Accelerating Map-Reduce using Distributed Cache as Middleware with Partial Pre-shuffling on Hadoop Cluster and Securing Memcached Server

Accelerated Map-Reduce Implementation on Hadoop Cluster

[1]Chandrashekhar Singh, [2]Ayushi Srivastava
[1]B.Tech - Information Technology, [2]B.Tech – Information Technology
[1]Department of Information Technology,
[1]SRM University, Kattankulathur, Tamil Nadu, India – 603203

Abstract **- Map-Reduce is a widely-used model for data parallel applications enabling easy development of scalable programs on clusters of commodity machine. Advancements in disk capacity have greatly surpassed those in disk access time and bandwidth. As a result, disk based systems are finding it increasingly difficult to cater to the demands of a cluster-based system. A cache memory has faster data access rate than normal disk. In this paper, we have implemented a method to improve the performance of map-reduce by using distributed memory cache as middleware between the map and reduce phase with highly secure memcached server. Moreover, it provides partial combining of data along with the Map task itself, so called partial pre-shuffling. The project also provides data assurance of intermediate key-value pairs via a web interface. The results of the overall setup were promising over a Hadoop cluster.**

Index Terms **- Distributed File System, Hadoop, Map-Reduce Memcached, High throughput, High speed data access, Cluster-based computing, Middleware.**

I. INTRODUCTION

In the current environment of technology, a practical solution to large scale applications is to distribute computations among a cluster-based organization. With original parallel programming models, programmers have to handle extremely tedious issues like concurrency and a lot of effort is spent upon fault tolerance, data-integrity and system-level inter-process details.

Recently, Map-reduce proposed by Google has emerged as an attractive option. [1] Through a simple interface with two functions, map and reduce, map-reduce facilitates implementation of many real world applications. Map and reduce operations can be parallelized pretty easily using a cluster-based setup where the input data is divided and fed to separate nodes of the cluster. The reduce operation is parallelized by partitioning the reduction of keys across many machines. Load balancing, communication, task-scheduling and fault-tolerance are the aspects which Map-reduce takes care of very well.

Short jobs are very common in map-reduce, for example, queries on logs for debugging, monitoring and business-intelligence. The key issue here, is response time. Almost all the massive hosting sites like Yahoo, Google, Amazon and an innumerable others use map-reduce for various data-analysis. There are many data-analysis systems providing SQL like queries on top of Map-reduce. For example Google's Sawzall, Yahoo!'s Pig and Facebook's Hive. These systems prove to be extremely time-intensive to Map-reduce. Therefore there is a need to optimize execution time allotted to each source.

Traditionally, Map-reduce was developed to cater to the needs of fault tolerance in cluster-based setups. Therefore, since it relies on the replication of data, there is an extensive use of disk space and disk reading. This affects the disks' latency and throughput seriously. Since all map outputs will be read by the reduce tasks remotely, the shuffling of the intermediate data becomes a very time-intensive task.

II. MAP-REDUCE PROGRAMMING MODEL

Map-reduce is a programming model and computing platform well suited to distributed and parallel computations. In Map-reduce, a program contains two user defined functions: the map function and the reduce function. The Input and Output formats are at the exclusive discretion of the programmer as per the demands of the application as long as they conform to the <key, value> pairing system.

$$\begin{aligned} &\text{Map: } (k1, v1) \text{ ---> } [(k2, v2)] \\ &\text{Reduce: } (k2, [v2]) \text{ ----> } [k3, v3] \end{aligned} \qquad (1)$$

As showed in Eq. 1, the map function traditionally applies user defined logic on every input (k1,v1) and transforms it into a list of intermediate key, value pairs [(k2,v2)]. Then the reduce function applies user-defined logic to all the intermediate values [v2] associated with the same k2 to produce a list of output key, value pairs [k3,v3]. Initially proposed by Google, Map-reduce has been

used in a wide range of applications, such as web access log stats, wen link graph reversal, statistical machine translation, graph reversal, Artificial intelligence and so on.

III. MAP-REDUCE FRAMEWORKS, EVOLUTION AND INTRODUCTION TO HADOOP

Map-reduce is a highly versatile process. Apart from cluster-based computations it may also be implemented in multi-core environments. Phoenix is an implementation of Map-reduce to program multi-core ships as well as shared memory multi-processors. It has also been implemented in GPUs, such as Mars. But this paper's main focus is on cluster-based computations. There are two popular Map-reduce frameworks. Google's Map-reduce framework is in C++ and built atop Google File System (GFS) [2]. It is not open-source and publicly available. The second one is Hadoop, an open-source Map-reduce implementation in Java as part of Lucene project, allows any programmer taking advantage of Map-reduce to build parallel applications. Since it simulates GFS and Map-reduce of Google, it inherits the same set of features. [3] The framework transparently handles all the other aspects of execution on clusters, irrespective of their extensiveness. It schedules tasks on unreliable commodity machines and yet, is fault tolerant. In DFS i.e, Distributed file system, file blocks are duplicated and stored on disks of various nodes. The map-reduce framework schedules map tasks on the machines where the input partition resides therefore moving the code to data, rather than the reverse which is the traditional approach to most computing algorithms. The DFS aim at huge clusters of homogeneous commodity machines, however, Map-reduce is often used for short jobs where the response time and fault-tolerance is critical.

