A Novel Approach for Data Intensive Caching for Big Data Application Using Hadoop Framework

Shakil B. Tamboli, Student of MEIT,
Department of Information technology,
Smt.Kashibai Navale College of Engineering, Pune.
Email:stamboli7@gmail.com Mobile: 9850144360

Smita Shukla Patel, Assistant Professor,
Department of Information technology,
Smt.Kashibai Navale College of Engineering, Pune.
Email: smshuklapatel@sinhgad.edu

***Abstract*— Big Data technologies and analysis tools are applied for immediately needed business outputs, change business into easy IT architecture, provisioning real time openings, rapidly identify risks, increase predictive capabilities, modernize operations and expose new income ways, provide complete and accurate information to any application or user with active data management from a variability of new and developing data sources. Due to this academicians & IT industry societies are attentive to mine fascinating features of big data.**

While processing a large amount of data, framework produces big amount of intermediary data and such ample data is not used after the task is over as jobs are independent. But if there is an incremental processing of the same application then there might be potential duplicate computations being performed. Therefore with the help of cache mechanism we can accelerate such duplicate job execution computations and identify, access them as cache items in a MapReduce job to speedup operations. Observations from some existing cache systems over incremental computations on massive data sets are time consuming, wasting input output, network bandwidth, assets like CPU, memory as well as such systems are very complex in design, difficult to execute, understand and does not have expected improvements as demanded and expected by users and developers of such systems.

So to process an efficiently incremental computation with the help of intermediately generated big data, this work recommends a cache based methodology to improve the performance of incremental computations by consuming internally generated large data on MapReduce platform. It is achieved by caching the preprocessed results earlier than executing the incremental computations. The preprocessed results are divided into diverse parts which are cached on various nodes in the cluster. The expected results for the recommended methodology are shown to appreciate the better performance.

Keywords—MapReduce, Hadoop, Caching, Distributed File System, Massive data sets

I. INTRODUCTION

Now a days Big Data or data science buzz words are read and discussed lot and many organizations have already working on extracting potentials hidden behind these terms. Big data technologies and analysis tools are applied for immediately needed business outputs, change business into easy IT architecture, provisioning real time openings, rapidly identify risks, increase predictive capabilities, modernize operations and expose new income ways, provide complete and accurate information to any application or user with active data management from a variability of new and developing data sources, comprising social media data, locality data created by smartphones and other traveling devices, community information available online and data from sensors implanted in cars, houses and other objects and like ample things. Analysts apply 3V model to describe Big Data. Volume denotes to the fact that Big Data comprises analyzing comparatively enormous amounts of information, normally starting at tens of terabytes. Velocity imitates the sheer speed at which this data is generated and modified. Variety refers to the fact that Big Data can come from many different sources, in various formats and structures. As social media sites and networks of sensors produce a stream of regular changing data which might include geographical information in the form of text, images, videos and audio.

Google's MapReduce is commanding, easy programming interface and outstanding performance framework useful for executing an enormous range of applications. Application developers plan the computational work in the form of a map and reduce functions and the ultimate MapReduce job organizing method designed parallelizes the computation through a cluster of machines. As soon as record is created in the map phase, in between results are shuffled and sorted by the MapReduce system and are then transformed into the workers in the reduce phase. At the end; results are executed by multiple reducers and recorded to the disk [1]. The scheme of the MapReduce framework is influenced by major principles as Low cost consistent commodity hardware, exceptionally scalable RAIN cluster, fault tolerant but simple to manage, highly parallel but abstracted nature. The example of MapReduce execution flow is shown in fig.1.

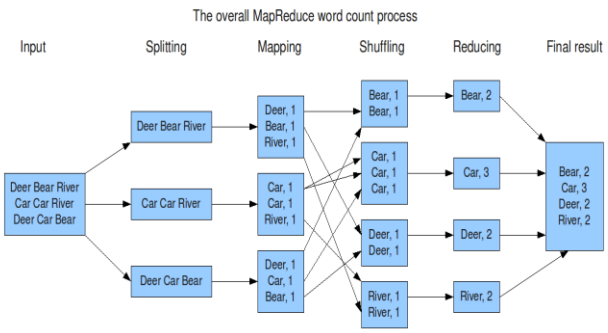

Fig.1 Example of MapReduce execution flow

Apache's Hadoop is one more significant open source software project technology used to manage big data using their analytics and data processing techniques. A small Hadoop cluster has a single master and many worker nodes. The master node accomplishes various functions like JobTracker and NameNode. The JobTracker is responsible for managing consecutively jobs in the Hadoop cluster. The NameNode, manages the HDFS. The JobTracker and NameNode are normally collocated on the same physical machine. Other servers in the cluster run a TaskTracker and DataNode processes. A MapReduce job is divided into tasks. Tasks are managed by the TaskTracker. The TaskTrackers and DataNode are collated on the same servers to provide data locality in computation [1].

