J. Parallel Distrib. Comput. 74 (2014) 2770–2779

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Enhancing throughput of the Hadoop Distributed File System for interaction-intensive tasks

Xiayu Hua *, Hao Wu, Zheng Li, Shangping Ren

Department of Computer Science, Illinois Institute of Technology, 10 West 31st, Chicago, IL, USA

HIGHLIGHTS

- Analyzed the performance degradation of HDFS caused by interaction-intensive tasks.
- Designed a two-layer structure to improve the performance of handling I/O request.
- Integrated caches to reduce the overhead of accessing interaction-intensive files.
- Developed a PSO-based storage allocation algorithm to improve the I/O throughput.
- Designed a set of experiments to evaluate the performance of the proposed methods.

ARTICLE INFO

Article history:
Received 16 December 2013
Received in revised form
22 March 2014
Accepted 30 March 2014
Available online 3 April 2014

Keywords:
HDFS
Interaction intensive task
Cache
Hierarchical structure
PSO
Storage allocation algorithm

ABSTRACT

The Hadoop Distributed File System (HDFS) is designed to run on commodity hardware and can be used as a stand-alone general purpose distributed file system (Hdfs user guide, 2008). It provides the ability to access bulk data with high I/O throughput. As a result, this system is suitable for applications that have large I/O data sets. However, the performance of HDFS decreases dramatically when handling the operations of *interaction-intensive files*, i.e., files that have relatively small size but are frequently accessed. The paper analyzes the cause of throughput degradation issue when accessing interaction-intensive files and presents an enhanced HDFS architecture along with an associated storage allocation algorithm that overcomes the performance degradation problem. Experiments have shown that with the proposed architecture together with the associated storage allocation algorithm, the HDFS throughput for interaction-intensive files increases 300% on average with only a negligible performance decrease for large data set tasks.

1. Introduction

The Hadoop Distributed File System (HDFS) is designed as a massive data storage framework and serves as the storage component for the Apache Hadoop platform. This file system is based on commodity hardware and provides highly reliable storage and global access with high throughput for large data sets [15]. Because of these advantages, the HDFS is also used as a stand-alone general purpose distributed file system and serves for non-Hadoop applications [6].

However, the advantage of high throughput that the HDFS provides diminishes quickly when handling *interaction-intensive* files, i.e., files that are of small sizes but are accessed frequently. The reason is that before an I/O transmission starts, there are a few necessary initialization steps which need to be completed, such as data location retrieving and storage space allocation. When transferring large data, this initialization overhead becomes relatively small and can be negligible compared with the data transmission itself. However, when transferring small size data, this overhead becomes significant. In addition to the initialization overhead, files with high I/O data access frequencies can also quickly overburden the regulating component in the HDFS, i.e., the single namenode that supervises and manages every access to datanodes [1]. If the number of datanodes is large, the single namenode can quickly become a bottleneck when the frequency of I/O requests is high.

In many systems, frequent file access is unavoidable. For example, log file updating, which is a common procedure in many applications.[1] Since the HDFS applies the rule of "Write-Once–Read-Many", the updating procedure first reads these files, modifies

* Corresponding author.
E-mail addresses: xhua@hawk.iit.edu (X. Hua), hwu28@hawk.iit.edu (H. Wu), zli80@hawk.iit.edu (Z. Li), ren@iit.edu (S. Ren).

http://dx.doi.org/10.1016/j.jpdc.2014.03.010
0743-7315/

[1] In the Hadoop framework, the computing task is allocated to the worknode where the data is required. Hence, this task's log file can be updated locally and hence the overhead is negligible; however, in this paper, we treat the HDFS as a stand-alone distributed file system without necessarily having the Hadoop framework support.

them and then writes them back. Such an updating procedure generates I/O requests with high frequency. Another example is the synchronization procedure in a distributed system using incremental message delivery via the file system. In this case, an incremental message file is generated by a changed component of a distributed system. This file is relatively small and it is read by all participating components to synchronize the entire system. As the HDFS allows multiple tasks reading the same file simultaneously, it is possible that a file is read frequently within a short period of time.

To overcome these issues for interaction-intensive tasks, efforts are often made from three directions: (a) improving data structure or system architecture to provide faster I/O with less overhead [3,13], (b) extending the namenode with a hierarchical structure [12,2,5] to avoid single namenode overload, and (c) designing a better storage allocation algorithm to improve data accessibility [7,8].

In this paper, we use the HDFS as a stand-alone file system and present an integrated approach to addressing the HDFS performance degradation issue for interaction-intensive tasks. In particular, we extend the HDFS architecture by adding cache support and transforming single namenode to an extended hierarchical namenode architecture. Based on the extended architecture, we develop a Particle Swarm Optimization (PSO)-based storage allocation algorithm to improve the HDFS throughput for interaction-intensive tasks.

The rest of the paper is organized as follows. Section 2 discusses the related work focusing on the cause of throughput degradation when handling interaction-intensive tasks and possible solutions developed for the problem. Section 3 presents an enhanced HDFS namenode structure. Section 4 describes the proposed PSO-based storage allocation algorithms to be deployed on the extended structure. Experimental results are presented and analyzed in Section 5. Finally, in Section 6, we conclude the paper with a summary of our findings and point out future work.

2. Related work

As the application of the HDFS increases, the pitfalls of the HDFS are also being discovered and studied. Among them is the poor performance when the HDFS handles small and frequently interacted files. As Jiang et al. [9] pointed out, the HDFS is designed for processing big data transmission rather than transferring a large number of small files, hence it is not suitable for interaction-intensive tasks.

Shvachko et al. [15] notice that the HDFS sacrifices the immediate response time of individual I/O requests to gain better throughput of the entire system. In other words, the HDFS chooses the strategy of enlarging the data volume of a single transmission to decrease the time ratio of data transmission initialization overhead in the overall operation. The transmission initialization includes permission verification, access authentication, locating existing file blocks for reading tasks, allocating storage space for incoming writing tasks, and establishing or maintaining transmission connections, etc. Another issue with the current HDFS architecture is its single namenode structure which can quickly become the bottleneck in the presence of interaction-intensive tasks and hence cause other incoming I/O requests to wait for a long period of time before being processed [1].

Efforts have been made to overcome the shortcomings of the HDFS. For instance, Liu [13] combines several small files into a large file and then uses an extra index to record their offsets. Hence, the number of files is reduced and the efficiency of processing access requests is increased. To enhance the HDFS's ability of handling small files, Chandrasekar et al. [3] also use a strategy that merges small files into a larger file and records the location of file blocks inside the client's cache. This strategy also allows the client application to circumvent the namenode to directly contact the datanodes, hence alleviates the bottleneck issue on the namenode. In addition, this strategy uses pre-fetching in its I/O process to further increase the system throughput when dealing with small files. The drawback of the strategy is that it requires client side code change.

Preventing access requests from frequently interacting with datanodes at the bottom layer of the HDFS can also significantly improve the I/O performance for interaction-intensive tasks. As illustrated in [9], one way to achieve this is to add caches to the original HDFS structure. In [17], a shared cache strategy named HDCache, which is built on the top of the HDFS, is presented to improve the performance of small file access.

