KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS VOL. 10, NO. 9, Sep. 2016

RDP: A storage-tier-aware Robust Data Placement strategy for Hadoop in a Cloud-based Heterogeneous Environment

Nawab Muhammad Faseeh Qureshi[1] and Dong Ryeol Shin[2]
[1,2]Department of Computer Science and Engineering,
Sungkyunkwan University, Suwon, Korea.
[e-mail : faseeh,drshin@skku.edu]
*Corresponding author : Dong Ryeol Shin

Received March 24, 2016; revised July 5, 2016; accepted July 21, 2016; published September 30, 2016

Abstract

Cloud computing is a robust technology, which facilitate to resolve many parallel distributed computing issues in the modern Big Data environment. Hadoop is an ecosystem, which process large data-sets in distributed computing environment. The HDFS is a filesystem of Hadoop, which process data blocks to the cluster nodes. The data block placement has become a bottleneck to overall performance in a Hadoop cluster. The current placement policy assumes that, all Datanodes have equal computing capacity to process data blocks. This computing capacity includes availability of same storage media and same processing performances of a node. As a result, Hadoop cluster performance gets effected with unbalanced workloads, inefficient storage-tier, network traffic congestion and HDFS integrity issues. This paper proposes a storage-tier-aware Robust Data Placement (RDP) scheme, which systematically resolves unbalanced workloads, reduces network congestion to an optimal state, utilizes storage-tier in a useful manner and minimizes the HDFS integrity issues. The experimental results show that the proposed approach reduced unbalanced workload issue to 72%. Moreover, the presented approach resolve storage-tier compatibility problem to 81% by predicting storage for block jobs and improved overall data block placement by 78% through pre-calculated computing capacity allocations and execution of map files over respective Namenode and Datanodes.

Keywords: MapReduce, Hadoop, Data placement.

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0113-15-0002, Automotive ICT based e-Call standardization and after-market device development)

http://dx.doi.org/10.3837/tiis.2016.09.003 **ISSN : 1976-7277**

1. Introduction

With the advent of big data era, processing of large data-sets became a prime challenge. Many approaches were adopted i.e. serial computing, adhoc computing, distributed computing and parallel computing to build state-of-the-art programming models, which could process large data size with limited computing capacity [1]. Finally, distributed computing in parallel processing approach complied with large data-block management. With the concept of parallel processing in distributed environment, Google proposed MapReduce framework in 2004. Apache Group enhanced features of MapReduce programming model using Google File System (GFS) and introduced an open-source software framework known as Hadoop. Apache Hadoop is a reliable, scalable and efficient ecosystem, which consists of Hadoop Common, Hadoop Distribution File System(HDFS), YARN and MapReduce [2]. Hadoop Common is a set of utilities that support environment modules. YARN schedule jobs and allocate resources to cluster management. MapReduce is a YARN-based programming model, which process large data-sets in parallel computing environment. HDFS is a file system, which process data blocks to their respective repositories.

HDFS consists of Namenode, Datanodes and clients. The client access Namenode to store files on Datanodes. Recently, Hadoop introduced the concept of HDFS federation, which states that a client can access multiple Namenodes to process multiple jobs and store data on Datanodes. The federation supports heterogeneous storage funtionality, through which data blocks are stored over SSD, HDD and RAM storage media. The storage policy permits HDFS to store data blocks in four different types i.e. DISK, SSD, RAM_DISK and ARCHIVE storage types. The DISK is default storage type over Hard Disk Drive (HDD) and Solid State Drive (SSD) storage type. The RAM_DISK is in-place memory storage type and ARCHIVE is high density storage type with less compute power [3].

When a job is executed in Hadoop, the output is generated in the form of data blocks over storage media. HDFS executes Block placement policy function to store resultant data blocks over storage-tier of Hadoop cluster. According to the block placement policy, all block jobs are equally distributed in FIFO order. The Application Master (AM) executes block tasks simulaneously to respective Datanodes. In this way, all the Datanodes execute block jobs at the same time. However in reality, when a fast Datanode finishes job tasks earlier than slow computing Datanode and become idle, Namenode switches unprocessed job tasks of slow Datanode to idle fast Datanode. To this end, Datanodes exchange job tasks between each other and result block job switching time overhead in the distributed computing topology. The fast Datanode receive block job with transfer time overhead while slow Datanode send block job with dispatch time overhead. As a result, overall performance of a Hadoop cluster is severely affected due to unbalanced workloads, network congestions, storage-tier inefficiency and HDFS federation issues.

For Example: A Hadoop Cluster executes data block tasks to Nodes ‘A’, ‘B’ and ‘C’ with default distribution ratio of 4 SSD jobs and 8 HDD jobs. Node ‘A’ finishes SSD and HDD jobs earlier than Node ‘B’ and ‘C’. As a result, Namenode switches unprocessed block jobs of Datanode ‘B’ and ‘C’ to Datanode ‘A’. The results of task execution depict that Namenode allocates 5 SSD and 11 HDD unprocessed block jobs to Datanode ‘A’ having transfer time overhead of 40 seconds, while Datanode B process 2 SSD and 3 HDD block jobs with dispatch time overhead of 25 seconds and Datanode ‘C’ process 1 SSD and 2 HDD block jobs with dispatch time overhead of 22 seconds as also seen from **Fig. 1**.

In order to resolve the unbalanced workload issue over cluster, we propose storage-tier-aware Robust Data Placement (RDP) scheme, which systematically reduce the transfer and dispatch time overhead issue. The proposed approach is divided into three phases. The first phase collects and stores cluster task processing information, predicts storage device type of block jobs and collects capacity computing analytics over Datanodes. The second phase generates configuration files based on first phase data configuraton over Namenode and Datanodes respectively. The third phase executes block jobs map files configuration to the Hadoop cluster.

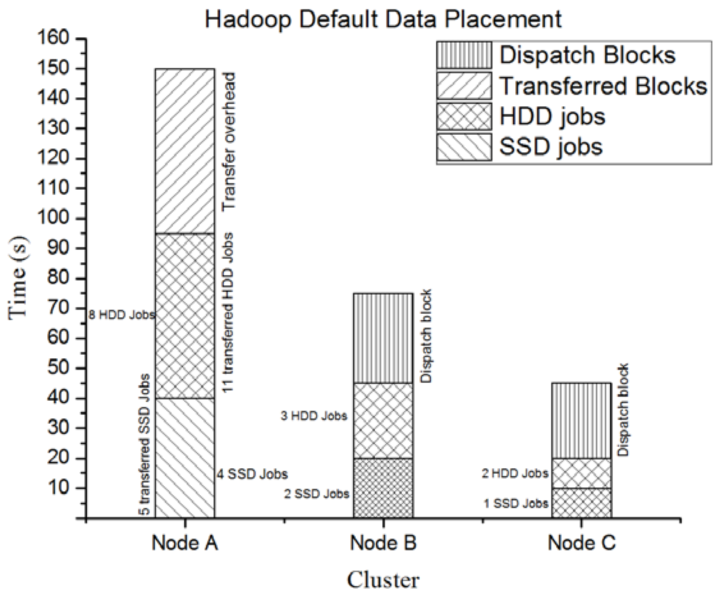

Fig. 1. Hadoop default Data placement

The significant contributions of our approach can be highlighted as:

- An In-place job processing mechanism having Resource Processing Manager (RPM), which includes:
  - A compact novel MessageSync subroutine, which collect and store data block job activity messages over Namenode through enhanced belief propagation model. The MessageSync container provides customized block job information, which help HDFS block placement operation to reduce delay and network congestion while deploying block jobs in Datanodes.
  - A novel block job predication approach through MessageSync data module, which train and predict data blocks to store over job type storage media of Namenode and Datanodes. The MFG reduces block job storage time overhead and dispatches enlisted block jobs to predicted storage media of Datanode.
  - A robust Computing Capacity Ratio (CCR) subroutine to calculate job processing performance of a node and remove unbalanced workload and network congestion by making pre-computed block job processing between slow and fast Datanodes.

- An initial data placement mechanism which includes:
    - Generation of executable pre-computed map files to deploy in-place job processing configuration parameters over respective Namenode and Datanodes.
- A Data Block placement mechanism execution algorithm to process pre-computed map files configuration over Hadoop cluster.

The remaining paper is organized as follows. Section II briefly gives overview of Hadoop architecture and motivation to solve the problem. Section III elaborates previous study and discuss previous approaches to address similar problem and includes list of acronyms used throughout this paper. Section IV briefly explains storage-tier-aware Robust Datablock Placement (RDP) scheme to reduce transfer and dispatch time overhead issue. Section V explains experimental environment and RDP scheme results. The comparative analysis with existing schemes is also included in this section. Finally, section VI shows conclusion with significant contributions and future research directions.

2. Overview and motivation

2.1 Hadoop Cluster

Hadoop is an open source apache project, which provides a data processing framework. When a MapReduce job is submitted to Hadoop cluster, YARN schedules the job and allocates memory resources over cluster. MapReduce split the jobs into various independent tasks and process them sequentially. Tasks are divided into two sets i.e. map tasks and reduce tasks. Initially,map tasks are processed in parallel and output result is sent as an input to reduce task. Furthermore, reduce tasks are processed in parallel and results an output in the pre-defined directory. To this extent, a job is processed over a piece of data and results data block jobs in the cluster. HDFS is responsible to process resultant data blocks to respective Datanodes.

HDFS is an only file system of ecosystem, which consists of Namenode, clients and Datanodes. Namenode stores file system and Datanode metadata. The HDFS architecture explains basic communication pattern among Namenode, clients and Datanodes. A client is a user, which requests file addition and modification to Namenode and process block jobs to Datanodes as seen from **Fig. 2**. Recently [3], Hadoop has introduced the concept of HDFS Federation, which consists of two layers i.e. Namespace and Block Storage service. Namespace consists of directories, files and blocks. The primary job of namespace is to create, delete and modify directories in file-system. Block storage service consists of two parts i.e. Block Management and Storage.

