# Intelligent Data Placement Mechanism for Replicas Distribution in Cloud Storage Systems

Ibrahim Adel Ibrahim*, Wei Dai *, Mostafa Bassiouni †
* Department of Electrical & Computer Engineering
† Department of Computer Science
University of Central Florida
Orlando, USA
efahad@knights.ucf.edu , wdai@knights.ucf.edu, bassi@cs.ucf.edu

*Abstract*— **The Hadoop Distributed File System (HDFS) is a distributed storage system that stores large volumes of data reliably and provide access to the data by the applications at high bandwidth. HDFS provides high reliability and availability by replicating data, typically three copies, and distribute these replicas across multiple data nodes. The placement of data replicas is one of the key issues that affect the performance of HDFS. In the current HDFS replica placement policy the replicas of data blocks cannot be evenly distribute across cluster nodes, so the current HDFS has to rely on load balancing utility to balance replica distributions which results in more time and resources consuming. These challenges drive the need for intelligent methods that solve the data placement problem to achieve high performance without the need for load balancing utility. In this paper, we propose an intelligent policy for data placement in cloud storage systems addressing the above challenges.**

***Keywords—Hadoop Distributed File System; Replica Placement; Data Replication; Load Balance; Hadoop; Cloud Computing***

## I. INTRODUCTION

The advances in Information Technology and the storage demands of cloud computing have been growing rapidly in the last few years. In order to meet the ever-growing data storage demands from users, The storage system for cloud computing consolidates large numbers of distributed commodity computers into a single storage pool to provide online services for data storage with immense capacity and high quality of service in an unreliable and dynamic network environment at low cost[1]. To build such a cloud storage system, an increasing number of companies and academic institutions have started to rely on the Hadoop Distributed File System (HDFS) [2]. Many cloud vendors have given attractive storage service offerings that provide a giant cloud-based storage space for users, such as Amazon, Dropbox, Google Drive, and Microsoft's OneDrive [3]–[5].

HDFS has been widely used and become a common storage appliance for cloud computing. It is the storage part of Hadoop framework, it is a distributed, scalable, and portable file system designed to run on low-cost hardware. Although it has many similarities with other existing distributed file systems, the differences from other distributed file systems are significant. HDFS is especially designed to be highly fault-tolerant, and to provide high throughput access to application data and is suitable for applications that have large data sets.

In HDFS each data file is stored as a sequence of blocks, The blocks of a file are replicated for reading performance and fault tolerance. HDFS uses an uniform triplication policy (i.e. three replicas for each data block) to improve data locality and ensure data availability and fault tolerance in the event of hardware failure. The placement policy of replicas is critical to HDFS performance and reliability. This policy could also achieve load balancing by distributing work across the replicas. For the common case, the triplication policy in HDFS works well in term of high reliability and high performance. The current HDFS Replica Placement Policy (RPP) is a rack-aware policy. The drawback of the policy is that it cannot evenly distribute replicas to cluster nodes. Data placement problem therefore needs to be considered to obtain an optimal placement solution that minimize the retrieval time. HDFS provides a balancing utility to address the issue of unbalanced HDFS cluster which can seriously degrade the performance of Hadoop applications. This utility is used to analyze replica placement and rebalance replicas across the cluster nodes at the cost of extra system resources and running time.

In this paper, we develop a fast heuristic algorithm to solve the data placement problem. it distributes replicas to cluster data nodes as evenly as possible, and also meet all replica placement requirements of HDFS, as a result, there is no need to run the balancing utility. The proposed policy in this paper is the first that addresses the load balancing problem by generating an even replica distribution to the data nodes at the beginning of distribution then during the normal operations on data nodes.

The remaining of the paper is organized as follows. Section II introduces the HDFS RPP and related work. Section III describes the proposed policy. In Section IV the Evaluation results are presented, and we conclude in Section V.

## II. BACKGROUND AND RELATED WORK

HDFS introduces a simple but highly effective policy to allocate replicas for a block. The default HDFS RPP (as of Hadoop 2.7.1) If the client of HDFS runs outside the Hadoop cluster is to put one replica on one node in the local rack; another on a node in a remote rack; and the third on a different node in the same remote rack [6].This policy can reduce the inter-rack write traffic when distributing data to data nodes because the three replicas of every data block are stored on only two racks. High fault tolerance are maintained because replicas are placed

978-1-5090-5263-9/16 $31.00 © 2016 IEEE
DOI 10.1109/SmartCloud.2016.23

on three different data nodes. However, this policy cannot generate an even replica distribution across the data nodes.

