ISSN 2348 – 8034
Impact Factor- 3.155

GLOBAL JOURNAL OF ENGINEERING SCIENCE AND RESEARCHES

"HADOOP: OPTIMAL UTILIZATION OF CACHE"

Rupali V.Pashte*1 and Ritesh Thakur2
*[1]M.E(Computer), Computer Department, Institute Of Knowledge COE, Savitribai Phule Pune University, Pimple Jagtap,Pune,Maharashtra,India.
[2]Assistant Professor, Computer Department, Institute Of Knowledge COE, Savitribai Phule Pune University, Pimple Jagtap,Pune,Maharashtra,India.

ABSTRACT

Data is being generated at an enormous rate, due to online activities and use of resources related to computing. To access and handle such enormous amount of data spread, distributed systems is an efficient mechanism. One such widely used distributed file system is Hadoop distributed file system (HDFS). HDFS follows a cluster approach in order to store huge amounts of data; it is scalable and works on low commodity. It uses MapReduce framework to perform analysis and carry computations parallel on these large data sets. Hadoop follows the master/slave architecture decoupling system metadata and application data where metadata is stored on dedicated server NameNode and application data on DataNodes, Processing of MapReduce is slow whereas, It is known that accessing data from cache is much faster as compared to disk access. Mutual caching is one such mechanism in which the cache distributed over the clients or dedicated servers or storage devices form a single cache to serve the requests. This mechanism helps in improving the performance, reducing access latency and increasing the throughput.

In order to enhance and improve the performance of MapReduce, the thesis proposes solution of new design of HDFS by introducing caching references, Mutual caching along with customize Adaptive Replacement Cache Algorithm. Each of the Data Nodes would have a dedicated Cache Manager to maintain information about its local cache, remote caches and follow cache replacement algorithm. Customize-ARC helps in organizing the cache in a different way as recent, frequent and history of evicted items which is a better cache replacement policy and improves the execution time and performance.

Keywords: *Hadoop, HDFS, Hadoop, Customize ARC, Cache.*

I. INTRODUCTION

Technological advancement has led to introduction of lots of computing related resources and with advent use of these applications online has led to the production of enormous amount of data. A challenge involves faster retrieval of these resources along with high performance. An effective mechanism to achieve this goal is the use of distributed systems. Distributed systems are fault tolerant, scalable and should provide high availability which is achieved through clustered approach of these systems.

Hadoop is one such storage system following clustered approach that stores large data sets, is highly fault tolerant and has large throughput. It runs on commodity hardware and its key feature being separation of metadata from actual data. Hadoop's file system architecture and data computational paradigm has been inspired by Google File System and Google's MapReduce [3]. MapReduce paradigm [20] helps us perform analysis, transformations and computations on these massive amounts of data. Over the years, Hadoop has gained importance because of its scalability, reliability, high throughput, analysis and large computations on these massive amounts of data. It is being used by all the leading industries like the Amazon, Google, Facebook, and Yahoo. As data storage has crossed over 100PB [3].

A. Problem Statement

Now a day's use of huge online applications has been increased leading to the production of excessive amount of data. A challenge involves faster retrieval of these resources along with high performance. Hadoop, which is widely used and popular with its MapReduce framework to analysis and compute large amount of data, but processing and execution of MapReduce framework, is slow.

ISSN 2348 – 8034
Impact Factor- 3.155

Caching data in such a scenario will help improve the execution time of the task hence improving the performance. Mutual caching together with Customize-ARC defines better way of organizing cache by tracking the recent, frequent and evicted items.

B. Motivation

It is known fact since years that accessing data from memory or cache is faster as compared to disk I/O's. For these data-intensive computations, focus has been more on processing of data rather than speeding up the process. Hence disk access has not been taken into consideration. According to, the initial phase which is map phase in MapReduce involves reading raw data from the disk and this task is I/O intensive. Caching data in such a scenario will help improve the execution time of the task hence improving the performance. A cache hit would lead to faster processing of the request, but a cache miss would lead to disk access. Instead of disk I/O if we retrieve the required data from the neighboring caches or remote memory, this mechanism is called mutual caching. Caches of all the participating machines taken together form a single cache or global cache and requests are satisfied from the remote caches instead of local disk access. It helps to improve the overall performance of the distributed systems and overcome the latency problem caused by the slower disk I/O. Mutual caching helps in further enhancing the performance of the system.

