Global Journal of Computer Science and Technology: C
Software & Data Engineering
Volume 14 Issue 9 Version 1.0 Year 2014
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Improved Approaches to Handle Bigdata through Hadoop

By K. Sandeep, K. Kondaiah, A. ineetha & Ch. Monica

KLEF University, India

Abstract- Big data is an evolving term that describes any voluminous amount of structured, semi-structured and unstructured data that has the potential to be mined for information.Today's world produces a large amount of data from various sources, records and from different fields termed as "BIG DATA". Such huge data is to be analyzed, and filtered using various techniques and algorithms to extract the interested and useful data to gain knowledge. In the new era with the boom of both structured and unstructured types of data, in the field of genomics, meteorology, biology, environmental research and many others, it has become difficult to process, manage and analyze patterns using traditional databases and architectures. It requires new technologies and skills to analyze the flow of material and draw conclusions. So, a proper architecture should be understood to gain knowledge about the Big Data. The analysis of Big Data involves multiple distinct phases such as collection, extraction, cleaning, analysis and retrieval.

GJCST-C Classification : *H.2.8, H.2.6*

Strictly as per the compliance and regulations of:

Improved Approaches to Handle Bigdata through Hadoop

K. Sandeep [α], K. Kondaiah [σ], A. vineetha [ρ] & Ch. Monica[ω]

Abstract- Big data is an evolving term that describes any voluminous amount of structured, semi-structured and unstructured data that has the potential to be mined for information.Today's world produces a large amount of data from various sources, records and from different fields termed as "BIG DATA". Such huge data is to be analyzed, and filtered using various techniques and algorithms to extract the interested and useful data to gain knowledge. In the new era with the boom of both structured and unstructured types of data, in the field of genomics, meteorology, biology, environmental research and many others, it has become difficult to process, manage and analyze patterns using traditional databases and architectures. It requires new technologies and skills to analyze the flow of material and draw conclusions. So, a proper architecture should be understood to gain knowledge about the Big Data. The analysis of Big Data involves multiple distinct phases such as collection, extraction, cleaning, analysis and retrieval.This paper presents detailed analysis of Hadoop and MapReduce programming Model and also the challenges that Apache Hadoop, the popular data storage and analysis platform used by major number of large companies is facing and future scope of implementation of Hadoop and various other new improvements to the challenges.

I. Introduction

Apache Hadoop, the popular data storage and analysis platform, has generated a great deal of interest recently. Large and successful companies are using it to do powerful analyses of the data they collect. Hadoop offers two important services: It can store any kind of data from any source, inexpensively and at very large scale, and it can do very sophisticated analysis of that data easily and quickly.

Unlike older database and data warehousing systems, Hadoop is different and those differences can be confusing to users. What data belongs in a Hadoop cluster? What kind of questions can the system answer? Understanding how to take advantage of Hadoop requires a deeper knowledge of how others have applied it to real-world problems that they face.

This paper presents detailed analysis of Hadoop and MapReduce programming Model and also the challenges that hadoop is facing and future scope of implementation of hadoop and various other new algorithms.

Author α σ ρ ω: Department of Electronics and Computer engineering KLEF University. Vaddeswaram, Tadepalli (Mandal), Guntur(dist), 522502 A.P, INDIA. e-mails: Sandeep.k@gmail.com, Kondaya.Kuppala@gmail.com, mailvivacious@gmail.com, monica cherukuri09@gmail.com

II. What is Hadoop?

Hadoop is data storage and processing system. It is scalable, fault-tolerant and distributed. Hadoop was originally developed by the world's largest internet companies to capture and analyze the data that they generate. Unlike older platforms, Hadoop is able to store any kind of data in its native format and to perform a wide variety of analyses and transformations on that data. Hadoop stores terabytes, and even petabytes, of data inexpensively. It is robust and reliable and handles hardware and system failures automatically, without losing data or interrupting data analyses. Hadoop runs on clusters of commodity servers. Each of those servers has local CPU and storage. Each can store a few terabytes of data on its local disk.

