ARTICLE IN PRESS
J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution

Jun Wang [a,b,*], Xuhong Zhang [b], Junyao Zhang [b], Jiangling Yin [b], Dezhi Han [a,**], Ruijun Wang [b], Dan Huang [b]

[a] College of Information Engineering, Shanghai Maritime University, Shanghai, 201306, China
[b] EECS, University of Central Florida, Orlando, FL 32816, United States

HIGHLIGHTS

- Decoupled two-step block distribution using an invertible math function.
- Autonomous block-node mapping maintenance on each data node.
- Optimal amounts of data movement during node addition and removal.
- About half of the RAM space and 30% of processing capacity saving in master node.

ARTICLE INFO

Article history:
Received 15 June 2015
Received in revised form
20 February 2016
Accepted 17 March 2016
Available online xxxx

Keywords:
Data intensive file system
Metadata server scalability
Data block distribution

ABSTRACT

During the last few decades, Data-intensive File Systems (DiFS), such as Google File System (GFS) and Hadoop Distributed File System (HDFS) have become the key storage architectures for big data processing. These storage systems usually divide files into fixed-sized blocks (or chunks). Each block is replicated (usually three-way) and distributed pseudo-randomly across the cluster. The master node (namenode) uses a huge table to record the locations of each block and its replicas. However, with the increasing size of the data, the block location table and its corresponding maintenance could occupy more than half of the memory space and 30% of processing capacity in master node, which severely limit the scalability and performance of master node. We argue that the physical data distribution and maintenance should be separated out from the metadata management and performed by each storage node autonomously. In this paper, we propose Deister, a novel block management scheme that is built on an invertible deterministic declustering distribution method called Intersected Shifted Declustering (ISD). Deister is amendable to current research on scaling the namespace management in master node. In Deister, the huge table for maintaining the block locations in the master node is eliminated and the maintenance of the block-node mapping is performed autonomously on each data node. Results show that as compared with the HDFS default configuration, Deister is able to achieve identical performance with a saving of about half of the RAM space and 30% of processing capacity in master node and is expected to scale to double the size of current single namenode HDFS cluster, pushing the scalability bottleneck of master node back to namespace management.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

In the era of "Big Data", vast amounts of data are analyzed by both scientists and Internet companies to make scientific discoveries and study business trends, etc. [2]. To speedup this analysis process, data-intensive computing frameworks *i.e.* MapReduce Framework have been proposed. Accompanied with this kind of new framework, co-located storage architectures referred to as Data Intensive File Systems (DiFSs) have been proposed to provide high performance for these types of jobs. DiFSs are featured as high-throughput, highly-reliable and cost-effective. GFS [19], HDFS [3], and Quantcast File System (QFS) [18] are leading examples of DiFS.

* Corresponding author at: College of Information Engineering, Shanghai Maritime University, Shanghai, 201306, China.
** Corresponding author.
E-mail addresses: jun.wang@ucf.edu (J. Wang), dezhihan88@sina.com (D. Han).

http://dx.doi.org/10.1016/j.jpdc.2016.03.005
0743-7315/© 2016 Elsevier Inc. All rights reserved.

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

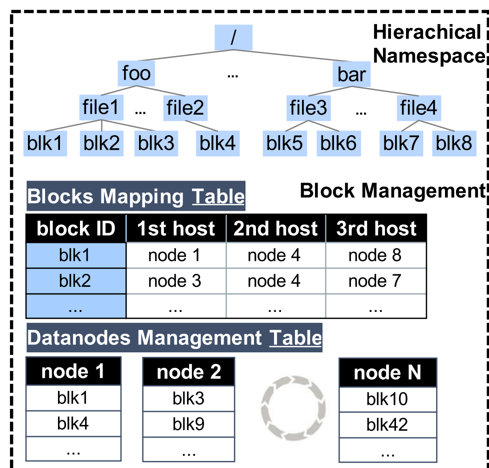


Fig. 1. Metadata management method in HDFS, a typical DiFS.

Current DiFSs adopt a master–slave architecture and are built on top of the local file system, where all of the metadata is managed by the master nodes and the physical data is managed by the local file systems on the slave (data) nodes. To maintain high availability, these storage systems usually divide files into fixed-sized blocks (or chunks). Each block is replicated (usually three-way) and distributed pseudo-randomly across the cluster with consideration of rack-awareness. In order to track these distributed blocks on hundreds or thousands of nodes, the master node must record the node locations of all blocks, as shown in Fig. 1. This is referred to as *block-node mapping*. Moreover, to guarantee the accuracy of block-node mapping, the master node must periodically receive block reports from each data node to check the blocks' locations and update the mapping information. We refer to these management schemes as *DiFS block management* (block management is used for simplicity in the rest of this paper). This is an extra layer beyond the traditional namespace management. DiFS block management offers great data distribution flexibility because each block is placed correlation-free. However, it often creates high cost on the master node in terms of memory and maintenance.

- **Memory cost**: The block-node mapping's memory consumption grows linearly with the number of blocks. It is observed that on each master node, with a file-to-block ratio of 1:1, the block-node mapping takes more than 40% of the total used memory. In addition, the proportion will reach more than 60% with a file-to-block ratio of 1:5 [20].
- **Maintenance cost**: The physical blocks are managed in the local file system on the slave nodes, blocks may be lost in the case of data corruption, node failure, etc. To guarantee mapping accuracy, the block management scheme requires a high cost on the master node's CPU and network bandwidth to synchronize the block-node mapping with the actual stored blocks on the data nodes [21]. For example, in a 10,000-node cluster with a storage capacity of 60 PB (file to block ratio is 1:1.5) [20], 30% of the master node's total processing capacity is used to process the block reports.

As data size continues to increase, these costs become non-negligible, which soon makes the master node a scalability and performance bottleneck due to the hardware limitations of the master node's memory heap size and processing capability. To address this bottleneck, the Namenode (master node) Federation [1] schema is proposed to split the single master node into many *independent* master nodes and thus allows the metadata management to scale "horizontally". However, the performance and scalability of each master node, namely the "vertical scalability", are barely improved since the resource contention between block management and the traditional namespace management still exists. We believe a more fundamental solution is to "vertically" scale each master node.

One intuitive approach is to separate the block management from the master node. For example, standalone block management has been proposed by a *Yahoo!* team, which aims to move the block management module out of the master node to some dedicated block management (BM) nodes [21]. Though the memory and maintenance cost are reduced on the master node, this approach may slow down the metadata lookup operations due to the extra hop of network lookup that is introduced.

To solve the issue of maintenance and memory cost, while maintaining the merits of current block management schema, we propose Deister. Deister consists of a deterministic two-step block distribution algorithm called Intersected Shifted Declustering (ISD) and an autonomous block-node mapping maintenance scheme.

The basic idea of Deister is to distribute the data deterministically based on an invertible mathematical function, so that each block location can be calculated, thus allowing the removal of the centralized/decentralized record-based block-node mapping. Moreover, the block-node mapping maintenance can be performed on each data node autonomously by using the inverse function of ISD. With ISD's inverse function, each data node could calculate the list of blocks it should store, which can be compared with the locally generated block report for further checking operations. In this way, the two largest overheads of block management, memory spaces and maintenance cost, can be minimized. Deister is designed to solve the scalability issue beyond thousands of nodes scale where its performance gains are more substantial. Our preliminary results show that our placement policy has the same I/O throughput with the HDFS default random layout, while the memory space and processing capacity savings on the master node are about 50% and 30% respectively, which enable the master node to support more metadata operations and about double size of current single namenode clusters.

2. Background

The booming size of "big data" imposes a variety of demands on storage organizations such as maintaining reliability and allowing for scale-out ability. Distributed file systems such as GFS and HDFS distribute blocks randomly and record each block-node mapping entry in memory. However, this approach does not scale out, since a large amount of memory and CPU cost will be spent in maintaining the block-node map. To completely design a scale-out block-node mapping/maintenance system for large-scale distributed systems, we formally define following desirable properties to achieve.