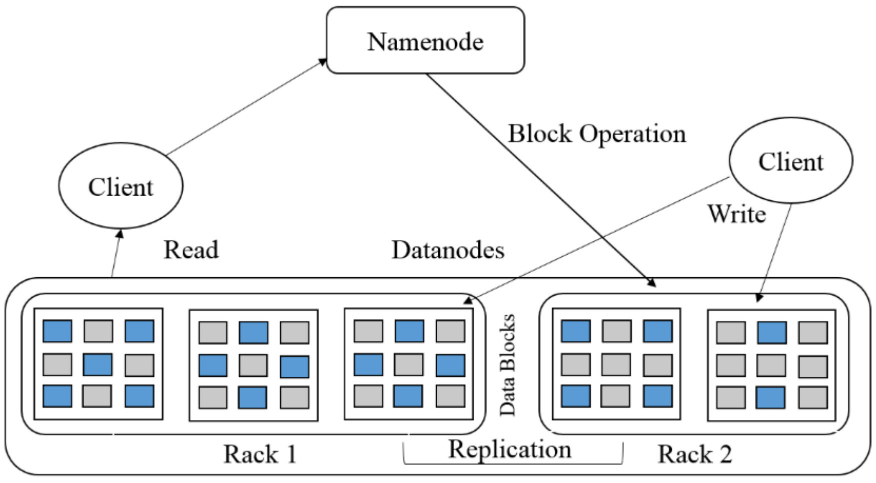

Fig. 2. HDFS Architecture

Block management process block operations i.e. dispatch block jobs to datanodes over block placement policy and create, modify and get block locations. Storage layer is provided by Datanodes to store blocks on local file system and allows read/write access. A single namespace manages block queue through a block pool and together become Namespace volume as seen from **Fig. 3**.

2.2 Motivation

HDFS distribute data blocks equally to all Datanodes. It is assumed that block jobs are processed in ideal condition having same computing capacity over multiple storage-tier Datanodes. However, in real-world environment, clusters do not have same configuration of computing capacity i.e. processing speed and storage media. The cluster consists of multiple Datanodes with different computing capacity and become more unpredictable with multiple storage media i.e. HDD, SSD and RAM as seen from **Fig. 4**.

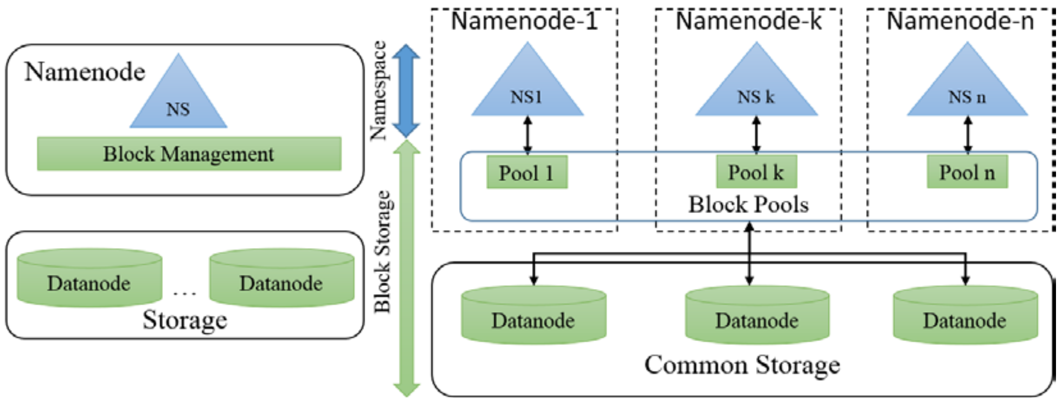

Fig. 3. HDFS Federation

The previous research works lack information about handling multiple storage-tier and HDFS federation environment. The state-of-art schemes are specifically designed for single storage i.e. Hard Disk Drive and manage single Namenode only. Moreover, when a SSD block job is processed, they do not recognize it. As a result, cluster pass numerous error exceptions and data block placement is stopped. This paper proposes RDP to resolve unbalanced workload issues in a systematic order. Our proposed scheme predicts storage media type from block jobs and avoids the issue of being halt in Federation environment.

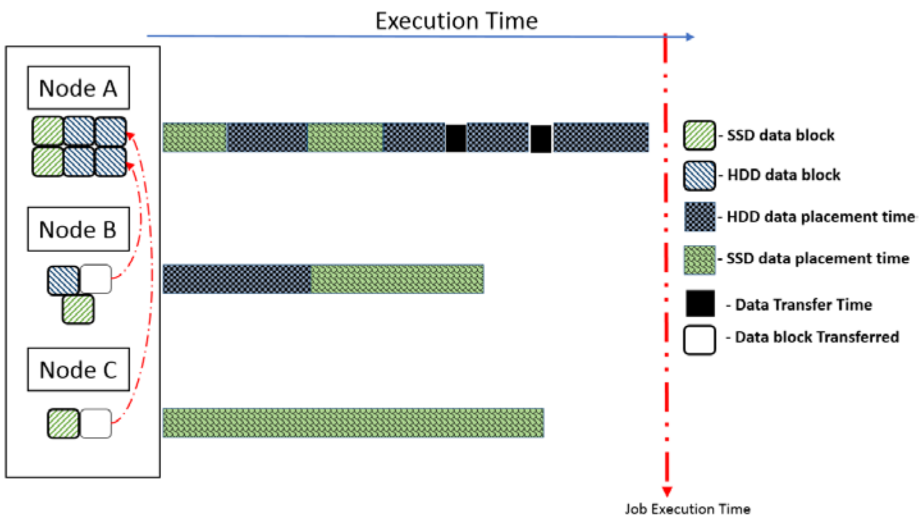

Fig. 4. Hadoop default Data placement

3. RELATED WORK

Researchers have contributed many schemes to optimize data block placement in Hadoop cluster. Lee et al [4] proposed Dynamic Data Placement (DDP) strategy to process data blocks by using an information register known as “Ratio Table”. DDP record job types, compute capacity ratio of each Datanode and store into RatioTable. Moreover, Namenode calculate computing capacity through a heartbeat message and dispatch data blocks to the Datanodes. However, DDP did not explain the mechanism that sent and received datanode information i.e. processing capacity and storage information for a block job. DDP works on single storage media i.e. Hard disk drive (HDD) and generate runtime exceptions when RatioTable handles multiple jobs at a time. Lee et al [5] presented another data block placement approach known as Innovative Data Placement (IDP), which reduced task transfer time by transferring block job to nearest possible Datanode. However, HDFS may consist of hundreds of Datanodes at the same time and IDP had created a huge overhead time between dispatch and transfer of block jobs among Datanodes. Lili et al [6] proposed Partition-based Intelligent data block placement scheme, which processed block jobs in parent-child hierarchal order. The scheme also proposed parent-child hierarchy in Datanodes and compute capacity and calculate disk space utilization in peer-to-peer environment. However, HDFS worked over client/server environment and was not dependent on any accidental failure of a Datanode within parent-to-child hierarchy. Moreover, Datanodes shared block job information, which drastically increased dispatch and transfer times overhead between them.

Table 1. The generation configuration of Data Blocks

Acronym	Description	Acronym	Description
MSM (A)	Message Synchronization Module	**NSV**	Namespace Volume
N	Namenode	**NS**	Namespace
D	Datanode	**P**	Pool
ST	Storage-tier	**AP**	Application
NM	Node Manager	**SR**	Storage & Retreive
NI	Node Information	**YTS**	Yarn Timeline Server
APS	Applications	**RM**	Resource Manager
CA	Containers	**NMS**	Node Manager Information
DI	Disks (HDD)	**AM**	Application Master
SD	SSD	**Zo**	Zoo Keeper
RA	RAM_DISK	**HFI**	HDFS Federation Instance
HS	History	**RPM**	Resource Processing Manager
MRH	Map Reduce History Server	**FIFO**	First In First Out

Changjian et al [7] proposed optimal data placement in MapReduce (OPTAS) to improve data block placement by reconfiguring MapReduce model parameters. OPTAS fulfilled the shuffle time delay gap between map and reduce tasks. However, default data placement policy dispatched block jobs to Datanodes and did not prefer resultant output of MapReduce model. Julio et al [8] proposed a new MapReduce framework (MRA++), which considered heterogeneity of Datanodes with enhanced data distribution, task schedule and job control. However, MRA++ is limited to MapReduce model with default block job placement policy. Lingjun et al [9] proposed a network load sensitive block placement strategy, that worked over default data block placement policy with edited network parameters. The scheme shared network load by shuffling replicas from high group to low group by using node selection algorithm. However, HDFS preferred that initial data blocks must reach to Datanodes. Meanwhile, Datanodes created replicas as per replica policy in Namenode. Yuanquan et al [10] proposed a MapReduce-based data distribution and data migration scheme. Their scheme addressed performance degradation issue during map phase in cluster. Moreover, it processed block jobs on default data block placement policy, therefore, could not contribute significantly at HDFS module. In contrast to all these, our proposed scheme of RDP is compatible to data block placement policy and emphasizes specifically to process block jobs in an efficient way to respective Datanodes.

4. Storage-tier-aware ROBUST DATA BLOCK PLACEMENT (RDP)

In this section, we have explained RDP scheme in detail. The proposed scheme distributes operational process in three phases i.e. (i) In-place job processing, (ii) Initial data placement and, (iii) Data block Placement. The first phase is further categorized in three subsections i.e. (a) MessageSync, (b) Mapping File-to-Storage Generation (MFG) and, (c) Capacity Computing Ratio (CCR).

When a block job is processed into HDFS Federation, it is buffered into Job Buffer. The HFI initiates phase-1 and forward block job parameters from buffer to RPM. The RPM activates the MessageSync module, record block job entry into DataTable and data block parameters are passed to the MFG. MFG predicts storage media for the block job repository into the cluster and forward parameters to the CCR, which calculates computing capacity of Datanode and submit configurations back to HFI. The HFI executes initial data placement phase to deploy phase-1 configurations to map files of Namenode and Datanode. Finally, the NFI executes Data Block Placement process to deploy map file configurations over Namenode and Datanodes, as seen from **Fig. 5**.

4.1 Phase-1: In-place Job processing (Namenode processing)

In this section, RDP performs in-place data block processing. When a block job arrives at the HDFS, the HFI receives the job and send to the Buffer block. The buffer contains two processing queues. The new job is submitted to Queue$_{MFG}$, where block job is passed over MFG module to predict preferred storage media type i.e. HDD, SSD and RAM. The prediction parameters are added to the block job parameters and block job is forwarded to Queue$_{CCR}$, where computing capacity of predicted storage node is calculated. MessageSync container is an information container, which provides cluster block job statistics and provide input data statistics to predict storage media type. Finally, block job is wrapped with computed processing information and sent to NFM as seen from **Fig. 6**.

