Inverted Indexing In Big Data Using Hadoop Multiple Node Cluster

Kaushik Velusamy
Dept. of CSE Amrita University
Coimbatore, India

Nivetha Vijayaraju
Dept. of CSE Amrita University
Coimbatore, India

Deepthi Venkitaramanan
Dept. of CSE Amrita University
Coimbatore, India

Greeshma Suresh
Dept. of CSE Amrita University
Coimbatore, India

Divya Madhu
Dept. of IT Amrita University
Coimbatore, India

***Abstract*—Inverted Indexing is an efficient, standard data structure, most suited for search operation over an exhaustive set of data. The huge set of data is mostly unstructured and does not fit into traditional database categories. Large scale processing of such data needs a distributed framework such as Hadoop where computational resources could easily be shared and accessed. An implementation of a search engine in Hadoop over millions of Wikipedia documents using an inverted index data structure would be carried out for making search operation more accomplished. Inverted index data structure is used for mapping a word in a file or set of files to their corresponding locations. A hash table is used in this data structure which stores each word as index and their corresponding locations as its values thereby providing easy lookup and retrieval of data making it suitable for search operations.**

Keywords—Hadoop; Big data; inverted indexing; data structure

I. INTRODUCTION

Wikipedia is an online encyclopaedia which contains over four million articles. In general, searching over such text based documents involves document parsing, index, tokenisation, language recognition, format analysis, section recognition. Hence a search engine for such large data which is done in a single node with a single forward index built over all the documents will consume more time for searching. Moreover the memory and processing requirements for this operation may not be sufficient if it is carried out over a single node. Hence, load balancing by distribution of documents over multiple data becomes necessary.

Google processes 20PB of data every day using a programming model called MapReduce. Hadoop, a distributed framework that processes big data is an implementation of MapReduce. Hence it is more apt for this operation as processing is carried out over millions of text documents.

Inverted index is used in almost all web and text retrieval engines today to execute a text query. On a user query, these search engines uses this inverted index to return the documents matching the user query by giving the hyperlink of the corresponding documents. As these indices contain the vocabulary of words in dictionary order only a small amount of documents containing the indices need to be processed.

Here, the design of a search engine for Wikipedia data set using compressed inverted index data structure over Hadoop distributed framework is proposed. This data set containing more than four million files needs an efficient search engine for quick access of data. No compromise must be made on the search results as well as the response time. Care should be taken not to omit documents that contain words synonymous user query. Since accuracy and speed is of primary importance in search, our methods could be favoured in such cases.

II. LITERATURE SURVEY

[2] For large-scale data-intensive applications like data mining and web indexing MapReduce has become an important distributed processing model. Hadoop–an open-source implementation of MapReduce is widely used for short jobs requiring low response time. Both the homogeneity and data locality assumptions are not satisfied in virtualized data centres. This paper [2] shows that ignoring the data locality issue in heterogeneous environments can noticeably reduce the MapReduce performance.

The authors also address 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, their 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 their 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. The new mechanism distributes fragments of an input file to heterogeneous nodes based on their computing

capacities. Their approach improves performance of Hadoop heterogeneous clusters.

According to [1], a virtualized setup of a Hadoop cluster that provides greater computing capacity with lesser resources is presented, as virtualized cluster requires only fewer physical machines with master node of the cluster set up on a physical machine, and slave nodes on virtual machines (VMs).

The Hadoop virtualized clusters are configured to use capacity scheduler instead of the default FIFO scheduler. The capacity scheduler schedules tasks based on the availability of RAM and virtual memory (VMEM) in slave nodes before allocating any job. Instead of queuing up the jobs, the tasks are efficiently allocated on the VMs based on the memory available. Various configuration parameters of Hadoop are analysed and the virtualized cluster is fine-tuned to for best performance and maximum scalability. The results show that the approach gives a significant reduction in execution times, which in turn shows that the use of virtualization helps in better utilization of the resources of the physical machines used. Given the relatively under power of the machines used in the real cluster the results were fairly relevant. The addition of more machines in the cluster leads to an even greater reduction in runtime.

