GLOBAL JOURNAL OF ENGINEERING SCIENCE AND RESEARCHES

MAP REDUCE: PARALLEL DATA PROCESSING

Shilpa Suhag
Department of Computer Science, Ganga Technical Campus

ABSTRACT

Techniques that effectively process such massive amount of data. One such technique is MapReduce a parallel programming model which is invented by researchers at Google. It is meant for processing large amounts of data by a large cluster. It was hard to parallelize the big data over hundreds of machines, so MapReduce model was developed. Recent developments in open source software, the Hadoop project and associated software provide a foundation for scaling data warehouses on Linux clusters, which are providing fault-tolerant and parallelized analysis. MapReduce consists of a *map phase* where input data is split into discreet chunks to be processed followed by the *reduce phase* where the output of the map phase is aggregated to produce the desired result. The simple nature of the programming model lends to very efficient and extremely large-scale implementations across commodity nodes. Apache Hadoop MapReduce is the most popular open-source implementation of the MapReduce model. MapReduce is paired with a distributed file-system such as Apache Hadoop HDFS, which can provide very high aggregate I/O bandwidth across a large cluster. Also the MapReduce tasks can be scheduled on the same the cluster. This significantly reduces the network I/O patterns and keeps most of the I/O on the local disk or within the same rack. This paper introduces MR-J, a MapReduce Java framework for multi-core architectures.

Keywords: *Map Reduce, parallel software framework, Hadoop.*

I. INTRODUCTION

Apache Hadoop MapReduce is an open-source, which is an implementation of the MapReduce programming paradigm described above. Apache Hadoop MapReduce project can be broken down into the following major facets:

The end-user **MapReduce API for** programming the desired MapReduce application.

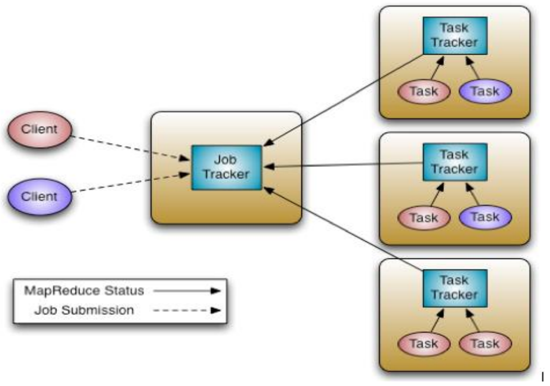
The **MapReduce framework,** which is the runtime implementation of various phases such as the map phase, the sort/shuffle/merge aggregation and the reduce phase.

The **MapReduce system**, which is the backend infrastructure required to run the user's MapReduce application, manage cluster resources, schedule jobs.

This concern has significant benefits, particularly for the end-users: they can completely focus on the application via the API and allow the combination of the MapReduce Framework and the MapReduce System to deal with the details such as resource management, fault-tolerance, scheduling etc. The current Apache Hadoop System is composed of the **JobTracer,** which is the master, and the per-node slaves called **TaskTracer.**

The JobTracker is responsible for *resource management* (managing the worker nodes i.e. Task Trackers), *tracking resource consumption or availability* of job *life-cycle management* (scheduling individual tasks of the job, tracking progress, providing fault-tolerance for tasks etc).The TaskTracer has simple responsibilities to distribute tasks on orders from the JobTracker and provide task-status information to the JobTracker periodically.

Since the MapReduce programming pattern was invented by Google. Its roots can be traced back to functional programming. Also, it has attracted attention from industry, academics and open-source projects (Hadoop [11]), since Google made public this pattern was easy to use and provided a highly effective means of attaining massive parallelism in large data centres, in their experience. Our objective is to investigate the MapReduce pattern in the context of multi-core architectures and not within data-centers as commonly used by Amazon, Facebook, Google and Yahoo [5].



Here Java is selected as the programming language because the main open-source implementation of MapReduce, i.e. Hadoop, is also developed in this language. So, our efforts and findings can contribute directly to the evolution of Hadoop. These efforts have produced MR-J, a MapReduce Java framework for multi-core architectures. The Phoenix framework [11] is the only other MapReduce framework that targets multi-core architectures. Since Phoenix is implemented in C, the implementation can take advantage of pointers (e.g., to avoid copying data) in ways that are not feasible within Hadoop or MR-J.

