H2Cloud: Maintaining the Whole Filesystem in an Object Storage Cloud

Minghao Zhao
Tsinghua University
zhaominghao.thu@gmail.com

Zhenhua Li
Tsinghua University
lizhenhua1983@tsinghua.edu.cn

Ennan Zhai
Yale University
ennan.zhai@gmail.com

Gareth Tyson
Queen Mary University of London
gareth.tyson@qmul.ac.uk

Chen Qian
University of California, Santa Cruz
cqian12@ucsc.edu

Zhenyu Li
ICT, CAS
zyli@ict.ac.cn

Leiyu Zhao
Tsinghua University
zhaolythu@gmail.com

ABSTRACT

Object storage clouds (*e.g.,* Amazon S3) have become extremely popular due to their highly usable interface and cost-effectiveness. They are, therefore, widely used by various applications (*e.g.,* Dropbox) to host user data. However, because object storage clouds are flat and lack the concept of a directory, it becomes necessary to maintain file meta-data and directory structure in a separate index cloud. This paper investigates the possibility of using a *single* object storage cloud to efficiently host the whole filesystem for users, including both the file content and directories, while avoiding meta-data loss caused by index cloud failures. We design a novel data structure, Hierarchical Hash (or H2), to natively enable the efficient mapping from filesystem operations to object-level operations. Based on H2, we implement a prototype system, H2Cloud, that can maintain large filesystems of users in an object storage cloud and support fast directory operations. Both theoretical analysis and real-world experiments confirm the efficacy of our solution: H2Cloud achieves faster directory operations than OpenStack Swift by orders of magnitude, and has similar performance to Dropbox but yet does not need a separate index cloud.

CCS CONCEPTS

• **Information systems** → **Cloud based storage**; • **Networks** → **Cloud computing**; • **Software and its engineering** → Cloud computing;

1 INTRODUCTION

Recent years have seen enormous growth in the use of object storage clouds, such as Amazon S3, OpenStack Swift and Aliyun OSS [1]. The growth is largely due to their highly usable nature (*i.e.,* the simple *flat* data object operations like `PUT`, `GET` and `DELETE`), as well as their cost-effectiveness (*i.e.,* low unit price of storing/accessing data). As a result, they have been widely used by various applications (*e.g.,* Dropbox, Netflix and Airbnb) to host user data. Their simplicity, however, is also a weakness, as it forces users to translate familiar POSIX-like [46] file directory transactions into flat operates. Consequently, applications wishing to use file-based abstraction are required to build a secondary sub-system that can map a hierachical filesystem into the flat object store offered. This is due to the functional gap between the flat object storage cloud and *hierarchical* directory structures. To date, no object storage systems can perform functional mappings from (relatively complex) filesystem operations to simple object-level operations natively.

An example application performing such mappings is modern cloud storage services (*e.g.,* Dropbox, Google Drive, iCloud Drive and Microsoft OneDrive), which all tend to provide a hierarchical full-fledged filesystem to their users. Specifically, they usually support POSIX-like file and directory operations such as `READ`, `WRITE`, `MKDIR`, `RMDIR`, `MOVE`, `LIST`, and `COPY`. The object storage platforms that cloud storage services rely on, however, only provide flat operates (*i.e.,* `PUT`, `GET` and `DELETE`). Therefore, mainstream cloud service providers like Dropbox are required to use two clouds to host a user's file content and directories separately [10, 24, 25], *i.e.,* an object storage cloud (like Amazon S3 and more recently Magic Pocket [19]) and a dedicated index cloud (mostly built on top of Amazon EC2 and EBS [51], where EBS is responsible for automatically snapshoting EC2 instances).

The abovementioned two-cloud architecture, therefore, incurs additional cost and effort for achieving high data reliability and scalability as it is necessary to maintain two sub-systems. For example, Dropbox, which does offer filesystem operations [2, 19], needs enormous additional, disparate efforts to achieve these two properties for maintaining the directories on top of Amazon EC2 and EBS. Otherwise, the filesystem of every user will be put at risk of getting lost or corrupted [35, 57]. And it is generally taken that such risks are highly related to reliability and scalability problems inside the index cloud [11, 13] [1]. Hence, we pose a challenging question: *Can we use an object storage cloud to efficiently host both the file*

[1]This is potentially the main reason why Dropbox hopes to build its own data centers and network infrastructures to manage its index [55].

ICPP 2018, August 13–16, 2018, Eugene, OR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225083

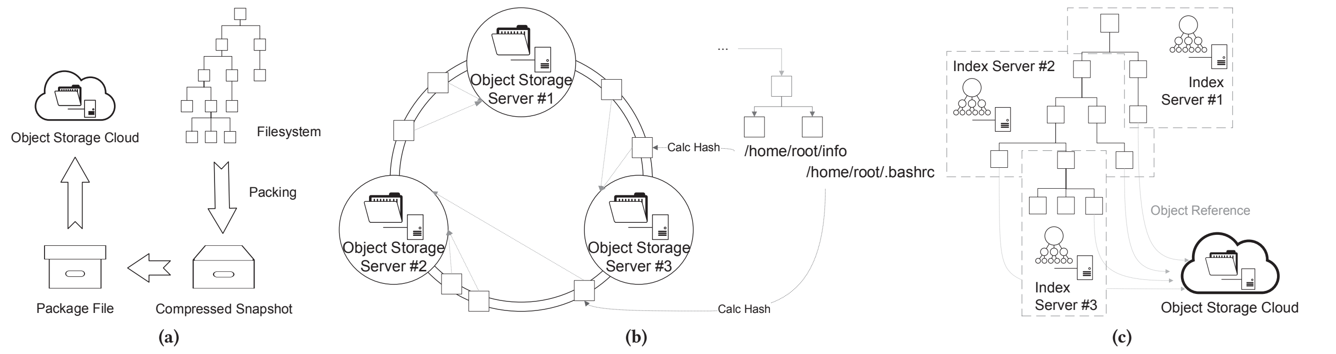

Figure 1: (a) Cumulus backups the filesystem to an object storage cloud using Compressed Snapshots; (b) As to Consistent Hash, files are distributed to object storage servers according to the hash values of their full file paths; (c) For Dynamic Partition, the filesystem is partitioned to a few index servers and an object storage cloud.

content and directories? If the answer is positive, we can re-take advantage of the object storage cloud to automatically provide high reliability and scalability for directories without additional efforts. As for Dropbox, the application instances working on Amazon EC2 virtual machines (VMs) would then become stateless (since they do not need to maintain the meta-data for users any more), and thus can easily scale and never be subject to meta-data loss/corruption problems. Besides, there will be no need to employ Amazon EBS for VM snapshots.

