I.J. Information Technology and Computer Science, **2016, 8, 50-58**
Published Online August 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2016.08.06

Analyzing Cost Parameters Affecting Map Reduce Application Performance

N.K. Seera
Research Scholar, Banasthali Vidyapeeth, Jaipur, INDIA
E-mail: narinder.k2010@gmail.com

S. Taruna
Associate Professor, Banasthali Vidyapeeth, Jaipur, INDIA
E-mail: staruna71@yahoo.com

***Abstract*—**Recently, big data analysis has become an imperative task for many big companies. Map-Reduce, an emerging distributed computing paradigm, is known as a promising architecture for big data analytics on commodity hardware. Map-Reduce, and its open source implementation Hadoop, have been extensively accepted by several companies due to their salient features such as scalability, elasticity, fault-tolerance and flexibility to handle big data. However, these benefits entail a considerable performance sacrifice. The performance of a Map-Reduce application depends on various factors including the size of the input data set, cluster resource settings etc. A clear understanding of the factors that affect Map-Reduce application performance and the cost associated with those factors is required. In this paper, we study different performance parameters and an existing Cost Optimizer that computes the cost of Map-Reduce job execution. The cost based optimizer also considers various configuration parameters available in Hadoop that affect performance of these programs. This paper is an attempt to analyze the Map-Reduce application performance and identifying the key factors affecting the cost and performance of executing Map-Reduce applications.

***Index Terms*—**Map-Reduce, Hadoop, Cost Parameters, Cost-Optimizer.

I. INTRODUCTION

Current studies reveal that due to advancements in technology organizations are now able to gather large amounts of data and efficiently analyze values in them. "Big Data" management is one of the biggest challenges of the digital era. Google's Map-Reduce is one of the most successful parallelization framework that allows the users to write their own code for analytical data processing. Among the various proposed implementations of the Map-Reduce programming model, Hadoop framework is the most widely adopted one. The most attractive features of Hadoop include HDFS (Hadoop Distributed File System) and resource management layer.

Map-Reduce framework gains its popularity due to the powerful features it offers such as flexibility to write application code, scalability, fault-tolerance etc. But despite of its advantages, it also suffers from severe criticism due to its limitations and performance drawbacks. For example, it allows a program to scale to process very large data sets, but it puts a restriction on the program to process smaller data items.

There exist a wide range of studies reporting the shortcomings of Map Reduce model. Some of the features that contribute negatively in its performance are frequent data materialization, the lack of support for iterations and state transfer between jobs, no index and schema support, dependency on Hadoop's configuration parameters etc.

Reducing the execution time of Map-Reduce jobs is very important to make it attractive to a wide class of analytical applications. For the above reasons, in this paper we study the Map-Reduce framework in great detail and associated performance and cost parameters affecting the execution of Map-Reduce jobs. Through in-depth analysis, we conclude that the cost associated with the sub phases of map-reduce model greatly impacts the performance of map-reduce job under execution. The extent of our paper is limited to studying the parameters that affect the cost and performance of executing map-reduce job and does not include the impact of these factors on map-reduce applications based on column oriented storage such as MongoDB. The contributions of this paper are:

- Studying an existing Cost Optimizer
- Analyzing the effect of modeling the Map-Reduce sub-phases
- Studying Hadoop logs and performance factors

The paper is organized as follows: Section 2 gives the background of map-reduce framework along with its advantages. Section 3 briefs the research work performed by various researchers to improve the performance of map- reduce programs. In Section 4, we elaborate the map-reduce sub-phase and the cost associated with these sub-phases. Section 5 and 6 discuss how to analyze Hadoop logs for map-reduce programs and the related performance factors. At the end we conclude the paper in

Section 7.

II. BACKGROUND

Hadoop is an open-source Java implementation of Map-Reduce framework. Users can opt to run Hadoop either on a virtual cluster in the cloud environment or on Linux configured machines. Hadoop architecture is divided among two main layers: **HDFS (Hadoop Distributed File System) layer** for data storage and a **Map-Reduce layer** for data processing.

