2018 IEEE 38th International Conference on Distributed Computing Systems

D^2-Tree : A Distributed Double-Layer Namespace Tree Partition Scheme for Metadata Management in Large-Scale Storage Systems

Xinjian Luo*, Xiaofeng Gao*§, Zhaowei Tan‡, Jiaxi Liu*, Xiaochun Yang¶ and Guihai Chen*
*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai, China
‡Department of Computer Science, University of California, Los Angeles, Los Angeles, CA, USA
¶School of Computer Science and Engineering, Northeastern University, Liaoning, China
§Corresponding author: gao-xf@cs.sjtu.edu.cn

***Abstract*—The behavior of metadata server (MDS) cluster is critically important to the overall performance of today's petabyte-scale or even exabyte-scale distributed file system. How to maintain a high level of both system locality and load balancing is a significant challenge to MDS clusters. However, traditional metadata management schemes, including hash-based mapping and subtree partitioning, have severe bias on either system locality or load balancing. In this paper, we propose D^2-Tree, a distributed double-layer namespace tree partition scheme, for metadata management in large-scale storage systems. The innovative idea is to design a greedy strategy to split the namespace tree into global layer and local layer subtrees, of which global layer is replicated to maintain load balancing and the lower-half subtrees are allocated separately to MDS's by a mirror division method to preserve locality. Both theoretical analysis based on empirical cumulative distribution and extensive experiments are provided to validate the efficiency of D^2-Tree. Experiments using actual trace data on Amazon EC2 also exhibit the superior performance of D^2-Tree compared with much previous literature.**

***Index Terms*—Metadata Management; Distributed File Systems; Locality; Load Balancing.**

I. Introduction

In today's large-scale distributed file systems, metadata management is an essential issue. Metadata describes the organization and structure of file systems, which usually includes file attributes, file block pointers, etc. [1]. The metadata size, typically 0.1% to 1% of data space [2], is still large in petabyte-scale or even exabyte-scale file systems. More importantly, about 50% to 60% of the access to file systems are pointed to metadata [3]. Thus an efficient metadata management scheme is becoming more and more important to the performance improvement of distributed file systems.

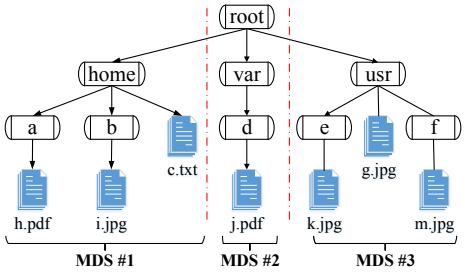
Traditional file systems, such as GFS [4], HDFS [5] and Lustre [6], use a single machine to manage all metadata (we call this machine as *Metadata Server* or MDS henceforth), but this single machine architecture can easily become a bottleneck. Therefore, distributed metadata management schemes have received more and more attention in large-scale distributed file system. Previous designs, like hash-based mapping [7] and subtree partitioning [8], cannot achieve a satisfactory trade-off between locality and load balancing. In addition, few of these works have given a clear and complete *definition* of locality or load balancing, which makes the system performance measurement imprecise and unconvincing.

It is already well-known that the general metadata allocation problem is difficult. Typically how to maintain a high level of both system locality and load balancing is a significant challenge. For instance, in Fig. 1, the basic goal is to partition the file namespace tree and designate those nodes to 3 MDS's. According to POSIX-style standard, accessing a metadata node requires visiting all the ancestor nodes to perform pathname traversal and permission check [9]. Fig. 1(a) shows the traditional subtree partitioning strategy, where the system designates each subtree to a server. This strategy can greatly reduce accessing jumps but may lead to imbalanced workloads among MDS's. Fig. 1(b) illustrates a more balanced partitioning strategy like hash-based mapping, but the metadata locality is worse since accessing the same nodes may require more jumps than in Fig. 1(a).

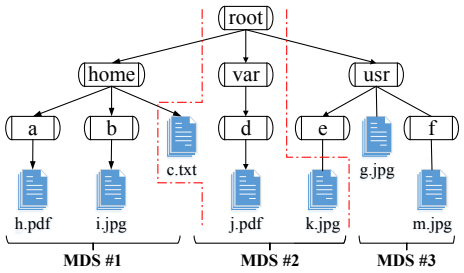
This paper is devoted to developing an efficient and scalable distributed metadata management scheme which can maintain a high level of both locality and load balancing. But the trade-off of locality and load balancing is a great challenge as traditional partitioning/migrating strategies, e.g. dynamic subtree partitioning, are not suitable for realistic workloads of severely skewed access among MDS's [10]. Therefore, we are motivated to develop a superior scheme to solve this issue. In this paper, we first formulate the definitions of both locality and load balancing, and then describe the measurement of distributed metadata system performance formally. Next, we design a novel scheme named D^2-Tree, a ***D**istributed **D**ouble-layer namespace **Tree** partition scheme*, to achieve balanced metadata management with locality preservation under both statical and dynamical scenarios. The key idea of D^2-Tree is to split the namespace tree into two layers: global layer and local layer. Unlike traditional hash-based or subtree-based strategies, our *layer-based* D^2-Tree strategy splits the most popular nodes of file namespace tree into global layer and replicates the upper-half global layer to representative MDS's, which can help greatly maintain the load balancing. The lower-half subtrees are allocated separately to MDS's according to a pre-defined strategy, which means metadata locality can be further preserved by the two-layer structure.

This work has been supported in part by Program of International S&T Cooperation (2016YFE0100300), the China 973 Project (2014CB340303), China NSF Projects (Nos. 61672353, 61472252, 61532021), Shanghai Science and Technology Fund (17510740200), and CCF-Tencent Open Research Fund. Mr. Zhaowei Tao has completed his work when he was an undergraduate student of Shanghai Jiao Tong University, China.

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00021



(a) Better locality



(b) Better load balancing

Fig. 1. Examples of metadata partition.

There are two main challenges in D^2-Tree scheme: first, how to split the namespace tree to handle workloads with severe access bias; second, how to coordinate global layer and local layer to maintain both locality and load balancing. In general, D^2-Tree implements a greedy strategy to determine the cut-line on namespace tree and employ a mirror-division mechanism to allocate subtrees to the MDS cluster. The contribution of our paper is summarized as follows:

- We propose a novel distributed double-layer namespace tree partitioning scheme, named D^2-Tree, to divide the namespace tree into several metadata servers through Tree Splitting, Subtree Allocation and Dynamic Adjustment in Sec. IV, which can fit various statical and dynamical scenarios.
- We define and formularize system locality and load balancing in Sec. III, and in-depth theoretical analysis based on empirical cumulative distribution is provided in Sec. V. We show that D^2-Tree can achieve load balancing with bounded difference while maintaining locality property.
- We conduct extensive experiments on Amazon EC2 to validate the efficiency of D^2-Tree in Sec. VI. The experiments using trace data exhibit the superb performance of D^2-Tree compared to previous literature. The system throughput of D^2-Tree can greatly exceed other schemes in some scenarios.