There has been large amount of research work conducted on the HDFS RPP due to its importance regarding improving the performance of HDFS. Shabeera et al. propose a RPP for Hadoop that depends on the available bandwidth between the HDFS client and cluster nodes in [7], the bandwidth can be measured and compared periodically, and the node that has the maximum bandwidth is selected to place the replica on it in order to reduce the time of data transfer. Khan et al. [8] presents an algorithm that finds the optimal number of codeword symbols needed for recovery for any XOR-based erasure code and produces recovery schedules that use a minimum amount of data. It improves I/O performance in practice for the large block sizes used in cloud file systems, such as HDFS. Zhang et al. address the dynamic block replication issue in [9]. They propose several constant-factor local search approximation algorithms, and present a dynamic replica distribution mechanism that implements the algorithms in HDFS. Xie et al. address the HDFS running on heterogeneous clusters and propose a replica placement mechanism for it in [10], which distributes replicas to nodes depending on their computing capacities in order to balance the data processing load across all cluster nodes. In our previous work [11], we address the load balancing issue from the perspective of balancing replicas assignment across all cluster nodes of Hadoop, and propose a new placement policy that split the nodes in to three different sections and run an algorithm to distribute the replicas on the three sections across all cluster nodes. . In our other previous work [12], we address the load balancing issue from the perspective of task assignment of Hadoop, and propose an improved task assignment scheme that strives to balance the map task processing load of MapReduce jobs across all cluster nodes. Finally, the recent research work in [13] and [14] helped with the formation of our research idea.

## III. Balanced Replica Placement Policy

The replicas assignments across all cluster nodes generated by the HDFS RPP is unbalanced because the policy places two replicas on two different nodes belong to one random rack and the third replica on a node located on another random rack. To overcome this problem, our new policy places the replica on a rack contains a data node has the lowest load among all the nodes on the cluster with keeping same rules of existing HDFS RPP. It is implemented by keeping pointer for each rack in the cluster to keep tracking the load on this rack. This policy makes it possible to generate an even replica distribution because the placement of replicas in this policy is driven by the load on the nodes of the racks instead of selecting a random racks. Since the intelligent balance scheme is the key the new policy, we call our new policy the Intelligent Data Placement Mechanism Policy (IDPM). Under IDPM, the replica placement process consists of two parameters: rack selection phase, and node selection phase. Before the placement of a replica, rack is selected then a node inside this rack is selected to place the replica on it, the rack and node selection is the core of this policy.

In rack selection phase, IDPM selects two racks with the least load on it depending on a pointer called **Rack_Pointer** which is continuously updated in every time a replica placement is achieved, the Rack_Pointer of a specific rack is the number of free nodes in this rack for the current iteration. Node selection phase starts after rack selection, in this phase a free node from the selected rack in the current iteration is selected, in every iteration only one replica is assigned to every node, when every node is assigned with a replica, the second iteration starts and so on. The number of iterations needed to complete the whole replicas placement can be calculated as following: Let $\boldsymbol{R}$ be a collection of $\boldsymbol{L}$ racks, each of which has $\boldsymbol{r_i}$ ($\boldsymbol{i}$ = 0,1, 2, …, $\boldsymbol{L}$-1) available nodes on it. Let $\boldsymbol{N}$ be the total number of all available nodes, $\boldsymbol{n}$ is the total numbers data blocks to be distributed, and $\boldsymbol{df}$ is the duplication factor which is assumed to be three replicas for each data block in this policy, $\boldsymbol{p}$ is the total number of replicas to be distributed to the nodes in main distribution table, and $\boldsymbol{I}$ is the number of Iterations to complete the distributions. $\boldsymbol{N}$, $\boldsymbol{p}$, and $\boldsymbol{I}$ can be calculated according to the following formulas:

$$p = n \text{ x } df \tag{1}$$

$$N = r_0 + r_1 + r_2 + \ldots + r_{L-1} \tag{2}$$

$$I = \begin{cases} p \,/\, N & if\ (p\ \%\ N = 0) \\ (p/\,N) + 1 & if\ (p > 0) \end{cases} \tag{3}$$

, where "/" denotes integer division, and "%" integer remainder.