- **HDFS** is a distributed block-structured file system which has multiple data-nodes and a single name-node. Data-nodes contain the actual data (or blocks of data) whereas Name-node contains the metadata of the data stored on Data-nodes.
- **Map-Reduce framework** is managed by a single master and multiple worker nodes. Master node (or *JobTracker)* has the responsibility of creating and allocating the tasks among Worker nodes *(orTaskTrackers)*. When the input file is loaded on HDFS, it is first partitioned into fixed size data blocks also called chunks, which are generally 64MB in size and then these data blocks are assigned to different mappers and reducers by JobTracker.

The main advantage of using Map-Reduce model is that it provides a simple programming interface for writing analytical applications with high fault-tolerance guarantee. It is comprised of two user-defined functions – map() and reduce() – both of which work on (key, value) pairs. The map() function accepts a list of (key1, value1) pairs as input, processes them and produces intermediate results. These results are further passed to reduce() function that processes them to produce aggregated results in form of (key2,value2) pairs [9].

The execution of Map-Reduce framework is based on runtime scheduling algorithm where no execution plan is create in advance to specify what tasks will go on which nodes. The number of Map tasks to be scheduled for a particular job depends on the number of data blocks in the input file and not on the number of nodes available. Moreover, all map tasks need not to be executed concurrently. For instance, if an input is broken down into N number of blocks and there are M mappers available in a cluster, then number of map tasks are N and these tasks are executed N/M times by mappers.

III. RELATED WORK

Various implementations of Hadoop Map-Reduce have been developed in past few years that propose improvement gains in performance, programming model extension and automation of use and tuning. Few examples include Hadoop++, Llama, Cheetah, SHadoop, HAIL. Below we brief some of the studies that worked upon various factors that affect the performance of Map-Reduce application.

Wottrich *et al* [1] identified five essential features that affect the performance of Map-Reduce applications. They conducted five separate experiments, each to identify the effect of a single factor on the performance. The results describe a tractable model of Map-Reduce application performance and the initial steps of benchmarking the key factors affecting that performance. Their study illustrates that the size of input data set for a given Map-Reduce application has a linear effect on total run time of the application, where the required run time for an application increased at a rate of 13 sec/GB of data. The application run time can be improved by increasing either the number of Map Tasks or the number of Reduce Tasks up to a limit of 512. Number of reducers beyond this limit causes an adverse effect on application run time. This implies that to achieve good performance results only an optimal number of Map and Reduce Tasks should be launched for a given Map-Reduce application running on a specific cluster.

Hadoop has approximately 190 configuration parameters which can be set to optimize the cost of map-reduce applications. Of these 190 parameters, 10-20 parameters cause significant impact on the application performance. It is the job of the user who executes the Map-Reduce program to specify settings for all those configuration parameters. S. Babu [2] developed techniques to automate the settings of performance parameters for Map-Reduce applications. The automation assists users to only focus on the execution of Map-Reduce application without even knowing the effect of various parameters on the application performance. Hence this feature improves the productivity of users who do not have the expertise to optimize their programs due to the lack of familiarity with Map-Reduce architecture.

Herodotou [3] developed a self-tuning system, Starfish for big data processing. It includes a Cost-based Optimizer that automatically identifies configuration settings for Map-Reduce programs. The Optimizer employs two other components: a Profiler and a what-if analysis engine. The profiler generates the detailed statistical information of Map-Reduce job execution including logs, counters, resource utilization metrics, and profiling data. The user can also get information of how many tasks were running at any given time on each node, when each task started and ended etc. The user can alter the cluster and input specifications for the same Map-Reduce program executed over different input datasets and different clusters.

Herodotou *et al* [4] modeled a 'what-if Engine' which is used for cost estimation. It predicts the performance of Map-Reduce job by considering the job profile generated by the profiler, configuration settings, input dataset and cluster resource properties.

