Scaling Distributed File Systems in Resource-Harvesting Datacenters*

Pulkit A. Misra[⋆] *Íñigo Goiri*[†] *Jason Kace*[†] *Ricardo Bianchini*[†]
[⋆]*Duke University* [†]*Microsoft Research*

Abstract

Datacenters can use distributed file systems to store data for batch processing on the same servers that run latency-critical services. Taking advantage of this storage capacity involves minimizing interference with the co-located services, while implementing user-friendly, efficient, and scalable file system access. Unfortunately, current systems fail one or more of these requirements, and must be manually partitioned across independent subclusters. Thus, in this paper, we introduce techniques for automatically and transparently scaling such file systems to entire resource-harvesting datacenters. We create a layer of software in front of the existing metadata managers, assign servers to subclusters to minimize interference and data movement, and smartly migrate data across subclusters in the background. We implement our techniques in HDFS, and evaluate them using simulation of 10 production datacenters and a real 4k-server deployment. Our results show that our techniques produce high file access performance, and high data durability and availability, while migrating a limited amount of data. We recently deployed our system onto 30k servers in Bing's datacenters, and discuss lessons from this deployment.

1 Introduction

Each datacenter costs billions of dollars to build, populate, and operate. Even though procuring servers dominates this cost [5], servers are often poorly utilized, especially in clusters that host interactive services [5, 10].

Resource harvesting. The co-location of useful batch workloads (*e.g.*, data analytics, machine learning) and the data they require on the same servers that run interactive services is effective at extracting more value from the servers, and at reducing the overall number of servers that must be procured. Effectively, the batch workloads can harvest the spare cycles and storage space left by the services. However, the services must be shielded from any non-trivial performance interference produced by the batch workloads or their storage accesses; prior work [20, 21, 30] has addressed this problem. At the same time, we must ensure that the services' resource requirements and management do not unnecessarily degrade batch workloads' performance or compromise the availability and durability of the batch workloads' data.

*Pulkit Misra was a summer intern at Microsoft Research.

Zhang *et al.* [30] built a resource-harvesting distributed file system for the batch workloads' data that achieves these characteristics by smartly placing the file block replicas across the servers. However, they did not address how to scale the file system for full datacenter-wide (*e.g.*, 50k servers) harvesting, which is our target.

Scaling distributed file systems. Most distributed file systems have not been designed to scale (transparently or at all) to such large sizes. For example, HDFS [2], Cosmos Store [7], and GFS [13] do not typically scale well beyond a few thousand servers, as they rely on a centralized metadata manager (with *standby* replicas). To scale beyond this size, administrators must create separate subclusters (of whatever maximum size can be efficiently handled), each running an independent manager for an independent portion of the namespace.

Unfortunately, this approach to scaling has several drawbacks. First, users are presented a partitioned view of the namespace and often have full control over which subcluster to place their folder/files in. Second, exercising this control, users may inadvertently fill up a subcluster or overload it with a high access demand. Third, to mitigate these situations, administrators must manage folder/file placement manually (via folder/file migration and/or forcing users to the more lightly used subclusters). Fourth, it is very difficult for administrators (and impossible for users) to understand the characteristics of the co-located services in each subcluster well enough to make appropriate folder/file placement decisions, especially as services are complex and numerous.

Another approach to scaling is to implement multiple, strongly consistent, *active* metadata managers, *e.g.* [1, 27]. Though no information has been published about Google's Colossus (the follow-on to GFS), we understand that it implements such managers. However, this approach also has two key drawbacks: (1) the system becomes more complex, and this complexity is only required for large installations (simpler systems that also work well for more popular smaller systems are preferrable); and (2) any software bugs, failures, or operator mistakes have a greater impact without the isolation provided by subclusters, as highlighted in [26].

Given the drawbacks of these two approaches, it is clear that a cleanly layered, automated, and manageable approach to distributed file system scalability is needed.

Our work. Thus, in this paper, we design techniques for

automatically and transparently scaling file systems to entire resource-harvesting datacenters with tens of thousands of servers. We use Zhang's replica placement algorithm within each subcluster, but focus on how to "federate" the subclusters transparently and efficiently. We achieve these high-level characteristics by inserting a layer of software between clients and metadata managers that understands the federated namespace and routes requests to the appropriate subclusters.

Moreover, our techniques seek to (1) avoid interference from co-located services; and (2) promote behavioral diversity, good performance, and good space usage across subclusters. Achieving these goals at the same time is challenging, given the large number of services and servers, and the widely varying folder/file size and access characteristics of the batch workloads.

To simplify the problem, we divide it into two parts. First, we select the servers to assign to each subcluster in a way that maximizes data availability and durability. Specifically, we use consistent hashing [18] for this new purpose, which has the added benefit of limiting data movement when resizing subclusters. Second, we assign folders/files to subclusters and efficiently migrate them when either a subcluster starts to run out of space, or a subcluster's metadata manager starts to receive an excessive amount of access load. We model this rebalancing as a Mixed Integer Linear Program (MILP) problem that is simple enough to solve efficiently. Migrations occur in the background and transparently to users.

Implementation and results. To explore our techniques concretely, we build them into HDFS and call the resulting system "Datacenter-Harvesting HDFS" or simply "DH-HDFS". We selected HDFS because (1) it is a popular open-source system that is used in large Internet companies, *e.g.* Microsoft, Twitter, and Yahoo (we are contributing our system to open source [17]); (2) our target workloads are mostly analytics jobs over large amounts of data, for which HDFS provides adequate features and performs well; and (3) many data-analytics frameworks, like Spark and Hive, can run on HDFS.

Our evaluation uses a real deployment in a production datacenter, real service and file access traces, and simulation of 10 real datacenters. The results show that our server-to-subcluster assignment prevents interference from co-located services, minimizes data movement when subclusters are added/removed, and promotes data availability and durability for batch jobs. The results also show that our folder migration policy is efficient, migrating a small percentage of the folders; just enough to manage severe space shortages or load imbalances. When combining our techniques, DH-HDFS improves durability and availability by up to 4 and 5 orders of magnitude, respectively, compared to prior approaches. Finally, the results show that the federation layer imposes little performance overhead.

Production use. We currently have 4 DH-HDFS deployments in production use in our datacenters; the largest deployment now has 19k+ servers spread across 6 subclusters. We discuss lessons from these deployments.

Implications for other datacenter types. Though we focus on resource-harvesting datacenters, some aspects of our work also apply to scenarios where the batch workloads have the same priority over the resources as the co-located services, or where there are no co-located services. Specifically, our federation architecture and techniques for folder/file mapping to subclusters with periodic migrations apply to any scenario. Our technique for server-to-subcluster mapping would work in other scenarios, but is not strictly needed.

