August 2015

Data Analysis With Map Reduce Programming Paradigm

Mandana Bozorgi
University of Nevada, Las Vegas

DATA ANALYSIS WITH MAP REDUCE PROGRAMMING PARADIGM

by

Mandana Bozorgi

Bachelor of Engineering (B.Sc.)
University of Islamic Azad Tehran
2000

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
November 2014

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 24, 2014

This thesis prepared by

Mandana Bozorgi

entitled

Data Analysis with Map Reduce Programming Paradigm

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Kazem Taghva, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Ajoy Datta, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Ebrahim Saberinia, Ph.D.
Graduate College Faculty Representative

Abstract

Abstract In this thesis, we present a summary of our activities associated with the storage and query processing of Google 1T 5-gram data set. We first give a brief introduction to some of the implementation techniques for the relational algebra followed by a Map Reduce implementation of the same operators. We then implement a database schema in Hive for the Google 1T 5-gram data set.

The thesis will further look into the query processing with Hive and Pig in the Hadoop setting. More specifically, we report statistics for our queries in this environment.

Acknowledgements

" I would like to express the deepest appreciation to my committee chair professor Dr. Taghva, who without his help this work would not be possible. Also, special thanks to my committee members Dr. Datta, Dr. Gewali and Dr. Saberinia to accept to be part of this defense."

Mandana Bozorgi

University of Nevada, Las Vegas
November 2014

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is an experiment in the storage and query processing of the Google Web1T-5gram data set. From now on, we will use *Web1T-5gram* instead of *Google Web1T-5gram*. *Web1T-5gram* addresses the frequency of words in a corpus: "A corpus is a large, principle collection of naturally occurring examples of language stored electronically[1]".

In 2006, Google released the *Web1T-5gram* database based on 1 trillion words. This database keeps the frequency of *Unigram*, *Bigram*, *trigram*, *four-grams* and *five-grams*. In statistical language processing, this method is called an *N-gram* model[13]. The N-grams are used to train language models based on the hidden Markov paradigm. N-gram model is designed to keep track of the frequency of word sequences in a text. *Google* also created an open source software solution called *Web1T5-easy* based on a Relational Database system to manipulate the data.

In this thesis, we show an efficient method for indexing and storing the *Web1T-5gram* in Hive. The system Hive takes advantage of Hadoop's distributed clustering environment. The Hive and Hadoop can decrease the number of storage costs and reduce computational processing while also increasing reliability and speed on big data searches.

This thesis contains eight chapters. Chapter One is this *introduction*. Chapter Two is a background section is describing *Web1T-5gram* and *Web1T-easy*. Chapter Three provides details about the *MapReduce* programming paradigm. The MapReduce framework is widely used parallel computing platforms for processing big data. *MapReduce* was initially used by Google for indexing tasks, but later on other entities including, Yahoo, Facebook, Amazon, Ebay, CISCO, Intel, IBM and Zettaset use it for analytics on their data sets[3] . Chapter Four is an introduction to Big Data. The Big Data is typically a large dataset that's hard to process with traditional data processing methods.

Chapter Five focuses on the structure of *Hadoop* and also introduces data warehouse system

layers on top of Hadoop like *Hive* and *Pig*. Chapter six describes Hive and Pig and their differences. In Chapter Seven, we show our implementation in Hive. Chapter Eight addresses the conclusion and future recommendations.

Chapter 2

Google Web 1T-5gram

Knowing the probability of how many times a sequence of words in a specific context has occurred is important in *Statistical Language Processing*. One of the best-known language modeling tools is N-gram. *Google Web1T-5gram* database contains the frequency of N-gram words extracted from one trillion words of English text. This dataset includes one, two, three, four, and five-grams that have a frequency greater than 40. Because of the dataset's sheer size, *Web1T-5gram* is now used by many researchers for building better language modeles[5]. *Web1T-5gram* stored in 6 CDs for the total size of 24.4 GB gzip. In Table 2.1., we show the list of text files on each N-gram.

Table 2.1: number of gzip(text file) for each N-grams based on Google Web1T-5gram CDs

$N-gram$	$NumberOfGzipFiles$
$Bigrams$	31
$Trigrams$	97
$Four-grams$	131
$Five-grams$	117

In *Web1T-5gram*, the text was tokenized with Penn Treebank tokenization.In the same manner as Penn tokenizer, words are separated by whitespace (e.g. *today is*, will be two tokens: today(token1) and is(token2)). Verb contraction tokenizer (e.g. *mother's* converting to two tokens: mother(token1)) and 's(token2) or *won't* converting to two tokens, wo(token1) and n't(token2)). Marks the sentences with $< S >$and $< /S >$. The special token $< UNK >$ shows words that occurred less than 200 times. The difference between Web1T-easy tokenizer and Penn tokenizer is that text such as email addresses, URLs, hyphenated compound words, and dates counts as single token. Table 2.2 is a sample of Web1T-5gram frequency.

Web1T-5gram has $1,024,908,267,229$ tokens (cf. Table 2.3).

Table 2.2: sample Of Web1T-5gram Frequency(Web1T-5gram,extracted from 4gm-0014.gz)

$4gram$	$word1$	$word2$	$word3$	$word4$	$Frequency$
1 tablespoon 8 ounces	1	*tablespoon*	8	*ounces*	42
1 tablespoon ; $< S >$	1	*tablespoon*	;	$< /S >$	202
1 tablespoon $< UNK >$ *sauce*	1	*tablespoon*	$< UNK >$	*sauce*	65
1 tablespoon all -	1	*tablespoon*	*all*	–	4008

Table 2.3: Number of tokens in Google 1T-5grams

$Numberoftokens:$	$1,024,908,267,229$
$NumberofSentences:$	$95,119,665,584$

Google has generated around 1 trillion word tokens from public web pages. This corpus is 10,000 times larger than the *British National Corpus* (BNC), which is extracted from 100 million words. Besides, Web1T-5gram is 2500 times larger than the *Corpus Of Contemporary American English* (COCA), which is extracted from 385 million words. Examples of corpora and their sizes are presented in Table 2.4.

Table 2.4: List of corpora with size in billion tokens[8]

$CorpusName:$	$Tokenssize$
BNC	$0.1G$
$WP500$	$0.2G$
$Wackypedia$	$1.0G$
$ukWac$	$2.2G$
$WebBase$	$3.3G$
$UKCOW$	$4.0G$
LCC	$0.9G$
$Web1T5$	$1000.0G$

The *Web1T-5gram* has some disadvantages raw data. For example, raw data lacks linguistic annotation . The data also omits n-grams with the frequency less than 40. Without indexing, a query requires a linear scan of the entire data. One solution that can increase the speed of the searches is to store the data in a relational database system. With the relational implementation, we have the ability to index the n-grams to increase the search process.

Google has also built an open-source, user-friendly software "Web1T-easy that is a collection of Perl scripts for indexing and querying the database with the open-source database engine SQLite." [5] In indexing, words are first normalized to lower case, and then each word is given word-id for decreasing the database size and for increasing the search performance. Then each 5-gram is stored as a sequence of 5 integers.

Since working with SQL may not be suitable for most users, Google has built Web 1T-easy, a user-friendly open-source software that is translated from SQL code.

With the *Web1T-easy* interface, we have the ability to run queries on *Web1T-5gram* database and show the highest frequency of a specific search. *Web1T-easy* is designed to support queries on multi-words patterns with words frequencies [4]

(The MapReduce model is designed to work with parallelization- for instance, it can determine the number of times a specific word appears in a data set.)

Chapter 3

MapReduce

In 2004, Google introduced MapReduce, which provides its user with a useful way to work with both parallel paradigm processing and run time programming[7]. In 2005, Nutch developers created Nutch file system(NDFS) based on MapReduce. Early in 2006, Lucene created Hadoop based on NDFS and MapReuce. In 2008, Yahoo! announced a search index based on MapReduce with over 10,000 core Hadoop clusters [12] .

MapReduce which uses the runtime programming and can work across a large cluster of computers. It is one of the most frequently used parallel computing platforms for processing big data . MapReduce is reliable, easy-to-use, and easy-to-program for distributed paradigm and occur in three stages [2]: *map stage, shuffle stage, reduce stage.*

The MapReduce programs take advantage of the cluster of commodity servers. First it divides a job among a group of servers known as Mappers. It then uses a shuffle to distribute the output of Mappers to the group of servers known as Reducers. The Mapper assumption is that the data can be parallel processed by Mapper and Reducer. Google runs approximately 1,000 MapReduce jobs daily[8].

The following describes the three stages of MapReduce on a word count example. In this example, we describe a MapReduce application that computes the words with the highest frequency in a given set of text.

3.1 Map Phase

In the following word count example, the input is files of text. We need to know how many times each word has occurred within this specific corpus. Whitespace separates the words.

Consider the following example where each line represents a file of text:

today is Monday

today is sunny

today is a sunny Monday

To counting the words in this example, Mapper will first split the words by whitespaces, and the output of Mapper is the pair of words with their frequencies know as *(key, value)* pair. Several mappers should come across the words based on our text. MapReduce transform output of our Mapper contains multiple *(key, value)* pairs.

Table 3.1: Map Function Output

Key	$Value$
$today$	1
is	1
$Monday$	1
$today$	1
is	1
$sunny$	1
$today$	1
is	1
a	1
$sunny$	1
$Monday$	1

You can see the diagram of this example on figure 3.1.

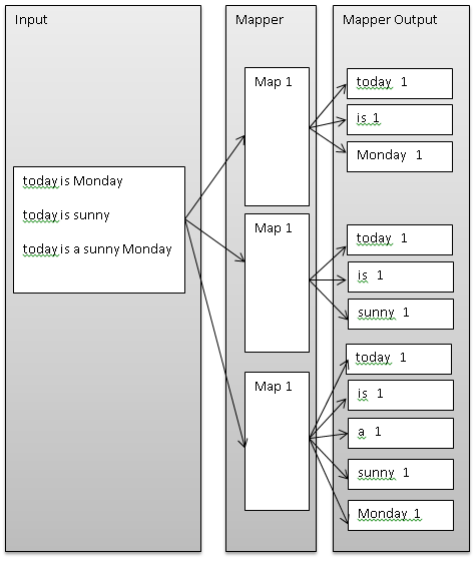

Figure 3.1: Map Function

The code below shows the word count example in Python. ”

```
#!/usr/bin/env python
""" Mapper, using Python."""

import sys

def read_input(file):
    for line in file:
        # split the line into words
        yield line.split()

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_input(sys.stdin)
    for words in data:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step
        # tab-delimited; the trivial word count is 1
        for word in words:
            print '%s%s%d' % (word, separator, 1)

if __name__ == "__main__":
    main()
"[1]
```

3.2 Shuffle Phase

Shuffle stage is implemented by the system that sort and send all the *values* to the Reducer. Those pairs that have the same key go to the same Reducer.

