Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 6, June 2015, pg.287 – 298

RESEARCH ARTICLE

Batch Inherence of Map Reduce Framework

Jaswender Malik, Ms. Kavita

[1,2]COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

Maharshi Dayanand University, Rohtak Haryana

[1] Jassi15malik@gmail.com

Abstract: Big Data is dealt by every organization which serves large number of users. Efficiently fetching, transferring, storing, cleaning, sanitizing, querying and extracting information from Big Data is a daunting task because a single machine and the traditional algorithms can't handle this staggering amount of data tractable. The open source Map Reduce system Hadoop doesn't provide any API to view the partial results programmatically or manually. In this Research paper we will extend Map Reduce to stop a job early during execution if the partial results meet a certain user specified constraint. This enhancement can save a lot of time for certain kind of batch processing applications prevalent in industry.

Keywords – Big Data Analysis, HDFS, Map Reduce

Introduction

Map Reduce is a batch text processing system which makes the user waits till the end of execution to view the job's output and perform an action based on the same. In certain use cases, user may want to terminate a job long before the finish of execution and start a new job based on partially available result of the current job in the case that the partial result

satisfies a certain user specified constraint. For example, a user while running a Word Count Map Reduce job may want that if a certain word has occurred more than n times, then no further execution is required and the job should be terminated. In current Map Reduce implementations, this is not possible because the user must wait till the end of the execution to see the result.

Batch inherence of Map Reduce framework

Map Reduce [1] framework was modeled with the aim of providing batch execution of data processing jobs on a large cluster of distributed systems. Developing a typical Map Reduce application [1] involves three steps from the user's part –

- Collection of input data to be processed, in form of ASCII text files
- Writing map and reduce functions to transform the input data according to the requirement
- Collecting and analyzing output data, and possibly use it as an input for another MapReduce job

Availability of partial results

Partial results can be made available by making some changes in existing Map Reduce implementation. In current Map Reduce implementations, the Map tasks finish, then the reducers start which apply the reduce function on the Map output. By modifying this implementation in such a way that reducers start early and reduce function is applied on partially available Map output, we can make partial results available. But this requires more careful treatment of fault tolerance. But reducers start which apply the reduce function on the Map output. By modifying this implementation in such a way that reducers start early and reduce function is applied on partially available Map output, we can make partial results available. But this requires more careful treatment of fault tolerance.

Pipelining in Map Reduce

For making partial results available during the execution, the idea is to pipeline the data between mappers and reducers to the maximum possible extent. We start the reducers in parallel with mappers and then let the map tasks send their output to reduce tasks as and when produced. This allows the reducers to apply the reduce function on the map output available to them till a particular instant of time. This has been achieved and described in detail in [2].

Constraint based job termination

This provides the programmer with facility to stop the execution of Map Reduce jobs when certain user specified constraints are met. The conditions can be:

- Accuracy of the estimate reaches within a certain percentage Certain percentage of input data successfully processed
- A key has been encountered certain number of times
- Certain amount of time elapsed since the start of job execution

Pipelining in Map Reduce

For making partial results available during the execution, the idea is to pipeline the data between mappers and reducers to the maximum possible extent. We start the reducers in parallel with mappers and then let the map tasks send their output to reduce tasks as and when produced. This allows the reducers to apply the reduce function on the map output available to them till a particular instant of time. This has been achieved and described in detail in [2].

Constraint based job termination

This provides the programmer with facility to stop the execution of Map Reduce jobs when certain user specified constraints are met. The conditions can be:

Accuracy of the estimate reaches within a certain percentage Certain percentage of input data successfully processed.

A key has been encountered certain number of times

Certain amount of time elapsed since the start of job execution

Hadoop Architecture

The key aims in architecting Hadoop are as follows:

- Easily distribute data and computation across all the nodes in the cluster
- Provide fault tolerance assuming that nodes, network, disks can go down any time during the course of execution

Architectural components

Hadoop comprises of two main architectural components viz. Hadoop Distributed File System [3] i.e. HDFS, and Hadoop Map Reduce Job Execution Framework.

