Everyone Loves File: Oracle File Storage Service

BRADLEY C. KUSZMAUL, MATTEO FRIGO, JUSTIN MAZZOLA PALUSKA, and ALEXANDER (SASHA) SANDLER, Oracle

Oracle File Storage Service (FSS) is an elastic filesystem provided as a managed NFS service. A pipelined Paxos implementation underpins a scalable block store that provides linearizable multipage limited-size transactions. Above the block store, a scalable B-tree holds filesystem metadata and provides linearizable multikey limited-size transactions. Self-validating B-tree nodes and housekeeping operations performed as separate transactions allow each key in a B-tree transaction to require only one page in the underlying block transaction. The filesystem provides snapshots by using versioned key-value pairs. The system is programmed using a nonblocking lock-free programming style. Presentation servers maintain no persistent local state making them scalable and easy to failover. A non-scalable Paxos-replicated hash table holds configuration information required to bootstrap the system. An additional B-tree provides conversational multi-key minitransactions for control-plane information. The system throughput can be predicted by comparing an estimate of the network bandwidth needed for replication to the network bandwidth provided by the hardware. Latency on an unloaded system is about 4 times higher than a Linux NFS server backed by NVMe, reflecting the cost of replication. FSS has been in production since January 2018 and holds tens of thousands of customer file systems comprising many petabytes of data.

CCS Concepts: • **Information systems → Distributed database transactions**; **B-trees**; **Distributed storage**; Parallel and distributed DBMSs; • **Networks → Network File System (NFS) protocol**; • **Computer systems organization → Cloud computing**; • **Software and its engineering** → *File systems management*;

Additional Key Words and Phrases: Distributed filesystem, B-tree-based filesystem, Paxos, two-phase commit, cloud filesystem

ACM Reference format:
Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. 2020. Everyone Loves File: Oracle File Storage Service. *ACM Trans. Storage* 16, 1, Article 3 (March 2020), 29 pages.
https://doi.org/10.1145/3377877

1 INTRODUCTION

This article describes Oracle's File Storage Service (FSS), a managed, multi-tenanted NFS service in Oracle's Cloud Infrastructure. FSS, which has been in production for over two years, provides customers with an elastic NFSv3 file service [17]. Customers create filesystems that are initially empty, without specifying how much space they need in advance, and write files on demand. The performance of a filesystem grows with the amount of data stored. We promise customers a convex

Authors' address: B. C. Kuszmaul, M. Frigo, J. Mazzola Paluska, and A. (Sasha) Sandler, Oracle, 95 Network Drive, Burlington, MA, Massachusetts, 01803; emails: kuszmaul@gmail.com, athena@fftw.org, jmp@justinmp.com, sasha.sandler@oracle.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1553-3077/2020/03-ART3 $15.00
https://doi.org/10.1145/3377877

RIGHTSLINK

combination of 100 MB/s of bandwidth and 3,000 operations per second for every terabyte stored. Customers can mount a filesystem on an arbitrary number of NFS clients. The size of a file or filesystem is essentially unbounded, limited only by the practical concerns that the NFS protocol cannot cope with files bigger than 16 EiB and that we would need to deploy close to a million hosts to store multiple exabytes. FSS provides the ability to take a snapshot of a filesystem using copy-on-write techniques. Creating a filesystem or snapshot is cheap, so that customers can create thousands of filesystems, each with thousands of snapshots. The system is robust against failures, since it synchronously replicates data and metadata five-ways using Paxos [50].

We built FSS from scratch. We implemented a Paxos-replicated block store, called DASD, with a sophisticated multipage transaction scheme. On top of DASD, we built a scalable B-tree with multikey transactions programmed in a lockless nonblocking fashion. Like virtually every B-tree in the world, ours is a B+-tree. We store the contents of files directly in DASD and store file metadata (such as inodes and directories) in the B-tree.

Why not do something simpler? One could imagine setting up a fleet of storage appliances. Each appliance would be responsible for some filesystems, and we could use a replicated block device to achieve reliability in the face of hardware failure. Examples of replicated block devices abound [2, 4, 29, 61, 68]. We have such a service in our cloud, so why not use it? It is actually more complicated to operate such a system than a system that is designed from the beginning to operate as a cloud service. Here are some of the problems you would need to solve:

- How do you grow such a filesystem if it gets too big to fit on one appliance?
- How do you partition the filesystems onto the appliance? What happens if you put several small filesystems onto one appliance and then one of the filesystems grows so that something must move?
- How do you provide scalable bandwidth? If a customer has a petabyte of data, then they should get 100 GB/s of bandwidth into the filesystem, but a single appliance may have only a 10 Gbit/s network interface (or perhaps two 25 Gbit/s network interfaces).
- How do you handle failures? If an appliance crashes, then some other appliance must mount the replicated block device, and you must ensure that the original appliance does not restart and continue to perform writes on the block device, which would corrupt the filesystem.

