Indonesian Journal of Electrical Engineering and Computer Science
Vol. 4, No. 3, December 2016, pp. 629 ~ 636
DOI: 10.11591/ijeecs.v4.i3.pp629-636

A New Memory MapReduce Framework for Higher Access to Resources

ZuKuan Wei[1], Bo Hong*[2], JaeHong Kim[3]
[1,2]College of Computer Science & Engineering, UESTC, ChengDu 611-731, P.R. China
[3]Dept. of Computer & Info. Engi. YoungDong Univ. Chungbuk, 370-701. Korea
Corresponding author, e-mail: anlexwee@uestc.edu.cn[1], owen00sun@qq.com*[2],
jhong@youngdong.ac.kr[3]

Abstract

The demand for highly parallel data processing platform was growing due to an explosion in the number of massive-scale data applications both in academia and industry. MapReduce was one of the most meaningful solutions to deal with big data distributed computing. This paper was based on the work of Hadoop MapReduce. In the face of massive data computing and calculation process, MapReduce generated a lot of dynamic data, but these data were discarded after the task completed. Meanwhile, a large number of dynamic data were written to HDFS during task execution, caused much unnecessary IO cost. In this paper, we analyzed existing distributed caching mechanism and proposed a new Memory MapReduce framework that has a real-time response to read or write request from task nodes, maintain related information about cache data. After performance testing, we could clearly find MapReduce with cache significantly improved in IO performance.

Key words: *massive data processing, MapReduce, cache*

1. Introduction

MapReduce platform is the latest technological achievements in the field of big data processing. It is a well known framework for programming commodity computer clusters to perform large-scale data processing [4]. A MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner [2]. The platform simplifies the process into two stages: map and reduce. Developers simply need to implement a map function and a reduce function to finish a distributed program design. Map make key/value pair as input data and generate new key/value pair data, which will be written to local disk as an intermediate result, after map task done its function. MapReduce automatically combine these intermediate data by key. Data with same key should pass to a reduce function for a further operation and produce a final result as key/value pair saved in HDFS. A job has been finished after its reduce function done .Depend on strong programming ease of use, parallel processing efficiency and scalability, MapReduce gained extensive attention and usage in the industrial and scientific fields. As Hadoop provides a most widely used platform for MapReduce, the prototype system of our work is Hadoop and MapReduce.

In this paper, we focus on lifting huge amounts of data read efficiency to optimize MapReduce framework. While there are oceans of IO extensive jobs running within MapReduce and operating large-scale data from HDFS or local disks, IO bottleneck will be an urgent problem to be solved, or it could reduce operation efficiency, even has a bad effect on the execution of other jobs in the cluster. To handle this problem, we figure out a new storage solution for MapReduce, that cache will be introduced into our new Memory framework. Media data stores from disk into memory, speeding up the read speed for subsequent jobs and reducing the impact caused by IO. We consider Hadoop running on a cluster or distributed environment, therefore Redis, a very mature distributed cache database, is an alternative solution for our Memory framework. Redis is an open source, high-performance, distributed memory database, widely used in industrial fields, and industry recognition, community support is high and easy to use for developers. Our solution is built on the improvement and optimization for MapReduce framework and is transparent to user. These advantages can help us achieve rapid deployment and make the smallest impact on the before practical application

Received August 3, 2016; Revised September 26, 2016; Accepted October 11, 2016

or system. Firstly, we summarize the current MapReduce workflow and point out a series of existing practical problems in it. Furthermore, we present a new system framework, MapReduce on Cache. Finally, after the introduction of caching strategy, it is possible to occur on data hot-spots. So we need to make a new scheduler to address this trouble.

2. Background

The major contribution of Hadoop MapReduce solves the poor scalability as search engines are processing large-scale data. MapReduce implementation is largely draws Google MapReduce design ideas, including simplified programming interface, and improved the fault-tolerance of system.

Traditional distributed programming, such as MPI, users need to focus on the much details,for example, data partitioning, communications nodes and data transmission etc., resulting in the development of heavy workload and error-prone. MapReduce simplifies distributed programming, completely modularized details that often needed to be focused by parallel programming. Developers just put all attention on business logic implementation, improving the efficiency of the development to a certain extent.

