An Efficient Ring-Based Metadata Management Policy for Large-Scale Distributed File Systems

Yuanning Gao, Xiaofeng Gao, *Member, IEEE*, Xiaochun Yang, *Member, IEEE*, Jiaxi Liu, and Guihai Chen, *Senior Member, IEEE*

Abstract—The growing size of modern file system is expected to reach EB-scale. Therefore, an efficient and scalable metadata service is critical to system performance. Distributed metadata management schemes, which use multiple metadata servers (MDS's) to store metadata, provide a highly effective approach to alleviate the workload of a single server. However, it is difficult to maintain good metadata locality and load balancing among MDS's at the same time. In this paper, we propose a novel hashing scheme called AngleCut to partition metadata namespace tree and serve large-scale distributed storage systems. AngleCut first uses a locality preserving hashing (LPH) function to project the namespace tree into linear keyspace, i.e., multiple Chord-like rings. Then we design a history-based allocation strategy to adjust the workload of MDS's dynamically. Besides, we propose a two-layer metadata cache mechanism, including server-side cache and client-side cache to provide the two stage access acceleration. Last but not least, we introduce a distributed metadata processing 2PC Protocol Based on Message Queue (2PC-MQ) to ensure data consistency. In general, our scheme preserves good metadata locality as well as maintains a high load balancing between MDS's. The theoretical proof and extensive experiments on Amazon EC2 demonstrate the superiority of AngleCut over previous literature.

Index Terms—Metadata management, locality preserving hashing, distributed storage system, namespace tree

1 Introduction

Today's file systems are designed to support storing PB-scale or even EB-scale data across tens of thousands of servers. Under this circumstance, metadata processing is a key issue to improve the system performance. Metadata are the data describing file attributes, including directory contents, file size as well as file block pointers [1]. Moreover, metadata preserve the mapping from file name to record location, which can be used to retrieve the file data among data storage nodes. Although the size of metadata is typically 0.1 percent to 1 percent of actual data, it can still reach up to being PB-scale in an EB-scale file system [2]. More importantly, around 50 to 80 percent access of the files is related to metadata [3]. Therefore, it is significant to design an efficient and scalable metadata management scheme.

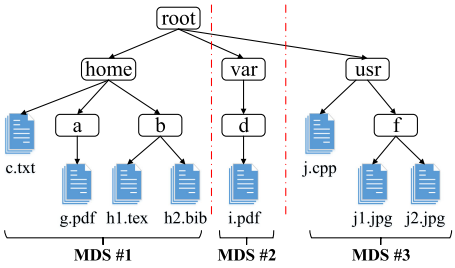
Distributed metadata management [4], [5], which manages metadata by multiple metadata servers is urgently required for large-scale storage systems. In distributed management, maintaining a hierarchy locality [6], [7] of metadata namespace tree and load balancing [8], [9] between MDS's are research hotspots nowadays. Fig. 1 shows two kinds of metadata management schemes, in which the basic goal is to allocate the nodes of a metadata namespace tree to 3 MDS's. According to the POSIX-style permission checking and interaction standard, accessing a metadata node requires the access of all its ancestor nodes (prefix inodes) up to the root to check the Read/Write permission or other properties. Thus, the access to "/home/a/g.pdf" requires accessing 4 nodes: "/", "/home", "/home/a", and "/home/a/g.pdf" in order. One kind of partition scheme in Fig. 1a, e.g., subtree partition [10], is POSIX-compliant, which keeps the metadata locality well since all 4 nodes are stored in the same MDS, thereby reducing the hop-count between MDS's significantly. However, it may suffer from serious accessing unbalance problem since MDS #1 with a popular namespace subtree could be overloaded under burst access. The other scheme shown in Fig. 1b, e.g., hash-based mapping [11], [12] which distributes the metadata among MDS's uniformly, can balance the system workload well, but it may have poor metadata locality. E.g, it needs 3 "switches" between MDS's when we access "h1.tex".

In general, hash-based mapping [11], [12], static subtree partition [10], [13] and dynamic subtree partition [14], along with other novel designs like Mantle [15], LazyHybrid [11], IndexFS [16], and DROP [17] are major ways to separate metadata into different servers. However, most literature simply design an architecture and some algorithms but fail to give a uniform definition or description on the distributed metadata management problem, let alone a trade-off research among metadata hierarchy locality and load balancing among MDS's brought from the distributed manner.

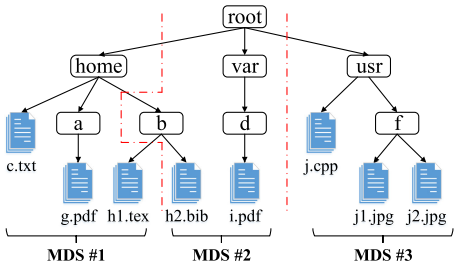
In this paper, we first give concise definitions of both locality and load balancing degree to interpret the measurement

- *Y. Gao, X. Gao, J. Liu, and G. Chen are with the Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, P.R.China. E-mail: {gyuanning, liujiaxi}@sjtu.edu.cn, {gao-xf, gchen}@cs.sjtu.edu.cn.*
- *X. Yang is with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, P.R.China. E-mail: yangxc@mail.neu.edu.cn.*

Manuscript received 7 Feb. 2018; revised 12 Feb. 2019; accepted 20 Feb. 2019. Date of publication 26 Feb. 2019; date of current version 7 Aug. 2019. (Corresponding author: Xiaofeng Gao.) Recommended for acceptance by T. Kosar. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2019.2901883



(a) Scheme with good metadata locality



(b) Scheme with good access load balancing

Fig. 1. Two typical schemes of partitioning metadata to MDS's.

of the system. Then, we present a novel scheme called AngleCut to partition metadata namespace tree and serve large-scale distributed storage systems. Our scheme first uses a novel locality preserving hashing (LPH) function to make a ring-projection and angle-assignment on the namespace tree and adopts multiple Chord-like rings as the keyspace (also called *identifier* space). The LPH function preserves the relative location of nodes from metadata namespace tree to a linear keyspace, which essentially keeps the metadata locality. Then, we design a novel history-based allocation strategy to allocate metadata uniformly to MDS's and adjust the workload dynamically. For each MDS, it adopts a sampling method based on a random walk [18] on metadata nodes to estimate *cumulative distribution function* (CDF) of the access frequency and then dynamically adjusts the workload. The strategy maintains good load balancing between MDS's and also keeps the metadata locality, which is compatible with the LPH design. Besides, we design a two-layer metadata cache mechanism to improve the metadata query efficiency, including client-side cache and server-side cache. Moreover, we propose a 2PC Protocol Based on Message Queue (2PC-MQ), which can guarantee the data consistency of distributed metadata transactions effectively. We conduct a series of experiments on Amazon EC2 to validate its efficiency. The comparison results exhibit the superiority of AngleCut over the previous literature.

Contributions. In this paper, we study how to design an efficient distributed metadata management scheme. We propose a novel locality preserving hashing for metadata management, which uses the angle of metadata node to partition the namespace tree. Such design is formally introduced in Section 4 and initially presented in [19].

This paper extends the initial study, via (i) surveying up to date literature and summarizing the comparison of metadata management schemes in Section 2; (ii) formulizing the migration cost in Section 3, which is affected by the quantity of the metadata nodes to be immigrated and emigrated as well as the system throughput; (iii) describing the system architecture in Section 4.1, which uses DHT [20] to partition the hash space; (iv) designing a two-layer metadata cache mechanism, including the client-side cache and the server-side cache, in Section 5.1, to improve the metadata query efficiency; (v) proposing a consistent processing protocol called 2PC-MQ in Section 5.2 to efficiently execute distributed metadata transaction; (vi) performing a more comprehensive experimental study on real datasets on Amazon EC2 platform in Section 7; and (vii) improving the organization and presentation of the paper by a major revision and careful proofreading.

2 Related Work

In this section, we first introduce the related work about metadata management policy, including *centralized* strategy and *distributed* strategy, which is classified by the number of MDS's. Then we introduce the metadata distribution schemes, i.e., *Hash-Based Mapping* and *Subtree Partition* as well as many other novel schemes. Finally, we introduce some core concepts of distributed hash table.

2.1 Metadata Management Policy

2.1.1 Single Metadata Server

Centralized strategy adopts single MDS to manage metadata, which simplifies the design and facilitates the performance of the distributed file system. Some commercial file systems such as Lustre [23], GFS [25], and HDFS [22] use this strategy. Nonetheless, single MDS tends to suffer from the bottleneck with growing size of file system, since metadata operations are highly frequent, causing overload of corresponding MDS. Moreover, there exists single-point failure because of the single MDS. When the only metadata server crushed, entire file system may break down.