This article describes our implementation. Section 2 provides an architectural overview of FSS. The article then proceeds to explain the system from the top down. Section 3 describes the lock-free nonblocking programming style we used based on limited-size multipage transactions. Section 4 shows how we organize metadata in the B-tree. Section 5 explains how we implemented a B-tree key-value store that supports multikey transactions. Section 6 explains DASD, our scalable replicated block storage system. Section 7 describes our pipelined Paxos implementation. Section 8 discusses congestion management and transaction-conflict avoidance. Section 9 describes the performance of our system. Section 10 describes some of the foundations of our control plane. Sections 11 and 12 conclude with a discussion of related work and a brief history of our system.

2 FSS Architecture

This section explains the overall organization of FSS. We provision many hosts, some of which act as storage hosts and some as presentation hosts. The storage hosts, which include local NVMe solid-state-drive storage, store all filesystem data and metadata replicated five-ways,[1] and provide a remote-procedure-call (RPC) service using our internal FSS protocol. The presentation hosts speak the standard NFS protocol and translate NFS into the FSS protocol.

[1]Data are erasure coded, reducing the cost to 2.5, see Section 3.

RIGHTSLINK

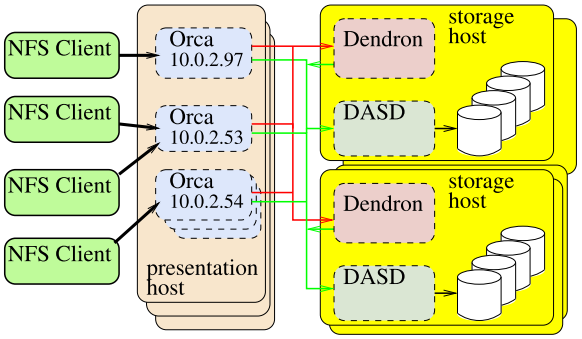

Fig. 1. FSS architecture. The NFS clients are on the lef and belong to various customers. Hosts are shown as boxes with solid edges and processes are shown with dashed lines. The presentation hosts are in the middle, each running several Orca processes. The Orca processes are connected to the various customer virtual cloud networks (VCNs) on the left. The IP addresses of each Orca's mount target is shown. The Orca process are also connected to our internal VCN, where they can communicate with the storage hosts. The storage hosts contain NVMe drives and run both the Dendron and DASD processes.

A customer's filesystems appear as exported filesystems on one or more IP addresses, called ***mount targets***. A single mount target may export several filesystems, and a filesystem may be exported by several mount targets. A mount target appears as a private IP address in the customer's virtual cloud network (VCN), which is a customizable private network within the cloud. Most clouds provide VCNs in which hosts attached to one VCN cannot even name hosts in another VCN. Each mount target terminates on one of our presentation hosts. A single mount target's performance can be limited by the network interface of the presentation host, and so to get more performance, customers can create many mount targets that export the same filesystem.

Figure 1 shows how the FSS hosts and processes are organized. The customer sets up NFS clients in their VCN. Our presentation hosts terminate NFS connections from the clients in per-mount-target Orca processes. The Orca processes translate NFS requests into the FSS protocol and send the FSS to our storage hosts. In the future, the presentation hosts might speak other client protocols, such as SMB [62] or NFSv4 [76].

To ensure isolation between filesystems we depend on a combination of process isolation on our servers, VCN isolation, and encryption. All data stored at rest in the storage hosts or in flight in the FSS protocol is encrypted with a file-specific encryption key that derives from a filesystem master key. The NFSv3 protocol is not encrypted, however, so data arriving at an Orca is potentially vulnerable. To mitigate that vulnerabilty, we rely on VCN isolation while the data are in flight from the NFS client to the presentation host and use the presentation host's process isolation to protect the data on the presentation host. All data and file names are encrypted as soon as they arrive at an Orca, and each Orca process serves only one mount target.

Each storage host contains NVMe drives and runs two processes, DASD and Dendron. ***DASD***, described in Section 6, provides a scalable block store. ***Dendron*** implements a B-tree (Section 5) in which it maintains the metadata (Section 4) for the filesystem.

We chose to replicate filesystems within a data center rather than across data centers within a metropolitan area or across a long distance. There is a tradeoff between latency and failure tolerance. Longer-distance replication means the ability to tolerate bigger disasters but incurs longer network latencies. We chose local replication so that all of our operations can be synchronously replicated by Paxos without incurring the latency of long-distance replication. It turns out that most of our customers rely on having a functional disaster-recovery plan, and so they are more

RIGHTSLINK

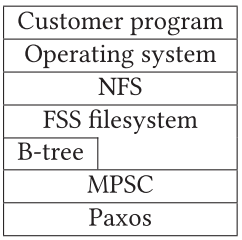

Fig. 2. Each module is built on the modules below.

interested in single-data center file system performance than synchronous replication. In the future, however, we may configure some filesystems to be replicated more widely.

