization. Spark has improvement on saving CPU resource especially the maximum CPU usage. Hadoop has 13x more maximum CPU consumption than Spark when the input size is 128 GB. Spark also has high throughput when the input is larger than 16 GB as shown in Figure 5.12. Spark gives the maximum throughput with 120 MB per second when the input equals 160 MB.



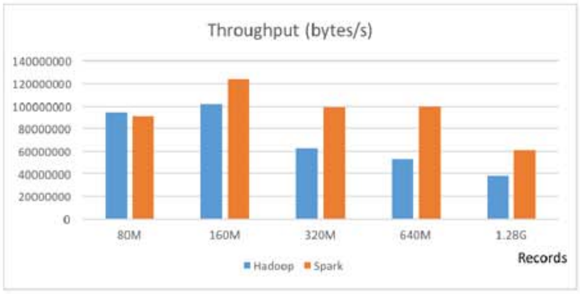
(a) Memory Usage Percentage

(b) CPU Usage Percentage

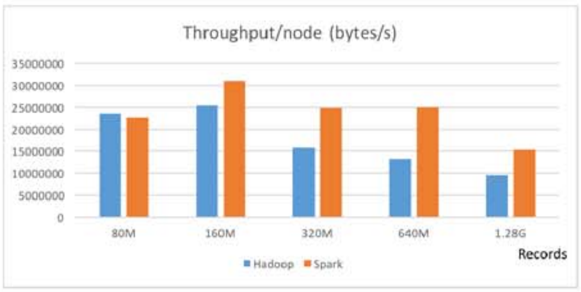
Figure 5.11: Memory and CPU Comparison for Running TeraSort on Hadoop and Spark

5.3.1.5 Naive Bayes

The following graph 5.13 represents a running time performance comparison for Naive Bayes between Hadoop and Spark. And Table 5.7 summarizes Spark's speedup over Hadoop on the same data sets.



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.12: Throughput Comparison for Running TeraSort on Hadoop and Spark

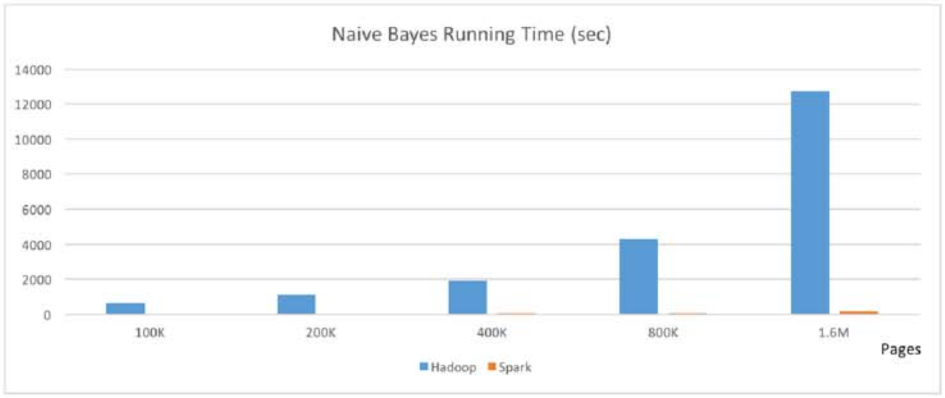
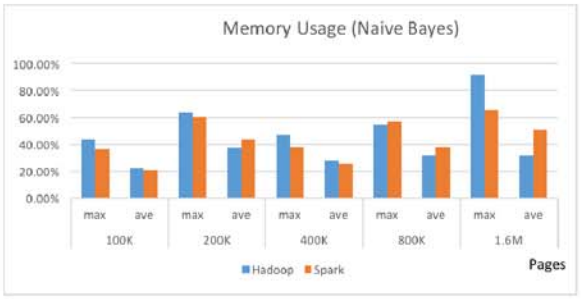


Figure 5.13: Running Time Comparison for Naive Bayes Running on Hadoop and Spark

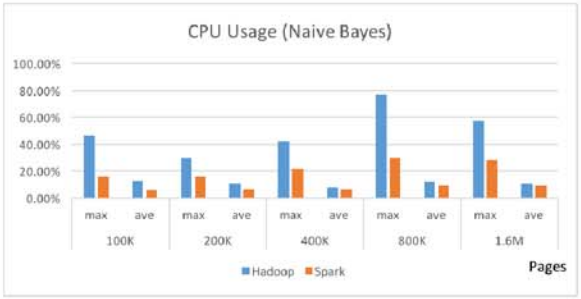
The input sets are range from 100K to 1.6M and the input sizes are range from 0.45 GB to 7.2 GB. Naive Bayes is a machine learning benchmark. Spark is designed for iterative jobs which reuse the same data set to optimize a parameter that it supposes to have a better performance than Hadoop on machine learning algorithms. As shown in Figure 5.13, Spark has an outstanding advantage on running Naive Bayes workload especially with large input data size. According to Figure 5.7, the speedup goes from 13.83x to 89.01x as the input changes. Also, Spark has little improvements on average CPU utilization and large improvements on maximum CPU utilization. The maximum CPU usage for Spark is only half of Hadoop. They have roughly same memory utilization. Spark has higher throughput than Hadoop, and the throughput increases as the input size goes larger according to Figure 5.15. The maximum throughput for Spark is 50 MB per second when the input equals 1.6 million.