Liao et al. [12] adapt the conventional hierarchical structure for the HDFS to increase the capability of the namenode component. Borthakur [2] discusses in detail the architecture of the HDFS and presents several possible ways to apply the hierarchical concept to the HDFS. Fesehaye et al. provide a solution called EDFS [5] which introduces middleware and hardware components between the client application and the HDFS to implement a hierarchical storage structure. This structure uses multiple layers of resource allocators to ease the quick saturation issue of the single namenode structure.

Designing optimized storage allocation algorithms is another effective way to improve the throughput of the HDFS. In [7], a load re-balancing algorithm is developed to always keep each file in an optimal location in a dynamic HDFS environment to gain the best throughput. An algorithm for optimizing file placement in the Data Grid Environment (DGE) is presented in [8]. This algorithm consists of a data placement and migration strategy to help the regulating components (i.e. the namenode in the HDFS) in a distributed data sharing system to place files in a large number of data servers.

As can be seen from the brief summary above, to overcome the throughput degradation issue for interaction-intensive tasks, most of the research in this area focuses on three mechanisms, i.e., optimizing metadata or adding cache, using an extended structure for the regulating component and designing new storage allocation algorithms. However, if these mechanisms are applied individually, the bottleneck issues are only shifted from one place to the other, rather than be rooted out. The solution presented in this paper intends to integrate all three approaches and solve the performance degradation issue for interaction-intensive tasks from the root. Our experiment results have shown successful evidences of the proposed approach.

3. Extended namenode with cache support

In this section, an extended namenode architecture for the HDFS with cache support is specified. The following two subsections describe the details of the structure and its functionalities.

3.1. Extended namenode structure

The original HDFS structure, which consists of a single namenode, is redesigned with a two-layer namenode structure as shown in Fig. 1. The first layer contains a Master Name Node (MNN) which is actually the namenode in the original HDFS structure. The second layer consists of a set of Secondary Name Nodes (SNNs), which are applied to every rack of datanodes. Reliable caches, such as SSD, are deployed to every SNN. It is worth pointing out that the SNN does not have to be a new machine; it can be configured from an existing data node machine, though in our implementation, we use a dedicated server for each SNN. To the MNN, each SNN is its datanode; to the datanodes, they see the SNN in its rack as their namenode. Hence, the new structure fully preserves the original

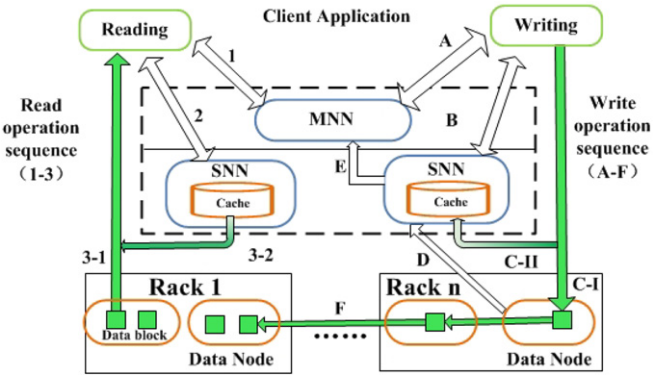


Fig. 1. Hierarchical namenode structure.

HDFS relations between MNN and SNN, and SNN and datanode structures. Therefore all the original functions and mechanisms of the HDFS are intact and the number of modifications needed for the structural extension is minimal.

The addition of the new layer to the original HDFS structure causes changes on both the reading and the writing procedures. Specifically, when a reading request arrives, the MNN first inquires the SNNs and then navigates the client application to the destination racks (Step 1 in Fig. 1). Afterwards, the SNN in the rack checks its mapping table and continues navigating the client application to the datanodes that hold the file blocks (Step 2). At the end, the client application contacts the datanodes and accesses the file blocks (Step 3-I). If a block is cached, the SNN plays the role of a datanode and lets the client application read the block from its cache (Step 3-II).

Similarly, to write data into a datenote, the MNN first allocates one or more racks for the client application to write (Step A). Then, the SNN further allocates datanodes for the client application (Step B). Afterwards, the client application begins to write file blocks into these datanodes (Step C-I). If the file block is assigned to the caches in that rack, then the SNN plays the role of a datanode and lets the client application write the file block into its cache (Step C-II). When the writing is done, the datanodes report their changes to the SNN and the SNNs report their changes to the MNN (Steps D and E), replicas for each incoming block are created in different datanodes (Step F) and a notification is sent to the client application.

3.2. Functionality of the SNN

As illustrated in Fig. 1, the SNN layer is deployed between the MNN layer and the datanode layer. Each rack of datanodes has an SNN that maintains a cache dedicated to this rack. This layer has three functionalities.

The first functionality is to keep the monitoring and messaging mechanisms between the original namenode and the datanode unchanged. To maintain the original monitoring mechanism running on the MNN, the SNN submits all the information about its rack to the MNN after receiving the MNN's heartbeat inquiry. In addition, the SNN sends its own heartbeat inquiry to the datanodes in its rack in order to monitor their status. Furthermore, in order to keep the HDFS's message mechanism intact, the SNN responds to all the messages generated by the MNN and sends its own control messages to the datanodes in its rack.

The second functionality of the SNN is to store interaction-intensive file blocks in its cache. Once a file is marked by the MNN as an interaction-intensive one and assigned to a rack, instead of relaying the writing request to the datanode layer, the SNN of this rack intercepts the request and stores the incoming blocks into its caches. As the I/O speed of a cache is much faster than the speed of hard disks used in the datanode, the throughput of interaction-intensive files is improved. Furthermore, as the number of SNNs is much smaller than the number of datanodes, file blocks in the cache can be located much faster and hence have a shorter I/O response time. Moreover, since the cost of transmission initialization on the datanode is high, the system overhead can be further reduced by using the SNN to prevent interaction-intensive tasks from frequently accessing the datanode. When a file is no longer interaction-intensive, the SNN writes all of the cached file blocks to datanodes.

The third functionality is to increase the MNN capability of handling frequent requests. In the original structure, the single namenode becomes the bottleneck when the datanode pool becomes large or when the requests come too frequently. The new layer turns the single namenode component into a hierarchical namenode structure and thus significantly increases the capability of the HDFS in managing a large resource pool and handling large incoming requests. As a result of the new layer, the size of the datanode pool managed by the MNN and individual SNN is reduced. For the MNN, the size of its datanode pool is the number of SNNs. For each SNN, the size of its datanode pool is the size of the original datanode pool divided by the number of SNNs.

4. Storage allocation algorithm

As presented in Section 3, by changing the single namenode to a hierarchical namenode structure, the HDFS's capability of handling frequent requests increases. In addition, caches introduced in this structure provide the ability of faster data access with shorter response time. Under this new structure, the throughput degradation caused by the access to interaction-intensive files can be further reduced by applying an optimized storage allocation strategy. The development of this strategy is done in three steps, i.e., for a given file, we need to determine (1) file block's interaction-intensity value, (2) its re-allocation triggering condition, and (3) file block storage location based on the file's interaction-intensity.