Hadoop Distributed File System (HDFS)

Hadoop Distributed File System is a fault tolerant Distributed File System built for high volume data storage with very high performance for read/write through-puts. It uses replication of data, to provide fault tolerance by failover and high throughput of read operations by parallel reads. It consists of two components - a master called namenode which is responsible for storing the metadata of all the files stored on HDFS, and other nodes called datanode which store the actual data. By default, the data is broken down into 64MB splits/chunks and stored on different datanodes with a replication factor of 3. Replication factor determines the number of copies of each chunk in HDFS When a client application needs to read and write data, it contacts the namenode with the file path and the namenode returns the chunk handles and addresses of the data nodes that contain the chunks/splits of that file. The client can then directly issue file append/read requests to the chunk owning datanode. Details of Implementation of HDFS has been provided in [4].

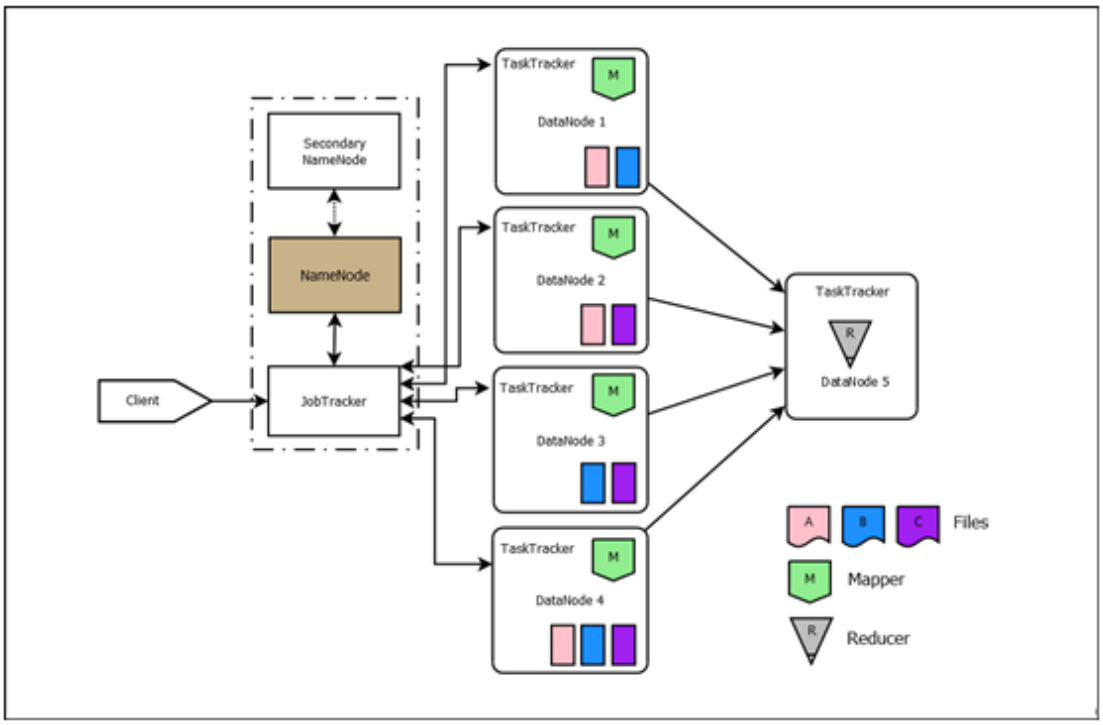

Figure 4.0 : Hadoop cluster

Hadoop Map Reduce Job Execution Framework

Hadoop Map Reduce Job Execution Framework allows a user to submit jobs writ-ten in MapReduce programming model and executes these jobs. It consists of a master called JobTracker which accepts MapReduce jobs from clients and divides them into tasks and assigns them to TaskTrackersas shown in fig 4.0. There are several workers called TaskTrackers which are responsible for the execution of tasks assigned to them.