With the increasing amount of data, the existing cluster may perform high-latency and low-throughput due to its lack of resources, such as CPU, Memory, network bandwidth and the like. As MapReduce has good scalability, we can improve the system's computing and storage capacity through adding new machine to Hadoop cluster. Under the distributed environment, cluster failures, for example, disk corruption, node communication failure, etc., put the system stability at an unpredictable risk. A single node failure also would lead to mission failed and data loss In terms of this situation, calculating migration or data migration policies are taken into consideration by Hadoop to perfect the availability and fault tolerance for the cluster.

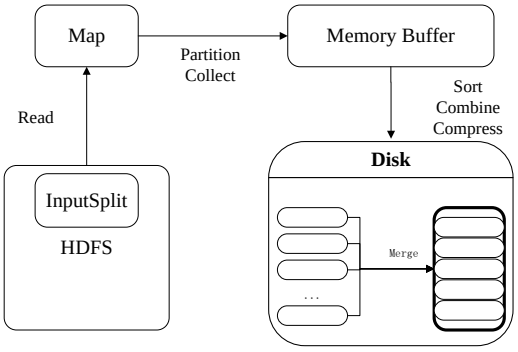

Figure 1. Map Execution Overview

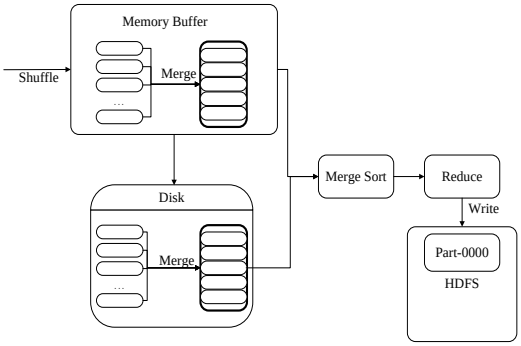

Figure 2. Reduce Execution Overview

Figure 1 shows a map task's workflow. Firstly, map task reads Input Splits (mapper input partition) as data source from HDFS and further assemble them to some key/value pairs, then call an user-defined map function to process these pairs. Second, after finishing the map function, map task invokes collect function so as to put new generated data to a cyclic memory buffer until it filled. MapReduce takes data from buffer to write to a temporary file in the local disk and clear the buffer prepared for next fill. At last, all temporary files will be merged into one ultimate file.

Figure 2 shows a reduce task's workflow. Firstly, reduce task usually performs copy and shuffle the corresponding partition of mapper output. During the data transmission, reduce task combines data in the memory and disk, prevents from creating much files in the node. Second, the reduce task invokes a user-defined reduce function to do further operation for partitions. Eventually, the final result (reduce output) is written to HDFS.

According to observe the workflow of map and reduce task above, we can point out that the initial obtainment and ultimate storage are all relevant to HDFS, a distributed file system that usually choose disk to store each job's input and output data. Unfortunately, this option inevitably produces I/O, an important factor to restrict data read and write speed, especially

under the circumstances of processing massive data-extensive jobs. After the map task is finished , MapReduce writes intermediate results to the local disk of node that perform tasks, but the existing framework does not consider the locality while allocating reduce tasks, easily leading to task=failure or partial deletion during data transmission. Reduce task cannot continue to do their parts without completed transmission. When a copy of data is stored in one node, if a large number of tasks need to access this copy, the data becomes the hot-spot, one of the factors bringing about MapReduce performance bottleneck.

Nowadays, there are some researches and products related to MapReduce improvement, such as Twister, a lightweight MapReduce, produced by Indiana University. It provides Pub/sub messaging based communication/data transfers In order to avoid repeated loading of data in inter-iteration, it supports long running (cacheable) map/reduce tasks Static data can always be loaded in memory among tasks, so that reducing the disk I / O overhead. But Twister is based on such premise: data collection and intermediate data can be fully stored in memory. Due to it failed to adopt disk that force it to must guarantee to have enough memory space to hold all the data. If the application encounters TB-level data, it is much difficult to have enough memory to complete big data processing. Therefore, Twister is not suitable for use in business mode.