Input	**100K**	**200K**	**400K**	**800K**	**1.6M**
Hadoop	636	1092	1920	4262	12728
Spark	46	56	70	84	143
Speedup	13.83	19.50	27.43	50.74	89.01

Table 5.7: Spark's Speedup over Hadoop on Running Naive Bayes

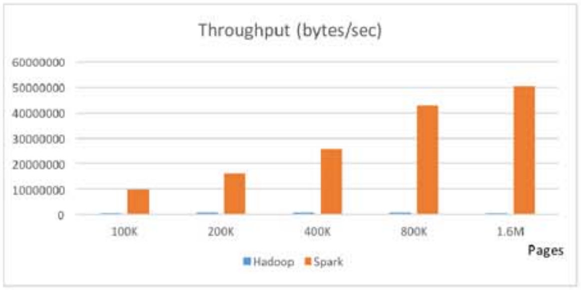


(a) Memory Usage Percentage

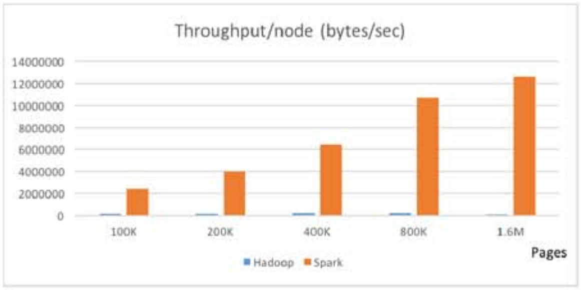


(b) CPU Usage Percentage

Figure 5.14: Memory and CPU Comparison for Naive Bayes Running on Hadoop and Spark



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.15: Throughput Comparison for Naive Bayes Running on Hadoop and Spark

5.3.1.6 K-means

The following graph 5.16 represents a running time performance comparison for K-means between Hadoop and Spark. And Table 5.8 summarizes Spark's speedup over Hadoop on the same data sets.

K-means is a machine learning algorithm which should be suitable for Spark. The input data points are assigned to clusters with a closest centroid and new centroids are created by these points assigned in the clusters. These steps are repeated until it converges. And for each time, Hadoop needs to store the intermediate results back to

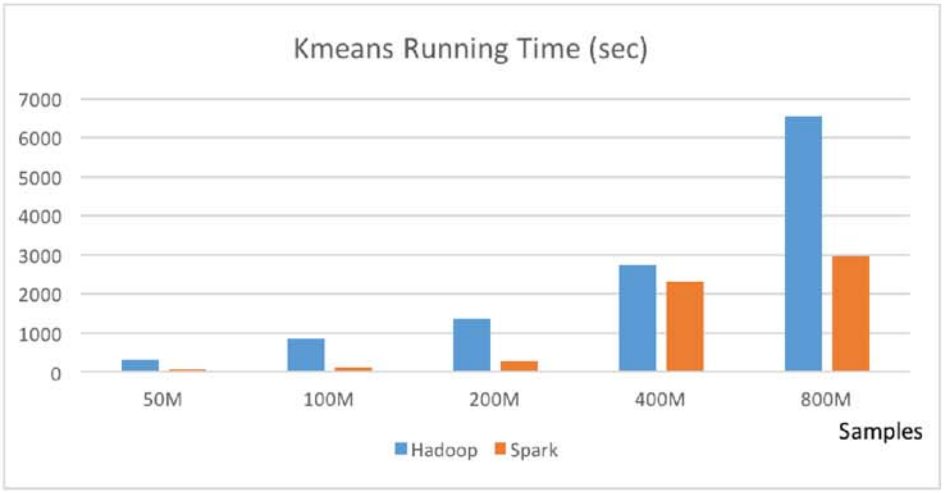
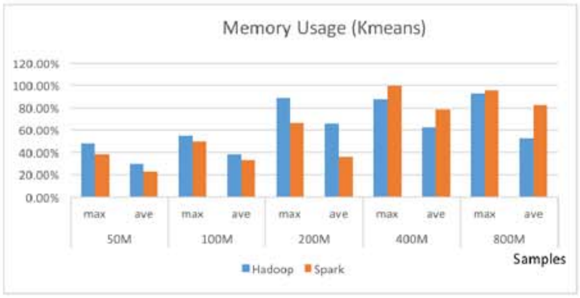


Figure 5.16: Running Time Comparison for K-means Running on Hadoop and Spark

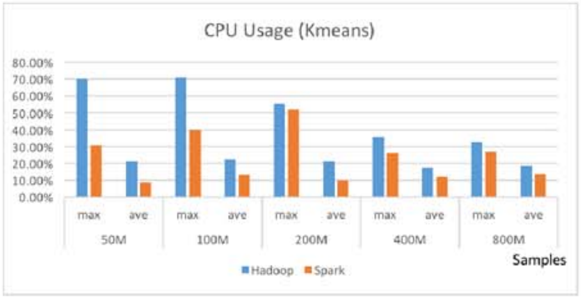
Input	50M	100M	200M	400M	800M
Hadoop	333	858	1357	2755	6558
Spark	82	137	297	2322	2978
Speedup	4.06	6.26	4.57	1.19	2.20

Table 5.8: Spark's Speedup over Hadoop on Running K-means

the disk. However, Spark just keeps them in memory. The input sets are range from 100K samples to 1.6M samples and the input sizes are range from 10 GB to 160 GB. We ran the K-means algorithm with 5 clusters. The Figure 5.8 clearly shows that Spark has better performance than Hadoop that the speedup is up to 6.26 times. However, the advantage is bounded by the memory. The speed up goes down when the input is more than 100 million samples and has the minimum value with 1.19x when the input is 400M. As shown in Figure 5.17a, the maximum memory usage for Spark is almost 100 percent with 400M and 800M input that Spark can not create more RDDs at that point. Spark saves more CPU resource than Hadoop especially with small input. The maximum CPU consumption for it is only half of Hadoop with the input equals to 50M and 100M. When the input is smaller than 200 million samples, Spark has improvements on throughput than Hadoop which is up to 6x. When the input is greater than 200 million samples, the throughput for Spark has an obvious decrement as represented by Figure 5.18.