Hadoop supports applications under a free license. Three critical components of Hadoop system are:

1. Hadoop Common : Common Utilities Package
2. HFDS: Hadoop Distributed File System with high throughput access to application data.
3. MapReduce: A software framework for distributed processing of large data sets on computer clusters.

*The Hadoop Distributed File System, or HDFS:*HDFS is the storage system for a Hadoop cluster. When data arrives at the cluster, the HDFS software breaks it into pieces and distributes those pieces among the different servers participating in the cluster. Each server stores just a small fragment of the complete data set, and each piece of data is replicated on more than one server.

A distributed data processing framework called Map Reduce: Because Hadoopstores the entire dataset in small pieces across a collection of servers, analytical jobs can be distributed, in parallel, to each of the servers storing part of the data. Each server evaluates the question against its local fragment simultaneously and reports its results back for collation into a comprehensive answer. MapReduce isthe plumbing that distributes the work and collects the results.

Hadoop is high-performance distributed data storage and processing system. Its two major subsystems are HDFS, for storage, and MapReduce, for

parallel data processing.Hadoop automatically detects and recovers from hardware and software failures. HDFS and MapReduce will help in performing this.

Hadoop stores any type of data, structured or complex, from any number of sources, in its natural format. No conversion or translation is required on ingest. Data from many sources can be combined and processed in very powerful ways, so that Hadoop can do deeper analyses than older legacy systems. Hadoop integrates cleanly with other enterprise data management systems. Moving data among existing data warehouses, newly available log or sensor feeds and Hadoop is easy. Hadoop is a powerful new tool that complements current infrastructure with new ways to store and manage data at scale.

MapReduce: Simplified Data Processing on Large Clusters

MapReduce is a programming model and software framework first developed by Google (Google's MapReduce paper submitted in 2004) intended to facilitate and simplify the processing of vast amounts of data in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. Computational processing occurs on both:

- Unstructured data: file system.
- Structured data: database.

MapReduce framework

1. Per cluster node:
   1.1) Single JobTracker per master
      a. Responsible for scheduling the jobs' component tasks on the slaves.
      b. Monitors slave progress
      c. Re-executing failed tasks
   1.2) Single TaskTracker per slave
      a. Execute the tasks as directed by the master.

MapReduce Core Functionality:

1. Code usually written in Java- though it can be written inother languages with the Hadoop Streaming API.
2. Two fundamental pieces:
   a. Map step
      i. Master node takes large problem input and slices it into smaller sub problems; distributes these to worker nodes.
      ii. Worker node may do this again; leads to a multi-level tree structure
      iii. Worker processes smaller problem and hands back to master
   b. Reduce step
      i. Master node takes the answers to the sub problems and combines them in a predefined way to get the output/answer to original problem.
3. Data flow beyond the two key pieces (map and reduce):
   a. Input reader – divides input into appropriate size splits which get assigned to a Map function.
   b. Map function – maps file data to smaller, intermediate<key, value> pairs
   c. Partition function – finds the correct reducer: given the key and number of reducers, returns the desired Reduce node.
   d. Compare function – input for Reduce is pulled from the Map intermediate output and sorted according to this compare function.
   e. Reduce function – takes intermediate values and reduces to a smaller solution handed back to the framework.
   f. Output writer – writes file output.
4. A MapReduce Job controls the execution
   i. Splits the input dataset into independent chunks.
   ii. Processed by the map tasks in parallel.
5. The framework sorts the outputs of the maps.
6. A MapReduce Task is sent the output of the framework to reduce and combine.
7. Both the input and output of the job are stored in a file system.
8. Framework handles scheduling.

MapReduce Input and Output

1. MapReduce operates exclusively on <key, value> pairs.
2. Job Input : <key, value> pairs.
3. Job Output : <key, value> pairs. Conceivably of different types.
4. Key and value classes have to be serializable by the framework.
5. Default serialization requires keys and values to implement Writable.
6. Key classes must facilitate sorting by the framework.

