Survey Paper on Traditional Hadoop and Pipelined Map Reduce

Dhole Poonam B[1], Gunjal Baisa L[2]
[1]M.E.ComputerAVCOE, Sangamner, India
[2]Dept. of Computer Engg.AVCOE, Sangamner, India

ABSTRACT

Recent days Map Reduce programming model have shown great value in processing huge amount of data. Map Reduce is a common framework for data-intensive distributed computing of batch jobs. To modify fault tolerance, several implementations of Map Reduce go on the entire output of every map and Reduce task before it is consumed. During this paper, we are going to study a modified Map Reduce design that permits knowledge to be pipelined between Mapper and Reducer. This elaborates the Map Reduce programming model on the far side process execution, and can scale reduce completion times and improve system utilization for batch jobs in addition. The Study illustrates that the implementation of Pipelined Map Reduce will scale well and with efficiency method large data sets on artifact machines. In pipelined map reduce hadoop is modified in such way that it can send data directly from Mapper to reducer.

KEYWORDS:*Hadoop, Map-Reduce, Parallel Processing, Pipelined Map-Reduce.*

I. INTRODUCTION

Distributed process on a “cloud” a giant assortment of artifact computers, every with its own disk, connected through a network has enjoyed a lot of recent attention. The hardware is an infrastructure that supports information and task distribution and strong component-failure handling. Google’s use of cloud computing, that employs the company’s proprietary infrastructure, [1] and also the subsequent open supply Hadoop cloud computing infrastructure [2] have for the most part generated cloud computing attention. Each of those environments give data-processing capability by hosting questionable Map Reduce jobs, that do their work by sequencing through information keep on disk. The technique will increase scale by having an outsized variety of freelance (but loosely synchronized) computers running their own instantiations of the Map Reduce job elements on their information partition.

Map Reduce has emerged as a well-liked way to harness the ability of enormous clusters of computers. Map Reduce permits programmers to suppose during a data-centric fashion: they concentrate on applying transformations to sets of knowledge records, and permit the small print of distributed execution of task, network communication and fault tolerance to be handled by the Map Reduce framework. Map Reduce is often applied to batch-oriented computations that are related primarily with time to job deadline. The Google Map Reduce framework [5] and Hadoop system reinforce this usage model through a batch-processing implementation strategy: the whole output of every map and reduce task is materialized to a local file before it will be consumed by succeeding stage. Materialization permits for an easy and stylish checkpoint/restart fault tolerance mechanism that’s vital in giant deployments that have a high probability of slowdowns or failures at worker nodes.We tend to propose a changed Map Reduce design in which intermediate information is pipelined between operators, where as conserving the programming interfaces and fault tolerance models of previous Map Reduce frameworks.

To validate this style, we tend to develop the Hadoop on-line prototype (HOP); a pipelining version of Hadoop Pipelining provides several important benefits to a Map Reduce framework:

I) Because of the reducers begin processing data as soon as it is produced by mappers, they can create and refine an approximation of their final answer at the time of execution. This is called as online aggregation [3]; it can give initial estimates of results several orders of magnitude at high efficiency than the final actual results.

II) Pipelining extends the domain of the problem to which Map Reduce can be applied. Continuously running Map Reduce job accept new data as its come and analyze it immediately. Applications like Event monitoring and stream processing can be handled by using pipelined Map Reduce.

III) Concept of pipelining send data to the succeeding operator which increases chances for parallelism, improve utilization and response time.

II. BACKGROUND

2.1 Programming Approach

To use Map Reduce, the computer programmer expresses their desired computation as a series of jobs. The input to job is associate input specification that may produce key-value pairs. Each job consists of two steps: initial, a user-defined map function is applied to every input record to provide listing of intermediate key-value pairs. Second, a user-defined reduces function called once for every distinct key in the map output and passed the list of intermediate values associated with that key. The Map Reduce programming model parallelizes the execution of those functions and ensures fault tolerance automatically. Optionally, the user can also provide combiner function [5]. Combiners are almost like to the reducer functions, except that they are not passed all the values for a given key: instead, a combiner emits associate output value that aggregates the input values it was passed. Combiner's are generally used to perform map-side "pre-aggregation," that reduces the amount of network traffic needed between the maps and reduce steps.

```
Public interface Mapper<K1, V1, K2, V2>
 {
Void map (K1 key, V1 value,
Output Collector<K2, V2> output);
Void close ();
}
```

