Big Data Analysis using MapReduce

Saurabh Kumar
Indiana University
Bloomington, Indiana
kumarsau@iu.edu

ABSTRACT

With the amount of data created everyday reaching exorbitant limits and a push for analysis across all major fields gaining traction, we need an advanced framework to analyze the large data. Hadoop is such a framework for developing distribution applications. MapReduce is an application model for Hadoop and other distributed file systems. Word count is a small analysis that can be done using MapReduce. It can also be used for complex tasks like implementing search functionality. The main implementation of MapReduce is processing and generating large data sets. A key value pair is processed by the user created map function, which produces intermediate key value pair. The reduce function merges all intermediate values for the same key. Codes written in this programming model are automatically parallelized and executed on a cluster of machines.

KEYWORDS

MapReduce, distributed, fault tolerance, master, big data

1 INTRODUCTION

Many MapReduce models are developed for handling big data. Google distributed file system is one of them. Hadoop is the most popular open source framework for handling big data. Google's MapReduce framework has allowed the search for million of pages and delivering the results in milliseconds. The input data to be processed is large and the computation times have to be lowered. Therefore the data and the computations have to be distributed across thousands of machines to achieve the goal of low computation time while handling large data. The main problems that such a programming model should handle are data distribution, handling failures, parallelizing computation and fast execution. MapReduce handles all these complexities by providing an abstraction which in turn is done by hiding the details of data distribution, error handling, parallelization and load balancing.

The map and reduce functions help to parallelize the computations and re-execution is used for fault tolerance. The map function is applied to every record in input. This creates an intermediate key value pair to which a reduce function is applied. All the intermediate values with same key go to the same reducer. The number of mapper depends on the number of distributed file system blocks and the amount of data. Usually there are 10 to 100 mappers for each data node. The execution time on each machine should not be very high or low, as the task setup takes a while. The number reducer depends on the requirement of final data output and the number of unique keys in the intermediate data. The number of reducers should not be too high as it would slow down the computation by generating stragglers. We will see this later. The number should also not be too low so that parallelization cannot take its effect.

2 MODEL

The computation is divided as two functions: Map and Reduce. The user defined Map function produces an intermediate key value pair. These values are grouped together according to their keys by the MapReduce library and then passed to the Reduce function. The user defined Reduce function receives an intermediate key and list of all the values associated with that key. These values are merged together by the reduce function and the values are supplied to the function in different iterations. So that when there are large list of values, the memory does not get full.

To better understand this procedure we take the example of word frequency in a document. The Map function produces an intermediate key value pair of a word and its respective count of occurrences. The reduce function then groups by each word and sum up all the counts of occurrences. The final result is every word and its respective frequency in the document. This procedure can also be shown by the help of the two equations below:

map (k1,v1) -> list(k2,v2)
reduce (k2,list(v2)) -> sum(v2)

There can other applications of MapReduce. For a URL access frequency count problem, the map function produces web page requests and the respective output as the key value pair. The reduce function sums up all the counts for a specific page and returns a final output of the pair(URL, total count).

In case of a reverse web link graph example, the map function produces a target and source pair for each link of a target URL in a source page. The reduce function produces the same key, target, and the value as a concatenated list of all the sources that belong to the same target. The output pair is (target, list(source)).

Another example can be inverted index. The map function parses each document and produces a word as a key and the document id as the value. The reduce function takes this as an input and outputs a sorted list of document for every word.

3 IMPLEMENTATION

The type of implementation of MapReduce depends on the environment. The implementation is different from a small shared memory machine to a large cluster of machines. A cluster consists of hundreds of machines. When a job is submitted to a scheduler, it assigns each task in the job to an available machine in the cluster.

3.1 Overview

Different machines in the cluster act as a mapper ie implement the map function. The data in partitioned into sets of M splits, where M is the number of mappers. Each split of data is fed to an individual

mapper and thus all splits can be processed in parallel. There are few machines that implement the reduce function. They are called reducers. The intermediate key space is split into R partitions, where R is the number of reducers. The number of partitions is provided by the user.

The user calls the MapReduce function. The MapReduce library splits the data into M partitions, where each partition size ranges from 16 megabytes to 64 megabytes. This size can be specified by the user through a parameter. Then the library starts many copies of the program on the cluster of machines. One copy of the program acts as a master. The rest are workers or slaves which assigned tasks by the master.