Simple pipelining

Pipelining between tasks within a job

With an assumption that enough slots are available for every map and every reduce task, we can implement pipelining as follows. Each reduce task opens a TCP socket (RFC793, accessed May 10, 2013) connection to every map task. As soon as a map task produces a record, it determines the target reduce task using the partition function and sends the record to the corresponding reduce task over the communication channel. Reduce tasks keep receiving the records continuously and store them in an in-memory buffer temporarily. When the buffer gets full, the reduce tasks sort the data and persist them to the disk. When every map is finished, each reduce performs a final merge of all its spills.

Pipelining between jobs

Reduce tasks of a job can send their output to map tasks of the next job bypassing the need to write output to HDFS and hence avoiding the overheads.

Fig. 4.1 depicts the data flow in traditional hadoop and our proposed pipelined hadoop. The left part shows the data flow in case of traditional hadoop. Right one shows the data flow in case of pipelined hadoop. The difference in the architecture can be clearly visualized. A map in traditional hadoop pushes data to map TaskTracker's Local FS. While in pipelined hadoop it pushes the data to map TaskTracker's in-memory buffer as well as its local file system in parallel. The reducer then pulls this data synchronously in case of traditional hadoop. In case of pipelined hadoop, the reducers either asynchronously pull the data or the data gets pushed when the destined reduce is assigned a slot.

Partial results

Partial results can be made available by applying user defined reduce function to the set of key value pairs sent to reduce tasks at certain points of time. Each such point of time can be called an snapshot. Estimating the accuracy of result for user defined map and reduce function is very difficult. But, we can supply the user with execution progress reports. The user can estimate the accuracy by interpreting the job progress values. The user can specify the points of time for example 5%, 10%, ... 95%, 100% of the input processed. As the execution of a map task proceeds, it is assigned a progress value by Hadoop in the range [0,1] depending on how much input has been processed by it. We can utilize this already existing facility in Hadoop to determine how much progress should be shown by the reduce task. For this, we can modify the spill file sent by mappers to reducers to include the progress score of the mapper too find the progress status, we can take an average of individual progress scores included in every spill file that was used to produce a particular snapshot.