The above tools processes large amount of data and generates intermediate big amount of data as well but not utilize such internally generated data as jobs are processed independently and acyclic graphs in nature. But whenever same application is processed again with small changes in data then there might be potential duplicate computations being performed. Therefore with the help of cache mechanism we can accelerate such duplicate job execution computations and identify, access them as cache items in a MapReduce job to speedup operations.

So to process incremental computations using internally generated enormous data efficiently, this work proposes a cache based methodology to improve the performance of incremental computations by using internally generated large data on MapReduce platform. This methodology improves the performance of processing incremental computations on MapReduce platform by caching the preprocessed results before accomplishing the incremental computations. The preprocessed results are divided into various parts which are cached on various nodes in the cluster.

The remaining paper is prepared as follows. The related work is presented in section II. Section III presents proposed system and implementation details for efficient data aware caching mechanism for big data application using Hadoop framework. Section IV contains experimental settings details. Section V highlights results and discussion predicted and conclusion is given in section VI.

II. RELATED WORK

A. *Literature Review*

The aim of comprehensive survey on incremental compuation is to optimize and minimize compuatational time and reduce memory overhead, IO bandwidth and CPU resources.

In this paper [2] run times is not controlled by application code in map and reduce. The extra load is of library itself on commodity multicore computer. Tiled-MapReduce (TMR) [3] is further operational model for MapReduce to iteratively process small pieces of data so as to process an enormous amount of data at one time on shared memory multicore platforms.

In[4] to reduce the gap in disk access time and bandwidth for big cluster based systems, the hard work is done to design a proactive fetching and caching mechanism based on Memcached distributed caching system and integrated with Hadoop. The Incoop system [5] allows the automatic update of the outputs of the MapReduce jobs by providing a fine grained output recycle mechanism.

In [6] recommended Efficient and Flexible index operated technique to better support big data applications. EFind gathers index statistics and completes cost based adaptive optimization to improve index operated technique based performance. In [7] XTrie and ETrie developed partitioning techniques to increase load balancing for distributed applications.

The HaLoop system [8] developed new task scheduler to leverage data locality. It has also caches and indices application data on slave nodes. The paper [9] allows input output devices to effectively leverage the high bi section bandwidth and speed interconnect of massively parallel high end computing systems. In [10] the Redoop infrastructure is validated. Redoop presents Window Semantic Analyzer for optimization, Dynamic Data Packer as partition executor, Execution Profiler gathers the statistics, Local Cache Manager and Window Aware Cache Controller to maintain window aware metadata of reduce input and output data which is cached on any of the task nodes' local file systems.

In [11] PACMan implemented two cache replacement policies LIFE and LFU-F which are designed to reduce average completion time of jobs and maximize output of the cluster.

Table 1. Findings from literature survey

Sr. No.	Paper theme	Optimization Type	Time completion	CPU overhead	Memory Overhead	Network Bandwidth consumption	Design complexity
1	Dache	Framework	Medium	Medium	Average	High	High
2	Memcached	Distributed caching	Low	High	High	Low	Average
3	Incoop	Framework	Medium	Low	Medium	Low	High
4	HaLoop	Framework	Low	High	High	Medium	High
5	PACMan	Coordinated memory	Low	Average	Average	Medium	Medium
*	Our system	Algorithmic approach	Low	Low	Low	Low	Low

B. *Existing System*

A piece of cached data is saved in a Distributed File System (DFS). The data of a cache item is described by the original data and the operations performed. Normally, a cache item is described by a 2-tuple: Origin, Operation. Origin is the name of a file in the DFS. Operation is a linear list of existing operations performed on the Origin file. As an illustration, in the word count application, individual mapper node or process produces a list of words, summerize the tuples that record the count of each word in the file which the mapper processes. Dache saves this list to a file. This file turns out to be a cache

item. Consider an original input data file, word list xyz.txt, the cache item is well-defined by word list xyz.txt, item count [1].

