The MapReduce Framework

In Partial fulfillment of the requirements for course CMPT 816

***Abstract*—MapReduce is a simple programming approach that applies to various large-scale computing problems. This model usually used for processing and generating large data sets. Programmers specify a map function that processes a key-value pair to generate a set of intermediate key-value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. This simple idea has been shown to be an effective method of programming complex parallel problems. Many real world computing tasks are expressible in this model such as distributed pattern-based searching, distributed sorting, document clustering, machine learning, and web access log statistics. In this paper, we present a brief survey about MapReduce framework including: definition, examples, architecture, execution overview, and how MapReduce Framework is used in Hadoop.**

***Index Terms*—MapReduce, Framework.**

I. Introduction

MapReduce has been gaining popularity and it has been used at Google extensively to process 20 petabytes of data per day [1]. Also, MapReduce has used in Facebook for production jobs including data import, hourly reports, etc [2]. Amazon proposed Amazon Elastic MapReduce [3], based on MapReduce, to help users perform data-intensive tasks in the cloud.

MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. MapReduce was firstly introduced in [1] by Google. Although the map-reduce technique is well known in the worlds of distributed and parallel computing, Googles MapReduce paper [1] has brought new research interests to this programming paradigm. The paper attracted considerable interest and enabled MapReduce pattern to become a common technique for parallelization. The idea of MapReduce is very simple, the user provides the system with a *map* function (a function taking a tuple and outputting a tuple) and a *reduce* function (a function taking a tuple and outputting a single value).

MapReduce also could be defined as a framework for processing parallelization problems across huge datasets using a large number of computers. This group of computers either referred to as a cluster– if all nodes are on the same local network, and use similar hardware; or a grid– if the nodes are shared across geographically distributed systems, and use heterogenous hardware [4]. Computational processing can occur on data stored either in a filesystem (i.e., unstructured) or in a database (i.e., structured). MapReduce can take advantage of locality of data, processing data on or near the storage assets to decrease transmission of data.

At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web. It replaced the old ad hoc programs that updated the index and ran the various analysis. MapReduce's stable inputs and outputs are usually stored in a distributed file system. The transient data is usually stored on local disk and fetched remotely by the reducers. MapReduce is important because it allows ordinary developers to use MapReduce library routines to create parallel programs without having to worry about programming for intra-cluster communication, task monitoring or failure handling. It is useful for tasks such as data mining, log file analysis, financial analysis and scientific simulations. Several implementations of MapReduce are available in a variety of programming languages such as Java, C++, Python, Perl, Ruby, and C.

Programs written using MapReduce are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

In this paper, we present a simple tutorial about MapReduce framework starting from a brief overview about the framework, to its architecture, simple examples, execution overview, and Hadoop MapReduce framework. The rest of the paper is organized as follows. Section II presents a brief overview about MapReduce framework. Section III shows the programming model presented in MapReduce framework. Section IV presents simple examples of the framework. Section V presents the architecture of MapReduce. Fault tolerance in Map/Reduce is presented in Section VI. Section VII shows how Map/Reduce Framework is used in Hadoop. Section VIII concludes the paper.

II. Map/Reduce Overview

MapReduce is a programming model for large-scale distributed data processing. The main idea of MapReduce model comes from the map and reduce functions commonly used in functional programming. Map and reduce terms are used in this framework in the following meaning:

- *Map*: extract something we care about from each record of data.
- *Reduce*: aggregate, summarize, filter, or transform.