II. LITERATURE SURVEY

An observation regarding Hadoop applications is that they generate and store a large amount of intermediate data [1], and this abundant information is thrown away after the processing vanishes. Inspired by this observation, Yaxiong Zhao et al proposed a data aware cache framework for big data applications, which they called Dache. In Dache, tasks submit their intermediate results to the cache manager. As, intermediate data is very big and have to store that abundant data into the limited sized cache memory. We can use different protocols and different customized indexing in cache memory. To store the newly created cache items old cache items need to be deleted as storing capacity of the cache memory not enough.

Jeffrey Dean et al, in MapReduce Simplified Data Processing on Large Clusters designed a MapReduce program that can be automatically paralyzed and executed on large cluster of the machines [2].As this model is easy to use even for programmers dont have experience with parallel and distributed systems and divided jobs are well balanced using load balancing technique. In MapReduce model of the Jeffrey Dean and Sanjay Ghemawat cache memory is not taking in consideration hence computation overhead increases. Daniel Peng et al. in Large-scale Incremental Processing Using Distributed Transactions and Notifications [3] proposed, a system for incrementally process sing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, Author process the same number of documents per day. Weikuan Yu et al. in Design and Evaluation of Network-Leviated Merge for Hadoop Acceleration [4] proposed, Hadoop-A, an acceleration framework that optimizes Hadoop with plugin components for fast data movement, overcoming the existing limitations. A novel network-levitated merge algorithm is introduced to merge data without repetition and disk access. In addition, a full pipeline is designed to overlap the shuffle, merge and reduce phases. Our experimental results show that Hadoop-A significantly speeds up data movement in MapReduce and doubles the throughput of Hadoop. Jiong Xie et al.in Improving Mapreduce Performance Through Data Placement in Heterogeneous Hadoop Cluster[5],proposed that ignoring the data locality issue in heterogeneous environments can noticeably reduce the MapReduce performance. In this paper, author addresses the problem of how to place data across nodes in a way that each node has a balanced data processing load. Given a data intensive application running on a Hadoop MapReduce cluster, our data placement scheme adaptively balances the amount of data stored in each node to achieve improved data-processing performance. Experimental results on two real data-intensive applications show that our data placement strategy can always improve the MapReduce performance by rebalancing data across nodes before performing a data-intensive application in a heterogeneous Hadoop cluster.

Zhenhua Guo et al.in Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization[6] proposed , Benefit Aware Speculative Execution which predicts the benefit of launching new speculative tasks and greatly eliminates unnecessary runs of speculative tasks. Finally, MapReduce is mainly optimized for homogeneous environments and its inefficiency in heterogeneous network environments has been observed in their experiments. Author investigates network heterogeneity aware scheduling of both map and reduces tasks. Overall, the goal is to enhance Hadoop to cope with significant system heterogeneity and improve resource utilization. Chuncong Xu et. al in, Using Memcached to Promote Read Throughput in Massive Small-File Storage

ISSN 2348 – 8034
Impact Factor- 3.155

System[7] proposed a method of memcached buffer space management based on prioritized cache. And we integrated the memcached into the distributed cache as part of a distributed storage system, which achieved high performance for small file reading. The main aim defined is caching mechanism having Customize replication policy, allowing concurrent access to the data and algorithms relating to locality of data which focuses on the decrease in the overall job completion time. Their caching mechanism is based on Memcached [8] which are a set servers storing the mapping of block-id to data node-ids. These servers also serve the remote cache requests. The caching of blocks is carried out on DataNodes,a request for block. At this point simultaneous request is sent to NameNode and Memcached servers. Reply is sent to the DataNode by NameNode as well as the Memcached servers, once both do lookups for the requested block. If DataNode receives a reply true from MemCache then block is accessed from cache, else block is accessed from the disk whose location is indicated by the NameNode. In the second design architecture, again DataNode request is sent to NameNode as well Memcached. But this time NameNode does not reply although it performs a lookup for the block. Memcached does a lookup too and replies back to DataNode. If Memcached does not find the block then it replies to both NameNode as well as DataNode. In such a case, NameNode sends the block locations to DataNode. Whenever a request for block is seen by Memcached, lookup is performed for neighboring blocks as well. If neighboring blocks are missing then Memcached requests NameNode to look for replicas and if available, corresponding DataNodes are requested to cache those blocks and accordingly Memcached updates its locations. MapReduce works on multiple jobs parallel and produces the different intermediate result. These intermediate data is very big in size and storing at customized index of the cache memory. As the size of the cache memory is limited, so we need to replace the cache index data by the newly created intermediate data for that we are going to be use extended LRU replacement algorithm i.e. adaptive replacement cache algorithm with better performance to optimal use of cache storage.