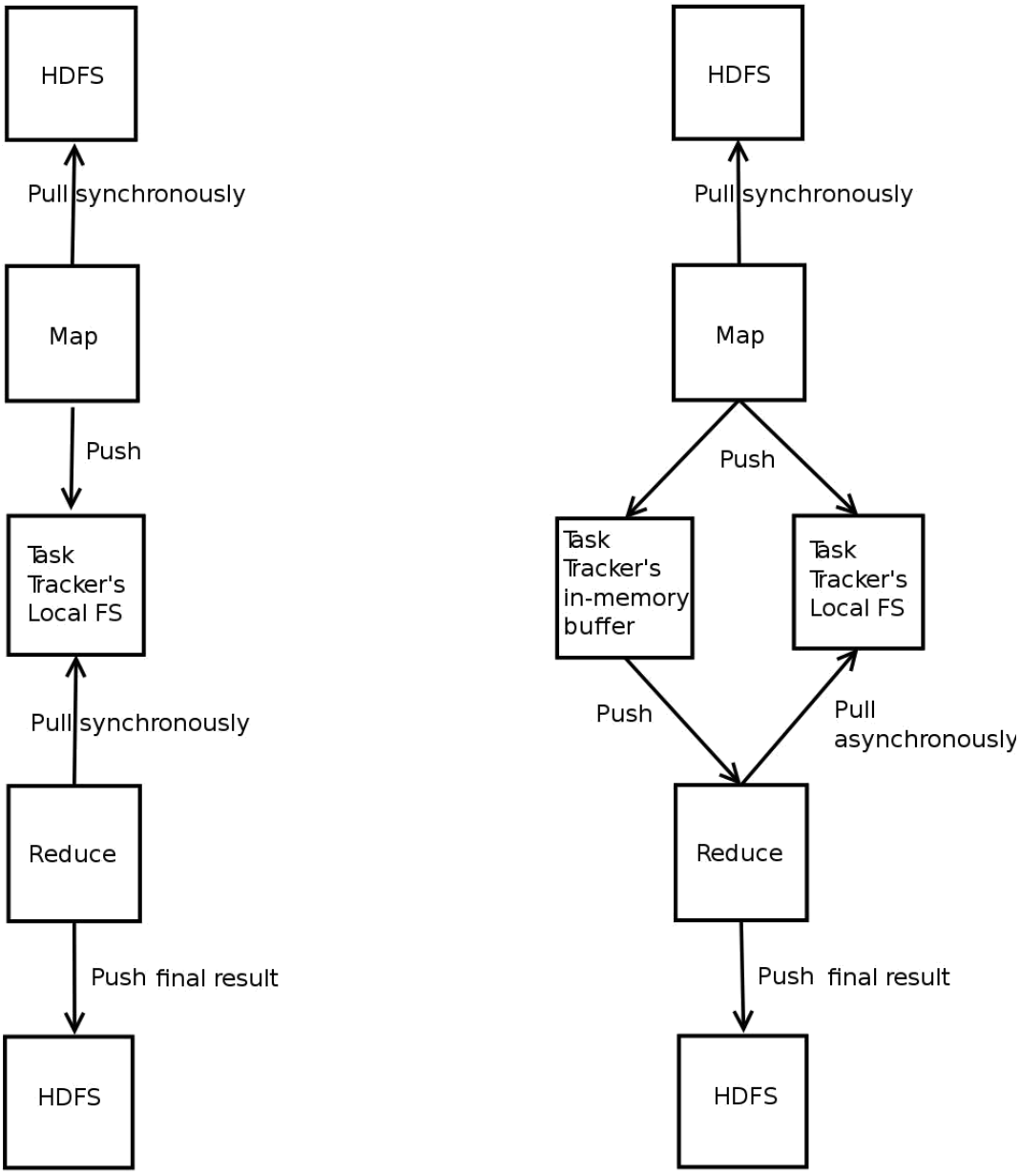

Figure 4.1: Data flow in normal Hadoop vs. pipelined Hadoop

To when a map task didn't send any output to reduce task because it wasn't scheduled due to unavailability of slots or because the reduce task was bound to fetch output from a limited number of mappers, we can normalize our progress score by multiplying by 1=n when the reduce task has received data from n map tasks.

Implementing constraint based job termination

We used the modified version of Hadoop as discussed in Section 4.3, which provides us the feature to access partial results. We provide the user a programming abstraction to specify a constraint, as discussed below.

Programming Model

We have provided a programming model to the user for specifying constraints in the form of a user defined function that takes two parameters - a key and a value and returns true or false. The constraint is in form of a relation between the key and the value. The provided interface is shown in Listing 4.1.

Listing 4.1: Termination Condition interface

```
     *
     * @param key The key Object
     * @param value The value object
     * @return boolean value representing the outcome of the test of relation between
       key
     * and value
     */
    public boolean matchConstraint(K key, V value);
}
```

Based on this interface user can write a Constraint class. An example is in Listing 4.2 where the user specifies a constraint that the key IITM appears more than 100 times in a word count program. The class implements the interface TerminationCondition while specifying the generic parameters as Text and IntWritable. In the class, matchConstraint method has been overridden to represent the constraint.

Listing 4.2: An example termination constraint class

```
/**
 * A constraint terminator class */

public static class TConstraint implements

    TerminationCondition<Text, IntWritable> { @Override
```



```
    public boolean matchConstraint(Text reduceKey, IntWritable reduceValue)
        {


        if(reduceKey.toString().equals("IITM") && reduceValue.get() > 100)


            return true; return
        false;
    }
}
```

Now to add this constraint in the program so that it can be evaluated, the user needs to add a configuration parameter named mapred.job.termination.constraint to the JobConf object. An example is shown in Listing 4.3.