Algorithm-1 depicts that data blocks of a job i.e. wordcount are processed through phase-1, MFG predicts the suitable storage media for wordcount job and CCR proposes Datanode to deploy wordcount resultant data blocks.

4.1.1 MessageSync Module (MSM)

MessageSync is a data component of RPM, which is used to collect data block processing information from a Hadoop cluster. It is designed to synchronize data block messages of related components to Namenode and Datanode. **Fig. 7** depicts MessageSync module architecture, which request data block job processing information over Namenode and Datanode layer components and receive a response message with processed job parameters. By default, Namenode provides a mechanism to send and receive cluster activity messages but is limited to block generation messages over Namenode and execution messages at Datanodes. Furthermore, data blocks are manually transmitted over single storage-tier node through an administrator control and cluster is not aware of pre-computing node capacity calculation. Therefore, MSM facilitates cluster to keep data block processing information over it. In order to get data block processing information, we have used Belief Propagation method [11], which receives and sends log messages to destination components having small overheads than original messages. Furthermore, we need to perform inference on belief propagation so we opted Message Propagation Model [12], which states that, a message *m* of a variable component *i* having value $\varkappa_i$ with a belief $b_i(\varkappa_i)$ can be propagated from source component *a* to destination component *i* represents likeliness of random variable X_i where $\varkappa_i \in X_i$ by,

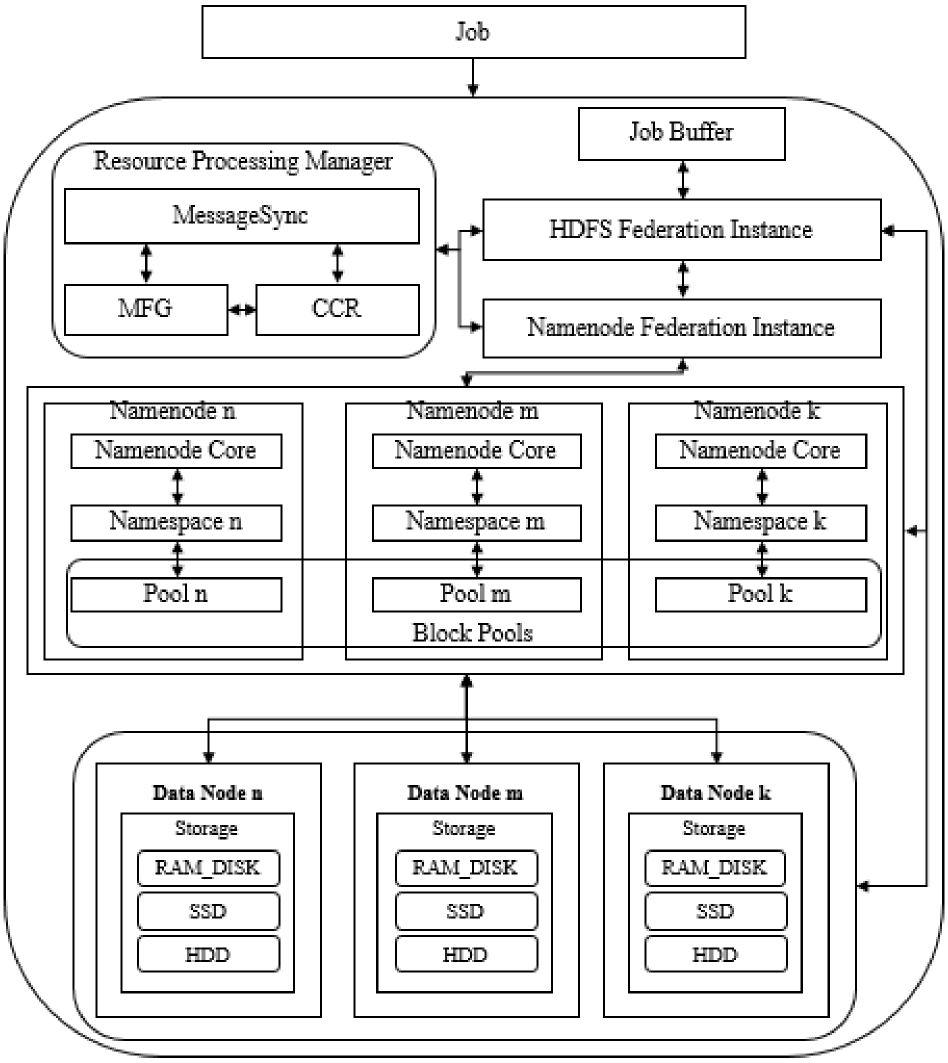

Fig. 5. Robust Data Block Placement Architecture

Algorithm-1. In-Place job Processing Algorithm

```
Initialize $Queue_{MFG}$, $Queue_{CCR}$, $MSM_{Data}$ , HFI, NFM
S = {SSD job, HDD job, RAM job}
$Queue_{MFG} \leftarrow S_i$
$MSN_{DATA} \leftarrow \boldsymbol{m}_{(D_i,N_i)\to L}(\varkappa_L)$
Calculate $R_{MFG}(S_i) = Queue_{MFG(S)_i}\{Si[(argmax_s P(S; O; \lambda). (MSN_{DATA})]\}$
$Queue_{CCR} \leftarrow R_{MFG}(Si)$
    For Each $S_i$ do
        $Queue_{MFG} \leftarrow Queue_{MFG} - 1$
        $Queue_{MFG} \leftarrow S_i$
    End For
Calculate $R_{CCR}(S_i) = Queue_{CCR(S)_i}\{Si\ [((Node\ Ratio_{(D_i,N_i)}; D; B_n)$
$HFI \leftarrow R_{CCR}(Si)$
$NFM \leftarrow HFI$
Produce (HFI,NFM)
```

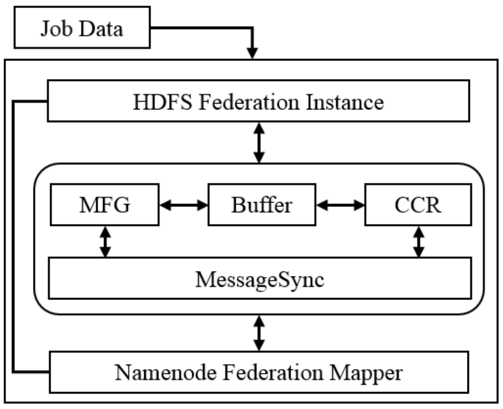

Fig. 6. In-place Block Job Processing

$$message\ m_{a\rightarrow i}(\varkappa_i) \tag{1}$$

After receiving request message, a response message to stated request is passed containing data block information through *RA*, *SD* and *DI* to *ST* using eq (1) as,

$$message\ m_{DI_i\rightarrow ST_i}(\varkappa_{ST_i}) \tag{2}$$
$$message\ m_{SD_i\rightarrow ST_i}(\varkappa_{ST_i}) \tag{3}$$
$$message\ m_{RA_i\rightarrow ST_i}(\varkappa_{ST_i}) \tag{4}$$

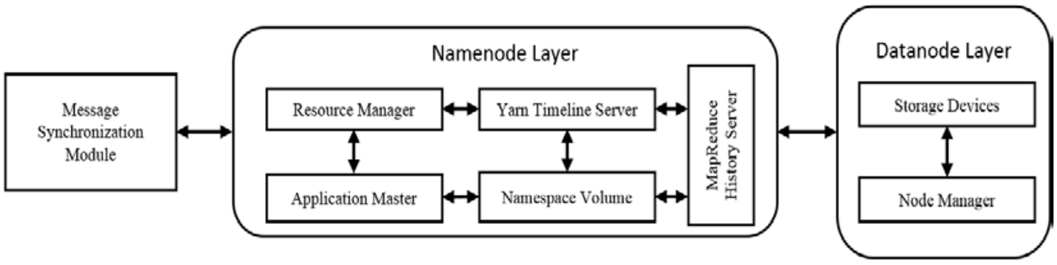

Fig. 7. MessageSync Architecture

After receiving component messages to Storage-tier component, belief of *ST* component is calculated and can be obtained by,

$$b_i(\varkappa_{ST_i}) \propto \prod_{(DI_i,SD_i,RA_i)\in N(ST_i)} m_{(DI_i,SD_i,RA_i)\rightarrow ST_i}(\varkappa_{ST_i}) \tag{5}$$

Similarly, we pass *NI*, *APS* and *CA* response messages to component *NM* as,

$$message\ m_{NI_i\rightarrow NM_i}(\varkappa_{NM_i}) \tag{6}$$
$$message\ m_{APS_i\rightarrow NM_i}(\varkappa_{NM_i}) \tag{7}$$
$$message\ m_{CA_i\rightarrow NM_i}(\varkappa_{NM_i}) \tag{8}$$

After receiving of component messages by the Node Manager component, belief of *NM* component is calculated and can be obtained by,

$$b_i(\varkappa_{NM_i}) \propto \prod_{(NI_i,APS_i,CA_i)\in N(NM_i)} m_{(NI_i,APS_i,CA_i)\rightarrow NM_i}(\varkappa_{NM_i}) \tag{9}$$

Furthermore, Datanode (D_i) layer compiles collective blief of *NM* and *ST* components as,

$$\sum b_i(\varkappa_{NM_i},\varkappa_{ST_i}) \propto \left\{ \begin{array}{l} \prod_{(DI_i,SD_i,RA_i)\in N(ST_i)} m_{(DI_i,SD_i,RA_i)\rightarrow ST_i}(\varkappa_{ST_i}) \\ \prod_{(NI_i,APS_i,CA_i)\in N(NM_i)} m_{(NI_i,APS_i,CA_i)\rightarrow NM_i}(\varkappa_{NM_i}) \end{array} \right\} \tag{10}$$

As we know that, belief of Node D_i is represented as collective belief of components ST_i and NM_i as,

$$b_i(\varkappa_{D_i}) = \sum b_i(\varkappa_{NM_i},\varkappa_{ST_i})$$

Therefore, belief of Node D_i can be written as,

$$b_i(\varkappa_{D_i}) \propto \left\{ \begin{array}{l} \prod_{(DI_i,SD_i,RA_i)\in N(ST_i)} m_{(DI_i,SD_i,RA_i)\rightarrow ST_i}(\varkappa_{ST_i}) \\ \prod_{(NI_i,APS_i,CA_i)\in N\ (NM_i)} m_{(NI_i,APS_i,CA_i)\rightarrow NM_i}(\varkappa_{NM_i}) \end{array} \right\} \tag{11}$$