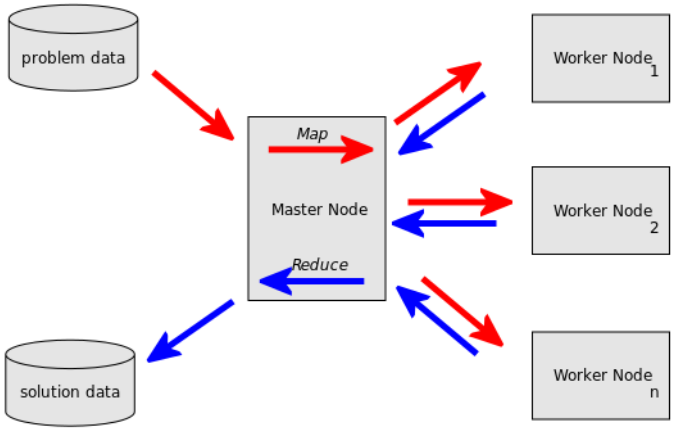

Fig. 1. Mapreduce Overview

Multi-core processors require parallelism, but many programmers are uncomfortable writing parallel programs. Using MapReduce, users do not need to be experts in writing parallel programs. The system automatically adapts to the available number of cores and machines. The framework is divided into two parts: (i) `Map`, a function that divides out work to different nodes in the distributed cluster; (ii) `Reduce`, another function that collates the work and resolves the results into a single value. The MapReduce framework is fault-tolerant because each node in the cluster is expected to report back periodically with completed work and status updates. If a node remains silent for longer than the expected interval, a master node makes note and re-assigns the work to other nodes.

Figure 1 [1] shows a simple illustration to Mapreduce framework. As shown in the figure, Map step works as following: The master node takes the input, divides it into smaller subproblems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node. While the Reduce step: The master node then collects the answers to all the subproblems and combines them in some way to form the output the answer to the problem it was originally trying to solve [1].

The key to how MapReduce works is shown in Figure 2, it takes input as, a list of records. The records are split among the different computers in the cluster by Map. The result of the Map computation is a list of key/value pairs. Reduce then takes each set of values that has the same key and combines them into a single value. So, Map takes a set of data chunks and produces key/value pairs and Reduce merges things, so that instead of a set of key/value pair sets, you get one result. However, MapReduce isn't intended to replace relational databases: it's intended to provide a lightweight way of programming things so that they can run fast by running in parallel on a lot of machines.

[1] http://en.wikipedia.org/wiki/File:Mapreduce_Overview.svg

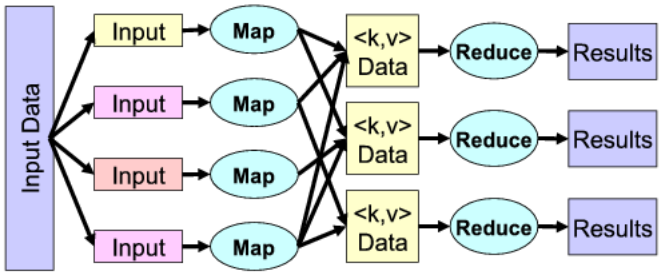

Fig. 2. MapeReduce Flow

III. Programming Model

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The programmer who uses the MapReduce library expresses the computation as two functions: Map and Reduce. Both functions are written by the programmer.

- *Map function*, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key 'I' and passes them to the Reduce function.
- *Reduce function*, accepts an intermediate key 'I' and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

IV. Map/Reduce Examples

A. Example 1

This simple example [2] illustrates the basic concept of MapReduce through a set of cities and its corresponding temperature values. Assume we have five files, and each file contains two columns that represent a city and the corresponding temperature recorded in that city for the various measurement days. In this example, city is the key and temperature is the value.

- Toronto, 20
- Cairo, 25
- New York, 22
- Rome, 32
- Toronto, 4
- Rome, 33
- New York, 18

Out of all the above data, we want to find the maximum temperature for each city across all of the data files (note that each file might have the same city represented multiple times). Using the MapReduce framework, we can break this down into five map tasks, where each mapper works on one of the five files and the mapper task goes through the data and returns the