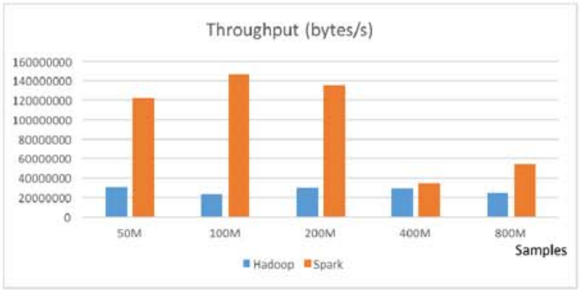


(a) Memory Usage Percentage

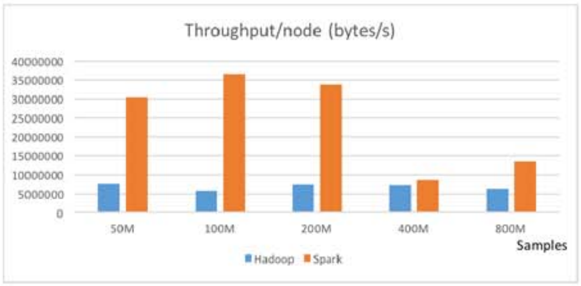


(b) CPU Usage Percentage

Figure 5.17: Memory and CPU Comparison for K-means Running on Hadoop and Spark



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.18: Throughput Comparison for K-means Running on Hadoop and Spark

5.3.2 Comparison between Hadoop, Spark and HAMR

In this part, we run PageRank and WordCount workloads on HAMR and represent the experiment result compared with Hadoop and Spark. To evaluate the performance, we record the running time, maximum and average memory usage, maximum and average CPU usage. We also calculate the speedup, throughput and throughput per node.

5.3.2.1 PageRank

The following graph 5.19 represents a running time performance comparison for PageRank running on Hadoop, Spark and HAMR. And Table 5.9 summarizes HAMR's speedup over Hadoop and Spark on the same data sets.

The input data sets are range from 2 million pages to 30 million pages and the input sizes are range from 1G to 19.9 GB. The workload is run with 3 iterations.

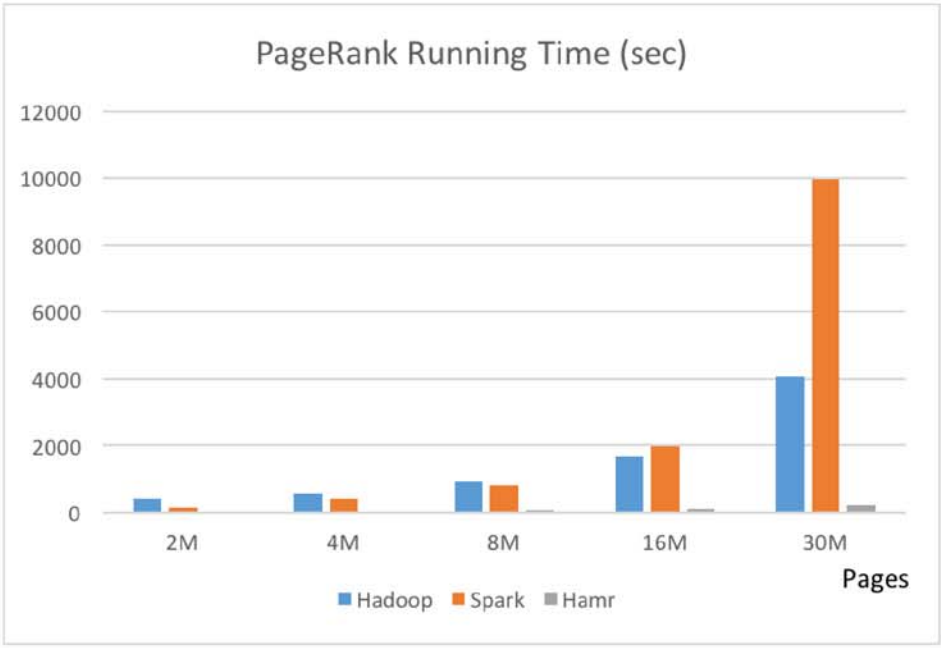
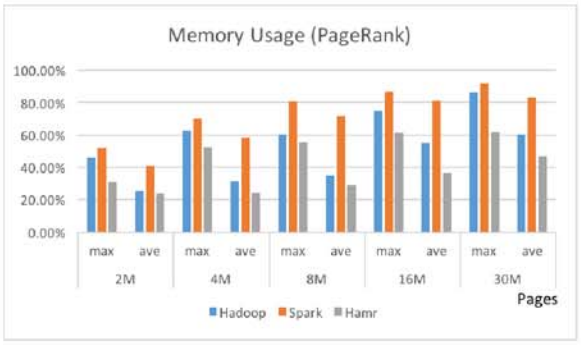


Figure 5.19: Running Time Comparison for PageRank Running on Hadoop, Spark and HAMR

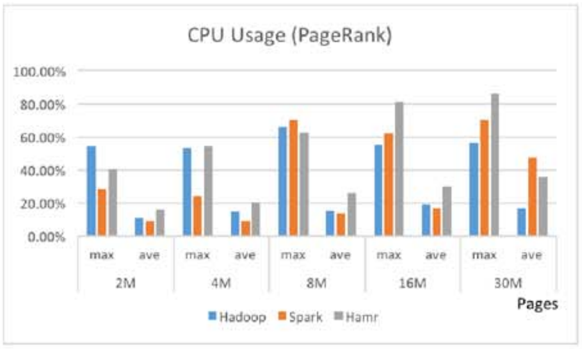
Input	**2M**	**4M**	**8M**	**16M**	**30M**
Hadoop	405	575	926	1691	4078
Spark	154	404	834	1965	9953
HAMR	36	46	72	118	211
Speedup Over Hadoop	11.25	12.50	13.86	14.33	19.33
Speedup Over Spark	4.28	8.78	11.58	16.65	47.17