2.1.2 Multiple Metadata Servers

Distributed strategy [14], [16], [21] adopts multiple MDS's to manage metadata, which partitions the namespace into small-size units and then maps these units to corresponding MDS's. Obviously, this architecture eliminates the single-point failure and bottleneck caused by centralized strategy, and enhances scalable performance of overall system. In spite of these advantages, distributed strategy may face several serious problems such as data consistency when it comes to multiple replications, system availability, data concurrency, etc. Each of these issues should be taken into consideration when designing an efficient file system.

2.2 Metadata Distribution Policy

Hash-Based Mapping and *Subtree Partition* are mainstream metadata partition schemes. Table 1 summarizes the comparison of different metadata distribution schemes, which will be discussed in detail in the following section.

2.2.1 Hash-Based Mapping

Hash-Based Mapping usually maps a file's pathname or other identifiers into some hash keys and distributes the metadata to MDS's by the projection from keys of metadata to keys of MDS's. zFS [24] and CalvinFS [12] adopt this idea. Hash-based strategy distributes the file metadata among MDS's uniformly, so this scheme can balance the system workload

TABLE 1
Comparison of Metadata Distribution Schemes

Scheme	Distribution Policy	Locality	Load Balancing	Hotspot	Query Time	Namespace Modifying	Add/Del MDS's
Ceph [21]	Dynamic Subtree	Good	Good. Migrate subtree among MDS's dynamically	Yes	$O(\log d)$	Fast. Locality is relatively good	Migrate subtree/dir items automatically by Monitor Node
HDFS [22]	Static Subtree	Good	Poor. Administrator adjusts workload manually	Yes	$O(\log d)$	Fast. Locality is good	Migrate subtree/dir manually
GIGA+ [14]	Directory Subtree	Good	Good. Split large directory into several MDS's automatically	Yes	$O(\log d)$	Fast. Most directories would not split because 90% directories are small	Depend on the underlying file system
Lustre [23]	Static Subtree	Good	Poor. Administrator adjusts load manually	Yes	$O(\log d)$	Fast. Locality is good	Migrate subtree/dir manually
Lazy Hybrid [11]	Subtree-Hashing Hybrid	Medium	Good. File metadata distribute uniformly	No	$O(1)$	Medium. "Lazy update mechanism" to avoid network congestion	Lazy migration to reduce network load
DROP [17]	Locality Preserving Hash	Medium	Good. HDLB-based load balancing strategy	No	$O(1)$	Medium. LPH makes locality relatively good	Costly
IndexFS [16]	Directory Subtree	Good	Good. "Power of Two choices" allocation strategy	Yes	$O(\log d)$	Fast. Use GIGA+ underneath	Equal to GIGA+
zFS [24]	Pure Hash	Poor	Good. File metadata distribute uniformly	No	$O(1)$	Poor. Multiple lookup hops between MDS's	Costly
Dir-Grain [1]	Directory Subtree	Good	Medium. Depends on the given values of the $\langle D, D, F\rangle$	Yes	$O(\log d)$	Fast. Locality is relatively good	Migrate directories dynamically

well, thus avoiding the MDS to become a "Hotspot" effectively, e.g., zFS in Table 1. Moreover, in the absence of the POSIX's path travel, the query time is $O(1)$ due to the mechanism of hash function. However, the hierarchical directory structure used to support directory operations for POSIX-compliant purpose is severely broken, since file metadata are scattered among MDS's, causing the poor metadata locality. Sometimes it will be even worse when part of MDS's join or leave the MDS cluster, resulting in all the hash values to be recalculated. Consequently, "Namespace Modifying" and "Add/Del MDS's" operations with respect to hash-based strategy in Table 1 are costly.

2.2.2 Subtree Partition

Different with hash-based mapping, subtree partition preserves the structure of namespace tree. Correspondingly, "Namespace Modifying" operations are fast, as shown in Table 1. The query time of subtree partition is $O(\log d)$, where d denotes the number of partitioned sub-directories.

Static Subtree Partition. Subtree partition includes static and dynamic scenarios. The conventional technique is to partition the global namespace into several subtrees and each MDS takes charge of one or some of them. Some distributed file systems such as Lustre [23], CODA [26], and CXFS [4] follow this approach. As shown in Table 1, it provides better locality and greater MDS independence than hash-based mapping, while the workload may not be evenly partitioned among MDS's, which will incur "Hotspot" easily. Besides, with the growth or contraction of individual subtrees over time, it requires intervention of system administrators to repartition or manually re-balance metadata storage load across MDS's.

Dynamic Subtree Partition. Dynamic subtree partition is an optimization of static manner. File systems like Ceph [21] and Kosha [6] use this scheme. The key idea is that a directory hierarchy subtree can be subdivided and delegated to different MDS's and the metadata workload will be dynamically redistributed along with the variations of workloads, e.g., Ceph in Table 1. This approach requires an accurate load measurement method and all servers need to periodically exchange the load information.

2.2.3 Other Schemes

Among other state-of-the-art schemes of metadata management, Dynamic Dir-Grain [1] observes that the partition granularity of static subtree partition and dynamic subtree partition may be too large. It proposes a triple $\langle D, D, F\rangle$ to determine the maximum granularity, and thus "Load Balancing" of Dir-Grain in Table 1 depends on the values of $\langle D, D, F\rangle$. DROP [17] is an efficient and scalable distributed scheme to serve EB-scale file systems. It exploits an LPH to distribute metadata among MDS's and adopts HDLB strategy to quickly adjust the metadata distribution. The "Locality" of DROP in Table 1 is relatively better than pure hash-based scheme owing to the LPH. However, DROP requires much extra space since it preserves many bytes to store an augmented hash key. Also, it does not provide the analysis on its locality and balancing features. As for some other schemes in Table 1, GIGA + [14] dynamically splits the large directory into multiple MDS's to avoid "Hotspot". IndexFS [16] uses GIGA+ as its underlying storage system and adopts "Power of Two choices" allocation strategy.

2.3 Distributed Hash Table

In peer-to-peer (P2P) systems, a distributed hash table (DHT) is a class of a decentralized distributed data structure

TABLE 2
List of Notations

Notation	Definition
N	Number of metadata items
T	Set of all metadata nodes
n_i	The ith metadata node
J_i	Hop distance of n_i
f_i	Access frequency of n_i
d_i	Depth of n_i from the root of namespace tree
M	Number of MDS's
L_i	Current load of the ith MDS
C_i	Maximum capacity of the ith MDS
μ	Global ideal load factor
I_i	Ideal load of the ith MDS
R_i	Remaining capacity of the ith MDS
P_i	Normalizations of throughput of the ith MDS
c_j	The jth component of n_i's coordinate
In_i	Quantity of immigrated metadata nodes of the ith MDS
Out_i	Quantity of emigrated metadata nodes of the ith MDS

that provides a lookup service. Chord [27] is one of the most classic DHT protocol design which adopts a variant of consistent hashing to assign keys to Chord nodes. It applies an m-bit identifier ring as the hash key space, which improves the scalability and enhances the performance of the system. In contrast to evenly partitioning all files and thus distribute the keyspace to the corresponding participating peer, Chord suggests the use of virtual servers to balance the loads of the peers further. A virtual server is actually a replica of the physical server in the hash key space. A physical server corresponds to some virtual servers which are distributed among the hash key space.

The LPH techniques are sometimes adopted in DHT [20] since they preserve the neighborhood structure of dataset, i.e., ensure that close keys are assigned to close objects. In general, AngleCut is inspired by several designs in DHT. It is essentially an LPH and adopts a variant of *identifier* ring in Chord as the keyspace.

3 Problem Formulation

In this section, we present precise definitions of the distributed metadata management problem in a formal way. We first give a clear mathematical definition of the metadata locality, load balancing degree between MDS's and migration cost to interpret the measurement of the system. Next, we present the goal of our optimization problem. Table 2 lists the symbols and their definitions.

Locality. Given a typical metadata namespace tree with N metadata items, let $T = \{n_i \mid 0 \le i \le N\}$ denote the set of all metadata nodes in which n_0 is the root. We define *Jumps* J_i of a metadata node n_i as the number of "switches"(i.e., the hop distance) between MDS's during the recursive search from top to bottom on the tree. Considering that different metadata nodes have different popularity and contribute to the total *Jumps* distinctively, we use access frequency f_i to describe the popularity of n_i. Notably, the access of each metadata node may come from the dedicated access to itself or the process of accessing its descendants. Now, we give the definition of the *locality* to describe the aggregation degree of the metadata stored in several MDS's.

Definition 1. *Define Loc as the global locality value of the whole system, which can be deduced from*

$$Loc = 1 \Big/ \sum_{i=1}^{N} J_i \cdot f_i. \tag{1}$$

Good *locality* means that "close" metadata nodes are more likely to be assigned to a same MDS, which reduces the response delay and improves the query efficiency (The definition of "close" will be discussed in Section 6.1). In our algebraic expression, large Loc in Eqn. (1) means the *locality* is good. Loc is $+\infty$ under single MDS scenario intuitively.