After we normalize Eq (11) with constant Z, Belief of Node D$_i$ can be represented as,

$$b_i(\varkappa_{D_i}) = \frac{1}{Z}\left\{\begin{array}{l} \prod_{(DI_i,SD_i,RA_i)\in N(ST_i)} m_{(DI_i,SD_i,RA_i)\to ST_i}(\varkappa_{ST_i}) \\ \prod_{(NI_i,APS_i,CA_i)\in N(NM_i)} m_{(NI_i,APS_i,CA_i)\to NM_i}(\varkappa_{NM_i}) \end{array}\right\} \tag{12}$$

Similarly, we formulate collective belief of components *MRH*, *YTS*, *RM* and *NSV* in the Namenode (N_i) layer and can be represented as,

$$\sum b_i(\varkappa_{MRH_i},\varkappa_{YTS_i},\varkappa_{RM_i},\varkappa_{NSV_i}) = \frac{1}{Z}\left\{\begin{array}{l} \prod_{HS_i\in N(MRH_i)} m_{HS_i\to MRH_i}(\varkappa_{MRH_i}) \\ \prod_{(SR_i,AP_i)\to\in N(YTS_i)} m_{(SR_i,AP_i)\to YTS_i}(\varkappa_{YTS_i}) \\ \prod_{(ZO_i,NMS_i,AM_i)\in N(RM_i)} m_{(ZO_i,NMS_i,AM_i)\to RM_i}(\varkappa_{RM_i}) \\ \prod_{(NS_i,P_i)\in N(NSV_i)} m_{(NS_i,P_i)\to NSV_i}(\varkappa_{NSV_i}) \end{array}\right\} \tag{13}$$

As we know that, belief of Node N_i is represented as collective belief of components MRH_i, YTS_i, RM_i and NSV_i as,

$$b_i(\varkappa_{N_i}) = \sum b_i(\varkappa_{MRH_i},\varkappa_{YTS_i},\varkappa_{RM_i},\varkappa_{NSV_i})$$

Therefore, belief of Node N_i having normalization contant z can be written as,

$$b_i(\varkappa_{N_i}) = \frac{1}{Z}\left\{\begin{array}{l} \prod_{HS_i\in N(MRH_i)} m_{HS_i\to MRH_i}(\varkappa_{MRH_i}) \\ \prod_{(SR_i,AP_i)\to\in N(YTS_i)} m_{(SR_i,AP_i)\to YTS_i}(\varkappa_{YTS_i}) \\ \prod_{(ZO_i,NMS_i,AM_i)\in N(RM_i)} m_{(ZO_i,NMS_i,AM_i)\to RM_i}(\varkappa_{RM_i}) \\ \prod_{(NS_i,P_i)\in N(NSV_i)} m_{(NS_i,P_i)\to NSV_i}(\varkappa_{NSV_i}) \end{array}\right\} \tag{14}$$

The belief of Node D_i and N_i can also be observed from **Fig. 8(a)** and **Fig. 8(b)**.

At this stage, we have received response data block information messages from multiple components to Namenode (N_i) and Datanode (D_i) layers. In order to forward response data block information messages to MessageSync component, we calculate joint belief of node D_i and N_i layers. The joint belief represents a logical container i.e. MessageSync where Namenode (N_i) and Datanode (D_i) messages are stored. The logical component L in message propagation model can be expressed as,

$$b_L(\varkappa_L) = b_A(X_A) \tag{15}$$

Where $X_A = \{\varkappa_{D_i}, \varkappa_{N_i} : D_i, N_i \in N(A)\}$ and $\varkappa_L$ is the domain space associated with logical component L as observed from **Fig. 9**.

The domain space $\varkappa_L$ *is equal to* $\left\{(\varkappa_{D_i}, \varkappa_{N_i}) \mid f_A\,(\varkappa_{D_i}, \varkappa_{N_i}) = 1,\ \varkappa_{D_i} \in \chi_{D_i}, \varkappa_{N_i} \in \chi_{N_i}\right\}$ where factor f_A is the bipartite string between all joint components. Therefore, joint belief of component L can be expressed as,

$$b_L(\varkappa_L) = \frac{1}{Z} \prod_{(D_i,N_i)\,\in N(L)} m_{(D_i,N_i)\to L}(\varkappa_L) \tag{16}$$

In order to remove the original messages overhead and provide minimum message transaction

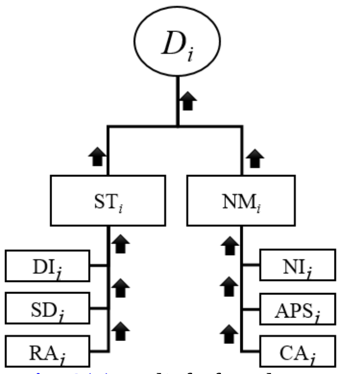

Fig. 8(a). Belief of Node D_i

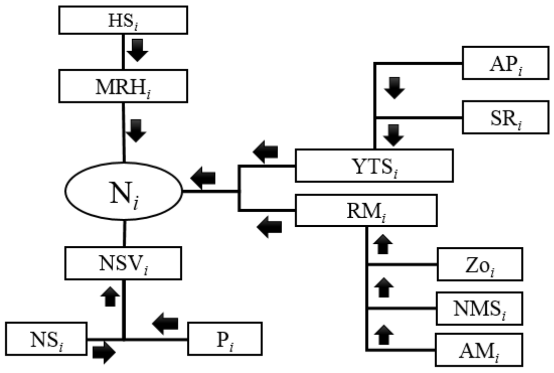

Fig. 8(b). Belief of Node N_i

time, we add factor to Namenode (N_i) and Datanode (D_i). The belief of logical component L with Factor F_A can be expressed as,

$$b_L(\varkappa_L) = \frac{1}{Z} f_A(D_i, N_i) = \left[\begin{array}{l} \left\{\begin{array}{l} \prod\limits_{(DI_i,SD_i,RA_i)\in N(ST_i)} m_{(DI_i,SD_i,RA_i)\to ST_i}\left(\varkappa_{ST_i}\right) \\ \prod\limits_{(NI_i,APS_i,CA_i)\in N(NM_i)} m_{(NI_i,APS_i,CA_i)\to NM_i}\left(\varkappa_{NM_i}\right) \end{array}\right\} \\ \left\{\begin{array}{l} \prod\limits_{HS_i\in N(MRH_i)} m_{HS_i\to MRH_i}\left(\varkappa_{MRH_i}\right) \\ \prod\limits_{(SR_i,AP_i)\in N(YTS_i)} m_{(SR_i,AP_i)\to YTS_i}\left(\varkappa_{YTS_i}\right) \\ \prod\limits_{(ZO_i,NMS_i,AM_i)\in N(RM_i)} m_{(ZO_i,NMS_i,AM_i)\to RM_i}\left(\varkappa_{RM_i}\right) \\ \prod\limits_{(NS_i,P_i)\in N(NSV_i)} m_{(NS_i,P_i)\to NSV_i}\left(\varkappa_{NSV_i}\right) \end{array}\right\} \end{array}\right] \tag{17}$$

After applying Factor F_A to MSM, we simplify Eq (16) and Eq (17) and receive a close form solution as,

$$m_{(D_i,N_i)\to L}(\varkappa_L) = \sum f_A(X_A) \prod_{(D_i,N_i)\in N(L)} m_{(D_i,N_i)\to L}(\varkappa_L) \tag{18}$$

Where $m_{(D_i,N_i)\to L}(\varkappa_L)$ represents collection of block data messages in MessageSync module equivalent to factor F_A filtered messages at respective Namenode and Datanode layers.
Fig. 10 depicts the process of collecting data block information messages into MSM having less overhead than original messages.

4.1.2 Mapping File-to-Storage-Prediction Generator (MFG)

When a MapReduce job is executed i.e. wordcount, the Hadoop generates an output file. The file is stored in HDFS, where it gets divided into multiple data blocks. The generation of data blocks are dependent to distribution size as seen from **Table 2**.

Table 2. The generation configuration of Data Blocks

Job Type	Size	Distribution Size	Block Chunk	No. of Blocks
Wordcount	2 GB	64 MB	2GB = 2048/64=32	32 Blocks
Grep	4 GB	128 MB	8GB = 8192/128= 32	64 Blocks
Wordmean	2 GB	64 MB	2GB = 2048/64=32	32 Blocks

By default, the HDFS store data blocks in DISK storage. When a client requests filesystem to process data blocks in other storage i.e. SSD, the block jobs add an storage overhead to the data block description.

Furthermore, block manager performs lookup process to identify a SSD storage in Datanodes and execute block job with lookup time overhead. The data blocks are dispatched to SSD Datanodes having transfer and lookup time overheads with additional storage overhead. In order to the overheads, the RDP presents a prediction model. The MFG initially trains the block job data present in the MSM and DataTable repository. Secondly, it executes prediction of block jobs over fresh arrival of data block jobs. The “DataTable” is a buffer to store job type of data blocks and MSM is a data block processing information container as discussed in previous section. After the block jobs are trained, prediction model identifies storage with job types in block jobs.