1. **Low memory space cost**. The memory space used to store or track the locations of each block should be constant and should not grow with the increasing number of blocks.
2. **Low maintenance cost**. The cost of synchronizing block-node information with the physical data should not grow with the increasing size of the cluster.
3. **Efficient addressing**. The locations of requested data blocks should be efficiently retrieved without consuming excessive computation or memory resources.
4. **Low-overhead scale-out ability**. When storage nodes are added or removed, the block-node remapping should be incrementally built on the existing block-node mapping such that the data shuffled can be minimized.
5. **High recoverability**. This contains two aspects:
   (i) Multiple failure recovery. An r-way ($r \geq 2$) replication architecture is able to provide $(r-1)$ failure recovery.
   (ii) Parallelism recovery. In the event of node failure, the lost blocks are able to be recovered (re-replicated) in parallel.

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

Table 1
Comparison among block-node mapping schemes, where N is the number of data nodes, B is the number of blocks, c is the metadata size of each block, *net* means one round of network, *CM* is the cluster map is Crush, and *DM* is the data node map in Deister.

		Examples	Memory space	Maintenance cost	Efficient addressing	Scale-out	Recoverability
Directory based	DiFS default	GFS, HDFS	$O(B*c)$	High	$O(1) \sim O(B)$	Y	High
	Standalone BM	Standalone BM	$O(B*c)$	Low	$O(1) \sim O(B) + O(net)$	Y	High
Computation based	Hashing	Ceph CRUSH	$O(CM)$	High	$O(\log N)$	Y	High
	Declustering	SD, Chain	0	Low	$O(1)$	N	Low
	Deister	Deister	$O(DM)$	Low	$O(1)$	Y	High

2.1. Current block mapping schemes

In this section, we examine the two possible solutions to reduce the BM costs for the master nodes, including directory-based mapping and computation-based mapping. And we analyze their satisfaction of five properties. A comparison summary is given in Table 1.

2.1.1. Directory-based mapping

As mentioned above, current DiFSs distribute the blocks pseudo-randomly with the consideration of network topology and rack-awareness and then store the locations of all blocks in a lookup table/tree. And the block distributed by pseudo-random placement is correlation-free. This directory-based mapping naturally satisfies Property 3, 4 and 5.

However, directory-based mapping approach incurs high cost on memory space and maintenance cost: (1) It consumes a large memory heap of the master nodes. As each block has multiple replications, the size of this block map grows much faster than the size of the file/block inodes. (2) Extra care is needed to maintain the consistency between the block map and the actual status of the data nodes and blocks. This includes block reports and replication monitor/queue. The block reports are used for the sanity check caused by software bugs or unauthorized access. The replication monitor/queue tracks the actual replica numbers of each block to prevent losing data from hardware/software failures. These services consume a large number of resources on the master node. For example, block reports, replication queue and namespace management share the same coarse-grain locks, which slows down the master server's performance.

To reduce the memory and maintenance cost in current DiFS, D. Sharp et al. proposed a standalone block management in HDFS, which aims to separate the block management out of the namenode [21]. As the namenode in the approach is solely handling the namespace operations, this approach will improve the performance of the namenode and further improve the cluster's scalability. However, HDFS may suffer a slow metadata lookup. Because compared to the original lookup process that finishes within the namenode memory, each metadata lookup involves an extra round of network messaging between namenodes and block manager nodes.

2.2. Computational-based mapping

A more reasonable solution for reducing the block management costs is to use computational-based mapping, which uses a deterministic addressing function for both block distributing and locating. Representatives include hashing and declustering.

2.2.1. Hashing

Hashing is widely adopted by many distributed file systems, such as Amazon Dynamo [8], OceanStore [14], Ceph [22], etc. It allows the elimination of the cost for directory-based mapping and the system can be balanced due to the random nature of hash functions. Based on the scope of hash mapping, we divide the hash mapping methods into two categories: fully decentralized and partially decentralized.

Fully decentralized hashing distributes both namespace information (file inodes, etc.) and blocks on all servers. For example, peer-to-peer system such as OceanStore [14] and CFS [7] maintain a distributed hash table (DHT) as the distribution method. It first hashes the directory/file names to generate a key and distribute them across a unified key space across the cluster. To achieve scale-out ability, linear hashing [15], extensible hashing [9] and consistent hashing [13] are proposed. Among them, systems such as Amazon Dynamo [8] and GlusterFS [5] adopt consistent hashing. While LH* [16,17] uses linear hashing. However, it is hard to apply fully decentralized hashing in DiFS because the architecture is fundamentally different. While DiFS uses a master–slave model, in which a few master servers manage and guarantee strong consistency, block replications, etc., the fully hashing scheme is a peer-to-peer model and eventual consistency [8].

Partially decentralized hashing uses hash mapping to only distribute blocks across datanodes, while the namespace is maintained by other methods such as tree/table mapping. For example, Ceph maintains its namespace on a few metadata servers using sub-tree partitioning; and distribute the blocks using CRUSH [11], a decentralized data distribution algorithm. CRUSH is built upon a data structure called cluster map which keeps track of the hardware infrastructure and failure domains. Both distribution and data lookup process use the CRUSH algorithm so no location information needs to be stored. As the blocks are distributed deterministically, finding each block can be achieved via calculation on any data nodes. While CRUSH provides similar advantages as Deister, namely fast deterministic mapping and little storage cost on metadata servers, its essential distribution hash function is not reversible. As CRUSH does not have a centralized point of block management, the consistency of a block is maintained by "peering", which is the process of exchanging block reports within the same placement group. This process involves a considerable amount of network messaging, while the consistency check of Deister can be achieved gracefully by using a reversible addressing function with little computation overhead.

2.2.2. Deterministic declustering

The deterministic declustering, such as Chain-declustering [12], group-rotational declustering [4], etc., is widely adopted in small-sized structures such as RAID systems. Different from hashing, the object placement is pre-determined and calculated by a certain *invertible* math function on the RAID controller. Using these approaches, all the block locations can be fast and easily calculated and no mapping information needs to be maintained. Moreover, the maintenance cost can be significantly reduced by offloading the internal workload to each datanode utilizing the inverse math distribution function.

However, the declustering mapping cannot scale-out because the location of each block is calculated based on the total number of nodes in the system Any changes, such as node addition or

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

removal, will result in large amount of data reshuffle. Also this layout is designed for homogeneous environments where all nodes are identical. Moreover, declustering placement is designed for node reconstruction, which takes far longer time than the DiFS re-replication time.

Evaluation of existing methods. We briefly examine existing approaches used for block-node mapping information management. As discussed in the first section, the major drawback of random block placement methods [19,3] and the standalone method [21] is the large memory space overhead for storing block-node information. In comparison, computational-based methods such as Hashing or Crush [8,23] could reduce the memory space overhead but maintain block-node information inefficiently. Methods such as declustering [12,4,6] could reduce the memory space overhead and support consistency checking efficiently. However, declustering techniques cannot be directly applied to large-scale systems since they are not able to efficiently scale out during system changes. We summarize how these methods satisfy the properties listed above in Table 1.

3. Deister

The Intersected Shifted Declustering distribution algorithm is extended from our previous work, Shifted Declustering (SD) [26,25], which belongs to the deterministic declustering layout described the background section. It obtains optimal parallelism in a wide range of configurations, and obtains optimal high performance and load balancing in both fault-free and degraded modes. Similar to other deterministic layouts, both data distribution and data lookup are achieved using the placement algorithm, which can significantly reduce the memory space consumption and maintenance cost. However, as this layout is initially designed for small-scale RAID like systems, directly applying shifted declustering to HDFS will cause two main problems. First, the storage system will not scale out, because the location of each block is calculated based on the total number of nodes in the system, any changes, such as node addition or removal, will result in reshuffling of the whole data blocks. Second, the recovery ability is unacceptable. If a node fails, the recovery option is either reconstructing it with a new node or re-replicating the missing blocks to current live nodes with a reshuffling of the whole data blocks. Writing all missing blocks to a new node or a system wide reshuffling both require extremely long time. The group based shifted declustering in [25] did not address the low recovery ability issue and expanded the scalability of shifted declustering layout only in a limited degree. In summary, there are two major design challenges when applying SD in the DiFS: **Low-overhead scale-out ability** and **High recoverability**.

In order to effectively address the two challenges above as well as eliminate memory and maintenance overhead from the root, we have developed a new system called Deister, as shown in Fig. 2. Deister exploits a deterministic block mapping function, which is composed of a series of mathematical functions, for block-node distribution and retrieval. This technique will allow for the millions or billions of block-node records to be removed. Also this block mapping's reverse lookup function enables autonomous block mapping maintenance on each data node. Therefore, Deister will achieve light-weight autonomous block management for large-scale data-intensive file systems, e.g., HDFS.