An adaptive replacement cache algorithm [9] that keeps track of both recently used and frequently used pages with history of the deleted pages for both. In LRU, it stores only entry for last recently used pages. New entry of the page is added to the top of the list after last entry has been deleted. If cache hit then it move to the top by pushing all entries in the list down. ARC is used to improve the LRU strategy by dividing cache list into two sub-lists, t1 for recently used and t2 for frequently used entries. These two lists are extended with a ghost list b1, and b2 which is attached to the bottom of the list respectively. These two extended list are used to keeping track of the recently deleted or evicted cache entries. New entries are added to the list t1 and page hits are added to the list t2. If the hit is occur in the b1 then size of the list t1will increase by pushing to the right and last entry of the list t2 is pushed into the list b2. If hit is occur in the list b2 then t1 will shrink. So using this it wills not loss any valuable block, hence it achieve better performance in of system.

III. EXISTING SYSTEM

Big-data refers to the large-scale distributed data processing applications that operate on large amounts of data. Google's MapReduce and Apache's Hadoop, its open-source implementation, are the software systems for big-data applications. An observation of the MapReduce framework is that the framework generates a large amount of intermediate data. Such abundant information is thrown away after the tasks finish, because MapReduce is unable to utilize them. In this paper, in existing system Dache, a data-aware cache framework for big-data applications. In Dache, tasks submit their intermediate results to the cache manager. A task queries the cache manager before executing the actual computing work.

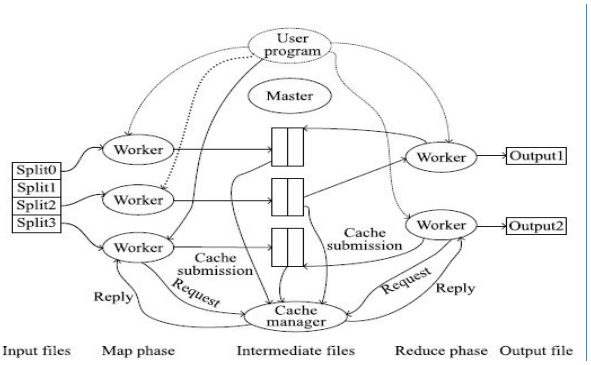

Figure 1. Data Aware caching

ISSN 2348 – 8034
Impact Factor- 3.155

A. Drawbacks

1. Performance and Availability:

NameNode is a dedicated server that maintains the namespace and carries out all the operations such as mapping of blocks, block operations, maintaining namespace, serving client requests, instructing DataNodes. With addition of more DataNodes, a threshold would arise beyond which the performance of NameNode starts degrading. Since NameNode is a single point of failure, if NameNode is down, the whole filesystem has to be taken offline. To restore NameNode, Secondary Name Node's fsimage and edit log are used which are checkpoint files.

2. Scalability

HDFS keeps all the namespace and block locations in memory but the heap size of NameNode is limited. Also the entire cluster depends on the dedicated server NameNode for all its operations and namespace. Increasing DataNodes is directly proportional to the increase of workload. To add to it, overall memory size of NameNode is limited. Taking into consideration all these factors scalability of NameNode is limited.

IV. PROPOSED SYSTEM

A. System Architecture

The proposed architecture, algorithm and reason that led proposed architecture figure 2. help in achieving the desired output of lowering the overall MapReduce job execution time. There has been change in the architecture as mentioned in my proposal and a new architecture is adopted. For sections below where changes have been made, a description is provided along with reason of this new architecture and modifications. Functionality of mutual caching on DataNode a new process of Cache Manager has been introduced into DataNode. There are other additional functionalities on the components; DFSClient and the NameNode.