Table 5.9: HAMR's Speedup over Hadoop and Spark on Running PageRank

HAMR is more efficient than Hadoop and Spark in running PageRank algorithm. It is 19x faster than Hadoop. It has up to 47x faster than Spark when the input size is large because Spark's performance is restricted by the memory. HAMR can execute with larger input size than Spark in memory. As shown in Figure 5.20a, HAMR has up to 50 % improvement in memory utilization than Spark and it is even memory efficient than Hadoop. However, HAMR costs more CPU resource compared with Hadoop and Spark. It takes around 2x more CPU usage than Spark when the input is less than 16 million pages. HAMR has extremely higher throughput than Hadoop and Spark as represented in Figure 5.21. The maximum throughput is around 95M per second at input equals to 30M that is 19x more than Hadoop and 47x more than Spark.

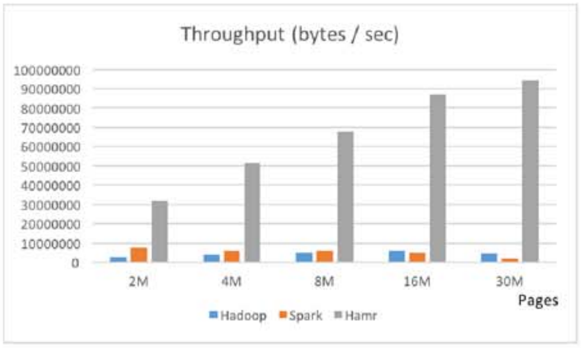


(a) Memory Usage Percentage

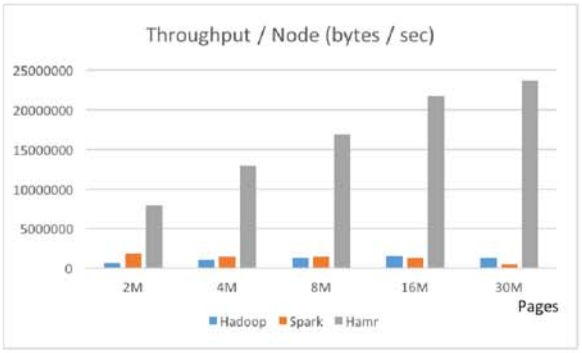


(b) CPU Usage Percentage

Figure 5.20: Memory and CPU Comparison for PageRank Running on Hadoop, Spark and HAMR



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.21: Throughput Comparison for PageRank Running on Hadoop, Spark and Hamr

5.3.2.2 WordCount

The following graph 5.22 represents a running time performance comparison for WordCount running on Hadoop, Spark and HAMR. And Table 5.10 summarizes HAMR's speedup over Hadoop and Spark on the same data sets.

The input data sizes are range from 2 GB to 64 GB. HAMR has improvement than Hadoop in executing WordCount workload with an average speedup around 1.1x. Spark has better performance than HAMR but the advantage is getting smaller with larger input size. Spark is 1.44x faster than HAMR when the input is 4 GB but

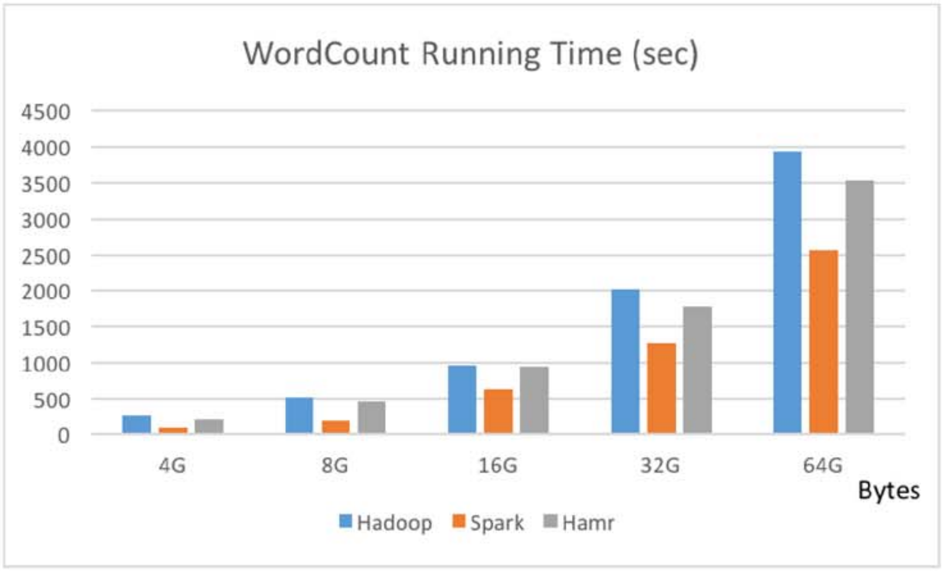
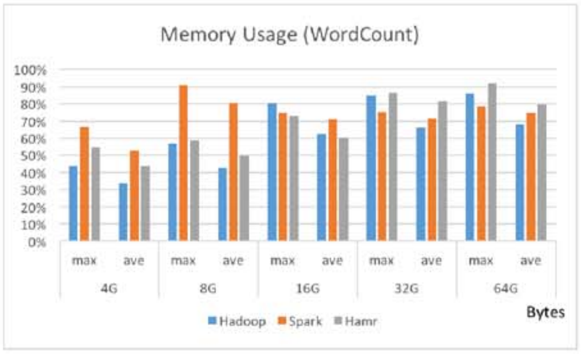