[2] http://publib.boulder.ibm.com

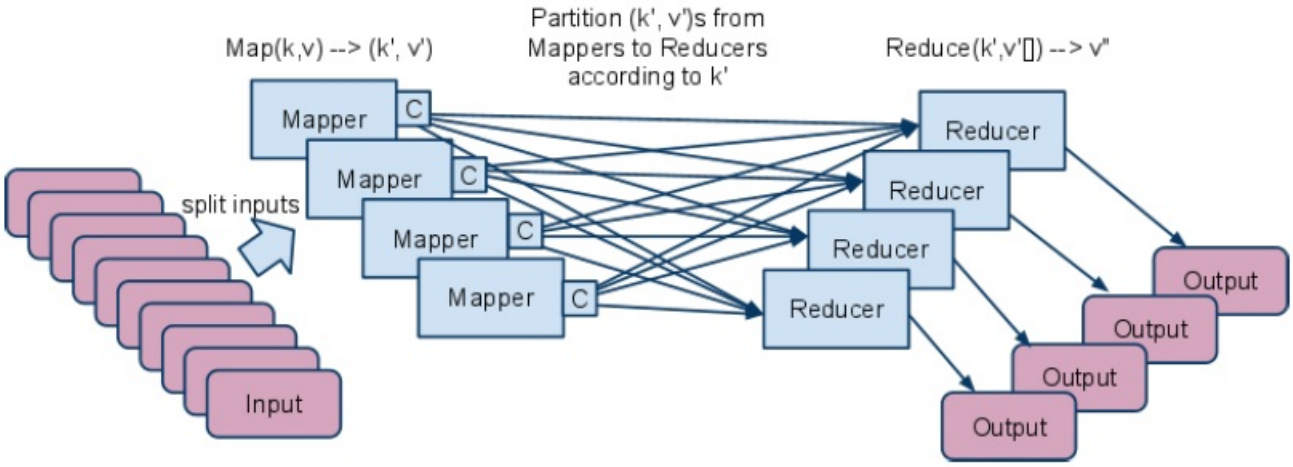

Fig. 3. Word Count Example

maximum temperature for each city. For example, the results produced from one mapper task for the data above would look like this: (Toronto, 20) (Cairo, 25) (New York, 22) (Rome, 33)

We assume that the other four mapper tasks (working on the other four files not shown here) produced the following intermediate results: (Toronto, 18) (Cairo, 27) (New York, 32) (Rome, 37)(Toronto, 32) (Cairo, 20) (New York, 33) (Rome, 38) (Toronto, 22) (Cairo, 19) (New York, 20) (Rome, 31)(Toronto, 31) (Cairo, 22) (New York, 19) (Rome, 30)

All five of these output streams would be fed into the reduce tasks, which combine the input results and output a single value for each city, producing a final result set as follows: (Toronto, 32) (Cairo, 27) (New York, 33) (Rome, 38)

B. Example 2

In the prototypical MapReduce example, proposed in [1], each document is split into words (Figure 3), and each word is counted by the map function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to reduce, thus this function just needs to sum all of its input values to find the total appearances of that word. The user would write code similar to the pseudo-code shown in Figure 4. The map function emits each word plus an associated count of occurrences, just '1' in this simple example. The reduce function sums together all counts emitted for a particular word.

C. Example 3

In this example [3], we can think of map and reduce tasks as the way a census was conducted in Roman times, where the census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital city. There, the results from each city would be reduced to a single count (sum of all cities) to determine the overall population of the empire. This mapping of people to cities, in parallel, and then combining the results (reducing) is much more efficient than sending a single person to count every person in the empire in a serial fashion.