Summary. Our main contributions are:

- We propose novel techniques for scaling distributed file systems while accounting for data durability, availability, storage capacity, and access performance in large resource-harvesting datacenters. In particular, we introduce (a) layering for transparent scalability of unmodified existing systems; (b) consistent hashing for subcluster creation; and (c) MILP-based dynamic file migration.
- We implement our techniques in HDFS to create DH-HDFS, which we have deployed in production.
- We evaluate our techniques and system, using real workloads, real experimentation, and simulation.
- We discuss lessons from DH-HDFS in production use.

2 Background and related work

2.1 Resource-harvesting datacenters

In resource-harvesting datacenters, most servers are allotted to native, often latency-critical, workloads. Because of their latency requirements, these workloads store data using their servers' local file system. We refer to these workloads as *"primary tenants"*. To improve utilization, lower priority workloads called *"secondary tenants"*, such as batch data analytics jobs, can harvest any spare capacity left idle by primary tenants. Primary tenants have priority over their servers' resources, *i.e.* a load spike may cause secondary tenants to be throttled (or even killed) and their storage accesses to be denied. Moreover, primary tenant developers own their servers' management, *i.e.* they are free to perform actions that destroy disk data. Among other scenarios, disk reimaging (reformatting) occurs when developers re-deploy their primary tenants from scratch, and when the management system tests the resilience of production services.

The resource-harvesting organization is reminiscent of large enterprises where different departments have their own budgets, without a central infrastructure group. Nevertheless, multiple Internet companies, such as Microsoft and Facebook, use this type of underlying system.

Though Google's infrastructure is fully shared, *i.e.* any workload is treated the same, large Google tenants may also request priority over their allotted resources [28].

2.2 Diversity-aware replica placement

A challenge in harvesting is protecting file block availability and durability: (1) if we store all of a block's replicas in primary tenants that load-spike at the same time, the block may become unavailable; (2) if developers or the management system reimage the disks containing all of a block's replicas in a short time span, the block may be lost. Thus, a replica placement algorithm must account for primary tenant and management activity.

Zhang's placement algorithm [30] places replicas within a single cluster (*i.e.*, a few thousand servers), while maximizing diversity: it does not allow multiple replicas of a block to be placed in any logical (*e.g.*, servers of the same primary tenant) or physical (*e.g.*, rack) server grouping that induces correlations in resource usage, disk reimaging, or failures.

We build upon Zhang's single-cluster work by creating a federation of clusters (we refer to each cluster in the federation as a subcluster). In this context, we also select which servers to assign to each subcluster, and automatically rebalance space and access load across subclusters.

2.3 Large-scale distributed data storage

Large-scale file systems. Several distributed file systems (*e.g.*, [1, 22, 27]) have been proposed for large installations. Though potentially scalable, they involve complexity and overhead in metadata management, and are hard to manage and maintain in large-scale production. Moreover, they are often optimized for general workloads, and not those of datacenter applications (*e.g.*, write-once, append-only).

For these reasons, file systems at Microsoft [2, 7], Facebook [2, 6], Twitter [2], and other datacenter operators are much simpler. They rely on a centralized metadata manager (*e.g.*, "Namenode" in HDFS) that hosts all metadata, handles all metadata accesses, and tracks the storage nodes. To scale, administrators manually partition the overall file set into independent file systems, each in a subcluster. Some systems (*e.g.*, ViewFS [3], Cosmos Store [7]) enable users to access multiple subclusters transparently, by exploiting "mount tables" that translate folder names to subclusters. However, the client-local mount tables are independent and not kept coherent. In contrast, Google's Colossus is rumored to implement multiple active metadata managers. This approach is more complex and does not benefit from the fault- and mistake-isolation provided by subclusters [26].

Rebalancing. Some systems rebalance metadata across metadata managers without subclusters, *e.g.* [1, 27].

A challenge with subclusters is that they may become imbalanced in terms of space usage and/or access load. In resource-harvesting datacenters, imbalance is also possible in the primary tenants' resource usage and management behavior; *e.g.*, changes in primary tenant disk reimaging patterns could start to harm a subcluster's durability. We are not aware of prior policies for folder/file rebalancing across subclusters.

Several works considered rebalancing within a single cluster. For example, [14] and [15] proposed balancing the access load. Considering space and access load, Singh *et al.* [24] accounted for multiple resources (*e.g.*, switches, disks) in making greedy rebalancing decisions. The built-in HDFS rebalancer can be used to manually rebalance data, and to populate newly added servers.

HDFS Federation. HDFS has an option to split the namespace (and block management) explicitly across independent metadata managers, while storing data in any server [12]. This approach does not involve subclusters, but exposes multiple namespaces that users must manage manually [3], and limits scaling as all servers still heartbeat to all managers. *Our system is quite different,* and should not be confused with this HDFS option.

3 Federation architecture

3.1 Overview

Our architecture assumes an *unmodified* underlying distributed file system similar in structure to HDFS [2], Cosmos Store [7], and GFS [13]. It federates subclusters of the distributed file system, each defined by its own metadata manager, data storage nodes, and client library.

Each subcluster operates independently, unaware of other subclusters. This characteristic simplifies our design, and means that all replicas of a file block live in the same subcluster. As we illustrate in Figure 1, we interpose a highly available and fault-tolerant layer of software between the clients and the subclusters' metadata managers (labeled "MM" in the figure). The layer comprises (1) multiple client request routers (labeled "R"); (2) a state store that maintains a *global mount table* (*i.e.*, the folder/file-to-subcluster mappings, which we call "mount table entries" or simply "mount points") and other pieces of state about the federation; and (3) a folder/file rebalancer. Next, we detail these components.