After figuring out $\boldsymbol{N}$, and I, IDPM builds the main assignment table, it is two dimensional array with N columns and I rows, all replicas is stored in the main assignment table where each column represents a node, and each row represents a distribution iteration, it consists of small tables attached together to form the main table, each small table represents a Rack with columns number equal to the number of nodes in that rack, and the rows corresponding to the number of iterations for that rack, in each iteration of building the main table the replicas are assigned to the whole current row before it jump to the next iteration with the new row.

The main table consists of many sections, each section corresponding to one rack, the column number on the main table $\boldsymbol{C}$ which dedicated for a selected rack $\boldsymbol{R}$ and selected node $\boldsymbol{d}$ can be calculated according to the following formulas:

$$C = d + \sum_{i=0}^{R-1} r_i \tag{4}$$

So every replica assigned to selected rack R and selected node $d$ is stored in column $\boldsymbol{C}$ in the main assignment table, Assume there are three racks, rack 0 has five available nodes, rack 1 three, and rack 2 seven, As shown in Fig. 1. In this example: the total number of columns in the main table is $\boldsymbol{N}$ = 15. The main table consists of three sections, IDPM first assigns all five nodes on rack 0 to section 1, all three nodes on rack 1 to section 2, and then the seven nodes of rack2 to section 3. The main assignment table can be formed from merging these three

sections. So the main distribution table has 15 columns and I rows, where I can be found according to formula (3).

IDPM uses a tabular scheme to distribute replicas, the replica that supposed to be placed on a rack are stored in table section corresponding to this rack, Each column in replica placement table is used to store the replicas assigned to one node. As shown in Fig. 2, replicas are filled into assignment table in block number order, from left to right, and from top to bottom. Each colored section is used for the replicas of one rack, the three sections forming the main table, at the beginning of distribution two replicas of block number 0 are assigned to the section of rack 2. However rack2 is selected at the beginning because it is the rack that has more free nodes, each of these two replicas are placed on different free node, node 0 and, the third replica is assigned on another rack that has higher free nodes, rack 0, on free node, node 0. When all the columns of the first row is filled, the distributions start on the second row and so on. As shown in Fig. 2.

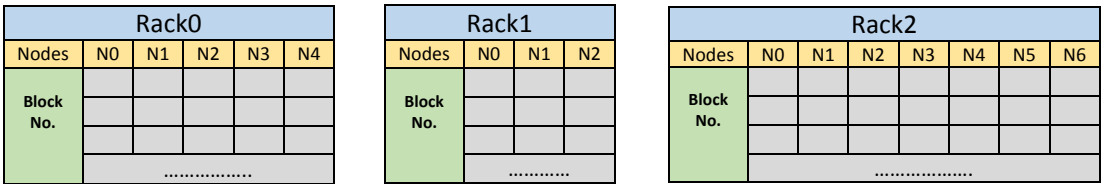


Fig. 1. Example of main table formation.

| Rack | 0 | | | | | 1 | | | 2 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Node | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| | Main Assignment Table | | | | | | | | | | | | | | |
| Block No. | 0 | 1 | 2 | 2 | 4 | 3 | 3 | 4 | 0 | 0 | 1 | 1 | 2 | 3 | 5 |
| | 4 | 5 | 7 | 8 | 9 | 6 | 8 | 8 | 5 | 6 | 6 | 7 | 7 | 9 | 9 |
| | 10 | 11 | 13 | 13 | 15 | 12 | 12 | 14 | 10 | 10 | 11 | 11 | 12 | 13 | 14 |
| | 15 | 16 | 17 | 19 | 19 | 14 | 18 | 18 | 15 | 16 | 16 | 17 | 17 | 18 | 19 |
| | …… | | | | | | | | …… | | | | | | |

Fig. 2. Main assignment table for Replicas placement.

The replicas are filled into the replica assignment table in block number order from replica 0 through replica ***n-1***, line by line, after filling one line, it moves to next line from top to bottom. Before it assigns the third replica of current block to a rack, IDPM tags the rack on which the first two replicas of current block reside on as **avoid_rack** to discard it during the process of finding the elected rack for the third replica. After each rack election IDPM assigns the replica to the elected node of this rack. The process continues until PRPP has assigned ***p*** replicas.