[3]http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += ParseInt(pc)
  emit (word, sum)
```

Fig. 4. The prototypical MapReduce example

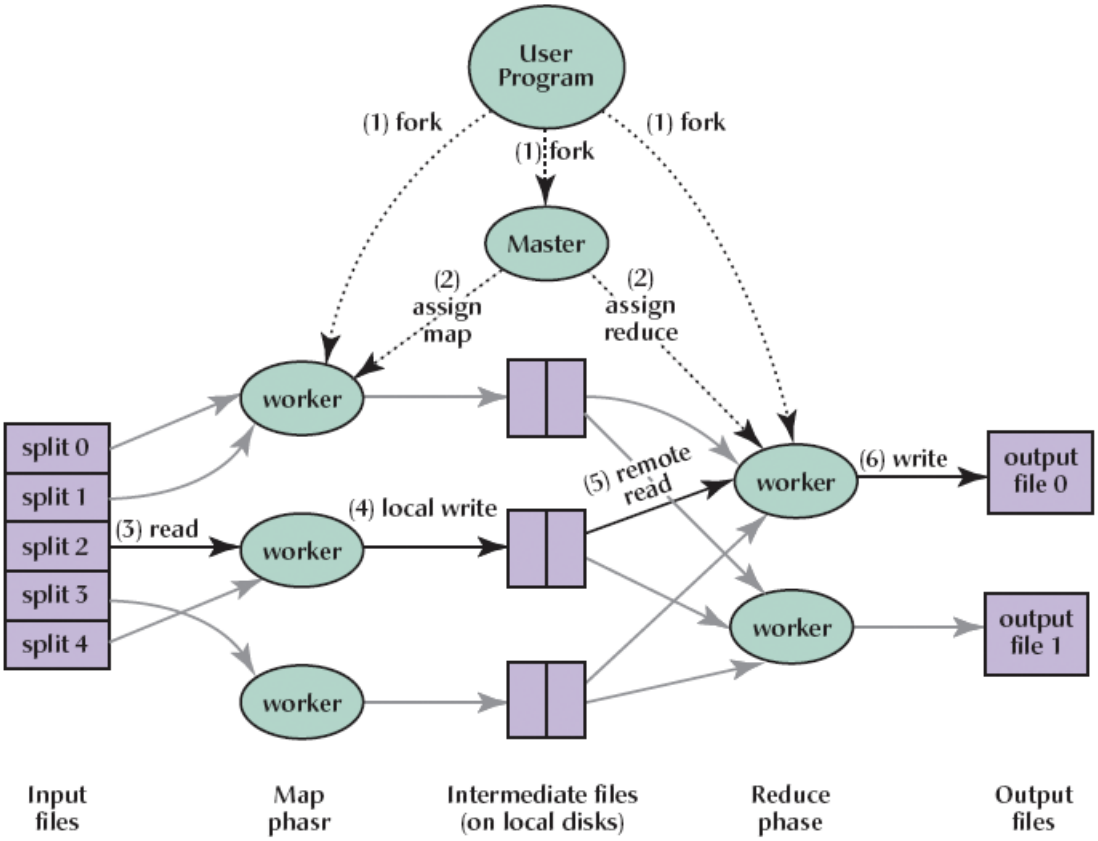

Fig. 5. Execution overview [1]

D. Example 4

The below two short examples show how MapReduce framework can be used to solve various realtime problems [1]:

Count of URL Access Frequency: The map function processes logs of web page requests and outputs (URL, 1). The reduce function adds together all values for the same URL and emits a (URL, total count) pair.

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word, frequency) pairs. The map function emits a (hostname, term vector) pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final (hostname, term vector) pair.

V. MapReduce Architecture

As described in [1], the Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function. The number of partitions (R) and the partitioning function are specified by the user.

Figure 5 shows the overall flow of a MapReduce operation. When the user program calls the MapReduce function, the following sequence of actions occurs– the numbered labels in Figure 5 correspond to the numbers in the list below:

1) The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2) The master node is a special program (worker) who assigns work to the rest of workers. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3) A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4) Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

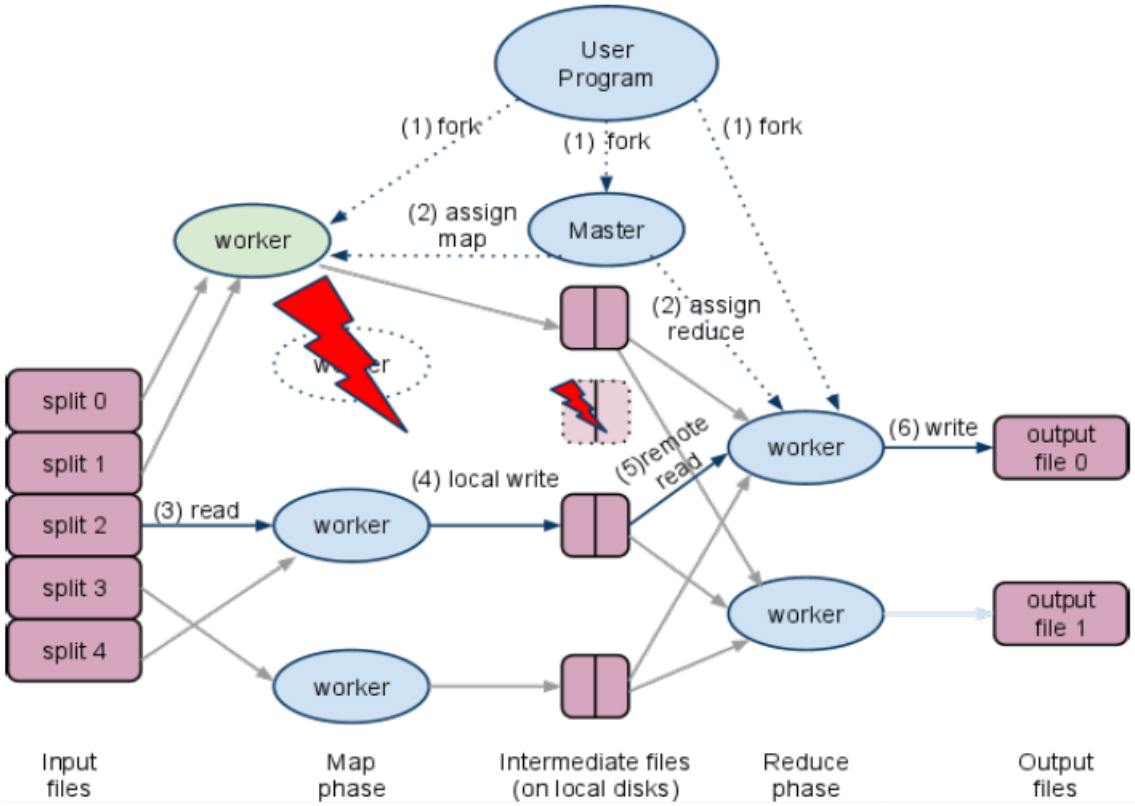

Fig. 6. Recover from Task Failure by Re-execution

5) When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6) The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the users Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7) When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the *R* output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these *R* output files into one file they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

VI. Fault Tolerance in Map/Reduce

Fault tolerance is one of the most critical issues for MapReduce. It is pointed out in [4] that for a system with 10 thousand of super reliable servers, there will be one failure per day. Also, every year 1-5% of disk drives will die and servers will crash at least twice for a 2-4% failure rate. Therefore, given that inexpensive commodity computers are used at Google, failures are expected to be much more often and hence fault tolerance becomes the most critical issue to be addressed. MapReduce handles failures through re-execution.

The master node pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. Master failures are not managed by current MapReduce implementations as designers consider failures unlikely in large clusters or in reliable Cloud environments.

Once a machine failure is detected, in-progress map or reduce tasks on that machine need to be re-executed on other machines as shown in Figure 6. Further, completed map tasks need to be re-executed as intermediate data produced by these tasks are stored in local disk and become inaccessible. However, completed reduce tasks do not need to be re-executed as

Stats for Month	Aug.'04	Mar.'06	Sep.'07
Number of jobs	29,000	171,000	2,217,000
Avg. completion time (secs)	634	874	395
Machine years used	217	2,002	11,081
Map input data (TB)	3,288	52,254	403,152
Map output data (TB)	758	6,743	34,774
reduce output data (TB)	193	2,970	14,018
Avg. machines per job	157	268	394
Unique implementations			
Mapper	395	1958	4083
Reducer	269	1208	2418