The worker who is assigned a map function is a mapper. A mapper reads the contents of an input split. The intermediate key value pair, produced by the mapper, is buffered in memory. The buffered data is periodically written to the local disk and partitioned into R splits. Location of this data is passed back to the master. The master in turns passes the location to the workers who are assigned reduce functions.

When a reducer is notified about a location by the master, it uses remote procedure call to read the data from the local disk in the respective mapper. The intermediate data read by the reducer is sorted and groups together data from each key. The reducer iterates over each unique key and passes the set of values to the reduce function. The output is appended to a final output file. After all map and reduce tasks are completed, the master wakes up the user program.

After the completion of this process the final output is available in R files. Each file comes from one reduce task and the names are specified by the user. These files can be combined to a single file or can be passed as input to another MapReduce function.

The master stores some metadata. It has the state information for every map and reduce task. The states are idle, in progress or completed. It also has the location of the buffered files to be read by the reduce tasks.

3.2 Fault Tolerance

Since the MapReduce library is designed to handle large amounts of data and work over a distributed platform, it must tolerate machine failure smoothly.

The master pings every worker after a specific interval. If the worker does not respond to the master then the worker is marked as failed by the master. If a mapper completes its task, then it is reset back to idle state and is available for rescheduling. The same happens when a mapper or reducer has failed. They are reset to idle and available for rescheduling. In case of a failed machine, the completed tasks are re-executed as the output is saved in the local disk, which would have otherwise been incomplete or inaccessible. The output for completed tasks is passed to the global file system ie their locations are passed to the master which handles the data from then on.

When a mapper A has failed and its map task is re-executed by mapper B, then all reducers are notified of this re-execution. Any reducer that has not already read data from mapper A will read the date from mapper B.

Master can write periodic checkpoints of the metadata it maintains. This helps when the master dies, re-execution can be started from the last checkpoint. When there is only one master, the chances that it dies are slim. User can check the state of master and retry the MapReduce operation when the master fails.

While handling large chunks of data, network bandwidth can the scarce resource in the distributed environment. If not handled properly it will bottleneck the entire operation. Therefore input data is stored in the local disks of the machines in the cluster. As we have seen earlier that the entire file is divided into chunks of 16 to 64 megabytes. Several copies of each block is stored in different machines. This information is also provided to the master node, which oversees the entire operation. When the MapReduce operation is in process, the input data is read locally and the network bandwidth is not consumed. A worker performs different tasks and this helps in load balancing and also speeds up the recovery in case of a failure.

4 PERFORMANCE

The total execution time for the MapReduce operation can be increased mainly due to straggler. Straggler is a machine that takes large time to complete a simple map or reduce task. This generally happens during the end of the MapReduce operation, when the reading of input data is slowest. To handle this situation, the master schedules backup executions for the remaining tasks. The task is deemed completed when the primary or backup machine has completed the task.

The arrangement of clusters and the specification of machines used also affect the performance. In the idle state the memory for the machines should be freed for better results. In case of a 1 terabyte file the data is split into 64 megabyte blocks. Therefore there will be 15000 map tasks and reducer can be user decided.

The performance can also be affected by the type of operation we choose. In case of a normal execution where backup tasks are used in case of stragglers, the performance is good. The run time is increased when the backup tasks are not executed. This is when stragglers push the run-time further. In the third scenario when few tasks are killed towards the end the performance is good and similar to the normal execution. In some cases it can be better than the normal execution.

5 CONCLUSION

The use MapReduce programming model has propelled a new phase better usage of Big Data. The processing of such large amount of data usually required high end machines with large memories and top of the line processors. This can now be done using common machine but in a distributed environment. This model has also improved the fault tolerance as well as the run time for processing such large data. Such a model has also helped in usage of large machine learning algorithms, clustering problems, large scale graph computation and extraction of large web content. Google uses this model for large scale indexing by creating inverted index. The computation from MapReduce model is efficient so we need not look into the complex internal workings and can create better abstracted programs. It also provides parallelization, load optimization and

load balancing. Such a model can also be re scaled to thousands of machine, thus making it future ready.

6 REFERENCES

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, May 1997.

[2] Guy E. Blelloch. Scans as primitive parallel operations. IEEE Transactions on Computers, C-38(11), November 1989.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In 19th Symposium on Operating Systems Principles, pages 29ÂŰ43, Lake George, New York, 2003.