In our work, memory and disk both have pivotal roles, when there is not enough memory space to accommodate data, disk would interact with memory to save data and release memory space for further processing.

3. Proposed Framework

In Figure 3 represents the workflow of Memory MapReduce. Compare with Figure 1, and 2, we can obviously find out traditional HDFS is replaced by cache. In any case, map and reduce tasks all operate input or output data from memory (cache). Memory MapReduce inherits the basic MapReduce framework and distributed computing model, relies on a distributed cache system, which is responsible for the input and output data management. Since the basic workflow has not changed, no matter select HDFS or memory as storage media, the workflow is transparent to developers.

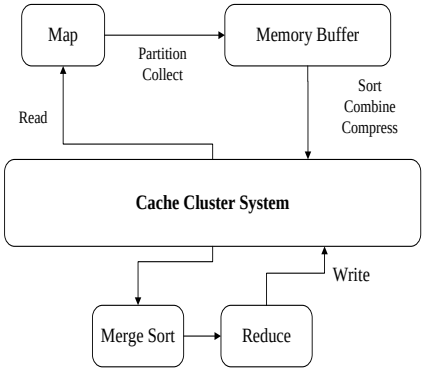

Figure 3. Cache MapReduce Workflow

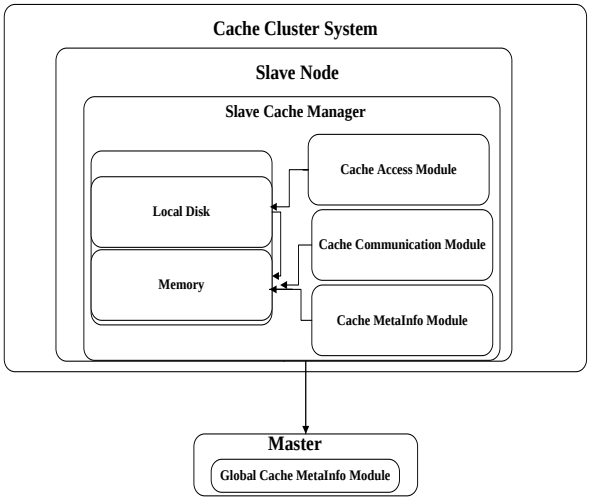

Figure 4. Cache Cluster System Architecture

Figure 4 illustrates the architecture of specific cache cluster system, which maintains the basic Master / Slave architecture. The master node maintains all cache Meta information in a distributed cache system. The slave node manages cache data, and its meta-information respectively. Also, with the purpose of building a user-friendly direct access to the specified data cache by the master, slave nodes use heartbeat to keep communicating with the master node

for regularly reporting their cache status. The slave cache manager includes three critical modules:

Cache Access Module provides service to read and write cache data. In the first place, selects a free memory space is in line with the size of the data, and then writes data to the memory as a cache. If the free memory space is insufficient, we need a cache scheduling strategy to ensure that the latest data can be stored in memory. The strategy involves the exchange of data between memory and local disk.

Cache Communication Module provides exchange service of cache data between nodes. In the slave cluster, node's data locality divides into three parts: local, rack and any. When the node i want the certain data that didn't be retrieved neither in the memory nor the disk, it will send a query request to the master node for the sake of the data location. As the master receives the request and do a search in its global Meta information of cache data. During this search, the master always gives priority to nodes where locate in the same rack with the node i. Considering that the same rack often within a local area network, data transmission speed between nodes are faster. The Master notifies the node i. to obtain data on a particular node.

Cache Meta-info Module provides exchange service of cache meta- information. When the cache has been modified in the node, such as new cache, cache replacement or cache update, the module will automatically notify the Master to update its corresponding global cache meta-information.

4. Cache Scheduler

This section describes the cache scheduling algorithm that is different from the traditional FIFO/LRU algorithm. Thanks to the distributed environment, our algorithm must take some special conditions into account. Part 4.1 and 4.2, respectively introduces the meta-information to be collected and the pseudo-code of scheduling algorithm.