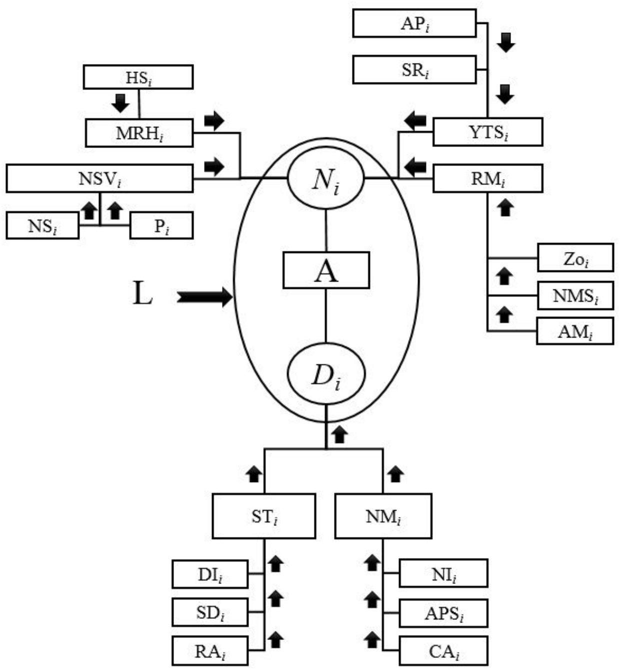

Fig. 9. Belief of Factor A (Logical L)

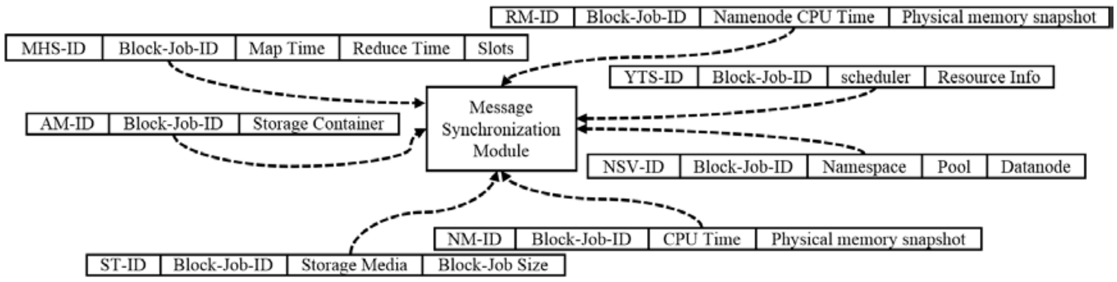
RM-ID | Block-Job-ID | Namenode CPU Time | Physical memory snapshot
MHS-ID | Block-Job-ID | Map Time | Reduce Time | Slots
YTS-ID | Block-Job-ID | scheduler | Resource Info
AM-ID | Block-Job-ID | Storage Container
Message Synchronization Module
NSV-ID | Block-Job-ID | Namespace | Pool | Datanode
NM-ID | Block-Job-ID | CPU Time | Physical memory snapshot
ST-ID | Block-Job-ID | Storage Media | Block-Job Size

Fig. 10. MessageSync Module Information Repository

For this purpose, we have used Hidden Markov Model (HMM) [13], which provides a robust self-learning mechanism. At first, we insert block job sequence into the model as an input feed. Each block job consists of BlockJobId, JobType and Time to complete the process. The data block storage type i.e. SSD, HDD and RAM is extracted from MessageSync data block information container and transition matrix is created to include block storage information. By default, the HMM model include hidden states as $X = \{x_1, x_2\}$, transition probability with conditions as, $A = a_{ij} = \{P[q_{t+1} = x_j | q_t = x_j]\}$, observations state $Y = \{y_1, y_2, y_3, y_4\}$ and emission probability $B = b_{ij}$. After applying our scenario to HMM, we find the observations $O = \{j_1, j_2, j_3, j_4, j_N\}$ are observable states of job type while storage type i.e. SSD, HDD and RAM are hidden states as seen from **Fig. 11**. Now according to definition of HMM (λ), we get,

$$\boldsymbol{\lambda} = (\boldsymbol{\pi}, \boldsymbol{A}, \boldsymbol{B}) \tag{19}$$

Where π is initial state transition probability matrix, A is the transition matrix whose members produce probability of transitioning from one state to another and B is the emission matrix which gives $b_j(Y_t)$.

Let *j* represents six observable jobs $(j_1, j_2, j_3, j_4, j_5, j_6)$ and three hidden storages SSD, HDD and RAM, that complete prediction cycle in the sliding window of time length Δt. After that, we observe that jobs declare storage status and time to predict storage device t_{SD} as seen from **Fig. 12**. The sequence of observations inserted for training the HMM model is, $O = \{Wordcount, Grep, Wordmean\}$. The probability of observations N=3 is computed in view of Forward-Backward and Message propagation algorithms.

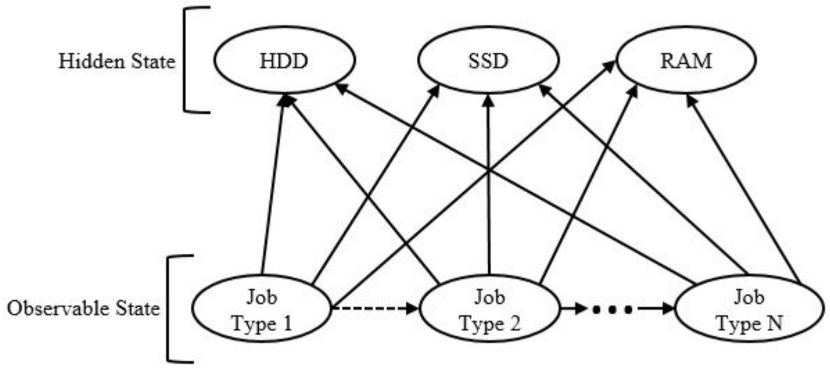

Fig. 11. Mapping Storage Device Type with Job Type

4.1.2.1 Model Training

The MFG model is trained with data block parameters available in MSM and DataTable. The difference among two training sources is, that MessageSync keep data information of Datanodes i.e. blockid, blocklocation, blockstoragedevice and blockjob while DataTable contain current block jobs processed by the Namenode. The model transit from initial state when first round of training is over and calculate probability of block job type from initial state x_1 to end state x_2 with starting probability π=0.33. Furthermore, hidden state transition can be calculated by model parameters $\lambda=(\pi,A,B)$. Moreover, the transition parameters are fetched from MessageSync and then we train model using Expectation-Maximization (EM) [14] algorithm having forward variable 'α' and backward variable 'β'. EM algorithm works over maximizing parameters with maximum likelihood strategy and takes random iterations for best fit as per our model. Therefore, block jobs are processed in several times until job type sequence results best fit parameters for prediction model and produces storage type in dataset.

Initially, we calculated probability of observable sequence (O_t,O_{t+1},O_{t+2}) and then utilized EM algorithm for model learning. EM consists of two steps: (i) Expectation-step (E) and, (ii) Maximization-step (M). Expectation-step calculates storage-likelihood from current estimation and maximization-step calculates parameters maximizing expected storage-likelihood. In this way, EM algorithm compares block jobs for training MessageSync and DataTable elements.

Algorithm-2 depicts that six block jobs are inserted as seed over three hidden states. As per the data trained through MessageSync and DataTable, we find Statepath probability and then update transition probability and emission probability. After executing Path probability again with the valued obtained over EM algorithm, we find block job pair classifications match as [(J_1,SSD), (J_2,HDD), (J_3,HDD), (J_4,SSD), (J_5,SSD), (J_6,RAM)] having t_{SD} time to lookup jobs.

Algorithm-2. Storage Device Type Prediction Algorithm

```
Initialize O = {$J_1, J_2, J_3, J_4, J_5, J_6$}
S = {SSD job, HDD job, RAM job}
M=3
N=6
Initialize Transprobability($A_{ij}$), EmissionProbability($B_{ij}$)
Y = {$Y_1, Y_2, Y_3, Y_4$}
Initialize Probability(StatePath)
Update $B_{ij}, A_{ij}$
Probability(Path) = $B_{ij}$ * Probability(StatePath)
    For Each state $S_j$ do
        Probability(StatePath)[j,i] ← $\max_k$ (Probability(StatePath)[k,i-1].$A_{kj}$.$B_{jyi}$)
        Probability(Path)[j,i] ← arg $\max_k$ (Probability(Path)[k,i-1].$A_{kj}$.$B_{jyi}$)
    End For
    $C_i$ ← arg $\max_k$ (Probability(StatePath))
    $Z_i$ ← $S_i$
    For I ← T-1, …, 1 do
        $C_i$ ← Probability(Path)[$Z_i$,i]
    $Z_i$ ← $C_i$
    End For
Produce (Time,Z)
```

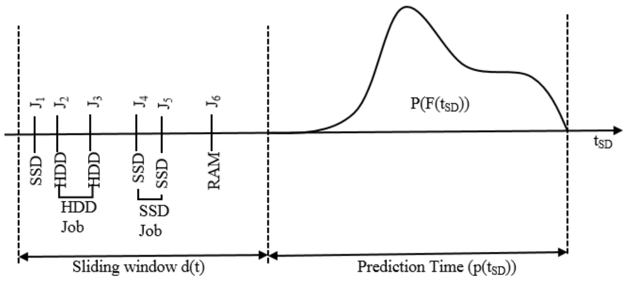

Fig. 12. Storage Device Type prediction over Job Types

4.1.2.2 Prediction

The MFG uses Viterbi algorithm [15] to calculate hidden states of storage device type. At first, the algorithm returns optimal state sequence and reveal hidden states of model *λ=(π,A,B)* with $O = \{j_1, j_2, j_3, j_4, j_N\}$ and finally calculate sequence of states $S_{states}= \{S_1,S_2,\ldots S_n\}$ as,

$$S_{opt} = argmax_s P(S_{states}; O; \lambda) \quad (20)$$

Where S_{opt} is optimal state sequence. Viberti algorithm permit S_{opt} to possess possible optimal paths at each step t that end at N states. At t+1, S retains to increase and optimal path for N is updated. At t+2, S reaches to maxima job-likelihood and optimal path for N is updated and predicts the hidden state i.e. storage device type from enlisted observations O of block jobs as seen from **Fig. 13**.

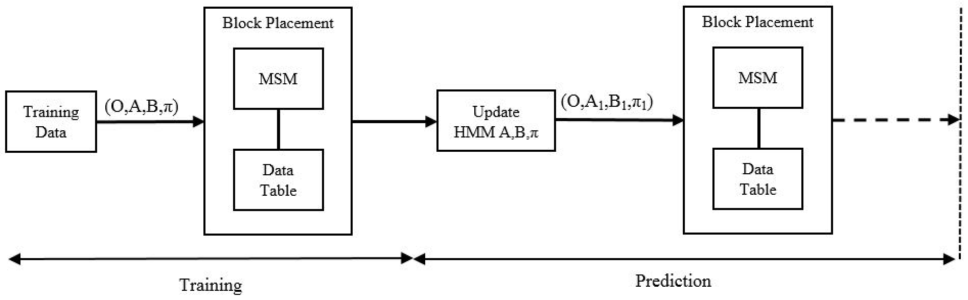

Fig. 13. MFG predicting Storage Device type states using MSM-Data Table

4.1.3 Capacity Computing Ratio (CCR)

The term ‘computing capacity’ of a node elaborates the time interval required to complete a job in a Datanode. In order to compute a Datanode capacity, we calculate available resources of a node i.e. processor, memory and storage device as,

$$Node\ Ratio = \left\{ \begin{matrix} P_{Processor}\left(T_{Total} - (U_{used} + R_{Reserve})\right) : M_{Memory} \\ \left(Total(T_{Total} - (U_{used} + R_{Reserve})\right) : S_{Storage}(S_{SSD} : S_{HDD} : S_{RAM})) \end{matrix} \right\} \quad (21)$$

The Node ratio represents available processing capacity of a node to perform a block job operation in Hadoop cluster.