Execution of Input and output parameters in typical MapReduce Framework

This execution of Map and Reduce algorithm is further explained in the implementation section.

Understanding Map and Reduce

Let us consider a simple problem wherein we have to search for a pattern 'cs396t' in a collection of files. We would typically run a command like this:
grep -r "cs395t" <directory>

Now, suppose you have to do this search over terabytes of data and you have a cluster of machines at your disposal? How can you make this grep faster?
Build a distributed grep!

Now the question arises, do we really need to consider a distributed grep? Why can't we just use our desktop for processing.Considering this in mind, let us estimate how much time will the average desktop system will take to process to search over terabytes of data.

In general, Considering an average read speed of 90MB/s: ~3.23 hours (Numbers are for Western Digital 1TB SATA/300 drive)

If you use an SSD with read speed of 350MB/s: ~50 minutes (Numbers are for Crucial 128 GB m4 2.5-Inch Solid State Drive SATA 6Gb/s)

This seems to be a huge amount of time considering the real time of data demanding requests from the internet. This is an approx. time only for searching through a collection of files. It would be huge amount of time when asked to sort a terabyte of data.

This definitely proves to be a wonder solution to the amount of time it takes to sort and work through huge collection of data. But, keeping this in mind, I could actually build a distributed system which does the same amount of work in this time. Can we ? The answer is an absolute NO!

Algorithmic Analysis :

```
var a = [1,2,3];
for (i=0; i<a.length; i++)
a[i] = a[i] * 2;
for (i=0; i<a.length; i++)
a[i] = a[i] + 2;
```

I can change it to:

```
function map(fn, a) {
for (i = 0; i<a.length; i++)
a[i] = fn(a[i]);
}
map(function(x){return x*2;}, a);
map(function(x){return x+2;}, a);
function sum(a) {
var s = 0;
for (i = 0; i<a.length; i++)
 s += a[i];
return s;
}
function join(a) {
var s = "";
for (i = 0; i<a.length; i++)
 s += a[i];
return s;
}
alert(sum([1,2,3]));
alert(join(["a","b","c"]));
function reduce(fn, a, init) {
var s = init;
for (i = 0; i<a.length; i++)
 s = fn( s, a[i] );
return s;
}
function sum(a) {
return reduce(function(a, b){return a+b;}, a, 0);
}
function join(a) {
return reduce(function(a, b){return a+b;}, a, “” );
}
alert(sum([1,2,3]));
alert(join(["a","b","c"]));
```

1. Passing functions as arguments – functional programming.
2. map – does something to every element in an array – can be done in any order! (amendable to parallelization)
3. So, if you have 2 CPUs, map will run twice as fast.
4. map is an example of embarrassingly parallel computation.

Suppose you have a huge array with elements which are all the webpages from the Internet. To search the whole internet:

1. you just need to pass a string_searcher function to map
2. reduce will be an identity function
3. run a MapReduce job on a cluster
4. that’s it! You are searching the Internet by writing just a few lines of code!

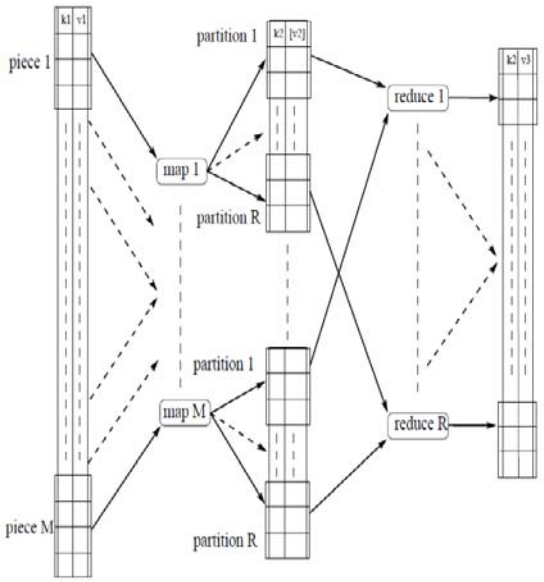
MAP- function that takes key/value pairs as input and generates an intermediate set of key/value pairs.