Herodotou [5] published a technical report that describes a detailed set of mathematical performance models for describing the execution of a Map-Reduce job on Hadoop. The model is used to identify the optimal configuration settings and the performance of Map-

Reduce jobs. The performance estimation of an arbitrary Map-Reduce job is done by accurately modeling all the sub-phases Map-Reduce tasks. A map task is modeled by modeling Read, Map, Collect, Spill and Merge sub-phases. Similarly, a reduce task is modeled by modeling Shuffle, Merge, Reduce and Write sub-phases.

In this model, the execution of a Map-Reduce job is represented using a job profile, which is a concise statistical summary of a Map-Reduce job execution. A job profile consists of dataflow fields and cost fields for a Map-Reduce job j - dataflow fields give information about the amount of data flowing through the different sub-phases of Map-Reduce whereas cost fields give information about the execution time of various phases and resource usage.

Rong Hu *et al* [6] worked upon Map-Reduce programming model to increase its performance by optimizing the job and task execution mechanism. The authors proposed two approaches to optimize Map Reduce job and task execution. In the first approach, they implemented setup and cleanup tasks for a Map Reduce job to reduce the time taken by the initialization and termination stages of the job. In the second approach, they implemented an instant messaging communication mechanism for accelerating performance-sensitive task execution rather than transmitting all messages between the Job Tracker and Task Trackers. These two approaches have been successfully implemented in SHadoop, an optimized and fully compatible version of Hadoop that aims at reducing the cost of executing Map Reduce jobs.

IV. MAP-REDUCE PROGRAMMING MODEL

Map-Reduce programming model is known for processing large sets of data in parallel fashion. The model is based on four basic steps:

- Iterating the input
- Computing key-value pairs
- Grouping intermediate results with same keys
- Iterating and reducing intermediate results to produce final output.

Though the model is simple with two phases – map phase and reduce phase; it may have many sub phases that depends on the requirements and input supplied. Here we discuss the phases in detail.

A. MR Sub-phases and Modeling

The map-reduce phases of MR programming model are actually implemented in various sub-phases, as described below:

The map phase goes through five stages:

- Input – Reading the blocks from HDFS and converting them into key-value pairs (k1,v1).
- Map – Running map() task to produce intermediate results in form of key-value pairs (k2,v2).
- Partition – The intermediate key-value pairs are partitioned by the Partitioner. The key (or a subset of the key) can be used to get the partitions, usually by a *hash function*. The total number of partitions is equal to the number of reduce tasks assigned for the job. Therefore this controls which intermediate key should be forwarded to which of the reduce tasks, for reduction.
- Spill – Sorting and performing compression if required, followed by writing to local disk so as to create file spills.
- Merge – Merging the file spills into a single map output file. This merging may be done in several rounds.

Similarly, the reduce phase is carried out in three stages:

- Shuffle –The sorted output produced by the mappers is passed as input to the Reducers. In this phase, the framework fetches the appropriate partition from the mappers, via HTTP.
- Merge – In this stage, the inputs to Reducers are grouped by keys because different map functions may have produced the same key. The shuffle and sort phases are carried out in parallel; while map-outputs being fetched are merged.
- Reduce – Executing the reduce() function to produce the final output data which is then written to the output file.