3.2 Client request routers

The routers transparently expose a single global namespace to the clients through the standard metadata manager interface of the underlying distributed file system. Clients are unaware of the routers. A client's file access may reach any router (arrow #1 in Figure 1), as routers may sit behind a load balancer or some other request distribution mechanism. The router then consults the state store to determine the metadata manager for the proper

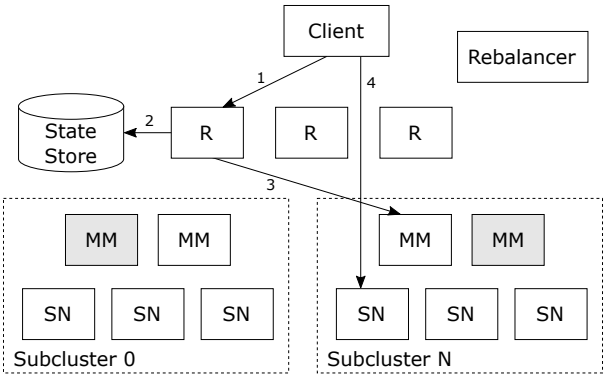

Figure 1: Federation layer comprising transparent request routers, a logically centralized state store, and a folder/file rebalancer. R = router; MM = metadata manager; SN = storage node; grey color = standby manager.

subcluster (arrow #2), and forwards the request to it (arrow #3). The reply from the manager flows back in the opposite direction. The reply lists the address of the storage nodes for all replicas of the file's blocks, so the client can communicate directly with the corresponding storage nodes on actual block accesses (arrow #4).

The routers intercept all calls. Most calls involve simply forwarding the same parameters to the proper metadata manager, perhaps after adjusting any pathnames. However, four types of calls may require additional processing: renames, deletes, folder listings, and writes. Routers fail any renames or deletes of mount points, like in other file systems (*e.g.*, Linux). Renames or deletes of folders/files that only affect one subcluster can simply be forwarded to the proper metadata manager. We handle renames of folders/files that affect multiple subclusters by performing the rename in the state store (*i.e.*, creating a new mount table entry) and having the rebalancer migrate the data later. Importantly, the routers "lock" the federated namespace during these renames to prevent inadvertent cycles [11]. Folder listing involves contacting the parent folder's subcluster, and including any mount points under the same parent folder. Finally, routers may fail folder/file writes during short periods, to guarantee consistency (*e.g.*, during rebalancing operations, as we discuss in Section 3.4).

To avoid frequent communication with the state store, the routers *cache* the folder/file-to-subcluster mappings locally. The router's cache entries may become stale, as a result of rebalancing or of losing contact with the state store for a period of time. To prevent uses of stale entries, we ensure all routers acknowledge mount table changes, and check the state store for freshness of their entries.

Dependability. Other than the disposable cache state, routers are stateless and can be replaced or restarted for high availability and fault tolerance. The routers send heartbeats to the state store, including information about metadata managers and subclusters. If a router cannot heartbeat for a period T, it enters a "safe" mode (no accesses allowed), and the other routers take on the full metadata access load. If a router does not update its status for a period $2T$, any locks it holds are taken away.

The router uses standard HDFS interfaces to query the availability of the redundant managers and the space available in the subcluster. For dependability, we associate routers with overlapping sets of metadata managers, and resolve conflicts using quorum techniques.

3.3 State store

The store maintains four pieces of state about the federation: (1) the global mount table; (2) the state of the routers; (3) the access load, available space, and availability state of the metadata managers/subclusters; and (4) the state of rebalancing operations (Section 3.4).

The mount table contains explicit mappings of folders and files to subclusters. For example, there could be a mapping from folder `/tmp/` to subcluster 3 in folder `/3/tmp/` in the federated namespace. Only the system administrators or the rebalancer can create or modify entries in the mount table. But, since there may be multiple concurrent accesses to it, writes to the mount table must be properly synchronized. In terms of structure, the logically centralized nature of the state store simplifies our architecture. However, for larger installations (*e.g.*, tens of active routers), the store must be physically distributed and provide strong consistency. Existing systems, *e.g.* Zookeeper [16], provide these features and can be used to implement the store. We use Zookeeper for our implementation (Section 5).

3.4 Rebalancer

Subclusters may be unbalanced in three ways: (1) the characteristics of their primary tenants are such that some subclusters do not exhibit enough diversity for high-quality replica placement; (2) the access load they receive may be skewed and overload some metadata managers or interfere with the primary tenants in some subclusters; and/or (3) the amount of data they store may be widely different, threatening to fill up some of them.

We address the first way with our server-to-subcluster mapping (Section 4.1). To address the other ways, our architecture includes a rebalancer component. The rebalancer migrates folders/files across subclusters (folders/files and subclusters are selected as discussed in Section 4.2) and then updates the mount table. The source data may be a sub-path of an existing mount point, *i.e.* the rebalancer can create new mount table entries.

Ensuring consistency. The rebalancer must ensure the consistency of the federated file system, as regular client traffic may be directed to the files it is migrating, and multiple failure types may occur. To achieve this, it first

records in the state store a write-ahead log of the operations it is about to start. As each operation completes, it updates the log to reflect the completion. A failed rebalance can be finished or rolled back using the log. The log also protects the system against inadvertently running multiple concurrent rebalancer instances: each instance checks the log before a migration, and aborts if it finds that another instance is actively altering the same part of the namespace.

Second, it takes a write lease on the corresponding mount table entries (it may need to renew the lease during long migrations) and records the state (*e.g.*, last modification time) of the entire subtree to be migrated. The lease prevents changes to the mount table points (by administrators or multiple instances of the rebalancer), but not to the source folders/files themselves by other clients.

Third, the rebalancer copies the data to the target subcluster. At the end of the copy, it checks whether the source data was modified during the copy. Via the state store, the rebalancer must instruct the routers to prevent writes to the source and target subtrees (and wait for routers to acknowledge), before it can compare the metadata for the subtrees. This prevents a client from modifying the source data after it has been inspected for recent changes, but before the mount table has been updated to point to the target subcluster. During this checking phase, clients are still able to read from the source data. If the source data is unchanged, the rebalancer updates the mount table, waits for all routers to acknowledge the change (at which point the source data can no longer be accessed), stops blocking writes to the source and target subtrees, and then removes the data from the source subcluster. If the source data was modified during the copy, the rebalancer rolls back and either re-starts the entire copy or simply re-copies the changed files. Our current implementation takes the latter approach. Similarly, a failure in any step of the rebalancer (*e.g.*, a file migration) causes a roll back. The rebalancer tries to complete the copy a few times (three times in our implementation). If these re-tries are not enough to complete the copy, the rebalancer rolls back but, this time, it blocks writes to the data before the copy starts.

Fourth, when the migration successfully completes, the rebalancer gives up the lease on the mount points.

3.5 Alternative architecture we discarded

We considered simply extending a system like ViewFS [3] with a shared mount table, but this design would not be transparent to the underlying file system; it would require changing the file system's client code to implement the functionality of our routers.