The notations used in the following sections are illustrated in Table 1 (sorted by the order of appearance).

4.1. Interaction-intensity

The interaction-intensity of a file, denoted as the I value, is a measurement of how frequent this file has been requested. The request can be either a read or a write request submitted from one or multiple applications. All the blocks of a file share the same I property of this file. Since the I property of a file varies from time to time, its value is represented as a function of time.

More precisely, for an existed file, its I value at time instance t is defined as

$$I(f, Q, t) = |R| \tag{1}$$

where Q is a given length of a time quantum during which the interaction-intensive value is calculated. It is a constant. The set R in Eq. (1) is defined as below:

$$R = \{(f_i, t_i) | f_i = f \wedge \max(0, t - Q) \leq t_i \leq t\} \tag{2}$$

where t is the current system time and (f_i, t_i) denotes a file access request with file name f_i and request arriving time t_i. The intuitive meaning of $I(f, Q, t)$ is that for the file f at the time instant t, the total number of requests of this file submitted to the HDFS in the last time quantum.

4.2. Trigger condition of re-allocation

The I value of a file is dynamic; it determines the best storage place of this file, so re-allocating the file every time its I value changes can improve the throughput for the whole system. However, the re-allocation is a resource-consuming procedure; using

Table 1
Notations in this section.

Notation	Definition
I	Interaction-intensity of a file
f	File name
Q	Time quantum
t	time instance
R	The access request set of a file
I_u	Upper bound of I value for migration
I_l	Lower bound of I value for migration.
S	Instant available I/O speed
F_s	File size
V_i	Node i.
D_m	Migration time cost.
B_d	Instant available network bandwidth.
C_r	Improvement caused by migration.
Y	Threshold of migration.
$\overrightarrow{V_M}$	PSO solution vector for MNN
F_M	ID set of incoming file blocks
F_{MI}	ID set of incoming interaction-intensive file blocks
Q_F	Queue for incoming file blocks sorted by arriving time
Γ	The ratio of a file's I value versus the average I value
Ψ	Estimated cost of placing blocks into cache
Ω	Estimated cost of placing blocks into datanode
$A[r]$	Number of blocks on rack r
$R[i]$	Remaining cache space on rack i
R^{avg}/W^{avg}	Average reading/writing throughput
R^{act}/W^{act}	Number of active reading/writing channels
B	Block size
P	Penalty coefficient for using cache
D_r	Number of datanode in a rack
Y_i	Number of blocks assigned to node i
E_r/E_w	Estimated reading/writing speed of a node
$\overrightarrow{V_S}$	PSO solution vector for SNN
$V[\,]$	PSO velocity array
ω	The inertia factor of PSO
C_1/C_2	The two acceleration factors of PSO

this procedure too frequently will make the performance even worse. Hence, to set up the trigger conditions of re-allocation for the existing interaction-intensive files is necessary.

Intuitively, there are two triggering conditions for each interaction-intensive file:

- If a file's I value is larger than its I_u, it indicates that the current storage node cannot provide enough available I/O speed and thus this file should be moved to a faster storage node.
- If a file's I value is smaller than its I_l, i.e. the lower bound of its I value, it indicates that the available I/O speed provided by the current storage node is higher than the requirement of this file. In this case, this file should be moved to another storage node with relatively lower I/O speed, or it will cause resource waste since there may not be enough storage spaces for the files with higher interaction-intensities.

Assume that there are n storage nodes. For each of them, the available I/O speed is denoted as S. Although both the access pattern and the data distribution can impact the performance of the hard disk, for simplicity, S is assumed to be the average case. The storage nodes are ordered by their available I/O speed, i.e., $S_i < S_j$ if $i < j$. In addition, assume that at time instance t, file f is at storage node j and the size of f is F_s, then within the constant time quantum of Q, the number of un-processed requests U is defined as follows:

$$U(f, j, Q, t) = I(f, Q, t) - \left\lfloor \frac{Q \cdot S_j}{F_s} \right\rfloor. \tag{3}$$

Suppose that the file is then re-allocated to the next storage node V_{j+1} which provides a higher available I/O speed than node V_j does. Also, assume that the I value of this file remains unchanged in the next time quantum, i.e., $I(t + Q) = I(t)$. Then, if the time cost of data migration (denoted as D_m) of this file is counted in, the number of expected un-processed requests U' for this file in the next time quantum is as follows:

$$U'(f, j + 1, Q, t + Q) = I(t) - \left\lfloor \frac{(Q - D_m) \cdot s_{j+1}}{F_s} \right\rfloor. \tag{4}$$

In Eq. (4), the value of migration time cost D_m is calculated as follows:

$$D_m = \frac{F_s}{\mathrm{Min}(B_d, V_j, S_{j+1})} = \frac{F_s}{\mathrm{Min}(B_d, S_{j+1})} \tag{5}$$

where B_d represents the network bandwidth between node j and $j + 1$.

Hence the improvement rate (denoted as C_r) based on the un-processed requests between a file on node or cache j and $j + 1$ is calculated as below:

$$C_r = \frac{U(f, j, Q, t) - U'(f, j + 1, Q, t + Q)}{I(t)}. \tag{6}$$

By comparing the C_r value with a given non-negative threshold Y, it can be determined whether a file should be re-allocated in the next time quantum. In other words, if a file's current I value is larger than its C_r value, then it should be re-allocated in the next time quantum, otherwise it should stay on the current node without invoking the allocation algorithm for a new storage destination. Therefore, the first triggering condition is $C_r = Y$. Threshold Y is an empirical value and is sensitive to the network configuration. From Eqs. (4)–(6), at the time instance t, the upper bound of a file's I value I_u can be represented as

$$I_u = \frac{\lfloor (Q - D_m) \cdot S_{j+1} \rfloor - \lfloor Q \cdot S_j \rfloor}{F_s \cdot Y}. \tag{7}$$

Clearly if $U(f, j, Q, t) < 0$, the node's or cache's I/O capacity is larger than the requests. Since the space of a high speed storage node (such as the light-workload cache) is limited, hence the file should move to a lower speed device right after the next time quantum starts. In other words, the second triggering condition is $U = 0$, Hence, the lower bound of a file's I value, i.e. I_l, can be calculated as below:

$$I_l = Q \cdot \frac{S_i}{F_s}. \tag{8}$$

At the end of each time quantum, a file's I value is updated. If the new I value is either larger than its I_u value or smaller than its I_l value, all the blocks of this file are re-assigned to the incoming block queue and then allocated to a new storage destination by the allocation algorithm.

4.3. Brief introduction to the Particle Swarm Optimization algorithm

By utilizing the I value, we present a Particle Swarm Optimization (PSO)-based storage allocation algorithm to decide storage locations for incoming file blocks.

The concept of the Particle Swarm Optimization (PSO) was first introduced in [10]. As an evolutionary algorithm, the PSO algorithm depends on the explorations of a group of independent agents (the particles) during their search for the optimum solution within a given solution domain (the search space). Each particle makes its own decision of the next movement in the search space using both its own experience and the knowledge across the whole swarm. Eventually the swarm as a whole is likely to converge to an optimum solution.