[1] http://www.quuxlabs.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

Table 3.2: Shuffle Function Output

Key	$Value$
$today$	1
$today$	1
$today$	1
is	1
is	1
is	1
$sunny$	1
$sunny$	1
$Monday$	1
$Monday$	1
a	1

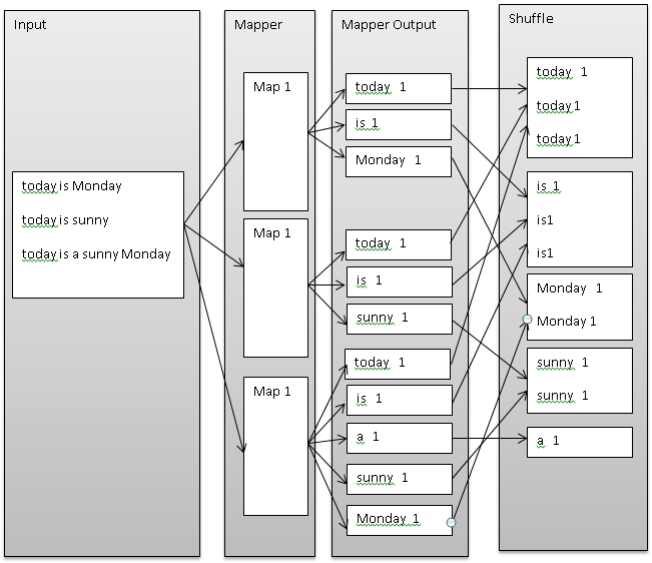

Figure 3.2: Shuffle Stage

3.3 Reduce Phase

Reducer joins and unites all intermediate values with the same intermediate *key*. Reducer is a program written by the programmer to sums the values for each key. Usually, Reducer produces Zero or one output value.

As we can see the result of reducer in below.

Reducer for the wordcount example in Python.

”

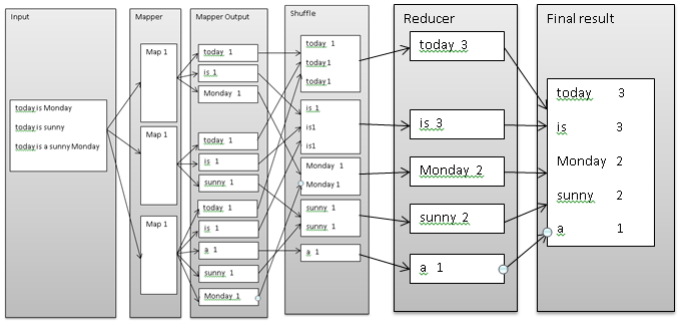

Figure 3.3: Reduce Stage

Table 3.3: Reduce Function Output

Key	*Value*
today	3
is	3
sunny	2
Monday	2
a	1

```
#!/usr/bin/env python
"""Reducer, using Python."""

from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator, 1)

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_mapper_output(sys.stdin, separator=separator)
    # groupby groups multiple word-count pairs by word,
    # and returns consecutive keys and their group:
    #   current_word - string containing a word (the key)
    #   group - yielding all ["&lt;current_word&gt;", "&lt;count&gt;"] items
```

```
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count = sum(int(count) for current_word, count in group)
            print "%s%s%d" % (current_word, separator, total_count)
        except ValueError:
            # count was not a number, so discard this item
            pass

if __name__ == "__main__":
    main()
```

" [2]

For testing the Mapper code for small data sets on the local machine, we can simply run this command line (on Mac):

```
$ cat /../ (the text file path)| /..../mapper.py(the path that mapper.py
  file has been saved) |sort -k,1 | /.../reducer.py(Path that reducer.py is saved)
```

[2]http://www.quuxlabs.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

Chapter 4

Big Data

The role of technology in collecting data from different devices have led to a vast amount of data. This consequently has created many opportunities to analyze and report on discoveries of certain patterns in the data. Because of their large size and complexity of this data, managing the data with traditional approaches is not suitable. This includes challenges associated with storage and processing.

To illustrate the big data problem, we consider the growth in the number of published pages on the Internet. In 1998, according to Google there were about one million pages. In 2000, this number had increased to one billion. In 2008, Google reported approximately one trillion web pages, and now, in 2014, estimated to be around 30 trillion. In the last six years, it increased up to 30 times as shown in Table 4.1.

Table 4.1: Number Of Google Unique Web Pages

$Year$	$NumberOfIndividualWebPages$
1998	$1,000,000$
2000	$1,000,000,000$
2008	$1,000,000,000,000$
2014	$30,000,000,000,000$

In 1998, Weiss and Indrukya [6] first reported some problems associated with the processing of the Big Data. In 2000, Diebold [4] published a paper on *Big Data* concepts and issues.

Big Data heavily depends on the capabilities of the systems for storage and processing. Traditional systems like *Relational Database* are not designed to handle smooth processing of Big Data. In particular, the big data challenges the processing speed, capturing and searching of unstructured data. It further has problems associated with sharing, transferring, analyzing and visualizing of big data.

In traditional database systems, data is structured and is stored in tables with fixed number

of columns. Each column has the specific data type. But in Big Data, we have variety of data formats like audio, video, and text that does not fit in the table formats.

In 2001, Doug Laney talked about measuring the *Big Data* with 3 V's [8], *Volume*, *Variety*, and *Velocity* as follows:

- *Volume* is related to the size of data.

- *Variety* refers to the number of data types.

- *Velocity* is related to the speed of processing.

According to 3V's model, Big Data challenges is not focusing just on the sheer amount of data, but from all three properties[9]. This model is extended by two more V's *Variability*, and *Veracity* as follows:

- *Variability* is designed as having an extremely large variety of data

- *Veracity* is focused on quality of Big Data. Having lots of incorrect data even if they processed fast can cause a lot of problems.

Chapter 5

Hadoop

Many companies have had different experiences working are with Big Data. It is important for them to have a method to analyze intensive data in an appropriate amount of time. With the use of Hadoop, working on Big Data computing becomes much easier for users. Hadoop is used in many organization such as Facebook and Yahoo!. Parallel processing with Big Data allows simultaneous access to multiple compute clusters. However, many organizations cannot afford to buy and maintain these clusters. Merging Hadoop with the cloud is a cost-effective solution. Hadoop is a data management framework that works on Big Data in the cloud.

5.1 Hadoop architecture

Hadoop requires processing massive amounts of raw input data. Distributed computing has been developed to process Big Data effectively. In distributed computing, each application runs on several compute nodes. The data is typically divided into several chunks that then processed in parallel on different compute node. This allows multiple tasks to run independently and eliminates the problem of the task waiting. They work both individually and parallel. Parallelism is a solution that works well with Big Data because it reads the data while also parallel processing that same data. Hadoop copies data a minimum of three times on at least three different machines, which provides valuable backup, and data redundancy. To help illustrate the Hadoop multiplication process, we will reference following the figure.

This is one of the many advantages of Hadoop. Another advantage is that if one of the systems fails, the other machines can continue processing because the data has been replicated.Another benefit of this paradigm is faster task processing speed. Multiple data reads and computations can occur in parallel instead of the most traditional process in which tasks are processed one at a time.

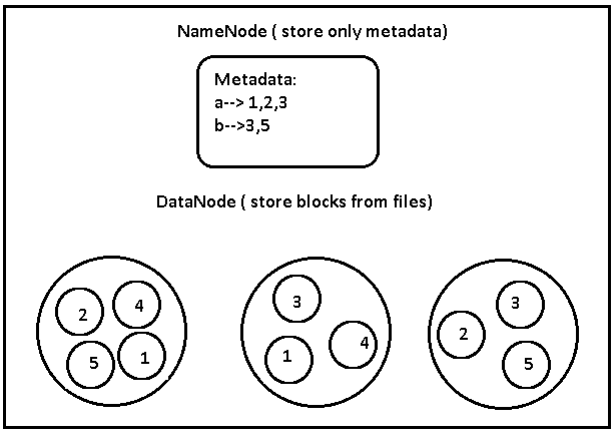

Figure 5.1: Name Node - Data Node

Hadoop is an open-source framework that was designed based on distributed computing for working with Big Data. Hadoop contains two components: HDFS (Hadoop Distributed File System) and MapReduce.

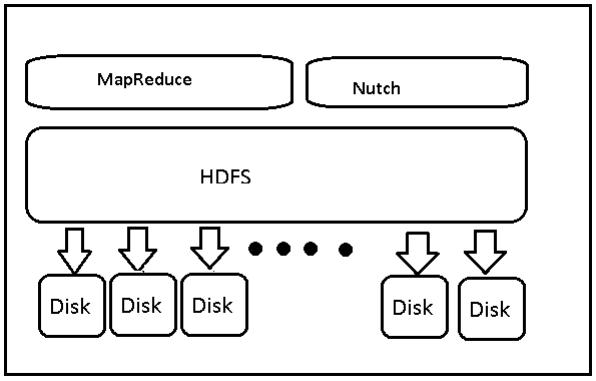

Figure 5.2: Nutch in Hadoop Distributed File System

Hadoop is the most popular implementation of MapReduce. Hadoop has a master-slave architecture; the master node is JobTracker, and the slave node is TaskTracker.

Also, Hadoop has been supported by GFS (Google File System), HDFS(Hadoop Distributed File System), and MapReduce. In Hadoop, HDFS works on master-slave architecture in which the master node directs application to multiple slave nodes, and slave nodes store the data. The master node cannot store data. In a large Hadoop cluster, the master node also serves both JobTracker and DataNode and TaskTracker and DataNode support each slave node. However, there are no slave nodes in small Hadoop clusters. The master nodes work alone with JobTracker and DataNode. When a MapReduce job is executed in Hadoop, Hadoop will first divide the input

file into several data blocks. Each block is typically 64 MB by default unless otherwise specified by the programer. Then these blocks are stored in a Hadoop Distributed File System(HDFS).