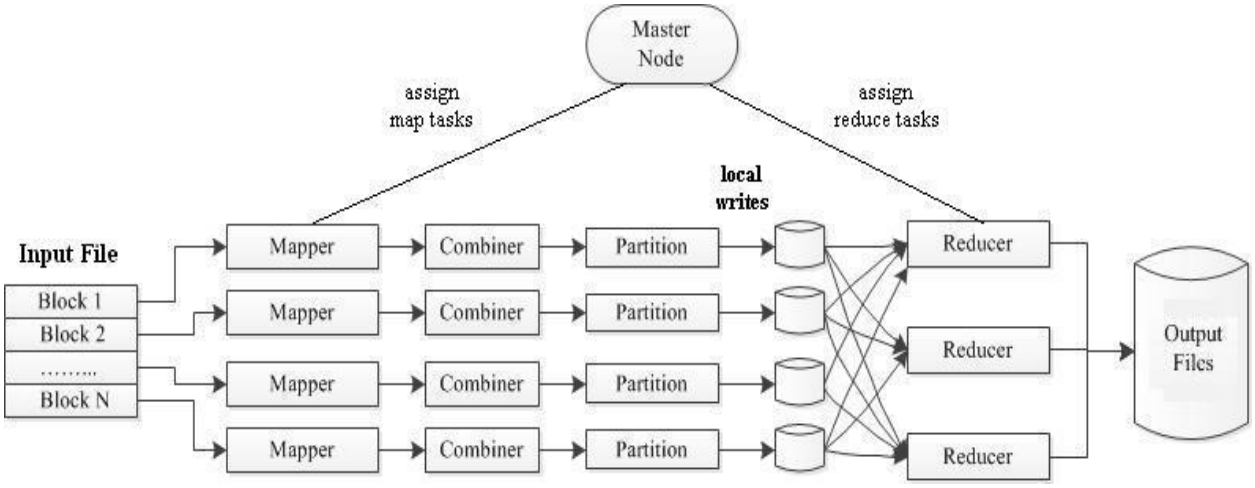

Fig.1. Phases of Map-Reduce Programming Model

Herodotou [5] analyzed all these sub phases of Map-Reduce in order to estimate the accurate statistics of the execution time of a map-reduce job. The overall cost of a Map-Reduce job is can be represented as the sum of the costs of map and reduce tasks, as given below:

$$\text{TotalJobTime} = \begin{cases} \text{TotalMapsTime} & \text{if NumReducers} = 0 \\ \text{TotalMapsTime} + \text{TotalReducesTime} & \text{if NumReducers} > 0 \end{cases}$$

Where,

$$\begin{aligned} TotalMapTimes = {} & \text{ReadPhaseTime} \\ & + \text{MapPhaseTime} \\ & + \text{CollectPhaseTime} \\ & + \text{SpillPhaseTime} \\ & + \text{MergePhaseTime} \end{aligned}$$

And

$$\begin{aligned} \text{TotalReducesTime} = {} & \text{ShufflePhaseTime} \\ & + \text{MergePhaseTime} \\ & + \text{reducePhaseTime} \\ & + \text{SpillPhaseTime} \end{aligned}$$

All the above mentioned parameters are actually the cost parameters that capture the information about the time spend in the execution of each individual sub-phase of a Map-Reduce job.

The number of Reducers (NumReducers) can be set manually by setting the value of *mapred.reduce.tasks*.

B. Job configuration

A job is the main interface for a user to set the configuration settings for the execution of a Map-Reduce program on the Hadoop framework. The framework executes the map-reduce program with the given configuration settings but some configuration parameters cannot be altered by the user as they are marked as final by the administrator. While some parameters can be set directly by changing their default values, as listed in the table below:

Table 1. Map-Reduce Parameter settings in Hadoop with description

Name	Description
mapred.tasktracker.map.tasks.max	Max maps per node in a cluster. Default is 2
mapred.tasktracker.map.tasks.max	Max reducers per node in a cluster. Default is 2
mapred.map.tasks	Number of mappers
min.num.spills.for.combine	Number of spills for combiner function
mapred.reduce.tasks	Number of reducers
mapred.compress.map.output	Whether output of map is compressed. Default is false
mapred.output.compress	Whether the output is compressed. Default is false
mapreduce.map.input.file	Name of the file - map is reading from
mapreduce.map.input.start	The offset that marks the beginning of the map input split
mapreduce.map.input.length	No. of bytes in the map input split
mapreduce.task.output.dir	Temporary output directory for the tasks
mapred.split.size	The size of the input split

To set or get configuration parameters needed by an application one can use Configuration.set(String, String) or Configuration.get(String) methods that take one or two String type values.

Job Configuration settings typically specify the number of Mappers, Reducers, combiners (if any), Partitioners, InputFormat, OutputFormat

implementations etc. FileInputFormat and FileOutputFormat specify where the input and output files should be written. Users can use FileInputFormat.setInputPath(Path) and FileOutputFormat.setOutputPath(Path) methods to set the location for the Input and Output files.