Load Balancing Degree. Earlier studies like [2] have proposed several load balancing algorithms, targeting static, small-scale and/or homogeneous environments. We now provide a concise review of them. Assume there are M MDS's, let $L_i, 1 \le i \le M$ be the current load, i.e., the sum of access frequencies, of the ith MDS and C_i be the maximum capacity. Define the global ideal load factor μ as $\mu = \frac{\sum_{i=1}^{M} L_i}{\sum_{i=1}^{M} C_i}$, then we can compute the ideal load I_i of the ith MDS with $I_i = \mu C_i$. Correspondingly, the remaining capacity is $R_i = L_i - I_i$. If $R_i > 0$, it means that the ith MDS is light. Otherwise, it is heavily loaded.

Considering the load of each MDS as *sample* in *Statistics*, the *load balancing degree* of the whole system is defined like the inverse of the *sample variance* of the load of all MDS's, which is consistent with the literal meaning of "balance".

Definition 2. *The* load balancing degree *of all MDS's is defined as*

$$Bal = 1 \Big/ \frac{1}{M-1} \sum_{i=1}^{M} \left(\frac{L_i}{C_i} - \mu \right)^2 = \frac{M-1}{\sum_{i=1}^{M} \left(\frac{L_i}{C_i} - \mu \right)^2}. \tag{2}$$

Good *load balancing degree* means that every MDS's current load is close to its ideal load, i.e., Bal in Eqn. (2) is large.

Migration Cost. As the access frequency of the system varies with time, we may need to migrate metadata between MDS's to keep a balanced workload. In this paper, we simply define Mig_i as the migration cost of the ith MDS involved in a dynamic adjustment without considering the topological structure of the MDS cluster. Define In_i and Out_i as the quantity of the metadata nodes to be immigrated and emigrated, respectively. Let P_i be the normalizations of the throughput of the ith MDS's.

Definition 3. *Define Mig as the migration measure of the whole system*

$$Mig = 1 \Big/ \sum_{i=1}^{M} Mig_k = 1 \Big/ \sum_{i=1}^{M} (In_i + Out_i) \cdot P_i. \tag{3}$$

The migration on MDS with high throughput will cause larger cost. Good migration means there are fewer metadata nodes to be moved on busy MDS's, i.e., Mig in Eqn. (3) is large.

Problem Statement. To keep the metadata locality as well as load balancing of the whole system, our goal is to improve the *locality* and the *load balancing degree* while partitioning the metadata tree. Since excessive migration may

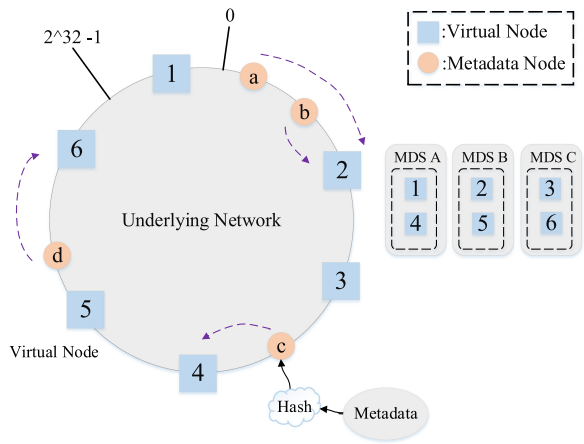


Fig. 2. System architecture.

cause huge update cost, the *migration cost* is also worth to be considered. Thus, we define the problem as an optimization problem. We set three coefficients λ, μ, and γ on *locality*, *load balancing degree*, and *global migration measure* and a constraint σ on *migration cost*. Then, the goal is to maximize a linear combination of Loc, Bal and Mig.

$$\max \quad \lambda Loc + \mu Bal + \gamma Mig \tag{4}$$

$$s.t. \bigcup_{i=0}^{N} n_i = T, \tag{5}$$

$$Mig_k \leqslant \sigma, \forall 1 \leq k \leq M. \tag{6}$$

Equation (5) means MDS's should contain all nodes while Equation (6) shows the constraint on *migration cost*.

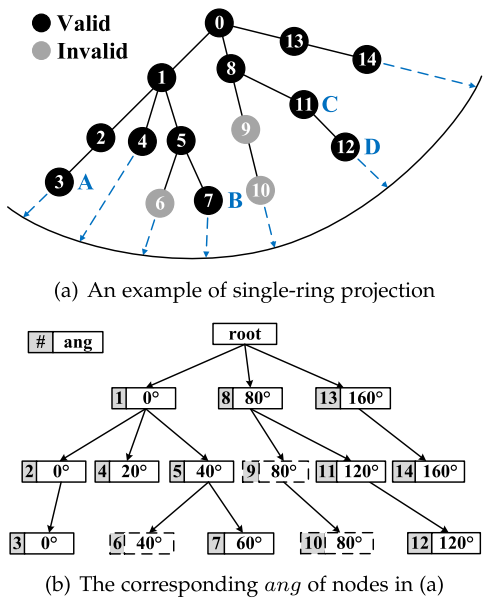
4 The AngleCut Scheme

In this section, we present a detailed description of AngleCut scheme. We first describe the system architecture in Section 4.1. Then we present the hashing design of AngleCut in Section 4.2 and Section 4.3, including the LPH design and the dynamic allocation strategy. In Section 4.4, we introduce the history-based allocation strategy which adjusts the workload of MDS's dynamically.

4.1 System Architecture

As shown in Fig. 2, the whole hash space, e.g., $0 - 2^{32}$, is organized as a hash ring. Each metadata node is mapped into a 32-bit value by a specific hash function, which represents a position on the hash ring, e.g., a, b, c, and d in Fig. 2. Similarly, each MDS is also mapped to a hash value standing for its position on the ring, e.g., server_1 to server_6 in Fig. 2, by the identical hash function. According to consistent hash algorithm [27], the metadata is allocated to the nearest MDS on the hash ring in the clockwise direction.

To better balance the system load, we adopt virtual nodes to substitute the real MDS's. Each MDS manages multiple virtual nodes, which are scattered on the hash ring acting as virtual servers. In Fig. 2, server_1 to server_6 are actually virtual nodes, managed respectively by the real MDS's A, B, and C. The metadata nodes allocated to these virtual nodes will be distributed to the host MDS in the end. Consequently, the load of the specific MDS is determined by the sum of its virtual nodes. Adopting virtual nodes not only improves the system load balancing degree, but also contributes to the system scalability. For example, when one MDS is added or deleted, the virtual nodes residing in it can be smoothly migrated to neighboring MDS's.



(a) An example of single-ring projection

(b) The corresponding ang of nodes in (a)

Fig. 3. An example illustrating the hashing design on calculating the ang.

4.2 The Hashing Design

As mentioned above, AngleCut uses a novel LPH function to hash the metadata namespace tree onto a linear keyspace. At first, we calculate each node's angle according to their corresponding $coordinate$ in namespace tree. The $coordinate$ of metadata is defined in Definition 4. Then Anglecut projects the metadata nodes to multiple Chord-like rings by their angle value. For simplicity, we assume that the hash space is $360°$. Fig. 3a illustrates an example of projection in AngleCut from a simple metadata tree to a single ring. The tree is composed of 4 layers and we define node n_0 as the 0th layer. For nodes except the root, they are projected to the ring in a left-hand depth-first-search order, in which "invalid" means the nodes may have been deleted. Notably, node C and D are projected to the same location as they have the same angle. In Fig. 3b, we present more information on the angle of the corresponding nodes in Fig 3a, e.g., the angle of both node C and D is $120°$.

In our hashing design, the hash key of a metadata node n_i is represented by two key components: tag and ang. The $n_i.tag$ is used to store some identification information of n_i while the $n_i.ang$ is used to store its angle. We now give a detailed interpretation of the design.

tag: The tag is a 1 byte key including effective bit, the order number ord of the ring and some reserved bits. The effective bit indicates whether the metadata node is valid, e.g., the node is invalid after being deleted. The order number ord is used to show which ring the metadata node locates on. At present we take the single ring scenario as an example, which means all metadata nodes have $ord = 0$. The multiple rings design is relevant to the update of the system, which will be introduced in Section 4.3.

ang: ang is the kernel component of the hash key, which is closely related to the *locality*. Given a metadata tree and a ring, the ang of one metadata node is the angle of the projective node on the ring, ranging in $[0°, 360°)$. To calculate the ang, we first give a definition of $coordinate$ to represent the "location" of node n_i in a metadata tree.

Definition 4. *Given a metadata tree, the* $coordinate$ *of a non-root node* n_i *is* $(c_1, c_2, \ldots, c_{d_i})$. *The dimension* d_i *of the* $coordinate$ *is the depth of* n_i *from the root and* $c_j, 1 \leq j \leq d_i$ *represents the number of left siblings of* n_i *on the* j*th layer in counter-clockwise direction.*

For example, as shown in Fig. 3a, the $coordinates$ of nodes A, B, C, and D are $(0, 0, 0)$, $(0, 2, 1)$, $(1, 1)$ and $(1, 1, 0)$ respectively.