Most current cloud storage services do not publish their technical details. However, we wonder whether there are any solution-based publicly-available techniques that can answer our question. To this end, we analyze a variety of existing solutions (as elaborated in §2) and summarize the major potential solutions as follows (meanwhile demonstrated in Figure 1):

- *Compressed Snapshots.* Vrable *et al.* proved it feasible to *backup* users' filesystems to an object storage cloud by developing Cumulus [49]. Cumulus flattens the directories to a linear list, packs the whole filesystem of a user to a Compressed Snapshot, and puts it in an object storage cloud, as in Figure 1a. Obviously, performing filesystem operations (either file access or directory operations) on Compressed Snapshot is extremely slow.
- *Consistent Hash with a file-path DB.* Consistent Hash [20] is adopted by OpenStack Swift to implement a "pseudo filesystem." As in Figure 1b, to determine the position of a file, it calculates the hash value of the full file path and looks it up in a consistent hashing ring [5]. Thus, any operation that needs to traverse or change the directory structure has to be performed across all files in the directory, thus leading to poor efficiency. In OpenStack Swift, an SQL-style file-path database is utilized to boost `LIST` and `COPY` operations, but the effect turns out fairly limited.
- *Dynamic Partition with a separate index.* Adopted by modern distributed storage systems like Ceph [52, 53] and Panasas [54], Dynamic Partition keeps the directories in a few index servers and strategically partitions the directories for load balance, as illustrated in Figure 1c. This enables efficient directory operations. Meanwhile, each leaf node in the directory tree refers to the content of a file in the object storage cloud. However, a separate index cloud is required to host the directories.

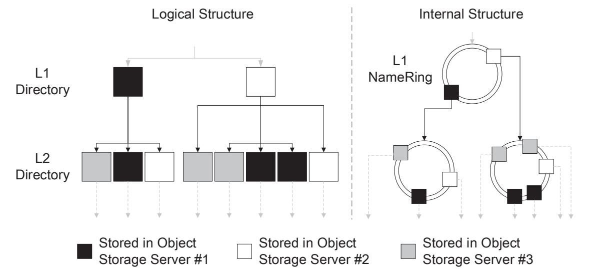
Logical Structure
Internal Structure
L1 Directory
L2 Directory
L1 NameRing
Stored in Object Storage Server #1
Stored in Object Storage Server #2
Stored in Object Storage Server #3

Figure 2: H2 utilizes NameRings to preserve and exploit the filesystem hierarchy in an object storage cloud.

In general, we find none of these existing solutions satisfy our goal of efficiently maintaining the whole filesystems of users in an object storage cloud. To achieve this goal, we design a novel data structure, Hierarchical Hash (or H2), to enable efficient mappings from filesystem operations to object-level operations without requiring a separately managed indexing and mapping service. Based on H2 we implement a cloud storage system, H2Cloud, that can maintain large filesystems of users in an object storage cloud. H2Cloud possesses two important properties: 1) Files and directories are all stored as objects, and no extra index or database is required. Thus, H2Cloud significantly simplifies the system management and reduces the operational cost. 2) H2Cloud supports fast filesystem operations especially for the directory operations. For example, typically, `LIST`ing 1000 files costs just 0.35 second and `COPY`ing 1000 files costs ~10 seconds.

As demonstrated in Figure 2, in H2 every directory corresponds to a NameRing—a data structure that maintains a list of *direct children* (*i.e.,* only including current-level files and sub-directories) under the directory. Each NameRing is linked in a hierarchy that represents the directory structure of the filesystem being stored. Importantly, the storage of these data structures is underpinned by a single, larger consistent hashing ring using a *namespace-decorated* method, so the overall load balance of objects is automatically kept.

H2 allows the directory structure to be natively represented within the object store. Instead of hashing the full path (as what Consistent Hash does in OpenStack Swift), H2 hashes each directory name along the path in a level-by-level manner when accessing a

Table 1: A quantitative comparison of representative data structures (including our proposed H2) for hosting users' filesystems. Here N is the total number of files in the filesystem, d denotes the depth of an accessed file in the directory tree (d is typically quite small, *e.g.*, lying between 1 and 19), n represents the number of files stored in a certain directory, and m is the number of direct children under a certain directory.

Data Structure	System Architecture	System Scalability	Time Complexity for				
			File Access	`MKDIR`	`RMDIR`, `MOVE`	`LIST`	`COPY`
Compressed Snapshot	Single Cloud	Yes	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(N)$
Content Addressable Storage (with Multi-Layer Index)	Single Cloud	Yes	$O(1)$	$O(N)$	$O(N)$	$O(m)$	$O(N)$
Consistent Hash (CH)	Single Cloud	Yes	$O(1)$	$O(1)$	$O(n)$	$O(N)$	$O(N)$
CH with a File-Path DB (OpenStack Swift)	Single Cloud	Limited	$O(1)$	$O(1)$	$O(n)$	$O(m \cdot logN)$	$O(n + logN)$
Single Index Server	Two Clouds	Limited	$O(d)$	$O(1)$	$O(1)$	$O(m)$	$O(n)$
Static Partition	Single Cloud	No	$O(d)$	$O(1)$	$O(1)$	$O(m)$	$O(n)$
Dynamic Partition (DP)	Two Clouds	Yes	$O(d)$	$O(1)$	$O(1)$	$O(m)$	$O(n)$
DP on Shared Disk	Single Cluster	Constrained	$O(d)$	$O(1)$	$O(1)$	$O(m)$	$O(n)$
Hierarchical Hash (H2)	Single Cloud	Yes	$O(1)$ or $O(d)$	$O(1)$	$O(1)$	$O(1)$ or $O(m)$	$O(n)$

file with an absolute path. Specifically, H2 uses the name of a Level-1 (L1) directory to locate the NameRing that includes all names of the direct children under the L1 directory. Then, H2 uses the name of an L2 directory to locate the NameRing of the L2 directory. It repeats until the desired file is reached. In contrast, for a file access with a relative path H2 achieves the optimal $O(1)$ efficiency, since directly hashing the provided relative path will get the location of the targeted file in the consistent hash ring. Because the filesystem hierarchy is preserved and explored, H2 achieves high efficiency especially for directory operations. Theoretical analysis (in Table 1) clearly indicates the sound performance of H2 in almost all aspects.

In addition, we have developed an open-source prototype system of H2Cloud based on OpenStack Swift (§4), and conducted comprehensive evaluations on the real-world performance of H2Cloud (§5). Compared with OpenStack Swift, H2Cloud achieves faster directory operations by orders of magnitude on handling `RMDIR`, `MOVE`, `RENAME` and `LIST` for large filesystems. File access and `MKDIR` are slower than those of OpenStack Swift [2], but still faster than Dropbox. Compared with Dropbox, H2Cloud achieves similar performance in all aspects without using a separate index cloud.

2 RELATED WORK

As we have noticed in §1, a suitable data structure is the key to efficiently maintaining users' filesystems in an object storage cloud. Thus, our target problem is reduced to how to construct an appropriate data structure to satisfy our goal. In this section, we investigate a variety of representative data structures coupled with relevant storage systems, and discuss why they are not suited to our goal. Also, we briefly present how our proposed H2 data structure outperforms the existing ones at the end. Particularly, the major properties of all investigated data structures (including H2) are listed in Table 1 for a quantitative comparison.

Compressed Snapshot and Cumulus. Using Compressed Snapshots (a Snapshot is a read-only view of the filesystem at a certain point), Cumulus packs and compresses the content of files to several TAR files (named *Segments*), and meanwhile flattens the directories to a one-dimensional (linear) list called the *Metadata log* [49] (as shown in Figure 1a). The Segments and Metadata logs collectively constitute the Compressed Snapshot, which is then stored in the object storage cloud.

Because Cumulus uses Metadata logs to snapshot the filesystem, it performs well when retrieving the whole filesystem from the object storage cloud. However, it gets extremely poor performance when accessing a specific file, since it has to traverse all the Metadata logs in the Compressed Snapshot so as to locate a file in the filesystem. As a result, the time complexity for random file access is as high as $O(N)$.