- Cache Manager

Each of the DataNode have their dedicated Cache Managers who have responsibilities of managing caches, lookup in local as well as global cache image upon request, replacement policy for cache, eviction policy for cache when cache is fully utilized. Each of the responsibilities are explained in detail as follows

- Caching

The local cache is divided into four sections namely recent cache, frequent cache, recent history and frequent history. Logically we can say two cache sections where recent cache and recent history as one cache and frequent cache and frequent history as other. Total Local cache is comprised recent cache as well as frequent cache. As seen the diagram 5.1, each of the Cache Managers have their own local cache which is recent + frequent. DN1's local cache has just block A, where DN2's local cache has blocks A and block C.

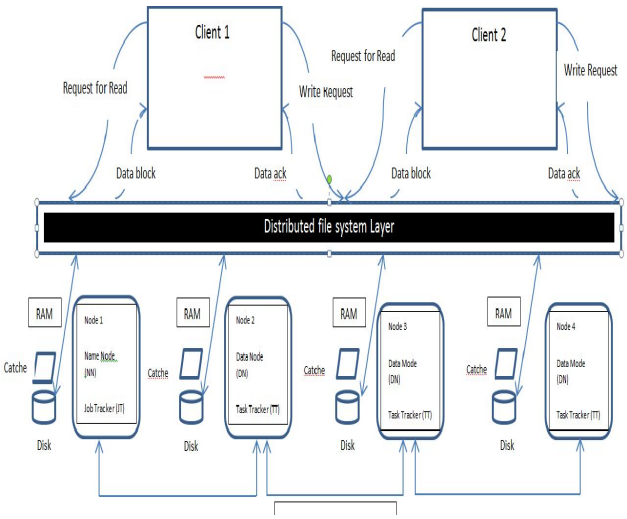

Figure 2. Proposed Architecture

ISSN 2348 – 8034
Impact Factor- 3.155

Replacement A cached object will be replaced when the cache is fully utilized. There are three sets of replacement policies.

- Whenever a new block arrives to be cached in the recent cache and the cache is fully utilized, in such case the LRU block is evicted from the cache, but its reference is placed in the recent history.
- A new block to be cached in the frequent cache and the frequent cache is fully utilized, in such a case the LRU block is evicted and its reference is placed in the frequent history.
- A fully utilized recent history or frequent history cache causes eviction of the references completely.

LRU cache eviction policy will be used for each of the individual caches which is recent, frequent, recent history, frequent history.

- Global Cache Image

Global Cache Image. Global Cache Image lookup is done by Cache Manager upon local cache miss. RemoteCache list is also named as global cache image (GCI), as seen in the diagram comprises of mapping of block to DataNodes. This denotes that this block is cached on which DataNodes in the cluster. This mapping is maintained by NameNode and copy of it is sent to all the DataNodes as a response to cache block report. As shown in diagram 5.1, each of the DataNodes have GCI and it indicates that if we are at DN3 and we want to perform a look for block A, then it is cached at DN1 and DN2.

- Block Forwarding

CacheManager does have the support for Block Forwarding. The reason this was discarded because streaming data across the network takes time and with huge block size, it reduces the overall performance of the system. Cache Manager Responsibilities It does not take care of singles, since that would involve Block Forwarding and Block Forwarding is not considered due to performance reasons.

- Replacement

The entries are evicted as defined in proposal but the cache is not adaptable to increase the size of the individual caches. Since with the current simple design which involves taking the idea of ARC.

- Eviction

Each of the individual caches in the Customize-ARC caches has LRU. The reason this has been chosen was, to keep the system more efficient and avoid lots of lookups.

- Remote Cache List

The reason previous remote cache list also known as Global Cache Image was updated, due to, two reasons. First reason, everything in the Hadoop system deals with block, hence using block as the key helps in better searching. Moreover, total local cache is comprised of recent and frequent hence we need not differentiate it into recent and frequent. There is no block forwarding due to extra time incurred by forwarding the block over the network. This data structure helps NameNode to better maintain Global Cache Image without extra cost.

- NameNode Management

NameNode is the central co-coordinator for maintaining the Global Cache Image. It builds its global cache image when it obtains the cached block report from the DataNodes. As soon as it obtains its report, it updates the mapping of cached block to DataNode. Upon updation of the Global Cache Image, as response to cached block report it sends a copy of Global Cache Image to the DataNode.