The PSO procedure starts with an initial swarm of randomly distributed particles, each with a random starting position and a random initial velocity. The algorithm then iteratively updates the position of each particle over a time period to simulate the

movement of the swarm to its searching target. The position of each particle is updated using its velocity vector as an increment. This velocity vector is generated from two factors [11]. The first factor is the best position a particle has ever reached through each iteration. The second factor is the best position the whole swarm has ever detected. The optimization procedure is terminated when the iteration time has reached a given value or all the particles have converged into an area within a given radius.

To apply the PSO to the storage allocation problem we need to (1) define the particle structure and the search space; (2) define the optimize objective functions for both MNN and SNN layers and; (3) use the PSO approach to explore the solution domain and eventually derive a near optimal solution vector. This solution vector is the allocation plan that maximizes the overall throughput of an incoming batch of file blocks.

On both the MNN and the SNN layers, a Particle Swarm Optimization (PSO)-based storage allocation algorithm is applied to search for the optimal allocation plans. The following two subsections define the solution vector as the particle structure, the value domain as the search space, and the I/O time cost function as the objective function for applying the PSO in these two layers, respectively.

4.4. Storage allocation at the MNN layer

In the MNN layer, the MNN only allocates blocks to either racks or caches. The solution vector ($\overrightarrow{V_M}$) at the MNN layer is used as an allocation plan to indicate the storage place for each incoming file block. In this section, we first define the structure of the solution vector ($\overrightarrow{V_M}$), and then introduce the cost function to evaluate the quality of a given solution vector.

Within the solution vector $\overrightarrow{V_M}$, F_M denotes a set of IDs of incoming file blocks, and the queue Q_F contains the IDs of all the files whose blocks are in F_M. As the HDFS uses the first-come–first-serve policy to schedule their tasks, the queue Q_F is hence ordered by the block's arrival time. The subset $F_{MI} \subseteq F_M$ contains all the interaction-intensive blocks in F_M.

If there are n racks at the SNN layer, to the MNN, there are $2n$ datanodes: the first n datanodes represent the datanode pool in each rack and the other n datanodes represent the caches in each rack. For example, assume that the MNN decides a block be allocated to the kth datanode, if $k \leq n$, then this block is placed into the datanode pool of rack k; otherwise, if $k > n$, that means this block is actually allocated to the cache on the $(k-n)$th rack.

Afterwards, we define the solution vector at the MNN layer as $\overrightarrow{V_M}$. Its size is $|F_M|$. $\overrightarrow{V_M}[i]$ represents the location where block $Q_F[i]$ is allocated. The value domain of entry i in vector $\overrightarrow{V_M}$ is defined as follows:

$$\overrightarrow{V_M}[i] \in \begin{cases} [1, 2n], & \text{if } Q_F[i] \in F_{MI} \\ [1, n], & \text{if } Q_F[i] \in (F_M - F_{MI}). \end{cases} \tag{9}$$

As given in Eq. (9), for blocks in F_{MI}, the value domain of their corresponding entries in $\overrightarrow{V_M}$ is $[1, 2n]$ which indicates that the blocks in F_{MI} may be stored in cache. For the rest of the blocks, as their domain is in $[1, n]$, they can only be allocated to datanodes in a rack.

As an example, assume that there are 5 racks, i.e., $n = 5$, and an incoming block $Q_F[i]$ is in set F_{MI}, then the value domain of entry $\overrightarrow{V_M}[i]$ is in the range of $[1, 10]$. If $\overrightarrow{V_M}[i] = 3$, it indicates that the incoming block $Q_F[i]$ is allocated to rack 3; while if $\overrightarrow{V_M}[i] = 9$, the block is allocated to the cache of rack 4.

To compare the interaction-intensity of incoming files with those already in the cache, the indicator Γ is introduced for each incoming file to represent the ratio of the file's current I value versus the average I value of all cached files. The definition of indicator Γ for file f is given below:

$$\Gamma_f = \frac{I_f}{\left(\sum_{i=1}^{N} I_i\right) \Big/ N} \tag{10}$$

where N is the total number of file blocks cached in the SNN layer.

Given the indicator Γ, assume that the incoming files are allocated to rack r, and there are $A[r]$ blocks on rack r; the cost function *Cost* for the solution vector $\overrightarrow{V_M}$ in the MNN layer is defined as

$$\begin{aligned} Cost = {} & k_1 \cdot \sum_{r=n}^{2n} \Psi(r-n) \cdot \frac{\sum_{\overrightarrow{V_M}[i]=r} \Gamma_i^{-1}}{A[r]} \\ & + k_2 \cdot \sum_{r=1}^{n} \Omega(r) \cdot \frac{\sum_{\overrightarrow{V_M}[i]=r} \Gamma_i}{A[r]} \end{aligned} \tag{11}$$

where the value *Cost* is determined by two factors: the estimated cost of placing blocks into caches (Ψ), and the estimated cost of placing blocks into datanodes (Ω) and k_1 and k_2 are the weight factors of these two components.

Allocating blocks with high (low) Γ value into a storage place that has low (high) I/O workload, such as an idle cache, reduces the *Cost* value; while putting blocks with low (high) Γ value to a low (high) workload storage place increases the *Cost* value. As the smaller cost of I/O tasks can bring larger throughput to the system, Eq. (11) becomes the objective function for the PSO-based algorithm.

In Eq. (11), $\Psi(r)$ is defined as follows and we leave the definition of $\Omega(r)$ to the next subsection:

$$\begin{aligned} \Psi(r) = {} & P^{\frac{B*A[r]}{R[r]}} \cdot \left(k_3 \cdot \frac{W^{act}[r] \cdot B \cdot A[r]}{W^{avg}[r]} \right. \\ & \left. + k_4 \cdot \frac{(R^{act}[r]+1) \cdot B \cdot A[r]}{R^{avg}[r]} \right) \end{aligned} \tag{12}$$

where array R records the size of the remaining cache space available in each rack, arrays R^{act} (W^{act}) and R^{avg} (W^{avg}) record the number of active reading (writing) channels connected to the cache and average reading (writing) throughput of the cache in each rack, respectively; they are obtained from the HDFS. B denotes the block size defined by the system. Constants k_3 and k_4 are the weight factors used to measure the ratio of reading and writing frequencies of the corresponding task.

P is a coefficient used to introduce penalty into the cost function for using caches. As the total space of caches is scarce compared to the storage volume provided by datanodes, the penalty $P^{\frac{B*A[r]}{R[r]}}$ increases exponentially with the ratio of required cache space versus remaining cache space. As a result, when the cache is nearly full, the PSO is more likely to allocate the interaction-intensive blocks to the datanodes with lighter workload rather than to the cache. Furthermore, if the size of blocks assigned to the cache of rack r is larger than its available space, i.e., $B * A[r] > R[r]$, the value of $\Psi(r)$ will be greatly scaled up and this allocation plan is unlikely to be chosen by the PSO.