II. Related Work

Several papers have given the definition of locality or load balancing. Spyglass [11] and DROP [12] provide straightforward examples to define locality, while some papers in the field of both file system and P2P network introduce the idea of load balancing mathematically, such as histogram-based load balancing [13] and load balancing in dynamic structured P2P system [14]. AngleCut [3] gives brief but not complete definitions of locality and load balancing.

Hash-Based Mapping. Static hash generally hashes the file identifier such as file pathname to a digital value and assigns its metadata to a server according to the modulus value with respect to cluster size, which can efficiently balance the workloads among MDS's. CalvinFS [9] and Giga+ [15] both use hash partitioning to distribute metadata across many servers. However, hashing can greatly weaken the metadata access locality since frequent jumps among MDS's are needed for directories traversal. In addition, the overhead of rehashing metadata when renaming an upper directory or scaling the cluster is also considerable.

Subtree Partitioning. Static subtree partitioning partitions the global namespace tree into subtrees and each MDS administers one or more subtree(s) of the directory hierarchy. It provides better locality and greater MDS independence than hash-based mapping, but the workloads may not be evenly partitioned and allocated among MDS's. Dynamic subtree partitioning is an improved scheme based on static subtree partitioning. Some file systems, such as Kosha [16] and Ceph [8], use such a scheme. The key idea is that when a server becomes heavily loaded, this busy server automatically migrates some subdirectories to other servers with less load. The metadata partition granularity of dynamic subtree partitioning is smaller and more flexible compared to that of static subtree partitioning. However, this strategy is too complicated and highly dependent on careful codes to balance workloads appropriately. Thrashing also exists since migrating some subtrees from overloaded server can cause other servers to become overloaded [10].

Other Schemes. Group-based hierarchical bloom filter array (G-HBA) [17] utilizes group-based bloom filters to route requests to target MDS's and improves the scalability of the MDS cluster, while complicating the lookup operations. Dynamic Directory Partitioning (DDP) [18] is a novel metadata management mechanism which manages directory and file metadata separately to avoid massive metadata migrations among MDS's when renaming a directory. DROP [12] exploits locality-preserving hashing to keep excellent namespace locality and uses HDLB strategy to quickly adjust the metadata distribution. However, it suffers space complexity, and inflexibility when the overhead for load balancing or locality changes. AngleCut [3] is another novel hash-based scheme which uses a locality preserving hashing function to project the namespace tree into multiple Chord-like rings, but AngleCut still suffers poor scalability and great rehashing overhead.

III. Problem Statement

In this section, we first present a mathematical description of several important concepts, and then define notations and elaborate the goal of our optimization problem.

A. Notations for the Namespace Tree

In a typical metadata system, let $\{n_j \mid 1 \leq j \leq N\}$ denotes the set of all metadata nodes on the namespace tree, no matter the node represents a file or a folder. Next, we focus on the hierarchical structure of the nodes. For a metadata node n_j, denote $A_j = \{a_j^1, a_j^2, ..., a_j^{s_j}\}$ the set of its ancestors up to the root, and $D_j = \{d_j^1, d_j^2, ..., d_j^{t_j}\}$ as the set of all its children down to the leaves.

Now we are ready to introduce the concept of *locality*. According to the POSIX-style permission checking and interaction standard, the access of a metadata node n_j requires the access of all its ancestor nodes up to the root. If these metadata nodes are located in different MDS's, then we have to *jump* among MDS's to complete one access. The formal definition of a jump is listed in Def. 1.

Definition 1. *Define jp_j as the number of "jumps" among MDS's during one accessing process: $jp_j = \sum_{i=1}^{s_j} \mathbb{1}(a_j^i$ and a_j^{i+1} locate on different MDS's). Here $\mathbb{1}(\cdot)$ is the truth function with output 1 if the inner statement is true, and 0 otherwise.*

We call jp_j the locality value, or the hop distance of metadata n_j. The lower jp_j is, the better locality n_j has.

When considering the *load balancing* property, we should notice that every metadata node has two types of access. One comes from its individual access popularity, while the other comes from the overall access popularity from its children passing by this node (denoted simply as popularity later).

Definition 2. *Denote p'_j as n_j's individual access popularity and p_j as its total access popularity. Then $p_j = p'_j + \sum_{i=1}^{t_j} (p_j^i)'$. Here $(p_j^i)'$ refers to as the individual access popularity of the i-th children of n_j.*

One of our goals is to distribute metadata nodes among MDS's with a satisfiable locality level. Thus it is reasonable to consider the locality property of the whole metadata management system as a weighted sum of all jp_j's.

Definition 3. *Define $locality$ as the global locality value of the whole system, which can be deduced as*

$$locality = 1 \Big/ \sum_{j=1}^{N} jp_j \cdot p_j \ . \tag{1}$$

In this definition, large $locality$ means the locality of system is good. $Locality$ equals $+\infty$ under single server scenario intuitively. Finally, given that the metadata maintenance is a dynamic process, we denote u_j as the cost of update on metadata n_j. Update cost of whole metadata management system is the sum of updates from individual nodes.

Definition 4. *Let $update$ be the total update cost of the whole metadata system, then $update = \sum_{n_j \in GL} u_j$.*

where GL is the metadata node set that will be involved in the update process. Since the updating process is closely related to our scheme design, we will introduce the definition of GL later in Sec. IV-A1.

B. Notations for MDS's

Given M MDS's in the system, we want to design a weighted M-partition of N metadata nodes on the namespace tree, thus each MDS stores a subset of them. Correspondingly, define $L_k = \sum_{n_j \in m_k} p_j$ as the load of the k-th MDS m_k, which is sum of the popularity of all nodes contained in it.

Due to physical limits, every MDS has different load capacity C_k, which can be simply taken as the limitation of an MDS's throughput. Mathematically, to balance the load among MDS's is to minimize $\left| L_k - \frac{\sum_{i=1}^{M} L_i}{\sum_{i=1}^{M} C_i} \times C_k \right|$.

For simplicity, let us define the ideal load factor $\mu = \frac{\sum_{i=1}^{M} L_i}{\sum_{i=1}^{M} C_i}$ as the perfect proportion factor for every MDS. Then we can compute the ideal load for the k-th MDS with $I_k = \mu C_k$. Correspondingly, the relative capacity of the k-th MDS is $Re_k = L_k - I_k = L_k - \mu C_k$. If $Re_k > 0$, we refer to the k-th MDS as heavily loaded. Otherwise, it is light.

If we consider the loads of MDS's as samples in statistics, the ideal load is equivalent to the mean of these samples. We now define the load balance degree of the entire framework.

Definition 5. *Define $balance$ as the load balance degree to all MDS's, which is computed as*

$$balance = 1 \Big/ \frac{1}{M-1} \sum_{k=1}^{M} \left(\frac{L_k}{C_k} - \mu \right)^2. \tag{2}$$

C. Optimization Objective

When partitioning the namespace tree, we aim to improve the locality and load balance degree of the whole system, and there is a trade-off between locality and load balancing properties. Therefore, it is important to find an equilibrium point. In addition, the update cost of global layer (defined in Sec. IV-A1) also needs to be taken into consideration simultaneously.