The input required by the reduce phase is a collection of key value pairs, in which the value would be a collection of values. Similar the scheme is applied for the map phase cache description, the original input and the applied operations are required. The original input is acquired by saving the intermediate results of the map phase in the DFS. The applied operations are identified by unique IDs that are specified by the user.

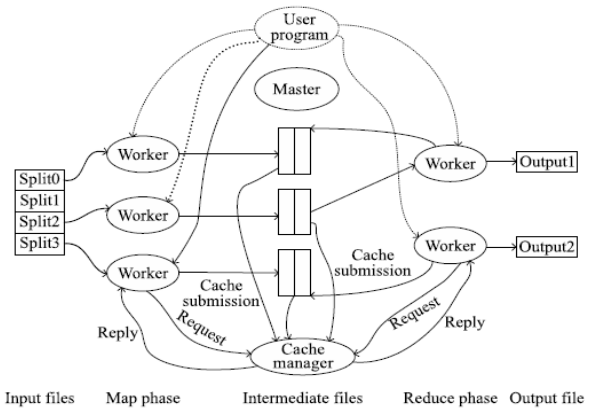

Fig.2 Existing system architecture for Dache

III. PROPOSED SYSTEM AND IMPLEMENTATION DETAILS

A. *System Overview*

Considering above said challenges, objectives and goals, this work focuses on improving the performance of summarized information, incremental computation and aggregate queries against massive data set. We implement our methodology by assigning caches to store the results of summarized information, incremental computations and aggregate queries on different MapReduce cluster nodes. The system architecture of proposed system is shown in Fig. 3

When assigned the caches on the MapReduce cluster's nodes, we execute the preloaded aggregate algorithms on the data saved on the MapReduce. The highest *k* results (like an example, the major *k* greatest numbers in the output of *Count*( ) operation) are carefully chosen on each node in decreasing order as per the predicates of queries and cached on different cluster nodes. It is considered as caches' initialization. Then, we show how to achieve summarized information, incremental computation and aggregate query based on the cached results.

As soon as a MapReduce cluster's node takes a query, the system generally searches the results cached on individual node distributed in the cluster. If the result is found successfully, it will be returned instantly and a message will be given to other nodes. The message communicates other nodes regarding the result has been found, so the search process can be closed. Or else, the system will have to read the data saved on disks and compute online. But, reading data from the disks requires time, I/O operations which will make the whole system incompetent. So to make the system further applicable, we recommend a strategy for replacing the cached data.

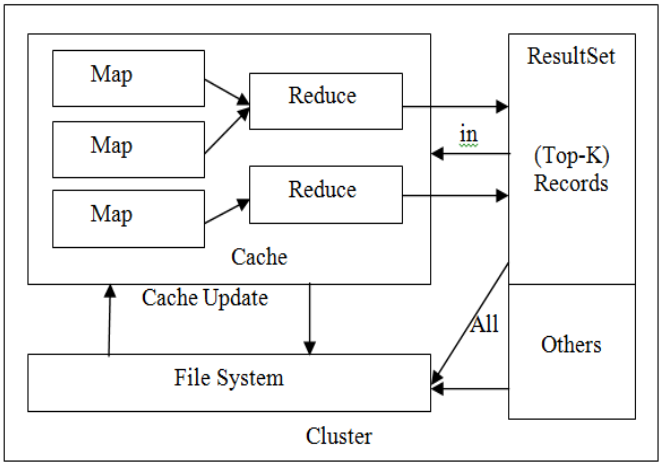

Fig.3 Proposed system architecture

Suppose there is a big size of data already saved in the cluster and distributed on various *Map* data nodes. The cluster has *m* nodes accountable for *Map* tasks and *r* nodes for *Reduce* tasks, and $r<<m$. Hence, there are *n* (where $n=r+m$) nodes in the cluster need to assign the caches. The size of cache would be assigned on a node is subject to the running state of it in the cluster mode. Taking earlier things into consideration, we use the saved data objects with *k* attributes and are indicated as $O(A1, A2, \ldots, Ak)$. As per our knowledge, it does not create logic to execute incremental operations on some attributes. Hence, we construct a list *Alist* to hold the attributes for probably incremented on. When incremental operations are delivered on every attribute, the results will be cached as the initial value.