Finally, IDPM results in even distribution of replicas of an application across all cluster nodes regardless the number of nodes in each rack in this cluster because the distribution of replicas is achieved depending on the total number of nodes available for this application. The following is the pseudocode of the replica distribution algorithm for all nodes.

**Algorithm: replica distribution algorithm**

**Input:**
A collection of ***L*** racks, each of which has $r_i$ (***i*** = 0, 1, 2, 3…, ***L***-1) available nodes on it.
***n*** data blocks to be distributed, which are numbered 0 through (***n*** - 1).
**Output:**
***T*** [***I***][***N***]: Replicas assignment table.

1. calculate ***N*** and ***I*** according to (2) and (3) respectively;
2. **for (*i* = 0 to *L* -1)**
3. Rack_compete [***i***] = ***r*[*i*]**; // initialize rack competition register by initializing it with the actual node number in every rack
4. **for (*j* = 0 to *r[i]-1*)**
5. *Node_free*[***i***][***j***] = **false;** // indicator whether a node assigned a block during the current assignment iteration or not.
6. **end for**
7. **end for**
8. **rack_avoid= -1** // used to avoid the same rack for the third replica
9. ***Block*** = 0; // Reset block number to start the the assignments from block number 0.
10. **while (*Block* < n)** *// Find the next unassigned block.*
11. **current_rack = Elected_rack (**Rack_compete**)** //call the function elected rack to find the next available //rack.
12. **current_node = Elected_node** (current_rack) // call the function elected node to find the next //available node in the current elected rack.
13. **Count**=0;
14. **for (*i* = 0 to** *current_rack-1*)
15. Count=Count + r[i] *// to count the number of //columns located before the section of current rack.*

```
16.     end for
17.     Column =Count + current_node;    //formula no. (4)
18.     T [ iteration[current_rack]][column]= Block;
19.     current_node = Elected_node (current_rack)
20.     Column =Count + current_node;
        //column of the selected node
21.     T [ iteration[current_rack]][column]= Block;
22.     rack_avoid= current_rack;
         //avoid current rack for 3rd replica
23.     current_rack  = Elected_rack (Rack_compete)
24.     current_node  = Elected_node (current_rack)
25.     for (i = 0 to current_rack-1)
26.         Count=Count + r[i]; // to count the number of
    //columns located before the section of current rack.
27.     end for
28.     T [ iteration[current_rack]][column]= Block;
29.     Block=Block + 1; // take the next block
30. end while
31.
32. FUNCTION Elected_rack (Rack_compete)  // //function
    //to return the elected rack.
33.    if (rack_avoide == 0)
34.      current rack =1;
35.    end if
36.    for (i = 0 to L -1)
37.      if (i == rack_avoide)
38.        exit;    // skip the current rack for  the second third
    replica
39.
40.    end if
41.    if (iteration[i] < iteration[current_rack])
42.          current_rack= i;
           // use the rack with the lowest iteration.
43.    end if
44.    if(Rack_compete[i]>Rack_compete[current_rack])&&
    (iteration[i] = = iteration[current_rack])
45.          Current_rack= i;
           // election of the rack with more free nodes.
46.    end if
47.   end for
48.   Return current_rack
49. END FUNCTION
50.
51. Function Elected_node (current_rack)   // return the
    elected node for the next block assignment.
52.  for (i = 0 to r[current_rack] -1)
53.     if (Node_ free[current_rack][i] = false)
54.          Node_ free[current_rack][i] = true;
55.          Current_node= i;
        // if the node is free, tag it as used and return its
    //order to be used as current node
56.          Rack_compete[current_rack]=
    Rack_compete[current_rack] -1;  // decrease the available
    // free nodes in the current iteration by 1.
57.     end if
58.   End for
59.   if (Rack_compete[current_rack] = = 0)
60.     Iteration[current_rack]= Iteration[current_rack] +1 ;
61.     Rack_compete[current_rack] = r[current_rack];
        // if the available free nodes in the current rack reaches
        //0, start new iteration and set number of nodes back.
62.     for (j = 0 to r[current_rack]-1)
63.        Node_ free[current_rack][j] = false;
64.      End for
65.   end if
66.   Return current_node
67. END FUNCTION
```