The map function takes <key, value> pair as input and produces an intermediate <key, value> pair as output. The inputs to the map and reduce functions are processed without any dependency on other elements of data, avoiding the need for synchronisation and contention for shared data. The majority of the frameworks implementing the MapReduce pattern has at least map, merge and reduce phases. In the map phase, the map function is executed in parallel by different worker threads as map subtasks. The output from each map subtask is written to a local data structure; (such as arrays, lists, or queues). The execution of the map phase is followed by the merge phase. In this phase the runtime system combines the intermediate <key, value> pair output from each map subtask so that values from the same keys are grouped together to form a unique <key, value> pair. The framework partitions the unique <key, value> pairs among the reduce task workers. As with the map task, the reduce tasks are executed in parallel without any dependencies on other elements. The reduce task usually performs some kind of reduction operation such as summation, sorting and merging operations. The output from each reduce task worker is written to either a DFS in the case of cluster-based implementations, or it is written to a local data structure, which is then merged to produce a single output, in the case of multi-core architectures. Cluster-based implementations of MapReduce (such as Google's and Hadoop) normally target large data-centers using commodity systems interconnected using high-speed Ethernet networks. These implementations rely on specialised distributed file systems such as Google's GFS and Hadoop's HDFS to manage data across the distributed network. The implemented runtime system spawns worker threads on each node in the cluster. Each of these worker threads is assigned either a map or a reduce task. Only one of the worker threads is elected to be the master. Each job consists of a set of map and reduces tasks along with the input data. The runtime system automatically partitions the input data based on a splitting function into smaller partitions or chunks. The selection of the chunk size is based on the block size of the distributed file system used by the framework. In the case of Google a default chunk size of 16MB to 64MB is used, whereas in the case of Hadoop a default chunk size of 64MB is used. The master worker assigns these partitioned data to map task workers, which are distributed among the cluster nodes. Each map task worker processes the input data in parallel without any dependency. The output from the map task worker is stored locally on the node on which the task is executing. Once all the map tasks are completed, the runtime system automatically sorts the output from each map task worker so that the values from the same intermediate keys are grouped together before it is assigned to the reduce task worker. The reduce tasks are assigned to the reduce task workers by partitioning the intermediate sorted keys using the partitioning function. The default partitioning function used in both Google's model and Hadoop is based on key hashing: hash(key) mod R, where R is the number of reduce task workers.

ISSN 2348 – 8034
Impact Factor- 5.070

In a distributed environment the master is responsible for the following tasks. It maintains the status information and identity for map and reduce tasks. It is responsible for transferring the location of the file from the map task worker to the reduce task worker. It delegates map or reduce tasks to each worker, maintains locality by assigning tasks locally to the workers and manages the termination of each worker. An essential feature for any cluster-based implementation is its ability to detect failures and slow tasks during execution. This is important because machine failures can be frequent due to the large cluster size. Both Google's framework and Hadoop implement effective fault-tolerance mechanisms that can detect slow nodes and node failures.

The paper is organised as follows. Section 2 presents the background on MapReduce and a brief description of its programming model. Section 3 takes a closer look at a MapReduce implementation i.e. Phoenix. Section 4 describes pros and cons of MapReduce. Section 5 shows the results from our first experiments on a small multi-core (Intel core i7) system using 4 different benchmarks. Finally Section 5 summarises MR-J and its first performance evaluation.

1.1 Background on MapReduce

Now we will discuss three widely accepted implementations of MapReduce framework that are used as a background study for this research. They include the following; Google's MapReduce model for large cluster based environments [1], Phoenix [6], a shared memory model for multi-core architectures and Hadoop, an open source implementation for large cluster based environments. MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been generalized. By 2014, Google was no longer using MapReduce as their primary Big Data processing model,[10]and development on Apache MapReduce had moved on to more capable and less disk-oriented mechanisms that incorporated full map and reduce capabilities[4].