In our proposed solution, we set two constraints L_0 and U_0 to bound the locality and update cost of the system, and then try to maximize the load balance degree. The optimization problem can be formalized as follows

$$\max \quad balance \tag{3}$$

$$s.t. \quad \left| \bigcup_{j=1}^{M} \{n_i \mid n_i \in m_j\} \right| = N \tag{4}$$

$$\sum_{i=1}^{M} L_i = \sum_{j=1}^{N} p_j \tag{5}$$

$$locality \leqslant L_0, update \leqslant U_0 \tag{6}$$

Eq. (4) specifies that MDS's contain all metadata nodes. Eq. (5) specifies the load of all MDS's is equal to the total access popularity of all metadata system. Eq. (6) specifies the locality constraint and update constraint.

The objective of this problem is to balance loads of participating peers by maximizing the load balance degree $balance$. In addition, the movement cost should be minimized. Next, we prove that this is a NP-complete problem through a reduction from the Partition Problem.

Theorem 1. *The metadata allocation problem for file system is NP-Complete.*

Proof. It is easy to check this problem is NP. Next, we give a reduction from Partition problem to this problem.

An instance of Partition problem is: given a finite set A and a $size(a_i) \in \mathbb{Z}^+$ for each $a_i \in A$, is there a subset $A' \subseteq A$ such that $\sum_{a_i \in A'} size(a_i) = \sum_{a_j \in A \backslash A'} size(a_j)$? Now we construct an instance of the metadata allocation problem. Suppose we have $|A|$ files and 2 homogeneous MDS's. Define the ideal load $I_1 = I_2 = \frac{1}{2}\sum_{i=1}^{|A|} size(a_i)$. Denoting the popularity of the j-th file by $size(a_j)$. This situation is that each file directly links to the root. We will replicate root into both MDS's to maintain a balanced load.

Correspondingly, if the original Partition problem has a YES solution, then the metadata allocation problem has a balanced partition with $balance = 0$. The items in each subset A' and $A \backslash A'$ matches the file metadata nodes in m_1 and m_2. On the contrary, if the metadata allocation problem has a partition result with $balance = 0$, then the Partition problem deducts a YES solution. Therefore, the metadata allocation problem is NP-Complete. □

IV. D^2-Tree Design

In this section, we design D^2-Tree, a distributed double-layer namespace tree partition scheme to achieve distributed metadata management.

A. The Double-Layer Partition

As mentioned before, a parent node has higher popularity than its children in a metadata tree, given that its total access popularity sums the popularity of its children. But traditional partitioning/migrating strategies cannot acquire an optimal trade-off between locality and load balancing due to the access bias of realistic workloads [10].

For namespace decomposition with easier access for each metadata node, we design a novel two-layer tree-splitting mechanism. The basic idea is to cut the namespace tree into upper-half and lower-half. We define the upper-half of the namespace tree, which contains the most popular nodes, as the *global layer*, and the rest lower-half subtrees as the *local layer*. The former is replicated to each MDS to balance the workloads, while the latter will be divided into M groups and each for one MDS to keep locality. D^2-Tree is a flexible design which allows the system to dynamically move the metadata node from the local layer to the global layer, and vice versa. Fig. 2 illustrates an example namespace tree with a red cut-line. The metadata nodes above this line belong to the global layer, while the nodes below this line form the local layer. The yellow nodes are denoted as *inter nodes*, which will be defined later in Sec. IV-A1.

1) The Global Layer and Local Layer: As shown in Fig. 3, nodes in global layer are replicated into every MDS. Denote $GL = \{n_j \mid n_j \in$ global layer$\}$ as the node set in global layer. We now define *inter nodes*. The inter nodes are in the global layer, some or all of whose subtrees are in the local layer. In order to find which MDS an inter node's subtrees lie,

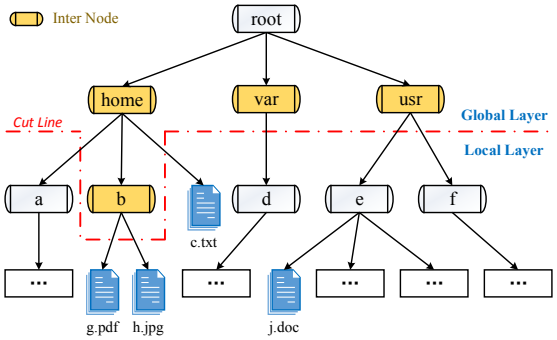


Fig. 2. An Namespace Tree with a Cut-Line

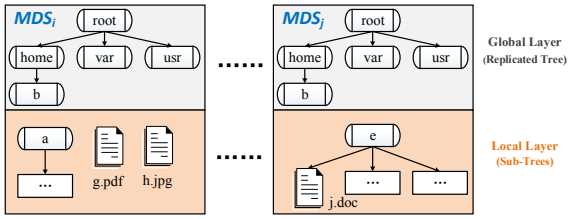


Fig. 3. An Example after the Subtree Allocation

we construct a *local index* for all the roots of subtrees to allow a quick search. The local layer consists of subtree metadata nodes as shown in Fig. 3. Note that the popularity of local layer nodes is relatively lower than global layer. We distribute subtrees to distinct MDS's. Similarly, we denote $LL = \{n_j \mid n_j \in$ local layer$\}$ as the node set of local layer.

Instead of cutting subtrees of local layer into smaller parts, each subtree is treated as an unit. Such design guarantees that at most one hop among MDS's is needed when accessing a node in local layer, thus increasing the system's locality. To simplify the description, we denote each subtree in the local layer as Δ_i, and there are totally H subtrees below the cut-line. Noticeably, the parent of any subtree Δ_i should be an inter node, and one inter node may have more than one subtrees. The popularity of each subtree, denoted as $s_i, i = 1, 2, \cdots, H$, is exactly the popularity of its root, and it also has a close relationship with its parent inter node.

Despite losing some load balancing (smaller pieces are more likely to be allocated evenly than some bigger blocks are), keeping the intact subtree simplifies the process as well. Since the metadata in the global layer has $jp_j = 0$ and the metadata n_j in the local layer has locality $1/(jp_j \cdot p_j) = 1/p_j$, the formula of $locality$ in Eqn. (1) can be written as

$$locality = 1 \Big/ \sum_{j=1}^{N} jp_j \cdot p_j = 1 \Big/ \sum_{n_j \in LL} p_j \qquad (7)$$

2) Cache Policy and Access Logic: Many traditional strategies require a large amount of cache memory to store prefix nodes such as dynamic subtree partitioning. For D^2-Tree, the cache overhead will be much lower than other strategies.

Recall that the global layer is replicated to each MDS and the local layer is dispersed among MDS cluster. A subtree is designated to at most 2 MDS's, which means that an MDS does not need to cache all the prefix nodes of a certain metadata node. The cache module of MDS's only needs to save hot or being-written metadata which are parts of global layer. Further, the clients also have a cache module, which is employed to store *inter nodes*, or *local index* more specifically. Version number, timeout and lease mechanism in GFS [4] are also employed to maintain the consistency and reliability of server/client cache.