4.1. Cache Scheduling Meta-Info

Cache is one of the core technologies in the management of memory-level data. Accurate scheduling policy offers a more feasible memory space for caching hot-spot data, reduces the number of disk accesses when reading data, and improves the service quality of data cache for the Memory MapReduce platform.

The traditional cache scheduling policy only consider the frequency of data access, as well as the complexity of space and time, however, MapReduce is closely associated with computing and storage resource, forcing it must pay attention to these resource factors while executing scheduling policy. In MRV2 or Yarn, resource can be divided into two main parts: CPU core(Computing) and memory(Storage).At the time while allocating map or reduce task to the nod that has available cache data in memory, we need to consider whether the node has enough resources to execute the task. In the process of executing tasks, MapReduce will split the required file into several more little files. Let's take an instance. There is an input file origin.txt (size=201000000 Bytes) in HDFS. MapReduce splits origin.txt refers to some corresponding parameters, such as minSplitSize, blockSize, each of which reads from the configure file. Slice meta-information saved in the file header as following structure: <slice start address, slice length, the storage node>. For example, we set blockSize = 67108864 Bytes to split origin.txt. And the processed result is:

a. {0,67108864,server3,server2}
b. {67108864,67108864,server2,server3}
c. {134217728,66782272,server2,server3}

All of these data above will be passed to the master node and be treated as an important factor in cache scheduling policy. Facing challenges posed by cache data scheduling design, our work takes slice hot-spot access, resource load of the node where slice located in and network performance as three major factors, draws up a reasonable scheduling strategy aims to avoid emerging mission of hungry death and overloaded single-node. We describe three detailed factors below:

1. Slice hot-spot access is determined by data access in a particular period of time, reflecting the the limitation of time. We defined such data structure to save data access record, <Slice, LocalNum, NoLocalNum, TotalNum, LastAccessTime>

LocalNum records the cache access times of executed tasks in the local node, NoLocalNum records the cache access times of executed tasks from other nodes, TotalNum equals LocalNum plus NoLocalNum and LastAccessTime means the last access timestamp of the cache.

2. Resource load includes CPU cores and memory size in current phase. Each slave node has its own resources that will be allocated or released by tasks (either map or reduce). Therefore, the remaining resources are constantly changing, and these changes need to be perceived by the master, thus facilitating the further task scheduling. Slave nodes keep connection with the master node through heartbeat mechanism. The message a slave sent contains a node's latest resources at a time. Because each node performs different tasks, they are required to obtain real-time execution progress of the task, so as to notify when each task finished. We define such structure<TaskId,NodeId,Progress,ExecutedTime,Timestamp>to track every task's execution progress. TaskId represents the task id identified by MapReduce, NodeId represents the node id, and Progress represents the current execution percentage of task. ExecutedTime represents how long it executed. According to these meta-info, the Master almost predict the time when a task finished. We use a formula to describe this meta-information's relationship

$$remainTime = \frac{1-progress}{progress} * processTime - (now - timestamp)$$

remainTime means the remaining time for task execution, now means the current timestamp.

3. Network performance is the last factor to be emphasized. Data copy is much possible to be transferred through network, so in the process of copying data, we prefer a better network, otherwise, it may cause a number of tasks waiting for a copy with slower data transfer rate.

4.2. Scheduling Algorithm

The scheduling algorithm considers factors, such as data slice hot-spot, resource load and network transmission, their meta-info all saved in the master node. Our algorithm is based on the normal communication between nodes (i.e., heartbeat is well).

```
List<Node> nodes= findCachedNode();
If nodes is not empty then
   Node node = chooseOneNode(nodes);
While(node is not null)
then;
If(checkResourceRequest(Task.ResourceRequest)) then
            allocate(node,task);
     updateMetaInfo
    else
            Scheduler cheduler = compare(transferTime,waitingTime);
If(scheduler is Transfer) then
    copyToOtherNode(fileSplit);
else
            waitingTask(task);
end if;
end if;
            end while;
end if
execute map & reduce task;
if(checkHotPotThreshold) then
   Node nearestNode = findNearstNode();
   copyToOtherNode(fileSplit, nearestNode);
            wakeupWaitingTasks();
end if;
```

Figure 5. Task Scheduling Algorithm

If there are submitted jobs, this function is called when master node allocate resources to tasks.