4 Federation techniques

In this section, we discuss the two tiers of techniques we propose to simplify the problem of organizing the federated file system in a resource-harvesting datacenter: (1) server-to-subcluster mapping, (2) folder/file-to-subcluster mapping and dynamic rebalancing. The first tier statistically guarantees that subclusters are diverse in terms of primary tenants' resource usage and disk reimaging behaviors. The second tier ensures that no subcluster undergoes an excessive access load or a storage space shortage due to secondary tenants, while other subclusters have available capacity. We finish the section with a discussion of alternative techniques.

4.1 Assign servers to subclusters

The components above provide the mechanisms we need to create and manage the federated file system. However, we still need a policy for assigning servers to subclusters in the first place. We have multiple goals for this policy:

1. Ensure that subcluster addition/removal (*e.g.*, when the administrator adds the servers of a large primary tenant into the harvesting infrastructure) does not cause massive data reorganization;
2. Promote network locality within subclusters;
3. Produce diversity in primary tenants' resource usage and reimaging behaviors in each subcluster for high availability and durability; and
4. Produce subclusters with balanced primary tenant storage space usage. (The rebalancer balances the subclusters with respect to secondary tenant access load and space consumption.)

To achieve these goals, we first define the number of subclusters as the total number of servers divided by the number of servers that can be efficiently accommodated by a metadata manager ($\sim$4000 servers per subcluster by default). Then, our policy leverages consistent hashing [18] of rack names for assigning server racks to subclusters. As consistent hashing is probabilistic, each subcluster is assigned multiple virtual nodes [9] on the hash ring to balance the number of servers assigned to each subcluster. Consistent hashing reduces the amount of data reorganization needed when subclusters are added to/removed – goal #1 above. We hash full racks to retain within-rack network locality (within a datacenter, there may be hundreds of racks, each with a few dozen servers and a top-of-rack switch) – goal #2. Finally, since each primary tenant is spread across racks for fault tolerance and most primary tenants are relatively small, randomizing the rack assignment to subclusters statistically produces evenly balanced diversity in primary tenant load and reimaging behaviors, as well as balanced space usage – goals #3 and #4. We are unaware of other work that has used consistent hashing for this purpose.

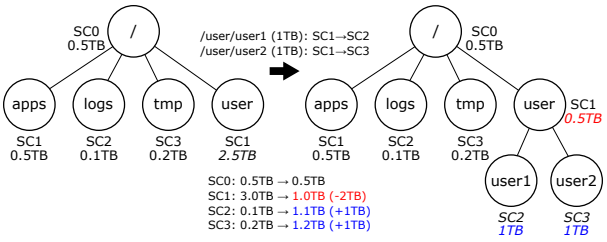

Figure 2: Example of rebalancing due to storage capacity. The file system is stored across 4 subclusters. After a rebalance, `/user` is split across 2 subclusters.

4.2 Assign/rebalance files to subclusters

Using the technique above, we now have subclusters with statistically balanced primary tenant resource usage, disk reimaging, and space usage. But we still need to assign the secondary tenants' folders/files to them, and possibly re-assign (rebalance) folders/files, when some of the current assignments are no longer appropriate.

Creation-time assignment policy. To avoid creating new mount points every time a new folder/file is created, the routers forward create operations to the same subcluster of the parent folder. This may eventually fill up some subclusters while others have plenty of free space. In addition, it may produce subclusters that receive high access loads while others receive much lower loads. We leverage rebalancing to correct these situations.

Rebalancing policy. The rebalancer wakes up periodically (*e.g.*, hourly) and compares the recent subclusters' metadata access load and free space to pre-defined watermark thresholds. We opt not to rebalance (an expensive operation) simply because the subclusters are imbalanced with respect to load or space. Instead, the thresholds define values beyond which a subcluster would be considered "under stress". Each rebalancing round tries to bring all subclusters below the watermarks. Administrators can also start a rebalancing round manually.

The rebalancer finds the subclusters' information in the state store. As the routers intercept all accesses to the metadata managers, they can easily accumulate this information and store it in the state store. (The routers cannot determine the access load imposed on a subcluster's storage nodes, only that on its metadata manager. Nevertheless, the manager is the first to overload, since it is centralized.) The routers periodically consult the managers to find out the amount of free space in each subcluster, and store it in the state store during heartbeats.

Figure 2 illustrates an example where subcluster 1 is highly loaded. The rebalancer decides to split `/user` and spread it across subclusters 2 and 3.

Rebalancing as optimization. To determine which folders/files to migrate to which subclusters, we model rebalancing as a MILP problem and use a standard solver for it. MILP is expressive enough and works well for our constraints and objectives. We are not aware of similar approaches to file system rebalancing.

We start by creating a representation of the federated namespace, where we annotate each tree node with (1) the peak amount of load it has received in any short time interval (*e.g.*, 5 minutes) since the last rebalance, (2) the current size of the subtree below it, and (3) its current subcluster. We prune nodes that exhibit lower load and lower size than corresponding administrator-defined low-end thresholds. This limits the size of the MILP problem, making it efficient to solve.

We use the pruned tree as the input to the MILP problem. The main constraints are the maximum access load a metadata manager can handle, and the maximum storage capacity of each subcluster. As its outputs, the solution produces the subcluster in which each of the tree nodes should be after rebalancing. As the objective function, we minimize a utility function combining several weighted factors: access load per subcluster, used storage capacity per subcluster, amount of data to move in the rebalance, and the number of entries in the mount table after rebalancing. The administrator is responsible for defining the weight for each factor.

Since these factors are measured in different units, we represent them as percentages over their corresponding watermarks. We introduced the access load and used capacity thresholds above. For the amount of data to move, we compute the percentage with respect to the combined size of the files that need to move to bring all subclusters below the watermarks. For the number of mount table entries, we use the percentage compared to the maximum between the number of subclusters and the number of folders in the first level of the federated namespace.

Besides its ability to derive efficient rebalances, our optimization approach is flexible in that different objective functions and constraints can be easily implemented.

4.3 Alternative techniques we discarded

Assigning servers and files to subclusters at once. For the file system organization, we considered solving the entire server and folder/file assignment problem as a large mathematical program, including primary tenants' characteristics and the federated file system. Doing so would be unwieldy; splitting the problem into two tiers of techniques makes the problem manageable.

Assigning servers to subclusters. We considered random assignment per server, per primary tenant, and per groups of primary tenants. These approaches produce subclusters with high diversity, but cause significant data movement when a subcluster is added/removed. Consistent hashing achieves diversity without this problem.

Assigning/rebalancing files to subclusters. We considered using consistent hashing of file names. There are two main problems with this approach: (1) the files in

each folder could be spread across multiple subclusters, leading to a very large mount table; and (2) a subtree rename would likely cause the entire subtree to move. Using consistent hashing of immutable file identifiers [11] would solve the latter problem but not the former.