With the $coordinate$ of metadata node n_i, we are ready to calculate its angle $n_i.ang$. First, all nodes on the 1-st layer share the $360°$ angle uniformly. We denote the ang difference between any two adjacent nodes on the 1-st layer by θ. For example, in Fig. 3b, we have $n_8.ang - n_1.ang = \theta = 80°$. Second, for children nodes on the 2-nd layer, all nodes on the rightmost location such as node n_5 and n_{11} own an ang increment of $\theta/2$ while the other nodes share the $\theta/2$ angle range uniformly since they are treated equally, e.g., $n_{11}.ang = 120°$ and $n_5.ang = 40°$. Similarly, the ang of the rightmost node in the 3-rd layer increases by $\theta/4$ and its left siblings share the $\theta/4$ angle range uniformly. We continue this process until all nodes are assigned with the ang key. The angle-assignment is executed inductively and the details are shown in Algorithm 1: the $coordinate$ of each metadata node is calculated in Lines $1-4$. Then we calculate the value of $n_i.ang$ in Lines $6-10$ and handle the hash collision in Lines 8-10.

Algorithm 1. Cycle-Projection

Input: A namespace Tree
Output: The ang keyspace
1 **for** $i \in [0, N]$ **do**
2 set n_i valid;
3 **if** $i > 0$ **then** Get the $coordinate$ of n_i;
4 $C_{n_i} = \#$ of children nodes of n_i;
5 $\theta = 360/C_{n_0}$;
6 **for** $i \in [1, N]$ **do**
7 $n_i.ang = \sum_{j=1}^{d_i} \frac{c_j\theta}{(C_{n_i.ancestor}-1)2^{j-1}}$;
8 **if** $n_i.ang$ *conflicts with existing rings'* **then**
9 Create a new ring r;
10 $n_{i.tag.ord} = r.ord$;
11 add n_i to the rings;

The angle-assignment of our hashing design can keep the *locality* of the namespace tree. Note that from the 2-rd layer to bottom, the largest ang increments of rightmost children nodes are $\theta/2$, $\theta/4$, $\theta/8\ldots$ respectively. Now, for one node n_i on the kth layer with $ang = \delta$, the ang of its any child node δ_c satisfies $\delta_c < \delta + \theta/2^k + \theta/2^{k+1} + \theta/2^{k+2} + \ldots < \delta + \theta/2^{k-1}$, $k \geq 1$. It means that any child node's ang will not go beyond the ang of the adjacent right sibling of n_0. This is how the AngleCut scheme keeps the *locality*. We will give a formal and detailed proof in Section 6.1.

The angle is a non-unique identifier (hash value) of a metadata node. Introducing the angle helps to keep a good locality and load balancing of the whole systems because such an LPH-based method breaks up the metadata allocation into two separate stages. The first stage is projecting the metadata nodes into a ring hash space in a LPH way, which essentially keeps the locality of metadata namespace tree. With the ring hash space, in the second stage it is more convenient to conduct the subsequent metadata allocations. However, as to more general scenarios where the access to the metadata tree is unbalanced and skewed, it will fail to keep the locality well.

4.3 The Multiple Ring Design

The aforementioned hashing design illustrates the angle-assignment of a given metadata namespace tree, which deals well with the static scenario with only read and write requests. However, when meeting dynamic adjustment transactions such as "insert" request on a metadata node, there may exist no reserved angle on the corresponding layer, i.e., the angle for the new node may be occupied by the nodes on the existing rings, which causes the hash collision. To deal with this, we propose to use multiple rings as hash keyspace, i.e., use an identifier ord which is stored in tag to denote which ring the metadata node locates on. Consider "insert" request on a new metadata node n_i, AngleCut checks the existing rings to make sure whether there is a reserved ang. If not, AngleCut would create a new ring to place it, i.e., set a new value to the ord of n_i. The ang are computed still in the same way, which means that the $coordinate$ and $n_i.ang$ are the same as some nodes in other rings. The ord is the only identifier to distinguish them. Note that the multiple ring design does not exhibit additional overhead when querying the position of a metadata node, since AngleCut only checks the ang value and find which MDS the node belongs to.

In addition to alleviating the hash collisions, the multiple ring design is also relevant to the update and maintenance of the whole system. First, for angle range with frequent insertions or deletions, the identifier ord can identify which nodes are newly inserted and which are set invalid. This plays a role of timestamp in update process and can also help restore deleted metadata when needed, which improves the robustness in distributed file systems. Second, multiple rings add another dimension to distinguish and manage metadata. It provides a reserved management choice, e.g., allocating metadata hash space to one or more MDSs according to the ring number.

4.4 Metadata Allocation

Since we have obtained multiple rings as the keyspace, the next step is to allocate metadata nodes uniformly to the M MDS's, which is an NP-hard problem reduced from the 0-1 Knapsack problem [17]. Here MDS's can be regarded as virtual servers. We now propose a novel history-based allocation strategy to allocate metadata uniformly to MDS's and adjust the workload dynamically.

In order to allocate the metadata nodes, an essential step is to get the distribution of metadata access frequency, while it is impractical to leverage all metadata nodes to simulate the *cumulative distribution function* since the quantity is considerably huge. Consider that we sample the metadata nodes in the namespace tree, random walk sampling method, which

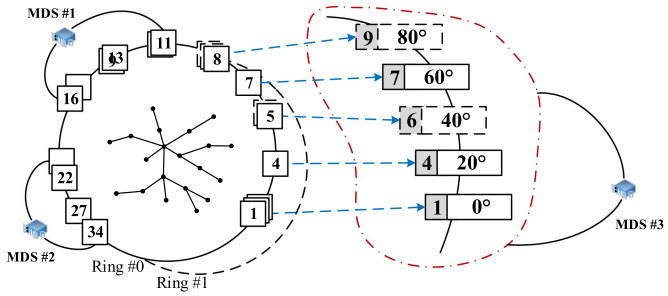


Fig. 4. An illustration for metadata allocation in AngleCut.

has been widely used in distributed networks [28], can appropriately accommodate the requirement. AngleCut adopts a sampling strategy based on a random walk [18] on metadata nodes to estimate a temporal CDF of the access frequency with respect to ang. Notably, the temporal CDF can only indicate the accessing patterns in the recent past. With this strategy, we are able to divide the angle range $[0, 360°)$ to M arcs in an approximately uniform manner and derive $M-1$ temporary angle range boundaries $\psi_1, \ldots, \psi_{M-1}$. Fig. 4 illustrates the metadata allocation strategy in which the projective nodes of metadata located on the rings are allocated to 3 MDS's. It also includes the angle-assignment which is consistent with Fig. 3 and the multiple rings design mentioned in Section 4.3. For example, in Fig. 4, assume that Ring $\#1$ managed by MDS $\#3$ covers the range from $0°$ to $100°$. Then all the metadata nodes that are mapped into $0°$, $20°$, $40°$, $60°$, or $80°$ will be allocated to MDS $\#3$.

As the access frequencies of nodes change over time, the workload of MDS's may become unbalanced. Once over a period of time, the MDS's need to renew the sampling and update the CDF estimation of access frequency. In our strategy, each MDS also keeps a history record of optimal CDF, which indicates the optimal accessing patterns in each period of accessing history. Similarly, we calculate the optimal angle range boundaries $\phi_1, \ldots, \phi_{M-1}$ based on the optimal CDF. To avoid the abrupt change of boundaries and match the subsequent accessing patterns, in each update process we set the optimal boundaries ϕ_i as a weighted sum of previous optimal boundaries and temporary boundaries by assigning weight $\alpha \in (0, 1)$, i.e., $\phi_i = \alpha \cdot \phi_i + (1-\alpha) \cdot \psi_i$, $\forall i \in [1, M-1]$.

There are many factors like similarities between recent accessing patterns of MDS's that affect α. Though it is better to determine α by a Deterministic Finite Automaton (DFA), we keep it as a simple parameter to avoid redundant records and calculation of the weight between several factors. In this way, α increases monotonically since the optimal CDF includes more and more information about the accessing patterns of the system. However, it should have an upper bound since the temporal CDF can indicate the future accessing trend, i.e., it should always get a certain weight. The upper bound is not easy to be theoretically determined. Thus, we prefer to adjust it in experiments.

5 Cache Management and 2PC-MQ

To improve the metadata query efficiency, we introduce a two-layer metadata cache mechanism in Section 5.1, including client-side cache and server-side cache, which provides the two stage access acceleration. Then in Section 5.2, we propose the 2PC-MQ, a distributed transaction scheme, which effectively guarantees the data consistency of distributed metadata transactions.