Each MapReduce job can run in parallel on slave nodes. The mapper function will input the raw (key, value) pairs into the shuffle that outputs the sorted data to the appropriate reducer function. The reducer function will then produce the final (key, value) pairs. There are multiple mappers and multiple reducers, which allows the map and reduce tasks to run simultaneously.

Each block sent to the mapper will divide files into individual (key,value) pairs. First, these output pairs are written into a memory buffer. Next they are written into a split file on a local disk .[2]. The shuffling process automatically organizes the (key, value) pairs into groups of identical keys prior to being sent to the reducer. For example, all keys titled "today" will be shuffled into the same partition. These shuffled files are then stored on a local machine. Each shuffle portion is then processed individually by the reducer function. The final output of the MapReduce job is a single file in which all identified keys have been tallied. The (key, pair) values consist of the key and the number of times that a key is counted in the data set. For example, let's say that the Reducer counts the key "today" a total of three times. The final (key, pair) value for this word would be "today 3". The reducer function runs in parallel processing.

The JobTracker node in Hadoop handles executing MapReduce jobs. JobTracker manages the mapper and reducer execution processes. TaskTracker performs all map and reduce tasks. JobTracker performs job scheduling (e.g., it leads and schedules jobs from different users) and task scheduling (e.g., how to assign tasks to TaskTracker). JobTracker will prefer to execute the local data first to reduce the data transfer across the network. This means that when we have task assignments and TaskTracker is available for that particular task, JobTracker will first try to find data blocks that are stored on local disk. If there is no data on the local disk or if JobTracker cannot find the data, then it will try to find data blocks starting with the nodes that are closest to the TaskTracker.

One of the Hadoop features is that it will make three copies of each file. HDFS stores these three copies separately for fault tolerance purposes and reduces the risk of losing data[2]. A reduce task consists of three stages: copy, sort, and reduce stages. In the copy stage, all of the data that produced by the mapper function will be copied. The sort stage will sort the data by keys. In the final stage, the reduce task will sort all of the (key, value) pairs identified sort stage and then store the output into a final output file. The reduce stage compiles the data into a single file and saves it to an HDFS.

5.1.1 fault tolerance in Hadoop

When Hadoop runs Big Data, there can be a risk of failure. Hadoop has been designed to have a fault tolerance mechanism, and if a failure occurs, Hadoop will try to repair it so that the failure will have minimal effect on the final output. We will discuss three of the most common types of failure and their solutions: task failure, TaskTrack failure, and JobTrack failure in Hadoop[2].

- The first common type of failure, task failure, is typically due to user error. For example, a programmer may have written bad MapReduce code. When a job runs from a poorly written MapReduce code, the TaskTracker will find task failure. TaskTracker will not perform the failed task and will notify the JobTracker about the failed task. Then the JobTracker will try to reschedule the failed task by pointing to a different TaskTracker and try to run the task again. In Hadoop, if the task fails four times then the entire job will fail and the user will be notified of the failure. However, the user will not be informed which particular task failed.

- The second type of failure, TaskTrack failure, is more dangerous than task failure. It occurs when JobTracker has not received any heartbeat message from TaskTracker for ten minutes. JobTracker will stop that TaskTracker job and will seek out another TaskTracker job. The user just receives a message that there is a task failure.

- JobTrack failure, the third common type of failure, is a severe error. Hadoop has a configuration option for this kind of problem that captures snapshots of jobs at periodic intervals. When a JobTracker failure occurs, Hadoop will use the snapshot to recover all of the jobs that ran during the failure time. Unlike task failure and TaskTrack failure, JobTrack failure will produce an error message for the user.

5.1.2 HDFS

Hadoop file system is created based on Google File System(GFS). HDFS is a distributed file system that execute on top of the file system on all the machines in the cluster. HDFS is one of the Hadoop components that help to have fault tolerance. Another HDFS job is working on Big Data, that the typical data will be around terabyte up to few petabyte. HDFS is a built-in support for working on massive data sets.

HDFS assumes that the data is written one time, and it will be read several times; Data replication is done by HDFS software layer to distribute the input file and store it in DataNode; This facility let the application to have access to local data, and also if the Data-Node crashed still keep a copy of the data in another Data-Node.

Hadoop Distributed File System wrote in Java for storing massive data sets in a large cluster. Hadoop is saving each particular file as a sequence of blocks; the input file is broken into several blocks. Each block has a size 64MB by default, but it's changeable by the user; also these blocks also replicated usually three times. Replication is one of the HDFS advantages. It is replicate to make sure the reliability, network utilization, and availability of blocks[1]. The HDFS has master-slave architecture, Name-Node (master) and Data-Node(slave).

HDFS architecture

HDFS supported by Name Node, a single node that has managed the access the files, and NameNode will store data into blocks. In next figure, you can see the HDFS architecture.

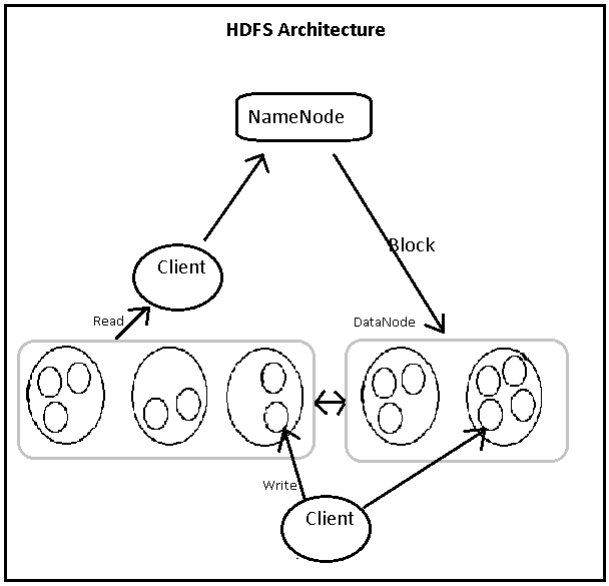

Figure 5.3: HDFS architecture

5.1.3 NameNode

NameNode runs the file system operations like opening and closing the files. NameNode splits the file into blocks and store them on metadata and makes the data ready to save on DataNode. NameNode has responsiibility for replication of data blocks. HDFS also has a secondary NameNode. The responsibility of this second node is to take the snapshot of NameNodes memory structures. If any failure happen to NameNode, the recent snapshot can be used for recovery.

5.1.4 DataNode

DataNode responsibility is to store data in Hadoop File System. There is more than one DataNode and data stored across several DataNodes. If a user needs any data, first NameNode has to find the location of data blocks, then it can talk directly to the DataNode. Then the DataNode get the instruction from NameNode.

5.1.5 Hadoop MapReduce

The Hadoop splits the data into data nodes and uses the process of mapper, shuffle and reducers to process data in parallel. The following is the execution of the word count problem on Apache Hadoop.

```
$ scp -i ~/.ssh/bozorgi.private mapper.py ec2-user@master:/mnt/opt/hadoop
```

The result of this command is:

```
mapper.py                                     100%  603       0.6KB/s   00:00
```

and we do the same for the reducer.

```
$ scp -i ~/.ssh/bozorgi.private reducer.py ec2-user@master:/mnt/opt/hadoop
reducer.py                                    100%  978       1.0KB/s   00:00
```

To execute this on Hadoop, we need to first login to our master and slaves. With the command:

```
$ ssh -i ~/.ssh/bozorgi.private ec2-user@master
```

By listing the Hadoop directory, we can see our mapper.py and reducer.py that we just copied. We copied a text file (sunny.txt) to Hadoop for testing the MapReduce job on Hadoop with Wordcount example. Our example contains three text files:
today is Monday
today is sunny
today is a sunny Monday
We use the same mapper.py and reducer.py that we explained before on Hadoop. For running mapper and reducer to get a frequency of each word, we need to run the command:

```
$ bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar -file
/mnt/opt/hadoop/mapper.py -mapper /mnt/opt/hadoop/mapper.py -file
/mnt/opt/hadoop/reducer.py -reducer /mnt/opt/hadoop/reducer.py
 -input /dataset/sunny.txt -output /ReducerOutput
```

The details of our MapReduce job is shown below:

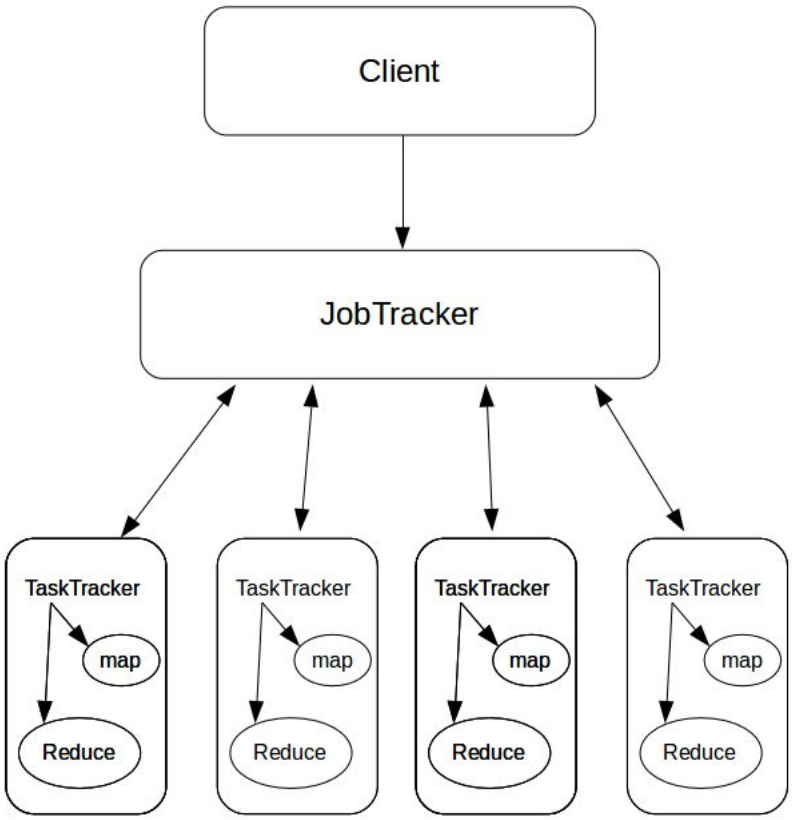

Figure 5.4: Data Node

```
14/11/09 21:08:30 INFO mapreduce.Job:  map 0% reduce 0%
14/11/09 21:08:39 INFO mapreduce.Job:  map 100% reduce 0%
```

```
14/11/09 21:08:46 INFO mapreduce.Job:  map 100% reduce 100%
14/11/09 21:08:47 INFO mapreduce.Job: Job job_1414437132736_0024
completed successfully
14/11/09 21:08:47 INFO mapreduce.Job: Counters: 43
        File System Counters
                FILE: Number of bytes read=105
                FILE: Number of bytes written=248430
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=264
                HDFS: Number of bytes written=34
                HDFS: Number of read operations=9
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=1
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=12698
                Total time spent by all reduces in occupied slots (ms)=4550
        Map-Reduce Framework
                Map input records=4
                Map output records=11
                Map output bytes=77
                Map output materialized bytes=111
                Input split bytes=178
                Combine input records=0
                Combine output records=0
                Reduce input groups=5
                Reduce shuffle bytes=111
                Reduce input records=11
                Reduce output records=5
                Spilled Records=22
                Shuffled Maps =2
```

```
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=161
                CPU time spent (ms)=2090
                Physical memory (bytes) snapshot=635981824
                Virtual memory (bytes) snapshot=2612731904
                Total committed heap usage (bytes)=515375104
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=86
        File Output Format Counters
                Bytes Written=34
```