3.1. Inversable intersected shifted declustering with decoupled address mapping

In order to deal with scale-out challenge, we add an abstract mapping layer referred to as, **Logical Group** (LG), between the original block to node mapping. Each data node can belong to multiple logic groups and blocks are mapped to these groups

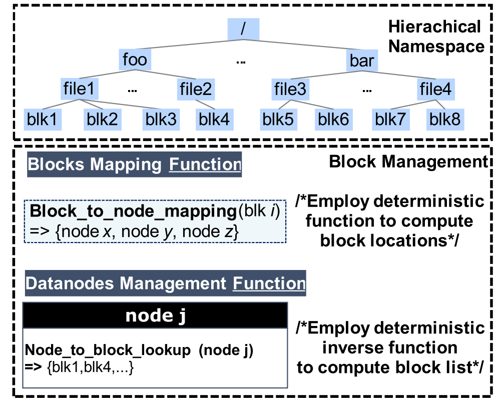


Fig. 2. Proposed metadata management solution.

before they are placed on the data nodes. The number of nodes in each group is the same, which is referred as *group size* in this paper. The mapping information between nodes and LG is recorded in a small compact table called **datanode map**, which can be stored on any of the nodes in the cluster. When given a block *id*, our methods firstly identify the block's mapped groups using a linear hash function as shown in Eq. (1), then we obtain the inner group *ID* through Eq. (2) and lastly return the nodes containing the given block using our Shifted Declustering [26], a novel placement-ideal data layout, as shown Eq. (3). Eqs. (1)–(3) constitute our deterministic calculation methods detailed in later *block-to-node mapping* steps. We show an example of Deister's mapping function in Fig. 3.

3.1.1. Block-to-node mapping

Given the block ID, the block-to-node distribution can be resolved into block-to-group mapping and inner group block distribution.

Step 1 *Block-to-group mapping*. During this step, a given block b is mapped to a group g. Assume the initial total number of groups X, is indexed from 0 to $X-1$ and is static; a module-based function is enough to map a block b to a group g as $g = b\%X$. However, in dynamic clusters, with new storage nodes deployed/removed, the number of groups, X, will change to X', so $b\%X \neq b\%X'$. This inequality implies that a large number of blocks need to be shuffled between the groups, resulting in system performance degradation. To avoid a large amount of data shuffling, we use linear hashing to map a block to a group. For a simple illustration, assume that the number of newly added groups s is smaller than X, $g = b\%(2\cdot X)$ if $s > b\%X$. Such a module strategy could allow us to achieve low remapping costs in group expansion. For instance, with respect to the first group added, only the blocks with $b\%(2\cdot X) = X$ will be remapped from group 0 to the newly added group, indexed as X, while all other blocks will not be affected. The complete equations for block to group mapping are as follows. x is the current number of groups. s in the following equations can be any number of newly added groups.

$$g = \begin{cases} b\%(X\cdot 2^{l}), & \text{if } b\%(X\cdot 2^{l}) \geq x+s-X\cdot 2^{l} \\ b\%(2^{l+1}\cdot X), & \text{if } b\%(X\cdot 2^{l}) < x+s-X\cdot 2^{l} \end{cases} \tag{1}$$

where $l = \lfloor \log_2 \frac{x+s}{X} \rfloor$.

Step 2(a) *Block ID reassignment*. Because we map the blocks b (labeled as $0, 1, 2, \ldots$) into multiple groups during *block-to-group mapping*, the blocks mapped to a specific group no longer have consecutive IDs. This can cause the data blocks to be unevenly distributed among the group nodes if the block placement method based on *id* is directly applied. To address this issue, each block is given a new block ID, a, such that all blocks

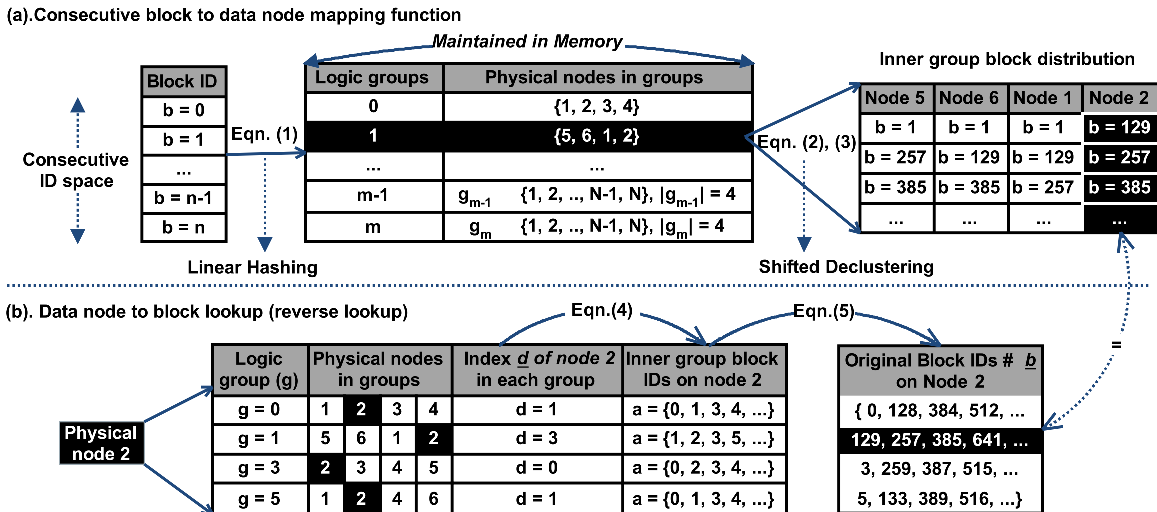


Fig. 3. Our proposed decoupled address mapping. n, N, m represent the number of blocks, Nodes, and groups, respectively. Here, use $N = 128$, *group size* $= 4$ as an example. (a) Instead of maintaining billions of block-node entries, our proposed deterministic methods only record the logic groups to provide block-node mapping. (b) Given a physical node, the reverse-lookup could be performed on each storage node for self-block checking.

are consecutively ordered before block-to-node mapping occurs. This can be achieved with the following functions, the definition of variables are the same as in Step 1.

$$a = \begin{cases} \lfloor b/(X \cdot 2^l) \rfloor, & \text{if } b\%(X \cdot 2^l) \geq x + s - X \cdot 2^l \\ \lfloor b/(2^{l+1} \cdot X) \rfloor, & \text{if } b\%(X \cdot 2^l) < x + s - X \cdot 2^l. \end{cases} \quad (2)$$

Step 2(b) *Inner group block-to-node mapping*. During this step, blocks and their replicas are distributed to nodes within each group. The most important requirement for block distribution is to ensure that the block replicas are evenly distributed within each group, i.e., any two nodes within a group should share the same amount of redundant data, which allows the lost data on the failed node to be recovered in parallel. Our proposed shifted declustering method, originally designed for the purpose of placing redundant data on disks, could satisfy this requirement [26]. As with placing data on disks, we employ the shifted declustering method to place a block and its replicas on nodes in a group in a circular fashion. To briefly illustrate this method, we assume that our cluster contains 4 nodes, and each block will have 3 replicas. The 4 nodes are indexed from 0, 1, 2, 3, and a redundant block is recorded with (a, i), where a is the block *id* and i is the replica number with the value of 0, 1, or 2 (three replicas). Given a and i, the index of the target node d is calculated with the function,

$$d = node(a, i) = (a + i)\%4. \quad (3)$$

3.1.2. Node-to-block lookup (reverse lookup)

The process of finding the *ids* of all blocks that reside on a given node is called *reverse lookup*. The reverse lookup is an essential step for autonomous block-node mapping maintenance.

Step 1 *Find the inner group block IDs*. Because the blocks are distributed within the group through a circular fashion using shifted declustering method, the blocks' *id*, a can be retrieved via an iteration calculation. For instance, in the example with group size of 4,

$$d = (a + i)\%4 \;\rightarrow\; reverse \;\rightarrow\; a = 4 \cdot j + (d - i)\%4 \quad (4)$$

where i iterates through 0, 1, 2 for each $j = 0, 1, 2 \ldots, n$, where $3n$ is the number of blocks distributed in this node calculated through our consecutive block *id* policy. The inverse function of shifted declustered for all cases is detailed in [25].