By definition, we calculate Data Blocks as,

$$B_n = Data\ Block\ No. = \frac{Data\ Size}{Block\ Size} \quad (22)$$

As we know that, Namenode and Datanode computing statistics are stored in the MSM. Therefore, By simplifying Eq (18) and Eq (21), we get Node Ratio of a cluster as,

$$Node\ Ratio_{(D_i,N_i)} = \{Node\ Ratio\ \equiv m_{(D_i,N_i)\to L}(\varkappa_L)\} \quad (23)$$

Where '≡' fetches similar data from MessageSync container. Furthermore, we have utilized block job prediction over storage media as,

$$D = Decision = S_{opt}$$

In order to provide a computing capacity formula, number of B_n over decision D at $Node\ Ratio_{(D_i,N_i)}$, calculate CCR as,

$$CCR = (Node\ Ratio_{(D_i,N_i)}; D; B_n) \quad (24)$$

Where B_n are no. of data blocks over predicted storage D ,processed at $Node\ Ratio_{(D_i,N_i)}$.

4.2 Phase-2: Initial Data placement

In this section, configurations from Phase-1 data block are deployed. Phase-2 prepare configuration files and data block deployment path over the Hadoop cluster. In order to deploy data block configuration, we generate two map files over Namenode and Datanode respective. At first, the map files are configured with phase-1 configuration. Secondly, they calculate Datanode deployment path delay, confirmation of storage media on respective Datanodes and HDFS cross storage integrity delay. The map files are divided into two mapper modules i.e. NFI and DBM. The NFI deploy Phase-1 configuration and calculate path delay, storage media availability and, cross storage integriy delay and forwards completion message to the DBM. The DBM receives NFI confirmation and prepares Datanode for in-place data block processing execution over storage media of Datanodes as seen from **Fig. 14**.

4.2.1 Map Files

The purpose of generating map file is to ensure exact deployment configuration of block job. In-place job processing mechanism generate configuration file for block job deployment. It includes number of blocks, proposed storage and Datanode. In order to deploy the exact proposed configuration, we divide configuration file into two sets i.e. Namenode configuration and Datanode configuration. The Namenode file requires enforcement access and metadata information while Datanode file requires straight deployment information. As a result, Namenode Map Template and Datanode Map Template are deployed as seen from **Fig. 15(a)** and **Fig. 15(b)**.

Algorithm-3 depicts that mapping file templates are initialized. After parsing the in-place job processing information data from phase-1, template file set Namenode,namespace and pool through which datablocks are to be executed. The Namenode template file keep execution template having the Namenode and Datanode information, while Datanode template file keep data block execution path over *n* storage media of *m* Datanode. At the end, at the time of execution of template files in phase-3, time t_{NMF} and time t_{DMF} is calculated.

4.3 Phase-3 Data Block Placement

In this phase, RDP execute data blocks over pre-computed Datanodes. In phase-3, Namenode performs data block execution procedure over map files configured in phase-2. When the data blocks are deployed over cluster, Datanode return time to HFI is $t_A = (t_N + t_D)$, where T_N is the time to generate metadata in Namenode and T_D is the time to place block job at respective Datanode. The total execution time t_A is sent to Namenode and MessageSync logs.

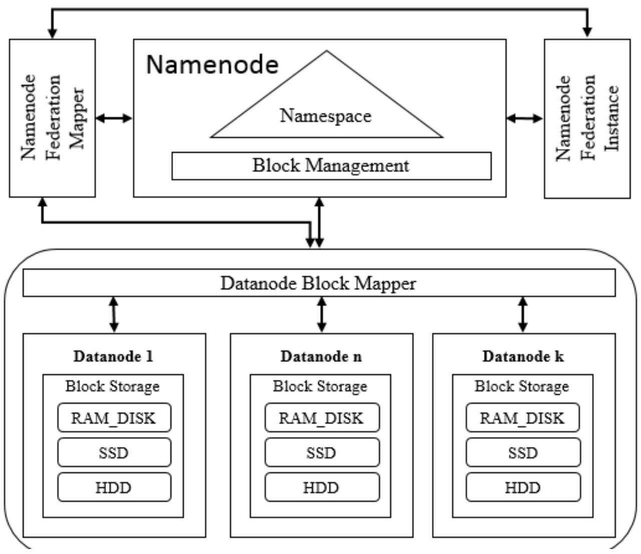

Fig. 14. Initial Data Placement

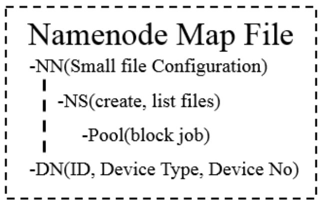
Namenode Map File
-NN(Small file Configuration)
-NS(create, list files)
-Pool(block job)
-DN(ID, Device Type, Device No)

Fig. 15(a). Namenode Template

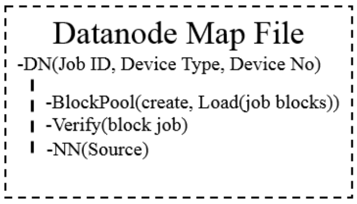
Datanode Map File
-DN(Job ID, Device Type, Device No)
-BlockPool(create, Load(job blocks))
-Verify(block job)
-NN(Source)

Fig. 15(b). Datanode Template

Algorithm-4 initializes Queue$_{BlockPlacement}$ and execute map file configuration over cluster. The number of data blocks executed over phase-2 map files are logged into the MessageSync and return t_A to the cluster.

Algorithm-3. Initial Data Placement Algorithm

```
Initialize $DMF_{MappingFile}$, $NMF_{MappingFile}$, NFM, DBM,
NFI, Namespace, Pool
$NFM = \{In\text{-}Place_{JobData}\}$
Storage = (SSD,HDD,RAM)
$NFI \leftarrow NFM$
$R_{(Namespace,Pool)} = Map\ Namenode_{(Namespace,Pool)}\ (NFI_{JobData})$
$NMF_{MappingFile} \leftarrow R_{(Namespace,Pool)}$
$R_{(Storage)} = Map\ Datanode_{(SSD,HDD,RAM)}\ (NFI_{JobData})$
$DMF_{MappingFile} \leftarrow R_{(Storage)}$
Produce $(NMF_{MappnigFile}, DMF_{MappingFile})$
```

Algorithm-4. Data Block Placement Algorithm

```
Initialize $Queue_{BlockPlacement}$, $HDFS_{Federation}$
Files = $(NMF_{MappnigFile}, DMF_{MappingFile})$
Block Jobs = $\{BJ_n\}$
$Queue_{BlockPlacement} \leftarrow BJ_n$
$Deploy_{(BJ_n)} = Queue_{BlockPlacement}\ [Files_{(Namenode,Datanode)}]$
    For Each $BJ_n$ do
        $Queue_{BlockPlacement} \leftarrow Queue_{BlockPlacement} - 1$
        $Queue_{BlockPlacement} \leftarrow BJ_n$
    End For
$HDFS_{Federation} \leftarrow Deploy_{(BJ_n)}$
Produce (Time, $HDFS_{Federation}$)
```

5. Experimental Work

In this section, we present evaluation of RDP through an experimental environment as shown in **Table 3**.

Table 3. Cluster Configuration

Machine	Specifications	No. of VM	
Intel Xeon E5-2600 v2	8 CPUs, 32GB memory, 1T Disk and 128 GB SSD	3	1 Master Node, 2 Datanodes
Intel core i5	4 Core, 16GB memory, 1T Disk and 128 GB SSD	2	2 Datanodes
Hadoop	Hadoop-2.7.2 (stable)		
Virtual Machine Management	VirtualBox 5.0.16		

5.1 Environment

We have used Intel Xeon with 8 CPUs, 32GB memory and three storage facilities i.e. HDD 1TB disk, 128 GB Samsung SSD and temfs utility as RAM_DISK storage. Similarly, we also use Intel core i5 with 4 Core, 16GB memory and three storage facilities i.e. HDD 1TB disk, 128 GB Samsung SSD and temfs utility as RAM_DISK storage. For virtual environment, we used virtualbox 5.0.16 for installing 5 virtual machines on discussed machine configurations as seen from **Table 4**.

Table 4. Virtual Machines Configuration over Hadoop Cluster

Node	CPU	Memory	Disk	Configuration
Master Node	6	16 GB	HDD,SSD,RAM	Intel Xeon
Slave1	2	4GB	HDD,SSD,RAM	Intel Xeon
Slave2	2	4GB	HDD,SSD,RAM	Intel Core i5
Slave3	2	4GB	HDD,SSD,RAM	Intel Core i5
Slave4	2	4GB	HDD,SSD,RAM	Intel Core i5

5.2 Experimental Dataset

The dataset used to process experimental work includes (i) 20,000 data block request messages (ii) 20,000 data block response messages (iii) 640 SSD wordcount data blocks of 64MB (40GB size) (iv) 640 HDD wordcount data blocks (40GB size) (v) 64 RAM wordcount data blocks (1 GB size) (vi) 640 SSD grep data blocks (40GB size) (vii) 640 HDD grep data blocks (40GB size) (viii) 64 RAM grep data blocks (1 GB size).

5.3 Experimental Results

The experiments conducted to evaluate our scheme are (i) Message Request and Response acknowledgement of MSM (ii) Block job predictions (iii) Node ratio (iv) In-place job processing execution (v) Initial Data Placement execution (vi) Data Block Placement execution (vii) DISK and SSD block job processing (viii) Network congestion and Block job device utilization

5.3.1 Message Request and Response acknowledgement of MSM

The purpose of MSM is to collect data block processing information i.e. total number of blocks processed at storage-tier, total number of data blocks processed by Node Manager with detail statistics of memory, secondry storage space utilization and processor computation. MSM utilizes Hadoop cluster subroutines of generating data messages. The MSM program tool initially send data request Request_Acknowledgement (RE) to cluster and receive Response_Acknowledgement (RA) to the MSM container. The messages are sent and received over components i.e. *ST*, *MRH*, *NSV*, *YTS* and *NM*. MSM keep individual tables for the components and stores data in FIFO order. Hadoop cluster exchange information messages between Namenode and Datanodes at bandwidth $0.5 \leq \text{Bandwidth} \geq 5$ MB/s. However, message propagation strategy reduces message overhead from original message and reduce bandwidth utilization by 72%. The bandwidth utilized to request and receive data block information messages successfully can be observed from **Fig. 16(a), (b), (c)** and **(d)**.