To see the result of example, we need to open space the output file.

```
$ bin/hdfs dfs -cat /ReducerOutput/*
```

The output is shown below :

```
    Monday 2
a 1
is 3
sunny 2
today 3
```

5.1.6 Hadoop Streaming

With the Hadoop streaming power, programmers can perform MapReduce jobs in several languages such as Ruby, Python, Java, C. The developer has to make sure that they take input from STDIN and output is going to STDOUT. In the next figure, we can see the STDIN and STDOUT:

We can read the input from STDIN line by line and reducer can process the collected lines from STDOUT. In the following script, we can see the code:

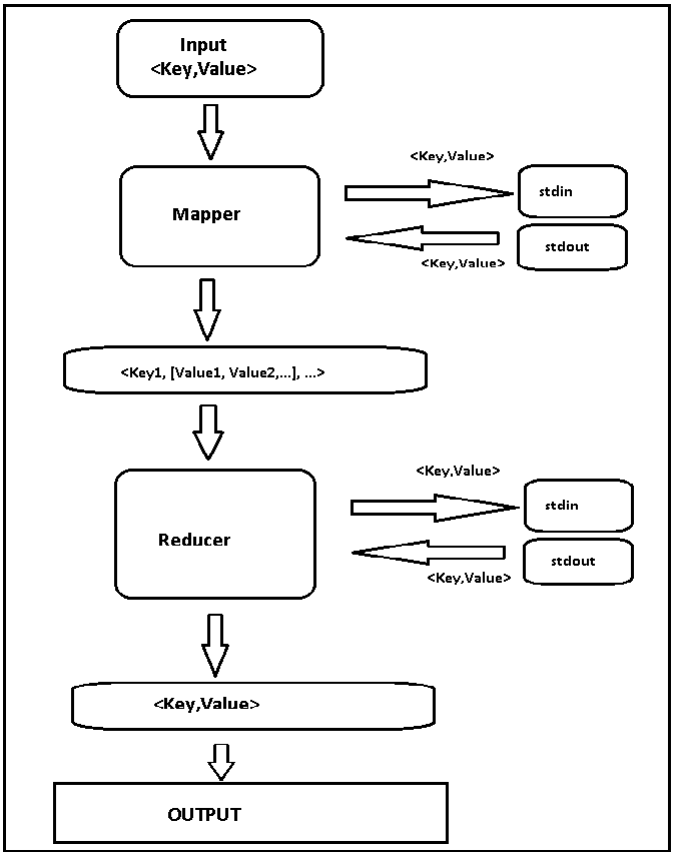

Figure 5.5: Hadoop Streaming

”

```
#mapper
#!/usr/bin/env python

import sys

# input is been writen on STDIN

for line in sys.stdin:
    line = line.strip()  # remove leading and trailing whitespace
      words = line.split()    # split the line into words

    for word in words:      # increase counters
        # The result of mapper will write on STDOUT
        # STDOUT will be input for reducer
```

```
        # the trivial word count is 1
            print '%s\t%s' % (word, 1)
```

Our reducer code based on STDIN, STDOUT:

```
#reducer
#!/usr/bin/env python

from operator import itemgetter
import sys



current_word = None
current_count = 0
word = None

# iread the input from STDIN

for line in sys.stdin:

    # remove  whitespace

    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert string count  to integer count
    try:
        count = int(count)
    except ValueError:
        # count was not a number, quit
        continue

    # shuffle sort the mapper output before sending it to reducer
    if current_word == word:
        current_count += count
```

```
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word


if current_word == word:
    print '%s\t%s' % (current_word, current_count)
"[1]
```

Then with streaming power developer doesnt need to learn a new language and it is reducing the overhead learning for the developer. For running our MapReduce job, after copied our files from our desktop to Hadoop we need to run this command and the result show as below:

```
$  bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar
-file /mnt/opt/hadoop/mapS.py -mapper /mnt/opt/hadoop/mapS.py
-file /mnt/opt/hadoop/RedS.py -reducer /mnt/opt/hadoop/RedS.py
-input /dataset/sunny.txt -output /Stream
```

after running the command, we will receive details about our MapReduce job:

```
14/11/10 06:19:00 INFO mapreduce.Job:  map 0% reduce 0%
14/11/10 06:19:08 INFO mapreduce.Job:  map 50% reduce 0%
14/11/10 06:19:09 INFO mapreduce.Job:  map 100% reduce 0%
14/11/10 06:19:15 INFO mapreduce.Job:  map 100% reduce 100%
14/11/10 06:19:16 INFO mapreduce.Job: Job job_1414437132736_0026
 completed successfully
14/11/10 06:19:17 INFO mapreduce.Job: Counters: 43
        File System Counters
                FILE: Number of bytes read=105
                FILE: Number of bytes written=248346
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=264
```

[1] http://www.quuxlabs.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

```
        HDFS: Number of bytes written=34
        HDFS: Number of read operations=9
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=2
        Launched reduce tasks=1
        Rack-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=13399
        Total time spent by all reduces in occupied slots (ms)=4389
    Map-Reduce Framework
        Map input records=4
        Map output records=11
        Map output bytes=77
        Map output materialized bytes=111
        Input split bytes=178
        Combine input records=0
        Combine output records=0
        Reduce input groups=5
        Reduce shuffle bytes=111
        Reduce input records=11
        Reduce output records=5
        Spilled Records=22
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=153
        CPU time spent (ms)=2040
        Physical memory (bytes) snapshot=649879552
        Virtual memory (bytes) snapshot=2630070272
        Total committed heap usage (bytes)=495976448
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
```

```
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=86
        File Output Format Counters
                Bytes Written=34
14/11/10 06:19:17 INFO streaming.StreamJob: Output directory: /Stream
```

In this example, the result of this command will save on /Stream. For showing the result we just use *cat* command:

```
$ bin/hdfs dfs -cat /Stream/*
```

and the result shows as below:

Monday 2
a 1
is 3
sunny 2
today 3

5.1.7 Job-Tracker

The Job-Tracker job is scheduling of MapReduce jobs, and also monitors the failure or success of those jobs. If Job-Tracker fails, all running jobs will stop.//

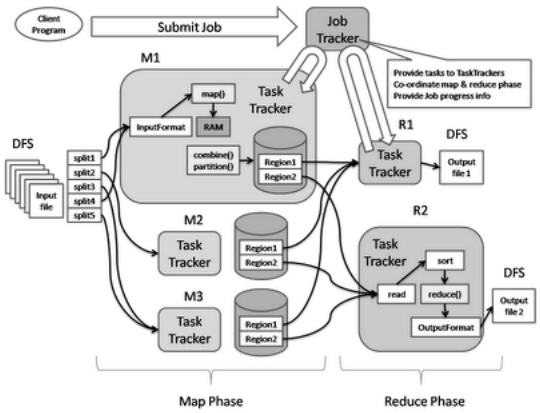

Figure 5.6: Job Tracker[2]

5.1.8 Task-Tracker

Task-Tracker is a node that accepts task like map, reduce, and shuffle from Job-Tracker. Task-Tracker contains several slots that shows the number of tasks that can accept. The Task-Tracker let the Job-Tracker know about the status of the task. If it is failed or if the task done successfully. Another Task-Tracker responsibly is to send a message every few minutes to Job-Tracker to make sure Job-Tracker is still working and also notifies of the number of available slots to Job-Tracker.

5.1.9 Hadoop configuration

Hadoop has been configured based on the users request. After installation of Hadoop, configuration files are generated automatically. The configuration files contain two categories, read-only default configuration that includes core-default.xml, hdfs-default.xml, and mapred.default.xml ,and site-specific configuration includes core-site.xml, hdfs-site.xml, and mapred-site.xml.

[2] `http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/3b_xz`

Chapter 6

Hive and Pig

6.1 Hive structure

Programming with low-level MapReduce requires an extensive knowledge of the system know-how. In addition, it is not easy to maintain and update the specific codes. In this thesis, we used Hive data management software that is built on top of Hadoop. The query language HiveQL provides the same functionality as SQL. HiveQL constructs are built with MapReduce tools that work efficiently with the distributed computing environment of Hadoop. Developers can plug in MapReduce programs directly to the HiveQL. Many companies like Facebook and Yahoo use HiveQL to manage the data analysis associated with the weblogs. Facebook uses Hive to process around 700TB of data daily[11].

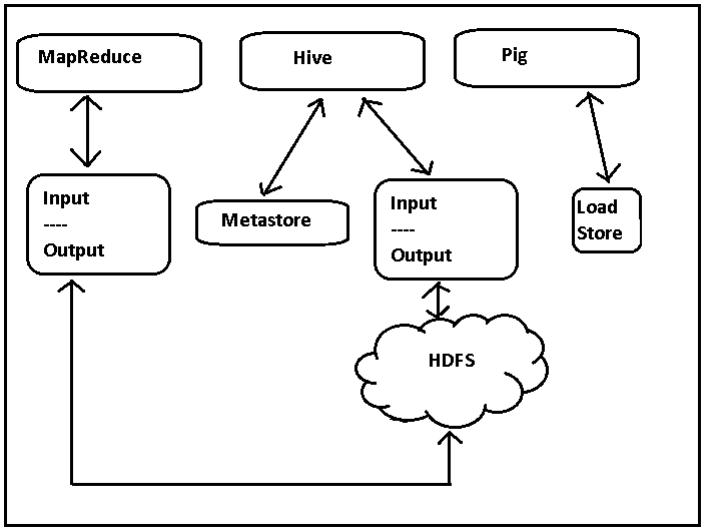

Figure 6.1: Hive Structure

In a relational database model, we need to have well-structured data. For less structured data such as text, audio and maps, the relational model is not a good fit.. Hadoop is designed to accept unstructured, semi-structured, and poly-structured data. The data can be text, video, XML or pdf, etc.