4.5. Storage allocation at the SNN layer

For each allocation solution generated by the MNN during the PSO searching procedure, the SNNs calculate their own feedback factor Ω. Based on Ω, the MNN can then evaluate the quality of this possible solution using Eq. (11). In fact, the factor Ω itself is a quality evaluation criterion for allocation plans generated at the SNN layer. In other words, this is the objective function for the PSO applied in this layer.

For the SNN in rack r, its solution vector is defined as $\overrightarrow{V_S^r}$. Use S_r to define the set of blocks assigned from the MNN to rack r, then the cardinality of $\overrightarrow{V_S^r}$ is $|S_r|$. Similar to the solution vector $\overrightarrow{V_M}$ of the MNN, each entry in $\overrightarrow{V_S^r}$ indicates the storage destination allocated to each block assigned from the MNN to this rack. Use D_r to denote the number of datanodes in rack r, the value domain for each entry in $\overrightarrow{V_S^r}$ is $[1, D_r]$, i.e., each block in the set S_r can be placed into any datanode in this rack.

For the SNN in rack r, its cost function Ω_r is defined as

$$\Omega(r) = \sum_{i=1}^{D_r} \left(k_3 * \frac{Y_i}{E_R(i)} + k_4 * \frac{Y_i}{E_W(i)} \right) \tag{13}$$

where $E_W(i)$ and $E_R(i)$ are the predicted writing and reading speed on node i; they are given by the auto-regressive integrated moving average prediction model presented in [16]. k_3 and k_4 are the weight coefficients for reading and writing time costs. Y_i is used to record the number of blocks allocated to datanode i.

For the datanode i, let arrays R_w and R_r record the average writing and reading speed respectively after each allocation. Let arrays α and β record the CPU occupation rate and the available network bandwidth of datanode i respectively after each allocation. For the jth allocation of datanode i, $E_W(i)$ is calculated as follows:

$$E_W(i) = \frac{\beta[j]}{\beta[j-1]} \cdot A_D[j] \cdot R_w[j-1] - k_{ev} \frac{\alpha[j-1]}{\alpha[j]} \cdot (E'_W(i) - R_w[j-1]) \tag{14}$$

where $E'_W(i)$ is the estimated writing speed of the $(j-1)$th allocation in datanode i. A_D is the regression coefficient array that is computed based on past occurrences:

$$A_D[j] = \frac{\frac{E'_W(i-1)}{R_w[i-1]} + A_D[i-1]}{2}. \tag{15}$$

In Eq. (14), the performance decrease is represented as $E'_W(i) - R_W[i-1]$ and k_{ev} is the weight factor for it. $R_W(i)$ is calculated in a similar way as Eq. (14) in which E_W is replaced by E_R, R_w is replaced by R_r and E'_W is replaced by E'_R.

4.6. Apply PSO to the storage allocation problem

In the case of the storage allocation problem, a particle in the PSO is a candidate allocation plan. The structure of particles in the MNN and SNN layers are $\overrightarrow{V_M}$ and $\overrightarrow{V_S}$, respectively.

The particle moves within the search space from one point to another. The edge of the search space is defined by the value domain of the solution vector. Each point in the search space represents an allocation combination. Since the incoming file blocks have different interaction-intensities and the storage places in the HDFS have different I/O performances, different combinations can provide different I/O throughput for incoming batch files. When one combination is selected (one particle moves to the point in the search space corresponding to this combination), the estimated cost of this combination (the quality of the corresponding point) can be evaluated by the object function. In our scenario, minimum cost indicates maximum throughput. Furthermore, the terminating condition of the PSO applied in this scenario is reaching a given value of the iteration time.

Let array $p[x]$ represent the coordinates of a particle's current location, array $b[x]$ represent the coordinates of the best known position within the history of this particle, and array $g[x]$ denote the coordinates of the best known position within the history of the entire swarm. According to the PSO procedure, the movement of a particle between two iterations is determined by the velocity vector denoted by array $V[x]$ which has the same cardinality as the particle. In this array, $V[i]$ represents the ith component velocity, which is determined by

$$V[i] = \omega \cdot V[i] + C_1 \cdot r_p \cdot (b[i] - p[i]) + C_2 \cdot r_g * (g[i] - p[i]). \tag{16}$$

In Eq. (16), the coefficient ω is the inertia weight. C_1 and C_2 are two acceleration parameters which control the weight of learning from the particle's own best position and the weight of learning from the global best position, respectively. r_p and r_g are two random numbers between zero and one.

Table 2
Accelerate parameter configuration.

Phase	f value	C_1	C_2
Exploration	[0.5, 0.75)	1.5	2.5
Exploitation	[0.25, 0.5)	1	3
Jumping-out	[0.75, 1)	2.5	1.5
Convergence	[0, 0.25)	2	2

4.7. Configure the PSO's parameters with evolutionary state estimation

Since the PSO applied to the storage allocation algorithm has limited iteration time, the speed of particle convergence is critical to the quality of the final solution. In addition, the local optima phenomenon can greatly decrease the quality of the final result. Therefore, accelerating the convergence speed and avoiding the local optima have become the two most important and appealing goals in the PSO. To improve the quality of the final solution, the Evolutionary State Estimation (ESE) technique [18] is introduced in the storage allocation algorithm.

The basic concept of the ESE is to use different parameter configurations in four different PSO search phases: exploration, exploitation, convergence and jumping out. Particles in different phases should have different behaviors, which are determined by the inertia weight ω and two acceleration coefficients C_1 and C_2.

To determine the search phase, an evolutionary factor f is introduced. This factor is based on the aggregation status of the whole swarm. Take the swarm in the MNN layer as an example. Denote the mean distance of each particle i as d_i; it is defined as

$$d_i = \frac{1}{N-1} \sum_{j=1, j\neq i}^{N} \sqrt{\sum_{k=1}^{|F_M|} (p_i^k - p_j^k)} \tag{17}$$

where N and $|F_M|$ are the swarm size and the number of dimensions, respectively. Denote d_i of the globally best particle as d_g. Compare all d_i's and determine the maximum and minimum distance $d_{\max}$ and $d_{\min}$. Then, the evolutionary factor f is defined by

$$f = \frac{d_g - d_{\min}}{d_{\max} - d_{\min}}. \tag{18}$$

According to [14,18], the inertia weight ω can be determined following f:

$$\omega = \frac{1}{1 + 1.5e^{-2.6f}}. \tag{19}$$

In addition, based on the f factor, the current search phase can be determined. Then, a proper combination of accelerate parameters can be chosen. In this paper, the determination of phase and the selection of accelerate parameters are defined as in Table 2.

The PSO-based algorithm with ESE is illustrated in Algorithm 1.

Algorithm 1: Algorithm of PSO with ESE

```
Generate an initial population of particles within search spaces;
Each particle evaluate its current location by the Optimal Object Function;
while iteration time limit not met do
    update current global best location g[x];
    use Eq. (18) to determine the evolutionary State;
    use Eq. (19) and Table 2 to configure ω, C1 and C2;
    for each particle do
        for each dimension do
            Use Eq. (16) to determine the velocity component;
        move to new location by updating array p[x];
        update particle's best location array b[x];
        Evaluate its current location by the Optimal Object Function;
return g[x];
```

4.8. Allocation algorithm implementation

The procedures of the *I-I* value updating, the re-allocation bounds calculation and the storage allocation algorithm are implemented on the different components of the extended multi-layer HDFS structure.