Within a data center, hosts are partitioned into groups called ***fault domains***. In a small data center, a fault domain might be a single rack. In a large data center, it might be a group of racks. Hosts within a fault domain are likely to fail at the same time (because they share a power supply or network switch). Hosts in different fault domains are more likely to fail independently. We employ five-way Paxos replicated storage that requires at least three of each group of five Paxos instances to access the filesystems. We place the Paxos instances into different fault domains. When we need to upgrade our hosts, we can bring down one fault domain at a time without compromising availability. Why five-way replication? During an upgrade, one replica at a time is down. During that time, we want to be resilient to another host crashing. If we employed five fault domains, then all the fault domains would need to be exactly the same size. Instead, we employ nine fault domains, which allows us to accommodate different fault domains being different sizes.

We also use the same five-way-replicated Paxos machinery to run a non-scalable hash table that keeps track of configuration information, such a list of all the presentation hosts, needed for bootstrapping the system.

All state (including NLM locks, leases, and idempotency tokens) needed by the presentation servers is maintained in replicated storage rather than in the memory of the presentation hosts. That means that any Orca can handle any NFS request for the filesystems that it exports. The view of the filesystem presented by different Orcas is consistent.

All memory and disk space is allocated when the host starts. We never run `malloc()` after startup. By construction, the system cannot run out of memory at runtime. It would likely be difficult to retrofit this memory-allocation discipline into old code, but maintaining the discipline was relatively straightforward, since the entire codebase is new.

3 MULTI-PAGE STORE CONDITIONAL

FSS is implemented on top of a distributed B-tree, which is written on top of a a distributed block store with multipage transactions (see Figure 2). This section describes the programming interface to the distributed block store and how the block store is organized into pages, blocks, and extents.

The filesystem is a concurrent data structure that must not be corrupted by conflicting operations. There can be many concurrent NFS calls modifying a filesystem: One might be appending to a file, while another might be deleting the file. The filesystem maintains many invariants. One important invariant is that every allocated data block is listed in the metadata for exactly one file. We need to avoid memory leaks (in which an allocated block appears in no file), dangling pointers (in which a file contains a deallocated block), and double allocations (in which a block appears in two different files). There are many other invariants for the filesystem. We also employ a B-tree that has its own invariants. We live under the further constraint that when programming these data

RIGHTSLINK

structures, we cannot acquire a lock to protect these data structures, since if a process acquired a lock and then crashed, then it would be tricky to release the lock.

To solve these problems, we implemented FSS using a nonblocking programming style similar to that of transactional memory [39]. We use a primitive that we call ***multi-page store-conditional (MPSC)*** for accessing pages in a distributed block store. An MPSC operation is a "mini-transaction" that performs an atomic read-and-update of up to 15 pages. All page reads and writes follow this protocol:

(1) Read up to 15 pages, receiving the page data and a ***slot number*** (which is a form of a version tag [44, p. A-44]). A page's slot number changes whenever the page changes. You can read some pages before deciding which page to read next, or you can read pages in parallel. Each read is linearizable [40].
(2) Compute a set of new values for those pages.
(3) Present the new page values, along with the previously obtained slot numbers, to the MPSC function. To write a page requires needs a slot number from a previous read.
(4) The update will either succeed or fail. Success means that all of the pages were modified to the new values and that none of the pages had been otherwise modified since they were read. A successful update linearizes with other reads and MPSC updates. A failure results in no changes.

In addition to reading and writing pages, an MPSC can allocate or free space in the distributed block store.

An MPSC could fail for many reasons. For example, if, between reading a page and attempting an MPSC, some other transaction wrote the page, the MPSC will fail. Even if there is no conflict, an MPSC may fail due to, e.g., packet loss or Paxos leadership changes. Even if a transaction succeeds, the caller may receive an error indication, e.g., if network fails between the update's commit and the caller notification. Our implementation deliberately introduces failures (sometimes called fuzzing [63]) with a small probability rate, so that all of the error-handling code paths are exercised frequently, even in production.

3.1 Pages and Blocks

We subdivide the distributed block store into a hierarchy of pages, blocks, and extents, as shown in Figure 3. MPSC performs an atomic update on a set of ***pages***. A ***block*** includes one or more pages and is the unit on which we do bookkeeping for allocation. To reduce bookkeeping overheads on small pages, we allocate relatively large blocks. To keep transactions small, we update relatively small pages. An ***extent*** is an array of pages, up to 256 GiB total and is implemented by a replicated Paxos state machine.