According to [8], Hadoop, the emerging technology made it feasible to combine it with virtualisation to process immense data set. A method to deploy cloud stack with Map Reduce and Hadoop in virtualised environment was presented in this paper. A brief idea on setting up a Hadoop experimental environment to capture the current status and the trends of optimising Hadoop in virtualised environment was mentioned. The advantages and the disadvantages of the virtualised environment was discussed, ending with the benefits of one's compromise over the other. Making use of the virtualised environment in Hadoop fully utilizes the computing resources, make it more reliable and save power and so on. On the other side, we have to face the lower performance of virtual machine. Then some methods to optimize Hadoop in virtual machines were discussed.

III. Problem Statement

The result of any user's search query must be fast, should not miss any relevant data related to the query. A search engine designed by using distributed framework like Hadoop and inverted index data structure is fast and returns all the relevant results. In order to do this and to analyse the feasibility of deployment of a search engine for Wikipedia various requirements and parameters to be considered must be well understood and analysed.

IV. Parameters For Performance Metrics

The performance of a search operation through an inverted index built over millions of Wikipedia documents distributed over a multiple node Hadoop cluster in a virtual node could be effectively measured using various parameters such as response time ,throughput, speed up, latency hiding, computation time, communication time and thereby computation and communication ratio. In terms of the search operation in this distributed and parallel platform, response time indicates the time taken for the first of the relevant wiki documents to appear when a query is made. Through put in other words can be defined as the number of transactions per second or the maximum number of search queries that can be made per second, speed up factor refers to the time that could be saved due to a fraction of process that could be parallelized .As the documents are distributed across multiple documents, the percentage of search operation that can be parallelized and thereby the speedup achieved could be measured. [6]

$$\text{Speed-up factor} = \frac{Ts}{Tp}$$

Where Ts -Time taken for serial execution of the process and Tp - time taken after parallelization. As more time is consumed in start-up of a communication between nodes, making use of this time effectively for completing as much computations as possible would improve performance. This can be achieved via non-blocking send routines thereby helping in achieving latency hiding. Sometimes, even blocking send routines allow computations to take place until the expected messages reach the destination aiding in improving latency hiding. Total processing time includes computations and the communications carried out.

$$T_{process} = T_{computation} + T_{communication}$$

The computation time for the search operation can be calculated by counting the number of computations per process. Computation involves locating the node that has the relevant documents. [9]Communication time depends on the size of the data transferred, start-up time for each message and number of messages in a process and the mode of data transfer. Communication in multiple cluster node involves requesting a node for certain documents based on the query and the nodes responding with the requested documents.

$$T_{communication} = T_{start\ up} + w * T_{data}$$

Where

$T_{start\text{-}up}$ – Time needed to send a blank message
T_{data} - Time to send/receive a single data word
W - No. of data words

$$\text{Speed-up factor} = \frac{Ts}{Tp} = \frac{Ts}{T\ computation + T\ communication}$$

The computation communication ratio throws light on the how much time communication takes as a result of increased amount of data.

V. Inverted Indexing

Indexing refers to creating a link or a reference to a set of records so as to enable better identification or location. Forward indexing and inverted indexing are two main types of indexing. When an element say 97 is accessed through its index say Arr [3] in Fig 1, then it is forward indexing. When the same element is searched based on its occurrence or the number of occurrences, then it is inverted indexing.

23	45	64	97	53	72	93
Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]	Arr[5]	Arr[6]

Fig. 1. Illustration of Forward and Inverted Indexing

An inverted index for a document or set of documents contains a hash table with each word as its index and a posting list as value of each index. A postings list consists of a document id, position of word in that document and frequency of occurrence of each word in that document.

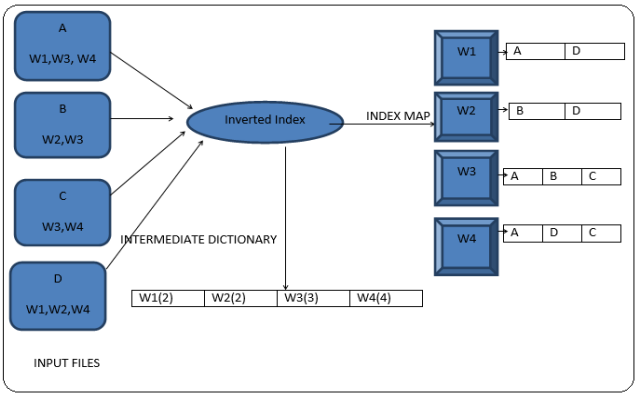

Fig. 2. Inverted Indexing-Working

If there are n documents to be indexed then a unique document id is set for each document from 0 to n-1. The postings list for a term is sorted based on various criteria. Though it is easy to sort it based on document id, for search operations other parameters are considered for sorting. Sorting done based on frequency of a term in a document is more apt for a search operation. At the end of sort processing this data structure returns the top k documents in the postings list where k is the maximum returning capacity of a search engine in a single stretch. [7].