The MNN is in charge of updating a file's *I-I* value. Since all the incoming requests are handled by the MNN first, the MNN has the knowledge of access status of all the existed files. Thus, the MNN can easily calculate the *I-I* value for files. On the other hand, recording the access count for a file only causes negligible overhead. As the workload of the MNN is greatly alleviated in the extended structure, the MNN can provide enough computing capacity to support the updating.

In addition, the MNN is used to calculate the re-allocation bounds for an interaction-intensive file at the beginning of each time quantum. For the MNN, a rack of datanodes is considered as a single storage node, so in the view of MNN there are $2n$ nodes, i.e., n caches and n racks of datanodes. On the other hand, by sending the heartbeat report, each SNN submits both its cache's status and the average I/O speed of the datanodes in its rack to the MNN, hence the MNN has the knowledge of the available I/O speed of all its $2n$ 'nodes'. As the calculation of both the bounds for an interaction-intensive file needs the I/O speed information of all the storage nodes, the bound calculation procedure can only be implemented on the MNN. This implementation brings negligible workload increase to the MNN.

For the PSO-based storage allocation algorithm, both MNN and SNN are involved. In each iteration of the PSO procedure, the MNN is in charge of calculating the *Cost* value depicted in Eq. (11) and the velocity V in Eq. (16) for each particle. To do so, both the Ψ value in Eq. (12) and the Ω value in Eq. (13) are needed. Since the MNN can obtain the running status of caches on the SNNs, Ψ can be calculated locally on the MNN. Otherwise, since the datanodes only send their heartbeat reports to a SNN, the MNN is not aware of the existence of datanode, hence the SNN is in charge of calculating the Ω value for each particle.

When a new iteration of PSO begins, the MNN broadcasts the locations of each particle to all the SNNs while calculating the Ψ value for each particle. In the mean time, SNNs calculate the Ω value for each particle with the particle's location information given by the MNN. It is easy to see that the MNN and the SNNs are working in a parallel way. Moreover, since one block can be only allocated to one storage place, the calculation of Ω on each SNN is independent, that means all the SNNs are also working in a parallel way. As a result, the PSO-based storage allocation algorithm runs very efficiently on the extended HDFS structure.

After the Ψ values and Ω values of all the particles are worked out, the MNN calculates the *Cost* value, the velocity V and the new position of each particle. Then, a new iteration is launched if the stop condition is not satisfied.

5. Experiment specifications and result analysis

In this section, we are to empirically show that modifications made to the original HDFS are able to (1) delay the time when the namenode becomes overloaded; and (2) the system throughput is increased for interaction-intensive tasks. The test-bed consists of 130 workstations with 2 Ghz CPU/4G RAM/5400 rpm HDD/1000 Mbps network connection. The router used are two Quidway S9306 routers with 1152 Mpps package forwarding rate and 6 Tbps backboard bandwidth. The test applications of the experiments are pure I/O programs combined with the Montage program that is dedicated to processing space photo image blocks in parallel [4]. We repeat each experiment for 100 times and use the average value.

The first set of experiments evaluates the performance of the extended namenode structure with respect to the time delay caused by the preparation for data transmission when handling I/O requests. Structures of one MNN with different number of SNNs are compared in this experiment.

Fig. 2 shows the comparison of the delays caused by processing reading requests. In the case of the original HDFS (with a single MNN), the time delay increases quickly with 150 or more incoming blocks, while with four or more SNNs, though the time delay also increases, the increase rate is rather small even with 1050 blocks.

Fig. 3 shows the comparison on the delays of handling writing requests. The performance differences between different structures are much more obvious than for reading requests. As shown in Fig. 3, when incoming file blocks increase to 330, the single namenode structure becomes quickly saturated, i.e. time cost for handling a writing request increases dramatically. Only the structure with one MNN and six SNNs can handle the writing requests without a large time increase when the number of incoming blocks is 1050.

The second set of experiments is based on a workstation with an i3 core and 4G RAM. This set of experiments compares the solution qualities of PSOs with and without ESE. Also, the calculation time of both the PSO and the PSO with ESE are compared. In the first step, the *Cost* value of the Linear Programming (LP) solution (the optimal solution) with realistic read/write speed is calculated. In the second step, the *Cost* values of both PSOs with and without ESE are calculated, respectively. Both of them are compared with the *Cost* value of the LP solution.

In this experiment, the weight factors in Eq. (11) are configured as: $k_1 = 0.65$ and $k_2 = 0.35$. The time quantum Q is 3 s. The weight factors of k_3 and k_4 in Eqs. (12) and (13) are configured as: $k_3 = k_4 = 1$, i.e., the reading and writing operation share the same weight in the interaction-intensive task. For both PSO algorithms, the number of particles is 100 and the iteration time is 40. Based on the configuration, the quality and speed comparison results are illustrated in Fig. 4 and Fig. 5, respectively.

Fig. 4 shows that when the number of node increases, the solution quality, i.e., the Cost value, for the pure PSO algorithm without ESE decreases at a faster rate than it is for the solution of the PSO algorithm with ESE. Even with 110 incoming blocks, the PSO with ESE can still provide solutions with quality that is no less than 80% of the optimal solution.

Fig. 5 illustrates the time cost for the pure PSO and the PSO with ESE. With the PSO algorithm, for each iteration, a new velocity vector for each particle is calculated. However, with the *PSO + ESE*

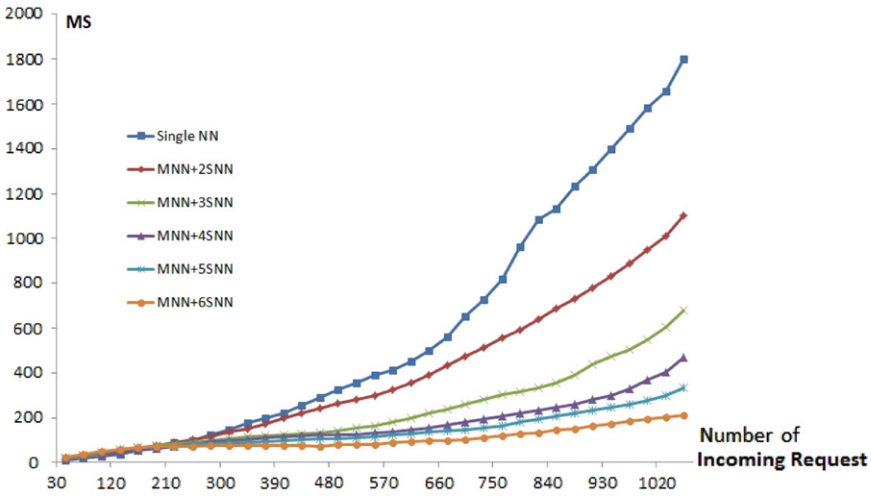


Fig. 2. Time delay of handling reading requests.