Listing 4.3: A sample JobConf

```
// assuming that conf is a JobConf object
conf.set("mapred.job.termination.constraint",
    TConstraint.class.getName());
```

Then the user should provide a parameter called mapred.snapshot.freq in the JobConf object which determines for how much percentage of input seen, the partial results should be processed. For example a value of 0.01 will make the system process partial results for every 1% of seen input. Full code listing with a working sample of WordCount program is provided in Listing A.1 The user can then compile this program adding our core.jar file in the class path and build a jar. A sample perl script in Listing B.1 performs that.

Implementation

Constraint based termination of MapReduce jobs has been implemented by using the pipelined hadoop implementation from [2] and making some architectural changes discussed in Section 4.3. After the user defined amount of input is seen, re-ducers are forced to apply reduce function on the partial map output they have. At each reducer, the

OutputCollector runs the matchConstraint function on the reduced key and value pair. If the function returns false, the OutputCollector continues to eval-uate matchConstraint for next key and value pair. If the function returns true for any key and value pair, the OutputCollector issues a job termination signal to the JobTracker which terminates the job cleanly.

Result

We conducted experiments on a sample data set of different sizes for different programs on a Hadoop cluster. The results for small dataset are in Fig. 4.2. The x-axis shows the program name that we ran and the y-axis shows the time it took to run. We can see from the graph, for the WordCount program the normal run took 85 seconds and upon adding the constraint the run finished in just 28 seconds. Similar results can be seen for our run of inverted index program and the geo code program.

The result for large dataset is in Fig. 4.3. More experiments have been run in an extension of this work in [2] in which a user can specify multiple constraints at once.

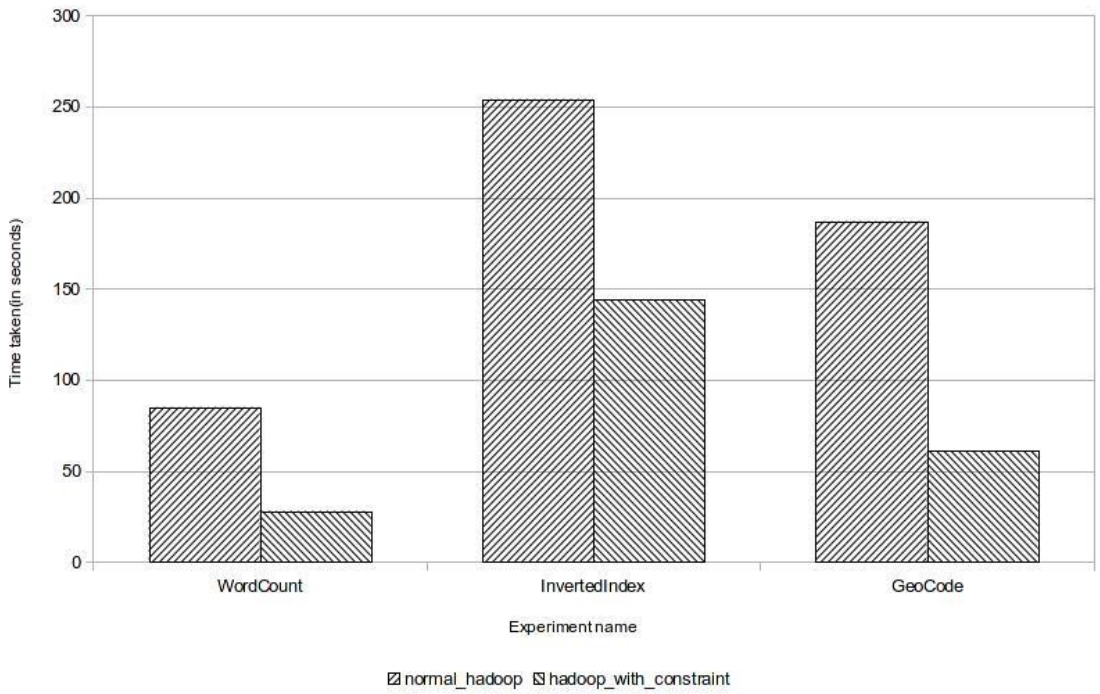

Figure 4.2: Normal hadoop vs. Hadoop with termination constraint for dataset size 3GB