For example, one kind of extent contains 256 GiB of disk-resident data, organized in 2 MiB blocks with 32 KiB pages and is replicated five ways using 5:2 erasure coding (an erasure-coding rate of 2/5) [70]. Thus the 256 GiB of disk-resident data consumes a total of 640 GiB of disk distributed across five hosts.

An extent's ***geometry*** is defined by its page size, block size, extent size, replication factor, and erasure-coding rate. Once an extent is created, its geometry cannot change. Figure 4 shows the extent geometries that we use for file data and metadata. All of our extents are five-way replicated within a single data center. The pages in extents used for file contents are erasure coded using a 5:2 erasure coding rate, so that the overhead of storing a page is 2.5 (each replica stores half a page, and there are five replicas). The B-tree data are mirrored, which can be thought of as 5:1 erasure coding.

RIGHTSLINK

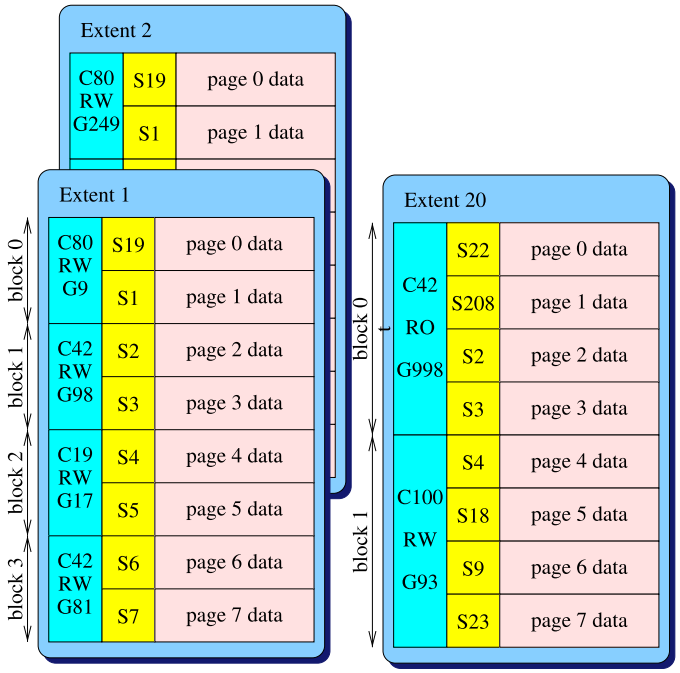

Fig. 3. Pages, blocks, and extents. Three extents are shown, each with an array of pages. Each page has a slot, e.g., page 0 of extent 1 has slot 19. Each block has ownership. The first block of extent 1 is owned by customer 80 ("C80"), is read-write ("RW"), and is on its 9th allocation generation ("G9"). Extents 1 and 2 each have 2 pages per block and 4 blocks, whereas extent 20 has 4 pages per block and only 2 blocks.

Geometry	Page size	Block size	Extent size	RF	EC
B-tree	8 KiB	1 MiB	16 GiB	5	1
8 KiB	8 KiB	8 KiB	32 GiB	5	5:2
32 KiB	32 KiB	32 KiB	128 GiB	5	5:2
256 KiB	32 KiB	256 KiB	256 GiB	5	5:2
2 MiB	32 KiB	2 MiB	256 GiB	5	5:2

Fig. 4. Extent geometries. The B-tree extents contain metadata organized as a B-tree. The other extents contain file contents and are identified by their block size. For each extent, the page size, block size, extent size, replication factor (RF), and erasure-coding (EC) are shown.

We size our extents so there are hundreds of extents per storage host to ease load balancing. We use parallelism to recover the missing shards when a host crashes permanently—each extent can recover onto a different host.

3.2 Block Ownership

When operating a storage system as a service, it is a great sin to lose a customer's data. It is an even greater sin to give a customer's data to someone else, however. To avoid the greater sin, blocks have ownership information that is checked on every access.

A block's ownership information includes a version tag, called its ***generation***, a 64-bit customer identifier, and a read-only bit. When accessing a block, the generation, customer id, and read-only

RIGHTSLINK

bit must match exactly. This check is performed atomically with every page access. When a block is allocated or deallocated its generation changes. A ***tagged pointer*** to a page includes the block ownership information, as well as the extent number and page number. A block's pointer is simply the tagged pointer to the block's first page.

The problem that block ownership solves can be illustrated as follows. When data are being written into a new file, we allocate a block and store the block's pointer in the B-tree as a single transaction. To read data from a file, Orca first obtains the block pointer by asking Dendron to read the B-tree. Orca caches that block pointer, so that it can read the data without the overhead of checking the B-tree again on every page access. Meanwhile, another thread could truncate the file, causing the block to be deallocated. The block might then be allocated to a file belonging to a different customer. We want to invalidate Orca's cached pointers in this situation, so we change the block ownership. When Orca tries to use a cached pointer to read a deallocated page, the ownership information has become invalid, and the access fails, which is what we want.