5.1 The Two-Layer Cache

POSIX-style standard requires many metadata operations to perform pathname traversal and permission checking across each ancestor node [12]. This accessing pattern is not well balanced across MDS's because metadata nodes near the top of the namespace tree are accessed more frequently than those at the lower part of the tree. The other disadvantage is that it may bring repetitive *Jumps* between MDS's and thus reduce the *Loc*. To improve query efficiency of metadata, we propose a two-layer metadata cache mechanism, including client-side cache (the first layer) and server-side cache (the second layer) to provide a two stage acceleration.

5.1.1 The First Layer Cache Issue

Each client will save the hot metadata in its local storage, which is *the first layer cache*. However, the cache consistency between multi-clients and MDS's is difficult to achieve due to concurrent access. During the period of metadata operations, multi-clients may access the same directory or file. For example, if client A writes to the file while client B is reading the same file from the local cache, then the data that client B gets is not up-to-date.

AngleCut adopts *lease* mechanism to maintain a consistent client cache of pathname components and their permissions. Once a client intends to cache metadata items, it first applies the caching authority to corresponding MDS. In the absence of writing confliction, MDS will assign a short term lease to the client and delay any operations that want to modify the metadata items until the largest lease expires. Thus, the client is able to obtain the metadata directly from local cache when performing path traversal and permission checking instead of communicating with MDS's, which significantly decreases the response latency. Corresponding cache will become invalid after the lease expires, and then the client needs to reapply to the MDS. While the lease mechanism may incur high latency of the mutation operations, client latency for non-mutation operations and network congestion are greatly alleviated. Furthermore, according to HopsFS [29], total read operations make up 95 percent in a workload generated by Hadoop operations while update operations for HDFS such as *create*, *delete*, *move* only occupy little proportion. Hence, the lease mechanism is efficient.

5.1.2 The Second Layer Cache Issue

As mentioned above, the metadata nodes near the top of the namespace tree are accessed more frequently due to the path traversal operations. For this reason, each MDS replicates the metadata nodes near the top of the namespace tree in memory, which is *the second layer cache*. Besides, each MDS may cache the metadata that do not belong to it when performing pathname traversal. Similarly, we adopt the same time-based lease mechanism introduced above to maintain the consistency of the cache data.

The simplest policy to set lease duration is to use a fixed time interval (e.g., 300 ms) for each lease. However, several

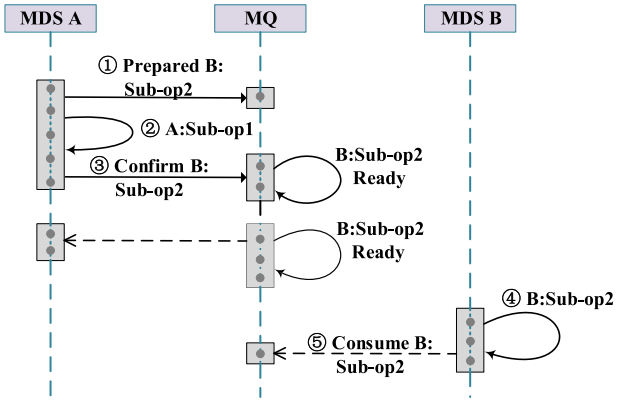


Fig. 5. Sequence diagram of 2PC-MQ.

factors should be well considered to set lease. For instance, metadata nodes that are at the top of the namespace tree as well as those that are frequently accessed are supposed to enjoy longer lease duration, rather than extending lease hourly. Different with IndexFS [16], AngleCut uses $(\eta \cdot f_i + \xi \cdot d_i) \cdot L$ to calculate time interval for the cached entries, where f_i hints *Access frequency* of the ith metadata item, and d_i represents the depth of metadata node in the namespace tree. $L = 3s$ in default. To increase the flexibility for different application environment, the value of η, ξ and L can be slightly adjusted by configuration file. Besides, this approach is based on global clock synchronization, which can be achieved by the clock synchronization algorithm with clock drift compensation [30], i.e., the clock synchronization process needs to correct the time error caused by the drift of different servers' clock counters.

5.2 Distributed Metadata Processing

Distributed metadata processing usually involves multiple servers participating in the same transaction. For instance, a simple "create" operation which creates a new file *file1* under a directory *dir1* may cause inconsistency. Assume *dir1* and *file1* are maintained by two different servers. A sub-operation (Sub-op1) from the first MDS allocates a file inode [31] and sets the flag to indicate that it is a regular file. Meanwhile, another sub-operation (Sub-op2) from the second MDS inserts a new entry in the parent directory *dir1* and updates parent inode. During the process, if one of MDS's crashes, the updated information may be lost, leading to inconsistency of the file metadata.

Two-phase Commit (2PC) [32] is a widely used atomic commitment protocol in transaction processing. However, to complete the entire process, there are too many network communications between coordinator and participants, which causes severe network latency. Moreover, if one participant crashes, the procedure will be blocked. Thus, it is not suitable for high concurrency applications. To solve this problem, we design the 2PC-MQ, which is an efficient mechanism based on message queue (MQ) technology [33].

Fig. 5 depicts the framework of 2PC-MQ: assume MDS A and B participate in one distributed metadata transaction, in which MDS A is the coordinator and B is the participant, respectively. At Step ①, MDS A first sends the "prepare" message to MQ, which indicates the information of Sub-op2 for MDS B. MQ will store the prepare message temporarily in the data structure of a queue and return "success" message. At Step ②, MDS A completes the Sub-op1 and sends the "commit" message to MQ at Step ③, which means that coordinator has successfully finished the Sub-op1; the participant is able to execute the remaining sub-operations. Next at Step ④, MDS B executes the local transaction after receiving the Sub-op2 message from MQ. In the end, the coordinator receives the "success" message by callback interface at Step ⑤. Consequently, the entire distributed metadata transaction is done.

The key idea of 2PC-MQ is to use MQ storing Sub-op2 temporarily. Sub-op2 will be sent to participant only when Sub-op1 succeeds, and this indicates that the coordinator has voted "yes". As a comparison to 2PC, the coordinator in 2PC-MQ does not have to be blocked to wait for Sub-op2 accomplishes. However, each procedure in Fig. 5 may fail in the entire message transaction. The details are as follows:

- If Step ① fails, the entire transaction will stop. Both Sub-op1 and Sub-op2 would not be executed. Correspondingly, the state of transaction keeps consistent.
- If Step ② fails, obviously, Sub-op1 will not be executed, which is the same as step ①.
- For Step ③, MQ will continuously run *callback interface* that is implemented by the coordinator to check whether Sub-op1 succeeds. If not, MQ will rollback the "prepare" message. Otherwise, MQ commits the message by itself, thereby accomplishing the entire message transaction.
- For Step ④ and ⑤, MQ is designed to support *message-retry* operation. If the participant fails to execute the Sub-op2, MQ will resend the message to the participant until it succeeds. In addition, the participant is supposed to record the operation that is executed successfully to avoid repetitive executions.

The above analysis demonstrates the efficiency and effectiveness of 2PC-MQ. However, there is only one MQ handing the distributed transactions. Once the MQ crashes, the whole system will break down. Besides, as the coordinator, single MQ may suffer from the performance bottleneck. In practical applications, we can employ the MQ cluster [34] to meet the requirements of high availability, reliability and high throughput. In most distributed metadata transactions, since there is only one participant [1], the process of 2PC-MQ can be efficiently finished. In addition, 2PC-MQ can guarantee the eventual consistency. By introducing small consistency penalty, this design can significantly bring the performance improvement for the applications that have no strong demand for data consistency.

6 Theoretical Analysis

6.1 Well-Definedness on Locality Preserving Hashing

Locality preserving hashing is a kind of hash function satisfying the property of preserving or keeping the *locality* as well as reducing the dimensionality of the input data. For three points A, B and C in a high dimensional metric space, an LPH function f needs to satisfy the condition: $|A - B| < |B - C| \Rightarrow |f(A) - f(B)| < |f(B) - f(C)|$.

Consider the previous example shown in Fig. 3a. Intuitively it seems that C and D have shorter "distance" than B

and D while the ang difference of C and D is smaller too. However, this has no theoretical basis. We now present a proof of well-definedness of AngleCut on LPH.

Theorem 1. *The hashing to ang in AngleCut is an LPH which maps the namespace tree space to linear space.*

Proof. First, we propose a definition of *distance* between any two nodes in a typical metadata namespace tree. □

Definition 5. *Consider two non-root metadata nodes n_1 and n_2 with respective coordinate denoted by $(x_1, x_2, \ldots, x_i)$ and $(y_1, y_2, \ldots, y_j)$, where i, j are the respective dimensions of coordinate. The* distance *between n_1 and n_2 is defined as follows:*

$$f_c(n_1, n_2) = \sum_{k=1}^{\min\{i,j\}} w_k \cdot |x_k - y_k|. \tag{7}$$

The setting of weight function w_k is because each coordinate component should have different weight since it represents "location" on different layers of the namespace tree. Typically, we define that the weight function is $w_k = 1/2^k$ here. In Fig. 3a, we have $f_c(B, D) = 7/8$ while $f_c(C, D) = 0$.