Figure 5.22: Running Time Comparison for WordCount Running on Hadoop, Spark and HAMR

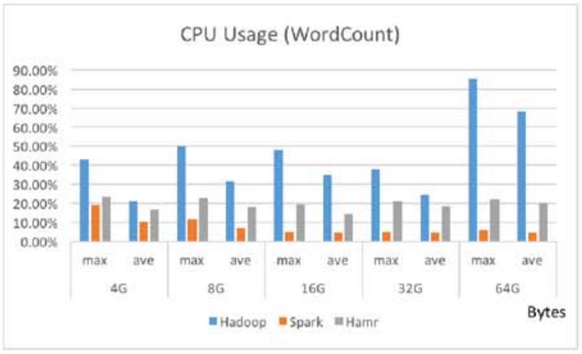
Input	**4G**	**8G**	**16G**	**32G**	**64G**
Hadoop	265	506	971	2023	3935
Spark	96	207	631	1273	2565
HAMR	221	452	952	1777	3536
Speedup Over Hadoop	1.20	1.12	1.02	1.14	1.11
Speedup Over Spark	0.43	0.46	0.66	0.72	0.73

Table 5.10: HAMR's Speedup over Hadoop and Spark on Running WordCount

decreases to 1.38x faster when the input is 64 GB. As shown in Figure 5.23, HAMR is more CPU efficient than Hadoop. The maximum CPU consumption for HAMR is less than half of Hadoop in general. When the input equals to 64 GB, the maximum CPU usage for Hadoop is 4x more than HAMR and the average CPU usage is 3.5x more than HAMR. Figure 5.24 represents the throughput comparison between Hadoop, Spark and HAMR. Spark has the highest throughput which is more than twice higher Hadoop and HAMR when the input is smaller than 16 GB.

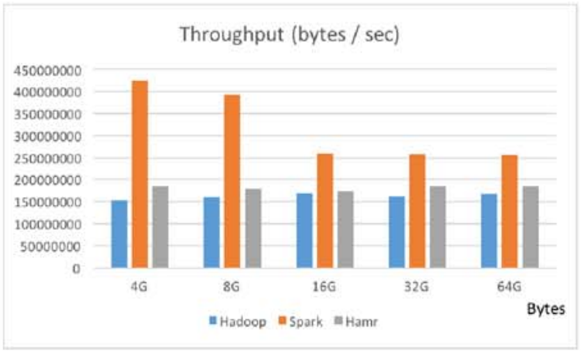


(a) Memory Usage Percentage

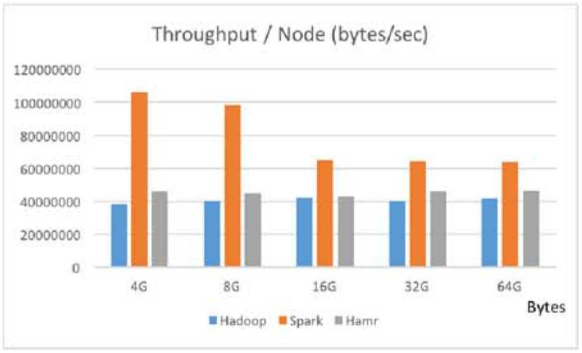


(b) CPU Usage Percentage

Figure 5.23: Memory and CPU Comparison for WordCount Running on Hadoop, Spark and HAMR



(a) Throughput (bytes/s)



(b) Throughtput per node (bytes/s)

Figure 5.24: Throughput Comparison for WordCount Running on Hadoop, Spark and Hamr

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In our experiment, we set up a platform for running big data applications. Also, we selected representative benchmarks from industry and academia, that stress different aspects of big data runtimes. By comparing the experiment results for running different benchmarks, we found that Spark is faster than Hadoop by using in memory processing. Spark is very efficient about machine learning and web search such as K-means and Naive Bayes since it is design to implement applications that apply a function repeatedly to the same dataset. For PageRank, Spark runs faster with small input size. It begins to fail behind Hadoop with 16 million pages input because of the memory restriction. The maximum memory usage has reached 100 percent. Spark is faster on WordCount. For Sort and TeraSort, Spark runs faster with large input. However, Spark consumes more memory resource than Hadoop. Also, when the input size is large and the system does not have abundant memory, the performance is restricted and can be slower than Hadoop.

Therefore, Spark is a good choice for implementing iterative applications. For programs with huge input size but not enough memory, Hadoop should be considered. Lei Gu and Huan Li also have a similar conclusion with us in paper “Memory or Time: Performance Evaluation for Iterative Operation on Spark and Hadoop” [17].

HAMR also provides improvements than Hadoop in both two benchmarks. The improvements on speedup, memory utilization and CPU utilization are contributed by the data flow and Flowlet technology. Compared with Spark, HAMR is efficient on the consumption of memory and CPU resource. HAMR is faster than Spark in implementing PageRank but slower in implementing WordCount. The performance of running time is not influenced by the memory capacity on HAMR cluster.

HAMR provides better performance than Hadoop. To compare with Spark, we still need more experiments.

6.2 Future Work

As our future work, we plan to set up Hadoop, Spark and HAMR on a bigger cluster to test the scalability of each platform. Also, we want to increase the memory capacity of the clusters to explore the influence of memory restriction on running time of Spark. To analyze the performance of HAMR distribution, we need to run more testing programs.