B. *Algorithm*

Aggregate operations in MapReduce

Take a relational table as an illustration shown in Table 1. Here, assuming a relation $R(A1, A2, \ldots, Ai, \ldots, Ak)$, we generate an attribute set *AList* to record attributes for the aggregate operations so as to meaningfully executed on. Next, aggregate operations are applied on each attribute which consist in AList and the results will be cached on various nodes.

To a provided tuple *t* of the table, we consider data of *t* as the *value*, and *t*'s position as the *key*. Consider that the cluster's *Map* nodes and *Reduce* nodes are organized on various nodes. On *Map* nodes, aggregate operations are run on single node, while *Reduce* nodes join all the results supplied by the *Map* nodes of the complete cluster. In a *Map*( ) function, the attributes that would be aggregated are saved in a list *AList*. We consider two aggregate operations *Count*( ) and *Sum*( ), as an illustrations. In the *Reduce*( ) function, the results are combined to form the final result which is output to the disk.

Table 2. Example Table

Number	Name	Gender	Age	Class
110711	Gopal	Male	21	1
110712	Geeta	Female	22	2
110713	Ramesh	Male	21	1
110714	Santosh	Male	21	2
110715	Rupa	Female	22	2

Consider original data is distributed on various *Map* nodes of MapReduce cluster. As the data is divided into *n* data files, referred as *f1, f2, f3, ... , fn*, and saved on the *Map* nodes *d1, d2, d3, ..., dn*, correspondingly. The *Map*( ) function is presented in Fig.4 Algorithm 1 whereas *Reduce*( ) function is presented in Fig.5 Algorithm 2. For Algorithm 1, as we have knowledge, *AList* is a subset of attributes and its size is much smaller than the number of data tuples. As per the analysis, we know that the time complexity of *Map( )* function is *O* (*n*). Fig.5 shows the *Reduce( )* function in which the output tuples are for *Ai* aggregation attributes and *Vj* values, and its complexity is also *O* (*n*).

Map (key, value)
1) To an individual tuple value, take attributes Ai which need to be aggregated and tuple as AList.
2) To an individual attribute Ai has a value Vij, so form a set named VList.
3) Cij ← 0; Sij ← 0; VList ={}
4) ***While*** *hasNext[AList]* ***is not Empty*** *› per attribute*
5) ***if*** *currentValue* ***is not in*** *VList*
6) VList.addValue(currentValue)
7) ***While*** *hasNext[VList]* ***is not Empty***
8) ***While*** *hasNext[Fi]* ***is not Empty***
9) ***if*** *Vij* ***is not NULL*** *›when value is not null*
10) Cij ← Cij+1 › value is sum
11) ***if*** *Vij* ***is Number*** *›when value is number*
12) Sij ← Sij+Vij ›sum result
13) ***else***
14) Sij ← 0 ›sum result
15) add[OutList] ← <Ai+Vij, Cij, Sij> ›sum result
16) end

Fig.4 Map( ) function Algorithm 1

Using these two developed algorithms, we can receive the simple aggregate results with format as <*RowKey*, *Ci,j*, S*i,j* >, in which *RowKey* is shaped with the attribute and its value, *Ci,j* is count of the tuples whose attribute *Ai* is value *Vj* and *Si,j* is the sum of the analogous attributes of all tuples. But, this is non-trivial to get the results using the above methodology. Hence, we spread the format by adding the tuple ID list Keylist at the end of the final results, that is, <RowKey, Cij, Sij, KeyList>.

Reduce (key,OutList)
1) To an individual row key in the OutList, form a set named RowKeyList.
2) Cij ← 0; Sij ← 0; ›assign count and sum values
3) ***While*** *hasNext[RowKeyList]* ***is not Empty***
4) ***While*** *hasNext[OutList]* ***is not Empty***
5) ***if*** *RowKey==(Ai+Vij)[OutList]*
6) Cij ←Cij+OutList(Cij) › count
7) Sij ←Sij+OutList(Sij) › sum
8) add[OutFile] ←<RowKey, Cij, Sij>
9) end

Fig.5 Reduce( ) function Algorithm 2

.