Firstly, the algorithm invokes function findCachedNode() to find out all nodes that meet task's input request, that is to say, these nodes have required data. Iterate over these nodes and use checkResourceRequest () to pick up the first node that meet the request. Second, the master node passes the task to the slave one, and then accordingly updates meta-info of data slice and resource load. If all nodes have no enough resource to execute this task, we choose a smaller cost scheduler by comparing the transmission time of copying data with waiting time. The selected scheduler will copy the data to other nodes (copyToOtherNode) or add the task to a waiting queue (waitingTask). After the task end, the LocalNum, NoLocalNum will be updated respectively. If the LocalNum value ups to a given threshold, the data is treated as hot-spot and invoke findNearstNode () to find out the nearest node from the node with the data.

At last, the algorithm invokes copyToOtherNode function to reduce subsequent tasks have large-scale access to the past node in the wake of adding a new copy. As the data transmission is over, WaitingTasks () makes a role to wake up tasks in the waiting queue. These tasks will be allocated to given nodes that have copies. Figure 5 gives pseudo code for the scheduling algorithm.

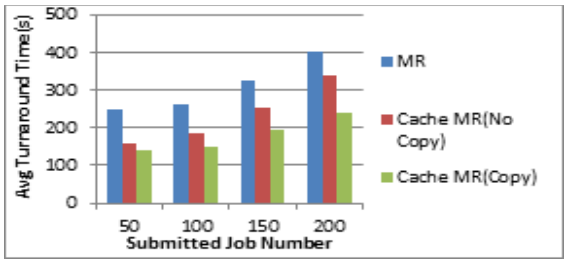

Figure 6. Slave Memory (2G * 8)

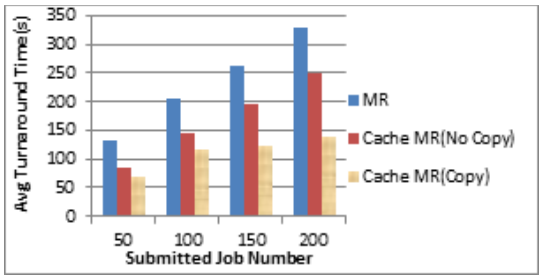

Figure 7. Slave Memory (4G * 8)

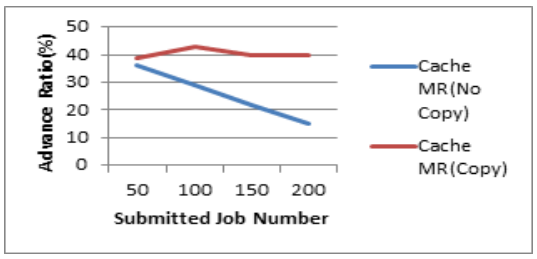

Figure 8. Slave Memory (2G * 8)

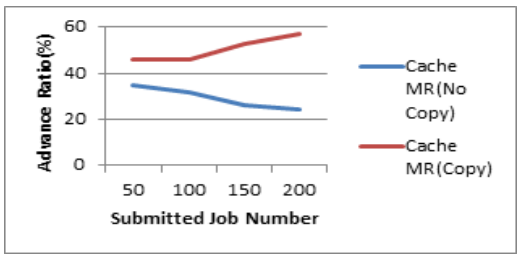

Figure 9. Slave Memory (4G * 8)

5. Experimental Evaluation

The averaging turnaround time of job is seen as a main performance indicator in our experiment. It indicates the job required time from submission to completion and its formula is:

$$AvgTime = \frac{\sum_{i=1}^{n}(finishTime - submissionTime)_i}{n}$$

In order to verify the effectiveness in the use of caching mechanism, our performance tests aimed at different memory space to figure out the relationship between the numbers of job and averaging turnaround time. The suite of experiment used cluster of 9 nodes, there is one master node and 8 slave nodes, each of which has a 4G memory. According to configure the memory parameter in MapReduce, we respectively set memory to 2G and 4G to make cluster's total memory size sums up to 16G and 32G (Note: only 8 slave nodes taken into consideration).Test data is one sample of customers check-in records from some hotels (almost

2G). We need to count where these customers come from in a given period, mainly involving word-count and Top-K operation. By comparing ordinary MapReduce, Cache MapReduce without Copy, Cache MapReduce with Copy, and our results shown below.