- DFSClient

DFSClient has to contact NameNode for getting the block locations for a particular file. Hence to this request along with non cached block locations, cached block locations are also provided as a response. If it has received the cached block locations, then obtained list is sorted by the DFSClient so as to read from cached block.

ISSN 2348 – 8034
Impact Factor- 3.155

- DataNode

Data Node provides a cached block report of its local cache to NameNode after periodic interval. As a response to this report NameNode commands it to update global cache image on DataNode.

B. Proposed Algorithm

Mutual Caching on Hadoop An approach proposed in order to improve the overall performance of execution of the MapReduce jobs. Mutual Caching takes place on the DataNodes with the help of Cache Managers. Here references refer to and block. A Customize-ARC algorithm is used as caching policy. Simple mutual scheme is used, if there is local cache miss, then data node notices DFSClient of next cached data node to connect if any.

- Customize-ARC Algorithm

Fig 3 shows the diagram of a Customize-ARC. A variant of this algorithm is implemented. Basic idea is to divide the caches into two different sections namely cached objects and history objects. Cached section contains the actual data and History section contains the references. Hence the cached section is further divided into Recent Cache and its Recent History and Frequent Cache and its Frequent History.
Recent Cache: A Recent cache is a cache where the block seen for the first time is placed.

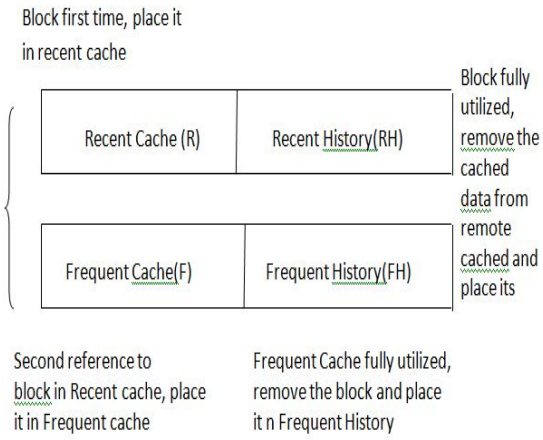

Figure 3. Customize -ARC.

Frequent Cache: A second reference to the same block will cause the block to be placed in this

- Variant of Customize-ARC Algorithm

The size of recent and frequent together is fixed. The basic idea behind this is,
STEP1. Initially on a request for block, check if the references any of the history caches. If references not found in any of the history caches then cache them, if references found in either of the history caches then place the blocks in their corresponding caches (recent or frequent cache) and then blocks can be served from those caches.
STEP 2. Use these references to cache the block if not already cached and since it is the first time, place it in recent cache. If in Step 1, found in recent history cache then place in the recent cache or else if found in the frequent history cache, then place in the frequent cache. Caching of blocks involves caching metadata for the block as well as the data for the block.
STEP 3. A next hit on the same block places the block into frequent cache.

STEP 4. When either of the caches is fully utilized then block is evicted from recent or frequent cache but its reference is placed into its corresponding history. When either of the history caches is fully utilized causes the references to just drop out of the cache.
STEP 5. A hit in either recent history or frequent history removes the reference from history and places the data in the cache. Simultaneously in the same turn, the references are served from the history as well since there is a hit in the history cache. So any hit in the history and data not in cache, initially serve from the references of history cache.

ISSN 2348 – 8034
Impact Factor- 3.155

C. Advantages of Proposed system

1. The overall cache hit rate increases.
2. Any hit in the history cache causes the request to be served by references.

D. Differences between new Customize-ARC and previous ARC

The difference between current Customize-ARC implementation and ARC is that there is no focus was in Hadoop framework and with this simple Customize-ARC implementation helped in achieving the performance.

V. SYSTEM DESIGN

A. Mathematical Models

Set (S) = {I, O, Fn, S, F}

I - The input as text File.
I={PT1, T2……T20}

O - The Output DataNode Location and BlockId.

Fn - Set of Processing functions required to generate output.

MapReduce – (M)
M= {input text file split, shuffle, reduce, Key (k), value (v)}

GCI (G)-Global Cache image

G= {Datanode Location (dl), BlockId (b)}

Customize-ARC (A)
A= {Recent Cache (RC), Recent History (RH), Frequent Cache (FC), Frequent History (FH)}

S - The set of success cases.