When querying a metadata node, a client C first checks if one of the prefix nodes dwells in the local index cached in the local-host. If yes, C just needs to send the query to the MDS which is responsible for the corresponding children subtrees of the inter node. Otherwise, it means this metadata node lies in global layer, any MDS randomly chosen can process the query. For example, suppose C wants to access "/root/home/b/h.jpg" in Fig. 2. First, C checks if one of "/root", "/root/home" and "/root/home/b" is included in the cached local index. If all these prefix nodes are not included in local index, which means "/root/home/b/h.jpg" is included in global layer, C just needs to send the request to any randomly chosen MDS in the cluster since global layer dwells in each MDS. If one of the prefix nodes is an inter node, e.g. "/root/home/b", and lies in m_i, C simply sends the request to m_i, and the request will be processed by m_i smoothly. Then the acquired metadata is returned back to C and the whole process is completed.

3) The Philosophy: Most metadata management systems, such as dynamic subtree partitioning, employ self-organizing architecture, which complicates system design and MDS behaviors and aggravates the workload of MDS in some ways. In D^2-Tree, a Monitor is added to the cluster to simplify the MDS behaviors, just like the monitor of object storage devices (OSD's) in Ceph. The Monitor needs to 1) accept periodically heartbeats from MDS's and maintain a pending pool for dynamic subtree adjustment; 2) maintain the consistency of global layer and server cache among MDS's; 3) detect cluster status, including MDS failure and new MDS added. Note that the monitor can hardly become the performance bottleneck since the MDS cluster size is much smaller than the size of OSD cluster. The lock service of Zookeeper is used to keep data consistency over global layer. Note that clients require a lock only when they want to modify the nodes in global layer.

B. The D^2-Tree Scheme

Next, we will introduce the three phase algorithms for D^2-Tree, i.e., *Tree-Splitting*, *Subtree-Allocation*, and *Dynamic-Adjustment*.

Tree-Splitting for Namespace Decomposition. If we consider the locality mentioned above and a given locality bound L_0 to this metadata tree, it is easy to find that if more metadata nodes are added to the global layer, the $locality$ increases, which means we should put nodes in the global layer as much as possible. However, putting more nodes in the global layer is likely to bring more cost when we execute the update procedure. Given update cost bound U_0, we should check whether the next procedure will transcend the bound U_0.

Therefore, from the root with a top-down manner, we check the children nodes of all nodes in the global layer and choose one node with the biggest p_j. Then, we check whether the sum of the locality satisfies: $locality < L_0$. If not, we need to proceed to decrease the locality value. Next, we need to check whether the sum of the update cost in the global layer satisfies: update cost $update \leq U_0$. If it does not, we should stop the procedure. Thus, we have the Alg. 1.

Algorithm 1: Tree-Splitting Algorithm

Input: A metadata tree, the metadata's p, u, the locality threshold L_0, and the update cost threshold U_0.
Output: A set GL, where the metadata in it is placed in the global layer.

$L_{tmp} = \sum p$, $U_{tmp} = 0$, $GL = \{n_{root}\}$, $S = \{n_{root}.children()\}$;
while *true* **do**
  S sorts according to p;
  n_x = the metadata with the biggest p in S;
  $U_{tmp} = U_{tmp} + update_x$;
  if $U_{tmp} \geq U_0$ **then** break;
  $GL.append(n_x)$;
  $S.delete(n_x)$;
  $S.append(n_x.children())$;
  $L_{tmp} = L_{tmp} - p_x$;
if $L_{tmp} > L_0$ **then return** { };
else return GL;

Subtree-Allocation for Metadata Assignment. After the splitting process, the local layer is now composed of many subtrees. The main challenge now is to find a way to uniformly allocate these subtrees on the M MDS's. It is an optimization problem. Let $L_k, 1 \leq k \leq M$ be the current load of the MDS i and the $C_k, 1 \leq k \leq M$ be the maximum capacity. The current load means the sum of the popularity $\sum p_j$ for all metadata in this MDS.

Our allocation algorithm intends to balance loads of participating peers by maximizing the load balance degree *balance*. It also aims to reduce the movement cost as much as possible. We have proven that it is an NP-Complete problem, before we present our algorithm, a nearly optimized algorithm, inspired by [19], needs to be introduced. We first describe the Histogram-Based Probability Distribution. Given a random variable $\boldsymbol{Z}$ and its probability distribution $Pr(\boldsymbol{z})$, we rely on histograms as defined below to approximate the distribution.

Definition 6. *The histograms which represent $Pr(\boldsymbol{Z})$ are denoted by $\{x_i, i = 1, 2...k; \Delta x\}$. Here, $x_i < x_j$ for any $1 \leq i \leq j \leq k$ and $[x_i, x_{i+1}], i = 1, 2, ..., k-1$ indicates the interval in the domain of $\boldsymbol{Z}$ such that*

$$Pr(x_i \leq \boldsymbol{Z} \leq x_{i+1}) = \Delta x \tag{8}$$

$$\sum_{i=1}^{k-1} Pr(x_i \leq \boldsymbol{Z} \leq x_{i+1}) = 1 \tag{9}$$

With histogram-based approximation, we can efficiently solve the allocation problem. The key design to "query" in our proposal is that for each Δ_i, we estimate *cumulative distribution function* (CDF), which is denoted by $F_\Delta(x) = Pr(\boldsymbol{X} \leq x)$. Here the random variable $\boldsymbol{X}$ represents the aggregate of some subtrees' s_i, which means $\boldsymbol{X} \in \left[0, \sum_{i=1}^{N} s_i\right]$. Similarly, we represent the CDF of the remaining capacities of some MDS's R_i by random variable $\boldsymbol{Y}$: $F_m(y) = Pr(\boldsymbol{Y} \leq y)$, which means $\boldsymbol{Y} \in \left[0, \sum_{k=1}^{M} R_k\right]$. Remarkably, the initial remaining capacity R_k is just C_k since all MDS's are empty and the remaining capacity is no more than C_k all the time.

Each light MDS m_k computes the $F_\Delta(x)$ and $F_m(x)$. Then, m_k queries and requests subtrees in the pending pool (see Dynamic Adjustment) which satisfy

$$\{t \in \boldsymbol{P} : F_\Delta(R_{i-1}) < F_\Delta(s_t) \leq F_\Delta(R_i)\} \tag{10}$$

It allocates subtrees to the corresponding MDS's, i.e., MDS's request the subtrees with popularity proportional to their remaining capacities, resulting in a load-balanced allocation.