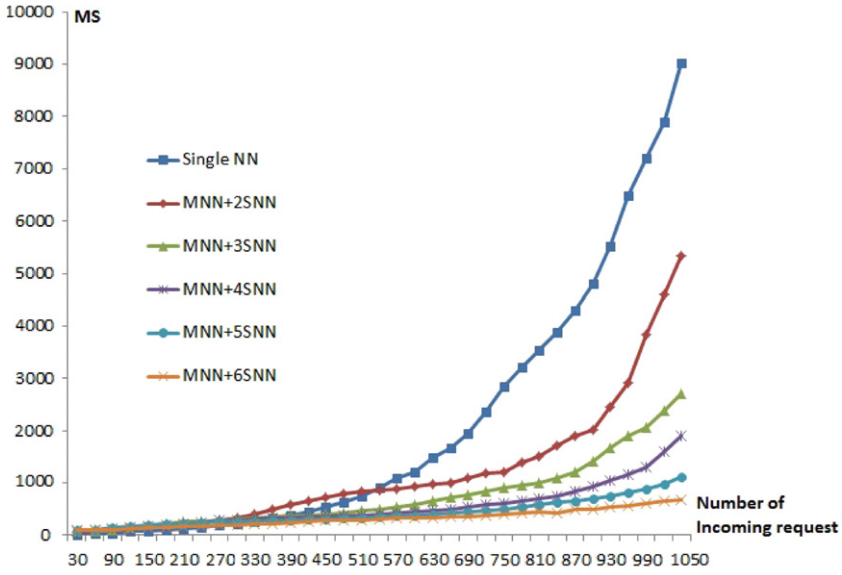


Fig. 3. Time delay of handling writing requests.

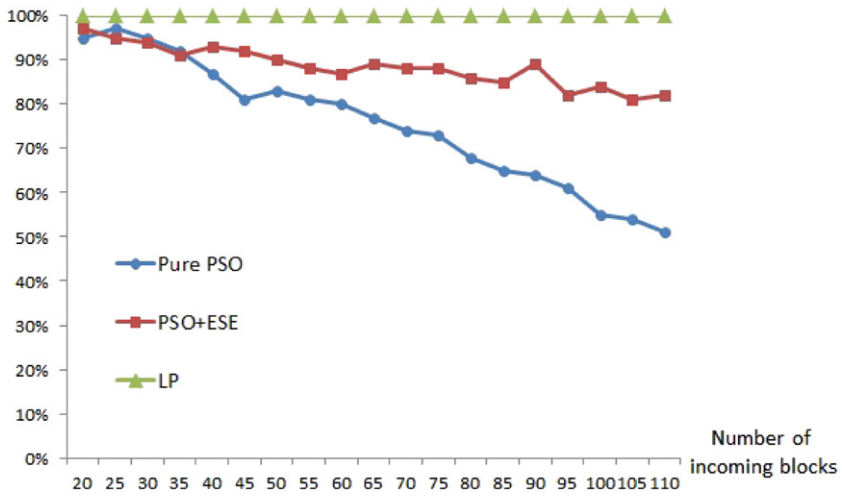


Fig. 4. Solution quality comparison of PSOs with and without ESE.

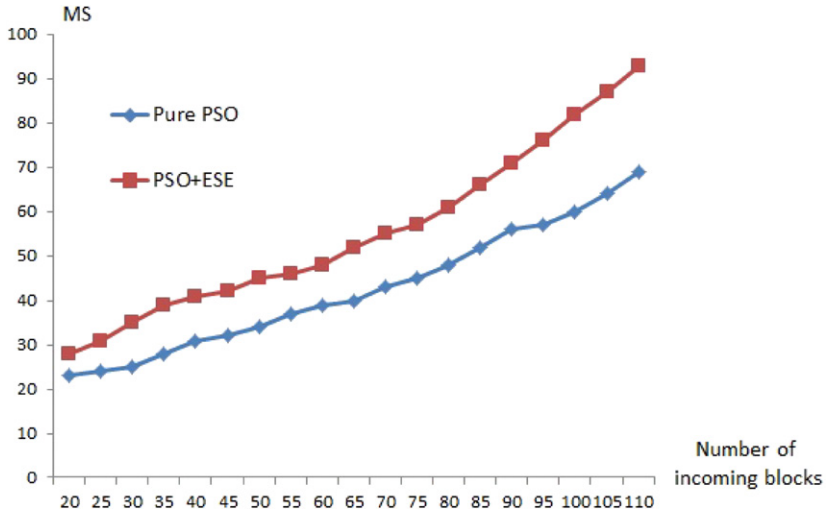


Fig. 5. Time cost comparison of PSOs with and without ESE.

Fig. 6. Comparison of reading.

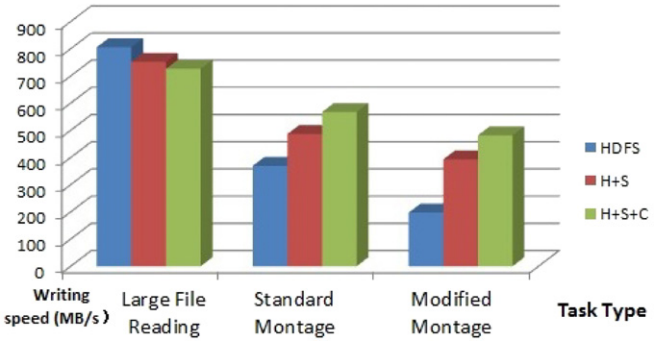


Fig. 7. Comparison of writing.

algorithm, in addition to the velocity vector calculation, for each iteration, the ESE method calculates extra three parameters (i.e., ω, C_1 and C_2) based on the searching status of particles, hence introduces some extra time overhead to the storage allocation algorithm.

However, from Figs. 4 and 5, it is easy to see that the *PSO* + *ESE* can significantly improve the quality of the solution with only slight increase in the calculation time.

The third experiment evaluates the I/O throughput of handling interaction-intensive tasks using the original HDFS structure, HDFS with SNN (H + S) and HDFS with SNN and cache support (H + S + C). It is implemented on the test-bed with 124 nodes and 6 SNNs. Storage allocation algorithms are deployed on H + S + C structures. We consider three test cases: (1) I/O tasks composed by reading and writing operations with large data; (2) standard Montage program which produces a large number of small files and has frequent I/O requests on these files; and (3) a changed Montage program which contains modifications intended to increase (40% more) its frequency of generating I/O requests. This experiment shares the same parameter configurations as used for the second experiment. The results are depicted in Fig. 6 (for reading operations) and Fig. 7 (for writing operations), respectively.

As shown in the figures, when dealing with pure I/O tasks, the original design of the HDFS has the best performance. This is because the hierarchical structure and the algorithm do introduce some overhead. However, the influence of the additional overhead on the performance is small. When the test case is changed from the large data I/O task to the standard Montage program and then the modified Montage program, the performance of the original HDFS decreases significantly. As a contrast, the performance of the H + S + B structure becomes significantly better than both the original HDFS and H + S structures.