A. *Algorithm*

```
Inverted_Index (int docID[n], string doc[n])
M ← new HashMap
Count ← 0
For all document with docID m from 0 to n-1
        For all term tm and position pos in doc
         With   docID m do
                  M {tm, previous pos, previous m} ← M
                  {tm, pos, m} +1
                  Count (tm, m) ++

        For each tm in M with docID m
                  Sort (count (tm, m))
```

As explained in Fig 2, input to the indexing algorithm is the set of document IDs and the contents of all the documents. Each new term in the document is formed as an index in the hash table. For each occurrence in a document its document ID is added to the postings list of that term along with its position. After each occurrence of a term in a document its corresponding frequency variable count is incremented. Postings list of each term is finally sorted based on the frequency of words in each document.

```
Algorithm Search (HashMap M, string word)
        return M[word]
```

In the search part of an inverted index, the word which is queried by the user is passed as input along with the hash map which has the set of all positions of the each word in the document. Hash map takes the word as its index and returns the value stored in that index.

VI. DATA SET - WIKI DUMPS

All the contents of Wikipedia are available in downloadable format as wiki dumps. This can be taken by users for archival/backup purposes, offline storage, educational purpose, for republishing, etc. There are over four million files in Wikipedia, compressed form as wiki dumps of size 9.5 Giga bytes approximately. When extracted from the compressed form, it comes to around 44 Giga bytes. Database backup dumps have a complete copy of all Wikipedia documents as wikitext and the set of all its metadata in XML. Static HTML dumps has copies of all pages of Wikipedia wikis in HTML form.

Contents of dumps include page-to-page link, media metadata, title, information about each page, log data, Misc bits, etc. These are in the wrapper format described at schema Export Format which is compressed in bzip2 and .7z format. They are provided as dumps consisting of entire tables using mysqldump. Internal file system limit must be taken into account before extracting these files from compressed format.

VII. HADOOP

Map Reduce method has emerged as a scalable model that is capable of processing pet a bytes of data. Fundamental concept of MapReduce: Rather than working on one, huge block of data with a single machine, Big Data is broken up into files that further are broken into blocks by Hadoop and parallel processing and analysis is carried out. [5]

The Hadoop is a map reduce framework that provides HDFS (Hadoop Distributed File Systems) infrastructure. HDFS was designed to operate and scale on commodity hardware. Breakdown in hardware is largely compensated by replication of blocks of data in multiple nodes.

A. *Hadoop Distributed Filesystem (Hdfs) Overview*

HDFS (Hadoop Distributed File System) is a distributed user level file system which stores, processes, retrieves and manages data in a Hadoop cluster. HDFS infrastructure that Hadoop provides, include a dedicated master node called Name Node which contains a job tracker, stores meta-data, controls the overall distributed process execution by checking out whether all name nodes are functioning properly through periodic heart beats. It also contains many other nodes called Data Node which contains a task tracker, stores applications data. The Ethernet network connects all nodes. HDFS is implemented in Java and it is platform independent. Files in HDFS are split into blocks and each block is stored as an independent file in the local file system of Data Nodes. Each block of a HDFS file is replicated at least three times in multiple Data Nodes. Through replication of application data, provides data durability.[9]

The Name Node manages the namespace and physical location of each file. File system operations are done by HDFS client by contacting the Name node. Name Node checks a client's access permission and gets the list of Data Nodes hosting replicas of blocks. Then, requirements are sent to the "closest" Data Node by requesting a particular block. Then, a

socket connection is created between the client and the Data Node. The data is transferred to the client application. When a client application writes a HDFS file, it first splits the file into HDFS blocks and the Name Node gets the list of Data Nodes which are replicas of each block and writing data is done by multithreading. [3]