C. Map-Reduce job Execution

The case study is based on the data regarding the number of electricity units consumed by of an organization. The table given below contains the monthly consumption of electricity units and the annual average for five consecutive years. The objective is to find out the year of maximum usage of electricity, year of minimum usage, and year of average usage.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Avg
1979	23	23	2	43	24	25	26	26	26	26	25	26	25
1980	26	27	28	28	28	30	31	31	31	30	30	30	29
1981	31	32	32	32	33	34	35	36	36	34	34	34	34
1984	39	38	39	39	39	41	42	43	40	39	38	38	40
1985	38	39	39	39	39	41	41	41	00	40	39	39	45

To achieve the goal, an application is required that reads the above input and produce the desired results. When the application is executed, it generates the complete details of the processes that undergo along with the execution time of each phase. We analyze the cost based on the previously discussed cost parameters. Following is the application, consisted of mapper and reducer classes that process our data. The execution results and cost measurement is given at the end of the code.

Example

Mapper class :

```
Public static class ElecMapper extends MapReduceBase
implements
Mapper<LongWritable,Text,Text,IntWritable>
{

Public void map(LongWritable key,Text value,
OutputCollector<Text,IntWritable> output,
Reporter reporter) throws IOException
{
String line =value.toString();
String lasttoken=null;
StringTokenizer s =new StringTokenizer(line,"\t");
String year =s.nextToken();

while(s.hasMoreTokens())
{
lasttoken=s.nextToken();
}

Int avgprice=Integer.parseInt(lasttoken);
output.collect(new Text(year),new IntWritable(avgprice));
}
}
```

Reducer class :

```
Public static class ElecReducer extends MapReduceBase
implements
Reducer<Text,IntWritable,Text,IntWritable>
{

Public void reduce(Text key,Iterator<IntWritable>
values,
OutputCollector<Text,IntWritable> output,Reporter
reporter) throws IOException
{
intmaxavg=30;
intval=Integer.MIN_VALUE;

while(values.hasNext())
{
if((val=values.next().get())>maxavg)
output.collect(key,newIntWritable(val));
}

}
}
```

Main function :

```
Public static void main(String s[]) throws Exception
{
JobConfconf=newJobConf(Eleunits.class);

conf.setJobName("max_eletricityunits");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(ElecMapper.class);
conf.setCombinerClass(ElecReducer.class);
conf.setReducerClass(ElecReducer.class);
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf,newPath(s[0]));
FileOutputFormat.setOutputPath(conf,newPath(s[1]));

JobClient.runJob(conf);
}
```


Upon execution the output will be displayed as shown below, containing the number of input splits, the number of Map tasks, the number of reduce tasks, time taken by map task, time taken by reduce tasks etc.

```
FILE:
No. of bytes read=61
No. of bytes written=279400
No. of read operations=0
No. of large read operations=0
No. of write operations=0
HDFS:
No. of bytes read=546
No. of bytes written=40
No. of read operations=9
No. of large read operations=0
No. of write operations=2


   Launched map tasks=2
   Launched reduce tasks=1
   Data-local map tasks=2
   Total time spent by all maps in
occupied slots (ms)=146137
   Total time spent by all reduces in
occupied slots (ms)=441
   Total time spent by all map tasks
(ms)=14613
   Total time spent by all reduce tasks
(ms)=44120
   Total vcore-seconds taken by all map
tasks=146137
   Total vcore-seconds taken by all reduce
tasks=44120
   Total megabyte-seconds taken by all map
tasks=149644288
   Total megabyte-seconds taken by all
reduce tasks=45178880

MR Framework

   Map input records=5
   Map output records=5
   Map output bytes=45
   Map output materialized bytes=67
   Input split bytes=208
   Combine input records=5
   Combine output records=5
   Reduce input groups=5
   Reduce shuffle bytes=6
   Reduce input records=5
   Reduce output records=5
   Spilled Records=10
   Shuffled Maps =2
   Failed Shuffles=0
   Merged Map outputs=2
   GC time elapsed (ms)=948
   CPU time spent (ms)=5160
   Physical memory (bytes)
snapshot=47749120
   Virtual memory (bytes)
snapshot=2899349504
   Total committed heap usage
(bytes)=277684224
```