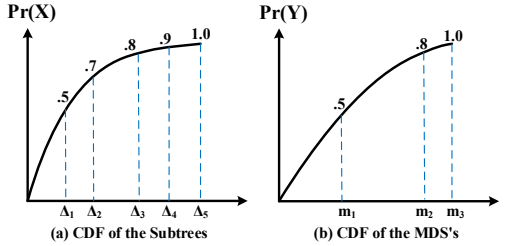


Fig. 4. Mirror-Division Strategy with 2 CDF Curves

Fig. 4 illustrates an example of our idea, where there are five subtrees $\{\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5\}$ with popularity s_1, s_2, s_3, s_4 and s_5 and three light MDS's $\{m_1, m_2, m_3\}$ with remaining capacities R_1, R_2 and R_3. As shown in Fig. 4, Δ_1 manages total popularity equal to 0.5. Such ratios for $\Delta_2, \Delta_3, \Delta_4, \Delta_5$ are $0.2, 0.1, 0.1, 0.1$ respectively. Thus, $\boldsymbol{X}_{\Delta_1}^{index} = 0.5$, $\boldsymbol{X}_{\Delta_2}^{index} = 0.7$, $\boldsymbol{X}_{\Delta_3}^{index} = 0.8$, $\boldsymbol{X}_{\Delta_4}^{index} = 0.9$ and $\boldsymbol{X}_{\Delta_5}^{index} = 1.0$. Similarly, the aggregate remaining capacities of the MDS's are $(0, \boldsymbol{Y}_{m_1}^{index}) = (0, 0.5)$, $(\boldsymbol{Y}_{m_1}^{index}, \boldsymbol{Y}_{m_2}^{index}) = (0.5, 0.8)$,$(\boldsymbol{Y}_{m_2}^{index}, \boldsymbol{Y}_{m_3}^{index}) = (0.8, 1.0)$.

As a result, MDS m_1 hosts subtree Δ_1, m_2 hosts subtrees Δ_2, Δ_3, m_3 hosts subtrees Δ_4, Δ_5. This indicates that the MDS's allocate the popularity of the subtrees in the pending pool proportional to their remaining capacities.

To large scale metadata storage system, the quantity of subtrees is considerably huge. It is not practical to leverage all the subtrees to simulate the CDF. In fact, each MDS in our proposal samples a number of subtrees based on a random walk [20], which aims to reduce the cost. Through the random walk, the samples precisely approximate $Pr(\boldsymbol{X})$ if the number of samples is sufficient. We analyze the precision of the approximation in Sec. V.

Dynamic-Adjustment for Update Process. The Tree Splitting Algorithm was designed for static model. However, both the size and popularity of subtrees change over time in an unpredictable manner, dynamic adjustment is needed for maintaining an optimal distribution of workloads. In D^2-Tree, each MDS node m_k periodically sends heartbeat information including a description of current load level L_k and relative capacity Re_k to Monitor. Monitor maintains a *pending pool* which contains information of subtrees from relatively overloaded MDS's, and lightly loaded server or new-coming server can initiatively request some subtrees from the pending pool to keep system balanced. Moreover, MDS's use access counters whose values decay over time to monitor the popularity of internodes and metadata nodes of local layer. MDS's periodically check these counters and record the change over time, then send these information to Monitor to help adjust global layer. Note that adjusting global layer, which may bring about update overhead, should be executed less frequently, typically once a day in our expriments.

V. Theoretical Analysis

The theoretical analysis on D^2-Tree performance is presented in this section. We will prove the accuracy of subtree allocation algorithm and give the bounds of load balance degree of D^2-Tree.

In our proposal, each MDS needs to sample to approximate the $F_\Delta(x)$ since the quantity of subtrees in large scale metadata storage system is considerably huge. We will analyze the quantity of subtrees every MDS needs to sample to guarantee that the approximation is accurate enough, and we will also analyze the bounds of load balance degree of our proposal.

Our analysis mainly relies on Dvoretzky-Kiefer-Wolfowitz inequality as stated in the following:

Theorem 2. *Let $\boldsymbol{Z}_1, \boldsymbol{Z}_2, ..., \boldsymbol{Z}_k$ be independent and identical distribution random variables with a CDF $F(\cdot)$. Let $F_k(\cdot)$ denote the empirical CDF, which approximates $F(\cdot)$, defined as $F_k(z) = \frac{\sum_{i=1}^{k} \mathbb{1}_{\{\boldsymbol{Z}_i \leq z\}}}{k}$. Then, for any $\varepsilon > 0$*

$$Pr\left(sup\,|F_k(z) - F(z)| > \varepsilon\right) \leq \frac{2}{e^{2k\varepsilon^2}}. \tag{11}$$

We use $\tilde{F}_\Delta(x)$ to estimate $\tilde{F}_\Delta(x)$ to reallocate its subtrees for light MDS m_i, the subtrees to be allocated is located in the interval $(F_m(R_{i-1}), F_m(R_i)]$. Suppose we now have two subtrees i and j which have the same index in two different CDF's, i.e. $\tilde{F}_\Delta(s_i) = F_\Delta(s_j)$. Here the CDF corresponding to Δ_i is the approximated CDF by an MDS and the CDF corresponding to Δ_j is the real CDF. We want to analyze the $|s_i - s_j|$ and make sure it is small enough.

Lemma 1. *Consider two different subtrees which have indices satisfying $\tilde{F}_\Delta(s_i) = F_\Delta(s_j)$. Denote $U = \max\{s_i \mid 1 \leq i \leq H\}$ and $L = \min\{s_i \mid 1 \leq i \leq H\}$. For any $0 < t < 1$ and any $\delta > 0$, if the MDS samples $\frac{ln(t \cdot H)}{2}(\frac{U-L}{\delta})^2$ subtrees uniformly at random from pending pool, we have $\mathbb{E}[|s_i - s_j|] < \delta$ with a probability no less than $1 - s_1(t, H)$.*

TABLE I
THE DESCRIPTION OF 3 DATASETS.

Trace Name	Size	Records	Max Depth	Brief Description
Development Tools Release	5.9 GB	34,349,109	49	Collected for Developers Tools Release server.
Live Maps Back End	15.1 GB	88,160,590	9	Collected for LiveMaps back-end server.
Radius Authentication	39.3 GB	259,915,851	13	Collected for RADIUS authentication server.

Proof. Prior studies [21] have proved the continuum approach to analyze the performance of large-scale system. Now, consider the differential of $F_\Delta(\cdot)$, we have

$$\frac{dF_\Delta(x)}{dx}\Big|_{x=s_j} = \frac{F_\Delta(s_i) - F_\Delta(s_j)}{s_i - s_j} = \frac{F_\Delta(s_i) - \tilde{F}_\Delta(s_i)}{s_i - s_j}.$$

As $\tilde{F}_\Delta(s_i) = F_\Delta(s_j)$ and the MDS samples $\frac{ln(t\cdot H)}{2}(\frac{U-L}{\delta})^2$ subtrees uniformly. From Thm. 2, if we take $\varepsilon = \frac{\delta}{U-L}$, we have

$$Pr\left(\sup|F_\Delta(s_i) - \tilde{F}_\Delta(s_i)| < \frac{\delta}{U-L}\right) \geq 1 - \frac{2}{t\cdot H}.$$

Thus $\frac{dF_\Delta(x)}{dx} < \frac{\delta}{(U-L)(s_i-s_j)}$ with a probability no less than $1 - \frac{2}{t\cdot H}$. Note that if $s_i < s_j$ here, $\frac{dF_\Delta(x)}{dx} > \frac{\delta}{(U-L)(s_j-s_i)}$ with a probability no more than $\frac{2}{t\cdot H}$. Then we have

$$\begin{aligned}\mathbb{E}\left[|s_i - s_j|\right] <& \mathbb{E}\left[\delta \Big/ (U-L)\left(\frac{dF_\Delta(x)}{dx}\right)\right] \\ =& \frac{\delta}{U-L}\int_{x=L}^{U} \frac{1}{\frac{dF_\Delta(x)}{dx}}\frac{dF_\Delta(x)}{dx}dx = \delta.\end{aligned}$$

□

Now that we have known the bound of the approximation error to one subtree, we are ready to propose the error bound of every MDS.