Step 2 *Map a to the original block IDs b*. Through Eq. (2), we reassign the block b, a new consecutively ordered ID a, in a specific group. The block ID a calculated in the above step, needs to be mapped to their original block b. The following function is the inverse function of Eq. (2) and the value g is the index of the logic group containing the given node,

$$b = \begin{cases} (2^l \cdot X \cdot a) + g, & \text{if } b\%(X \cdot 2^l) \geq s \\ (2^{l+1} \cdot X \cdot a) + g, & \text{otherwise.} \end{cases} \quad (5)$$

Step 3 *Join all*. Since a given node may be selected into multiple logic groups, the blocks' *id* associated with a given node should be found from all its participated groups and joined together. Thus, for each logic groups, we repeat above Steps 1 and 2, and then join all the results.

3.2. High recoverability and scale-out ability

In today's commodity clusters, cluster changes such as node failure or addition are very common. These changes would require that the file system frequently redistributes its data in order to account for the newly deployed hardware. In our Deister example system, we implement "decoupled address mapping" to efficiently reduce data shuffling in the event of cluster change.

As shown in Fig. 4, two logic groups are allowed to share the same physical node in order to minimize the number of blocks to be remapped and shuffled during node addition. Such an architecture requires an efficient method of assigning nodes to groups. One specific challenge associated with the selection of physical nodes and their assigned groups is ensuring that all nodes be assigned to approximately the same number of groups so as to maintain load balance between nodes.

In order to deal with the above challenge, we informally define the *coverage* for a node j, denoted cov_j, as a metric based on the number of existing groups containing $node_j$. *Coverage* is formally defined in Eq. (6). Intuitively, a $node_j$ appearing in more groups will store more data based on our *hashing* calculations in Eq. (1). We divide the $x + s$ groups into three sections based on their group *id*, shown as follows.

$section_1$	$[0,\ (x + s - X \cdot 2^l)]$
$section_2$	$[(x + s - X \cdot 2^l),\ X \cdot 2^l]$
$section_3$	$[X \cdot 2^l,\ x + s]$

We denote α_1 as the number of groups in $section_1$ and $section_3$ that contain node j and α_2 as the number of groups in $section_2$, that contains node j. Based on the hashing mapping in

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

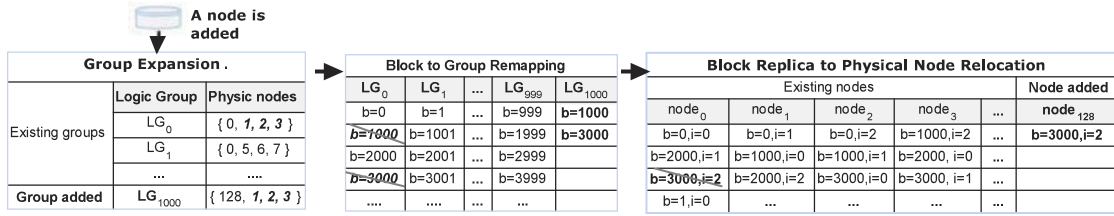


Fig. 4. Our proposed smart node to group mapping algorithm ensures the new group LG_{1000} shares 3 physical nodes with one existing group to minimize the number of remapped blocks. Blocks 1000 and 3000 along with their 3 replicas are re-mapped to new group LG_{1000}, but only one replica ($b = 3000$, $i = 2$) is physically moved.

Eq. (1), we define the coverage factor for node j as,

$$cov_j = \frac{\frac{\alpha_1}{2} + \alpha_2}{x + s} = \frac{\alpha_1 + 2\alpha_2}{2(x + s)}. \tag{6}$$

Algorithm 1 Smart Node-group Mapping: Node Failure

$N = \{node_0, node_1, \ldots, node_{n-1}\}$ // n cluster nodes
$COV_N = \{cov_0, cov_1, \ldots, cov_{n-1}\}$ // n nodes' coverage
$G = \{g_0, g_1 \ldots, g_{x+s-1}\}$, $g_k \subset N$ // all groups
$node_i$, //failed node
$R = \emptyset$
Steps:
Find: $G_i = \{g_k \in G \mid node_i \in g_k\}$
for $g_j \in G_i$ **do**
  Let: $L = \{N - g_j\}$
  Let: $COV_L = \{cov_k \mid node_k \in L\}$
  Find: $node_r \in L \mid node_r = \min COV_L$, $node_r \notin R$
  Append $node_r$ to R.
  Replace $node_i$ with $node_r$
  Update COV_N and G
end for

3.2.1. Node failure

In the event of node failure, the lost data on the failed node needs to be recovered. In our Deister system, each group containing the failed node will find a distinct live node to replace the failed one in order to achieve parallel recovery, as shown in Algorithm 1. For each affected group g_j, the set of live nodes that are not in g_j will be selected as candidate nodes L. Then the $node_r$ with the smallest coverage in L will be used to replace the failed node for g_j. Finally, the coverage of nodes COV_N and G will be updated accordingly, before selecting the replacement node for the next affected group. Fig. 6 shows an example of choosing recovery targets when Node 2 fails. As Node 2 is initially covered by Groups 0, 1, 2, 3, each group independently chooses its recovery target—Groups 0, 1, 2, 3 choose to reconstruct their missing blocks to Nodes 6, 3, 1, 5, respectively. The recovery parallelism of our proposed approach is determined by the number of groups that cover a failed node. The data movement caused by a node failure is optimal in Deister, which is w_{failed}/W of total data (where W is the total weight of all nodes). w_{failed} can be calculated based on failure node's coverage, $w_{failed} = cov_{failed} / \sum_{i=1}^{N} cov_i$.

3.2.2. Node addition

When one or multiple new cluster nodes are deployed on a storage system, some of the stored data will need to be transferred from existing nodes to the newly added ones. The goal of our node addition algorithm is to guarantee that the data is only shuffled between existing nodes and the newly added nodes. Algorithm 2 presents the process of group expansion when new nodes are added in our Deister system. If the *coverage* of the newly added node is known, the number of newly added groups, g, can be determined according to Eq. (6). According to Eq. (1), we know that half of the blocks belonging to a specific group will be remapped to the newly added group. Thus, through maximizing the number of

Algorithm 2 Smart Node-group Mapping: Nodes Addition

$COV_N = \{cov_0, cov_1, \ldots, cov_{n-1}\}$ // n nodes' coverage
$G = \{g_0, g_1 \ldots, g_{X \cdot 2^l}\}$ // all groups
Newly added s nodes
Steps:
Mirror current groups, $G = \{g_0, g_1 \ldots, g_{X \cdot 2^{l+1}}\}$
Calculate $cov_{ave} = \frac{X \cdot 2^{l+1} \cdot group\ size}{N+s}$
Calculate the number of candidate groups g with Eqn(6)
for each $node_i$ in x nodes **do**
  for $k = 1$; $k <= g$; $k++$ **do**
    Randomly selected $group_j$, $X \cdot 2^l < j < X \cdot 2^{l+1}$
    Select $node_{max}$ with the largest coverage in $group_j$
    while $group_j$ contains $node_i$ or $cov_{max} < cov_{ave}$ **do**
      Randomly selected $group_j$, $X \cdot 2^l < j < X \cdot 2^{l+1}$
      Select $node_{max}$ with the largest coverage in $group_j$
    end while
    Replace $node_{max}$ with $node_i$.
    Update cov_{max} and cov_i
  end for
end for

nodes shared between the newly added group and an existing one, the number of blocks to be shuffled will be minimized as shown in Fig. 4. So in our system, whenever $groups = X \cdot 2^l$ and new groups are being added, we double the current group size by mirroring all current groups. By default, the coverage of the added node is set to an average coverage as,

$$cov_{ave} = \frac{X \cdot 2^{l+1} \cdot group\ size}{N + i} \tag{7}$$

where N is current number of nodes; i is newly added number of nodes. For each newly added node, Deister will replace one node with this new node in each selected g new groups. Moreover, to maintain load balance, the node to be replaced shall have the largest coverage value within the new group. If the largest coverage value is still smaller than cov_{ave}, another new group will be re-selected into the g new groups. Such cluster expansion will also result in an optimal fraction, w_{new}/W, of total data to be shuffled.