Accordingly, the complexities of most directory operations are also $O(N)$, as indicated in Table 1. For the above reasons, *Cumulus is able to backup a filesystem but is not competent to maintain a "real" filesystem that frequently changes.*

Content Addressable Storage (CAS). Foundation [39] and Venti [37] are storage systems built using a content addressable approach, where the data locations are tightly coupled with the data content [38, 48]. Typically, a file (block) is located by the hash value of its content. Camlistore [6] extends the flexibility of traditional CAS architecture and achieves hierarchical structure among the files, by packing the hash values of lower-level files or file blocks into additional data blocks (called pointer blocks) which are also written to the storage devices. CAS is especially efficient for file access operation, as simply hashing the file content will obtain its location, which result in $O(1)$ time complexity. However, *a file (block) cannot be modified without changing its location, and even simple directory operations like* `MKDIR` *and* `RMDIR` *require reconstructing the whole hierarchical index*, which will result in $O(N)$ complexity. *Thus, like Cumulus, it is also particularly suitable for backup and data sharing systems with few file modifications.*

Consistent Hash (CH). Owing to its high scalability, self-organization, and load balancing, Consistent Hash has been widely used in flat object storage clouds such as OpenStack Swift [7] and Aliyun OSS [1]. Nonetheless, it does not support a hierarchical filesystem abstraction. To mitigate this shortcoming, OpenStack Swift suggests to implement a pseudo filesystem based on Consistent Hash (see Figure 1b). In other words, it mimicks the filesystem structure by treating the file path as a discrete string that can be hashed. This idea is also adopted by Aliyun OSS, CalvinFS [47], Lazy Hybrid [4], Cassandra [21] and some large-scale key-value storage systems

[2]For file access and `MKDIR` operations, round trip time (RTT) is the main factor to affect the user experience since RTT dwarfs their operation time. Hence, it is in fact unnecessary for us to further accelerate the two operations by increasing the design complexity.

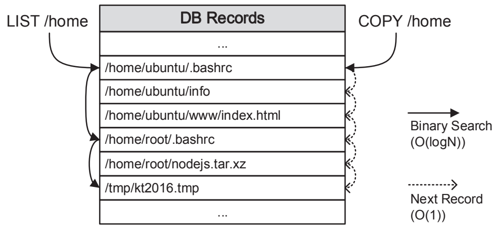

Figure 3: OpenStack Swift maintains an extra file-path DB to boost LIST and COPY operations with binary search.

such as Amazon's Dynamo [8] and Facebook's f4 [29]. Recently perfect hash based index is also proposed [56].

Specifically, by calculating the hash value of the full file path, Consistent Hash looks a file up in a consistent hashing ring [5] to determine which object storage server the file is located in, as depicted in Figure 1b. A benefit of this is that there is no need for a separate index. On the other hand, the side effect of the pseudo filesystem lies in that *any operation that needs to traverse or change the directory structure has to be performed across all files in the directory or the filesystem,* thus leading to high ($O(n)$ or even $O(N)$) complexity as shown in Table 1.

CH with a File-Path DB and OpenStack Swift. Since CH exhibits poor efficiency on handling `LIST` and `COPY` operations, OpenStack Swift maintains an extra file-path database (with SQLite or MySQL) for each user (account) to speed them up; every file corresponds to a record in this file-path DB. Therefore, via binary search of the DB records, it reduces the time complexities of `LIST` (from $O(N)$ to $O(m \cdot logN)$) and `COPY` (from $O(N)$ to $O(n+logN)$) (shown in Figure 3). Generally, file-path DB provides a feasible approach in accelerating `LIST` and `COPY` operations, and it has been adopted by some modern cloud filesystems such as CassandraFS [27], GiraffaFS [41] and HopsFS [30], which take full advantage of the recently advanced high performance distributed databases to manage the file-path (*e.g.,* HopsFS uses NewSQL [31, 44] database for file-path management). But in spite of this, *the achieved time complexities are still unsatisfactory as for a large system scale (N).* In addition, the usage of DB substantially increases the burden on the storage node that hosts the file-path DB, and thus compromises the scalability of the system. Getting rid of the usage of these secondary sub-systems is a core goal of our work.

Single Index Server and GFS/HDFS. GFS [12] and HDFS [42] keep a single index server (called *namenode* in HDFS) separately to maintain the filesystem for a whole storage cluster. *The centralized architecture results in limited scalability*, which is probably why GFS and HDFS are not used by mainstream cloud storage services.

Static Partition and AFS. The Andrew File System (AFS [14, 15]) statically partitions different users' files into different object storage servers. Although clumsy in dynamic situations, it has been popular in certain scenarios for its extreme simplicity. For example, Carnegie Mellon University uses AFS to provide 2-GB storage space for every new student when they are enrolled. This method is also adopted by some light-weight filesystems such as LOCUS [36], NFS [34], Coda [40], MapReduce-R [43] and XtreemFS [18]. But obviously, *statically partitioned files and directories have the negative effect on filesystem operations with different partitions involved, and thus scalability cannot be expected here.*

Dynamic Partition (DP). To address the limitations of Static Partitioning, a straightforward method is to keep multiple index servers, monitor the file access workload, and dynamically partition the directories into the index servers with sophisticated load-balance algorithms. DP has been implemented in many modern storage systems like Ceph [52], PanFS[54], SmartStore [16, 17], OrangeFS [28], GIGA+ [33], GlobleFS [32] and PROMES [26]. Moreover, DP is highly likely to be used by Dropbox, which is indicated by the experiment results in §5.3. As mentioned in §1, *DP enables efficient directory operations but requires a separate index.*

DP on Shared Disk. It is worth noting that some systems (*e.g.,* BlueSky [50], xFS [45] and SCFS[3]) implement DP with a different architecture, *i.e.,* on *Shared Disk.* They build only one shared-disk storage cluster that supports both the DP middleware and underlying file block storage. However, this architecture requires strong consistency among the shared disks, so partition tolerance is compromised according to the classical CAP theorem. Thus, *DP on Shared Disk is considered unsuitable for distributed cloud storage systems which usually sacrifice strong consistency for partition tolerance.*

Comparison with H2 and H2Cloud. As illustrated in Table 1, H2 generates sound performance in almost all aspects. Particularly for `LIST` operations, with the help of NameRing, the time complexity is $O(1)$ when the user only needs to list the names of the direct children under a directory. Otherwise (when the user wants to list more detailed information), the time complexity is $O(m)$. For file access, H2 provides both an internal, quick method and an external, (user-friendly) regular method; the former can be accomplished in $O(1)$ time and the latter in $O(d)$ time (detailed in §3.2). *Compared with H2, any other data structure is subject to crucial performance limitations in at least one aspect.* Therefore, when we implement the H2Cloud system based on H2, H2Cloud supports fast filesystem operations especially for the directory operations.

3 DESIGN OF H2

As the core design of our H2Cloud system, we propose a novel data structure, Hierarchical Hash (or H2), which aims at mapping the hierarchical file paths into a representation specific to object-level operations (called NameRing), thus enabling us to maintain this representation in an object storage cloud. In this section, we first elaborate on the design of the H2 data structure (§3.1). Then, we describe the coupled file access algorithm (§3.2), and the NameRing maintenance protocol (§3.3).