REDUCE- function that merges all the intermediate values associated with the same intermediate key.

User needs to define these two functions.

map: (k1, v1) □ list(k2, v2)

reduce: (k2, list(k2, v2)) □ list(v2)



EXAMPLE - WORD COUNT

Problem : counting occurrences of words in a large collection of documents.

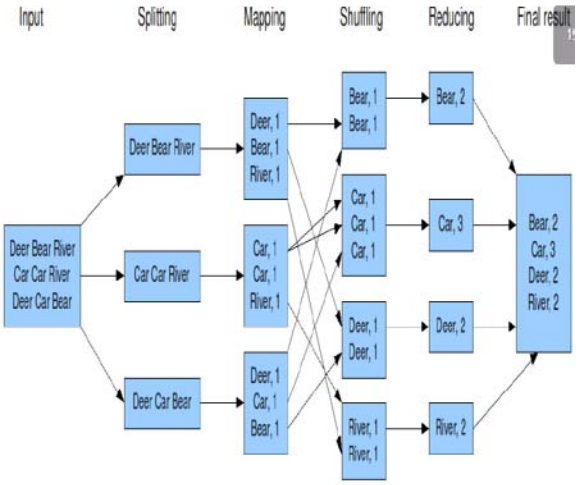
```
map(String key, String value):
 // key: document name
 // value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
 // key: a word
 // values: a list of counts
```

```
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

Other than map and reduce, user needs to provide:

1. names of input and output files
2. optional tuning parameters (size of split, M, R, etc.)

User's code is linked with MapReduce library and the binary is submitted to a task runner.



Word Counting using MapReduce

Other Examples of MapReduce :

1. Distributed grep
   a. map emits a line if it matches the given pattern
   b. reduce just copies input to output
2. Counting URL access frequency
   a. map processes web server logs and outputs <URL, 1>
   b. reduce sums all numbers for a single URL
3. Inverted index
   a. map function parses document and emits <word, docID>
   b. reduce gets all pairs for a given word and emits <word, list(docID)>

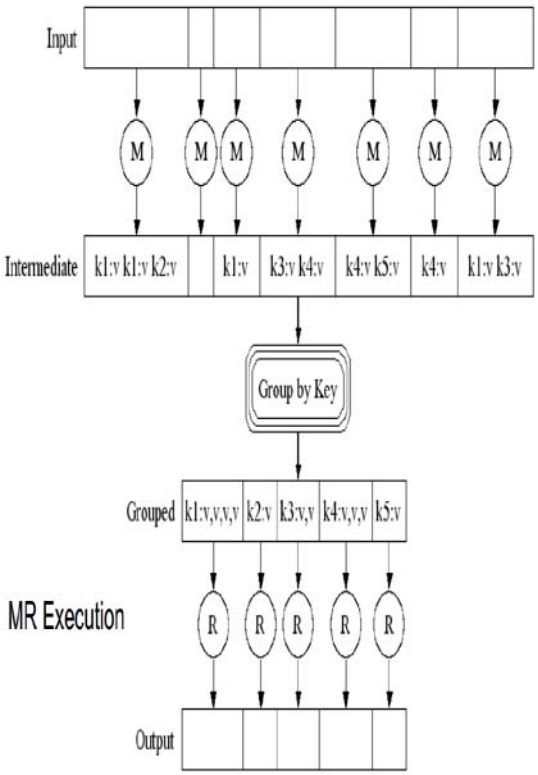
Implementing Map and Reduce in real world Scenarios :

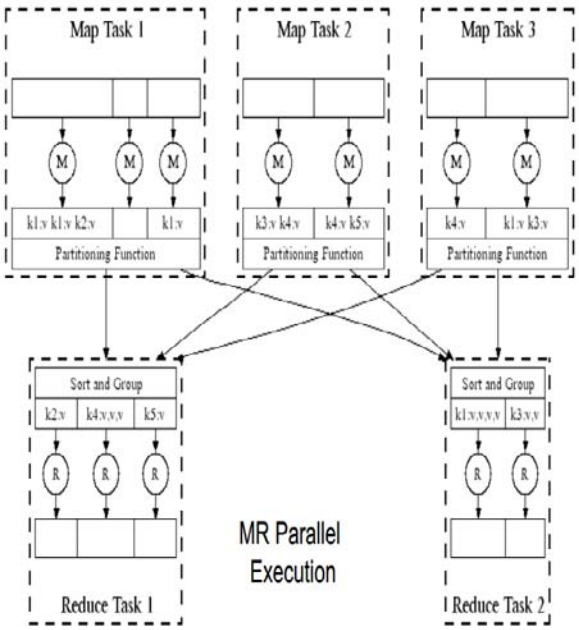
Map and Reduce can achieve the following achieving great scalability and speed.

1. Exploit parallelism in the computation.
2. Massively scalable – can run on hundreds or thousands of machines.
3. Hide the details of cluster management tasks like scheduling of tasks, partitioning of data, network communication from the user.
4. Fault tolerant (in large clusters failures are a norm rather than being an exception)

Opportunities for Parallelism using Map and Reduce:

1. Input – all key/value pairs can be read and processed in parallel by map
2. Intermediate grouping of data – essentially a sorting problem; can be done in parallel and results can be merged.
3. Output – All reducers can work in parallel. Each individual reduction can be parallelized.





MASTER : (in reference to fig 1)

1. Only 1 Master per MR computation.
2. Master:
   a. assigns map and reduce tasks to the idle workers.
   b. informs the location of input data to mappers.
   c. stores the state (idle, in-progress, completed) and identity of each worker machine.
   d. for each completed map task, master stores the location and sizes of intermediate files produced by the mapper; this information is pushed to workers which have in-progress reduce tasks.
3. Split the input into M pieces and start copies of program on different machines.
4. One invocation acts as the master which assigns work to idle machines.
5. Map task:
   a. read the input and parse the key/value pairs.
   b. pass each pair to user-defined Map function
   c. write intermediate key-value pairs to disk in R files partitioned by the partitioning function.
   d. pass location of intermediate files back to master.