5 Implementation and deployment

We implement our federation architecture and techniques in HDFS, and call the resulting system "Datacenter-Harvesting HDFS" or simply "DH-HDFS". We are contributing our system to open source [17].

In terms of structure and behavior, HDFS matches the underlying distributed file system in Figure 1: its metadata manager is called "Name Node" (NN) and its per-server block storage node is called "Data Node" (DN). The NN implements all the APIs of standard distributed file systems, and maintains the namespace and the mapping of files to their blocks. The (primary) NN is backed up by one or more secondary NNs. In our setup, the NN replicates each block (256 MBytes) three times by default. On a file access, the NN informs the client about the servers that store the replicas of the file's blocks. The client contacts the DN on one of these servers directly to complete the access. The DNs heartbeat to the NN; after a few missing heartbeats from a DN, the NN starts to recreate the corresponding replicas in other servers without overloading the network (30 blocks/hour/server). Within each subcluster, we use Zhang's replica placement algorithm [30] to achieve high data durability and availability in a resource-harvesting datacenter.

We place the routers behind a load balancer and configure clients (via their standard configuration files) to use the load balancer address as the NN. We implement the state store using Zookeeper [16]. At a high level, our router and state store organization purposely matches a similar architecture for YARN federation [4]. The rebalancer runs as a separate MapReduce program (one file per map task). For scalability, each DN determines its subcluster membership independently at startup time. If it needs to move to a different subcluster, the DN first decommissions itself from the old subcluster and then joins the new one. We also allow administrators to define the membership and trigger rebalances manually.

Based on our experience with the system, we define the number of subclusters as the number of servers in the datacenter divided by 4k (the largest size that HDFS handles efficiently in our setup). We set the routers to heartbeat to the state store every 10 seconds by default. In addition, we define the threshold for access load as an average 40k requests/second (near the highest throughput that an NN can handle efficiently in our setup) over any 5-minute period, and the space threshold as 80% of each subcluster's full capacity. We leverage an HDFS utility (DistCP) for copying file system subtrees. If writes occur during a copy, DistCP only re-copies the individual files written. It also transparently handles failures of NNs and DNs. We configure the rebalancer to try a subtree copy 3 times before re-trying with blocked client writes (Section 3). All settings above are configurable.

6 Evaluation

6.1 Methodology

Workloads. To represent the primary tenants, we use detailed CPU utilization and disk reimaging statistics of all the primary tenants (thousands of servers) in 10 real large-scale datacenters.[1] As our secondary tenants' file access workload, we use a real HDFS trace from Yahoo! [29]. The trace contains 700k files and 4M file accesses with their timestamps. The trace does not specify file sizes, so we assume each file has 6 blocks, for a total of 4.2M blocks (as we replicate each block 3 times, the total dataset is 3PB). The trace does not specify whether file access operations are for reading or writing, so we assume that each create operation represents a full file write and each open operation represents a full file read. This assumption is accurate for systems like HDFS, which implement write-once, read-many-times files. Overall, our trace contains 3.7M reads and 0.3M writes.

Simulator. Because we cannot experiment with entire datacenters and need to capture long-term behaviors (*e.g.*, months), we extend the simulation infrastructure from [30] to support multiple subclusters. We faithfully simulate the CPU utilization and reimaging behavior of the primary tenants, federation architecture, the techniques for server and folder/file assignment, and HDFS with diversity-aware replica placement. In the simulator, we use the same code that implements server assignment to subclusters, data placement and rebalancing in our real systems. For simplicity, we simulate each rebalance operation as if it were instantaneous (our real system experiments explore the actual timing of rebalances). The simulator replays the logs from the 10 datacenters for simulating the primary tenants' CPU utilization and disk reimages, and uses the Yahoo! trace [29] for simulating the secondary tenants' block accesses. All operations are faithfully executed based on their logged timestamps.

The simulator outputs durability (percentage of blocks retained, despite disk reimages), availability (percentage of successful accesses, despite primary tenant resource usage), usable space, access load for each subcluster, and amount of data migrated. For our durability results, we simulate 6 months of the primary tenants' reimages. For our availability results, we simulate 1 month of primary tenants' utilizations and repeat the Yahoo! trace over this

[1] Due to commercial reasons, we omit certain information, such as absolute numbers of servers and actual utilizations.

period. We run each simulation 5 times and report average results. The results are consistent across runs.

For comparison with DH-HDFS, we use a baseline system that assigns servers to subclusters randomly (per group of primary tenants), which provides high diversity per subcluster. This approach represents the manual assignment we have observed in production in the absence of DH-HDFS. The baseline assigns folder/files to subclusters in such a way that each subcluster gets three levels of the federated namespace in round-robin fashion. The baseline rebalances folders/files based on a greedy algorithm (which we adapted from [24] for the federated scenario), whenever the access load or usable space exceeds their watermark thresholds. The algorithm ranks the folders and subclusters from most to least busy (in terms of load or storage), and migrates folders from the top of the folder list to subclusters from the bottom of the subcluster list. Finally, the baseline leverages Zhang's algorithm for replica placement within each subcluster.

We also present an extensive sensitivity study, exploring the impact of the load threshold, space threshold, and rebalancing frequency in DH-HDFS.

Real experiments. We use the implementation of DH-HDFS from Section 5. We run the system on 4k servers across 4 subclusters in a production datacenter. The servers have 12-32 cores and 32-128GB of memory. Each subcluster has 4 NNs and we co-locate a router on each machine that runs a NN. We use 5 Zookeeper servers for the state store. We set the rebalancer to wake up every hour. We built a distributed trace replayer to reproduce the same load as in the Yahoo! trace.

6.2 Simulation results

We start our evaluation by isolating the impact of each feature in DH-HDFS. To conserve space, these comparisons use a single datacenter (DC-7); the other datacenters exhibit similar trends. Then, we study data durability and availability of the baseline and DH-HDFS systems across the 10 datacenters. Finally, we use a sensitivity study to quantify the impact of the load threshold, the space threshold, and the rebalancing frequency.

Within-cluster replica placement. Comparing the first two rows of Table 1 isolates the impact of the replica placement approach within each subcluster. All system characteristics other than the replica placement approach are set to the baseline system, except that we turn off rebalancing. Zhang's algorithm accounts for primary tenant behaviors, whereas stock HDFS places replicas in different racks irrespective of primary tenants. The results show that Zhang's algorithm is also effective in the federated scenario: both durability and availability improve by 4 orders of magnitude. Moreover, note that losing even a single block (*i.e.*, its 3 replicas) brings durability to six 9s ($< 100 \times 1/4.2M$) in our setup, so achieving higher durability is challenging especially as primary tenants' are free to reimage collections of disks at will.