3.1 H2 Data Structure

In designing H2, we aim to preserve and exploit the hierarchical information implied in the file path, and meanwhile keep the load balance of objects in an object storage cloud. Specifically, our goal is achieved in three steps as demonstrated in Figure 2. Suppose a user Alice wants to maintain her Ubuntu filesystem in an object storage cloud. First of all, we translate each full directory path or full file path in Alice's filesystem to a *namespace-decorated* relative path. For example, in Figure 4(a) the directory path `/home/ubuntu` is translated to `N94::ubuntu`, and the file path `/home/ubuntu/file1` is translated to `N02::file1`. Here `N94` represents a certain namespace, and it is the universally unique identifier (UUID) [23] of the corresponding directory path (`/home/`). More in detail, because `/home/`

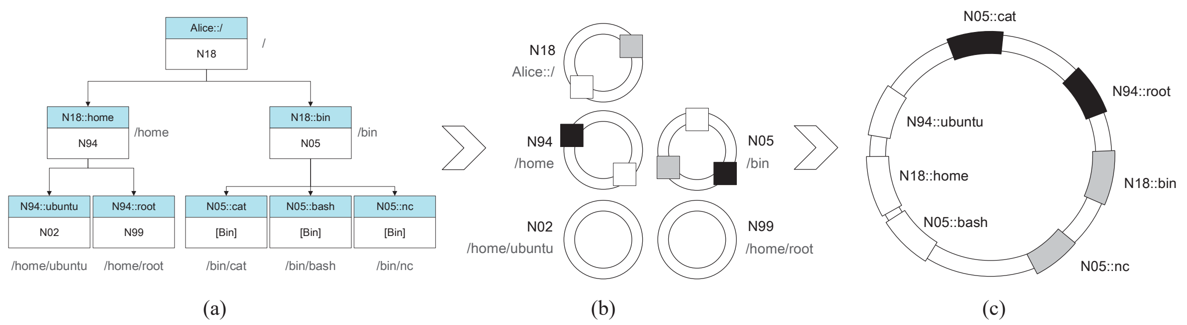

Figure 4: An example of the H2 data structure, showing how Alice's filesystem is maintained in an object storage cloud. The left sub-figure (a) is the namespace-decorated filesystem tree, the middle sub-figure (b) plots the corresponding NameRings (note that a regular file like `/bin/cat` does not possess a NameRing), and the right sub-figure (c) represents the single, larger consistent hashing ring on which the multiple NameRings are built.

is the 6th directory created by the 1st storage node at the UNIX timestamp `1469346604539`, this directory will be given a UUID `06.01.1469346604539` and thus `N94 = 06.01.1469346604539`. In practice, the sequence numbers (*e.g.,* 6th and 1st) of directories and storage nodes are assigned based on specific system implementations. With the above efforts, every directory in Alice's filesystem corresponds to a unique namespace.

Second, each namespace (directory) possesses a NameRing, which is the data structure we use to maintain a list of all files in the directory. Specifically, the NameRing goes through all the direct children of the directory by recording their (directory or file) names. Formally, a NameRing comprises a list of tuples looking like

$$(child_0, t_0), (child_1, t_1), \cdots, (child_{m-1}, t_{m-1}),$$

where $child_i$ is the file or directory name of the i-th direct child of the corresponding directory, m is the number of direct children, and t_i is a UNIX timestamp representing a creation or deletion time. For example in Figure 4(b), the NameRing of the namespace `N05` goes through `cat`, `bash`, and `nc`. Hence with the help of NameRing, a `LIST` operation on `N05` (representing the directory `/bin`) can be executed in $O(1)$ time when Alice only needs to list the names of the direct children under `/bin`. Further, when Alice wants to list more detailed information of the direct children, the time complexity increases to $O(m)$ (m = 3 for `/bin`) since we have to access the specific object that stores the information of each direct child of `/bin` using the NameRing. As a result, NameRing preserves the hierarchical information of a user's filesystem, and every directory operation can be translated to NameRing updates in H2.

Third, each directory and its NameRing are stored in a single, larger consistent hashing ring as an object, respectively. Note that in Figure 4(c), only a part of directory objects are drawn to make the figure tidy. At the moment, we use the consistent hashing ring of OpenStack Swift which keeps sound load balance for the data objects. Therefore, all the objects organized by the H2 data structure are also evenly distributed across storage servers.

3.2 File Access

H2 provides two methods to access a file, including 1) a quick method through a (namespace-decorated) relative path and 2) a regular method through a common full file path. First, given a relative path like `N02::file1`, accessing the file can be accomplished in $O(1)$ time since we can directly hash the relative path to locate the file in the consistent hashing ring. Since a common user can hardly remember or understand the namespace string (*e.g.,* `N02`), this quick method is mainly used by the system's internal operations. Second (as a user-friendly access method), given a full file path with a directory depth of d (like `/home/ubuntu/file1` whose d = 3), H2 needs to locate the file level by level along d NameRings, so the time complexity is $O(d)$.

3.3 NameRing Maintenance

As noticed in §3.1, each NameRing is linked in a hierarchy that represents the directory structure of the filesystem being stored. Therefore, when a directory is updated due to the creation of a new child or the deletion of an existing child, its NameRing should also be updated (by adding or removing a tuple) accordingly. Further, more complex directory operations like `RMDIR`, `MOVE`, and `COPY` are also achieved through NameRing maintenance.

3.3.1 *A Strawman Solution.*

An ideal and straightforward way for NameRing maintenance is to construct a ring-structured *synchronous* protocol that updates any involved NameRings among each node in real time. Nevertheless, such a strawman solution is not suitable for real-world object storage cloud systems for the following two reasons. First, while a synchronous protocol is able to offer stronger consistency, it provides poor data availability. In reality, mainstream object storage cloud systems tend to require better data availability rather than consistency. OpenStack Swift, for example, only provides eventual consistency to its customers in order to ensure high data availability. Second, synchronous updates need distributed locks [22] to enable the serialization of filesystem operations, which inevitably bring performance bottlenecks to filesystem operations on those frequently accessed directories.

3.3.2 *NameRing Maintenance Protocol.*

Due to the above two-fold drawbacks of the strawman synchronous protocol, we propose a practical asynchronous protocol for effective NameRing maintenance in H2. Our proposed NameRing maintenance protocol consists of two phases.

Phase 1: For every filesystem operation that changes NameRings, our protocol needs to submit a *patch*, which is a log file recording the update information (*e.g.,* file deletion). A submitted patch is assigned a UUID with the targeted NameRing concatenating the node number that releases it, as well as the incremental patch number. For example, `N97::/NameRing/.Node01.Patch03` indicates the third patch of the namespace `N97`'s NameRing, which has been submitted by the node of No. 01.

Phase 2: After patches are submitted successfully, these patches are applied to the affected NameRing(s) in the following two steps: 1) patches merging within the node who submits them, and 2) coordination among all the nodes.