Theorem 3. *Let p_k be the proportion of the maximum capacity of MDS m_k in all MDS's, i.e. $p_k = \frac{C_k}{\sum_{i=1}^{M} C_i}$. For any $0 < t < 1$ and any $\delta > 0$, if the MDS m_k samples $\frac{ln(t\cdot H^2)}{2}(\frac{H\cdot p_k\cdot(U-L)}{\delta\cdot\mu\cdot C_k})^2$ subtrees uniformly at random from the pending pool, we have $\mathbb{E}\left[\left|\frac{L_k}{C_k} - \mu\right|\right] < \delta\mu$ with a probability no less than $1 - \frac{2}{t\cdot H}$.*

Proof. As the MDS in chosen subtrees satisfying $\{t \in \boldsymbol{P} : F_m(R_{i-1}) < F_\Delta(s_t) \leq F_m(R_i)\}$, we denote the condition as Ω. Hence, we have

$$\left|\frac{L_k}{C_k} - \mu\right| = \frac{|\sum_{i\in\Omega} s_i - \tilde{s_i}|}{C_k} < \frac{\sum_{i\in\Omega}|s_i - \tilde{s_i}|}{C_k},$$

where $\tilde{s_i}$ denotes subtree with a popularity index $F_\Delta(\tilde{s_i}) = F_\Delta(s_i)$. As the number of chosen subtrees is related to the real distribution of the subtrees and the load error of every chosen subtree is related to the sampling, the condition $i \in \Omega$ is independent of $|s_i - \tilde{s_i}|$. Thus we have

$$\mathbb{E}\left[\left|\frac{L_k}{C_k} - \mu\right|\right] = \frac{\mathbb{E}[\sum \mathbb{1}_{\{i\in\Omega\}}]\mathbb{E}[|s_i - \tilde{s_i}|]}{C_k}.$$

TABLE II
OPERATION BREAKDOWNS FOR VARIOUS TRACES.

	DTR	LMBE	RA
Read	67.743%	78.877%	47.734%
Write	26.137%	21.108%	36.174%
Update	6.119%	0.015%	16.102%

From Lem. 1, we have $\mathbb{E}[|s_i - \tilde{s_i}|] \leq \frac{\delta\cdot\mu\cdot C_k}{H\cdot p_k}$ with a probability no less than $1 - \frac{2}{t\cdot H}$. Obviously, we also have $\mathbb{E}[\sum 1_{\{i\in\Omega\}}] = H \cdot p_k$. Thus

$$\mathbb{E}[|s_i - \tilde{s_i}|] \leq \frac{H\cdot p_k \cdot \frac{\delta\cdot\mu\cdot C_k}{H\cdot p_k}}{C_k} = \delta\cdot\mu$$

with a probability no less than $1 - \frac{2}{t\cdot H}$. □

Now, we have shown any MDS's expectation gap to its ideal load. Since all MDS's sample the subtrees independently, the load balance degree is thus the sum of every MDS's load balance degree. From theorems proposed above, we finally give the theoretical bound of the whole load balance degree.

Theorem 4. *For any $0 < t < 1$ and any $\delta > 0$, if any MDS samples $\frac{ln(t\cdot H^2)}{2}(\frac{H\cdot p_k\cdot(U-L)}{\delta\cdot\mu\cdot C_k})^2$ subtrees uniformly at random from the pending pool, we have*

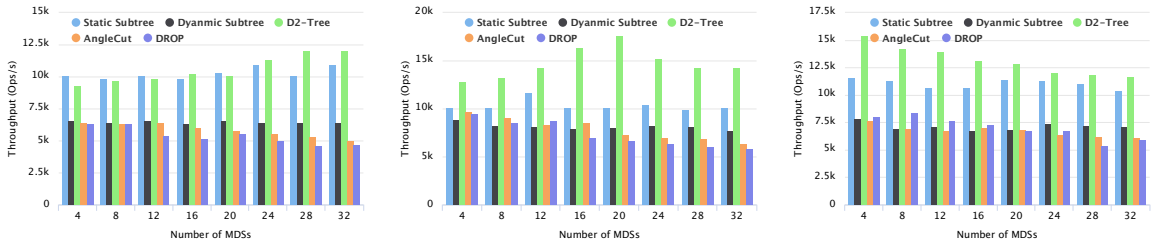
$$\mathbb{E}[balance] < \frac{M}{M-1}\delta^2\mu^2. \tag{12}$$

VI. EXPERIMENTAL RESULTS

We implemented a D^2-Tree metadata management system to evaluate the system performance and MDS behavior. The static subtree partitioning, dynamic subtree partitioning, DROP [12] and AngleCut [3] were also implemented for comparison.

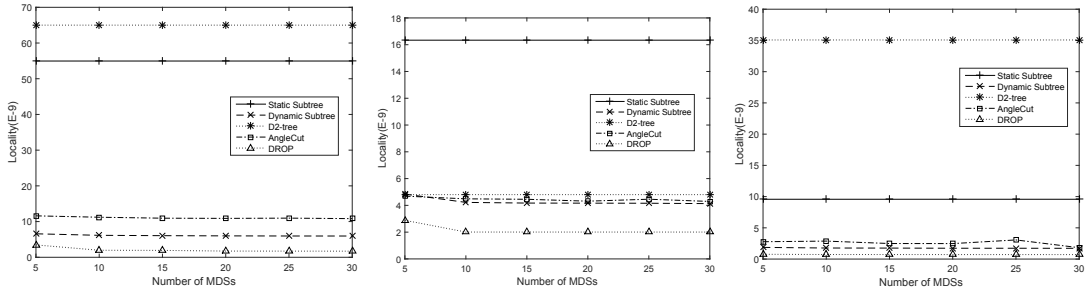
Datasets. To facilitate our simulation, we used 3 trace datasets, i.e. *Development Tools Release* (*DTR*), *Live Maps Back End* (*LMBE*) and *Radius Authentication* (*RA*) [22], to evaluate the behavior of D^2-Tree and other schemes for comparison. These datasets were all collected from Microsoft® servers for a duration of 24-hours. The detailed information of these datasets is described in Table I. Since the focus of our experiments was on the behaviors of MDS, we simply filtered the 3 datasets to get only metadata-related operations such as *read*, *write*, *update*, and the operation breakdowns for various traces are shown in Table II. Note that D^2-Tree is independent from underlying file objects, operations like *read* and *write* only cause simply a query operation to MDS's.