Each of our read operations is linearizable, meaning that they are totally ordered with respect to all MPSC operations and the total ordering is consistent with real time. Although our read operations linearize, if you perform several reads, then they take place at different times, meaning that the reads may not be mutually consistent. It is easy to trick a transactional-memory-style program into crashing, e.g., due to a failed assertion. For example, if you have two pages in a doubly linked list, then you might read one page, and then follow a pointer to the second page, but by the time you read the second page it no longer points back to the first page. Getting this right everywhere is an implementation challenge, leading some [12, 19] to argue that humans should not program transactional memory without a compiler. We have found this problem to be manageable, however, since an inconsistent read cannot lead to a successful MPSC operation, so the data structure is not corrupted.

4 A FILESYSTEM SCHEMA

This section explains how we represent the filesystem metadata in our B-tree. FSS implements an ***inode-based write-in-place*** filesystem using a ***single B-tree*** to hold its metadata. What does that mean? ***Inode-based*** means that each filesystem object (each file, directory, symlink, and so forth) is represented by an polymorphic object called an ***inode***. Inodes comprise a fixed-size data record that holds the fixed-size metadata about the object, and for some inode types, the inode further comprises additional information that is unbounded in size (the mapping from offsets to data blocks, for example). An object's inode contains all of the object's metadata. That is, the inode contains all the information about the object except for the file contents (the "data" of the file). In NFS file systems, each inode is named by an identifier called its handle. Every RPC in NFS takes in at least one handle to identify what filesystem objects are involved in the operation. In traditional unix filesystems, each inode has a number, sometimes called the ***inumber***. ***Write-in-place*** means that updates to data and metadata usually modify the data or metadata in the same location on disk as the old version was stored. (As we shall see, snapshots introduce copy-on-write behavior.) ***Single B-tree*** to hold the metadata means there is only one B-tree per data center. Our service provides many filesystems to many customers, and they are all stored together in one B-tree. We are taking the space to explain how to implement filesystem metadata in a B-tree, because, although there are many filesystems that use B-trees [10, 21, 26, 45, 46, 59, 71, 72, 74, 75, 82, 84, 85], there are few places to learn how they do it except to read their source code and internal design documents.

The inode represents everything the filesystem knows about a filesystem object. For example, every filesystem object contains "`stat()`" data, which include the type of the object (e.g., regular, directory, symlink), permissions bits (`rwxrwxrwx`), owner, group, file size, link count, and timestamps. Beyond the "`stat()`" data, the inode of each object type contains type-specific information.

RIGHTSLINK

A regular file's inode contains a data structure to translate between between file offsets and the filesystem blocks where data resides. Symlinks contain the text of the link. Directories contain directory entries, which name other filesystem objects. Device nodes contain major and minor device numbers indicating the type of device the node represents.

In traditional filesystems such as the Unix Fast File System [60] and the ext family of filesystems [18, 59], for each inode there is a particular page on disk where the fixed-size part of the inode is stored. The variable-sized part of the inode use additional dynamically allocated pages. The on-disk inode structures are designed to make common file system operations fast. For example, each directory inode contains directory entries that map file names to other inodes. The directory entry list is organized so that it is possible to both lookup a directory by name (the `LOOKUP` call in NFS or the `open()` system call locally) as well as iterate through directory entries using a cursor that can be represented by a single number (the `READDIR` call in NFS or the `readdir()` system call locally).

One could imagine a system that adds new entries to the end of long list, making `READDIR` simple at the cost of making `LOOKUP` slow. Or a system that uses a per-inode B-tree mapping file names to inumbers so that `LOOKUP` is quick, even for huge directories, at the cost of making iteration with `READDIR` more complicated. The challenge is to make both `READDIR` and `LOOKUP` fast.

We store all metadata in the B-tree, so we must represent all inode data as key-value pairs. How does one represent inodes as key-value pairs in a B-tree? To keep the explanation simple, we will employ a three-step approach. First, we will review inodes as an abstract data struture. Second, we will organize inodes as tables. Then, third, we will show how to represent the inode tables as B-tree key-value pairs.

A typical inode structure for part of a filesystem is shown in Figure 5. The root inode is numbered `0` and is a directory. It has two subdirectories `/home` and `/usr`. In turn `/home` contains a regular file `/home/README` that contains three 8 KiB blocks (numbered `A`, `B`, and `C`, respectively.) Thus, the pointers in the abstract data structure are represented by inumbers and block numbers.

Now consider how to represent the inode data structure as a set of tables. Since all filesystem objects contain fixed-size "stat()" data, we can employ a table that contains a row for every inode. This table contains a column for the filesystem number (fsnum) (to distinguish different filesystems in the same B-tree), a column for inumber (to identify the specific file in the filesystem), a column for the snapshot range (which we will discuss below), as well as a column for each entry in the stat structure. For example, if the four inodes in Figure 5 were all created in filesystem 42 for Snapshot 12, then the tabular representation would look like Figure 6.