As to the definition of *distance* in ang keyspace, for nodes n_1, n_2, we define their distance $f_a(n_1, n_2)$ is just the difference of their ang. e.g., $f_a(B, D) = 60^\circ$ and $f_a(C, D) = 0^\circ$. Finally, we get the deduction

$$f_c(C, D) < f_c(B, D) \Rightarrow f_a(C, D) < f_a(B, D). \tag{8}$$

Without loss of generality, we choose nodes B, C and D as examples in the proof. In fact, for any two nodes in the namespace tree, it is not hard to see the hashing to ang does preserve the locality.

6.2 Space Complexity of AngleCut

Theorem 2. *For a metadata namespace tree with N non-root nodes, the total space to store hashing key is $17N$ bytes on overage.*

Proof. As mentioned in Section 4.2, first, each node uses one byte for tag to store the effective bit, ord and so on. Second, each node needs a *double* value to record its ang, adding up to $8N$ bytes for the whole system. Besides, to find the MDS's where the child nodes are stored, each node should also record the angle of its children nodes. Let $D_i = \{d_i^1, d_i^2, \ldots, d_i^{|D_i|}\}$ denote the children nodes set of node n_i, the space for storing D_i is $8D_i$ bytes then. However, as each node has exactly one parent node, the space for storing all children nodes is $8N$ bytes in total. To summarize, the total space to store hashing key is $17N$ bytes on overage. □

For DROP [17], it preserves $48N$ bytes in total for the same namespace tree in order to append an augmented hash key to each node. It indicates that AngleCut outperforms DROP a lot as to space complexity.

7 Performance Evaluation

In this section, we evaluate the performance of AngleCut using detailed simulation as an actual measurement. We implement AngleCut in C++ and evaluate it based on our distributed storage framework [35]. We use Amazon EC2 platform to run our experiments. Each EC2 instance is composed of Dual-Core Intel Xeon E5-2676 v3 Processors and 8 GB memory, SSD is used as the underlying disks. Besides, each instance runs Ubuntu 16.04 LTS operation system. Experimental scale of MDS's ranges from 5 to 30, with an increment of 5. We also implement and make comparisons with subtree partition [10], dynamic subtree partition [14], hash-based mapping [24], and DROP [17]. To better illustrate the rationality and efficiency of the experiment results, in addition to the core metadata partition mechanism, all of these schemes implement the features discussed in previous sections, including the two-layer metadata cache mechanism and 2PC-MQ. The dataset we use are three real-world traces[1] called **MSN**, **DevTool** and **LiveMap**. The detailed information is shown in Table 3.

TABLE 3
The Data Traces used in Experiments

Trace Source	Data Size	Records Count	Collection Duration
MSN Storage File Server	5.77 GB	26,657,063	6 hours
Development Tools Release	5.90 GB	17,952,090	24 hours
Live Maps Back End	15.1 GB	44,755,552	24 hours

Experiments are conducted in the following ways. For each scale of MDS's and specific dataset, we evaluate aggregated *throughput*, the namespace *locality*, the *load balancing degree* and the *migration cost* of AngleCut. Besides, we test the performance of AnguleCut under different type of cache mechanism. We interact with the running MDS's with one master MDS in a centralized manner, mainly in charge of: 1) the distribution of query tasks. 2) the generation of ang hash tables. For each experiment, the master MDS continuously distributes query tasks randomly over all MDS's. After each 5 percent of all query records in the traces, every MDS renews a sampling on metadata nodes stored in it to estimate a temporal CDF of access frequency and sends it to the master. The master updates the hash table and broadcasts it to all MDS's. The update process is executed 20 times in total.

7.1 Scale and Performance

As is shown in Fig. 6, we use $throughput$ to evaluate the performance of file system, namely, the number of request operations that the system is able to handle per second. In Fig. 6, when the number of MDS's is less than 15, subtree partition strategy gains higher throughput than DROP and AngleCut, because it keeps great metadata locality, thereby reducing the response latency effectively. However, with the increase of cluster scale, the performance of AngleCut improves gradually and surpasses the subtree partition eventually. The same situation also happens on DROP, since DROP and AngleCut have well-defined LPH function to keep excellent load balancing among the MDS's. Compared with DROP, AngleCut possesses better scalability owing to the metadata allocation strategy based on the angle of namespace tree

1. Data source: http://iotta.snia.org/traces/158

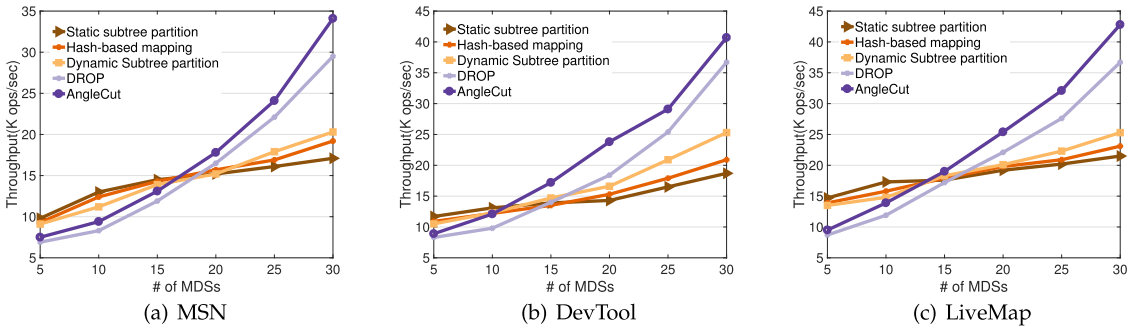


Fig. 6. The aggregate throughput under different schemes.

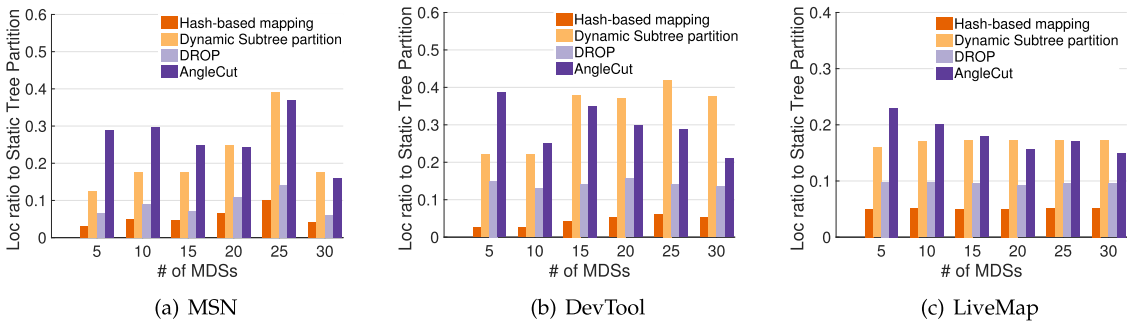


Fig. 7. The locality performance under different schemes.

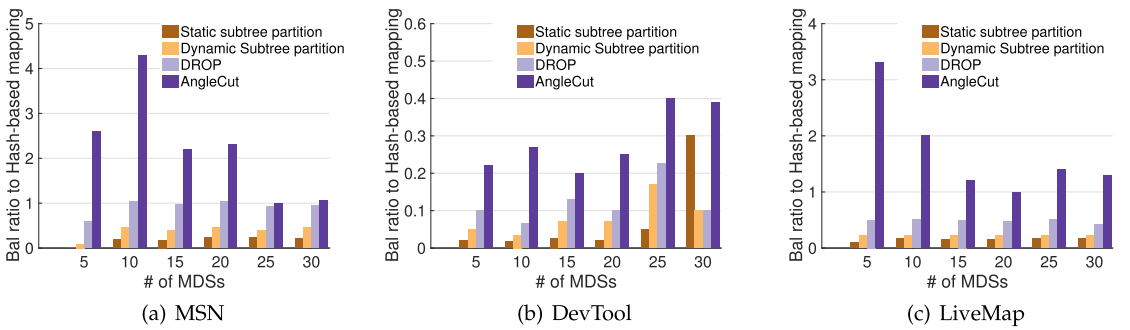


Fig. 8. The load balancing performance under different schemes.

node. Correspondingly, the *throughput* of AngleCut is superior to DROP, as shown in Fig. 6.

7.2 Metadata Locality

In this experiment, we use the *Loc* of static subtree partition scheme as a baseline as it is the ideal scenario for keeping high metadata locality. In Fig. 7, we present the *Loc* ratio from the other mechanisms mentioned above to static subtree partition on three traces. The results of varying M (the number of MDS's) show that AngleCut has great performance on *locality*, which is twice as good as DROP and three times better than the hash-based mapping for all the three traces. It has almost the same performance as dynamic subtree partition, especially when the number of MDS's is small. This is because when we have less MDS's, dynamic subtree partition distributes many small subtrees over MDS's, so the tree is more separated, while AngleCut cuts the tree into less parts. However, as the scale of MDS's increases, AngleCut breaks more edges of the tree, which adds to more "switches" between MDS's and the locality becomes worse then. In contrast, the small granularity in dynamic subtree partition shows its advantage.