## IV. EVALUATION

In replica distributions generated by the HDFS RPP the number of replicas on one single node is a Discreet Random Variable (DRV). To test the distribution of a DRV, we conducted large scale simulation to examine the replica distribution generated by HDFS RPP, where theoretical simulation is more appropriate than actual implementation because much more distribution samples can be obtained by simulation. Two scenarios are considered, in each scenario a simulation was conducted as shown in TABLE I. Scenario one and is similar to situations Hadoop applications encounter in practice. There are many cases in which application has a dedicated cluster where the application can use all the nodes of racks belong to this cluster which is the case of scenario one,

Assume Z is the number of replicas assigned to a single node resulted from replica distributions generated by HDFS RPP. Fig. 3 shows the probability distribution of Discreet Random Variable Z in Scenarios one, based on 15,000 simulation runs. HDFS RPP generates uneven replica distributions in scenario one as confirmed in simulation results.

TABLE I. SIMULATION SETTINGS

| Simulation Settings | Scenario One | Scenario Two |
|---|---|---|
| Total Number of Blocks | 6,000 | 35 |
| Duplication Factor | 3 | 3 |
| Total Number of Replicas | 18,000 | 105 |
| Total Number of Nodes | 600 | 15 |
| Average Number of Replicas on One Node | 40 | 7 |
| Minimum Number of Replicas on One Node | 8 | 2 |

| | | |
|---|---|---|
| maximum Number of Replicas on One Node | 70 | 11 |
| Total Number of Racks | 40 | 3 |
| Number of Available Nodes on Rack *i* | 15 | 5 (*i* = 0)<br>3 (*i* = 1)<br>7 (*i* = 2) |

The number of replicas assigned to one node Z spreads over a wide range from 8 to 70 in scenario one. HDFS RPP randomly selects nodes directly instead of selecting a rack of this node first therefore the replica distribution is not significantly affected by the node distribution across racks, but if the rack is randomly selected, the replica distribution would be more uneven when the node distribution is heterogeneous, because the nodes of rack with less nodes would have more replica assignment load than the nodes of rack with more nodes, where the selected rack for the first replica would be selected again for the second replica according to the policy which results in more load on the rack that has fewer nodes.

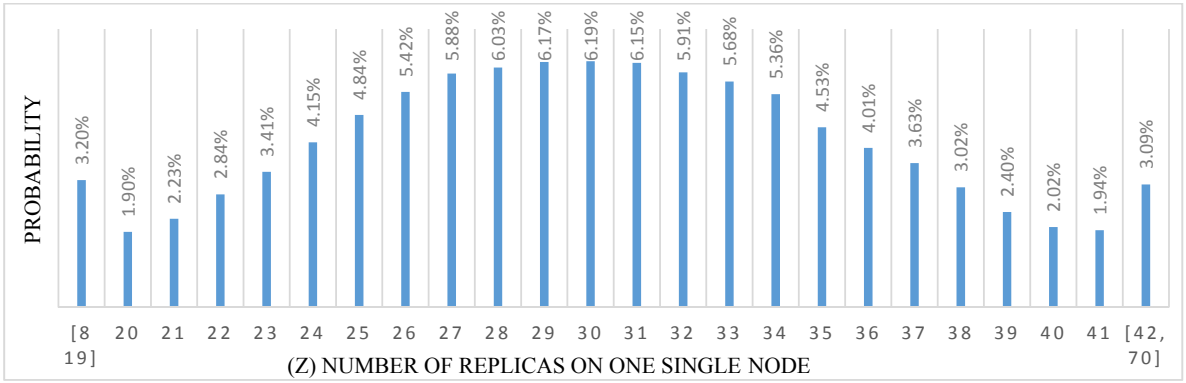


Fig. 3. Probability distribution of ***Z*** in scenario one.