Each row stores a snapshot range $[a, b)$ that covers all epochs r such that $a \leq r < b$. It turns out that we store the beginning of the range (a) in the B-tree key and the end of the range (b) in the B-tree value.

To represent type-specific metadata, we must create type-specific tables. There will be rows in the type-specific table only for inodes of that type. To form our B-tree schema, we will assign some columns of the table to the B-tree key and the rest to the value.

Figure 7 shows how the directory metadata needed for `LOOKUP` operations can be represented in a table. For `LOOKUP` we are given the inumber of a directory and a name, and must return the inumber of the object named in the directory. Directories contain named directory entries, so the key comprises the fsnum, the inumber of the directory, and the name. The value comprises the inumber of the object referred to by the the name. (It turns out that the directory names and the data are encrypted by Orca, so the actual key is the encrypted name.)

Figure 8 shows how to represent the metadata for `READDIR`. Since a directory can contain too many entries to return in a single NFS call, NFS employs a single 64-bit value (called the ***cookie***) as a cursor to indicate where to continue on a subsequent `READDIR`. Given a inumber and a cookie,

RIGHTSLINK

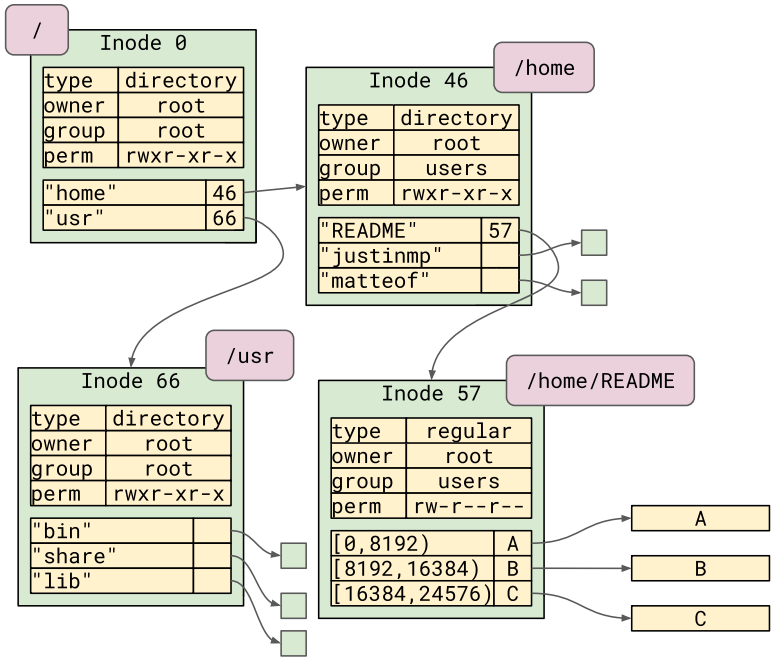

Fig. 5. The abstract inode structure for part of a filesystem. Each green box is an inode. The purple rounded box attached to the inode is the pathname of the inode. This pathname is not part of the data struture: It is an annotation to help understand the data structure. Each inode has a number (for example, "Inode 0") and a small fixed-sized table of metadata shown in yellow (the type, owner, group, and permissions are shown). Directory inodes have a mapping (that can be large) mapping names to inumbers. Regular fles have a (potentially large) mapping of file offsets to blocks. In the figure, blocks are shown as yellow boxes labeled A, B, and C.

Fixed-sized inode metadata						
fsnum	inumber	snapshot	type	owner	group	perm
⋮	⋮	⋮	⋮	⋮	⋮	⋮
42	0	$[12, \infty)$	directory	root	root	rwxr-xr-x
42	46	$[12, \infty)$	directory	root	users	rwxr-xr-x
42	57	$[12, \infty)$	regular	root	users	rw-r--r--
42	66	$[12, \infty)$	directory	root	root	rwxr-xr-x
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 6. Table of "stat" information for the inodes shown in Figure 5. All of the inodes belong to the same filesystem (number 42). Each row can be thought of as having a key and a value (which are separated by a vertical line). The filesystem number ("fsnum"), inumber, and snapshot number are the unique key for each row of the table, and we imagine that the table is kept in sorted order. The `type`, `owner`, `group`, and `perm` are all part of the value. The snapshot provides a range, and it turns out that the exclusive upper bound of the range is storedin the value, whereas the lower bound is stored in the key.