Fig. 4 shows an example of adding a new node, the block remapping only occurs in group LG_1, in which blocks 1000, and 3000 with 3 replicas are remapped to the new group LG_{1000}, but only one replica out of 6 is physically moved.

3.2.3. I/O behavior during node failure or addition

In Deister, the read or write I/O will not be stalled during node failure or addition. Deister maintains two versions of datanode map during nodes failure or addition. One is the currently used version before node changes, the other one is the temporary under-construction version after node changes. For node addition, the under-construction version will be sent to data nodes to direct the data shuffling process. As described in Section 3.2.2, some blocks will be moved to the newly added node from the old nodes. In this process, each old node does not delete its local copies of the moved blocks. Thus, all read operations can continue to be served by the old datanode map without failure, while all the write operations after the shuffling process begins will be served by the

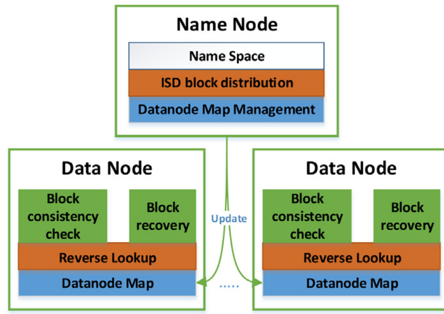


Fig. 5. Overall architecture of autonomous block-node mapping maintenance. Data nodes do not exchange any metadata with each other.

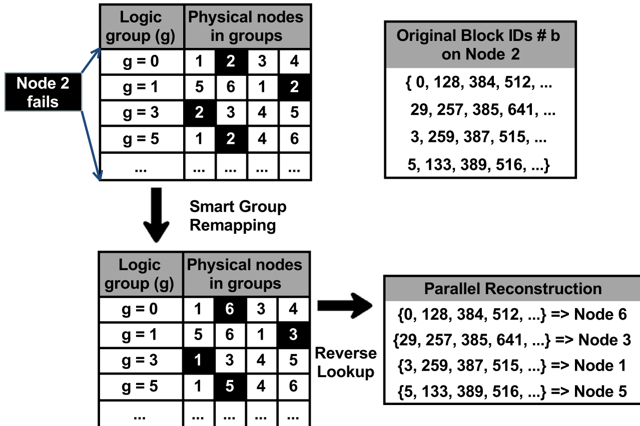


Fig. 6. Parallel reconstruction for Node 2 which is covered by Groups 0, 1, 3, and 5. Each group replaced Node 2 with Nodes 6, 3, 1 and 5 respectively.

under-construction datanode map. After the shuffling process is complete, the old datanode map will be completely replaced by the under-construction version and each old data node will then delete its local copies of the moved blocks. Similar for node failure, all the new write I/O after the node failure will be served by the under-construction datanode map while the read I/O will continue to be served by the datanode map before the node failure until the recovery process is complete.

3.3. Autonomous block-node mapping maintenance

In the previous sections, we use deterministic methods to replace the block-to-node mapping in order to eliminate the memory overhead on the master node. In this section, we propose our autonomous block-node mapping maintenance to eliminate the maintenance overhead on the master node. Two key maintenance tasks are needed to keep the block-node mapping up-to-date.

- Consistency checking: the recorded block-node mapping on the master node has to be synchronized with the blocks actually stored on each data node.
- Handling recovery: in case of block loss, the DiFS has to track the under-replicated blocks and recover them accordingly.

In current DiFSs, these two tasks can be completed only on the master node. The master node periodically receives block reports from all data nodes, updates its block-node mapping records, and recovers under-replicated blocks accordingly. However, these two maintenance tasks require non-negligible CPU and network resources on the master node. Therefore, in Deister, we propose that each data node performs these two tasks itself by using ISD's reverse lookup function. These two tasks are performed independently without any metadata exchange. Fig. 5 gives a high level architecture of this autonomous maintenance schema.

3.3.1. Autonomous block-node mapping consistency check

To perform consistency checking autonomously, each data node first has to know what blocks it should store. Instead of consulting the master node for this information, each data node could calculate this information locally with ISD's reverse lookup function. The only requirement for this calculation is a copy of datanode map, which is relatively small. Then, each data node scans its local file system and gets a list of blocks it currently stores. Finally, the comparison is conducted between the stored block list and the calculated block list. The physical block locations will be adjusted based on the calculated location information. The locations of the physical blocks should unconditionally follow the distribution function.

3.3.2. Autonomous block recovery

Lost blocks found during the block-node mapping consistency checking can be easily recovered by the data node itself, since it has a copy of the datanode map, and it can use ISD to calculate the locations of lost blocks' live replicas, and then copy the live replicas from the calculated locations.

In the case of node failure, the master node will first update the datanode map using algorithms in Section 3.2. Upon receiving the datanode map updates, each data node could autonomously start the recovery process for the node failure. Each data node will first check whether its LG membership has changed. If the data node's LG membership changed, it will use ISD's reverse lookup function to calculate a new list of blocks it should store and recover the physically missing blocks in this list. Fig. 6 shows an example of the process of recovering the lost blocks on a failed node.

3.3.3. Network efficiency

For block consistency check, no network metadata messages are required. All the required block metadata can be calculated locally. As for node failure, only a datanode map update need to be sent to each data node. However, in HDFS, recovering each block on the failed node will require one network metadata message from the namenode. In practice, the number of missing blocks is several magnitudes higher than the number of data nodes.

4. Evaluations

We have implemented a Deister prototype and only implemented a block management module of DiFS, rather than a whole new DiFS. Since block management is tightly coupled with other modules in current DiFS such as name space management, it is hard to directly measure the performance metrics, such as memory and CPU usage, of current block management. On the other hand, Deister's performance gains are more substantial beyond thousands of nodes scale, but we have limited cluster resources, it is not possible for us to conduct experiments at thousands of nodes scale. Therefore, in this paper, we extract the block management module of HDFS source code. Then, we remove its dependencies on other HDFS modules. The core components such as *BlocksMap, replicationThread, neededReplications, pendingReplications, processReport, DatanodeManager, HeartbeatManager,* etc., are all kept. Namenode and data nodes in both HDFS and Deister are implemented as two separate processes hosted on two machines. Each data node is implemented as a thread in the data nodes process, so it is easy to simulate thousands of data nodes. Finally, we build an independent block management program for each of HDFS and Deister. Deister is designed to satisfy all of the desired five properties of DiFS block management as discussed in Section 2. We evaluate each of the five properties of Deister relative to HDFS.

Experimental setup. All our experiments are conducted on Marmot and CASS clusters. *Marmot* is a cluster of PRObE on-site

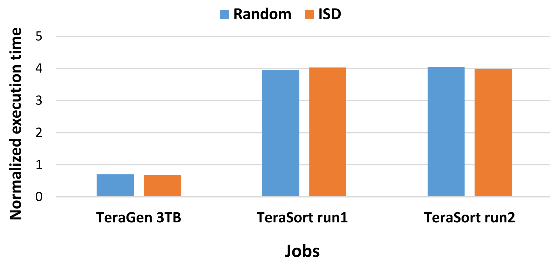


Fig. 7. Performance comparison between random and ISD replica Placement. TeraGen is all write while TeraSort has read and write.

project [10,24] and housed at CMU in Pittsburgh. The system has 128 nodes/256 cores and each node in the cluster has dual 1.6 GHz AMD Opteron processors, 16 GB of memory, Gigabit Ethernet, and a 2 TB Western Digital SATA disk drive. *CASS* consists of 46 nodes on two racks, one rack including 15 compute nodes and one head node and the other rack containing 30 compute nodes. Each node is equipped with two Dual-Core 2.33 GHz Xeon processors, 20 GB of memory, Gigabit Ethernet and a 500 GB SATA hard drive.

4.1. Block distribution

4.1.1. ISD placement vs. random placement

In this section, we test the effect of Deister's ISD placement on Hadoop jobs and I/O performance relative to HDFS's random placement. ISD placement is implemented into the original HDFS. We deploy a Hadoop cluster on Marmot with one node serving as the master while 100 nodes serve as slaves. The default block size is set to 64 MB. We choose TeraGen and TeraSort benchmark suit, because they are typical write and read heavy MapReduce jobs. TeraGen benchmarks are run with the same configuration on the same cluster as follows: a total amount of 1 TB of data is written; and each file has 3 replicas. Therefore, there are 3 TB of data generated by the TeraGen job. Then, the TeraSort is used to sort the generated data with the same number of mappers assigned.