- In the first step, patches within each node are arranged as a link-list. Specifically, each updated NameRing is associated with a list of unmerged patches, starting with the patch No. 0 (whose absence indicates that no other version exists in this node). The intra-node merging process starts with the No. 0 patch and merges the patch with its successor in the linked list by first fetching a patch, then getting its successor, and finally merging the two patches. After all the patches are merged into one, we merge this "big" patch into the NameRing with the merging algorithm (detailed later). Consequently, each node has its local (but not necessarily consistent) version for each NameRing.
- In the second step, different nodes need to coordinate the updates, so that each node can eventually have the same NameRing views with other nodes. We achieve this goal by gossip flooding. Specifically, each node propagates its update to other nodes using a gossip protocol [9]. Each gossip contains a list of tuples of (N_i, H_j, t_k), which indicate that the local version of NameRing N_i in node H_j has been updated at the timestamp t_k. Upon receiving a gossip, this node fetches the updated version and merges it into its local version, and then puts it forward. In addition, the propagation loop-back is avoided by timestamp comparison—aborting forwarding if the local timestamp is equal or bigger than the timestamp in the local NameRing, as this case indicates that the local version is the newest among the whole system. By this way, patches submitted from other nodes can be successfully synchronized.

NameRing merging algorithm. As a central part of our asynchronous NameRing maintenance protocol, the NameRing merging algorithm merges a patch into its targeted NameRing. In general, when a patch is intended to be merged into a NameRing N_A, it is firstly converted into another virtual NameRing N_B, given that a patch is in the same format as a NameRing. Afterwards, the algorithm merges the two NameRings N_A and N_B by iterating on each child of N_B. Specifically, when a child belongs to both NameRings, the one that has a larger timestamp will override the other. Otherwise (*i.e.,* in which case the child only appears in the patch but does not exist in the previous NameRing N_A), the child will be added to the new output NameRing N_C.

Note that there are only the aforementioned two actions (*i.e.,* inserting and overriding) for the children in a patch to be written into their corresponding NameRing, whereas no child is removed from the NameRing in the patch-NameRing merging phase. In fact, when a file or folder is expected to be deleted from a certain directory, we send a patch to this directory NameRing. In this patch, a tag `Deleted` is appended to the corresponding tuple $(child_i, t_i)$ and then the tuple looks like $(child_i, t_i, \texttt{Deleted})$. This tuple will override and replace the original one in the targeted NameRing afterwards (as it has a larger timestamp), such that the tagged tuple $(child_i, t_i, \texttt{Deleted})$ will appear in the NameRing. We leave the work of "really" removing the tuple from the NameRing until this NameRing is in use (*e.g.,* executing operations such as `MOVE` and `LIST`). We make this design not only for protocol simplification, but also for concurrency avoidance (detailed later).

3.3.3 Concurrency Avoidance. Two methods are adopted to avoid the potentially accompanied concurrency problem. (*a*) Fake operation. For some certain operations, we extend the tuple in a NameRing to enable fake operations. For example, in the case of file removal, we just append a tag `Deleted` instead of deleting this tuple from the NameRing, *i.e.,* the so-called "fake deletion". In this way, even if this removal operation causes a concurrency problem, the problem can still be easily fixed later by first checking the corresponding tag and then updating the stage of the file, when executing operations such as `MOVE` and `COPY`. (*b*) Blocking. When inserting a large file into the filesystem, it is required to generate a UUID and the corresponding metadata for this newly-added file, put the file into the cloud storage through the I/O stream interface, and finally send a patch to modify its parent directory's NameRing. As the file streaming operation takes longer time than directory operations, all the other merging procedures are blocked until the file is fully written into the storage interface and the patch is successfully submitted.

4 IMPLEMENTATION OF H2CLOUD

Based on the design of H2 in §3, we implement an open-source prototype system, named H2Cloud, on top of OpenStack Swift. Our source code is written in the Go language including 14000 lines and is publicly available at *http://github.com/h2cloud/h2cloud.* This section first presents an overview of the H2Cloud system, and then describes the implementation details of its important components.

4.1 System Overview

As depicted in Figure 5, H2Cloud is built on top of an object storage cloud, serving external PC/mobile clients or web requests. Our current prototype adopts OpenStack Swift as the underlying object storage cloud because it is a widely used, open-source, and the de facto standard of state-of-the-art systems. H2Cloud provides filesystem services to the users in the form of web services, *i.e.,* through a series of web APIs. Accordingly, the users can access H2Cloud via a web browser or a native client, by sending HTTP messages to and receiving HTTP messages from the *H2Layer*, which comprises a number of *H2Middlewares* (detailed in §4.2). In brief, H2Middleware embodies the H2 data structure and the coupled algorithms presented in §3. Typically, each H2Middleware wraps an OpenStack Swift proxy server, while other mappings are also acceptable. Like OpenStack Swift proxy servers, multiple H2Middlewares are deployed to distribute workloads for load balancing, or to reduce the service delay when the object storage cloud is geographically distributed across several data centers.

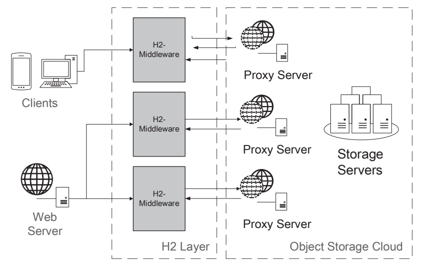

Figure 5: System architecture of H2Cloud.

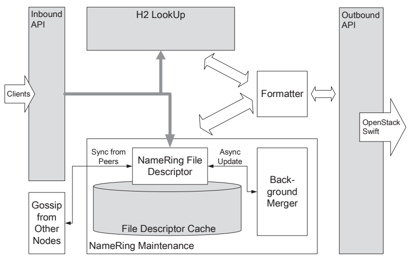

Figure 6: Components within an H2Middleware.

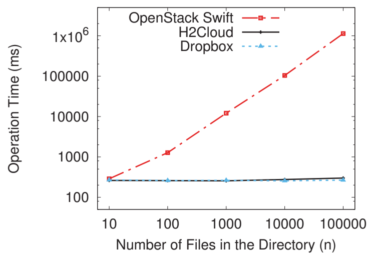

Figure 7: Operation time for MOVE and RENAME.

4.2 H2Middleware

As the key component of H2Cloud, H2Middleware implements the H2 data structure, the file access algorithm, and the NameRing maintenance protocol. For the users, H2Middleware serves as a proxy that provides web APIs for filesystem operations. For the object storage cloud, H2Middleware acts as a client that invokes `PUT`, `GET`, `DELETE`, and other primitives.

Figure 6 presents the components within an H2Middleware. Specifically, the Inbound API and Outbound API modules provide basic utilities to establish HTTP connections with user clients and the object storage cloud. The H2 Lookup module executes the file access algorithm with a (namespace-decorated) relative path or a common full file path. The Formatter module "stringifies" the content of different types of data into string-style objects, so that these data can be easily hosted in the object storage cloud. The NameRing Maintenance module is responsible for submitting patches and merging patches into their corresponding NameRings, and it implements the NameRing maintenance protocol (§3.3). This module is further made up of several sub-modules, including the NameRing File Descriptor, the File Descriptor Cache and the Background Merger, to coordinate the merging procedure and execute the merging algorithm. In addition, there are a few other modules inside an H2Middleware for inter-communications and system monitoring.