RIGHTSLINK

LOOKUP Directory Metadata				
fs id	dir inumber	name	snapshot	inumber
⋮	⋮	⋮	⋮	⋮
42	0	home	$[12, \infty)$	46
42	0	usr	$[12, \infty)$	66
42	46	README	$[12, \infty)$	57
⋮	⋮	⋮	⋮	⋮

Fig. 7. Table of LOOKUP directory entries for the inodes shown in Figure 5.

READDIR Directory Metadata					
fs id	dir inumber	cookie	snapshot	name	inumber
⋮	⋮	⋮	⋮	⋮	⋮
42	0	1	$[12, \infty)$	home	46
42	0	2	$[12, \infty)$	usr	66
42	46	3	$[12, \infty)$	README	57
⋮	⋮	⋮	⋮	⋮	⋮

Fig. 8. Table of READDIR directory entries for the inodes shown in Figure 5.

Regular File Block Map Metadata				
fs id	inumber	offset	snapshot	Block (a tagged pointer)
⋮	⋮	⋮	⋮	⋮
42	57	0	$[12, \infty)$	A
42	57	8192	$[12, \infty)$	B
42	57	16384	$[12, \infty)$	C
⋮	⋮	⋮	⋮	⋮

Fig. 9. Table of file block information for select inodes of Figure 5.

a READDIR implementation must return the next several directory entries, so we put the fsnum, directory inumber, and cookie in the B-tree key and the directory entry in the value.

Figure 9 shows how the block mapping information for a regular file can be represented in a table. Regular files contain a mapping from offsets to block numbers, so this table's key is the fsnum, the inumber, and the offset. The value comprises a tagged pointer that identifies the block containing the data.

Since we employ different sized blocks at different offsets, it is convenient that the block map is kept in sorted order: We can easily find the block map entry that holds the data for an arbitrary offset rather than requiring a lookup for exactly the right offset. For example, if we want to know which block holds byte 10,000 in the README file in this example, then we can discover that it is Block B, since its offset, 8,192, is the largest offset that is less than or equal to 10,000 whose length covers byte 10,000.

So far we have not explained snapshots. A read-only copy of a filesystem is called a snapshot. We employ copy-on-write to implement snapshots so that the cost of a snapshot is only the cost of whatever changes are subsequently made to the filesystem. Snapshots have names and appear

RIGHTSLINK

Fixed-sized inode metadata after a snapshot						
fs id	inumber	snapshot	type	owner	group	perm
⋮	⋮	⋮	⋮	⋮	⋮	⋮
42	0	[12, ∞)	directory	root	root	rwxr-xr-x
42	46	[12, ∞)	directory	root	users	rwxr-xr-x
42	57	[12, 13)	regular	root	users	rw-r--r--
42	57	[12, ∞)	regular	justinmp	users	rw-r-----
42	66	[12, ∞)	directory	root	root	rwxr-xr-x
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 10. We started with the fixed-sized data of Figure 6 which was created in Epoch 12, and then in Epoch 13 we changed the owner and permissions of Inode 57. The original row for Inode 57 is unchanged, and a new row for Epoch 13 has been created.

in a special `.snapshots` directory. Users can create snapshots by creating a subdirectory in the `.snapshots` directory (e.g., with `mkdir .snapshots/foo`) or via the control-plane. Snapshots can be deleted, which will end up freeing blocks. Users might create snapshots using a `cron` job (e.g., for backups) or as part of other administrative activity (e.g., just before installing a software upgrade).

Snapshots work as follows. Each of the tables shown above has a column in the key for the snapshot range. We maintain a snapshot number, called the ***epoch*** of each filesystem. If a client takes a snapshot of the filesystem, then we increment the epoch. We never modify rows that have snapshot numbers less than the epoch. Instead, we copy the relevant row with a new snapshot number and modify that row. In our example, all the files and data were created in Epoch 12. If we take a snapshot, then we are now in Epoch 13. If we now modify the permissions on the `README` file so that it is owned by `justinmp` and not world readable, then the fixed size table ends up as shown in Figure 10.

The situation for READDIR directory entries is also more complicated due to snapshots. Given a key that has everything but the snapshot, we can employ a "find-next" or "find-prev" operation to find the right record for a given snapshot, and this even works for block maps. But for READDIR, we need to do a "find-next," but we do not know the cookie part of the key. If, for example, there are a lot of directory entries that existed during Epoch 12, but have been deleted in Epoch 13, then we could end up looking through a lot of table entries to find a suitable entry. To provide a guarantee that we can return at least one `READDIR` result per B-tree operation we maintain a doubly linked list of `READDIR` entries.

Figure 11 shows an example of a series of operations in the `home` diretory (the directory inumber is 46). Whenever a row's next or prev records need to be updated, it may split a row into two rows to cover two different snapshot ranges. For example, in the transition from (b) to (c), the row for `B` was split, because in Snapshot 13, there is no next element after `B`, but in Snapshot 14, there is a next element. Similarly, in (d) we have deleted `B` so the next link of `A` and the prev link of `C` depends on which snapshot we are operating in.