S= {BlockId, DataNodeLocation }
Where Block Found in Cache.

F - The set of failure cases.
F= {BlockId, DataNode Location} Where Block Found in HDFS.

- Vein Diagram:-

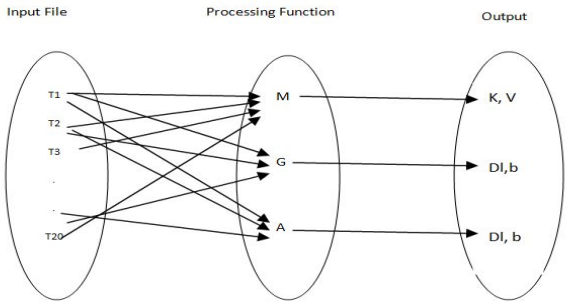

Figure: 4 Vein Diagram for system

ISSN 2348 – 8034
Impact Factor- 3.155

VI. COMPARISON OF EXISTING AND PROPOSED SYSTEM

- Comparison Of Existing And Proposed System

File Name	MapReduce (ms)	Distributed Cache(ms)	Customize -ARC(ms)
a1.txt	12310	12744	3630
a2.txt	18710	10988	5557
a3.txt	25946	10261	4775
a3.txt	15398	10213	5836

Table1. Execution time analysis using 3 techniques

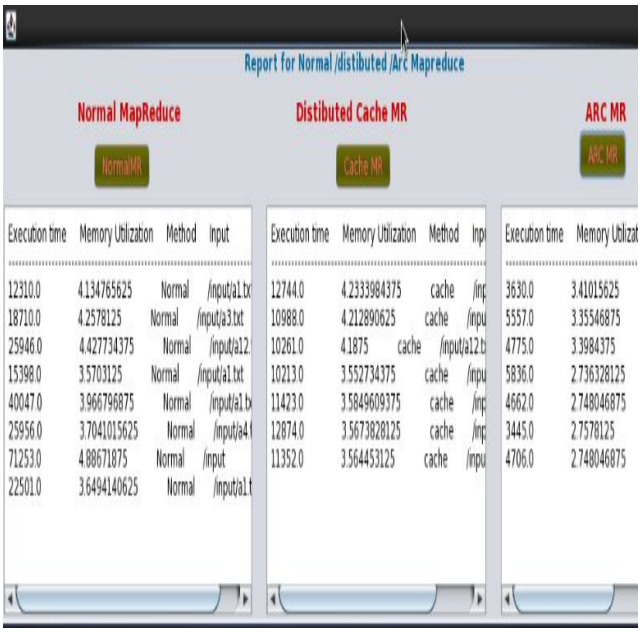

Figure 5 Comparison Report by system

VII. RESULTS AND DISCUSSION

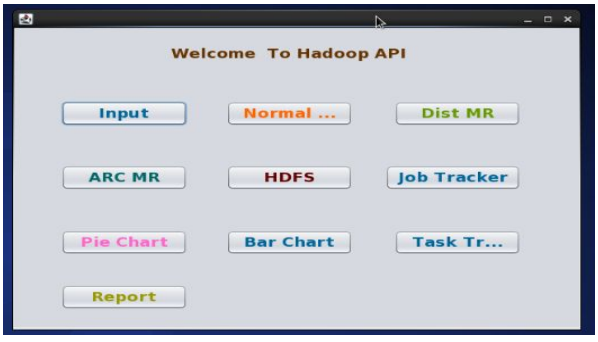

Figure 6. User Interface of System

ISSN 2348 – 8034
Impact Factor- 3.155

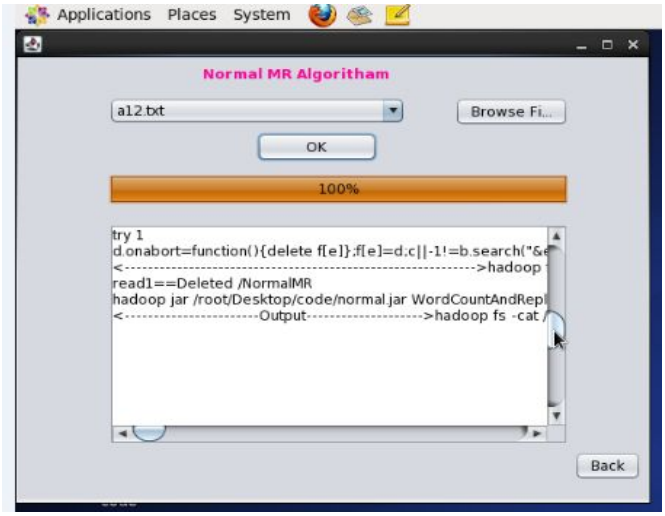

Figure 7. Job Processing Using MapReduce Algorithm

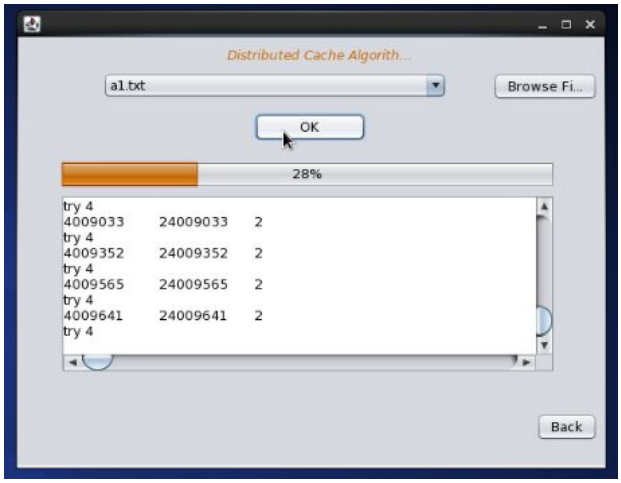

Figure 8. Job Processing Using Distributed MapReduce Algorithm

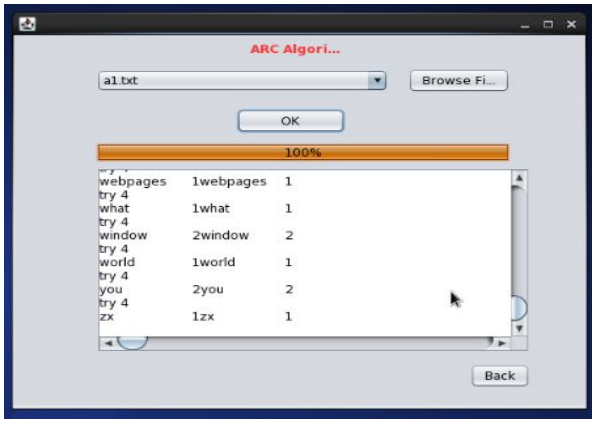

Figure 9. Job processing Using Customize-ARC Algorithm

ISSN 2348 – 8034
Impact Factor- 3.155

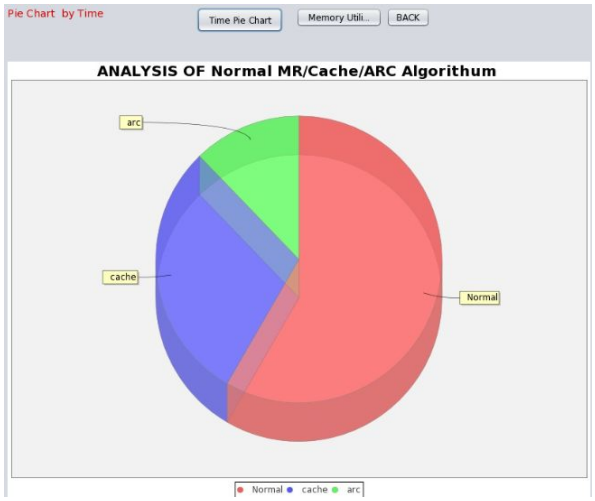

Figure 10. Pie chart analysis for Execution time

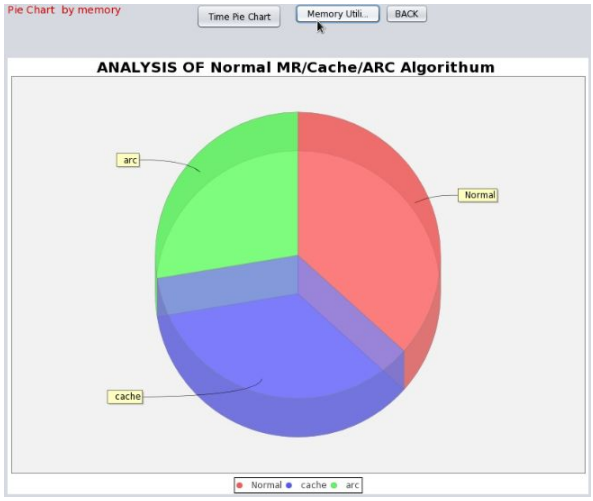

Figure 11. Pie chart analysis for Memory Utilization

ISSN 2348 – 8034
Impact Factor- 3.155

Figure 12. Bar chart comparison for Execution Time for methods.

VIII. CONCLUSION

A new architecture, which is named as Hadoop-(Mutual Caching). The architecture aimed at improving the overall MapReduce job execution time and efficiency of the system. This was done through the mechanism of mutual caching where data is served from local caches as well as remote