V. Examining Hadoop Logs

Hadoop maintains various log files on behalf of the execution of a Map-Reduce program. These files are located in /**hadoop/logs sub**-directory. One can examine all these log information to gain better understanding of the execution performance of map-reduce program. To access the logs through command line explore the ***logs*** sub-directory.

The log file contains various lines of information:

- Lines beginning with "**Job**", list information about the job such as job id, launch time, number of map tasks, number of reduce tasks and job status.

```
Job JOBID="job_201004011119_0025"
LAUNCH_TIME="1270509980407"
TOTAL_MAPS="12" TOTAL_REDUCES="1"
JOB_STATUS="PREP"
```

- Lines beginning with "**Task**" indicate the start and completion time of Map or Reduce tasks, also indicating on which host the tasks were scheduled and on which split (input data) they worked up on. On completion, all the counters associated with the tasks are listed.

```
Task
TASKID="task_201004011119_0025_m_00000
3" TASK_TYPE="MAP"
START_TIME="1270509982711"
\SPLITS="/default-
rack/hadoop6,/default-rack/hadoop4"

Task
TASKID="task_201004011119_0025_m_00000
3" TASK_TYPE="MAP"
TASK_STATUS="SUCCESS" \
FINISH_TIME="1270510023272" \

COUNTERS="{(org\.apache\.hadoop\.mapre
d\.Task$FileSystemCounter)(File
Systems) \
[(HDFS_READ)(HDFS bytes
read)(67112961)][(LOCAL_READ)(Local
bytes read)(58694725)] \
(LOCAL_WRITE)(Local bytes
written)(72508773)]}{(org\.myorg\.Word
Count$MyCounters) \
                    ...
[(MAP_INPUT_BYTES)(Map input
bytes)(67000104)][(COMBINE_INPUT_RECOR
DS) \
(Combine input
records)(11747762)][MAP_OUTPUT_RECORDS
)(Map output records)(9852006)]}"
```

- Lines beginning with "**MapAttempt**", gives status

update, except if they contain the keyword FINISHTIME, indicating that the task has completed successfully.

- Lines beginning with "**ReduceAttempt**", gives the intermediary status of the reduce tasks including the finish time of the sort and shuffle phases etc.

```
ReduceAttempt TASK_TYPE="REDUCE"
TASKID="task_201004011119_0025_r_00000
0"
\TASK_ATTEMPT_ID="attempt_201004011119
_0025_r_000000_0"
TASK_STATUS="SUCCESS" \
SHUFFLE_FINISHED="1270510076804"
SORT_FINISHED="1270510082505"
FINISH_TIME="1270510093979"
\HOSTNAME="/default-rack/hadoop4"
STATE_STRING="reduce > reduce" \

COUNTERS="{(org\.apache\.hadoop\.mapre
d\.Task$FileSystemCounter)(File
Systems) \
                          ...
(4416230)][(REDUCE_INPUT_RECORDS)(Redu
ce input records)(6888474)]}" .
```

VI. Performance Factors

There are various factors that may significantly influence the performance of map-reduce applications – the factors may depend on cluster resource settings, configuration settings of the machines, properties of map-reduce application etc. The use of partition and combine sub-phases of MR model greatly impacts the application performance. Below we discuss first how a job is carried out by these sub-phases and then we discuss them in detail.

To reduce I/O operations and network traffic, one can define a combiner function to perform map-side pre-aggregations. It is implemented before partitioning phase in order to perform pre-aggregation on the grouped key-value pairs so that the communication cost to transfer all the intermediate outputs to reducers can be minimized. The Map-Reduce framework ensures the number of times combiner function needs to be run – once or multiple times. Running the combiner ( ) function results in significant performance gains by lessening the amount of intermediate data to be transferred over the network.