C. *Mathematical Model of proposed system*

Cache Management: Initializing the Cache

As soon as executing summarized information, incremental computation and aggregate queries with *Map*( ) function on MapReduce, we receive the number of tuples for a certain value of a given attribute in a node files. If the attribute is numeric, the *Sum*() can be done, *i.e.*, *List<Ai+ Vij, Cij ,Sij>*. The return values are organized by their count in descendant order. The top *k* tuples are selected and cached in the cluster, where the value of *k* is analogous to the size of cache. Consider there are *n* assigned cache nodes and the average size of cache on each node is *a*, clearly, the total cache size *S* in the cluster is:

$$S=n*a \quad (1)$$

Let the size of tuple in *List<Ai+ Vij, Cij ,Sij>*be *l* and *k* is computed as :

$$k=S/l \quad (2)$$

Normally, the number of chosen incremental tuples at each node is not more than *k* and the exact number is determined at runtime.

Algorithm for Updating the Cache

Existing known cache updating algorithms only take some single issues into consideration which make them replace the cache in an incompetent way. Therefore, we recommend a methodology to calculate *the value degree Value i* of a cache tuple *i*:

$$Value\ i=Fi\times Ti/(Tci-Tli) \quad (3)$$

where *Fi* is the frequency of tuple *i being accessed*, *Ti* is the delay time of fetching tuple *i* from the disk, *Tci* is current time and *Tli* is the last access time. As per Formula 3, we see that *Value i* increases as the access frequency *Fi* and delay time *Ti* increase and decreases as the interval of fetching data (*Tci-Tli*) *increases*. The below shown updating algorithm is based on the concept of *value degree* formulated in Formula 3.

The data saved on a node modifies endlessly in the real situation, so as to timely cache the aggregated data it requires regularly update the cache. At the time of update, we substitute the last *n* tuples ranking with the *value degree* computed with Formula 3. The last *n* tuples cached on the nodes but seldom used in aggregate computation. These tuples are substituted by some other data stored on the disk. This

operation is executed during the process of periodic update. Fig. 6 describes the process of updating cache data.

```
Algorithm: ALG—update cached tuples
Input:aggregation query Q;
Output:query result R;
  1) UpdateCache(Q)
  2) while hasNext[cache] is not empty
  3)   if key[Q]==key[cache(i)]                 › when find
  4)     R ← cache(i)
  5)     Getback(R)                             › give result to client
  6)     F[cache(i)] ← F[cache(i)]+1            › take freq. add 1
  7)     Tl[cache(i)] ← currentTime             › update access time
  8)     Return                                 ›search finish
  9) while hasNext[file] is not empty
  10)   if key[Q]==key[file(i)]                 › got it
  11)       R ← file(i)
  12)       Getback(R)                          › send result to client
  13)       DeleteFromCache(random(min(Vi)))
  14)       cache(i) ← InsertIntoCache(file(i))
  15)       F[cache(i)] ←1                      ›take frequency add 1
  16)       Tl[cache(i)] ←currentTime
  17)     Return                                ›search finish
  18)   End
```

Fig.6 Algorithm for updating cached data

Maintenance of the Cache Coherency

High fault-tolerance and scalability are the two important features of MapReduce platform. In MapReduce platform, data files are replicated on various nodes, so it is not required to worry regarding the loss of data files when a node is in fail state. But, how to recover the misplaced cached data on a failure node is a serious matter. There is one possible resolution to address this matter is to cache the same data on a backup node and put the backup node into the cluster as a reserve which is achieved in our system, by adding a flag in the aggregation result files to label whether a tuple has already been cached. If a node is in fail state, all we need to do is to find the backup of data files on the failed node in the cluster and re cache it as per the flag. With this approach, the cached data can be recovered in a short time.

IV. EXPERIMENTAL SETTINGS

The Hadoop system is run on intel i3 processor running at 3GHz, 3GB RAM memory and 500GB Harddisk loaded by Ubuntu operating system, Hadoop, Java and Eclipse editor. We use two applications to benchmark speedup of our system over Hadoop i.e. word count and tera sort. Word count counts the number of unique words in large input text files; tera sort sorts key value records based on the lexical order of the key.

V. RESULTS AND DISCUSSION

Fig. 7 and 8 presents the expected completion time of two programs. Data is appended to the input file. The size of the appended data varies and is denoted as percentage amount to the provided input file size, that is 10 GB. Tera sort has CPU destined feature as compared to the application word count, therefore our system bypasses computation tasks which require extra time for the rising size of added data, but our system is intelligent to finish jobs faster than Hadoop in all circumstances. The map phase of tera sort perform very less computation, our system easily works.