7.3 Load Distribution

The adjustment of load balancing between MDS's is a dynamic process. In our experiment, the workload rebalance and boundary adjustment are executed 20 times in total. Similarly, we set the *Bal* of hash-based mapping as the baseline and present the ratio of *Bal* to it after the last rebalance in Fig. 8. The results show that the static subtree partition performs the worst while dynamic subtree partition is slightly better, since it adopts dynamic adjustment to make up for load imbalance. Both AngleCut and DROP have much better load balancing performance and AngleCut performs twice as good as DROP in most cases, sometimes even much better.

The results also show that the load balancing performance of hash-based mapping on **DevTool** trace is remarkably distinctive. It is different from the previous *locality* case in which the static subtree partition always has the best *locality*. The reason is that, in hashed-based mapping, all metadata items are just evenly assigned to each MDS without taking access frequencies into consideration. For example, in **MSN** and **LiveMap**, queries concentrate on one or two subtrees and the corresponding MDS's are heavily

TABLE 4
The Operation Percentage in Each Dataset

Trace Source	Create	Lookup	Update
MSN Storage File Server	8%	90%	2%
Development Tools Release	11%	82%	7%
Live Maps Back End	9%	83%	8%

loaded. Thus, AngleCut can even perform better than hashed-based mapping on these two traces.

7.4 Cache Policy

To evaluate two-layer metadata cache mechanism, we filter the three datasets that record the metadata operations issued the specific scenario. The detailed information is shown in Table 4. Based on the new trace data, we replay the workload in two stages. At first, we construct the namespace tree among the MDS's using multiple clients in parallel; each of clients takes charge of a portion of create operations. Second, 30 client threads are created to replay the trace concurrently. The trace data are divided into the blocks and assigned to client threads according to round-robin manner. Each data block contains 800 query items, which consists of lookup and update operations.

Fig. 9 shows the aggregated throughput of AngleCut at different cluster scales ranging from 5 to 30. We altogether measure 6 types of cache strategies that are combinations of client cache and server cache. "ClientFixCache" in Fig. 9 represents that we only set cache at client-side, and the lease time for cache is fixed (300 ms). On the contrary, "ClientDynamicCache" and "ServerCache" represents that the lease time is dynamically adjusted according to $(\eta \cdot f_i + \xi \cdot d_i) \cdot L$ as mentioned in Section 5.1. Obviously, AngleCut behaves differently under each cache mechanism. Without cache strategy, the system suffers poor scalability, having the lowest throughput among the others, because the performance is bottlenecked by too many RPCs between client and servers. Conversely, the system equipped with both server cache and client dynamic-lease cache acquires the highest throughput of more than 400,000 operations per second.

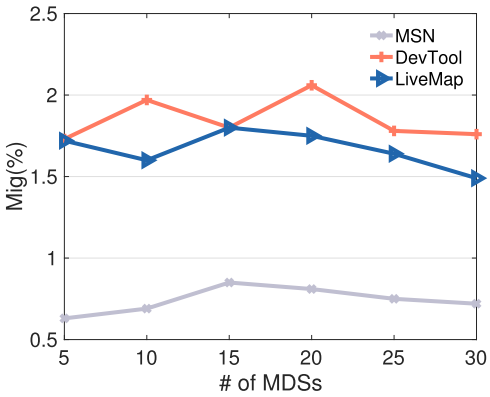
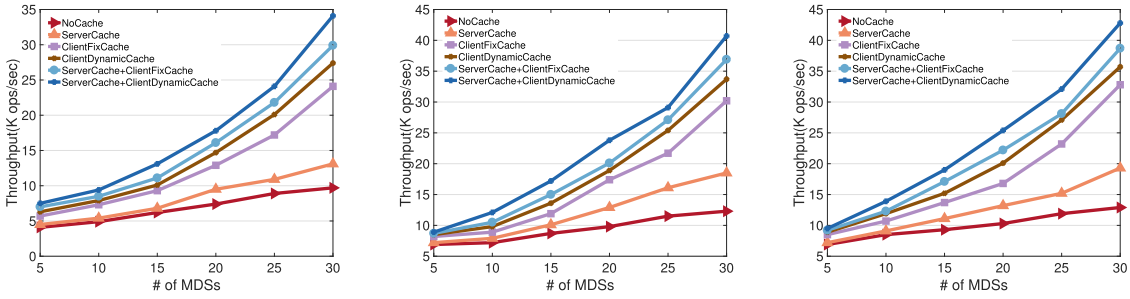


Fig. 10. Average migration cost.

The client dynamic-lease cache strategy is superior to fixed-lease strategy because it can set lease time at the client side according to the characteristic of specific metadata, thus providing more accurate predictions and improving the hit rate greatly. Besides, we can see that without client-side cache, the system suffers from evident performance degradation even when it has equipped with server-side cache, which owes to plenty of unnecessary RPCs between clients and servers increases the lookup latency badly.

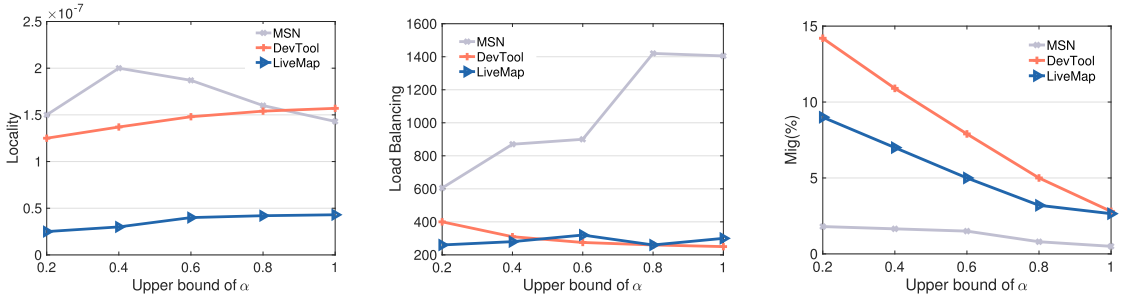
7.5 The Migration Cost

With the insertion or deletion of metadata nodes and the join or leaving of MDS's, AngleCut needs to migrate metadata between MDS's for maintaining a good *load balancing degree*.



(a) MSN (b) DevTool (c) LiveMap

Fig. 9. The aggregate throughput of AngleCut under different cache strategies.



(a) The effect of α on locality (b) The effect of α on load balancing (c) The effect of α on migration cost

Fig. 11. The aggregate throughput of AngleCut under different cache strategies.

Note that two factors may affect the migration cost Mig_k of the kth MDS. One is the scale of the trace, the other is the number of MDS's since more MDS's sharing the migration may decrease each Mig_k in general. Thus, we use the Mig which denotes the means of migration proportion, i.e., $\frac{\sum Mig_k}{\#\ of\ MDS's}$, to measure the performance. The result in Fig. 10 shows that less than 2 percent metadata items are migrated on average. The cost is quite low under a maintenance of high Bal. The Mig of **MSN** trace is relatively lower than others in that its accesses are quite evenly distributed over different subtrees.

7.6 Upper Bound of α

As mentioned in Section 4.4, α controls the proportion of optimal boundaries and temporal boundaries. We conduct the experiment with different upper bounds of α to evaluate its influence on *locality*, *load balancing degree* and *migration cost*. The number of MDS is set to 10, and the upper bound varies from 0.2 to 1, with an increment of 0.2. Note that setting upper bound to 1 means there is no upper bound to α. Under this scenario, the ang table will gradually become more and more stable. Figs. 11a, 11b and 11c show how *locality*, *load balancing degree* and *migration cost* change with respect to different upper bounds of α. The results show that neither *locality* nor *load balancing degree* has absolute positive correlation or negative correlation with the upper bound of α. Thus, it is hard to decide the best upper bound. Moreover, in the **MSN** and **DevTool** trace, the migration cost decreases rapidly as the upper bound increases. In general, the best upper bound of α depends on the accessing pattern of a data set, but a higher upper bound is usually better.

8 Conclusion

In this paper, we present AngleCut, an efficient and scalable metadata management to partition metadata namespace tree and serve EB-scale file systems. AngleCut first projects the namespace tree into multiple Chord-like rings using a novel ring-based locality preserving function. Then we design a well-designed history-based allocation strategy to allocate the metadata uniformly to MDS's. The two-layer metadata cache mechanism is proposed to improve the query efficiency and reduce the network overhead. Last but not least, we design an efficient distributed processing protocol called 2PC-MQ to guarantee the consistency of distributed metadata transactions. AngleCut preserves the metadata *locality* essentially as well as maintaining a high *load balancing degree* between MDS's. The theoretical proof and extensive experiments on Amazon EC2 exhibit the superiority of AngleCut over the previous literature.