Fig. 7. Use of MapReduce inside Google [1]

their output is stored in a global file system. Failed tasks are rescheduled when machines are available.

VII. Hadoop Map/Reduce Framework

MapReduce programs have been implemented internally at Google over the past eight years, and an average of one hundred thousand MapReduce jobs are executed on Googles clusters every day [1]. Figure 7 shows some statistics for a subset of MapReduce jobs run at Google in various months, highlighting the extent to which MapReduce has grown and become the optimal choice for nearly all data processing needs at Google.

MapReduce is considered the heart of the famous data-intensive distributed framework Hadoop [4], an open source Apache project now used by Yahoo, Amazon, IBM, Facebook, Rackspace, Hulu, the New York Times, and a growing number of other companies. This programming paradigm allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster. Although the Hadoop framework is implemented in Java, MapReduce applications need not be written in Java.

The MapReduce framework and the Hadoop Distributed File System (HDFS) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master `JobTracker` and one slave `TaskTracker` per cluster node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the job configuration. The Hadoop job client then submits the job (jar/executable) and configuration to the `JobTracker` which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Figure 8 shows a simple program written in Java that uses Hadoop. The program is a simple application that counts the number of occurrences of each word in a given input set. The WordCount application is quite straight-forward. The Mapper implementation (lines 11-25), via the map method (lines 16-24), processes one line at a time, as provided by the specified `TextInputFormat` (line 48). It then splits the line into tokens separated by whitespaces, via the `StringTokenizer`, and emits a key-value pair of ((word), 1). The Reducer implementation (lines 27-38), via the reduce method (lines 29-37) just sums up the values, which are the occurrence counts for each key (i.e. words in this example). The run method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc. It then calls the `JobClient.runJob` (line 52) to submit the and monitor its progress.

VIII. Conclusion

In this paper, we presented a brief overview about MapReduce, which considered a flexible programming framework for many applications through a couple of restricted Map()/Reduce() constructs. Google invented and implemented MapReduce around its infrastructure to allow programmers scale with the growth of the Internet, and the growth of Google products/services. Open source implementations of MapReduce, such as Hadoop are creating a new ecosystem to enable large scale computing over the off-the-shelf clusters. MapReduce proves that restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Accordingly, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] M. Zaharia, "Job scheduling with the fair and capacity schedulers," *Hadoop Summit*, 2009. [Online]. Available: http://www.cs.berkeley.edu/matei/talks/2009/hadoopsummitfairscheduler.pdf

[3] Amazon elastic mapreduce. [Online]. Available: http://aws.amazon.com/elasticmapreduce/

[4] J. Dean, "Designs, lessons and advice from building large distributed systems," in *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), Big Sky, MT*, October 2009.

[5] Map/reduce tutorial. [Online]. Available: http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html/

[4] http://hadoop.apache.org/