MapReduce Programming Model:

The advantage of using MapReduce model is that it provides automatic parallelisation through functional programming construct. The functional programming approach makes the programming model easy to use and highly effective in attaining massive parallelism. In general, the implementation of MapReduce model can be broken down into two main categories; the API and Runtime system.

The API can be broadly divided into User defined and System defined functions. User defined function generalises the functions implemented by the programmer to express the application logic. The core User defined function includes map and reduce functions. The system-defined function specifies the functions provided by the MR-J runtime system to programmer. These functions mainly consist of initialisation functions, functions used to store the output key/value pairs into a transient and persistent location and functions required to configure the runtime system

The important feature of the MapReduce API is its restricted interface. It has been proven that the use of a narrow and simplified interface has helped programmers in defining the application logic more accurately and has made it easier to debug and maintain [1,4]. Restricting the API had also helped to improve its usability by lowering the learning curve associated with it when compared to other commercially available API for parallelism such as OpenMP.

The following pseudo code shows the basic structure of a MapReduce program that counts the number of occurrences of each word in a collection of documents. The map function emits each word in the documents with the temporary count 1. The reduce function sums the counts for each unique word.

.

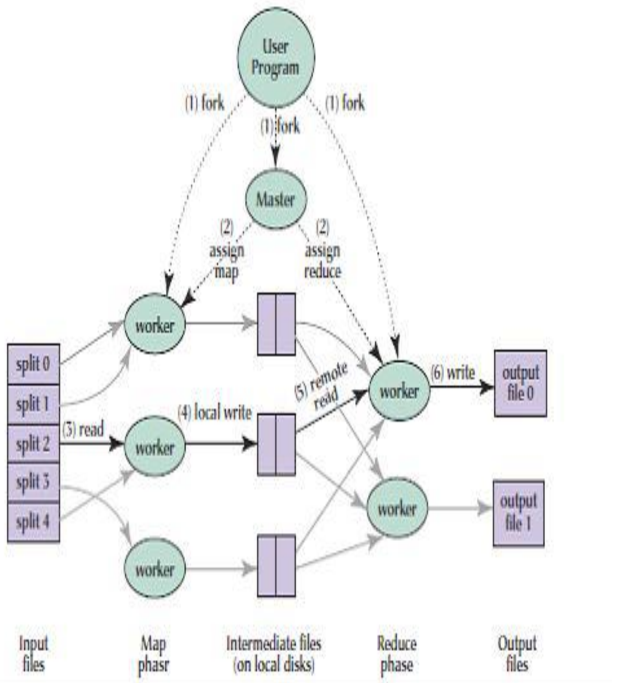
```
// input: a document
// intermediate output: key=word; value=1
Map(void *input) {
for each word w in input
```

ISSN 2348 – 8034
Impact Factor- 5.070

```
EmitIntermediate(w, 1);
}
// intermediate output: key=word; value=1
//output: key=word; value=occurences
Reduce(String key, Iterator values)
int result = 0;
for each v in values
result += v;
Emit(w, result);
}
```

The programme provides a simple description of the algorithm that focuses on functionality and not on parallelization. The actual parallelization and the details of concurrency management are left to the runtime system. Hence the program code is generic and easily portable across systems. The model provides sufficient high-level information for parallelization. The Map function can be executed in parallel on non-overlapping portions of the input data and the Reduce function can be executed in parallel on each set of intermediate pairs with the same key.

Theoretical Representation of Map Reduce:
1. Data are represented as a <key, value> pair.
2. Map: <key, value> → multiset of <key, value> pairs user defined, easy to Parallelize.
3. Shuffle: Aggregate all <key, value> pairs with the same key, executed by underlying system
4. Reduce: <key, multiset(value)> → <key, multiset(value)> user defined, easy to parallelize.
5. Can be repeated for multiple rounds [9].