Implements. For static subtree partitioning, the initial metadata partition was created by hashing directories near the root



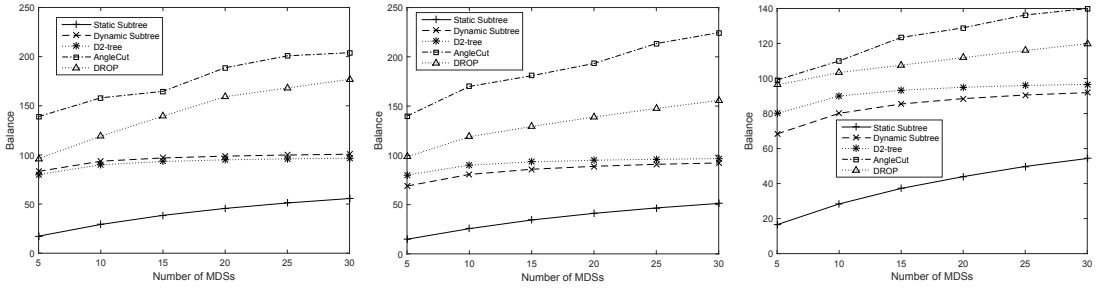
(a) Development Tools Release (b) Live Maps Back End (c) Radius Authentication

Fig. 5. Throughput as the MDS cluster is scaled.



(a) Development Tools Release (b) Live Maps Back End (c) Radius Authentication

Fig. 6. Locality performance under different schemes.



(a) Development Tools Release (b) Live Maps Back End (c) Radius Authentication

Fig. 7. Load balancing performance under different schemes.

of the hierarchy. The initiation of dynamic subtree partitioning is very similar to that of static subtree partitioning, except that the subtrees need to be split into smaller subtrees with finer granularity and dynamic migration is executed when the system is unbalanced. AngleCut and DROP were both implemented with locality-preserving hashing, but AngleCut projects the namespace tree to multiple Chord-like rings and then allocates metadata nodes to MDS's on the basis of their location on rings, which is greatly different from DROP.

Parameter Setting. Recall that in Sec.III-C, we split the file system namespace to obtain the global layer within the restrictions of U_0 and L_0. More specifically, these two constraints can determine how many nodes there are in global layer, which is an important factor that affects the behaviors of D^2-Tree. We chose proper U_0 and L_0 to make global layer account for 1% nodes of the whole namespace tree. More details about U_0 and L_0 will be discussed in Sec. VI-C.

Platform. All the experiments were run on Amazon® EC2 High-CPU instances. Each instance is composed of DualCore Intel $Xeon$ E5-2676 v3 Processors and 8GB memory. The bandwidth of network link is 100 Mbps. Given that MDS clusters usually have no more than 128 servers in reality, our experiments used 33 instances for evaluation, among which 1 for MDS Monitor and 32 for metadata servers.

A. Performance Evaluation

Initially, we evaluated the performance of D^2-Tree by fixing the client base to 200 and scaling MDS cluster. Fig. 5 shows the performance of different strategies as the MDS cluster is scaled. The performance of dynamic subtree partitioning, DROP and AngleCut are not as good as static strategy or D^2-Tree, since in dynamic strategy and hash-based strategies, subtrees are partitioned with finer granularity and designated among MDS's, which means queries to the same node will

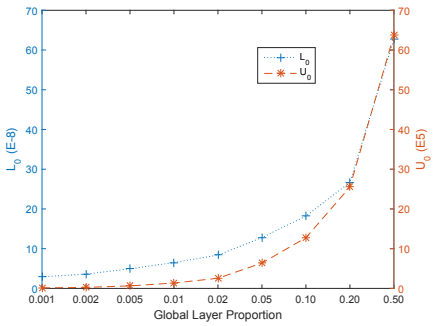


Fig. 8. L_0 and U_0 under different GL proportions.

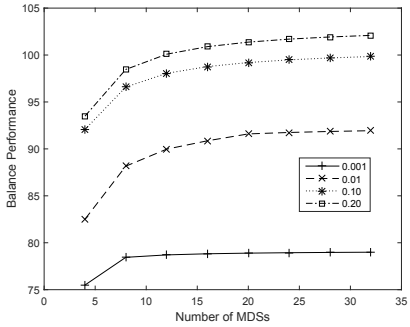


Fig. 9. Balance performance as the MDS cluster is scaled.

be forwarded multiple times to perform path traversal as the cluster is scaled.

Notice that the behaviors of D^2-Tree in these three datasets are different from each other due to the difference in workload distributions. Table II shows a breakdown of the operations. We can see that most operations in *DTR* and *LMBE* are queries, while 16% operations in *RA* are updates. In *DTR*, 83.06% queries are directed to global layer of D^2-Tree. Since each MDS of D^2-Tree can process queries directed to global layer, the performance of D^2-Tree improves as the MDS cluster is scaled. For *LMBE*, 58.57% of its queries are directed to local layer. When the MDS's are more than 20, its performance is degraded because many queries in the local layer need more jumps among MDS's to perform path traversal. For dataset *RA*, the workload includes 16% modify operations, of which 67% operations are directed to global layer. Since we employ lock mechanism to maintain the data consistency of global layer, more modify operations to global layer can cause longer request delay and the cluster performance will degrade simultaneously. But we can see that even bearing update overhead, D^2-Tree still outperforms dynamic subtree partition, DROP and AngleCut. Though static subtree partition outperforms D^2-Tree in *DTR*, it is rarely employed in actual file system since such strategy requires manual intervention and can cause a severe load imbalance problem.

B. Locality and Load Balancing

Next we evaluate the performance of locality and load balancing of D^2-Tree. Results are shown in Fig. 6 and Fig. 7. Note that all the values of $locality$ and $balance$ are computed according to Eq. (1) and Eq. (2).

In Fig. 6, *DTR*'s locality performance of D^2-Tree outperforms all the other schemes. As mentioned before, 83.6% queries of *DTR* are directed to global layer, which means most queries can be processed without jumps among MDS's. According to Eq. (1), $locality$ is concerned with jp_j and p_j. Note that the same workload leads to the same p_j, so D^2-Tree with the least jumps performs best. Notice that the $locality$ of D^2-Tree and static subtree partitioning remain unchanged as the cluster is scaled, the reason is without further partition to subtrees, jp_j keeps unchanged and that leads to the same $locality$. But in dynamic strategy, DROP and AngleCut, when the cluster is scaled, subtrees are partitioned with finer-granularity to balance the workloads among MDS's, so more jumps bring worse locality. In Fig. 6, *LMBE*'s locality of static subtree partitioning is better than D^2-Tree since 58.57% queries in D^2-Tree are directed to local layer. Fig. 6 also demonstrates locality performance is a main drawback of AngleCut and DROP.