Also, we plan to design an intelligent system that can help us to choose a platform and the configuration parameters based on the applications and the input data sizes to get the optimized performance.

BIBLIOGRAPHY

[1] Apache spark document. http://spark.apache.org/docs/1.3.1/.

[2] Apache zookeeper. https://zookeeper.apache.org/.

[3] Hamr document. http://hamrtech.com/docs.php?page=documentation.

[4] Hibench. https://github.com/intel-hadoop/HiBench.

[5] Introduction to yarn and mapreduce 2. http://www.slideshare.net/cloudera/introduction-to-yarn-and-mapreduce-2.

[6] Pivotal rabbitmq. http://www.vmware.com/products/vfabric-rabbitmq/overview.

[7] Powered by hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[8] Powered by spark. https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark.

[9] Rabbitmq. https://www.rabbitmq.com/.

[10] Spark streaming programming guide. http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[11] Executive summary: A universe of opportunities and challenges. http://www.emc.com/leadership/digital-universe/2012iview/executive-summary-a-universe-of.htm, December 2012.

[12] Hamr – beyond mapreduce. http://www.etinternational.com/index.php/news-and-events/press-releases/hamr-beyond-mapreduce/, August 2014.

[13] Apache hadoop document. http://hadoop.apache.org/docs/r2.7.1/index.html, June 2015.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *ACM*, pages 19–22, October 2003.

[16] Satish Gopalani and Rohan Arora. Comparing apache spark and map reduce with performance analysis using k-means. *International Journal of Computer Applications*, 113(1), March 2015.

[17] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. *IEEE International Conference*, 2013.

[18] Josh James. How much data is created every minute. https://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute/, June 2012.

[19] Josh James. Data never sleeps 2.0. https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/, August 2014.

[20] Josh James. Data never sleeps 3.0. https://www.domo.com/blog/2015/08/data-never-sleeps-3-0/, August 2015.

[21] U.S. Department of Energy. The opportunities and challenges of exascale computing. Fall 2010.

[22] Naniel A. Reed and Jack Dongarra. Exascale computing and big data: The next frontier. *Communications of the ACM*, 58(7):56–68, July 2015.

[23] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[24] Hai-Tao Wei, Jun-Qing Yu, Hua-Fei Yu, and Ming-Kang Qin. A method on software pipelined parallelism for data flow programs. *Jisuanji Xuebao(Chinese Journal of Computers)*, 34(5):889–897, 2011.

[25] Haitao Wei, Mingkang Qin, Weiwei Zhang, Junqing Yu, Dongrui Fan, and Guang R Gao. Streamtmc: Stream compilation for tiled multi-core architectures. *Journal of Parallel and Distributed Computing*, 73(4):484–494, 2013.

[26] Haitao Wei, Hong Tan, Xiaoxian Liu, and Junqing Yu. Streamx10: a stream programming framework on x10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, page 1. ACM, 2012.

[27] Haitao Wei, Junqing Yu, Huafei Yu, and Guang R Gao. Minimizing communication in rate-optimal software pipelining for stream programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 210–217. ACM, 2010.

[28] Haitao Wei, Junqing Yu, Huafei Yu, Mingkang Qin, and G.R. Gao. Software pipelining for stream programs on resource constrained multicore architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2338–2350, Dec 2012.

[29] Haitao Wei, Stéphane Zuckerman, Xiaoming Li, and Guang R Gao. A dataflow programming language and its compiler for streaming systems. *Procedia Computer Science*, 29:1289–1298, 2014.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauey, Michael J. Franklin, Scott Shenker, and Ion Stoica. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. USENIX, San Jose, CA, 2012.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Appendix A

MANUAL FOR HADOOP, SPARK AND HAMR INSTALLATION

A.1 SSH keyless setup

Hadoop, Spark and HAMR cluster require SSH keyless setup for every node in the cluster. Follow the 5 steps to setup SSH login without password from fatnode0 to fatnode1:

1. Log in fatnode0 and generate a pair of authentication keys.

   $ ssh-keygen -t rsa

2. Stay on fatnode0 and create .ssh directory on the fatnode1 by the following command.

   $ ssh username@fatnode1.capsl.udel.edu mkdir –p .ssh

3. Copy the public key to fatnode1.

   $ cat .ssh/id_rsa.pub | ssh username@fatnode1.capsl.udel.edu 'cat ≫ .ssh/authorized_keys'

4. Change the file permission for the key file and the .ssh directory.

   $ ssh username@fatnode1.capsl.udel.edu "chmod 700 .ssh; chmod 640 .ssh/authorized_keys"

5. Login to fatnode1 from fatnode0 without entering the password

   $ ssh username@fatnode1

6. A keyless login has been created from fatnode0 to fatnode1. Repeat steps from 2 to 5 for fatnode0, fatnode2 and fatnode 3 to create the keyless login from fatnode0 to fatnode0, fatnode2 and fatnode3.

Repeat the steps from 1 to 6 on fatnode1, fatnode2 and fatnode3 to set up SSH login without password for all nodes. This manual can also be found here: http://www.tecmint.com/ssh-passwordless-login-using-ssh-keygen-in-5-easy-steps

A.2 Hadoop setup

1. Hadoop requires SSH keyless setup for every node in the cluster. Follow the SSH keyless setup section as the prerequirement.

2. Download the Hadoop file from its website: http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-2.7.1/hadoop-2.7.1.tar.gz

3. Extract the tar file on the node.

   ```
   $ tar –xvf Hadoop-2.7.1.tar.gz
   ```

4. Update the .bashrc file for each node. Open the .bashrc file and copy the following to the file. We assume that the JAVA jdk has already been installed on the cluster. And the JAVA_HOME has been set pointing to the JAVA directory.