2.2 Architecture of Hadoop

The Map Reduce programming model is designed to process large data set in parallel by distributing the Job into a various independent Tasks. The Job considered to here as a complete Map Reduce program, which is the execution of a Mapper or Reducer across a set of data. A Task is an executing a Mapper or Reducer on a chunks of the data. Then the Map Reduce Job normally splits the input data into independent portions, which are executed by the map tasks in a fully parallel fashion. The Hadoop Map Reduce framework consists of a one Master node that runs a Job tracker instance which takes Job requests from a client node and Slave nodes everyone running a Task Tracker instance. The Job tracker is responsible for distributing the job to the Slave nodes, scheduling the job's component tasks on the Task Trackers, monitoring them as well as reassigning tasks to the Task Trackers when they at the time of failure. It also provides the status and diagnostic information to the client. The task given by the Job tracker is executed by the Task Tracker. Fig. 1 depicts the different components of the Map Reduce framework.

2.3 HDFS

The HDFS has some desired options for enormous information parallel processing, such as: (1) work in commodity clusters in case of hardware failures, (2) access with streaming information, (3) dealing with big data set(4) use an easy coherency model, and (5) moveable across various hardware and software platforms. The HDFS [6] is designed as master/slave architecture (Fig. 2). A HDFS cluster consists of Name Node, a master node that manages the filing system name space and regulates access to files by clients. Additionally, there are various Data Nodes, usually one per node within the cluster, that manage storage connected to the nodes that they run on. HDFS exposes a filing system name space and permits user information to be hold on in files. Internally, a file is split into one or a lot of blocks and these blocks are stored in a set of Data Nodes. The Name Node executes filing system name space operations like gap, closing, and renaming files and directories. It determines the mapping of blocks to Data Nodes. Clients read and write request are served by the Data node. The Data Nodes is also responsible for block creation, deletion, and replication as per the instruction given by Name Node.

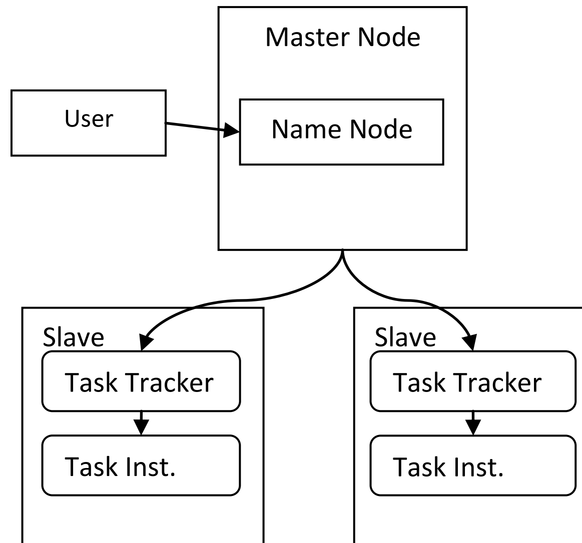

Figure 1: Hadoop Map Reduce

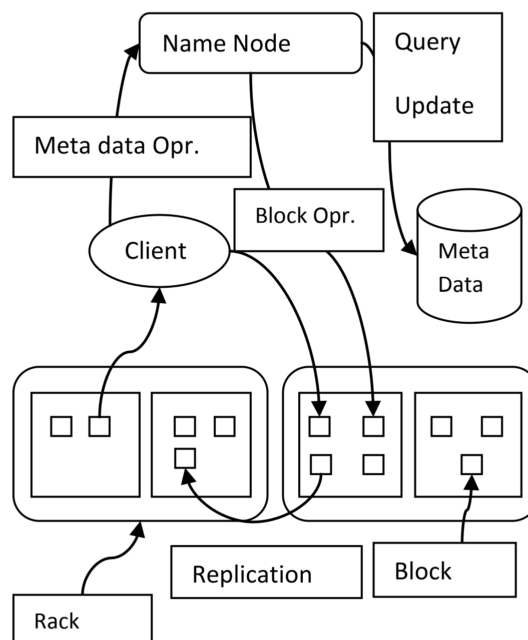

Figure2: HDFS Architecture

2.4 Execution of Map Task

Every map task is provided with a portion of the input file called as split. By default, a split contains a one HDFS block, so the total number of the number of map tasks is equal to the total number of file blocks. The execution of a map task is divided into two passes:

1) The map pass reads the task's split from HDFS, interpreted it into records (key/value pairs), and applies the map function to every record.
2) After applying map function to every input record, the commit phase stores the final output with the Task-Tracker, Then Task Tracker informs the Job-Tracker that the task has finished its execution. The map method specifies an Output Collector instance, which collects the output records created by the map function. The output of the map step is consumed by the reduce step, so that Output Collector stores output produce by mapper in a simpler format so that it is easy to consume for reduce task. By applying a partitioning function

intermediate key are assigned to reducers, so the Output Collector applies partition function to every key produced by the function map, and stores every record and partition number value in an in-memory buffer. The Output Collector spills this buffer to disk when buffer reaches its maximum capacity. At the time of commit phase, the final output of the map task is generated by merging all the spill files generated by this task into a single pair of data files and index files. These files are registered with the Task-Tracker before the completion of task. The Task-Tracker will read these data and index files when servicing requests from reduce tasks.

2.5 Execution Of Reduce Task

The execution of a reduce task is divided into three passes.

1) The shuffle phase is responsible for fetching the reduce task's input file. Each reduce task is appointed a partition of the key range generated by the map pass, so that the reduce task must receive the content of this partition from each map task's output.

2) The sort phase group together records having same key.

Public interface Reducer <K2, V2, K3, V3>
{Void Reduce (K2 key, Iterator<V2> values, Output Collector <K3, V3> output);
Void close ();}

3) The reduce phase appoint the user-defined reduce function to every key and corresponding list of values. Within the shuffle phase, a reduce task fetches data from every map task by sending HTTP requests to a configurable number of Task-Trackers at once .The Job-Tracker relays the location of each Task-Tracker that hosts map output to each Task-Tracker that executes a reduce task. Reduce task can't receive the output of a map task till the map has finished execution and committed its final output to disk. When receiving its partition from all map outputs, the reduce task enters into the sort phase. The map output for every partition is already sorted by the reduce key. The reduce task merges these outputs together to produce a single output that is sorted by key. After that task enters the reduce phase, in that it calls the user-defined reduce function for every distinct key in sorted order, passing it the associated list values list. The reduce function output is written to a temporary location on HDFS, When the reduce function has been applied to every key in the reduce task's partition .The output of both map and reduce tasks is written to disk before it can be consumed. This is normally expensive for reduce tasks, because output of both task is written to HDFS. Fault tolerance is simplified by output materialization, because it reduces the amount of state that must be restored to consistency at the time of a node failure. If any map or reduce task fails, the Job-Tracker simply schedules a new task to perform the same work as the failed task. Since a task never exports any data apart from its final answer, no additional recovery steps are needed.

III. ARCHITECTURE OF PIPELINED MAP REDUCE

In this section we are going to discuss extensions to Hadoop to support pipeline [4] .Figure 3 and 4 depicts data flow of two different Map Reduce implementations. The first data flow (fig 3) corresponds to the output materialization approach employed by Hadoop; the second dataflow (fig 4) on corresponds to pipelining and that we called it Pipelined-Map Reduce.