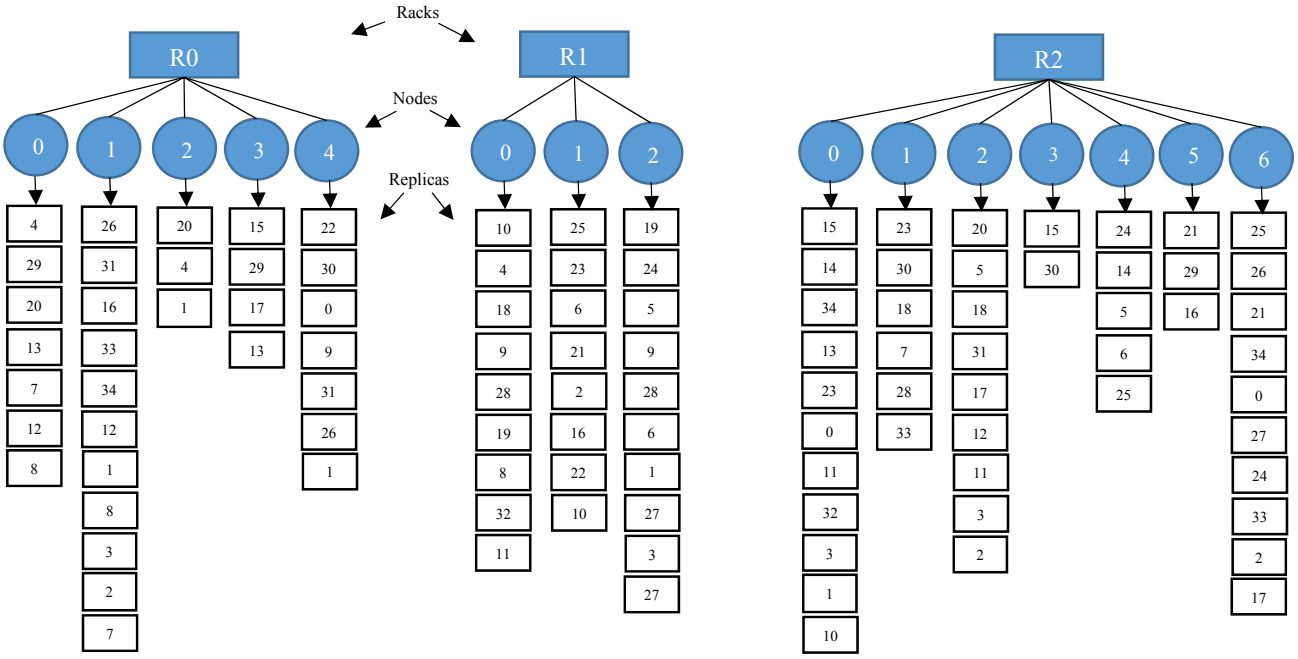


Fig. 4. Replica distribution generated by HDFS RPP in scenario two.

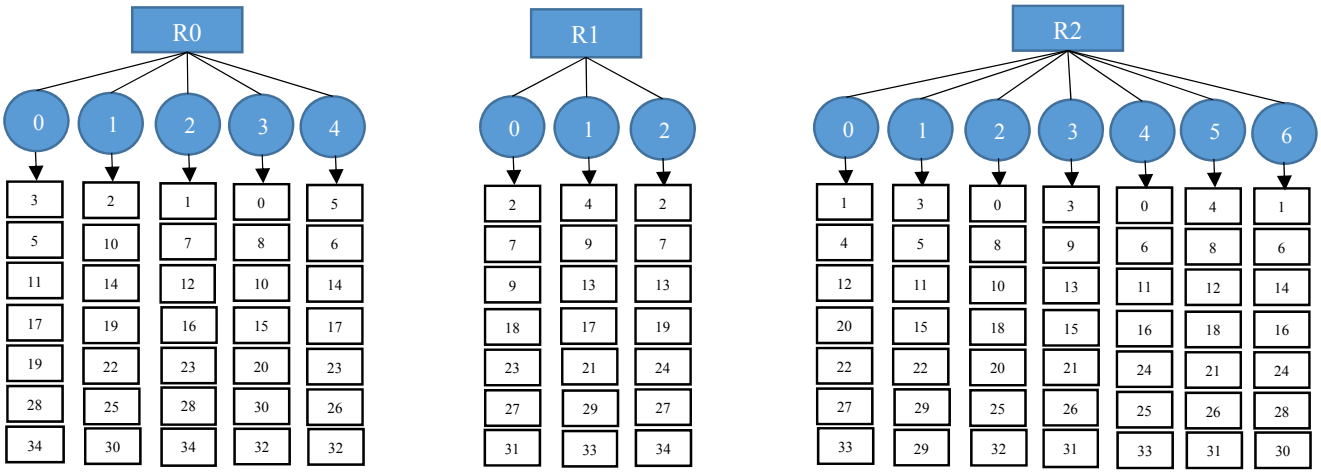


Fig. 5. Replica distribution generated by IDPM in scenario two.