6. Conclusion

This paper has presented an enhanced HDFS in which the performance of handling interaction-intensive tasks is significantly improved. The modifications to the HDFS are: (1) changing the single namenode structure into an extended namenode structure; (2) deploying caches on each rack to improve the I/O performance of accessing interaction-intensive files; and (3) using PSO-based algorithms to find a near optimal storage allocation plan for incoming files.

Structurally, only small changes were made to the HDFS, i.e. extending a single namenode to a hierarchical structure of namenode. However, the experimental results show that such a small modification can significantly improve (up to 300% on average) the HDFS throughput when dealing with interaction-intensive tasks and only cause slight performance degradation for handling large size data accesses.

In this work, if the file size is larger than the available cache space, this file has no chance of being cached even if its interaction-intensity is high. Our immediate future work is to design a preemptive mechanism to switch the cached files with a lower interaction-intensity out of cache to make enough cache space for the ones with a higher interaction-intensity. Moreover, as there are many factors that can affect the performance of the PSO-based algorithms, such as the interaction time, the particle swarm population and particle mutation strategies, therefore another line

of our future work is to investigate these factors to further improve the solution quality and reduce the algorithm's operation time. Furthermore, since the energy issues become more and more important today, we will study the trade-offs between system performance and energy cost in the future. In addition, we will investigate the cache manage method such as 'wait until full' strategy. Compared with our active switch out approach, it may have potential advantages over average cases. We will further investigate other newly emerged solutions. In particular, without considering that it may need client side code modification, the HDFS structure provided in [3] seems to have advantages over our two-layer structure and is worth further investigation and comparison.

Acknowledgments

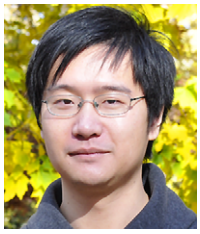
The authors wish to thank Professor Bin Su, the dean of software engineer department of Southwest Jiaotong University, China, for his help in providing the infrastructure for the experiments. The authors also wish to thank Professor Bin Su and his two graduate students, Zuowe Si and Xiao Wang, for their help in implementing and testing code, deploying system, and obtaining experimental data.

"This research made use of Montage, funded by the National Aeronautics and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology. Montage is maintained by the NASA/IPAC Infrared Science Archive".

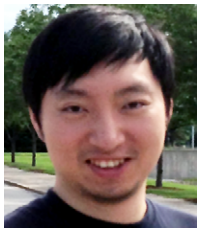
This research was supported in part by NSF (CAREER 0746643 and CNS 1018731).

References

[1] D. Borthakur, The hadoop distributed file system: architecture and design, Hadoop Project Website, 2007.
[2] D. Borthakur, Hdfs architecture guide, HADOOP APACHE PROJECT, 2008. http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf.
[3] S. Chandrasekar, R. Dakshinamurthy, P. Seshakumar, B. Prabavathy, C. Babu, A novel indexing scheme for efficient handling of small files in hadoop distributed file system, in: Computer Communication and Informatics (ICCCI), 2013 International Conference on, IEEE, 2013, pp. 1–8.
[4] E. Deelman, G. Singh, M. Livny, B. Berriman, J. Good, The cost of doing science on the cloud: the montage example, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, 2008, p. 50.
[5] D. Fesehaye, R. Malik, K. Nahrstedt, Edfs: a semi-centralized efficient distributed file system, in: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag New York, Inc, 2009, p. 28.
[6] Hdfs user guide. [Online], 2008. Available: http://hadoop.apache.org/docs/r0.19.0/hdfs_user_guide.html.
[7] H. Hsiao, H. Chung, H. Shen, Y. Chao, Load rebalancing for distributed file systems in clouds, 2013.
[8] A. Indrayanto, H.Y. Chan, Application of game theory and fictitious play in data placement, in: Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on, IEEE, 2008, pp. 79–83.
[9] L. Jiang, B. Li, M. Song, The optimization of hdfs based on small files, in: Broadband Network and Multimedia Technology, IC-BNMT, 2010 3rd IEEE International Conference on, IEEE, 2010, pp. 912–915.
[10] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Neural Networks, 1995. Proceedings., IEEE International Conference on, vol. 4, IEEE, 1995, pp. 1942–1948.
[11] J. Kennedy, W.M. Spears, Matching algorithms to problems: an experimental test of the particle swarm and some genetic algorithms on the multimodal problem generator, in: Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on, IEEE, 1998, pp. 78–83.
[12] H. Liao, J. Han, J. Fang, Multi-dimensional index on hadoop distributed file system, in: Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on, IEEE, 2010, pp. 240–249.
[13] X. Liu, J. Han, Y. Zhong, C. Han, X. He, Implementing webgis on hadoop: a case study of improving small file i/o performance on hdfs, in: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, 2009, pp. 1–8.
[14] Y. Shi, R.C. Eberhart, Parameter selection in particle swarm optimization, in: Evolutionary Programming VII, Springer, 1998, pp. 591–600.
[15] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, Ieee, 2010, pp. 1–10.
[16] S. Vazhkudai, J.M. Schopf, I. Foster, Predicting the performance of wide area data transfers, in: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, IEEE, 2002, pp. 34–43.
[17] J. Zhang, G. Wu, X. Hu, X. Wu, A distributed cache for hadoop distributed file system in real-time cloud services, in: Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on, IEEE, 2012, pp. 12–21.
[18] Z.-H. Zhan, J. Zhang, Y. Li, H.-H. Chung, Adaptive particle swarm optimization, IEEE Trans. Syst. Man Cybern. 39 (6) (2009) 1362–1381.



Xiayu Hua is a Ph.D. student in the Computer Science Department at Illinois Institute of Technology. His research interest is in distributed file system, virtualization technology, real-time scheduling and cloud computing. He earned his B.S. degree from the Northwestern Polytechnic University, China, in 2008 and his M.S. degree from the East China Normal University, China, in 2012.



Hao Wu is now a Ph.D. student in Computer Science Department at Illinois Institute of Technology. He received B.E. of Information Security from Sichuan University, Chengdu, China, 2007. He received M.S. of Computer Science from University of Bridgeport, Bridgeport, CT, 2009. His current research interests mainly focus on Resource Management in Cloud Computing.

Zheng Li received the B.S. degree in Computer Science and M.S. degree in Communication and Information System from University of Electronic Science and Technology of China, in 2005 and 2008, respectively. He is currently a Ph.D. candidate in the Department of Computer Science at the Illinois Institute of Technology. His research interests include real-time embedded and distributed systems.

Dr. Shangping Ren is an associate professor in Computer Science Department at the Illinois Institute of Technology. She earned her Ph.D. from UIUC in 1997. Before she joined IIT in 2003, she worked in software and telecommunication companies as software engineer and then lead software engineer. Her current research interests include coordination models for real-time distributed open systems, real-time, fault-tolerant and adaptive systems, Cyber-Physical System, parallel and distributed systems, cloud computing, and application-aware many-core virtualization for embedded and real-time applications.