We can see from Fig. 7 that the execution time on both TeraGen and TeraSort jobs is almost the same. We conclude ISD layout has negligible effect on the performance of Hadoop jobs. We also run the TestDFSIO benchmark to further test Deister's I/O performance. 100 mappers are configured with each reading and writing 1 GB data. The results are shown in Fig. 8. We can see that the I/O performance of Deister is almost the same with HDFS.

4.1.2. Memory space

We measure the memory usage of block management for storing millions of blocks on both HDFS and Deister on CASS cluster as shown in Figs. 9 and 10. In our experiment, each block has 3 replicas and each data node with 1 TB capacity is capable of storing 18,000 blocks. Deister's memory usage is measured with different number of groups and group sizes. Higher number of groups and larger group size will require more memory space. In Fig. 9, we first show the memory space cost with growing number

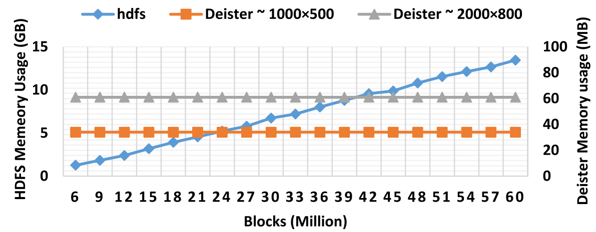


Fig. 9. Block management memory usage as the number of blocks increases. The number of data nodes is fixed to 10,000. Deister is configured with different *number of group* × *group size*.

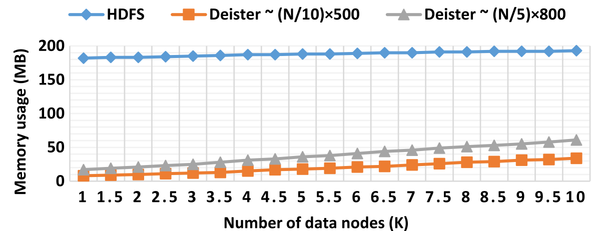


Fig. 10. Block management memory usage as the number of data nodes increases. The number of blocks is fixed to 1,000,000. N is the total number of data nodes, Deister is configured with different *number of group* × *group size*.

of blocks in a 10,000 nodes cluster. The memory cost of HDFS block management grows linearly with the increasing number of blocks, but that of Deister remains constant with only 34 MB with respect to 13.4 GB in HDFS. The reason behind this is that Deister only stores data nodes information, block locations are calculated in real time, requiring no memory space. Then, we examine the memory usage trends as the number of data nodes increases, while the number of blocks is fixed to 1,000,000. From Fig. 10, we can see the memory usage of HDFS block management grows little, since adding thousands of data nodes' information objects require negligible memory space. Deister's memory usage grows linearly, but with only tens of Megabytes, it can scale to 10,000 nodes. Our results are also confirmed in [20,21]. It reports that the total amount of metadata for storing 500 million blocks in HDFS will occupy about 130 GB RAM on a namenode, among which block management accounts for about 63% of total RAM used. And even a 3.5 K-nodes cluster becomes unresponsive under heavy loads, scaling to 10 K nodes is unachievable. We conclude that, with about 50% less memory usage, Deister is expected to scale to double the size of the current cluster.

4.1.3. Block location lookup

Since the location of a block in Deister is calculated in real time, we measure the lookup latency of Deister comparing to HDFS on CASS cluster as shown in Fig. 11. HDFS uses a hashmap to index each block object and the block locations are recorded in the block object. So the hashmap's performance determines HDFS's block location lookup time. The block location time in the figure is averaged by looking up 10 000 random blocks. From the result, we

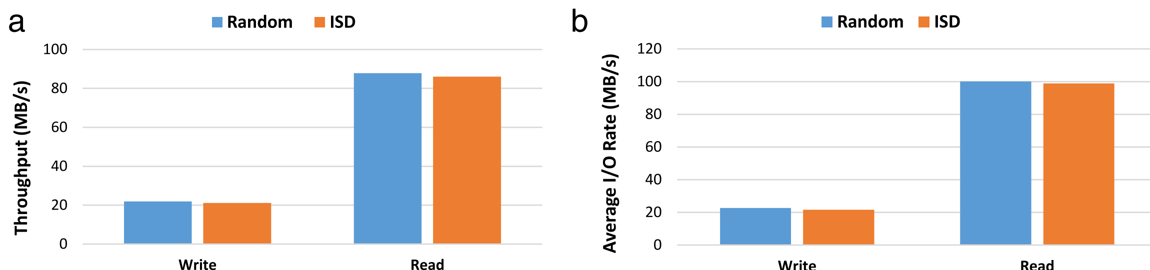


Fig. 8. TestDFSIO throughput and average I/O rate.

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

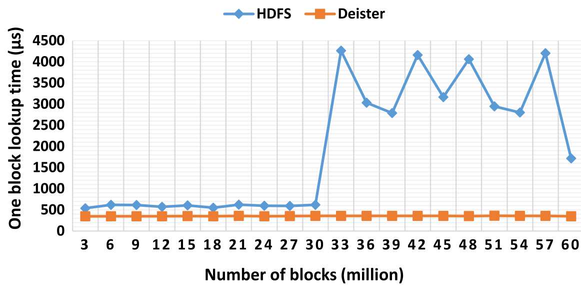


Fig. 11. Lookup latency.

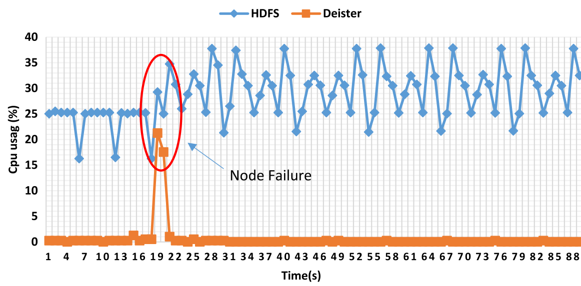


Fig. 12. CPU usage of block maintenance with a cluster size of 10,000.

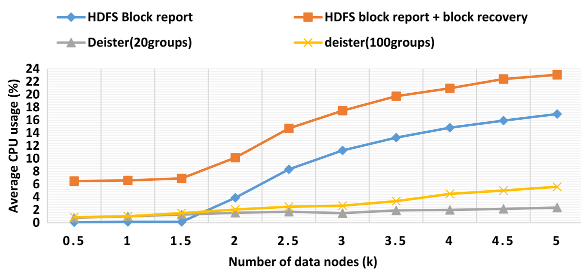


Fig. 13. CPU usage of block maintenance as system expands.

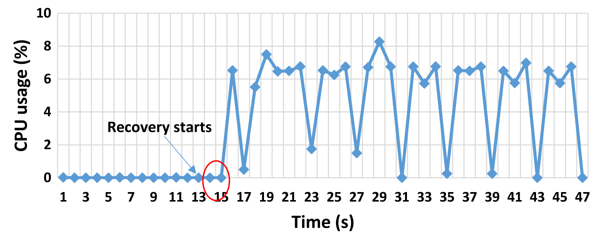


Fig. 14. CPU usage of block recovery in Deister data nodes.

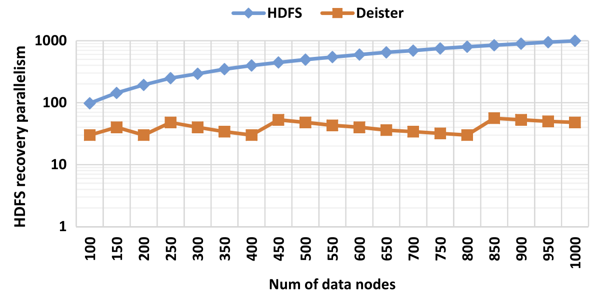


Fig. 15. Recovery parallelism. Deister has an initial cluster size of 100 and an initial recovery parallelism of 30.