Study	Version	Dur.	Avail.
Within-subcluster replica placement	Stock HDFS	two 9s	one 9
	Diversity-aware [30]	six 9s	five 9s
Server-to-subcluster assignment	Random per primary group	two 9s	two 9s
	Random per server	six 9s	five 9s
	Random per rack	six 9s	five 9s
	Consistent hashing per rack	six 9s	five 9s
Folder-to-subcluster assignment	Round-robin per subtree (RR)	six 9s	five 9s
	RR + rebalancing from [24]	two 9s	zero 9s
	RR + our rebalancing	six 9s	five 9s

Table 1: Simulation results for DC-7.

Server-to-subcluster assignment. The next set of rows compare approaches for assigning servers to subclusters. Again, features other than server-to-subcluster assignment are those of the baseline system without rebalancing. The results show that random per primary tenant group, which groups together all primary tenants that have related functionality, performs poorly. Due to their close relationship and potentially large size, these groups do not produce enough diversity even under Zhang's algorithm. The other three approaches achieve good results, as they leverage finer grain randomization and thus benefit from primary tenant diversity.

Consistent hashing has the additional advantage of requiring limited data movement as a result of subcluster additions/removals. For example, if we were to add a new subcluster to DC-7, only 5.5% of the data would move to populate it. In contrast, the random per rack approach would move 44% of the data. On the other hand, if we were to remove the subcluster with the most data, consistent hashing would require 20% of the data to move, while random per rack would move 68% of it.

Folder-to-subcluster assignment. The following set of rows compare techniques for assigning folders to subclusters; all other features are those of the baseline system. The results show that our rebalancing approach improves performance (not shown in the table) at the same time as retaining high durability and availability. Specifically, without rebalancing, some subclusters are exposed to extremely high load; at peak, 142k accesses/second over a period of 5 minutes. With our approach, the peak load on any subcluster goes down to 38k accesses/second after rebalancing, just under our watermark threshold of 40k accesses/second. To achieve this, our rebalancer migrates 24TB of data. In contrast, the greedy rebalancer achieves a peak of 37k accesses/second, but migrates 84TB of data. Worse, this rebalancer degrades durability and availability significantly, as it does not consider the diversity of primary tenants in the lightly loaded subclusters. Had we assumed consistent hashing (instead of random per primary tenant group) for the server assignment in this comparison, the greedy rebalancer would not have degraded durability and availability, but would still

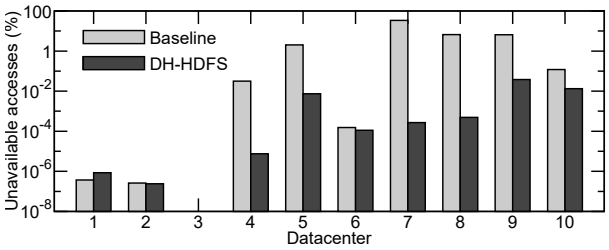

Figure 3: Data availability for baseline and DH-HDFS for 10 datacenters. The Y-axis is in log scale.

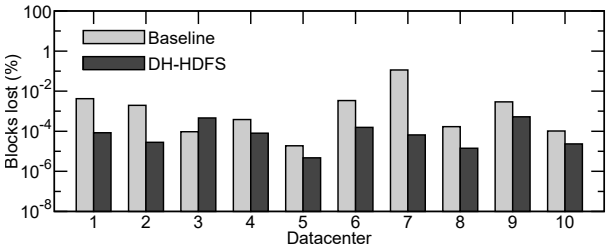

Figure 4: Data durability for baseline and DH-HDFS for 10 datacenters. The Y-axis is in log scale.

have moved 3.5× more data than our rebalancer.

Comparing baseline and DH-HDFS. Figure 3 quantifies the data availability (in percentage of failed accesses) of the baseline and DH-HDFS systems for our 10 real datacenters. The Y-axis in the figure is in log scale; a missing bar means that there were no failed accesses. To study a spectrum of utilization scenarios, we adjust the primary tenants' workloads (via acceleration/deceleration of their primary tenants' utilizations) to produce 3 groups of datacenters: the three leftmost datacenters exhibit low average primary tenant utilization (roughly 25% of the available resources), the four next datacenters exhibit mid-range average utilizations (roughly 50%), and the three rightmost datacenters exhibit high average utilizations (roughly 75%).

These results show that both systems exhibit negligible unavailability (> seven 9s availability) for the datacenters with low average utilization. For the mid-range datacenters, DH-HDFS improves availability by up to 5 orders of magnitude for three of them, while it matches the already high availability of the fourth (DC-6). The three high-utilization datacenters pose the greatest challenge to data availability. Still, DH-HDFS produces greater availability for all of them.

Figure 4 quantifies our datacenters' durability (in percentage of lost blocks) in the same order. Again, the Y-axis is in log scale. DH-HDFS exhibits greater durability than the baseline system by up to 4 orders of magnitude. The exception is DC-3 for which the baseline system produces slightly greater durability. The reason for this result is that consistent hashing provides statistical guarantees only. In exceptional cases, it may behave worse than assigning servers to subclusters by groups of primary tenants. We verified this by changing the hashing slightly to produce a different assignment, which makes our durability better than the baseline's.

Across all datacenters, our rebalancer migrates from 3.5× to 24× less data than the baseline's rebalancer.

Sensitivity of rebalancing to its parameters. Table 2 lists the comparisons we perform to assess the sensitivity of rebalancing to its main parameters. We show results for DC-7 (our largest production deployment), but other datacenters exhibit similar trends. Since durability and availability are not strongly affected by rebalancing (consistent hashing and Zhang's replica placement [30] are the dominant factors), we do not include these statistics. Instead, the table includes the range of subcluster peak loads ("Load range"), the range of subcluster space usage ("Space range"), and the amount of data migrated during rebalancing ("Data moved").

The first three rows of the table isolate the impact of the load threshold, assuming the other parameters are fixed at their default values and spare-based rebalancing is turned off. Looking at the load range and data moved columns, we can see that setting the load threshold at the average of the peak subcluster loads produces evenly balanced subclusters. Higher thresholds (including our default value of 40k accesses/second) produce more uneven peak loads, but can be satisfied with less data migration.