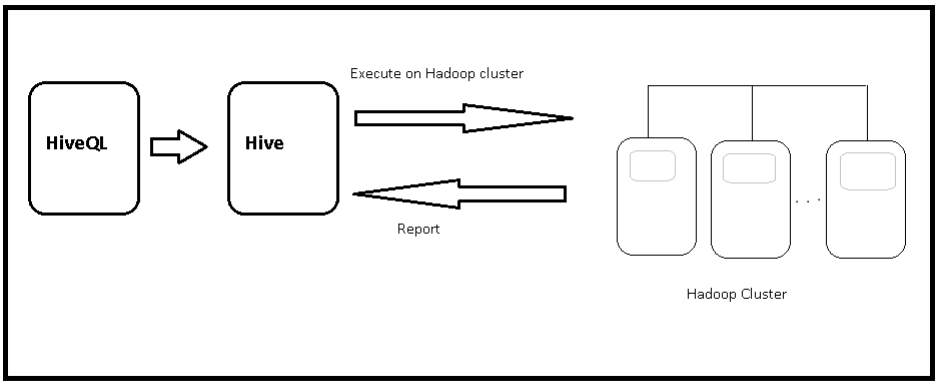

Figure 6.2: Hive Map

The Hive layer is on top of Hadoop; which can create and query large unstructured data with the power of MapReduce.

Hive structure shares many similarities with SQL, such as commands and logic. In Hive, we have concepts like database, tables, rows, columns. Hive supports many data types similar to the data types that are offered by SQL (e.g. double, integer, float). Furthermore, Hive supports complex data types such as maps.

Hive allows developers to extend the data types with their own types. Similar to the traditional relational model, Hive stores the data in tables. Hive tables have rows and columns. Each database is the collection of one or more tables. Each column contains one particular type. The type can be standard primitive types such as integer, complex types like map, or extended type developed by users. The complex type of the map can be defined as a new type as seen in Table 6.1.

Table 6.1: Complex types supported by Hive

Type	*Description*
$map:$	$map < key - type, value - type >$
$list:$	$list < element - type >$
$srtuct:$	$struct < file - name : file - type, ... >$

We can merge the data types together and create our complex data. For example:

```
{list <map < struct < string , struct<int , struct<c1:int , c2:int>>>>}
```

This shows the list of arrays that map the string to the struct. We can put all these commands together and make schemas for our database on Hive with the following command:

```
hive> CREATE TABLE table1(column1 int, column2 string, column3 float,
 column4  list <map < struct < string,
  struct<int, struct<c1:int, c2:int>>>>);
```

The above example has a table with four columns with integer, string, float and list types. We can access the fields of the table with . symbol, and values with '[ ]'. First element of the list shows as Table1.column3 [0]. We can access to c2 Field by Table1.column3 [0]['word'].c2.

As you see, the Hive can support arbitrary complex data types. The tables in Hive using default serialized and deserialized. In Hive, we can use the externally stored data without actually transferring the data to the Hive. This saves a significant amount of time by omitting transferring large amounts of data[11]. The jar in Hive implements the SerDe Java interface (SerDe.jar). The jar file and table connect to each other, and each query assumes that each jar file is a part of the table.

6.1.1 HiveQL structure

HiveQL implements a subset of SQL in addition to new constructs. As with SQL, it supports joins (e.g., inner join, left outer join, right outer join), group by, union, select, etc. SQL knowledge often helps users become proficient in Hive more quickly than those who are not familiar with SQL.

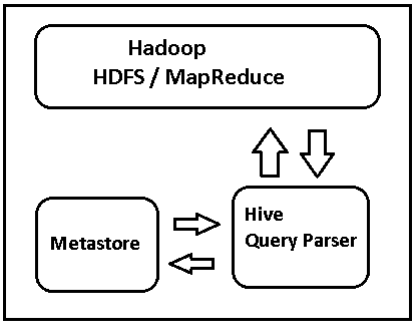

Figure 6.3: Hive Query Structure

The metadata store in metastore browsing features commands such as SHOW TABLES and DESCRIBE TABLE. Command line has access to the metastore to process the queries.

The following is an example of a join command:

```
hive> SELECT table1.a1 as c1, table2.b1 as c2
```

```
    > FROM table1, table2          # means: table1 join table2
    > WHERE table1.a2=table2.b2;
```

We have the ability to insert new data in our Hive table by using the overwrite command below:

```
hive> INSERT OVERWRITE TABLE table1
    > SELECT *
    > FROM table2;
```

We have several different joins in Hive such as join, inner join, and Map join. Below are examples of join, left outer join, right outer join, and full outer join.

- Join: the rows will be joined where they have the same key(the keys must be matched). In this join result, the rows that do not match the keys are eliminated and thus we cannot see them in the results.

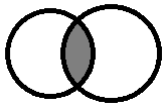

Figure 6.4: Join

- Left Outer Join: This join will be the rows from the first (left) of the two tables below; they do not look at the match keys.

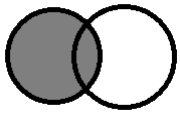

Figure 6.5: Left Outer Join

- Right Outer Join: In this join, the rows in the second (right) table will be shown and do not check the key.

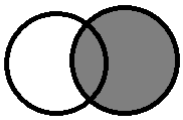

Figure 6.6: Right Outer Join

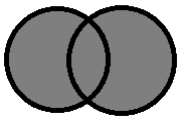

Figure 6.7: Full Outer Join

- Full Outer Join: Both tables will be shown and if unmatched rows are found, the results will return a null value in those appropriate fields.

Our Hive query is based on MapReduce, which means that our common join will have the map and reduce stages. This process is described visually in figure below. All mapper input runs using the join command. Mapper first read the fields and then sends the join (key, value) pairs to a file. The shuffle stage sorts and merges all of the (key, value) pairs and sends them to the reducer.

Reducer receives these sorted and merged pairs and runs the join process. The merge process within the shuffle stage is very time consuming and puts a massive load on the CPU. We can see the join process in the figure below .

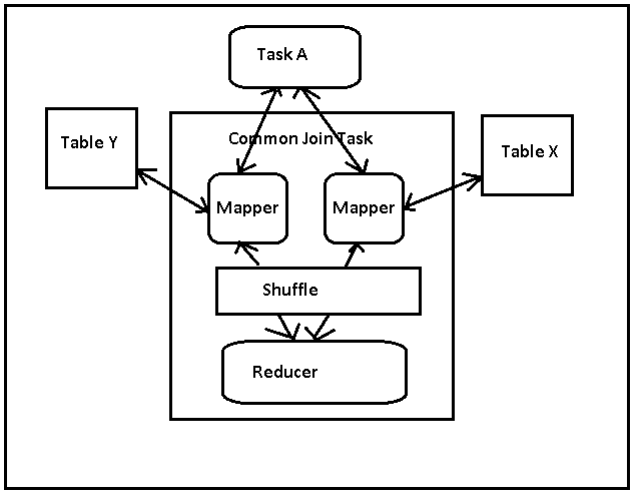

Figure 6.8: Map Join Pricess

In Map Join, the job performance is improved by removing the work for the shuffle and reduce stages, which it sends all of the work to the mapper stage. In Map Join, the mapper stage performs the join command by keeping the small joint table data stored in memory. The job is also performed in memory.

However, there are two problems associated with Map join. The first problem is that the data in the join table must be small enough to be stored in the memory; if the table is too large, then the process will not be able to run. The second problem is that the number of mappers that the

mapper wants to read from memory at the same time is limited. This means that if hundreds of mappers want to read the same memory-stored table simultaneously, the processes will crash.

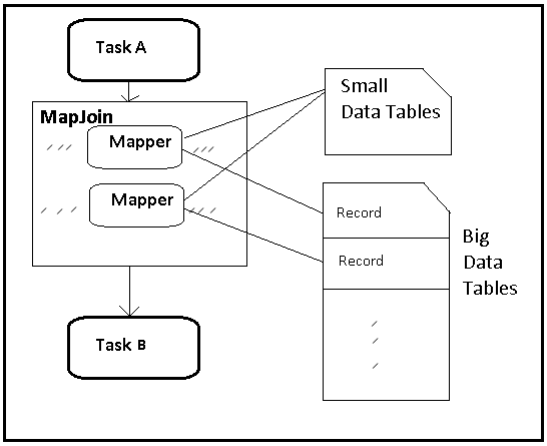

Figure 6.9: Optimized Map Join

To solve these problems, we can use a method of distributed cache called Optimized Map Join. In this model, each new task reads the small table from the Distributed File System to a hash table. After completing the read process, we have our hash table file. All the mappers can upload the hash table file to the Hadoop Distribute Cache and perform the join job. Even if hundreds of mappers run on the same system, the system will continue to operate without any problems because the Distributed Cache merely needs to send a copy of hash table file to the system. [11]

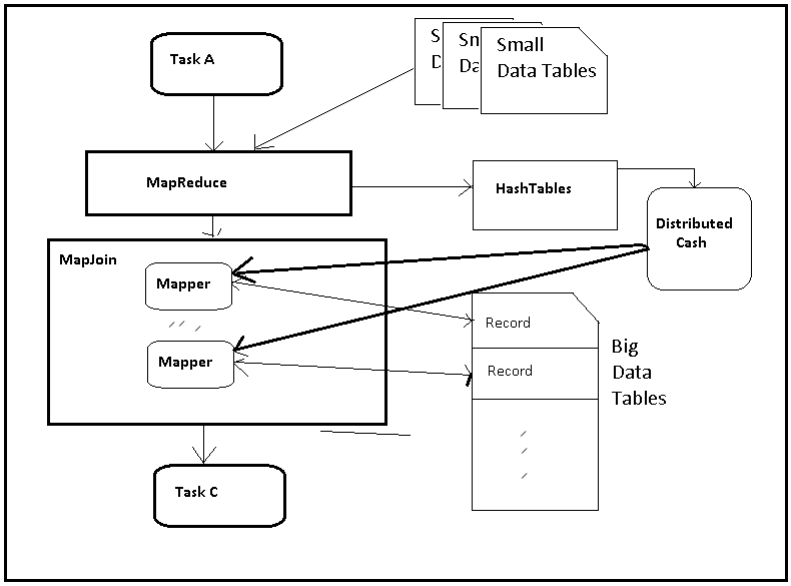

Figure 6.10: Using Hash Tables in Optimized Map Join

In Hive, we do not have INSERT INTO, DELETE and UPDATE commands.