The intermediate outputs are then partitioned into R partitions using a hash function such as ***hash (key) mod R,*** where R is the number of reduce tasks. Each partition is then written to the mappers local disk.

After the map stage is over, all the partitions with the same hash value are read by the same reducer, regardless of which mapper produced which partition. These partitions are grouped together using merge –sort and are written to the output file to be used by the reducer. This is depicted in figure 2.

- Partitioner - Hadoop uses *Partitioner* interface to identify which partition will receive which intermediate key/value pair. The most important point is that for any key, regardless of its Mapper instance, the destination partition is the same. The Map-Reduce architecture determines the number of partitions to be used when any Map-Reduce program begins, which is usually the same as the number of reduce tasks. For performance reasons, Mappers never communicate with each other to the partition of a particular key.

  Partitioners are actually implemented in Java, and may take two forms:

1. Default partitioning – that randomly distributes all the keys evenly. Hashing is used as default partitioning. For example :key.hashCode() % no_of_reducers
2. Custom partitioning – is required when an order in the output needs to be produced. It is done by implementing *Partitioner* interface, the signature is given below:

   public interface Partitioner<K2,V2> extendsJobConfigurable { .... }

   // <K2,V2> is a key-value pair supplied as an argument to the partition interface

   The Partition can be set by:
   job.setPartitionerClass(LogPartitioner.class);

   To get an ordering in the output, the map output keys can be divided into roughly equal buckets and used in partitioner.

- Combiners – As the outputs from the mappers may be large in size, so to limit the volume of data transferred between mappers and reducers, combiners are implemented. Combiners summarize the map outputs with the same key and forward the results to reducers. Each combiner works in isolation and thus it does not have any access to intermediate outputs from other mappers. The combiner acts as an optimizer, and it is never sure of how many times it will be called for any particular map output record. But regardless of the number of times it is called it should produce the same output every time for the reducer.

  Combiners have the same interface as Reducers, and can be set by:

  job.setCombinerClass(LogReducer.class);

  The difference between a partitioner and a combiner is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, the combiner functions similar

to the reducer and processes the data in each partition.

- Compression – In order to achieve huge performance gain it is worth to compress the input blocks, intermediate data and outputs produced by map phase. Compressing the data entails two major advantages – it reduces the storage space required to store it and it increases the speed of data transfer across the network (or across the nodes in a cluster). To compress the map outputs or final job output, the corresponding parameters can be set either by altering configuration files or programmatically, as given in Table 1. There are two types of compression methods for sequential files which can be set via *mapred.output.compression.type*.

1. Block-level compression – It compresses a group of key-value pairs together.
2. Record-level compression – It compresses each key-value pair individually

To set BLOCK type compression, use:

```
conf.set("mapred.output.compression.type", "BLOCK")
```

The most widely used compression formats are gzip (default in Hadoop), bzip2, LZO and snappy. A detail discussion on these formats is beyond the scope of this paper.

- Speculative Execution - The MR model can execute multiple instances of slower tasks by using the output from the instances that finish first. This can be done by setting the following configuration variable:

mapred.speculative.execution

It can significantly get in long tails on tasks.

VII. Conclusion and Future Scope

As we know, Hadoop and Map-Reduce offer several advantages such as scalability, elasticity, fault-tolerance and flexibility to handle big data - these benefits entail a significant performance sacrifice. The performance of a Map-Reduce application not only depends on the size of the input data set but also on several parameters that affect its performance and other configuration settings of the Hadoop installed machine. This paper is focused on analyzing the cost parameters for executing Map-Reduce programs using a case study. The cost based optimizer considered in the paper uses various factors and configuration parameters to examine the cost of map-reduce programs.

The case study used a simple text file as the input data set and the program was run on a single node cluster. We are trying to extend our analysis to column oriented stores where the columnar data will be used as the input data set. The organization of data and data structures used for column oriented storage such as the use of index files, compression techniques etc also somehow affect the performance of map-reduce applications. The performance of map-reduce applications along with columnar data layouts is yet to be explored.