caching. Name Node's response to client was Customize to send cached locations as well to the non cached locations of DataNode. NameNode maintains the Global Cache Image which is image of location of all the cached blocks on the cluster. This is mapping of cached block to DataNodes indicating this block is cached on these DataNodes. On the DataNode side, each of them has their dedicated cache managers which have responsibilities of caching data, replacement policy which is Customize-ARC. The cache was divided into four sections recent, recent history, frequent, frequent history instead of maintaining just a single cache. This mechanism helped in better caching replacement. Hence overall, the job execution time decreased by a considerable amount, efficiency of the system increased and there were more data-local jobs scheduled. Both theoretical and experimental results have shown that performance has been retained.

IX. ACKNOWLEDGEMENTS

Author would like to take this opportunity to express our profound gratitude and deep regard to my guide Prof. Ritesh Thakur, for his exemplary guidance, valuable feedback and constant encouragement throughout the duration of the project. His valuable suggestions were of immense help throughout my project work. His perceptive criticism kept me working to make this project in a much better way. Working under him was an extremely knowledgeable experience for me.

REFERENCES

1. *Yaxiong Zhao, Jie Wu, and Cong Liu, "Dache: A Data Aware Caching for Big- Data Applications Using the MapReduce Framework, TSINGHUA SCIENCE AND TECHNOLOGY ISSN 1007-0214 05/10 Volume 19, Number 1, pp 39- 50, February.*
2. *"MapReduce Simplified Data Processing on Large Clusters", Google, 2004.*