6. Master notifies the reduce worker.
7. Reduction is distributed over R tasks which cover different parts of the intermediate key's domain.
8. Reduce task:
   a. read the intermediate key/value pairs.
   b. sort the data by intermediate key (external sort can be used)
   (note: many different keys can map to the same reduce task)
   c. iterate over sorted data and for each unique key, pass the key and set of values to user-defined Reduce function.
   d. output of Reduce is appended to final output for the reduce partition.
9. MR completes when all map and reduce tasks have finished.

MapReduce OUTPUT:

1. The output of MR is R output files (one per reduce task).
2. The partitioning function for intermediate keys can be defined by the user.

By default, it is "hash(key) mod R" to generate well balanced partitions.

3. Result files can be combined or fed to another MR job.

MapReduce Fault tolerance : Worker Failures

1. Master pings every worker periodically (alternatively, the worker can send a heartbeat message periodically)
2. If worker does not respond, master marks it as failed.
3. Map worker:
   a. any completed or in-progress tasks are reset to idle state.
   b. completed tasks need to be re-run since output is stored on a local file system
   c. all reduce workers notified of this failure (to prevent duplication of data)
4. Reduce worker:
   a. any in-progress tasks are reset to idle state.
   b. no need to re-run completed tasks since output stored in global file system.

Fault tolerance : Master Failure

1. Master periodically checkpoints its data structures.
2. On failure, new master can be elected using some leader election algorithm.
3. Theoretically, the new master can start off from this checkpoint.
4. Implementation: MR job is aborted if the master fails.

Fault tolerance : Network Failure

1. Smart replication of input data by underlying file system.
2. Workers unreachable due to network failures are marked as failed since its hard to distinguish this case from worker failure.
3. Network partitions can slow down the entire computation and may need a lot of work to be redone.

Fault tolerance : File System/Disk Failure

1. Depend on the filesystem replication for reliability.
2. Each data block is replicated f number of times. (Default : 3)

Fault tolerance: Malformed Input

1. Malformed input records could cause the map task to crash.
2. Usual course of action: fix the input.
3. But what if this happens at the end of a long-running computation?
4. Acceptable to skip some records (sometimes)
   a. Word count over very large data set.
5. MR library detects bad records which cause crashes deterministically.

Fault tolerance: Bugs in User Code

1. Bugs in user provided Map and Reduce functions could cause crashes on particular records.
2. This case similar to the failure due to malformed input.

Task Granularity:

1. M map tasks and R reduce tasks.
2. M and R much larger than the number of machines.
   a. Improves dynamic load balancing (add/remove machines)
   b. Speeds up recovery
      i. less work needs to be redone