Fig. 9 and 10 shows the expected CPU consumption rate of two programs. It is computed by averaging the CPU consumption rate of the processes for MapReduce jobs over time. Tera sort takes extra CPU cycles than word count, that is decided by the CPU based environment of the sorting procedure. When observed from provided results, it is ensured that our system saves a significant amount of CPU cycles, and is illustarted by the much lower CPU consumption rate. Shown results are consistent with Fig. 7 and 8 because bigger incremental size, the CPU consumption rate of our system grows significantly, too. This is because our system needs to process the new data.

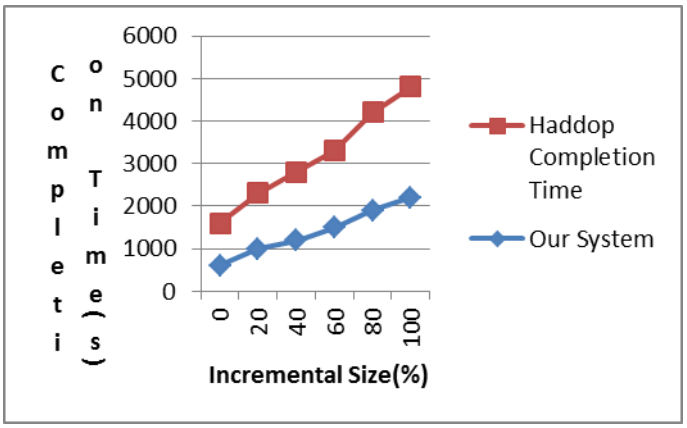

Fig.7 The completion time of our system over Hadoop of word count program

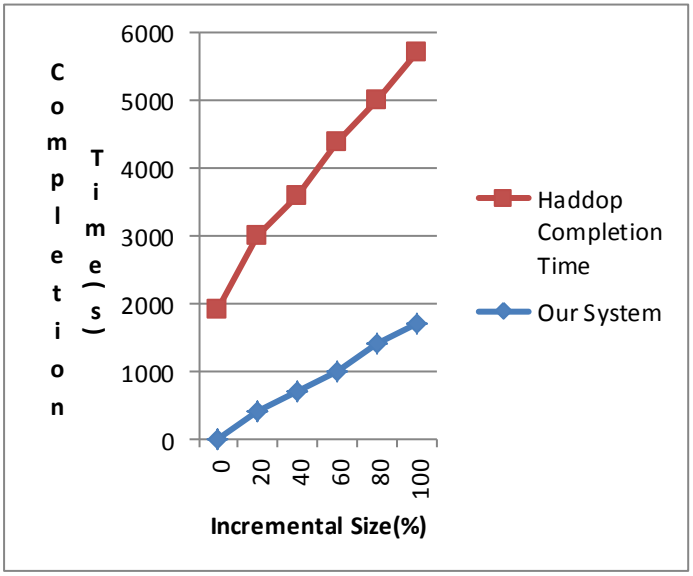

Fig.8 The completion time of our system over Hadoop of tera sort program

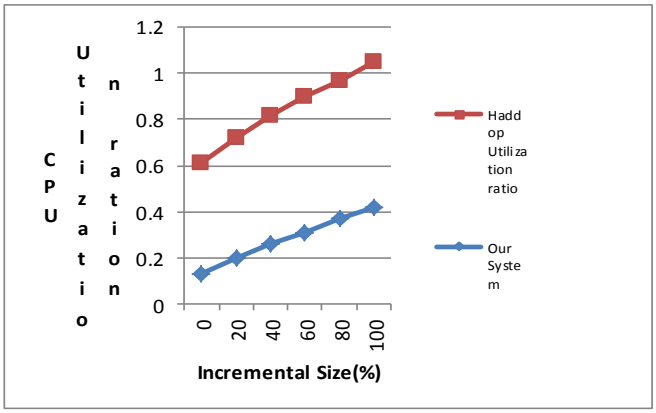

Fig.9 CPU utilization ratio of our system and Hadoop in word count program

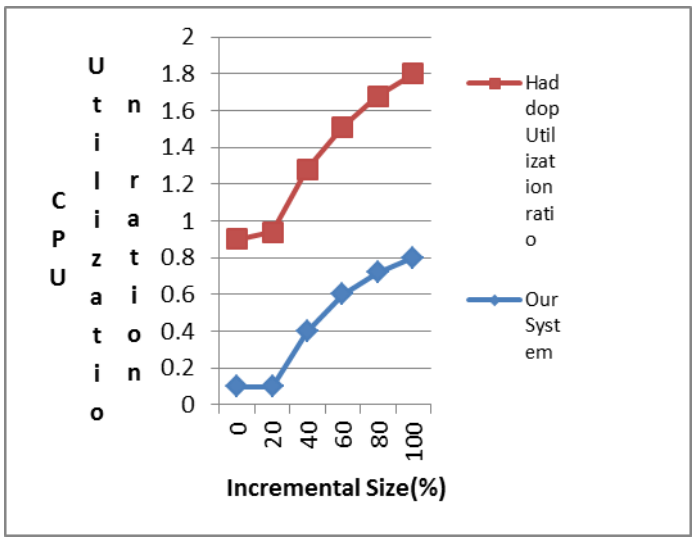

Fig.10 CPU utilization ratio of our system and Hadoop in tera sort program

VI. Conclusion

This paper recommends a cache-based methodology to summarized information, incremental computation and aggregate queries processing on MapReduce platform. Few algorithms are presented to improve performance of incremental operations by caching the top-*k* aggregate results. We recommend a strategy for cache replacement in which the value degree is used to decide which tuples should be replaced.

In future, we will make efforts to substitute the disk files with salable databases that will be advanced to organize big data set effectively in memory.

ACKNOLEDGEMENT

The authors would like to thank the researchers as well as publishers for making their resources available and teachers for their guidance. We are thankful to the authorities of Savitribai Phule Pune University, Pune and concerned members of IPGCON 2015 conference, for their constant guidelines and support. We also thank the reviewers for their valuable suggestions. We also thank the college authorities for providing the required infrastructure and support. Finally, we would like to extend a hearty gratitude to friends and family members.

VII. 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 ISSNl l1007-0214l l05/10l lpp39-50 Volume 19, Number 1, February 2014.

[2] Yandong Mao Robert Morris M. Frans Kaashoek "Optimizing MapReduce for Multicore Architectures" Massachusetts Institute of Technology, Cambridge, MA.