5.3.2 Block job predictions

After training the model for available dataset, we run multiple simulations of predictions for storage type jobs. At first hour of prediction, we predict 348 SSD block jobs, 233 HDD block jobs and 12 RAM jobs out of 640 block jobs. The jobs which could not be predicted due to missing headers of destination Datanodes are 47. In the second hour of prediction, we find 210 SSD jobs, 170 HDD jobs and no RAM jobs. The jobs which could not be predicted due to missing headers of destination Datanodes are 260. In the third hour of simulation, we find 178 SSD jobs, 218 HDD jobs and no RAM jobs. Again, the unpredicted jobs due to missing headers of destination Datanodes are 244. In order to identify individual storage media percentile of predicted data blocks, we calculate average of total predicted SSD, HDD and RAM data blocks and found 92% match for SSD with 3% simulation error, 83.4% match for HDD with 5% simulation error and 81.6% match for RAM data blocks with 7% simulation error, as seen from **Fig. 16(e), (f), (g)** and **(h)**.

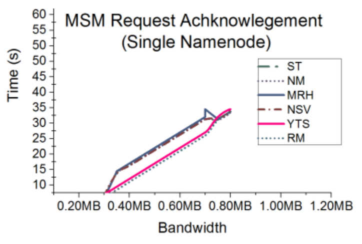

Fig. 16(a). RE Single Node

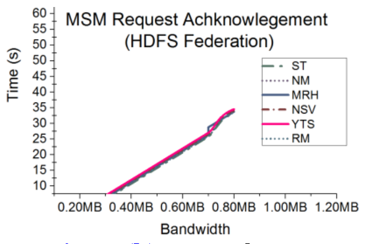

Fig. 16(b). RE Federation

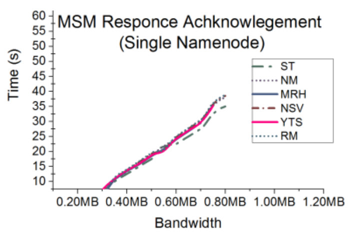

Fig. 16(c). RS Single Node

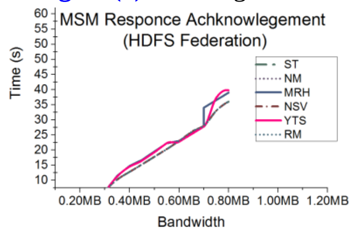

Fig. 16(d). RS Federation

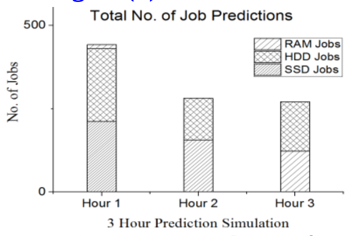

Fig. 16(e). Total No. of Predictions

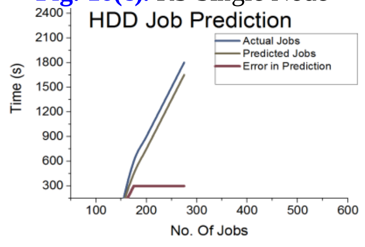

Fig. 16(f). HDD Job Prediction

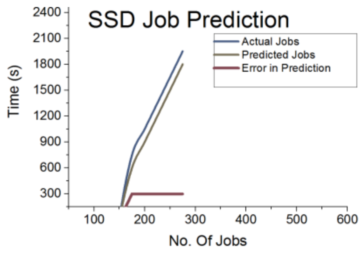

Fig. 16(g). SSD Job Prediction

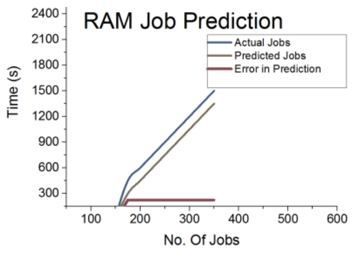

Fig. 16(h). RAM Job Prediction

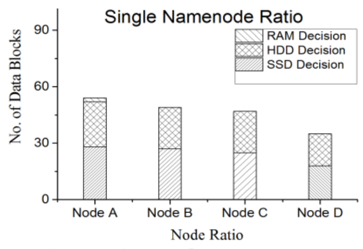

Fig. 16(i). Node Ratio over Single Namenode

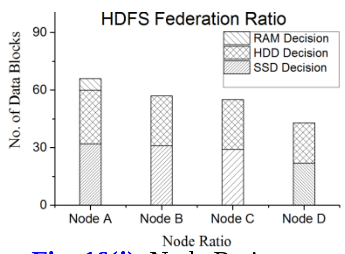

Fig. 16(j). Node Ratio over HDFS Federation

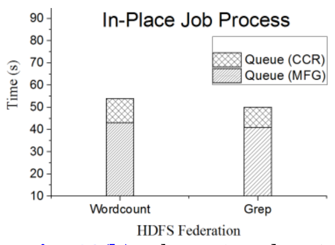

Fig. 16(k). Phase-1 Federation Execution

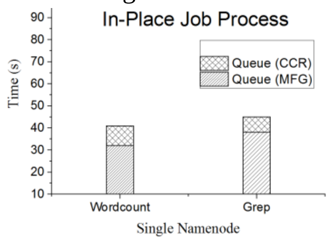

Fig. 16(l). Phase-1 Single Execution

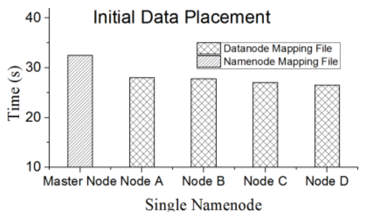

Fig. 16(m). Node Ratio (Single Namenode)

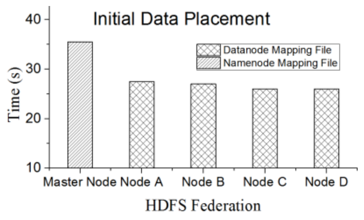

Fig. 16(n). Node Ratio (HDFS Federation)

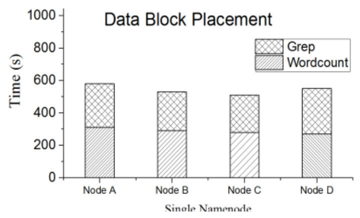

Fig. 16(o). Phase-1 Federation Execution

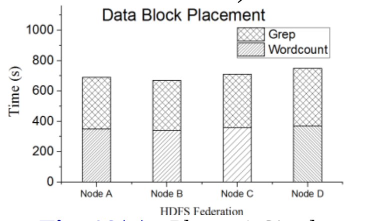

Fig. 16(p). Phase-1 Single Execution

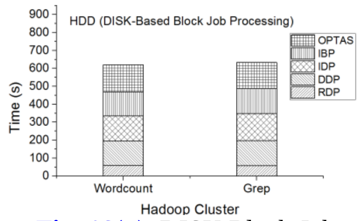

Fig. 16(q). DISK Block Job Processing

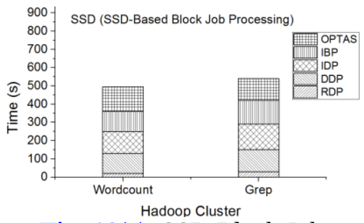

Fig. 16(r). SSD Block Job Processing

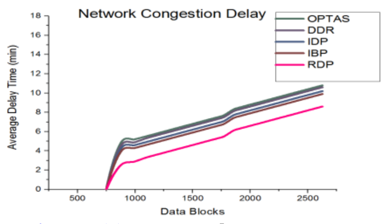

Fig. 16(s). Network Congestion

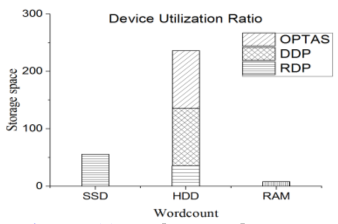

Fig. 16(t). Job Based Device Utilization

5.3.3 Node Ratio

The purpose of calculating Node Ratio is to evaluate a node's computing capacity and propose data block placement according to node ratio of Node A,B,C and D. To this end, we found that three nodes are having SSD and HDD storage capacity. Node A is the only fast node having RAM block job processing capacity, node B is also a fast node, Node C is 4% slow than node A and B , and Node D is the slowest node. After observing CCR values of Node A,B,C and D, we deploy SSD,HDD and RAM decision block jobs and found that Node A processed 30 SSD data blocks, 24 HDD data blocks and 2 RAM data blocks. Node B processed 30 SSD data blocks, 22 HDD data blocks. Node C processed 29 SSD data blocks, 24 HDD data blocks and Node D processed 20 SSD data blocks, 15 HDD data blocks simultaneously. In this way, we observe that when a Hadoop cluster process data blocks over defined capacity parameters, it results in an optimized utilization of storage devices on respective Datanodes. We also observed that storage media in respective Datanodes performed a 72% faster deployment of data blocks than normal cluster deployment due to guided environment and depicted an improvement of 48% secondry storage deployment than random writing of data block on each storage media of the cluster as seen from **Fig. 16(i)** and **(j)**.

5.3.4 In-place job processing execution

In the first phase, we executed two Mapreduce programs i.e wordcount and grep. The data blocks are predicted for storage media in Queue$_{MFG}$ and perform node capacity computing procedure in Queue$_{CCR}$. We observed that the processing of Queue$_{MFG}$ consumed 61% more time than Queue$_{CCR}$. Furthermore, the data blocks generated by wordcount program were consuming 43 seconds in Queue$_{MFG}$ and 9 seconds in Queue$_{CCR}$. Similarly, the data blocks generated by grep program were consuming 41 seconds in Queue$_{MFG}$ and 8 seconds in Queue$_{CCR}$. Meanwhile, average execution time of a Single Namenode cluster consumed 35% less time than average execution time of HDFS Federation as seen from **Fig. 16(k)** and **(l)**.