If the client application is running on a Data Node, the first replica of the file is written into the local file system of the running Data Node. If the client application isn't running on a Data Node, a socket connection is created between the client and the first Data Node. The client splits the block into smaller packets and starts a pipeline: the client sends a packet to the first Data Node; the first Data Node on getting this packet, writes this to the local file system, and also sends it to the next Data Node. A Data Node can receive the data from a previous node and at the same time forward the data to the next node. When all nodes in this pipeline write the block into local file system successfully, the block write is finished and then Data Nodes update the block physical information to the Name Node. The architecture of multiple cluster implementations has been explained in Fig 3.

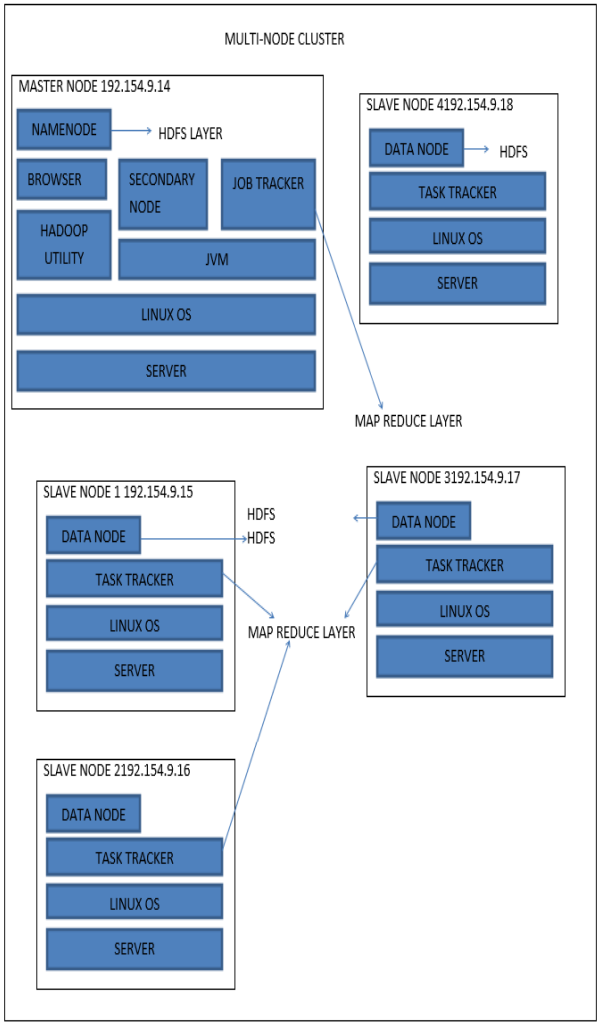

Fig. 3. Hadoop Multiple node Cluster Architecture

B. *Working process of Hadoop Architecture*

Hadoop is designed to run on a large number of machines that don't share any memory or disks. When a data is loaded into Hadoop, the software splits that data into pieces and spreads it across different servers. Hadoop keeps track of where the data resides. And because there are replica of single data, data stored on a server that goes offline or dies can be automatically replicated from a known good copy.

In a Hadoop cluster, every one of those servers has two or four or eight CPUs. Each server operates on its own little piece of the data. Results are then delivered back through reduce operations. MapReduce maps the operation out to all of those servers and then reduces the results back into a single result set. Since Hadoop spreads out data, it is possible to deal with lots of data. Since all the processors work in parallel and harness together, complicated computational questions can be performed. Node failures are automatically handled by the framework for both map and reduce functions.

VIII. Assumptions And Goals

Applications that run on HDFS have large data sets. A typical file in HDFS is Gigabytes to Terabytes in size. Therefore, HDFS must provide high bandwidth and scalability to hundreds of nodes. HDFS applications need a write-once-read-many access model for files. If a file is created and written, it is assumed that it will not be changed in future. This is to simplify data coherency and to get high throughput data access.

IX. Proposed Solution

A Hadoop cluster is established by passing Wikipedia files as input data and inverted indexing is done by taking advantage of Map Reduce.