Figure 6 and 7 shows the averaging turnaround time in the cluster of 16G and 32G.We discover that while the number of jobs and memory size are increasing, the averaging turnaround time of jobs are decreasing clearly. When jobs numbers become larger, data hot-spot will emerge .If we not use copying strategy to disperse jobs to other nodes, massive jobs will be suspended for waiting enough resources, thus increasing averaging turnaround time. And with the mounting memory size, the averaging turnaround time represents a drastic decline, because more data are stored in memory that decrease the data exchange between memory and disk, in other words, lessen the IO times so as to accelerate reading speed directly from memory instead of disk. At the same time, due to adopt cache scheduler policy, hot-spot data can be distributed to other nodes, reducing the large number of jobs' waiting time, which is one of the factors to reduce turnaround time.

Figure 8 and 9 shows the decline rate of averaging turnaround time. While lots of jobs are pending, the improved speed of turnaround time goes downhill quickly but the mode with copying cache steadies the improved speed.

6. Conclusion and Future Work

In this paper, we present a new Memory MapReduce architecture and better task execution workflow. By analyzing the cause slower job execution, we propose a solution based on the distributed cache system, effectively reducing the IO performance bottleneck during job process. In the end, we use turnaround time as the performance indicator to access out solution. Performance tests showed that, compared with the traditional MapReduce job execution, the improved MapReduce jobs improved significantly reduced in the averaging turnaround time

The future work mainly consists of persistent problems of memory data. In this article, since the large amount of data stored in memory, not written to disk in real time, so cached data may be lost due to system crash down. We would introduce log system to assure system reliability.

References

[1] Bu Y, Howe B, Balazinska M, Ernst MD. *HaLoop: efficient iterative data processing on large clusters.* Proc VLDB Endow Proceedings of the VLDB Endowment. 2010; 3(1-2): 285–96.
[2] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM Commun ACM*. 2008; 51(1): 107.
[3] Guangrui, Li. The Research Of Memory Data Caching Technology In MapReduce Massive Data Processing Platform. Beijing: Beijing University of Technology. 2013.
[4] Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R. *Incoop: MapReduce for incremental computations.* Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11. 2011;
[5] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S-H, Qiu J, et al. *Twister: a runtime for iterative mapreduce.* Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10. 2010;
[6] Zaharia M, Chowdhury M, Franklin MJ, et al. Spark: cluster computing with working sets. *HotCloud*, 2010; 10: 10-10.
[7] Wadkar S, Siddalingaiah M. Advanced MapReduce Development. Pro Apache Hadoop. 2014; 107-50.
[8] Li L, Tang Z, Li R, Yang L. *New improvement of the Hadoop relevant data locality scheduling algorithm based on LATE.* 2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC). 2011.
[9] Ghemawat S, Gobioff H, Leung ST. *The Google file system.* Proceedings of the nineteenth ACM symposium on Operating systems principles-SOSP '03. 2003;
[10] Bezerra A, Hernández P, Espinosa A, Moure JC. *Job scheduling for optimizing data locality in Hadoop clusters.* Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13. 2013;
[11] Zhao H, Yang S, Fan H, Chen Z, Xu J. An Efficiency-Aware Scheduling for Data-Intensive Computations on MapReduce Clusters. *IEICE Transactions on Information and Systems.* 2013; E96.D(12): 2654–62.

[12] Mansurova M, Akhmed-Zaki D, Shomanov A, Matkerim B, Alimzhanov E. *Iterative Mapreduce MPI Oil Reservoir Simulator.* Proceedings of the 10th International Conference on Software Engineering and Applications. 2015.
[13] Tao Y, Zhang Q, Shi L, Chen P. *Job Scheduling Optimization for Multi-user MapReduce Clusters.* 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming. 2011.
[14] Xu L. *MapReduce Framework Optimization via Performance Modeling*. IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. 2012.