5.3.5 Initial Data Placement execution

The phase-2 configure map files over Namenode and Datanodes. When a write_Nmap request appear in the Namenode, NMF_Invoke method receives storage media and pre-computed node parameters and pass the reference parameters to NMF_Create method. The NMF_Create method generates a map file with Namenode enforcement parameters at NMF. Similarly, when a write_Dmap request appear in Datanode, DMF_Invoke method receives the configuraton and enforce parameters to DMF_Create method. The DMF_Create method generate a map file having Datanode storage media configurations to deploy the data blocks and pre-computed node configurations to deploy data blocks to Node A,B,C and D in a balanced manner. Finally, the mapping_file_instance is generated, which transfers deployment configuration to Block_Manager class for a cluster reference. The average file generation time observed is 33 seconds at MasterNode, 28 seconds at Node A, 27 seconds at Node B, 28 seconds Node C and 26 seconds at Node D as seen from **Fig. 16(m)** and **(n)**.

5.3.6 Data Block Placement execution

In phase-3, we observed the deployment of data blocks related to wordcount and grep programs. It is observed that data blocks are divided into Nodes A, B, C and D in a balanced manner. The execution time to place datablocks in Node A is 299 seconds for wordcount and 247 seconds for grep. The execution time to place data blocks in Node B is 296 seconds for wordcount and 243 seconds for grep. The execution time to place data blocks in Node C is 297 seconds for wordcount and 245 seconds for grep. The execution time to place data blocks in Node D is 298 seconds for wordcount and 246 seconds for grep. We have observed from **Fig. 1** that a cluster create unbalanced workload, network congestion, improper storage media utilization and HDFS integrity issues. The default Hadoop data block placement scheme consumed an average of 58 seconds transfer time overhead at Node A to process data blocks of other slow nodes B and C. The Node B consumed 38 seconds and Node C consumed 43 seconds to dispatch unprocessed data blocks to Node A. The proposed RDP scheme places data blocks simultaneously to Datanodes A, B, C and D, having almost same average execution time and reduce unbalanced workload to 72%, storage-tier competibility issue to 81% and overall average improvement of 78% data block placement process than default scheme, as seen from **Fig. 16(o)** and **(p)**.

5.3.7 DISK and SSD block job processing

We performed comparitive analysis of RDP with existing schemes i.e. OPTAS, IBP, IDP and DDP. In order to identify the performance of previous schemes, we deployed the schemes to Hadoop cluster and executed two programs i.e. wordcount and grep. Initially, we executed wordcount program having 10 data blocks of 64M to HDD (Disk Drive) and evaluated that OPTAS process in 132 seconds, IBP process in 143 seconds, IDP process in 121 seconds, DDP process in 152 seconds and RDP process in 61 seconds. Therefore, RDP performed averagely 61% better than previous schemes. Similarly, we executed grep program having 10 data blocks of 64M to HDD and evaluated that OPTAS process in 127 seconds, IBP process in 149 seconds, IDP process in 118 seconds, DDP process in 146 seconds and RDP process in 56 seconds. Therefore, RDP perform averagely 43% better than previous schemes. Secondly, we executed wordcount program having 10 data blocks of 64M to SSD (Solid State Drive) and evaluated that OPTAS process in 128 seconds, IBP process in 131 seconds, IDP process in 117 seconds, DDP process in 136 seconds and RDP process in 39 seconds. Therefore, RDP perform averagely 71% better than previous schemes. Similarly, we executed grep program having 10 data blocks of 64M to SSD (Solid State Drive) and evaluated that OPTAS process in 131 seconds, IBP process in 142 seconds, IDP process in 122 seconds, DDP process in 159 seconds and RDP process in 47 seconds. Therefore, RDP perform averagely 73% better than previous schemes as seen from **Fig. 16(q)** and **(r)**.

5.3.8 Network congestion and Block job device utilization

We performed comparitive analysis to calculate average delay of data block placement among Datanodes in the network environment. In order to calculate data block packet delay, we could use simulation tools like NS-2 but such tools do not provide Hadoop data block packet libraries. Therefore, we calculated the delay by executing program at multiple Datanodes. We executed 3000 random data blocks over Hadoop cluster to perform data placement over OPTAS, DDR, IDP, IBP and RDP schemes. We evaluated that OPTAS and DDR are having default Hadoop parameters and were consuming an average delay time of 12 and 11 mins respectively. IDP consumed 9 mins while IBP consumed 8.6 mins. Our proposed RDP scheme consumed 7.9 mins for processing data blocks on Hadoop cluster as seen from **Fig. 16(s)**. Thus, RDP has performed averagely 8% better than previous schemes.

We further performed comparitive analysis to calculate storage media utilization ratio over OPTAS, DDP and RDP. The purpose of computing device utilization ratio is to calculate and compare scheme awareness of storage-tier in Hadoop cluster. For this purpose, we executed 300 MB data blocks of wordcount program and observed that OPTAS and DDP execute HDD (Disk Drive) data blocks, while RDP executed SSD and RAM data blocks as seen from **Fig. 16(t)**. Therefore, we concluded that, RDP is the only scheme among all state-of-art schemes that utilizes all data blocks for available storage devices i.e. HDD, SSD and RAM.

6. Conclusion

This paper proposes Robust Data Placement (RDP) scheme to efficiently process data blocks in Hadoop cluster. The RDP scheme systematically process data blocks by firstly generating in-place job processing configurations through MSM, MFG and CCR modules. Secondly, it deploys configurations to Initial Data Placement through map files. Finally, it process data blocks through deploying map files in respective Namenode and Datanodes. The experiments have shown that RDP is an efficient scheme and accelerate Hadoop cluster by reducing unbalanced workload, data block network congestion, efficient usage of storage media and decreased HDFS integrity problems.

In the future, we will explore multi homing issues, which enable Hadoop cluster to perform data placement in multiple networks at the same time.

References

[1] J. K. Verma and C. P. Katti, "Study of Cloud Computing and its Issues: A Review," *Smart Computing Review*, vol. 4, no.5, pp. 389-411, October 2014. Article (CrossRef Link).

[2] S. Khalil, S. A. Salem, S. Nassar and E. M. Saad, “MapReduce performance in heterogeneous environments: a review,” *International Journal of Scientific & Engineering Research*, vol. 4, no. 4, pp. 410-416, 2013. Article (CrossRef Link).

[3] Apache Hadoop Documentation. Article (CrossRef Link).
[4] C. W.Lee, K. Y. Hsieh, S. Y. Hseih and H. C. Hsiao, "A Dynamic Data Placement Strategy for Hadoop in Heterogeneous Environments," *Journal of Big Data Research*, vol. 1, pp. 14-22, 2014. Article (CrossRef Link).
[5] C. Lee, H. Huang and S. Hsieh, "IDP: An Innovative Data Placement Algorithm for Hadoop Systems," in *Proc. of Intelligent Systems and Applications: Proceedings of the International Computer Symposium (ICS)*, IOS Press, pp. 49, 2015. Article (CrossRef Link).
[6] S. Lili, Y. Yang, Z. Xiong and X. Zhao, "Intelligent Block Placement Strategy in Heterogeneous Hadoop Clusters," *Journal of Convergence Information Technology*, vol. 8, no. 8, 2013. Article (CrossRef Link).
[7] W. Changjian, Y. Qin, Z. Huang, Y. Peng, D. Li and H. Li, "OPTAS: Optimal Data Placement in MapReduce," in *Proc. of Int. Conf. on Parallel and Distributed Systems Parallel and Distributed Systems (ICPADS)*, pp. 315-322, 2013. Article (CrossRef Link).
[8] J. C. S. Anjos, I. Carrera and W. Kolberg, A. Luis Tibola, L. B. Arantes and C. R. Geyer, "MRA++: Scheduling and data placement on MapReduce for heterogeneous environments," *Future Generation Computer Systems*, vol. 42, pp. 22-35, 2015. Article (CrossRef Link).
[9] L. Meng, W. Zhao, H. Zhao and Y. Ding, "A Network Load Sensitive Block Placement Strategy for HDFS," *KSII Transactions on Internet and Information Systems*, vol. 9, no. 9, pp. 3539-3558, 2015. Article (CrossRef Link).
[10] Y. Fan, W. Wu, H. Cao, H. Zhu, X. Zhao and W. Wei, "A Heterogeneity-aware Data Distribution and Rebalance Method in Hadoop Cluster," in *Proc. of ChinaGrid Annual Conference (ChinaGrid)*, IEEE, pp. 176-181, 2012. Article (CrossRef Link).
[11] J. S. Yedidia, "Message-passing algorithms for inference and optimization," *Journal of Statistical Physics*, vol. 145, no. 4, pp. 860-890, 2011. Article (CrossRef Link).
[12] M. Khosla, "Message Passing Algorithms," *PHD thesis*, 9 , 2009. Article (CrossRef Link).
[13] Z. Ghahramani, "An introduction to hidden Markov models and Bayesian networks," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 15, no. 1, pp. 9-42, 2001. Article (CrossRef Link).
[14] Ajit Singh, EM Algorithm, 2005. Article (CrossRef Link).
[15] G. D. Forney, "The viterbi algorithm," in *Proc. of the IEEE*, vol. 61, no. 3, pp. 268-278, 1973. Article (CrossRef Link).

Nawab Muhammad Faseeh Qureshi received the B.E degree in Software Engineering from Mehran University of Engineering and Technology, Pakistan, in 2006 and M.E degree in Information Technology from Mehran University of Engineering and Technology, Pakistan, in 2013. Currently, he is working for Ph.D. in Department of Computer Science and Engineering at Sungkyunkwan University, Korea. His research interests include big data platform and cloud computing.

Dong Ryeol Shin received the B.S., M.S., and Ph.D. degrees in Electrical Engineering from the Sungkyunkwan University in 1980, the Korea Advanced Institute of Science and Technology (KAIST) in 1982, and the Georgia Institute of Technology in 1992, respectively. During 1992-1994, he had worked for Samsung Data Systems, Korea, where he was involved in the research of Intelligent Transportation Systems. Since 1994, he has been with the Department of Computer Science and Engineering at Sungkyunkwan University where he is currently a Professor in Network Research Group. His current research interests lie in the areas of mobile network, ubiquitous computing, cloud computing, and bioinformatics. And he is actively involved in the security of vehicular area networks, and the implementation and analysis of big data platform, applicable to 3D image processing of robotic arms.