On the other hand, our simulation also confirms that the proposed IDPM can generate perfectly even replica distributions in scenario one, where each node has 30 replicas on it. Because of the large scale this scenario, the replica distributions generated in it cannot be shown in this paper. Therefore, we include scenario two to present the replica distributions generated by both HDFS RPP and PRPP because it is in reduced scale as shown in TABLE I.

Fig. 4 shows one replica distribution generated by HDFS RPP in scenario three. The replica distribution is uneven with number of replicas on one single node ranging from 2 to 11, while the replica distribution generated by IDPM in scenario three, as shown in Fig. 5, is perfectly even, does not need to run balancing utility, and meets all HDFS replica placement requirements.

## V. Conclusion and Future Work

In this paper, we have proposed a new replica placement policy for HDFS. The issue of load balancing is addressed in this paper by evenly distributing replicas to cluster nodes. Therefore, there is no more need for any load balancing utility. IDPM can generate replica distributions that are perfectly even and satisfy all HDFS replica placement rules as confirmed by the simulation results. IDPM is designed for cluster environments in which all cluster nodes have the same computing capabilities. there is an exciting future work for the proposed policy, we plan to adapt IDPM for heterogeneous environments where hardware vary in computing capabilities are involved in cloud storage system.

## References


[1] W. Zeng, Y. Zhao, K. Ou, and W. Song, "Research on Cloud Storage Architecture and Key Technologies," in ICIS 2009, Seoul, Korea, Nov. 2009, pp. 1044–1048.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 10), 2010, pp. 1-10.T. P. Shabeera and S. D. Madhu Kumar, "Bandwidth-aware data placement scheme for Hadoop," in Proceedings of the 2013 IEEE Recent Advances in Intelligent Computational Systems, pp. 64-67 , Trivandrum, Kerala, India, 2013.

[3] [4] D. Howley. Is microsoft's onedrive the best cloud storage service? url=https://www.yahoo.com/tech/microsoft-kills-unlimitedonedrive-accounts-175927221.html.

[4] [5] K. Gai and S. Li. Towards cloud computing: a literature review on cloud computing and its development trends. In 2012 Fourth Int'l Conf. on Multimedia Information Networking and Security, pages 142–146, Nanjing, China, 2012.

[5] K. Gai and A. Steenkamp. A feasibility study of Platform-as-aService using cloud computing for a global service organization. Journal of Information System Applied Research, 7:28–42, 2014.

[6] HDFS Architecture, http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, Accessed Accessed May 20, 2016.

[7] T. P. Shabeera and S. D. Madhu Kumar, "Bandwidth-aware data placement scheme for Hadoop," in Proceedings of the 2013 IEEE Recent Advances in Intelligent Computational Systems, pp. 64-67 , Trivandrum, Kerala, India, 2013.

[8] O. Khan, R. Burns, J. Plank,W. Pierce and C. Huang, "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads." Conference on File and Storage Technologies (FAST), USENIX, 2012.

[9] Q. Zhang, S. Q. Zhang, A. Leon-Garcia, and R. Boutaba, "Aurora: adaptive block replication in distributed file systems," in Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems, Columbus, USA, 2015.

[10] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," in Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, pp. 1-9, Atlanta, USA, 2010.

[11] W. Dai, I. Ibrahim and M. Bassiouni, "A New Replica Placement Policy for Hadoop Distributed File System," 2016 IEEE 2nd International Conference on Big Data Security on Cloud, 2016

[12] W. Dai and M. Bassiouni, "An improved task assignment scheme for Hadoop running in the clouds," in Journal of Cloud Computing, Springer Publishing, Vol. 2:23, pp. 1-16, December 2013.

[13] M. Qiu, Z. Ming, J. Li, K. Gai, and Z. Zong, "Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm," in IEEE Transactions on Computers, Vol. 64, Issue 12, pp. 3528-3540, December 2015.

[14] Y. Li, W. Dai, Z. Ming, and M. Qiu, "Privacy Protection for Preventing Data Over-Collection in Smart City", in IEEE Transactions on Computers, Vol. PP, Issue 99, August 2015.