In the map phase, the Wikipedia files are divided equally among mappers and passed as inputs. Each Wikipedia file is given a unique document ID. Each mapper indexes each term in its file into the hash map with the corresponding document ID and position in that document as a posting list. When it finds that term for the first time it creates that term as the index and writes the corresponding postings list of that term. When the term is found again, the corresponding posting list for that position is appended with the previous list to index holding that term.

A. *Map function pseudo-code*

Algorithm Map (int docID[x], string doc[x])
M ← new HashMap
Count ←0
For all document of docID m from 0 to x-1
For all term tm and position pos in doc with docID m do
M {tm, previous pos, previous m} ← M{ tm, pos, m }+1
Count (tm, m) ++
emit (M, count (tm, m))

In the above algorithm X is the maximum number of documents processed within a mapper. The input file is read

word by word and indexed accordingly with its document ID and corresponding position in a hash map. The variable count keeps track of the frequency of a term within each document in that mapper. At the end each mapper returns its hash map with the count value of each term in a document.

In reduce phase each reducer takes in its responsibility a term or set of terms. These terms are given an index position in a global hash map where all the terms are stored as index. When a reducer encounters its term from a mapper it appends the posting list of that mapper to its value in this hash map. After appending the entire list of that term from all the mappers, reducer sorts posting list based on count value of each document. The more the value, the preference is higher. In the same way, all the terms in this whole document are indexed in the hash map in this reduce phase.

B. Reduce function pseudo-code

Algorithm Reduce (term tm, List of hash maps of each mapper[], count{tm, docID})

G ← new HashMap //G is common HashMap for all reducers

for each hash map H from all mappers

for each term tm in document with docID m and position pos in H

//n is the total number of documents

G{ tm , previous pos, previous m} ←

H{ tm, pos , m }+1

Sort(count (tm , m))

//values in list is sorted based on the count value of each term in a document

emit(G)

In the above pseudo code each reducer takes as its input all the hash maps of various mappers and the count values of each term in a document. Reducer checks each hash map with its allotted term and if it matches with any mapper's index it appends that value in global hash map. When all the values are appended for a term it is finally sorted based on its count value in each document.

C. Retrieval

The terms in global hash map is divided among the mappers along with their corresponding posting list. When the user queries a term, the name node sends this query to the corresponding data node. Value of the term is passed to the reducer as a complete list. Reducer returns the first k values of that term to the user where k is the maximum number of pages returned for a user query.

X. FUTURE WORKS

First a distributed, multiple node Hadoop cluster has been built and the millions of wiki documents has been distributed over these nodes. A compressed inverted index containing indices for words in dictionary order is to be built over these documents. After building inverted index, distributed performance evaluation for searching documents based on keyword is intended to be made. Further data analysis and text mining could be made based on index support. The results of text mining and data analysis would help in suggesting related pages based on data such as other documents where the synonyms of the query are predominantly found. Indexing can be further partitioned in to local index partitioning and global index partitioning. In term based partitioning or global index partitioning, each node in the multiple cluster stores posting list only for a subset of the term in the collection. Local index partitioning is each server building a separate index for the files that it contains. When this is done, each server indexes only the document that it contains, reducing the number of documents to thousands. This is very much lesser compared to the actual number of indices that had to be built if indexing is to be done for over a million documents. So, instead of building a single index over 4 million Wikipedia documents, local index would be built over documents on each node and an index would be built on these indices thereby quickening search and compressing indices. Further, indices built over articles (a, the, an) and other such common words would be deleted for adding accuracy.

XI. CONCLUSION

In this paper, a compressed inverted index data structure that could help in crawling for words in dictionary order such that all the indices built for millions of documents need not be processed has been proposed. In addition, basic factors for designing indices such as merge factors, storage technique, index size, look up speed, maintenance, fault tolerance etc. will also be taken into account. Building a local index for files within those system will prove to be a great way to solve problems that could arise in parallelism such as when a file is added, whether index updating should happen before the search operation that is currently going on and vice versa as only a portion of the entire set of documents need to be updated making the 'index merging' process very simple. In addition to storing a token word, its document id and the position in which it appears, we have suggested to add token word document id and its frequency to rank up the relevant documents. Our work has motivated several interesting open questions such as which type of inverted index data structure would be most useful for text mining. Other ways to optimise performance in search is being investigated and added over to the suggested methods.