Many people have tried to improve Map-reduce frameworks by designing new schedulers for heterogeneous clusters, or by treating the intermediate data as the main objects for dataflow programming frameworks in clouds. Some suggested replication of intermediate data or a different design for its storage. This project's prototype is built atop Hadoop's Map-reduce framework. For the purpose of fault-tolerance and concurrent Input/Output, the input files are split into even-sized chunks replicated on different nodes.

The individual map-reduce applications are referred to as 'Jobs', getting submitted to the Job Tracker or the master node, which is unique to the cluster. TaskTrackers, one per node in the cluster, ask the JobTracker for new tasks periodically. If unassigned tasks are recognized by the JobTracker, it allocates those to the requesting TaskTracker, which in return run each task in a separate JVM to prevent one fault from disrupting the entire process.

The whole application is processed in a parallelized manner, distributing the various map and reduce tasks to the nodes, where the code is brought to the node and the functions get executed to emit key-value pairs to a memory buffer file on the local disk. Each chunk, of the map output file corresponds to a reduce task. Once a reduce task has got its file trunks from all the map output tasks, it sorts and merges them according to their keys. Then the reduce function is applied to sets of separate keys to get the resulting key, value pairs to files on HDFS. By considering the progress scores of all map tasks and reduce tasks, JobTracker calculates the progress rates of map stage and reduce stage, and uses the rates as a reference of task scheduling. This whole process is described below in Fig. 1.

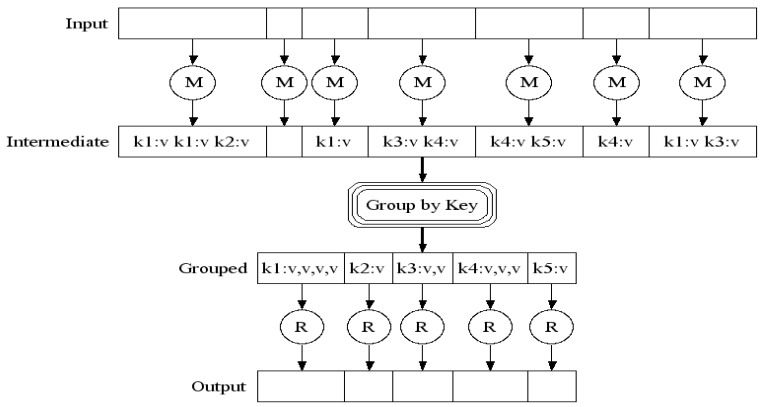

Figure 1 Actual Map-Reduce Architecture [8]

IV. MEMECACHED, IT'S ADVANTAGES AND IMPLEMENTATION WITH MAP-REDUCE

In order to verify the idea, an existing open-source tool is adapted to provide the required storage capabilities. Memcached is a high-performance, distributed memory caching system [4]. Although application-neutral in design, it was originally developed to accelerate database-driven web applications by caching the results of database queries. This tool is currently being used by a number of websites including Wikipedia, Facebook, Yahoo, Alibaba etc. Moreover, it is not processor intensive and very versatile. It has low latency and a high throughput. Memcached is initialized as a daemon process on servers and provides a global key-value store through a client interface. There are several clients and APIs available on the internet which scale well to a number of requests. The client API is responsible for keeping a list of available servers of memcached and hashing keys to the global store. [5] For this project, the use of Memcached proceeds in the following 3 steps:

1) The map outputs, i.e, the output key value pairs of the map function are loaded to the memcached servers. The key is made up of the map tasks' ID and its target reduce tasks' ID. The output is set as the value.

2) This step is more about successfully exploiting memcached APIs and a global storage. We check the server if a similar key has been added, if so, the next value is simply appended to it. This completes half the combining of the sorting stage and lessens the amount to be stored in the memcached server.

3) Once the reduce task is spawned by the Task Tracker, it gets all the finished map task's information from the Task Tracker. It gets it keys from the memcached servers and runs the reduce function as specified by the user.