ISSN 2348 – 8034
Impact Factor- 5.070

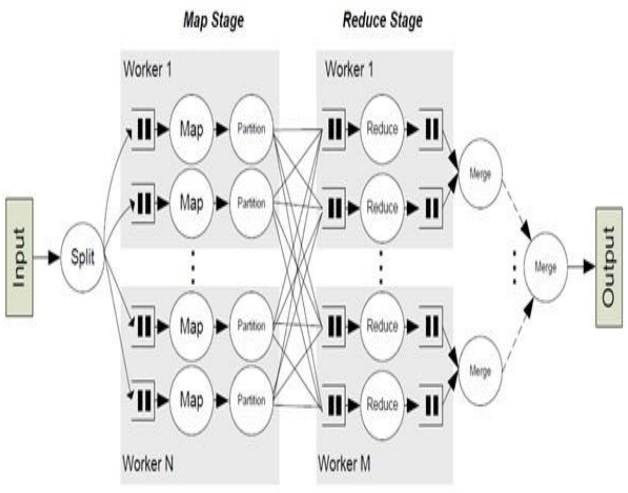
II. THE PHOENIX SYSTEM

The Phoenix System [11] is an implementation of MapReduce. It is made for shared-memory systems, instead of large, distributed clusters of computers. It only uses the threads that are available on one single computer. Its goal is to support efficient execution on multiple cores without burdening the programmer with concurrency management. Phoenix consists of a simple API that is visible to application programmers and an efficient runtime that handles parallelization, resource management, and fault recovery.

2.1 The Phoenix API

The Phoenix API is the interface of Phoenix that was originally written in C, but with version 2.0 an implementation for C++ was added. The programmer only has to create a few simple functions (e.g. Map and Reduce) and the library will handle the threading and load balancing.

Figure shows the model of the data flow in the Phoenix API.



The user program initialises the scheduler by invoking an initialisation function. The scheduler spawns several worker threads according to the number of cores supported by the multi-core chip. Each worker thread is assigned map or reduce tasks by the scheduler. The scheduler partitions the input data using the splitter function. The default splitter function uses the cache sizes of the chip to determine the chunk size. Partitioned data is forwarded to the map task workers assigned dynamically by the scheduler. Once the map task worker completes processing the input data, it stores the output in a buffer. It also merges the values from the same key to form a unique key for each set of values. Reduce tasks start only when the entire map task completes.

III. PROS AND CONS

3.1 Advantages

MapReduce is simple and efficient for computing aggregate. Thus, it is often compared with “filtering then group-by aggregation” query processing in a DBMS. Here are major advantages of the MapReduce framework for data processing.

Simple and easy to use TheMapReduce model is simple but expressive. With MapReduce, a programmer defines his job with only Map and Reduce functions, without having to specify physical distribution of his job across nodes.

Flexible MapReduce does not have any dependency on data model and schema. With MapReduce a programmer can deal with irregular or unstructured data more easily than they do with DBMS.

Independent of the storage MapReduce is basically independent from underlying storage layers. Thus, MapReduce can work with different storage layers such as BigTable and others.

Fault tolerance MapReduce is highly fault-tolerant. For example, it is reported that MapReduce can continue to work in spite of an average of 1.2 failures per analysis job at Google.

High scalability The best advantage of using MapReduce is high scalability. Yahoo! reported that their Hadoop gear could scale out more than 4,000 nodes in 2008.

3.2 Pitfalls

Despite many advantages, MapReduce lacks some of the features that have proven paramount to data analysis in DBMS. In this respect, MapReduce is often characterized as an Extract-Transform-Load (ETL) tool . We itemize the pitfalls of the MapReduce framework below, compared with DBMS.

No high-level language MapReduce itself does not support any high-level language like SQL in DBMS and any query optimization technique. Users should code their operations in Map and Reduce functions.

No schema and no index MapReduce is schema-free and index-free. An MR job can work right after its input is loaded into its storage. However, this impromptu processing throws away the benefits of data modelling. MapReduce requires parsing each item at reading input and transforming it into data objects for data processing, causing performance degradation.

A Single fixed dataflow MapReduce provides the ease of use with a simple abstraction, but in a fixed Data flow. Therefore, many complex algorithms are hard to implement with Map and Reduce only in an MR Job. In addition, some algorithms that require multiple inputs are not well supported since the dataflow of MapReduce is originally designed to read a single input and generate a single output.