Hive has the ability to support MapReduce programs, and the MapReduce code can be programmed in any language that accepted by Hadoop.For example, we can get a query from our word count example that we wrote the code based on Python.

```
hive> FROM (MAP text USING 'python mapper.py' AS (key, value)
    > FROM docs CLUSTER BY key) a
    > REDUCE key, value  USING 'python reducer.py';
```

In the above example, we are presented with the Map and Reduce functions. Depending on the job, it is possible that the only mapper will run and reducer is not required. Mapper is always mandatory, but reducer can be optional. Cluster BY specifies output columns from mapper.py and these columns will be the reducer.py input. The reducer.py output will be the final result. In Hive, besides CLUSTER BY, we have DISTRIBUTE BY and SORT BY. In Hive, we can insert data into a table with an INSERT OVERWRITE clause or INSERT OVERWRITE DIRECTORY (hdfs path), or INSERT OVERWRITE LOCAL DIRECTORY (on the local user's system)[11].

Hive Join Examples

Below are some examples of Join in Hive. The first is a simple example of a join command that is run on two tables on Hive. The next example consists of three tables and executes different join commands on each table. The third example also uses a join command on three tables, but then uses an STREAMTABLE command. We first create three tables in Hive by:

```
> create table one (c1 string, c2 string);


> create table two (b1 string, b2 string);



> create table three (a1 string, a2 string , a3 string);
```

- `> select one.* from one join two on (one.c1=two.b1);`
- `> create table three (a1 string, a2 string , a3 string);`
- `> SELECT /*+ STREAMTABLE(one) */ one.c1 , two.b1 , three.a1 FROM one JOIN two ON (one.c2=two.b2) JOIN three ON (three.a2=two.b2);`

- > select /*+ MAPJOIN(two) */ one1.c1 , one1.c2
  FROM one1 JOIN two ON one1.c1=two.b1;

  In our following OUTER LEFT JOIN command example, the result returns all the rows in table two. We assume two tables that are named *Table two* and *Table three*. Each table had a list of associated entities as shown in Table6.2 and 6.3.

Table 6.2: Table Two

*Column*1	*Column*2
sunny	*sunday*
rainy	*night*
foggy	*Thursday*

Table 6.3: Table Three

*Column*1	*Column*2
sunny	*night*
foggy	*day*
snowy	*night*

We will continue to use Table two and Table three from Table 6.3 and Table 6.4 This time, we will run the LEFT OUTER JOIN command on the first column of each of the two tables. The command took 22.656 seconds to run. The results of the command are presented in Table 6.5.

- > select three.a1 , two.b1 FROM three LEFT OUTER JOIN two ON (three.a2=two.b2);

```
Total MapReduce jobs = 1

sunny night     NULL
foggy day       NULL
snowy night     NULL
Time taken: 22.656 seconds, Fetched: 3 row(s)
```

The result of this join is shown in Table 6.4.

6.1.2 Data Storage in Hive

Tables in Hive can store in HDFS, sub-directory of the table, or in tablets directory.

- With PARTITION BY clause, we can store our table on an HDFS directory.

Table 6.4: The result of Left Outer Join

*Column*1	*Column*2
sunny	*night*
foggy	*day*
snowy	*night*

```
hive> CREATE TABLE test(c1 string, c2 int)
    > PARTITIONED BY (Date string, Hour int);
```

In the above example, the table will be stored in hdfs, and partition exists for every value of Date and Hour. The partition columns are stored in metadata table; they cannot store in table data. For creating new partition, we can use INSERT or ALTER clues.

- A partition stores the portions of tables on subdirectory as hive directory.

```
hive> INSERT OVERWRITE TABLE
    > test PARTITION
    > (Date='2014-03-05', Hour= 10)
    > SELECT * FROM table2;
```

```
hive> ALTER TABLE test
    > ADD PARTITION (Date='2014-03-05',Hour=10);
```

In the above example, we add the new partition to the test table and store the columns in the test table HDFS directory.

```
hive> SELECT * FROM test
    > WHERE ds='2020-05-05';
```

The compiler just scans the specified HDFS directory (e.g. Path like /hive/examples/ds=2020-05-05). Then we just can access to the part of the data that we need.

- Hive uses the Bucket concepts for storing the units. A bucket is a file with the directory of the table or a partition. The time that the user is creating the tables can specify the Buckets. For example we have a table that bucketed into 20 buckets.

```
hive> SELECT * FROM test TABLESAMPLE (3 OUT OF 20);
```

That in this example compiler will scan the data in the third bucket. And look at the data buckets store in HDFS directory.

- Hive has the ability to get a query from the data that are stored locally in hive directory, also can get a query from tables that stored in other locations in HDFS.

  The only different between the original table and external table is on DROP TABLE Claus that when we do the DROP TABLE, in external table we just delete the data from metadata and the data still stayed in external table. In normal tables DROP TABLE command delete the entire table.

  It depends if the data is partitioned or not, a bucket stored the data in a file on where the table is already stored.

```
hive> CREATE EXTERNAL TABLE test1(column1 int, column2 string, column3 string)
    > LOCATION '/data/dataset'; #hdfs path
```

6.1.3 SerDe

Hive has an ability to use the SerDe. With Serializer and Deserializer power, Hive can recognize to how the process a record.

- Deserializer read a record, take a string of the record and convert it into Java objects. Hive can handle these Java objects (e.g. when we execute the SELECT command)
- Serializer takes these Java objects and translates it to something that let the Hive write it on the HDFS. (e.g. For instance when we run the INSERT command).

Programmer developed the SerDe jar file and connected it to the table. In Hive, we have default SerDe that called LazySerDe, With the LazySerDe we assume the data stored in a file. It recognized the row delimiter and column delimiter.

```
hive> CREATE TABLE test2(c1 int, c2 string)
    > ROW FORMAT DELIMITED
    > FIELDS DELIMITED by '\003'
    > LINES TERMINATED BY '\012';
```

Also, we can delimit the serialized key, and also we can delimit the value of maps with DELIMITED clause.

```
hive> CREATE TABLE test3(c1 int, c2 list<map<string,int>>)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY'\t'
    > COLLECTION ITEMS TERMINATED BY '\n'
    > MAP KEY TERMINATED BY '\002';
```

The one of exciting features of SerDe is RegexSerDe that allows the programmer to specify the regular expression before compiler parse those columns. To assist with explaining of SerDe work, we reference the figure below.

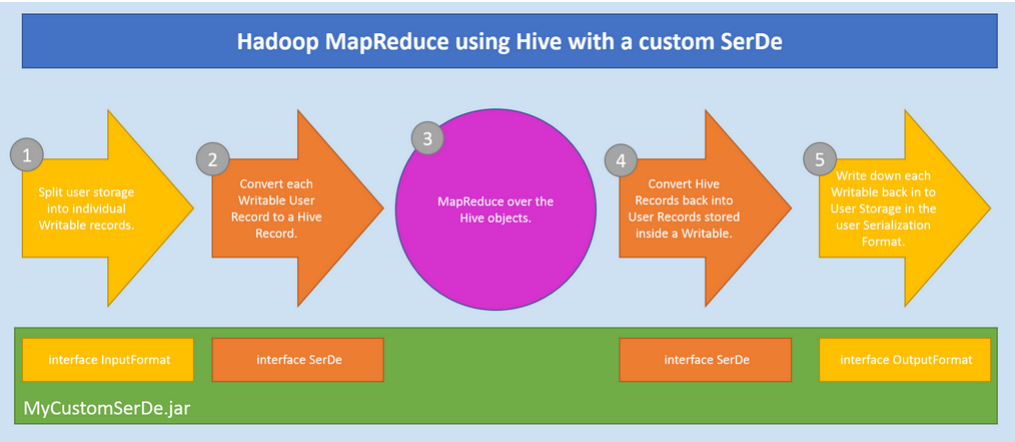

Figure 6.11: Hadoop MapReduce Using Hive with a Custom SerDe[1]

6.1.4 Storing Formats

Hive supports most of the formats by default. For example Text files stored in TextInputFormat. Moreover, the programmer can specify their file format. Developers needs to specify their format by using INPUTFORMAT and OUTPUTFORMAT clause and store it by STORED AS clause.

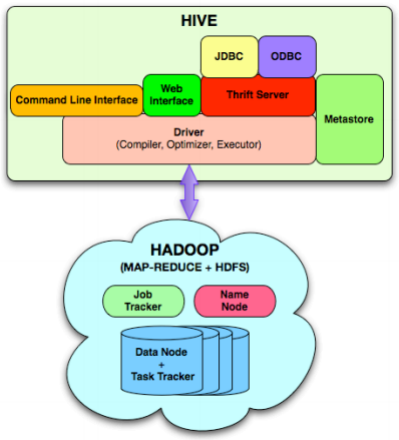

Figure 6.12: System Architecture[2]

[1] `http://ruibm.com/2013/06/02/hadoophive-writing-a-custom-serde-part-1/`
[2] urlhttp://www.programering.com/a/MTM0EzMwATM.html

To help the illustrate the Hive system architecture, we will reference the figure of a system architecture in Hive. We have several main components in Hive. Metadata is one of an important element in Hive, which store the tables, columns, and partitions. Matastore store location of the table, the table schema, the columns and their types and their partitions.

Compiler in Hive, process the HiveQL clauses. The same as other compilers it starts with parsing. In Hive for generating the tree from the command, Antlr will be used. We have type checker as well. In type checker stage, the compiler got the input and output tables information, from the Metastorea and used create a logical plan for that information. After finishing compiling the task is time for executing the task.