The next set of rows isolate the impact of the space threshold, assuming the other parameters stay at their default values and turning off load rebalancing. The space range and data moved columns show a similar effect: when the threshold is tight, rebalancing evens out the space usage at the cost of substantial data migration. Higher thresholds produce more unevenly distributed space usage, but involve less migration.

The last row shows the impact of using both average values for load- and space-driven rebalancing, assuming other parameters at their default values. This result shows that our full rebalancer brings both the peak load and space below their thresholds.

Finally, we study the impact of the frequency with which the rebalancer wakes up (not shown), while other parameters stay at their default values. We consider waking up every 30 minutes, 1 hour, and 2 hours. The results show that, for our setup, all these frequencies produce the same statistics as in the third row of the table.

6.3 Experimental results

We start this section by presenting experimental results on the performance of the DH-HDFS routers. We then study the performance of rebalancing operations.

Router performance. To explore the limits of our router's performance, we study two scenarios: a workload dominated by block reads, and a workload with

Study	Version	Load range	Space range	Data moved
Load threshold	average (30,500 acesses / sec)	27,250 - 30,600 accesses / sec	3 - 658 TB	33 TB
	35,000 accesses / sec	21,508 - 34,457 accesses / sec	6.5 - 665 TB	26 TB
	40,000 accesses / sec	21,508 - 37,834 accesses / sec	6.5 - 665 TB	24 TB
Space threshold	average (136 TB)	3,746 - 142,573 accesses / sec	122 - 132 TB	543 TB
	2 x average	5,873 - 141,789 accesses / sec	33 - 247 TB	439 TB
	4 x average	5,953 - 141,858 accesses / sec	16 - 495 TB	190 TB
Space and load	average, average	24,500 - 31,500 accesses / sec	117 - 136 TB	554 TB

Table 2: Rebalancing results for DH-HDFS and DC-7.

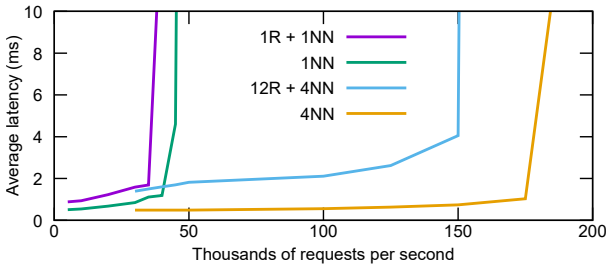

Figure 5: Performance for metadata operations.

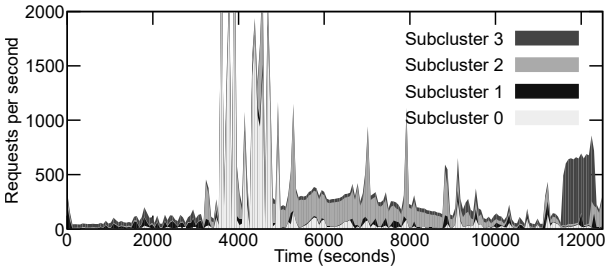

Figure 6: Subclusters' loads without rebalancing.

metadata-only operations. The former scenario is the best for our routers, and the latter is the worst; any real workload would perform between these extremes. In the best-case scenario, the performance of the routers is irrelevant. As each block is large (256MB), client-observed read latencies are measured in seconds (it takes seconds to read such a large block from disk), routers and NNs have little work to do (as clients contact DNs directly for blocks), and saturation occurs when the DNs saturate.

In the worst-case scenario, Figure 5 depicts the average metadata-only latency, as a function of load. This figure shows that one NN saturates at roughly 40k requests/second, whereas 4 NNs again saturate at roughly 4× higher load. In the small configuration, the routers add less than 1ms of latency and saturate slightly sooner. In the large configuration, the routers add up to 3ms of latency and saturate around 150k requests/second.

These results suggest that the routers perform well, adding relatively low latencies to metadata operations and negligible latencies to block accesses. Given that the latency of actual block transfers would likely dominate in real workloads, our routers should pose no significant overheads or bottlenecks in most scenarios.

Rebalancer performance. To explore the performance of rebalancing in our system, we study the Yahoo! trace when we replay it against a DH-HDFS setup with 4 subclusters and 4k servers. Figure 6 depicts the distribution of requests across the subclusters without rebalancing over time. The figure stacks the requests sent to each subcluster, representing them with different colors.

The figure shows that subcluster 0 receives a large amount of load around 4000 seconds into the execution. To demonstrate the rebalancer, we set the load watermark threshold at 2000 requests/second over any 5-minute period. As this threshold is exceeded, the rebalancer moves 4 folders of roughly the same size (400 files each) with a total of 13TB away from subcluster 0. Over repeated runs, we find that folder migrations take 354 seconds on average, 80% take less than 500 seconds, but one of them takes up to 25 minutes. Performance is especially variable when primary tenant traffic on the network is significant, *i.e.* during the weekdays. Most of the rebalancing time is spent in DistCP, with less than 34 seconds going into ensuring consistency and synchronizing the mount tables. The MILP solver takes negligible time (<100 milliseconds) to select the migrations.

These results demonstrate that the rebalancer itself is efficient, but the overall time to complete migrations can vary significantly, mainly due to primary tenant traffic. Nevertheless, recall that rebalances occur in the background and transparently to users, so the migration time variability is unlikely to be a problem.

File system performance. To illustrate the impact of the network traffic on the performance of our federated file system, Figure 7 shows Cumulative Distribution Functions (CDFs) of the performance of client-router interactions over the trace execution during a weekday (left) and during a weekend (right). The left graph shows much greater performance variability than the right one.

These results illustrate that harvesting spare resources for lower priority (secondary) workloads leaves their performance at the mercy of the primary tenants' resource demands. Most secondary workloads have lax performance requirements, so variability only becomes a problem when it is extreme. Nevertheless, if datacenter operators desire greater performance predictability for some of their secondary workloads, they must (1) account for these workloads in their resource provisioning, *e.g.* net-

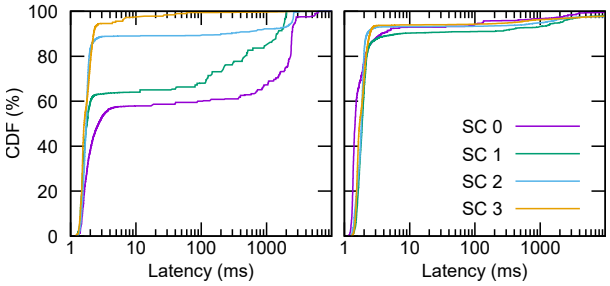

Figure 7: Client-router latency during weekday (left) and weekend (right).

work bandwidth; or (2) ensure that these workloads receive better than best-effort quality of service.