The adjustment of workloads among MDS's is a dynamic process, after the subtraces are replayed to these clusters for 20 times, a relatively balanced status is maintained among these clusters. Fig. 7 illustrates the load balancing performance of different schemes as MDS cluster is scaled. It is clear that DROP and AngleCut own the better balance performance than other schemes since the inherent superiority of hashing. The balance performance of D^2-Tree is better than dynamic subtree partitioning in datesets *LMBE* and *RA*. It is well known that dynamic subtree strategy can split file namespace into smaller subtrees to maintain load balancing. However, in realistic workloads, a few subtrees can control the most proportion of system query throughput. These flow-control subtrees cause the imbalance of whole metadata system and simple partition & migration to these subtrees cannot break the imbalance. D^2-Tree puts these flow-control nodes in global layer and replicates them to all the MDS's, which can disperse the query pressure and maintain a more balanced status.

C. Analysis on Global Layer

Review that given two constraints L_0 and U_0 for bounding the locality and update cost of the system, the optimization objective of D^2-Tree is to maximize the load balance degree. These two constraints can determine the number of nodes in global layer. Without loss of generality, we use dataset *DTR* to explain how U_0 and L_0 affect the performance of D^2-Tree.

Fig. 8 shows the values of L_0 and U_0 with different global layer proportions in a 4-MDS cluster. From Fig. 8 we can see that the locality and update overhead both increase as the proportion of global layer increases. When nodes of global layer account for a greater proportion of file namespace tree, there will be less nodes dwelling in local layer, the system locality will be better. D^2-Tree replicates global layer to all

the MDS's, however, update overhead of global layer will increase simultaneously. Fig. 9 shows the balance performance under different global layer proportions as the cluster is scaled. For simplicity, we chose 0.001, 0.01, 0.1 and 0.2 respectively as the proportion that global layer accounts for. We can see the balance performance of D^2-Tree becomes better as the global layer proportion increases. The reason is the greater proportion global layer accounts for, the more subtrees with finer-granularity will be split into local layer, and these smaller subtrees will be more evenly designated among MDS's.

In above experiments we adjust U_0 and L_0 to make 1% nodes of file namespace tree as the global layer, since this proportion can achieve a satisfactory performance without worrying about great update cost in most circumstances according to our plenty of trials.

VII. Discussion

An elaborate comparison between server replication and client caching on operation latency, consistency, scalability, availability, etc. is conducted in [23]. Client cache strategies require a large amount of client memory to store prefix nodes, such as hash-like metadata systems, and thus the performance of these metadata partitioning strategies is tightly linked to metadata cache efficiency. However, client caching can involve higher latency when cache misses and highly relies on cache locality for query efficiency and load balancing. On the contrary, even though causing some cost of consistency maintenance, D^2-Tree can achieve better locality, load balancing and higher throughput. Moreover, with concise design, D^2-Tree can be easy to scale linearly with *read* intensive workloads.

D^2-Tree replicates global layer (1% nodes of namespace tree in our experiments) to each MDS. While the MDS cluster is scaled, metadata consistency and performance degradation might be a challenge to D^2-Tree with *update* intensive workloads. There are several strategies to deal with such scenarios, like adjusting the nodes of global layer and setting a threshold to control the number of replications of global layer, we will put this in our future work.

VIII. Conclusions

In this paper, we first define and formularize system locality and load balancing elaborately, then present a novel distributed double-layer namespace tree partition scheme, called D^2-Tree, for metadata management. D^2-Tree can maintain a high level of both locality and load balancing. Theoretical analysis and experiments exhibit D^2-Tree's validity and superiority to other works. We are also working to integrate D^2-Tree to our object-based distributed file system to fully evaluate and improve D^2-Tree further.

References

[1] J. Xiong, Y. Hu, G. Li, R. Tang, and Z. Fan, "Metadata distribution and consistency techniques for large-scale cluster file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, pp. 803–816, 2011.

[2] E. L. Miller, K. Greenan, A. Leung, D. Long, and A. Wildani, "Reliable and efficient metadata storage and indexing using NVRAM," 2008, online talk at http://dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf.

[3] J. Liu, R. Wang, X. Gao, X. Yang, and G. Chen, "Anglecut: A ring-based hashing scheme for distributed metadata management," in *International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 71–86.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM Symposium on Operating System Principles (SOSP)*, vol. 37, no. 5, 2003, pp. 29–43.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[6] S. Donovan, G. Huizenga, A. Hutton, C. Ross, M. Petersen, and P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *The Ottawa Linux Symposium (OLS)*, 2003.

[7] S. A. Brandt, E. L. Miller, D. D. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2003, p. 290.

[8] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2004, p. 4.

[9] A. Thomson and D. J. Abadi, "Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems." in *FAST*, 2015, pp. 1–14.

[10] Q. Xu, R. V. Arumugam, K. L. Yang, and S. Mahadevan, "Drop: Facilitating distributed metadata management in eb-scale storage systems," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–10.

[11] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *USENIX Conference on File and Storage Technologies (FAST)*, vol. 9, 2009, pp. 153–166.

[12] Q. Xu, R. V. Arumugam, K. L. Yong, and S. Mahadevan, "Efficient and scalable metadata management in EB-scale file systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 11, pp. 2840–2850, 2014.

[13] Q. H. Vu, B. C. Ooi, M. Rinard, and K.-L. Tan, "Histogram-based global load balancing in structured peer-to-peer systems," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 21, no. 4, pp. 595–608, 2009.

[14] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *IEEE Conference on Computer Communications (INFOCOM)*, vol. 4, 2004, pp. 2253–2262.

[15] S. Patil and G. A. Gibson, "Scale and concurrency of giga+: File system directories with millions of files." in *FAST*, vol. 11, 2011, pp. 13–13.

[16] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu, "Kosha: A peer-to-peer enhancement for the network file system," *Journal of Grid Computing*, vol. 4, no. 3, pp. 323–341, 2006.

[17] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and adaptive metadata management in ultra large-scale file systems," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE, 2008, pp. 403–410.

[18] Y. Fu, N. Xiao, and E. Zhou, "A novel dynamic metadata management scheme for large distributed storage systems," in *International Conference on High Performance Computing and Communications (HPCC)*, 2008, pp. 987–992.

[19] H.-C. Hsiao and C.-W. Chang, "A symmetric load balancing algorithm with performance guarantees for distributed hash tables," *IEEE Transactions on Computers (TOC)*, vol. 62, no. 4, pp. 662–675, 2013.

[20] P. Fonseca, R. Rodrigues, A. Gupta, and B. Liskov, "Full-information lookups for peer-to-peer overlays," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 9, pp. 1339–1351, 2009.

[21] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 12, no. 10, pp. 1094–1104, 2001.

[22] "Development tools release, live maps back end and radius authentication." http://iotta.snia.org/traces/158.

[23] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "Shardfs vs. indexfs: replication vs. caching strategies for distributed metadata management in cloud storage systems," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 236–249.