4.3 Inbound API and Outbound API

As shown in Figure 6, the Inbound API and Outbound API modules handle the incoming and outgoing HTTP messages of an H2Middleware, respectively. In particular, Inbound API is embodied by an HTTP server that provides three types of web APIs: 1) Account APIs which create or delete an account for a certain user; 2) Directory APIs which traverse or modify the structure of one or more directories; and 3) File Content APIs which provide `READ` and `WRITE` accesses to files. Similarly, Outbound API is embodied by another HTTP server that interacts with the underlying object storage cloud through `PUT`, `GET`, `DELETE` and other primitives.

4.4 Formatter

While an object storage cloud hosts all types of data in the form of objects, certain types of data (*e.g.*, our devised NameRings and NameRing patches) are not suited to directly becoming objects. Thus, we make use of "stringifying" to convert every type of data to string-style objects. This task is accomplished by the Formatter module before any type of data is put in the object storage cloud. In the filesystems maintained by H2Cloud, there are mainly three types of data objects: files, directories, and NameRings (together with the NameRing patches). They are converted to strings in the following three ways: 1) Files can be taken as either ASCII or binary strings in themselves; 2) Directories are converted to ASCII strings corresponding to their namespaces; and 3) NameRings are represented in lists of tuples(refer to §3.1). These tuples are alphabetically sorted by their names and packed to ASCII strings one after another. As for a NameRing patch, it is firstly converted to the form of a normal NameRing and then represented in lists of tuples.

4.5 NameRing Maintenance Module

As described in §3.3, asynchronous update is adopted by H2Cloud and updating a NameRing is in fact submitting a patch to the system. The patch affects the system going through two steps: intra-H2Middleware merging and inter-H2Middleware synchronization. In order to achieve this functionality, some mutually cooperative sub-modules are designed. specifically, each NameRing corresponds to a unique File Descriptor. It controls the submission, updating and synchronization of NameRings, as well as invokes the synchronization mechanism. All the file descriptors are stored and organized in the File Descriptor Cache. The Background Merger is responsible for merging patches in the patch-chain and merging a patch into the NameRing automatically, under the direction of the File Descriptor. In addition, the Gossip Arrangement sub-module is designed for generating and spreading synchronization information.

5 EVALUATION

Although theoretical analysis in Table 1 has indicated the sound performance of H2 in almost all aspects, this section comprehensively evaluates the performance of H2Cloud system through real-world experiments. We also compare the performance of H2Cloud with those of OpenStack Swift (the state-of-the-art single-cloud solution using CH with a File-Path DB) and Dropbox (the state-of-the-art two-cloud solution probably using DP [3]).

5.1 Methodology

We make a rack-scale H2Cloud (and its underlying OpenStack Swift) deployment for conducting real-world experiments. The deployment involves nine HP ProLiant DL380p servers located in the same IDC rack. Each server is equipped with an 8-core Intel Xeon CPU

[3]We infer that Dropbox is highly likely to use the Dynamic Partition data structure from the experiment results in §5.3.

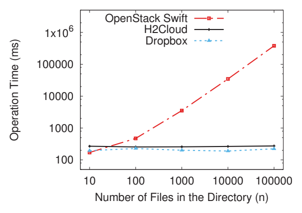

Figure 8: Operation time for RMDIR.

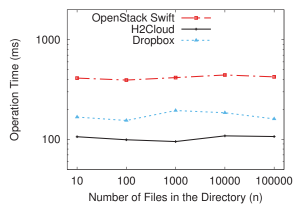

Figure 9: Operation time for LIST

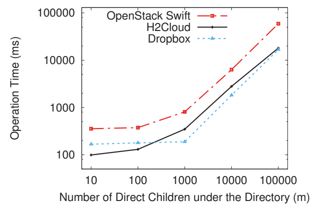

Figure 10: Operation time for LIST

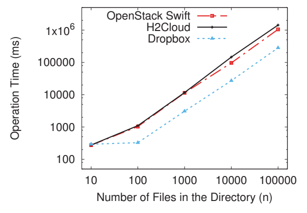

Figure 11: Operation time for COPY.

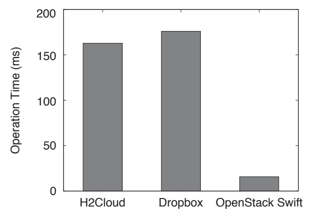

Figure 12: Operation time for MKDIR.

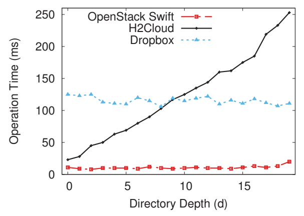

Figure 13: Operation time for file access.

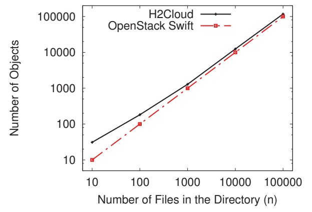

Figure 14: Number of objects.

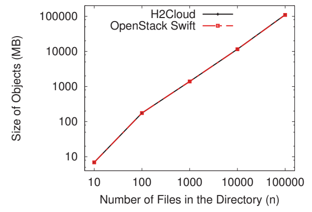

Figure 15: Size of objects.

@ 2.50 GHz, 32-GB 1600-MHz DDR3 memory, 7×600-GB 15K-RPM SAS disk storage, and four 1-Gbps Broadcom Ethernet interfaces. The operating system running on each server is Ubuntu 14.04 LTS x64. All these servers are connected by a commodity Huawei switch with 1-Gbps LAN bandwidth and 100-Mbps WAN bandwidth.

In our experiments, we use one of the nine servers (called *Node-0*) to run the OpenStack Keystone service for account/data authentication; meanwhile, *Node-0* plays the role of proxy node in the OpenStack Swift system. The other eight servers are used as OpenStack Swift storage nodes. Among these storage nodes, three replicas are kept for each data object. In addition, an H2Middleware is hosted in *Node-0*—when enabled, we run the experiments with H2Cloud; when disabled, we run the experiments with OpenStack Swift.

In order to obtain real-world workloads, we invited nearly 150 users from both campuses and companies to host their filesystems in H2Cloud. Among these invited users, some users' filesystems are "light", *i.e.,* consisting of several shallow directories and hundreds of files, while the filesystems of the rest of users are "heavy", *i.e.,* including thousands of directories in different depths and millions of files. In general, the number of files in a directory ranges from zero (empty folder) to nearly half a million, and the directory depth range from zero to more than 20. The users' manipulations cover most of the POSIX-like file and directory operations (detailed latter). The file type also is diversified, including videos and database backups with the file size of gigabytes (GB), text and configuration files with size less than one kilobyte (KB), and other file types (*e.g.,* documents and figures) with a medium file size. Such heterogeneous filesystems give our experiments abundant variances and generality.

For the experiments with OpenStack Swift, we replay these H2Cloud users' workloads by disabling the H2Middleware. Besides, to evaluate the performance of Dropbox (whose implementation is invisible to us), we send controlled HTTP requests to the Dropbox system to replay the workloads. Moreover, since these H2Cloud users' filesystem operations cannot fully cover all the desired usage scenarios and scales for comprehensive sensitivity analysis, we also conducted complementary benchmark experiments using controlled clients (based on the collected real-world workloads).