   ```
   $ export HADOOP_HOME=<path to Hadoop>
   $ export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$JAVA_HOME/
   bin:$PATH
   ```

5. Use the following command to get the .bashrc file work.

   ```
   $ source .bashrc
   ```

6. Configuration.

   (a) Open this file: HADOOP_HOME/etc/hadoop/core-site.xml and copy the following into the file between two <configuration>. The hadoop.tmp.dir is used as the base for temporary directories locally. You can change the path as needed.

   ```
   <property>
   <name>hadoop.tmp.dir</name>
   <value>/home/lu/tmp/ </value>
   <description >A base for other temporary directories.</description>
   </property>

   <property>
   <name>fs.defaultFS</name>
   <value>hdfs://fatnode0:9000</value>
   </property>
   ```

   (b) Open this file: HADOOP_HOME/etc/hadoop/mpred-site.xml and copy the followings into the file between two <configuration>.

   ```
   <property>
   <name>mapreduce.framework.name</name>
   ```

```
<value>yarn</value>
</property>
```

(c) Open this file: HADOOP_HOME/etc/hadoop/hdfs-site.xml and copy the followings into the file between two <configuration>. The dfs.replication is used to decided the number of replications for the hdfs. The dfs.namenode.name.dir and dfs.datanode.data.dir is pointed to the local temporary directory for the namenode and datanode. The path should be changed as needed.

```
<property>
<name>dfs.replication</name>
<value>3</value>
</property>

<property>
<name>dfs.namenode.name.dir</name>
<value>/home/lu/hadoop/tmp/dfs/name</value>
</property>

<property>
<name>dfs.datanode.data.dir</name>
<value>/home/lu/hadoop/tmp/dfs/data</value>
</property>
```

(d) Open this file: HADOOP_HOME/etc/hadoop/yarn-site.xml and copy the followings into the file between two <configuration>.

```
<property>
<name>yarn.resourcemanager.hostname</name>
<value>fatnode0</value>
</property>

<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>

</property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

```
<property>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>false</value>
</property>
```

(e) Open the file: HADOOP_HOME/etc/hadoop/slave. Copy the hostname of each slave node into this file as the following example.

```
fatnode0
fatnode1
fatnode2
fatnode3
```

7. Repeat the installation and configuration steps from 1 to 6 on each node.
8. If there is another installation of Hadoop working on the cluster, you may have the port binding error. The following files can be changed to solve that problem.

(a) HADOOP_HOME/etc/hadoop/hdfs-site.xml

```
<property>
<name>dfs.namenode.http-address</name>
<value>0.0.0.0:51070</value>
</property>

<property>
<name>dfs.datanode.http.address</name>
<value>0.0.0.0:51075</value>
</property>

<property>
<name>dfs.datanode.address</name>
<value>0.0.0.0:51010</value>
</property>

<property>
<name>dfs.datanode.ipc.address</name>
<value>0.0.0.0:51020</value>
</property>

<property>
<name>dfs.namenode.secondary.http-address</name>
```

```
<value>fatnode0:51090</value>
</property>

<property>
<name>dfs.namenode.secondary.https-address</name>
<value>0.0.0.0:51091</value>
</property>
```

(b) HADOOP_HOME/etc/hadoop/yarn-site.xml