6.1.5 Executing Tasks

Each task execute after all the task pre-request job be done. For example, for MapReduce task the pre-request is to first serialized the portion of the task to a file. Then added this file to a cache. At the end, task deserialized the file and then execute the task. The result of the execution saved in a temporary location.

6.2 Pig

Programmers for using Hadoop have to spend lots of time for coding for mapper and reducer. Yahoo! Developed Pig around 2007 that focuses on spending less time on coding. It can handle and use almost all kind of data. In Pig, we have two main concepts: Pig Latin that is programming language, and Pig Engine the execution part that run Pig Latin scripts on multiple MapReduce jobs in Hadoop clusters. So, the result of this part can show on screen or store in a file. Moreover, Pig is a flexible UDF (User Define Language) that allows the user to use many programming languages like Java, Jython, Groony, Python, and JRuby. As we said before, many companies are exciting to work with Hadoop, but they found some weakness during work with Hadoop.
The first limitation of Hadoop is that the developer needs to put a lot of efforts to code with low-level programming. The solution for solving this problem is using Pig that is a high-level data-flow programming language. Pig interactive shell is Grunt. With Grunt shell, users can use Pig Latin and also Grunt has relation to HDFS. Compare to Hadoop, with Pig we have more flexibility.
Many companies such as Google and Facebook use Pig/Hadoop for processing on their large amount of their data[10]. Pig is compatible with unstructured, semi-structured and structured data. Another Hadoop limitation is that it cannot support data flow. Hadoop MapReduce unable to merge several process of several data sets together.

[3] `http://www.itcandor.com/hadoop/`

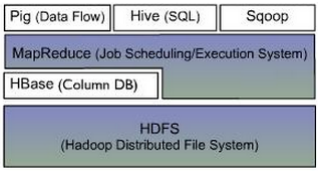

Figure 6.13: Hadoop Ecosystem[3]

Pig language called Pig Latin; a programmer can use the Pig Latin and produce a pig code, and then this program compiles it in MapReduce job, it can be one or more MapReduce jobs. Then those jobs are executed on Hadoop. Pig supports fault tolerance and scalability. Pig is easy to use and learn with who is familiar with SQL. Pig can work with typical data operation like Join; also can work with nested data service like Map.

6.2.1 Difference between Pig and Hive

There are several differences between Hive (a distributed data warehouse) and Pig(a dataflow programming language).

- The programmer does not need to wait for all the data loaded then start coding. During the data loading developer can coding. The hive that is kind of RDBMS based first has to wait for all the data to load. In some examples, like the data from satellites that is a stream data working with Pig is a lot easier than Hive. In Hive we need buckets to keep the data, and the programmer has to work on the one bucket and run it again on the other buckets while still new data are importing to the new buckets.
- Pig Latin is designed as a data flow language. Pig Latin is more flexible and is more accepted by software engineers. Hive is more related to data analysis.

Chapter 7

Experiments

Building an inverted index for web pages is a hard and time-consuming task. Google designed an innovative method to index and searched a big corpus using MapReduce methodology. This MapReduce technic was used to build the Google Web1T-5gram. Later Google developed the Web1T-easy package that contains two Perl scripts: indexing, access to the database. In indexing section, words normalized to lowercase. Each word given an unique identification number. Then the result was converted to a database table in SQLLite, MYSQL and PostgreSQL.

In our work, we used the Hive for counting the frequency of each word. Hive is a warehouse layer on top of Hadoop. Hive is using the MapReduce power, but the programmer doesnt need to write the mapper and reducer code.

We demonstrate the use of Hive in word counting application. The first example contains three lines, each line representing a text file. We first create a table with one column to store each textfile as a row. Then we load the data in the table and then run the query to build the inverted index in Hive.

```
> CREATE TABLE hivedocs (line STRING);
OK
Time taken: 0.424 seconds

> LOAD DATA INPATH '/data/sunnysmall.txt' OVERWRITE INTO TABLE hivedocs;

Loading data to table default.hivedocs
Table default.hivedocs stats: [num_partitions: 0, num_files: 1,
 num_rows: 0, total_size: 56, raw_data_size: 0]
OK
Time taken: 0.608 seconds
```

```
> select * from hivedocs;
OK
today is Monday
today is sunny
today is a sunny Monday

Time taken: 0.208 seconds, Fetched: 4 row(s)

> CREATE TABLE WordFrequency AS SELECT word, count(1) AS count FROM
    > (SELECT explode(split(line, '\t')) AS word FROM hivedocs ) w
    > GROUP BY word  ORDER BY word;

Total MapReduce jobs = 2

Launching Job 1 out of 2
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_1414437132736_0062, Tracking
URL = http://master:8088/proxy/application_1414437132736_0062/
Kill Command = /mnt/opt/hadoop/bin/hadoop job  -kill job_1414437132736_0062
Hadoop job information for Stage-1: number of mappers: 1;
number of reducers: 1
2014-11-16 10:36:56,782 Stage-1 map = 0%,  reduce = 0%
2014-11-16 10:37:04,012 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.75 sec
2014-11-16 10:37:05,043 Stage-1 map = 100%,  reduce = 0%,
Cumulative CPU 1.75 sec
2014-11-16 10:37:06,077 Stage-1 map = 100%,  reduce = 0%,
Cumulative CPU 1.75 sec
```

```
2014-11-16 10:37:07,109 Stage-1 map = 100%,  reduce = 0%,
Cumulative CPU 1.75 sec
2014-11-16 10:37:08,140 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.75 sec
2014-11-16 10:37:09,172 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.75 sec
2014-11-16 10:37:10,205 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.75 sec
2014-11-16 10:37:11,242 Stage-1 map = 100%,  reduce = 100%,
Cumulative CPU 3.28 sec
2014-11-16 10:37:12,276 Stage-1 map = 100%,  reduce = 100%,
Cumulative CPU 3.28 sec
2014-11-16 10:37:13,310 Stage-1 map = 100%,  reduce = 100%,
 Cumulative CPU 3.28 sec
MapReduce Total cumulative CPU time: 3 seconds 280 msec
Ended Job = job_1414437132736_0062
Launching Job 2 out of 2
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_1414437132736_0063, Tracking
URL = http://master:8088/proxy/application_1414437132736_0063/
Kill Command = /mnt/opt/hadoop/bin/hadoop job  -kill job_1414437132736_0063
Hadoop job information for Stage-2: number of mappers: 1; number of reducers: 1
2014-11-16 10:37:21,890 Stage-2 map = 0%,  reduce = 0%
2014-11-16 10:37:29,130 Stage-2 map = 100%,  reduce = 0%,
 Cumulative CPU 0.78 sec
2014-11-16 10:37:30,163 Stage-2 map = 100%,  reduce = 0%,
Cumulative CPU 0.78 sec
2014-11-16 10:37:31,195 Stage-2 map = 100%,  reduce = 0%,
 Cumulative CPU 0.78 sec
```

```
2014-11-16 10:37:32,228 Stage-2 map = 100%,  reduce = 0%,
Cumulative CPU 0.78 sec
2014-11-16 10:37:33,262 Stage-2 map = 100%,  reduce = 0%,
 Cumulative CPU 0.78 sec
2014-11-16 10:37:34,293 Stage-2 map = 100%,  reduce = 0%,
Cumulative CPU 0.78 sec
2014-11-16 10:37:35,330 Stage-2 map = 100%,  reduce = 0%,
Cumulative CPU 0.78 sec
2014-11-16 10:37:36,364 Stage-2 map = 100%,  reduce = 100%,
Cumulative CPU 2.16 sec
2014-11-16 10:37:37,406 Stage-2 map = 100%,  reduce = 100%,
Cumulative CPU 2.16 sec
MapReduce Total cumulative CPU time: 2 seconds 160 msec
Ended Job = job_1414437132736_0063
Moving data to: hdfs://master:54310/user/hive/warehouse/wordfrequency
Table default.wordfrequency stats: [num_partitions: 0,
 num_files: 1, num_rows: 0, total_size: 37, raw_data_size: 0]
MapReduce Jobs Launched:
Job 0: Map: 1  Reduce: 1   Cumulative CPU: 3.28 sec
 HDFS Read: 273 HDFS Write: 229 SUCCESS
Job 1: Map: 1  Reduce: 1   Cumulative CPU: 2.16 sec
 HDFS Read: 594 HDFS Write: 37 SUCCESS
Total MapReduce CPU Time Spent: 5 seconds 440 msec
OK
Time taken: 49.898 seconds


hive> select * from WordFrequency;
OK

Monday  2
a       1
is      3
sunny   2
today   3
```

```
Time taken: 0.149 seconds, Fetched: 6 row(s)
```

As it can be seen the process took a large amount of the time to index the three lines. We then run it on another text files with the larger sizes:

```
[ec2-user@master hadoop]$ du -sh book1.txt
168K            book1.txt
[ec2-user@master hadoop]$ du -sh book2.txt
2.3M            book2.txt
[ec2-user@master hadoop]$ du -sh book20.txt
23M             book20.txt
[ec2-user@master hadoop]$ du -sh book30.txt
50M             book30.txt
```

In book1, that the words separated by whitespace:

```
hive> CREATE TABLE book1 (info1 STRING);

OK
Time taken: 0.124 seconds

hive> LOAD DATA INPATH '/data/book1.txt' OVERWRITE INTO TABLE book1;

Loading data to table default.book1
Table default.book1 stats: [num_partitions: 0,
num_files: 1, num_rows: 0, total_size: 167515, raw_data_size: 0]

OK
Time taken: 0.476 seconds


hive> CREATE TABLE Book1Frequency AS SELECT word, count(1)
AS count FROM (SELECT explode(split(info1, ' ')) AS word FROM book1 ) w
GROUP BY word  ORDER BY word;
Total MapReduce jobs = 2
Launching Job 1 out of 2
```

```
Total MapReduce CPU Time Spent: 11 seconds 640 msec
OK
Time taken: 53.822 seconds
```

Similarly, we loaded book2 thru book30 into Hive. And the loading time is shown in Table 7.1.