3. *D. Peng and f. Dabek,"Large Scale incremental Processing using distributed Transaction and notification", in Proc. of OSDI'2010, Berkeley, CA, USA, 2010.*
4. *M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments", in Proc. of OSDI2008, Berkeley.*

5. *M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving Mapreduce performance in heterogeneous environments", in Proc. of OSDI' 2008, Berkeley, CA, USA, 2008.*
6. *Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, "Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters", Department of Computer Science and Software Engineering Auburn University, Auburn, AL 36849-5347.*
7. *Zhenhua Guo, Geoffrey Fox "Improving MapReduce performance in heterogeneous Network Environments and Resource Utilization" School of Informatics and Computing Indiana University Bloomington, IN USA.*
8. *Gurmeet Singh, Puneet Chandra, Rashid Tahir, "A Dynamic Caching Mechanism for Hadoop using Memcached", 2010.*
9. *Zhenhua Guo, Geoffrey Fox "Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization "School of Informatics and Computing Indiana University Bloomington, IN USA.*
10. *"Outperforming LRU with an adaptive replacement cache algorithm", Megiddo, Nimrod, IBM Almaden Res. Center, San Jose, CA, USA, pp.58-65.*
11. *Cache Algorithms, http://en.wikipedia.org/wiki/Cache algorithms, 2013.*
12. *http//developer.yahoo.com/hadoop/tutorial/module4.html*
13. *http//hadoop.apache.org/releases.html*
14. *Eric Anderson et.al. "New Algorithms for File System Cooperative Caching", Modeling,*
15. *Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium, pp.437-440.*
16. *http//bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-thenetwork.*
17. *http//hadoop.apache.org/docs/r0.20.2/quickstart.html#Local.*