V. MAP TASK PROCEDURE

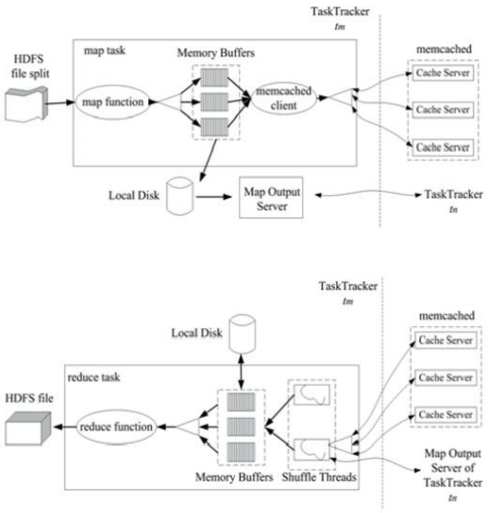

Figure 2 Map-Reduce with Memcached Architecture

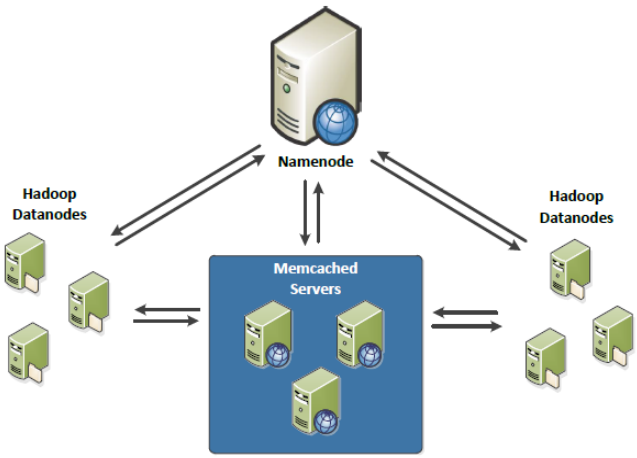

Figure 2 Memcached with respect to Datanodes and Namenodes

As shown in the Fig. 2, when processing an HDFS file split under the map function, a map task assigns each tuple to memcached. It is the responsibility of the memcached client to assign appropriate server to the tuples based on their reduce task IDs. This is done by hashing. If the memcached memory falls short, the contents are saved in the local memory but is still managed by the memcached client. Sizes of map outputs are directly related to the size of the HDFS File Split. To resolve this, while adding the tuples, the values of some tuples is appended if their key had already been added to the server by a different tuple. This greatly lessens the response time of the shuffle and sort problem. It is these combined tuples that get saved in the memcached. While in the map function, all the keys are stored in a local array list to help us keep their record and retrieve them without error. Now, we will proceed with the Reduce function as a combiner. We pass a master key value to the Output Collector function, with which it runs the combiner.

Since maximum work of the sorting and shuffling is done, a reduce function runs in no time to do the rest of the work and get a record of the keys. As soon as this is over, the reduce function runs upon the keys in a distributed context (a set of keys on one node), and computes the total output of final tuple as per the user's definition.[7]

In this approach, the map task becomes a little more intensive but it greatly simplifies the work and response time of the reduce function and the final outputs are stored in the local file system connected to the master node.

VI. NETWORK TRAFFIC DEMAND

We save network bandwidth by taking advantage of the fact that the input data (managed by HDFS) is stored on the local disks of the machines that make up the cluster. HDFS divides each file into blocks, and stores several copies of each block on different machines. The JobTracker takes the location information of the input files into account and attempts to schedule a map task on a machine that stores a replica of the input data, which means "moving the code to the data". Otherwise, it attempts to schedule a map task near a replica of the data. In reality, most map input data is read locally and brings no network traffic. Traffic is also lowered by the pre-shuffling that is done during the map task.

VII. FAULT TOLERANCE

Due to the decoupled operation of tasks, the concern is only with map task failure, reduce task failure, memcached server failure and TaskTracker failure separately. This is taken care of by the HDFS and memcached transparently. If a map task fails, another map task would complete it. If the need arises for re-spawning a reduce task, the key-value pairs which have not been read by the failed one, are read by the newer one.

If memcached fails, data will be lost and this calls for a reset of all the map tasks from their initial states. The same restart is called for if the Task-Tracker fails, however, the key value pairs already sent are retained. This might result in the loss of accuracy and replication might be seen in the output. However, a cluster of memcached itself can be implemented i.e, the infrastructure provided by memcached is a whole cluster and provide complete fault tolerance and data assurance.

VIII. OUTCOMES AFTER EXECUTING THE SETUP ON HADOOP CLUSTER

Vulnerabilities in the memcached server and it's patch

The memcached creates a server on the system and unfortunately by default this server has no authentication or data access restrictions. The data inside the memcached server can be accessed through telnet or through a PHP API. This allows user to delete and modify the data inside it. So, any system present in the network can access the data and can modify. To overcome this vulnerability, usage of data could be bound only to a particular system by binding IP of the system to the memcached server, so that

no other IP can access the server's data. Moreover, we can create a new user without root permission and can bind the memcached to be accessed only through that user. Above two points makes the memcached very secure.