[3] Rong Chen, Haibo Chen, and Binyu Zang "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling" Parallel Processing Institute Fudan University PACT'10, September 11–15, 2010.

[4] Gurmeet Singh, Puneet Chandra and Rashid Tahir "A Dynamic Caching Mechnism for Hadoop using Memcached" Department of Computer Science, University of Illinois at Urbana Champaign.

[5] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, Rafael Pasquini "Incoop: MapReduce for Incremental Computations" Max Planck Institute for Software Systems (MPI-SWS) SOCC'11, Cascais, Portugal, October 27–28, 2011.

[6] Zhao Cao, Shimin Chen, Dongzhe Ma, Jianhua Feng, Min Wang "Efficient and Flexible Index Access in MapReduce" Published in Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.

[7] Kenn Slagter, Ching-Hsien Hsu,Yeh-Ching Chung Daqiang Zhang "An improved partitioning mechanism for optimizing massive data analysis using MapReduce" Published online: 11 April 2013 © Springer Science+Business Media New York 2013.

[8] Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D. Ernst "HaLoop: Efficient Iterative Data Processing on Large Clusters" Department of Computer Science and Engineering University of Washington, Seattle, WA, U.S.A. 36th International Conference on Very Large Data Bases, Singapore, September 13-17, 2010.

[9] Dongfang Zhao, Kan Qiao, Ioan Raicu "HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems" Illinois Institute of Technology, USA and Argonne National Laboratory, USA.

[10] Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh "Redoop Infrastructure for Recurring Big Data Queries" Worcester Polytechnic Institute, Worcester, MA USA VLDB '14, Hangzhou, China, September 15, 2014.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, AndrewWang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, Ion Stoica "PACMan: Coordinated Memory Caching for Parallel Jobs" University of California, Berkeley, Facebook, Microsoft Research, KTH/Sweden.

[12] James Manyika, Michael Chou, Brad Brown, Jacques Bughin, Rihards Dobbs, Charlas Roxburgs, Angela Hung Bayer "Big data: The next frontier for innovation, competition, and productivity" McKinsey Global Institute, May 2011.

[13] Min Chen · Shiwen Mao · Yunhao Liu "Big Data: A Survey" Published online © Springer Science+Business Media New York, 22 January 2014.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung "The Google File System" SOSP'03, Bolton Landing, New York, USA, October 19–22, 2003.

[15] Dominique A. Heger " Hadoop Design, Architecture & MapReduce Performance" DHTechnologies - www.dhtusa.com.