Low efficiency with fault-tolerance and scalability as its primary goals, MapReduce operations are not always optimized for I/O efficiency. (Consider for example sort-merge based grouping, materialization of intermediate results and data triplication on the distributed file system.) In addition, Map and Reduce are blocking operations. A transition to the next stage cannot be made until all the tasks of the current stage are finished. Consequently, pipeline parallelism may not be exploited. Moreover, block-level restarts, a one-to-one shuffling strategy, and a simple runtime scheduling can also lower the efficiency per node. MapReduce does not have specific execution plans and does not optimize plans like DBMS does to minimize data transfer across nodes. Therefore, MapReduce often shows poorer performance than DBMS[12]. In addition, the MapReduce framework has a latency problem that comes from its inherent batch processing nature. All of inputs for an MR job should be prepared in advance for processing.

Very young MapReduce has been popularized by Google since 2004. Compared to over 40 years of DBMS,codes are not mature yet and third-party tools available are still relatively few.

IV. CONCLUSIONS

MapReduce programming model was originally proposed by Google for cluster-based environments and is found to be a simple and effective programming model that can process massive amounts of data in parallel. Several implementations of this model have proved its suitability on various architectures. The primary advantage of this model is that it automatically parallelises a given sequential implementation of the application logic expressed using Map and Reduce functions by hiding the low-level parallelisation details. It is this abstraction of low level parallelisation details that helps in maximising the programming efficiency for the programmers by enabling them to focus more on the application logic rather than the parallelisation details. This chapter has provided a brief discussion on three widely accepted implementations of MapReduce model that are used as background research for developing MR-J, a MapReduce framework for multi-core architecture.

REFERENCE

1. *Anurag International Journal of Advance Research in Computer Science and Management Studies Volume 3, Issue 7, July 2015 pg. 256-272*
2. *Harris, Derrick (2014-03-27). "Apache Mahout, Hadoop's original machine learning project, is moving on from MapReduce". Retrieved 2015-09-24.*
3. *Siddaraju, Sowmya C.L., Rashmi K. and Rahul M., "Efficient analysis of Big Data using MapReduce framework," International Journal of Recent Development in Engineering and Technology (IJRDET), vol. 2 issue 6, June 2014.*
4. *http://www-01.ibm.com/software/in/data/bigdata/ Shilpa et al., International Journal of Advanced Research in Computer Science and Software Engineering 3(10), October - 2013, pp. 991-99.*
5. *Kyuseok Shim, MapReduce Algorithms for Big Data Analysis, DNIS 2013, LNCS 7813, pp. 44 -48, 2013.*
6. *Yuri Demchenko —The Big Data Architecture Brainstorming Session at the University of Amsterdam 17 July 2013.*
7. *E. Dumbill, "What is Big Data? An introduction to the Big Data landscape." Retrieved from https://beta.oreilly.com/ideas/what-is-big-data, 2012.*
8. *K.H. Lee, Y.J. Lee, H. Choi, Y.D. Chung and B. Moon, "Parallel data processing with MapReduce: A survey," SIGMOD Record, vol. 40 no.4, December 2011.*
9. *X. Feng, R. Grossman and L. Stein, "PeakRanger: A cloud-enabled peak caller for ChIP-seq data," BMC Bioinformatics 2011, 12:139.*
10. *"Hadoop: Open source implementation of MapReduce", http://lucene.apache.org/hadoop/*
11. *T. White, "Hadoop: The Definitive Guide", O'Reilly, 2009.*
12. *RAM Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a NUMA System" In Proceedings of the International Symposium on Workload Characterization (IISWC), pp. 198-207, 2009.*
13. *B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008.*
14. *D.J. DeWitt, and M. Stonebraker, "MapReduce: A major step backwards" The Database Column, 2008.*
15. *J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters" Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008.*
16. *"The Hadoop Distributed File System," The Apache Software Foundation, 2006. [Online] Available: http://hadoop.apache.org/common/docs/current/hdfs_design.html.[Accessed:Feb,2009].*
17. *[C. Ranger, R.Raghuraman, A.Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems" In Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA), pp. 13-24, 2007*