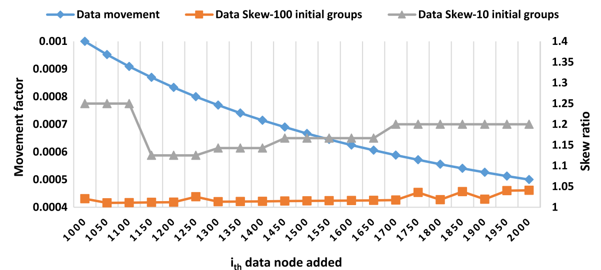


Fig. 16. Data movement and data distribution skew ratio when expanding a cluster from 1000 nodes to 2000 nodes.

can see Deister's lookup latency remains constant with only about 350 µs, while HDFS's hashmap lookup time is constantly longer and experiences a jump when the number of blocks increases to 30 million. This is because Deister's math lookup function is irrelevant to the number of blocks, but for HDFS, as hashmap's complexity is $O(1) \sim O(n)$, more blocks in the hashmap will produce more hash collisions, which will require longer lookup time.

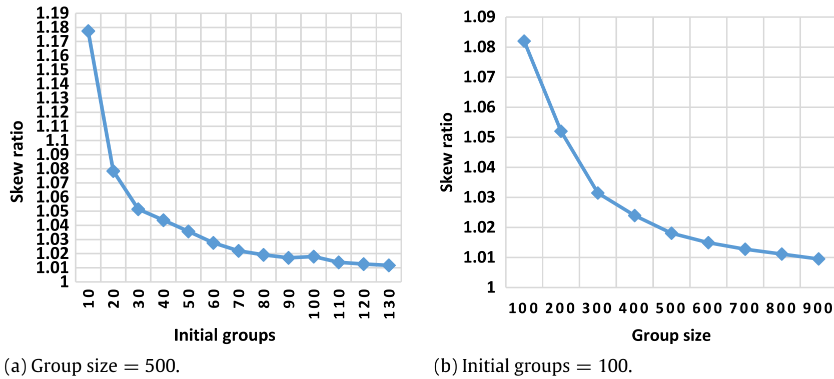
4.2. Block maintenance

Block maintenance, including block consistency check and block recovery is another overhead in current DiFS block management. In Fig. 12, we show the CPU usage of block maintenance in both HDFS and Deister. Both HDFS and Deister are initialized with 10,000 data nodes and 60 million blocks, each with 3 replicas. Both namenodes are deployed on CASS cluster. The block report interval in HDFS is set to 1 h and the heartbeat interval in both HDFS and Deister is set to 3 s. We assume the block reports' arrival time is evenly distributed in an hour, then there will be about 3 reports processed in one second. Each report contains the information of 18,000 blocks. And we place a node failure at the 19_{th} second. As shown in Fig. 12, the block reports processing in HDFS occupies about 25% of total namenode processing capacity. After HDFS namenode starts blocks recovery, the CPU usage increases to about 35%. While in Deister, there is only a short CPU usage jump when node failure is detected. After the namenode finishes updating the datanode map, CPU usage goes back to extremely low usage state in which namenode only processes heartbeats from data nodes.

In Fig. 13, we further investigate how the CPU usage of block maintenance varies with the increasing number of data nodes. For HDFS, we measure the average CPU usage of both block report processing and block report processing plus block recovery. And for Deister, we capture its average CPU usage of updating datanode map triggered by node failure. With increasing number of data nodes, the CPU usages of block reports processing and block recovery in HDFS increase by about 18% at 5000 data nodes scale. And they follow the same increasing trend. This implies that the CPU usage increase in HDFS block maintenance is mainly caused by block reports processing, since the namenode has more block reports to process with more data nodes. While in Deister, the CPU usage has an increase of only about 5% when the cluster scales to 5000 data nodes. On the other hand, the CPU usage of updating datanode maps increases with more groups in Deister. The reason is well explained in Algorithm 1: each group containing the failed node will have to find a live data node with minimum coverage to replace the failed node, more groups a failed data node belongs to, more computation is needed.

In Deister, the block maintenance is performed autonomously on each data node. Fig. 14 shows the CPU usage of block maintenance on a data node. In our experiment, we configure each data node to reverse lookup all its stored blocks every 3 s for block consistency check. We find that reversing 18 000 blocks incurs only about 0.025% CPU usage. In case of node failure, each data node could independently compute the list of blocks it needs to recover using the reverse lookup function. In this experiment, the failed node is covered by 10 groups, so each node is responsible for recovering 1800 blocks. After getting the list of blocks it needs to

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005



(a) Group size = 500.

(b) Initial groups = 100.

Fig. 17. Average data distribution skew ratio when expanding the number of data nodes from 1000 to 2000.

recover, the block recovery process is the same as in namenode of HDFS, which incurs only an average of about 6% CPU usage.

4.3. Reorganization and data movement

In this section, we evaluate Deister's ability to handle cluster node addition and removal.

4.3.1. Nodes failure

The data movement incurred by node failure in Deister is optimal and the same as in HDFS. For node failure, we will be focusing on evaluating the recovery parallelism. Fig. 15 shows the recovery parallelism of both HDFS and Deister with increasing number of data nodes. It is not a surprise that HDFS's recovery parallelism is always near cluster size, since each missing block's recovery target is randomly chosen across the cluster. But in Deister, the recovery parallelism is determined by the number of groups the failure node covers. In Fig. 15, the initial cluster size for Deister is 100. We measure Deister's recovery parallelism under two settings. One has initial 60 groups with group size of 50 and the other has 150 groups with group size of 20. Group size is fixed throughout expansion, while the number of groups increases as system expands. In the result, we can see the recovery parallelism is the same as the initial parallelism at points: 100, 200, 400, 800, 1600, but fluctuates at their intervals. The reason can be well explained by Fig. 4 and Algorithm 2: newly added groups will overlap all of its data nodes except the new nodes with existing groups. This will make the old nodes cover more groups than new nodes, resulting in higher recovery parallelism in old nodes. Thus the average recovery parallelism will increase, but as old nodes in the new groups are gradually replaced with new nodes, the average recovery parallelism will drop to its initial size.

4.3.2. Nodes addition

In this section, we study the data movement and storage balance when the system expands. Fig. 16 shows the data movement factor decreases exponentially as the number of data nodes expands. As Deister is designed to maintain system balance dynamically, each time a new node is added, only $\frac{1}{N}$ of total data is moved to this new node, where N is the current number of nodes. Then, we examine Deister's real time data distribution balance as the system expands. The skew ratio in Fig. 16 is calculated as cov_{max}/cov_{min}. In the results, we can see Deister's data distribution skew ratio varies little as system expands, but with more initial groups, Deister has better overall balance. Therefore, we conduct two more experiments to further explore how distribution balance is related to the initial number of groups and group size. The results are shown in Fig. 17. The results indicate that higher number of groups or group size will both produce better storage balance. For data nodes with 1 TB capacity, 100 groups and 500 group size can generate at most about 10 GB storage imbalance.

5. Conclusion

In this paper, we propose a light-weight autonomous block management scheme for current data-intensive file system such as HDFS. Deister features decoupled two-step block distribution and autonomous block-node mapping maintenance on each data node using invertible ISD. It could achieve optimal amounts of data movement during node addition and removal. Deister is the first system that is able to satisfy all the five desired properties: low memory space cost, low maintenance cost, efficient addressing, low-overhead scale-out ability and high recoverability. Results show that, as compared with the HDFS default configuration, Deister is able to achieve identical performance with a saving of about half of the RAM space and 30% of processing capacity in master node and is expected to scale to double the size of the current clusters.

Acknowledgments

This material is supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE). http://www.nmc-probe.org/

This work is supported in part by the US National Science Foundation Grants CCF-1337244, CCF-1527249, National Science Foundation Early Career Award 0953946, and the National Science Foundation of China (Grant no. 61373028 and 61070154).

References

[1] An Introduction to HDFS Federation, http://hortonworks.com/blog/an-introduction-to-hdfs-federation/.
[2] Big data analytics, http://www.sas.com/offices/europe/uk/downloads/bigdata/eskills/eskills.pdf.
[3] D. Borthakur, The Hadoop Distributed File System: Architecture and Design, The Apache Software Foundation, 2007.
[4] M.-S. Chen, H.-I. Hsiao, C.-S. Li, P.S. Yu, Using rotational mirrored declustering for replica placement in a disk-array-based video server, Multimedia Syst. 5 (6) (1997) 371–379. http://dx.doi.org/10.1007/s005300050068.
[5] Cloud storage for the modern data center—an introduction to gluster architecture, http://moo.nac.uci.edu/hjm/fs/An_Introduction_To_Gluster_ArchitectureV7_110708.pdf.
[6] G. Copeland, T. Keller, A comparison of high-availability media recovery techniques, in: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD '89, ACM, New York, NY, USA, 1989, pp. 98–109. http://dx.doi.org/10.1145/67544.66936. URL: http://doi.acm.org/10.1145/67544.66936.
[7] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with CFS, in: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP '01, ACM, New York, NY, USA, 2001, pp. 202–215. http://dx.doi.org/10.1145/502034.502054. URL: http://doi.acm.org/10.1145/502034.502054.
[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store, in: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, ACM, New York, NY, USA, 2007, pp. 205–220. http://doi.acm.org/10.1145/1294261.1294281.