```
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>fatnode0:8131</value>
</property>
```

9. Format the HDFS fily system

   ```
   $ hdfs namenode –format
   ```

10. Start the Hadoop cluster and the YARN

    ```
    $ HADOOP_HOME/sbin/start-all.sh
    ```

11. Start the Hadoop cluster and the YARN

    ```
    $ HADOOP_HOME/sbin/stop-all.sh
    ```

A.3 Spark Setup

1. Spark SSH keyless setup for every node in the cluster. Follow the SSH keyless setup section as the prerequirement.

2. Install Scala on each node. Download Scala from its website: http://www.scala-lang.org/download/. Extract it to local file.

   ```
   $ tar –zxvf scala-2.11.7.tgz
   ```

3. Update the .bashrc file for each node.

   ```
   export SCALA_HOME=/home/lu/scala-2.11.7
   export PATH=$PATH:$SCALA_HOME/bin
   ```

4. Use the command to get the .bashrc file work

   ```
   $ source .bashrc
   ```

5. Download the Spark from its website: http://spark.apache.org/downloads.html. And extract it. I downloaded the spark-1.3.1-bin-hadoop2.6.tgz.

   ```
   $ tar –zxvf spark-1.3.1-bin-hadoop2.6.tgz
   ```

6. Configuration

   (a) Copy the spark-env.sh.template to spark-env.sh.

   (b) Open this file: SPARK_HOME/conf/spark-env.sh. Copy the following setting into the file. The path to each file should be changed as needed.

   ```
   export SCALA_HOME=/home/lu/scala-2.11.7
   export JAVA_HOME=/share/pinogal/opt/jdk1.7
   export HADOOP_HOME=/home/lu/hadoop-2.7.1
   export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
   export SPARK_MASTER_IP=fatnode0
   export SPARK_LOCAL_DIR=/home/lu/spark-1.3.1-bin-hadoop2.6
   export SPARK_DRIVER_MEMORY=1G
   ```

   (c) Open the file: SPARK_HOME/conf/slaves. Add the hostname of each slave node into it.

   ```
   fatnode0
   fatnode1
   fatnode2
   fatnode3
   ```

7. Repeat the install and configuration steps on each node.

8. Start Hadoop and YARN before staring Spark.

   ```
   $ start-all.sh
   ```

9. Start the Spark

   ```
   $ SPARK_HOME/sbin/start-all.sh
   ```

10. Stop the Spark

   ```
   $ SPARK_HOME/sbin/stop-all.sh
   ```

A.4 HAMR Setup

1. The installation of HAMR is provided on HAMR website and can be found by the following address: http://hamrtech.com/docs.php?page=documentation

2. HAMR requires at least on install of ZooKeeper for a cluster. It is important to install version 3.4.6 or later.

   (a) Download the correct version from: http://zookeeper.apache.org/releases.html
   (b) Extract the file
       $ tar –xzvf zookeeper-3.4.6.tar.gz
   (c) Copy zoo_sample.cfg to zoo.cfg
       $ cp zoo_sample.cfg zoo.cfg

3. HAMR requires RabbitMQ to send messages between running HAMR processes.it is important to install version 3.3.5 or later.
   (a) Download the correct version from: https://www.rabbitmq.com/releases/rabbitmq-server/v3.3.5/.
   (b) Install according to RAbbitMQ's documentation: http://www.rabbitmq.com/download.html
   (c) If a RabbitMQ server was already installed, the following setup steps are needed on each host running RabbitMQ to ensure that the new software is correctly installed. Create the file: RABBIT_HOME/etc/rabbitmq/rabbitmq-env.conf and add the following lines:
       RABBITMQ_HOME_PORT=<nnnn>
       RABBITMQ_NODENAME=hamr_rabbit@<hostname>

4. Download HAMR from http://hamrtech.com/download.php

5. Extract the file

   $tar –xzvf hamr-0.4.1.tgz

6. Open the file: HAMR_HOME/conf/hamr-site.xml and edit the following properties. The path can be changed as needed.

   <property name='rabbitmq.connection.server'>fatnode0, fatnode1, fatnode2, fatnode3</property>

   <property name='zookeeper.connection.servers'>fatnode0:2181, fatnode1:2181, fatnode2:2181, fatnode3:2181</property>

   <property name='cluster.home'>/home/lu/hamr-0.4.1</property>

   <property name='license.file'>/home/lu/hamr-0.4.1/conf/license.xml</property>

   <property name='cluster.tmpdir'>/share/lu/tmp/hamr</property>

   <property name='cluster.java'</share/pinogal/opt/jdk1.7/bin/java</property>

   <property name='cluster.javaArg'>

   <value>-Djava.library.path=/home/lu/hadoop-2.7.1/lib/native</value>

   </property>

<property name=‘rn.horn.master.host’>fatnode0</property>

<property name=‘rn.horn.slave.host’>

<value>fatnode0</value>
<value>fatnode1</value>
<value>fatnode2</value>
<value>fatnode3</value>

</property>

7. Repeat the installation and configuration steps from 3 to 5 on each node.

A.5 Running WordCount

This example assumes that Hadoop, Spark and HAMR are already installed and configured.

1. The HiBench requires an installation of Maven on the same node of HiBench. Download the Maven from: https://maven.apache.org/download.cgi

2. Create the MAVEN_HOME in the .bashrc file and add it to the PATH.

   ```
   export MAVEN_HOME=/home/lu/apache-maven-3.3.3
   export PATH=$MAVEN_HOME/bin:$PATH
   ```

3. Update the .bashrc file

   ```
   $ source .bashrc
   ```

4. Install Hibench benchmark suite

   (a) Get the HiBench from https://github.com/intel-hadoop/HiBench. Extract it.

   (b) Build HiBench
   ```
   $ HIBENCH_ROOT/bin/build-all.sh
   ```

   (c) Configuration. Create and edit the file: HIBENCH_ROOT/conf/99-user_defined_properties.conf.
   ```
   $ cd conf
   $ cp 99-user_defined_properties.conf.template  99-user_defined_properties.conf
   ```

   (d) Set the following properties the path can be changed as needed.
   ```
   hibench.hadoop.home        /home/lu/hadoop-2.7.1
   hibench.spark.home         /home/lu/spark-1.3.1-bin-hadoop2.6
   hibench.hdfs.master        hdfs://fatnode0:9000
   hibench.spark.master       yarn-client
   hibench.hadoop.version     hadoop2
   hibench.hadoop.release     apache
   ```

hibench.masters.hostnames	fatnode0
hibench.slaves.hostnames	fatnode0 fatnode1 fatnode2 fatnode3
hibench.spark.version	spark1.3

5. The size of input data can be chosen in HIBENCH_HOME/conf/99-user_defined_properties.conf by changing this property: hibench.scale.profile. You can also added more scale profiles in HIBENCH_HOME/conf/10-data-scale-profile.conf

6. Generate the input data for WordCount

   $ cd HIBENCH_ROOT/workloads/wordcount/prepare

   $ ./prepare.sh

7. Run the WordCount on Hadoop

   $ cd ../mapreduce/bin/

   $ ./run.sh

8. To optimize Hadoop, the YARN and MapReduce configuration such as the amount of memory, the number of CPU cores and the number of disks, can be set according to this website: `http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html`. These files are located in HADOOP_HOME/etc/hadoop/mapred-site.xml; HADOOP_HOME/etc/hadoop/yarn-site.xml.

9. Run the WordCount on Spark

   $ cd ../../spark/scala/bin/

   $ ./run.sh

10. Memory, cores and executors number Spark can be tuned in this file: HIBENCH_HOME/conf/99-user_defined_properties.conf

    hibench.yarn.executors.num
    hibench.yarn.executors.cores
    spark.executors.memory
    spark.driver.memory

11. Run the WordCount on HAMR

    $ cd

    $ hamr horn com.etinternational.hamr.examples.wordcount.WordCount
    /HiBench/WordCount/Input -o <path>/myOutputfile.out