References

[1] K. Wottrich, Thomas Bressoud, “The Performance Characteristics of Map-Reduce applications on scalable clusters”, MCURCSM 2011.

[2] S. Babu, “Towards Automatic Optimization of MapReduce Programs”, in SOCC, pages 137-142, 2010H.

[3] Herodotou et. Al, Starfish: A Self Tuning System for Big Data Analytics, 5th Biennial Conference on Innovative Data Systems Research (CIDR ’11) January 912, 2011, Asilomar, California, USA.

[4] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. PVLDB, 4, 2011.

[5] Horodotou, “Hadoop Performance Model”, Technical Report, CS-2011-05, CS Department, Duke University

[6] Rong Gu et al, “SHadoop: Improving Map Reduce performance by optimizing job execution mechanism is Hadoop cluster”, Journal of Parallel and Distributed Computing, Elsevier, Vol 74 Issue 3, March 2014, pg 2166-2179.

[7] Chang, M. Kodialam, R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee, “Scheduling in mapreduce-like systems for fast completion time,” in Proc. IEEE INFOCOM’11, Shanghai, China, 2011.

[8] H. Herodotou, F. Dong, S. Babu, MapReduce Programming and Cost based Optimization? Crossing this Chasm with Starfish, Proceedings of the VLDB Endowment, 21508097/11/08, Vol. 4, No. 12, 2011

[9] Narinder, S. Taruna, “Efficient data layouts for cost optimized Map-Reduce operations”, Proceedings of INDIACom 2015, BVICAM, Delhi.

[10] Arun C Murthy, ”Programming Hadoop Map-Reduce”, Yahoo CCD, ApacheCon US 2008.

[11] D. Borthakur, “The Hadoop Distributed File System: Architecture and design”, Apache Software Foundation, 2007.

[12] K. Lee, Y. LeeH. Choi, Y. Chung, B. Moon, “Parallel data processing with Map Reduce: A Survey”, SIGMOD Record, December 2011 (Vol. 40, No. 4).

[13] Dittrich, Jens, J. Arnulfo, "Efficient big data processing in Hadoop MapReduce." Proceedings of the VLDB Endowment 5.12 (2012): 2014-2015.

[14] J. Tan, S. Meng, X. Meng and Li Zhang, “Improving ReduceTask Data Locality for Sequential MapReduce Jobs”, in Proc. IEEE INFOCOM’13, Turin, Italy, 2013.

[15] H. Chang, M. Kodialam, R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee, “Scheduling in mapreduce-like systems for fast completion time,” in Proc. IEEE INFOCOM’11, Shanghai, China, 2011.

[16] C. Doulkeridis, K. Norvag, “A Survey of Large Analytical Query Processing in Map-Reduce”, the VLDB Journal.

[17] A. Floratou et al, “Column-Oriented Storage Techniques for Map-Reduce”, In proceedings of VLDB Endowment, Vol 4, No. 7, 2011.

Authors' Profiles

N.K. Seera is a Research Scholar in Banasthali Vidyapeeth, Jaipur, India. She is carrying out her research work in the field of Data Processing in Map-Reduce. Her interest areas include databases, Query Processing and Big Data Processing. Her research papers have been published in various Journals as well as National and International Conferences sponsored by IEEE and other bodies.

Dr. S. Taruna is working as an Associate Professor in Banasthali Vidyapeeth, Jaipur in the department of Computer Science. She has done M.Sc (CS) and carried out her Ph.D in the field of Data Mining. Her areas of Interest are Data Mining, Data Processing etc.

How to cite this paper: N.K. Seera, S. Taruna, "Analyzing Cost Parameters Affecting Map Reduce Application Performance", International Journal of Information Technology and Computer Science (IJITCS), Vol.8, No.8, pp.50-58, 2016. DOI: 10.5815/ijitcs.2016.08.06