5.2 Metrics

In our experiment, we mainly focus on the performance of the filesystems, which is measured by the *operation time* of MOVE, RENAME, RMDIR, LIST, COPY, MKDIR, and file access. Here operation time denotes how long the system needs to process a filesystem operation, excluding the round trip time (RTT) that is spent delivering HTTP requests and responses over the Internet.[4] We do not take RTT into account, because 1) it depends on the network environments rather than our evaluated systems, and 2) it may well dwarf the operation time of our evaluated systems, thus making our comparison unclear and unfair. Afterwards, we evaluate the impact of RTT to the overall performance of the cloud filesystems. In addition, for file access operations we only record the lookup time while exclude the read/write time of the file, in order to avoid the (often dominant) impact of file size on the file access time. Meanwhile, we examine the *storage overhead* (the major system overhead) of H2Cloud in comparison to OpenStack Swift.

5.3 Experiment Results

MOVE, RENAME and RMDIR. Figure 7 records the operation time of MOVE and RENAME when the number of files in the directory (*n*) varies from 10 to 100,000. We do not distinguish MOVE and RENAME here since the latter is in fact a special case of the former and thus has identical performance. Our key finding is that as *n* exponentially grows, the operation time for OpenStack Swift also exponentially

[4]The RTT of Dropbox is estimated by PINGing the relevant Dropbox storage server(s) in real time, because we are unable to take precise measurements of Dropbox's communication and data flows. Moreover, this time delay will also be used to evaluate the influence of RTT to the overall performance of a cloud filesystem.

increases, while the operation time for H2Cloud and Dropbox is generally unchanged. The similar phenomenon happens to the operation time of RMDIR, which is illustrated in Figure 8. The above experiment results are tightly consistent with our theoretical analysis in Table 1: OpenStack Swift needs $O(n)$ time to execute MOVE and RMDIR operations, while DP (highly probably for Dropbox) and H2 only need $O(1)$ time.

LIST and COPY. The operation time of LIST (with detailed information of each file) is recorded in Figure 9 and Figure 10. In Figure 9, n varies from 10 to 100,000, and in Figure 10, the number of direct children under the directory (m) varies from 10 to 100,000. As shown in the above two figures, the operation time of LIST depends on m rather than n. Also, we observe that OpenStack Swift costs more time than Dropbox and H2Cloud on handling LIST operations. This can be directly explained by our complexity analysis in Table 1: $O(m \cdot logN)$ for OpenStack Swift, $O(m)$ for DP, and $O(m)$ for H2. On the contrary, when dealing with COPY operations the three systems exhibit quite similar performance, as demonstrated in Figure 11. This is also consistent with our complexity analysis: $O(n + logN)$ for OpenStack Swift, $O(n)$ for DP, and $O(n)$ for H2.

MKDIR and file access. Although OpenStack Swift are poor at handling the above filesystem operations, it is better at handling MKDIR and file access operations. As depicted in Figure 12, the operation time for each system to perform MKDIR is almost constant since the directory created is always empty at that time. OpenStack Swift is, in fact, the fastest. Both H2Cloud and Dropbox need more time to make a directory, but their consumed time (between 150 and 200 ms on average) is well acceptable to users.

Moreover, Figure 13 shows how the file access time changes as the directory depth (d) of an accessed file increase. Given the full path of a file, OpenStack Swift directly calculates the hash value of the full path and looks it up in a consistent hashing ring, so its file access time is stably as low as 10 ms. With regard to H2, it hashes the file path step by step instead of as a whole, so its file access time is proportional to d. According to our collected workloads, the average and maximum directory depths are 4 and 19. Hence in practice, the file access time in H2Cloud is merely 61 ms in average, which is even shorter than that of Dropbox.

As for Dropbox, its file access time seems to be constant with fluctuations according to Figure 13, while theoretically $O(d)$ according to Table 1. The gap between $O(1)$ and $O(d)$ can be explained by delving into the working principle of DP (Dynamic Partition, refer to Figure 1c). Specifically, when the filesystem is partitioned to a number of index servers, the file access time will be more like $O(d)$; otherwise, the file access time will be approaching $O(1)$ since all the d steps for locating a file are usually locally performed in one index server.

Till now, we have unraveled the time complexities of all kinds of filesystem operations for Dropbox, and we find that all the time complexities for Dropbox are basically in line with those for DP (refer to Table 1). As a result, we infer that Dropbox is highly likely to be using the DP data structure.

Storage overhead. As H2Cloud is built based on OpenStack Swift, it inevitably brings storage overhead in terms of both additional number of objects and extra size of objects. In H2Cloud, each directory or NameRing corresponds to a separate object, so the number of objects in H2Cloud is obviously larger than that in OpenStack Swift, as illustrated in Figure 14. On the other hand, the average size of a directory object or a NameRing object (less than 1 KB) is significantly smaller than that of a file object (nearly 1 MB in average). Hence, as shown in Figure 15, the extra size of objects in H2Cloud is almost negligible.

The Impact of RTT. In order to know to what extent the RTT has effect on a typical cloud filesystem, we measure it by PINGing Dropbox in real time. We set our client at Santa Cruz, California and find the average latency (measured by PING with 56 data bytes) between the host and Dropbox is 58 ms (ranging from 24 to 83 ms). We use $\alpha = \frac{Round\text{-}Trip\ Time}{Filesystem\ Operation\ Time}$, the ratio of RTT and filesystem operation time (*i.e.,* the time duration required for file access or directory operations), to indicate the influence of RTT to the overall performance. In terms of directory operations, the ratio α for H2 stays stable at about 0.2~0.3 for operations like MOVE, MKDIR, RMDIR *etc.*, and decreases from 0.2 to a negligibly small number for operations like LIST and MOVE. As for OpenStack Swift and Dropbox, the ratio is also within 0.3. This indicates that RTT does not outweigh the directory operation time, and the directory operations, compared with RTT, are main factors to determine the user experience. On the contrary, in terms of the file access operation, as the depth of a file in the directory tree increases from 0 to 20, the ratio α decreases from 2.7 to 0.3 for H2; whereas it fluctuates around 5 and 0.5 for OpenStack Swift and Dropbox, respectively. This indicates that RTT takes a great part in the overall performance, and its deterministic effect is especially prominent in the low-depth cases. In summary, the overall performance (which determines the user experience) is greatly affected by the RTT for file access operation, as long as the accessed file is not in large depth (*e.g.,* $d > 15$); whereas the filesystem operation time is the main factor for directory operations. Thus, it is especially worthwhile to concentrate on directory operations optimization.

Performance summary. All the above results show that H2Cloud is applicable to large-scale industrial scenarios due to its overall excellent/acceptable performance and affordable overhead. Compared with OpenStack Swift, H2Cloud achieves significantly faster directory operations by orders of magnitude on handling RMDIR, RENAME, MOVE, and LIST for large filesystems. File access and MKDIR are slower than those of OpenStack Swift; but for these operations, RTT is the main factor to affect the user experience. Nevertheless, they are still faster than Dropbox. Compared with Dropbox, H2Cloud achieves similar performance in all aspects without using a separate index.

6 CONCLUSION

This paper investigates the possibility of hosting users' whole filesystems in an object storage cloud, to avoid extra cost of maintaining a separate index and meanwhile enable efficient filesystem operations. In particular, to bridge the functional gap between the flatness of the object storage cloud and the hierarchy of the directory structure, we design the Hierarchical Hash (H2) data structure as well as its coupled algorithms. Also, we implement a prototype system called H2Cloud based on OpenStack Swift to embody H2. Both theoretical analysis and real-world experiments confirm the effectiveness of our solution. The results demonstrate that H2Cloud is applicable to large-scale industrial scenarios due to its fast filesystem operations and reduced maintenance overhead.