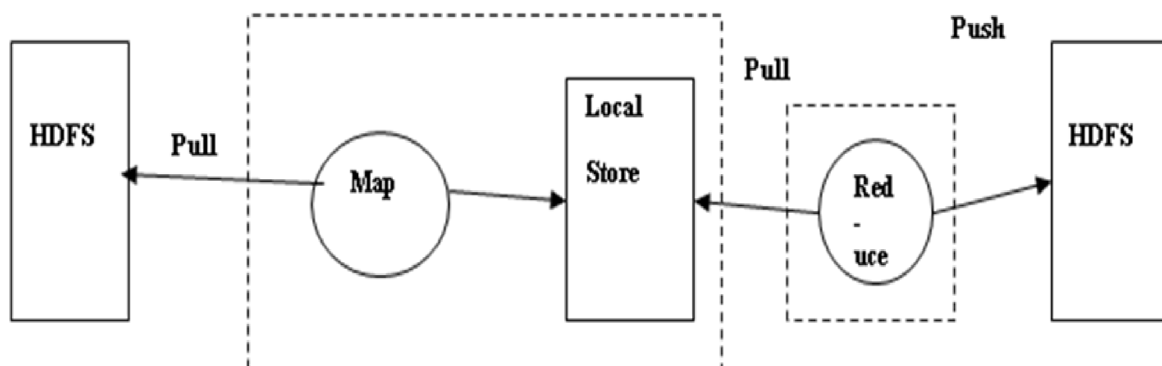

Figure 3: Hadoop data flow for batch

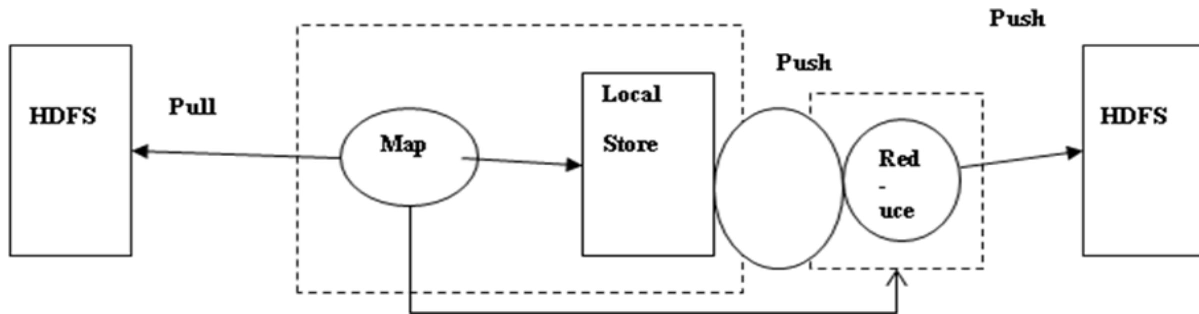

Figure4: Pipelined Map Reduce data flow

In general, reduce tasks traditionally send HTTP requests to pull their output from every Task Tracker. It means that map task execution is completely separated from reduce task execution. To support pipelining, the modification is done in the map task to instead push data to reducers as soon as it is generated. To give an intuition for how this is going to be work, we begin by studying a straight forward pipelined work flow, and then discuss the changes we have to make to achieve good performance. As per the paper Hadoop is modified to send data directly from map task to reduce task. When a client send a new job to HDFS, Job Tracker assigns the map task and reduce task associated with the job to the available Task Tracker slots. For the purpose of discussion, we consider that there are sufficient free Slots to assign all the tasks for every job. Due to the modified version of Hadoop every reduce task can contact to each map task at the time of initiation of the job, as well as it opens a TCP socket which will be used for pipelining the output of the map function. When every map output record is produced, the mapper determines to which partition (reduce task) the record should be sent, and immediately sends it through the appropriate sock. The pipelined data is received by the reduce task from every map task and it stores it in an in-memory buffer, it also spills the sorted outputs of the buffer to disk if needed. Whenever the reduce task learns that each map task has completed its work, it performs final merge operation of all the sorted outputs and applies the user defined reduce function, write the final output to the Hadoop Distributed File System.

IV. CONCLUSION

In this paper, we studied the Hadoop data flow and Pipelined Map Reduce data flow. Pipelined Map Reduce is much better than the traditional one. It reduces the completion time of tasks. That means the implementation of Pipeline Map Reduce can processes large datasets effectively.

REFERENCES

[1] J. Dean and S. Ghemawat, “Map Reduce: Simplified Data Processing on Large Clusters,” Comm. ACM, vol. 51, no. 1, 2008, pp. 107–112.
[2] http://hadoop.apache.org
[3] HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. Online aggregation. In SIGMOD (1997).
[4] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein in the Proceedings of the 7th USENIX symposium on Networked Systems Design and Implementation (NSDI 2010), April 2010.
[5] DEAN, J., AND GHEMAWAT, and S. Map Reduce: Simplified data processing on large clusters. In OSDI (2004).
[6] D.Borthakur: The Hadoop Distributed File System: Architecture and Design (2007).