      ii. I work already completed by a failed task can be distributed across multiple idle workers.

c. Bounds:
   1. Master makes O(M+R) scheduling decisions
   2. Master maintains O(M*R) state in memory.
   3. M is chosen such that each task works on one block of data.
   4. R is usually constrained by users to reduce the number of output files.

Requirements of applications using MapReduce

1. Specify the Job configuration
   a. Specify input/output locations
   b. Supply map and reduce functions via implementations of appropriate interfaces and/or abstract classes.
2. Job client then submits the job (jar/executablesetc) and the configuration to the JobTracker.

What are Hadoop/MapReduce limitations?

1. Cannot control the order in which the maps or reductions are run
2. For maximum parallelism, you need Maps and Reduces to not depend on data generated in the same MapReduce job (i.e. stateless)
3. A database with an index will always be faster than a MapReduce job on unindexed data.
4. Reduce operations do not take place until all Maps are complete (or have failed then been skipped)
5. General assumption that the output of Reduce is smaller than the input to Map; large data sourceused to generate smaller final values.

III. Conclusion

Traditional data processing and storage approaches are facing many challenges in meeting the continuously increasing computing demands of Big Data. This work focused on MapReduce, one of the key enabling approaches for meeting Big Data demands by means of highly parallel Processing on a large number of commodity nodes.

Issues and challenges MapReduce faces when dealing with Big Data are identified and categorized according to four main Big Data task types: data storage, analytics, online processing, and security and privacy. Moreover, efforts aimed at improving and extending MapReduce to address identified challenges are presented. By identifying MapReduce challenges in Big Data, this paper provides an overview of the field, facilitates better planning of Big Data projects and identifies opportunities for future research.

References RéférencesReferencias

1. H. Yang and S. Fong, "Countering the concept-drift problem in Big Data using iOVFDT," IEEE International Congress on Big Data, 2013.
2. S. Ghemawat, H. Gobioff and S. Leung, "The Google file system," ACM SIGOPS Operating Systems Review, 2003.
3. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun ACM, 51(1), pp. 107-113, 2008.
4. Apache Hadoop, http://hadoop.apache.org
5. Z. Xiao and Y. Xiao, "Achieving accountable MapReduce in cloud computing," Future Generation Computer Systems, 30,pp. 1-13, 2014.
6. W. Zeng, Y. Yang and B. Luo, "Access control for Big Data using data content," IEEE International Conference on Big Data, 2013.
7. C. Parker, "Unexpected challenges in large scale machine learning," Proc. of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, 2012.
8. www.ibm.com/software/data/infosphere/hadoop/mapreduce/
9. hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
10. research.google.com/archive/mapreduce-osdi04.pdf