ACKNOWLEDGMENTS

This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the High-Tech R&D Program of China ("863 – China Cloud" Major Program) under grant 2015AA01A201, the National Natural Science Foundation of China (NSFC) under grants 61471217, 61432002, 61632020 and 61572475. C. Qian was supported by NSF CNS-1717948 and CNS-1750704.

REFERENCES

[1] Aliyun Object Storage Service 2018. (2018). https://intl.aliyun.com/product/oss.
[2] Amazon S3 (Simple Storage Service) 2018. (2018). http://aws.amazon.com/s3.
[3] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. 2014. SCFS: A Shared Cloud-backed File System. In *Proc. of ATC*. USENIX, 169–180.
[4] Scott A Brandt, Ethan L Miller, Darrell DE Long, and Lan Xue. 2003. Efficient Metadata Management in Large Distributed Storage Systems. In *Proc. of MSST*. IEEE, 290–298.
[5] Building a Consistent Hashing Ring (for OpenStack Swift) 2018. (2018). http://docs.openstack.org/developer/swift/ring_background.html.
[6] Camlistore 2018. (2018). https://camlistore.org.
[7] Thierry Titcheu Chekam, Ennan Zhai, Zhenhua Li, Yong Cui, and Kui Ren. 2016. On the Synchronization Bottleneck of OpenStack Swift-like Cloud Storage Systems. In *Proc. of INFOCOM*. IEEE, 1–9.
[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
[9] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. of PODC*. ACM, 1–12.
[10] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. 2012. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proc. of IMC*. ACM, 481–494.
[11] Dropbox confirms that a bug within Selective Sync may have caused data loss 2014. (2014). https://news.ycombinator.com/item?id=8440985.
[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 29–43.
[13] How a bug in Dropbox permanently deleted my 8000 photos 2014. (2014). https://news.ycombinator.com/item?id=8101579.
[14] John Howard, Michael Kazar, Sherri Menees, et al. 1988. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.
[15] John H Howard et al. 1988. *An Overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center.
[16] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. 2009. SmartStore: A new Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems. In *Proc. of SC*. ACM, 10.
[17] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. 2012. Semantic-aware Metadata Organization Paradigm in Next-generation File Systems. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 337–344.
[18] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. 2008. The XtreemFS Architecture-a Case for Object-based File Systems in Grids. *Concurrency and computation: Practice and experience* 20, 17 (2008), 2049–2060.
[19] Inside the Magic Pocket 2018. (2018). http://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket.
[20] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of STOC*. ACM, 654–663.
[21] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[22] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 18–25.
[23] Paul J Leach, Michael Mealling, and Rich Salz. 2005. A Universally Unique Identifier (UUID) URN Namespace. (2005).
[24] Zhenhua Li, Cheng Jin, Tianyin Xu, et al. 2014. Towards Network-level Efficiency for Cloud Storage Services. In *Proc. of IMC*. ACM, 115–128.
[25] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. 2013. Efficient batched synchronization in dropbox-like cloud storage services. In *Proc. of Middleware*. Springer, 307–327.
[26] Jinjun Liu, Dan Feng, Yu Hua, Bin Peng, and Zhenhua Nie. 2015. Using Provenance to Efficiently Improve Metadata Searching Performance in Storage systems. *Future Generation Computer Systems* 50 (2015), 99–110.
[27] Jake Luciani. 2012. Cassandra File System Design. *DATATAX Blog [online] http://www. datastax. com/dev/blog/cassandra-file-system-design* (2012).
[28] Micheal Moore, David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles, Sam Sampson, Shuangyang Yang, and Boyd Wilson. 2011. OrangeFS: Advancing PVFS. In *Proc. of FAST poster*. USENIX.
[29] Subramanian Muralidhar et al. 2014. f4: Facebook's Warm BLOB Storage System. In *Proc. of OSDI*. USENIX Association, 383–398.
[30] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *Proc. of FAST*. USENIX, 89–104.
[31] Fatma Özcan, Nesime Tatbul, Daniel J Abadi, Marcel Kornacker, C Mohan, Karthik Ramasamy, and Janet Wiener. 2014. Are We Experiencing a Big Data Bubble?. In *Proc. of SIGMOD*. ACM, 1407–1408.
[32] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. 2016. GlobalFS: A Strongly Consistent Multi-site File System. In *Proc. of SRDS*. IEEE, 147–156.
[33] Swapnil Patil and Garth A Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proc. of FAST*. USENIX, 13–13.
[34] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. 1994. NFS Version 3: Design and Implementation. In *USENIX Summer*. Boston, MA, 137–152.
[35] T. S. Pillai et al. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proc. of OSDI*. 433–448.
[36] Gerald Popek and Bruce J Walker. 1985. *The LOCUS Distributed System Architecture*. The MIT press.
[37] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Storage. In *Proc. of FAST*. 89–101.
[38] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A Scalable Content-Addressable Network. In *Proc. of SIGCOMM*. ACM.
[39] Sean Rhea, Russ Cox, and Alex Pesterev. 2008. Fast, Inexpensive Content-Addressed Storage in Foundation. In *Proc. of ATC*. USENIX Association, 143–156.
[40] Mahadev Satyanarayanan, James Kistler, and Kumarand others. 1990. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.* 39, 4 (1990), 447–459.
[41] Konstantin Shvachko and Yuxiang Chen. 2017. Scaling Namespace Operations with Giraffa File System. *USENIX ;log in:* 42, 2 (2017), 27–30.
[42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proc. of MSST*. IEEE, 1–10.
[43] Mandayam C Srivas et al. 2017. Map-Reduce Ready Distributed File System. (2017). US Patent App. 15/668,666.
[44] Michael Stonebraker. 2012. NewSQL: An Alternative to NoSQL and Old SQL for New OLTP Apps. *Commun. ACM* (2012), 07–06.
[45] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *Proc. of ATC*. USENIX.
[46] The Open Group Base Specifications Issue 7–IEEE Std 1003.1 2018. (2018). http://pubs.opengroup.org/onlinepubs/9699919799/.
[47] Alexander Thomson and Daniel J Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proc. of FAST*. USENIX, 1–14.
[48] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas Bressoud, and Adrian Perrig. 2003. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. of ATC*. 127–140.
[49] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. 2009. Cumulus: Filesystem Backup to the Cloud. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 14.
[50] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. 2012. Bluesky: A Cloud-backed File System for the Enterprise. In *Proc. of FAST*. USENIX, 19–19.
[51] H. Wang, R. Shea, F. Wang, and J. Liu. 2012. On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services. In *Proc. of IWQoS*.
[52] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of OSDI*. USENIX Association, 307–320.
[53] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proc. of SC*. IEEE.
[54] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System. In *Proc. of FAST*. USENIX, 17–33.
[55] Why Dropbox decided to drop AWS and build its own infrastructure and network 2017. (2017). https://techcrunch.com/2017/09/15/why-dropbox-decided- to-drop-aws-and-build-its-own-infrastructure-and-network.
[56] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. 2018. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* (2018).
[57] Yupu Zhang, Chris Dragga, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proc. of FAST*. USENIX, 119–132.