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005

[9] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, Extendible hashing—a fast access method for dynamic files, ACM Trans. Database Syst. 4 (3) (1979) 315–344. http://dx.doi.org/10.1145/320083.320092. URL: http://doi.acm.org/10.1145/320083.320092.

[10] G. Gibson, G. Grider, A. Jacobson, W. Lloyd, Probe: A thousand-node experimental cluster for computer systems research, 38, 2013. URL: https://www.usenix.org/publications/login/june-2013-volume-38-number-3/probe-thousand-node-experimental-cluster-computer.

[11] R. Honicky, E. Miller, Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution, in: Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004. (26-30 April 2004) p. 96.

[12] H.-I. Hsiao, D.J. DeWitt, Chained declustering: A new availability strategy for multiprocessor database machines, in: Proceedings of the Sixth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1990, pp. 456–465.

[13] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97, ACM, New York, NY, USA, 1997, pp. 654–663. http://dx.doi.org/10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, Oceanstore: an architecture for global-scale persistent storage, SIGPLAN Not. 35 (11) (2000) 190–201. http://doi.acm.org/10.1145/356989.357007.

[15] W. Litwin, Linear hashing: A new tool for file and table addressing, in: Proceedings of the Sixth International Conference on Very Large Data Bases—Volume 6, VLDB '80, VLDB Endowment, 1980, pp. 212–223. URL: http://dl.acm.org/citation.cfm?id=1286887.1286911.

[16] W. Litwin, M.-A. Neimat, D.A. Schneider, Lh*—a scalable, distributed data structure, ACM Trans. Database Syst. 21 (4) (1996) 480–525. http://doi.acm.org/10.1145/236711.236713.

[17] W. Litwin, T. Risch, LH*g: a high-availability scalable distributed data structure by record grouping, IEEE Trans. Knowl. Data Eng. 14 (4) (2002) 923–927. http://dx.doi.org/10.1109/TKDE.2002.1019223.

[18] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, J. Kelly, The quantcast file system, Proc. VLDB Endow. 6 (11) (2013) 1092–1101. http://dx.doi.org/10.14778/2536222.2536234.

[19] S.-T.L. Sanjay Ghemawat, Howard Gobioff, The google file system, Oper. Syst. Rev. 37 (2003) 29–43.

[20] K.V. Shvachko, HDFS scalability: The limits to growth, LOGIN: Mag. USENIX 35 (2) (2010) 6–16.

[21] Standalone block management for hdfs namenode, https://issues.apache.org/jira/secure/attachment/12618294/Proposal.pdf.

[22] S. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, C. Maltzahn, Ceph: A scalable, highpesrformance distributed file system, 2006.

[23] S.A. Weil, S.A. Brandt, E.L. Miller, C. Maltzahn, Crush: controlled, scalable, decentralized placement of replicated data, in: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, New York, NY, USA, 2006, p. 122. http://doi.acm.org/10.1145/1188455.1188582.

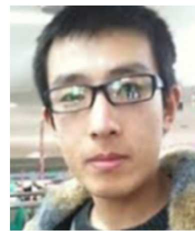
[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, in: Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI), USENIX, 2002, URL: http://www.cs.utah.edu/flux/papers/netbed-osdi02-base.html.

[25] J. Zhang, P. Shang, J. Wang, A scalable reverse lookup scheme using group-based shifted declustering layout, in: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 604–615.

[26] H. Zhu, P. Gu, J. Wang, Shifted declustering: a placement-ideal layout scheme for multi-way replication storage architecture, in: ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing, ACM, New York, NY, USA, 2008, pp. 134–144. http://doi.acm.org/10.1145/1375527.1375549.

Jun Wang received his Ph.D. in Computer Science and Engineering from the University of Cincinnati in 2002. He received the B.Eng. degree in Computer Engineering from Wuhan Technical University of Surveying and Mapping (now Wuhan University) and M.Eng. degree in Computer Engineering from Huazhong University of Science and Technology. He is a tenured Associate Professor of Computer Science and Engineering, and Director of the Computer Architecture and Storage Systems (CASS) Laboratory at the University of Central Florida, Orlando, FL, USA. He has won Dean's Research Professor Award in 2013, and was named Charles N. Millican Faculty Fellow in EECS during 2010–2012. He is the recipient of National Science Foundation Early Career Award 2009 and Department of Energy Early Career Principal Investigator Award 2005. He has authored over 60 publications in premier journals such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, and leading HPC and systems conferences such as HPDC, EuroSys, ICS, Middleware, FAST, IPDPS. He has graduated 6 Ph.D. students who upon their graduation were employed by major US IT corporations (e.g., Google, Microsoft, etc.). He has served as numerous US NSF grant panelists and US DOE grant panelists and TPC members for many premier conferences such as IPDPS, ICPP, HiPC. He currently serves on the editorial board for the IEEE Transactions on Parallel and Distributed Systems since 2012, and was associate editor for International Journal of Parallel, Emergent and Distributed Systems (IJPEDS) during 2010–2012. He serves in the steering committee for the Second International Workshop on Energy Consumption and Reliability of Storage Systems (ERSS 2012), California, USA. He is an IEEE ScalaCom'2012 program committee vice chair. He serves as a program co-chair (storage track) for the 7th IEEE conference on Network, Architecture and Storage (NAS), June 2012. He has co-chaired the 1st International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED 2008) held together with HPCA. He is a Senior Member of IEEE.

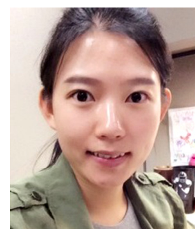
Xuhong Zhang received his B.S. Degree in Software Engineering from Harbin Institute of Technology in 2011 and received his M.S. degree in Computer Science from Georgia State University in 2013. He is currently working toward the Ph.D. degree in the School of EECS at the University of Central Florida, Orlando. His research interests include big data analytic and massive storage/File System.



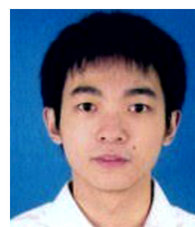
Junyao Zhang received his Ph.D. in Department of Electrical Engineering and Computer Science from University of Central Florida, Orlando. He received his M.S. and B.E. degrees in Software Engineering from the Jilin University, Changchun, China. He is currently a senior software engineer in Ericsson Inc. His primary research interests include scalability, fault-tolerant (fast recovery) and power management in distributed storage systems.

Jiangling Yin received his B.S. and M.S. degrees in software engineering from the University of Macau in 2011. He is working towards his Ph.D. degree in computer engineering from the Electrical Engineering and Computer Science Department, University of Central Florida. His research focuses on energy-efficiency computing and file/storage systems.

Dezhi Han received the B.S. degree from Hefei University of Technology, Hefei, China, the M.S. degree and Ph.D. degree from Huazhong University of Science and Technology, Wuhan, China. He is currently a professor of computer science and engineering at Shanghai Maritime University. His specific interests include storage architecture, cloud computing, cloud computing security and cloud storage security technology.



Ruijun Wang received her B.S. degree in Electrical Information Engineering from the University of Petroleum China (Beijing) in 2007 and received the M.S. degree in Information systems from the University of Central Queensland in 2009. She is currently working toward the Ph.D. degree in the School of EECS at the University of Central Florida, Orlando. Her research interests include energy-efficient computing and storage systems.



Dan Huang received his B.S. and M.S. degrees in Computer Science and Technology from Jilin University, Southeast University and Georgia State University. He is working towards his Ph.D. degree in Department of ECE, University of Central Florida. His research focuses on the I/O of Distributed System and Virtualization Technology.

Please cite this article in press as: J. Wang, et al., Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.03.005