REFERENCES

[1] Raj, A. Kaur, K. ; Dutta, U. ; Sandeep, V.V. ; Rao, S. "Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler", Third International Conference on Services in Emerging Markets (ICSEM),Mysore, India, Dec 2012. Page(s): 50 - 57. Print ISBN: 978-1-4673-5729-6. INSPEC Accession Number: 13343537. D.O.I: 10.1109/ICSEM.2012.15. Link: http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?arnumber=6468179

[2] Jiong Xie; Shu Yin ; Xiaojun Ruan ; Zhiyang Ding ; Yun Tian ; Majors, J. ; Manzanares, A. ; Xiao Qin. "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters". IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, April, 2010. Page(s): 1 - 9. Print ISBN: 978-1-4244-6533-0. INSPEC Accession Number: 11309800. D.O.I : 10.1109/IPDPSW.2010.5470880. Link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5470880

[3] Kala Karun, A ; Chitharanjan, K ; "A review on hadoop — HDFS infrastructure extensions ", IEEE Conference on Information & Communication Technologies (ICT), JeJu Island, April 2013. Page(s): 132 - 137. Print ISBN: 978-1-4673-5759-3. INSPEC Accession Number: 13653440. D.O.I: 10.1109/CICT.2013.6558077. Link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6558077

[4] Richard Mccreadie ; Craig Macdonald ; Iadh Ounis; "MapReduce indexing strategies: Studying scalability and efficiency". International Journal of Information Processing and Management. Volume 48 Issue 5, September, 2012. Pages: 873-888. Publisher Pergamon Press, Inc. Tarrytown, NY, USA. ISSN: 0306-4573 doi>10.1016/j.ipm.2010.12.003. Link: http://dl.acm.org/citation.cfm?id=2337723

[5] Apache Hadoop, Hadoop, HDFS, Avro, Cassandra, Chukwa, HBase, Hive, Mahout, Pig, Zookeeper are trademarks of the Apache Software Foundation. http://www.hadoop.apache.org/ Last Published: 10/16/2013 06:37:41. Copyright © 2012. The Apache Software Foundation. 2nd October 2013.

[6] Barry Wilkinson; Michael Allen; "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers" (2nd Edition). Publication Date: March 14, 2004, ISBN-10: 0131405632, ISBN-13: 978-0131405639 , Edition: 2. Link : http://www.amazon.com/Parallel-Programming-Techniques-Applications-Workstations/dp/0131405632

[7] Gal Lavee ; Ronny Lempel ; Edo Liberty ; Oren Somekh ; " Inverted index compression via online document routing" Published in: WWW '11 Proceedings of the 20th international conference on World Wide Web. Pages 487-496. ISBN: 978-1-4503-0632-4 doi:10.1145/1963405.1963475. Publisher ACM New York, NY, USA ©2011. Link: http://dl.acm.org/citation.cfm?id=1963475

[8] Guanghui Xu; Feng Xu; Hongxu Ma; "Deploying and researching Hadoop in virtual machines". Published in: IEEE International Conference on Automation and Logistics (ICAL), Zhengzhou, Aug. 2012. Page(s): 395 - 399. ISSN: 2161-8151. E-ISBN: 978-1-4673-0363-7. Print ISBN: 978-1-4673-0362-0. INSPEC Accession Number: 13000378. D.O.I:10.1109/ICAL.2012.6308241. Link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6308241

[9] Shvachko, K.; Hairong Kuang ; Radia, S. ; Chansler, R. " The Hadoop Distributed File System". Published in: IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, May 2010. Page(s): 1 - 10. E-ISBN: 978-1-4244-7153-9. Print ISBN: 978-1-4244-7152-2. INSPEC Accession Number: 11536653. D.O.I: 10.1109/MSST.2010.5496972. Link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5496972

[10] Ishii, M.; Jungkyu Han; Makino, H; "Design and performance evaluation for Hadoop clusters on virtualized environment" Published in: International Conference on Information Networking (ICOIN), Bangkok, Jan. 2013. Page(s): 244 - 249. ISSN: 1976-7684. E-ISBN: 978-1-4673-5741-8. Print ISBN: 978-1-4673-5740-1. INSPEC Accession Number: 13431469. Digital Object Identifier: 10.1109/ICOIN.2013.6496384. Link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6496384