Acknowledgments

This work was supported by the National Key R&D Program of China [2018YFB1004703]; the National Natural Science Foundation of China [61872238, 61672353, 61532021]; the Shanghai Science and Technology Fund [17510740200]; the Huawei Innovation Research Program [HO2018085286]; the State Key Laboratory of Air Traffic Management System and Technology [SKLATM20180X]; the Tencent Social Ads Rhino-Bird Focused Research Program; and the Fundamental Research Funds for the Central Universities [N171602003].

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 Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, pp. 803–816, May 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]. Available: http://dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf

[3] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proc. 10th ACM Symp. Operating Syst. Principles*, 1985, pp. 15–24.

[4] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM storage tank–A heterogeneous scalable san file system," *IBM Syst. J.*, vol. 42, no. 2, pp. 250–267, 2003.

[5] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger, "A transparently-scalable metadata service for the ursa minor storage system," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 13–13.

[6] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu, "Kosha: A peer-to-peer enhancement for the network file system," in *Proc. ACM/IEEE Supercomputing Conf.*, 2004, p. 51.

[7] J. Xiong, S. Wu, D. Meng, N. Sun, and G. Li, "Design and performance of the dawning cluster file system," in *Proc. IEEE Int. Conf. CLUSTER Comput.*, 2003, pp. 232–239.

[8] M. H. Cha, D. O. Kim, H. Y. Kim, and Y. K. Kim, "Adaptive metadata rebalance in exascale file system," *The J. Supercomputing*, vol. 73, no. 4, pp. 1–23, 2017.

[9] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and adaptive metadata management in ultra large-scale file systems," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2007, pp. 403–410.

[10] W. Yu, S. Liang, and D. K. Panda, "High performance support of parallel virtual file system (PVFS2) over quadrics," in *Proc. ACM Int. Conf. Supercomputing*, 2005, pp. 323–331.

[11] S. A. Brandt, E. L. Miller, D. D. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *Proc. IEEE Symp. Mass Storage Syst.*, 2003, pp. 290–298.

[12] A. Thomson and D. J. Abadi, "CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 1–14.

[13] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," in *Proc. USENIX Annu. Tech. Conf.*, 1994, pp. 137–152.

[14] S. Patil and G. A. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. USENIX Conf. File Storage Technol.*, 2011, vol. 11, pp. 177–190.

[15] Y. Fu, N. Xiao, and E. Zhou, "A novel dynamic metadata management scheme for large distributed storage systems," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, 2008, pp. 987–992.

[16] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.

[17] Q. Xu, R. V. Arumugam, K. L. Yong, and S. Mahadevan, "Efficient and scalable metadata management in EB-scale file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 2840–2850, Nov. 2014.

[18] P. Fonseca, R. Rodrigues, A. Gupta, and B. Liskov, "Full-information lookups for peer-to-peer overlays," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 9, pp. 1339–1351, Sep. 2009.

[19] J. Liu, R. Wang, X. Gao, X. Yang, and G. Chen, "AngleCut: A ring-based hashing scheme for distributed metadata management," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2017, pp. 71–86.

[20] X. Zhang, L. Shou, K.-L. Tan, and G. Chen, "iDISQUE: Tuning high-dimensional similarity queries in DHT networks," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2010, pp. 19–33.

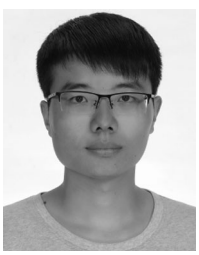
[21] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2004, p. 4.

[22] A. K. Karun and K. Chitharanjan, "A review on hadoop–HDFS infrastructure extensions," in *Proc. IEEE Conf. Inf. Commun. Technol.*, 2013, pp. 132–137.

[23] P. J. Braam, "The lustre storage architecture," arXiv preprint arXiv:1903.01955, (2019).

[24] O. Rodeh and A. Teperman, "zFS-a scalable distributed file system using object disks," in *Proc. IEEE Mass Storage Syst. Technol.*, 2003, pp. 207–218.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.
[26] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "CODA: A highly available file system for a distributed workstation environment," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 447–459, Apr. 1990.
[27] I. Stoica, R. Morris, D. Liben-Nowell , D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003.
[28] A. D. Sarma, A. R. Molla, and G. Pandurangan, "Efficient random walk sampling in distributed networks," *J. Parallel Distrib. Comput.*, vol. 77, pp. 84–94, 2015.
[29] S. Niazi, M. Ismail, S. Grohsschmiedt, M. Ronstrm, S. Haridi, and J. Dowling, "HopsFS: Scaling hierarchical file system metadata using newSQL databases," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 89–103.
[30] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design* (5th Edition). Boston, MA, USA: Addison Wesley, 2011.
[31] K. D. Fairbanks, Y. H. Xia, and H. L. Owen, "A method for historical Ext3 inode to filename translation on honeypots," in *Proc. Int. Comput. Softw. Appl. Conf.*, 2009, pp. 392–397.
[32] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the R* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, 1986.
[33] J. A. Zounmevo and A. Afsahi, "An efficient MPI message queue mechanism for large-scale jobs," in *Proc. Int. Conf. Parallel Distrib. Syst.*, 2012, pp. 464–471.
[34] A. Videla and J. J. W. Williams, *RabbitMQ in Action : Distributed Messaging for Everyone*. Shelter Island, NY, USA: Manning Publications, 2012.
[35] Y. Hong, Q. Tang, X. Gao, B. Yao, G. Chen, and S. Tang, "Efficient R-tree based indexing scheme for server-centric cloud storage system," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 6, pp. 1503–1517, Jun. 2016.



Yuanning Gao received the BE degree in computer science and technology from the Ocean University of China, in 2017. He is currently working toward the doctor degree majoring in computer technology in Shanghai Jiao Tong University. He completed this work when he was working in Data Communication and Engineering group. His research interests include distributed file system, indexing and data science.

Xiaofeng Gao received the BS degree in information and computational science from Nankai University, China, in 2004; the MS degree in operations research and control theory from Tsinghua University, China, in 2006; and the PhD degree in computer science from The University of Texas at Dallas, in 2010. She is currently a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Her research interests include distributed system, wireless communications, data engineering, and combinatorial optimizations. She has published more than 160 peer-reviewed papers in the related area, including well-archived international journals such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Mobile Computing*, the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Journal on Selected Areas in Communications*, and also in well-known conference proceedings such as SIGKDD, INFOCOM, ICDCS, etc. She has served on the editorial board of Discrete Mathematics, Algorithms and Applications, and as the PCs and peer reviewers for a number of international conferences and journals. She is a member of the IEEE.

Xiaochun Yang received the BS, MS and PhD degrees in computer science from Northeastern University, China, in 1995, 1998, and 2001, respectively. She is currently a professor with the Department of Computer Science, Northeastern University, China. She had been invited by many universities including Brigham Young University, UC Irvine, and the Hong Kong University of Science and Technology. Her research interests include big data management and analysis, data quality, and data privacy preserving. She has published more than 100 peer-reviewed papers in the related area, including the *IEEE Transactions on Knowledge and Data Engineering*, the *ACM Transaction on Database Systems*, SIGMOD, VLDB, ICDE, AAAI, etc. She is a member of the ACM, the IEEE Computer Society, and a senior member of the CCF. She is a member of the IEEE.

Jiaxi Liu received the BE degree in mathematics & applied mathematics from Shanghai Jiao Tong University, China, in 2016. He is currently working toward the graduate degree majoring in computer technology at Shanghai Jiao Tong University. He completed this work when he was working in Data Communication and Engineering group. He has won the Best Paper Award of DASFAA 2017. His research interests include indexing, distributed systems and machine learning.

Guihai Chen received the BS degree from Nanjing University, in 1984, the ME degree from Southeast University, in 1987, and the PhD degree from the University of Hong Kong, in 1997. He is a distinguished professor of Shanghai Jiao Tong University, China. He had been invited as a visiting professor by many universities including Kyushu Institute of Technology, Japan, in 1998, University of Queensland, Australia, in 2000, and Wayne State University during September 2001 to August 2003. He has a wide range of research interests with focus on sensor networks, peer-to-peer computing, high-performance computer architecture and combinatorics. He is a senior member of IEEE and has published more than 250 peer-reviewed papers, and more than 170 of them are in well-archived international journals such as the *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, the *Wireless Networks*, the *The Computer Journal*, the *International Journal of Foundations of Computer Science*, and the *Performance Evaluation*, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNEXT and AAAI. He has won several best paper awards including ICNP 2015 best paper award. His papers have been cited for more than 10000 times according to Google Scholar. He is a CCF fellow.