Table 7.1: Loading the data

Name	*Size*	*CreateTableTimeTaken*	*LoadDataTimeTaken*	*SplitTheDataOnTable*
*Book*1	168*K*	0.124	0.476	53.822
*Book*2	2.3*M*	0.109	0.409	54.631
*Book*20	23*M*	0.103	0.502	55.642
*Book*30	50*M*	0.275	0.552	62.773

We now constructed and ran queries based on certain words. The first query looks for the word Alice in the index. If we want to know how many time name Alice repeated in our file we run this command:

```
hive> select * from book20Frequency where word = "Alice";
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1414437132736_0074,
Tracking URL = http://master:8088/proxy/application_1414437132736_0074/
Kill Command = /mnt/opt/hadoop/bin/hadoop job  -kill job_1414437132736_0074
Hadoop job information for Stage-1: number of mappers: 1;
number of reducers: 0
2014-11-16 11:50:19,025 Stage-1 map = 0%,  reduce = 0%
2014-11-16 11:50:27,276 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.94 sec
2014-11-16 11:50:28,317 Stage-1 map = 100%,  reduce = 0%,
 Cumulative CPU 1.94 sec
MapReduce Total cumulative CPU time: 1 seconds 940 msec
Ended Job = job_1414437132736_0074
MapReduce Jobs Launched:
Job 0: Map: 1   Cumulative CPU: 1.94 sec   HDFS Read: 64932
HDFS Write: 11 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 940 msec
OK
```

```
Alice    3094
Time taken: 17.145 seconds, Fetched: 1 row(s)
```

We run the same query on the other books as shown in the table 7.2.

Table 7.2: Loading the data

Name	*Size*	*Word*	*Frequency*	*TimeTaken*
*Book*1	168*K*	*Alice*	221	17.048
*Book*2	2.3*M*	*Alice*	543	17.121
*Book*20	23*M*	*Alice*	3094	17.145
*Book*30	50*M*	*Alice*	68068	17.118

As we can see, it is working in less time foe bigger size of data.

Uploading the Google Web1T-5gram in Hadoop, which took a lot of time. We use a similar table as in our book example to store the all gzip file in Hive.

First we send the files to Hadoop. For sending the files to Hadoop first, we use the below command.

```
$ scp -c blowfish -i ~/.ssh/bozorgi.private *.gz
ec2-user@master:/mnt/opt/hadoop/web1t/5g
```

For sending data to the server, for each gzip file we spend 5:56 up to 54:19 minutes. The reason that it's not fast enough could be Internet connection speed, Internet traffic, or resources busy at the server side. After sending all the gzip file, we run the query in one gzip file, and then run the queries in all the gzip file. Working with Hive is a good choice for working with Big Data.

In the first experience, we decompress one of the files (for example file 4gm-0010.gz) by *gunzip* command.

```
$ gunzip /mnt/opt/hadoop/web1t/5g/4gm-0010.gz
```

Then we send the new decompress file to HDFS layer.

```
$ bin/hdfs dfs -copyFromLocal /mnt/opt/hadoop/web1t/5g/4gm-0010  /data
```

We create a table with one column and send the file into it.

```
> create table 4gm (c1 string);

> LOAD DATA INPATH '/data/4gm-0010' OVERWRITE INTO TABLE 4gm;
```

The data in Web1T separated by tab. We split the data on the table and store it in a new table with two column: one for words and the other one for the frequency of the words.

```
hive> CREATE TABLE 4gmcount
    AS SELECT word, count(1)
    AS count FROM(SELECT explode(split(c1, '\t')) AS word FROM 4gm) w
    GROUP BY count ORDER BY count;

    CREATE TABLE fin
    AS SELECT word, count(1)
    AS count FROM(SELECT explode(split(c1, '\t')) AS word FROM words) w
    GROUP BY word;
```

We run some queries and the result shown in Table 7.3.

Table 7.3: Sample query result from file 4gm-0010

$Sequenceofwords$	$Frequency$	$TimeTaken$
$-ElectronicsandMicroelectronics$	111	$24.55seconds$
$-ElectricityNetworkCompany$	128	$24.55seconds$
$-ElectricityMarketModel$	54	$24.55seconds$
$-ElectricitySupplyIndustry$	941	$24.55seconds$
$-ElectricityandMagnetism$	2184	$24.55seconds$

We run the same process to get query on all data. First we send all the gzip file into the table that has one column; then we spilled the gzip file and store it into another table with two columns and then run some queries. The gzip files contain vocabulary, one to five-gram word frequency.

The following is a 3-gram data contained in this Web1T-5gram corpus:

Table 7.4: Sample query result from file 4gm-0010

$words1$	$words3$	$words3$	$frequency$
A	$MEMBERSHIP$	$ORGANIZATION$	69
A	$MEMBERSHIP$	GET	253
A	$MEMORABLE$	DAY	69
A	MEN	$< /S >$	410
A	$MEMORY$	!	184

There are some differences between the traditional database and Hadoop. A traditional relational database designed to work on structured data. It requires data to be fit into the relational paradigm of columns and rows schema. Based on IBM report 80 percent of the collected data is unstructured. However, on Hadoop with MapReduce help, the programmer does not need to have structured data. MapReduce developers are free to structure the data; they can have their structure or even can work with no structure. MapReduce paradigm is flexible. In the traditional method, the size of tables is limited by the local storage size. Apache Hadoop is not a database.

However, it is a distributed file system that can run large volumes of data in parallel. In the figure below, we describe some of the differences between Traditional Database and Hadoop.

Table 7.5: Different between RDBMS and Hadoop

	$RDBMS$
$DataSize$	$GigabyesorTerabytes$
$Structure$	$JustWorkWithStructuredSata(StaticSchema)$
$Read/WriteThroughoutLimits$	$1000sQueriesPerSecond$
$DataLayout$	$RowFamilyOriented$
$Update$	$Read/WriteManyTimes$
	$Hadoop$
$DataSize$	$PetabytesOrHexabytes$
$Structure$	$WorkWithStructured, Semi-StructuredandUnstructuredData$
$Read/WriteThroughoutLimits$	$MillionsOfQueriesPerSecond$
$DataLayout$	$ColumnFamilyOriented$
$Update$	$WriteOnceAndReadManyTimes$

We used the MapReduce on the Hadoop and Hive jobs. It is important to know that MapReduce is not a database system. MapReduce is an algorithm for the processing on BigData.

Hive is built in on top of Hadoop with an SQL type query language using for querying and complex analysis. After developer define tables and columns, data is loaded into these tables. When we run a query, the MapReduce jobs run automatically. Hive is designed for batch processing, which means is not working on real time queries. MapReduce is an algorithm for the processing on BigData.

Chapter 8

Conclusion

Knowing the frequency of the word is part of the standard methodology in linguistics for exploiting corpus. In 1991, Sinclair mentioned that "anyone studying a text is likely to need to know how often each different word form occurs in it". Google released Web1T-5gram from the source of 1 trillion words that in that database keep the frequency of the sequence of words. Since there is no comparable corpus with this data size, Web1T became one of valuable reference. Google released an open source software called web1T-easy for indexing and querying of that data based on RDBMS. MapReduce programming paradigm is used for parallel computing on massive data. Map function read the input file and divide it into multiple (key, value) pairs. Reduce function aggregates the pairs with the same key and the result of reduce function will be one file. To imagine how is working we can think the map function as a GROUP-BY command in SQL, and reduce function can be the aggregation on SQL-like maximum, average that applied over all the rows.

Hadoop is using MapReduce functions and is a software framework. For using Hadoop we do not need to have expensive hardware, and it can be installed in Linux cluster. Hadoop is reliable by storing the data minimum three times. Also, it is fault-tolerance and scalable. Jobs that written in different languages like Python, C and Java, can create and run on Hadoop. Also, Hadoop has TaskTracker and JobTracker that it can keep track of all execution part and keep track of the nodes job of each cluster. Hive is a data warehouse that support SQL commands. In SQL data must be the matched to the schema if the load data doesn't confirm the schema type it is stop working. But in Hive doesn't verify the type of input data during loading data. Hive use batch processing. It is not working on real-time data. Pig programming language called Pig Latin. It has some similarity to SQL like GROUP BY and DESCRIBE commands. But there are several different between Pig and SQL. For instance, in RDBMS structured data stored in tables but in Pig can run on unstructured data. Pig Latin focuses on data processing on complex data. For programming with Pig, the developer doesn't need to know Java, and the same as Hadoop developer can use

several different programming languages.

Our future work will be work on computer Poker data. Video game developers gain and adopt advanced analytics for supporting each game. They need to have all the unstructured data and at the same time they need to have the ability to analyse those data. Each video game has different players, platforms and different type of data. We will design the efficient method based on Hadoop and Hive for analyzing the Poker big data.

Bibliography

[1] Ziad Benslimane. Optimizing hadoop parameters based on the application resource consumption. 2013.

[2] Wei Dai and Mostafa Bassiouni. An improved task assignment scheme for hadoop running in the clouds. *Journal of Cloud Computing*, 2(1):1–16, 2013.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Francis X Diebold. big datadynamic factor models for macroeconomic measurement and forecasting. In *Advances in Economics and Econometrics: Theory and Applications, Eighth World Congress of the Econometric Society,(edited by M. Dewatripont, LP Hansen and S. Turnovsky)*, pages 115–122, 2000.

[5] Stefan Evert. Google web 1t 5-grams made easy (but not for the computer). In *Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop*, pages 32–40. Association for Computational Linguistics, 2010.

[6] Wei Fan and Albert Bifet. Mining big data: current status, and forecast to the future. *ACM SIGKDD Explorations Newsletter*, 14(2):1–5, 2013.

[7] Jared Gray and Thomas C Bressoud. Towards a mapreduce application performance model. In *Midstates Conference*, 2012.

[8] D Laney. 3-d data management: Controlling data volume, velocity and variety, meta group, research note, february 2001.

[9] Attila Marton, Michel Avital, and Tina Blegind Jensen. Reframing open big data. 2013.

[10] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki BN Rao, Vijayanand Sankarasubramanian, Siddharth Seth, et al. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1081–1090. ACM, 2011.

[11] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[12] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[13] Xiaoyang Yu. Estimating language models using hadoop and hbase. *University of Edinburgh*, 2008.

Vita

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Mandana Bozorgi

Degrees:

Bachelor of Engineering in Computer Software Engineering 2000

University of Islamic Azad Tehran

Thesis Title: DATA ANALYSIS WITH MAP REDUCE PROGRAMMING PARADIGM

Thesis Examination Committee:

Chairperson, Dr. Kazem Taghva, Ph.D.

Committee Member, Dr. Ajoy Datta, Ph.D.

Committee Member, Dr. Laxmi Gewali, Ph.D.

Graduate Faculty Representative, Dr. Ebrahim Saberinia, Ph.D.