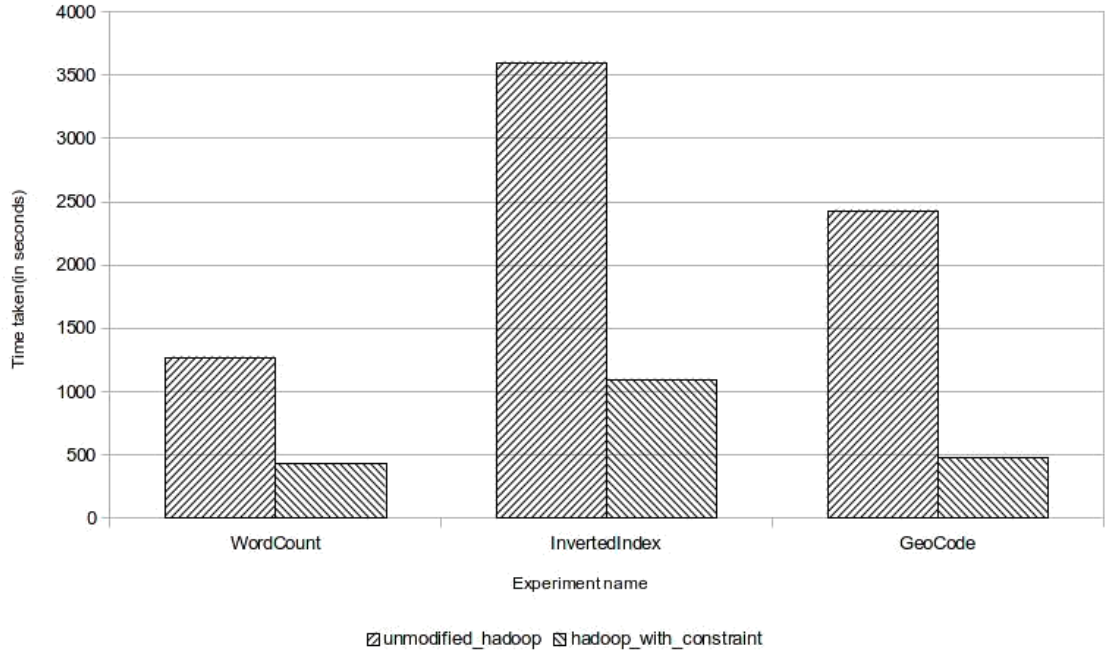

Figure 4.3: Normal hadoop vs. Hadoop with termination constraint for dataset size 40GB

Extensions and future possibilities

A beautiful extension of this work has been done in [2]where the author has implemented this system for multiple constraints connected with each other using a propositional formula. Another possibility is to launch another job after terminating the current job, hence supporting MapReduce job workflow definitions. This will require defining a workflow in terms of jobs and constraints

References

[1] Dean, J. and S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1), 107–113. ISSN 0001-0782. URL http://doi. acm.org/10.1145/1327452.1327492.

[2] Agarwal, S. (2013). Efficient Processing of industry scale data using NoSQL and MapReduce. Master's thesis, Indian Institute of Technology Madras, Chennai, India.

[3] Shvachko, K., H. Kuang, S. Radia, and R. Chansler, The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-1-4244-7152-2. URL http://dx.doi.org/10.1109/MSST. 2010.5496972.

[4] Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). The google file system. SIGOPS Oper. Syst. Rev., 37(5), 29–43. ISSN 0163-5980. URL http://doi.acm.org/ 10.1145/1165389.945450.