```
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {
   public static class Map extends MapReduceBase implements Mapper<LongWritable,
   Text, Text, IntWritable> {
     private final static IntWritable one = new IntWritable();
     private Text word = new Text();

     public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
     output, Reporter reporter) throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
       }
     }
   }

   public static class Reduce extends MapReduceBase implements Reducer<Text,
       IntWritable, Text, IntWritable> {
     public void reduce(Text key, Iterator<IntWritable> values,
         OutputCollector<Text, IntWritable> output, Reporter reporter)
         throws IOException {
       int sum = 0;
       while (values.hasNext()) {
         sum += values.next().get();
       }
       output.collect(key, new IntWritable(sum));
     }
   }

   public static void main(String[] args) throws Exception {
     JobConf conf = new JobConf(WordCount.class);
     conf.setJobName("wordcount");
     conf.setOutputKeyClass(Text.class);
     conf.setOutputValueClass(IntWritable.class);
     conf.setMapperClass(Map.class);
     conf.setCombinerClass(Reduce.class);
     conf.setReducerClass(Reduce.class);
     conf.setInputFormat(TextInputFormat.class);
     conf.setOutputFormat(TextOutputFormat.class);
     FileInputFormat.setInputPaths(conf, new Path(args[0]));
     FileOutputFormat.setOutputPath(conf, new Path(args[1]));
     JobClient.runJob(conf);
   }
}
```

Fig. 8. WordCount Program [5]