Getting Intermediate data assurance

There are memcached APIs available in Java, Python, PHP and other languages, through which the content of memcached can be easily dumped in the mid of the process in real-time [6]. For this implementation, a PHP web interface can be created which allows to view, modify and delete the data inside memcached at any intermediate stage. This provides complete data assurance in real-time.

Plots

In the following plots, x-axis represents the various file sizes and y-axis represents time in milliseconds. The experiment was done with different file sizes on Hadoop cluster. The file sizes are ranging from very small size of few kilobytes to gigabytes.

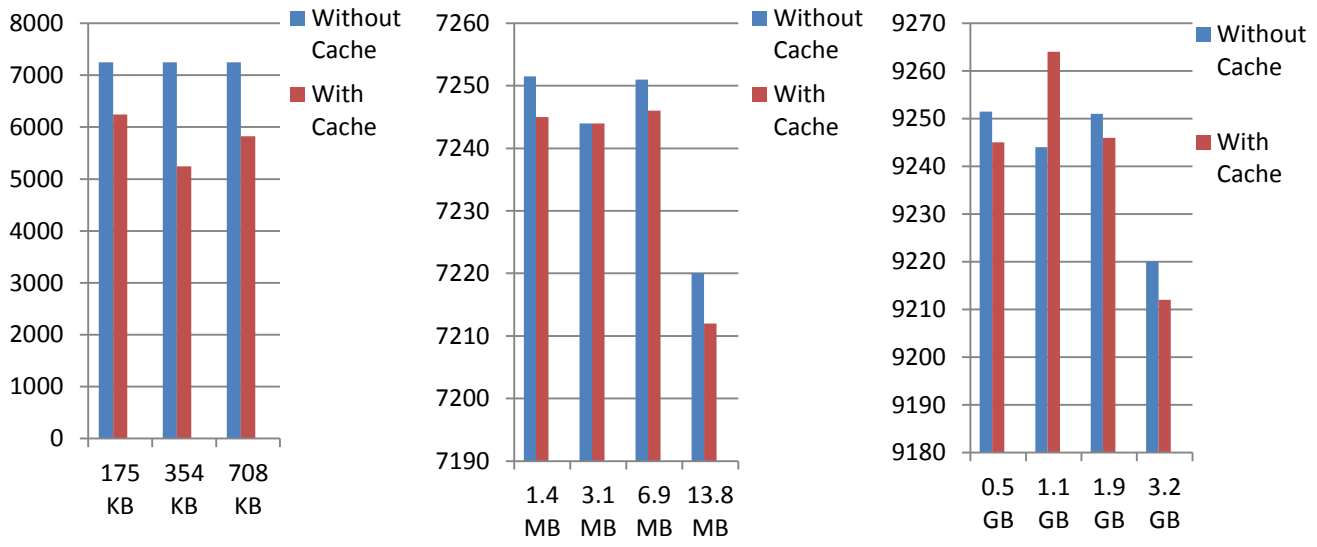

Figure 2 The plot showing the outcomes of the experiment

IX. Conclusion

In this paper, we have implemented memcached or distributed cache as a middleware at the intermediate stage of MapReduce between map and reduce phases. Since a cache memory provides a higher data transfer rate than disk, our results were quite faster than the usual way. However, with a better quality memcahced, the above discussed model can achieve much higher speed. Few graphs are presented to compare the outcomes of usual method and our method. With this method MapReduce tasks will become meteoric and will be very useful for the various organizations that perform extensive data processing. Further research can be done on using the physical cache memory or a SSD as the middleware between map and reduce phases, which are the fastest memory available.

References

[1] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters", Google White paper, Research at Google, http:// research.google.com/archive/mapreduce-osdi04.pdf

[2] Sanjay Ghemawat, Howard Gobioff, Shun- Tak Leung, "The Google File System", Google white paper, Research at Google, http:// research.google.com/en//archive/gfs-sosp2003.pdf.

[3] D. Borthakur. "The hadoop distributed file system: architecture and design." The Apache Software Foundation, 2007.

[4] Memcached wiki page, Revised BSD License, en.wikipedia.org/wiki/Memcached.

[5] Memcached. http://www.memcached.org/, March 2012.

[6] Performance Benchmark for Various Memcached Clients in Java. http://xmemcached.googlecode.com/svn/trunk/benchmark/benchmark.html

[7] GE Junwei, Xian Jiang, Yiqiu Fang, "Improvement of the MapReduce Model Based on Message Middleware Oriented Data Intensive Computing", 2011 Seventh International Conference on Computational Intelligence and Security

[8] Research at Google, Map-Reduce, http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0007-0001.gif