7 Lessons from production deployment

We deployed DH-HDFS in 4 production datacenters roughly 6 months ago. The deployments currently involve a total of more than 30k servers, and range from roughly 1k servers across 3 subclusters to more than 19k servers across 6 subclusters. We learned several lessons from these deployments and from the process of onboarding users onto DH-HDFS.

Server-to-subcluster assignment and bootstrapping. Once we started deploying DH-HDFS, switching from manual server assignment to consistent hashing caused many servers to switch subclusters. This implied moving large amounts of data, which produced heavy network traffic and long downtimes. To avoid this data reshuffling, the administrators introduced a new service called the *Subcluster Controller*. This component maintained the server-to-subcluster assignments and authorized (or not) servers to join a subcluster. Servers with data from a subcluster are not allowed to join a different subcluster. Once a server is reimaged or decommissioned, the controller allows it to join the new subcluster assigned through consistent hashing.

File-to-subcluster assignment and onboarding users. Before introducing DH-HDFS, users submitted their batch workloads pointing to data of one subcluster (metadata manager). To onboard workloads gradually, we deployed the routers to listen to their own RPC and HTTP ports (instead of the metadata managers' ports).

Workloads that do not yet fully leverage the single DH-HDFS namespace still want to access subclusters directly. For this reason, we added special mount points that point to the root of each subcluster.

Spreading large datasets across subclusters. Even under DH-HDFS, workloads operating on large datasets were having difficulty (1) storing all their data in a single subcluster and (2) overloading the metadata manager. One option would have been to spread the files across folders in different subclusters, but users wanted this data in a single folder. For these users, we created special mount points that span multiple subclusters. Each file within such a mount point is assigned to one of the subclusters using consistent hashing. As explained in Section 4.3, this approach adds additional complexity for renaming. For this reason, we disallow renames and restrict these special mount points to certain workloads.

Rebalancing and administrators. Currently, the rebalancer is a service triggered by the administrator. It collects the space utilization and access statistics, and proposes which paths to move across subclusters. Our design expected paths to be unique across the namespace. However, administrators created multiple mount entries pointing to the same physical data (in the same subcluster). In this case, the federated namespace had loops and counted multiple times the same physical entity. In addition, we had the special mount points (*i.e.*, subcluster roots and folders spread across subclusters), which made the namespace even more complex. To handle these situations when collecting the statistics, we modified the rebalancer to (1) ignore the special mount points; and (2) map all aliases to a single federated location. For example, if `/tmp/logs` and `/logs` both point to `/logs` in subcluster 0, we assign all the accesses to just one path.

Performance in production. Our largest deployment has 24 routers for 6 subclusters, and typically runs large data analytics workloads on an index of the Web. The load across the routers and their latency are fairly even. The latency of the routers is around 3 milliseconds, whereas the latency of the metadata managers is around 1 millisecond. These match the latencies from Section 6.

For this deployment, we use a 5-server Zookeeper ensemble for the state store. On average, a router sends 5 requests to the store every 10 seconds. This is a low load compared to the other services that use the ensemble.

8 Conclusions

In this paper, we proposed techniques for automatically and transparently scaling the distributed file systems used in commercial datacenters. We focused on systems where interactive services and batch workloads share the same servers, but most of our work also applies to dedicated servers. Our results show that our techniques introduce little overhead, and our system behaves well even in extreme scenarios. We conclude that it is possible to scale existing systems to very large sizes in a simple and efficient manner, while exposing a single namespace.

Acknowledgments

We would like to thank Bing's Multitenancy team, Chris Douglas, Carlo Curino, and John Douceur for many suggestions and discussions about this work, their comments to our paper, and help open-sourcing our system and deploying it for production use.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI* (2002).

[2] APACHE FOUNDATION. HDFS Architecture Guide, 2008. `http://hadoop.apache.org/docs/current/hdfs_design.html`.

[3] APACHE FOUNDATION. ViewFs Guide, 2016. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.html`.

[4] ARENE, M., CHALIPARAMBIL, K., CURINO, C., DOUGLAS, C., FUMAROLA, G. M., HEDDAYA, S., KRISHNAN, S., RAMAKRISHNAN, R., RAO, S., SAKALANAGA, S., SHAH, R., SHI, B., AND ZHOU, B. Enable YARN RM Scale Out via Federation using Multiple RM's. `https://issues.apache.org/jira/browse/YARN-2915`.

[5] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[6] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., VAJGEL, P., ET AL. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI* (2010).

[7] CHAIKEN, R., JENKINS, B., LARSON, P. Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB* (2008).

[8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. BigTable: A Distributed Storage System for Structured Data. In *OSDI* (2006).

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP* (2007).

[10] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS* (2014).

[11] DOUCEUR, J. R., AND HOWELL, J. Distributed Directory Service in the Farsite File System. In *OSDI* (2006).

[12] FOUNDATION, A. HDFS Federation, 2016. `https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/Federation.html`.

[13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. T. The Google File System. In *SOSP* (2003).

[14] HILDRUM, K., DOUGLIS, F., WOLF, J. L., YU, P. S., FLEISCHER, L., AND KATTA, A. Storage Optimization for Large-Scale Distributed Stream-Processing Systems. *Transactions on Storage (TOS) 3*, 4 (2008), 5.

[15] HSIAO, H. C., CHUNG, H. Y., SHEN, H., AND CHAO, Y. C. Load Rebalancing for Distributed File Systems in Clouds. *TPDS 24*, 5 (2013), 951–962.

[16] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC* (2010).

[17] KACE, J., AND GOIRI, I. Router-based HDFS federation, 2017. `https://issues.apache.org/jira/browse/HDFS-10467`.

[18] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC* (1997).

[19] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. In *LADIS* (2009).

[20] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving Resource Efficiency at Scale. In *ISCA* (2015).

[21] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing Utilization in Modern Warehouse Scale Computers Via Sensible Co-Locations. In *MICRO* (2011).

[22] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A Read/Write Peer-to-Peer File System. In *OSDI* (2002).

[23] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware* (2001).

[24] SINGH, A., KORUPOLU, M., AND MOHAPATRA, D. Server-Storage Virtualization: Integration and

Load Balancing in Data Centers. In *Supercomputing* (2008).

[25] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM* (2001).

[26] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *EuroSys* (2015).

[27] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI* (2006).

[28] WILKES, J. Private communication, 2016.

[29] YAHOO! Yahoo! Research Webscope Program, 2008. `https://webscope.sandbox.yahoo.com/`.

[30] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI* (2016).