At this point, we are done except for one consideration: scale. In all of the above pairs, we use inode numbers to identify specific files. Every inode number must be unique. For busy filesystems that create many files per second, the inode number allocator could become a bottleneck. Rather than represent inode numbers as a single number, we use a pair of numbers consisting of a unique directory identifier pulled from a global allocation pool and a sequence number within the directory where the file was created. This scheme lets us allocate non-directory inode "numbers" using a per-directory sequence number allocator. While directories still require FSS to allocate directory

RIGHTSLINK

READDIR Directory Metadata - With Linked list							
fs id	dir inumber	cookie	snapshot	name	inumber	prev	next
42	46	1	$[12, \infty)$	A	101	$\perp$	$\perp$

(a) File A was created in epoch 12.

READDIR Directory Metadata - With Linked list							
fs id	dir inumber	cookie	snapshot	name	inumber	prev	next
42	46	1	$[12, 13)$	A	101	$\perp$	$\perp$
42	46	1	$[13, \infty)$	A	101	$\perp$	2
42	46	2	$[13, \infty)$	B	102	1	$\perp$

(b) Create B in epoch 13. In Epoch 12, A's next is $\perp$, but in Epoch 13 the next is 2.

READDIR Directory Metadata - With Linked list							
fs id	dir inumber	cookie	snapshot	name	inumber	prev	next
42	46	1	$[12, 13)$	A	101	$\perp$	$\perp$
42	46	1	$[13, \infty)$	A	101	$\perp$	2
42	46	2	$[13, 14)$	B	102	1	$\perp$
42	46	2	$[14, \infty)$	B	102	1	3
42	46	3	$[14, \infty)$	C	103	2	$\perp$

(c) Create C in epoch 14.

READDIR Directory Metadata - With Linked list							
fs id	dir inumber	cookie	snapshot	name	inumber	prev	next
42	46	1	$[12, 13)$	A	101	$\perp$	$\perp$
42	46	1	$[13, 15)$	A	101	$\perp$	2
42	46	1	$[15, \infty)$	A	101	$\perp$	3
42	46	2	$[13, 14)$	B	102	1	$\perp$
42	46	2	$[14, 15)$	B	102	1	3
42	46	3	$[14, 15)$	C	103	2	$\perp$
42	46	3	$[15, \infty)$	C	103	1	$\perp$

(d) Delete B in epoch 15.

Fig. 11. READDIR directory with doubly linked list for each snapshot. In each row, the "prev" (and "next") values are the previous (and next) cookie that is appropriate for the snapshot described by the row. The figure shows a series of states.

ids from a filesystem-wide allocator, most workloads create orders of magnitude fewer directories than files. Similarly, rather than using a filesystem-wide READDIR cookie, we allocate READDIR cookies out of a per-directory sequence number. Both per-directory sequence numbers are stored in the "stat()" metadata of the inode B-tree values.

Figure 12 shows the schema for representing our filesystems. We encode the B-tree keys in a way that disambiguates the different kinds of key value pairs and sorts the key-value pairs in a convenient order. For example, all the pairs for a given filesystem F appear together, with the superblock appearing first because of the "0" in its key. Within the filesystem, all the non-superblock pairs are sorted by D, the directory number. For a directory, the directory inode sorts first, then come the name map entries for the directory, and then the cookie map, then come all the inodes for the files that were created in that directory. In that set for each file, the file inode sorts first, followed by the block maps for that file. Finally, two entries that are the same except for the snapshot number are sorted by snapshot number.

RIGHTSLINK

Key-value pairs:

leaderblock:	$0 \rightarrow$ next F.
superblock:	$F, 0 \rightarrow$ next D, next C, keys.
inode:	$F, 1, D, C, 2, S \rightarrow$ stat-data.
name map:	$F, 1, D, C = 0, 3, N, S \rightarrow F, D', C', S$.
cookie map:	$F, 1, D, C = 0, 4, c, S \rightarrow F, D', C', S, N$.
block map:	$F, 1, D, C, 5, o, S \rightarrow$ block ID and size.

Glossary:

F	filesystem number.
D	Directory unique id.
C	File unique id.
S	Snapshot number.
o	Offset in file.
N	Filename in directory.
c	Directory iteration cookie.
F, D', C', S	The handle of a file in a directory.

Fig. 12. Filesystem schema showing key → value pair mappings. The small numbers (e.g., "1") are literal numbers inserted between components of a key to disambiguate key types and force proper B-tree sort ordering. For directories, $C = 0$.

We implement snapshots using copy-on-write at the key-value pair level rather than doing copy-on-write in the B-tree data structure or at the block level [15, 33, 41, 49, 55, 71, 73, 74, 81]. One effect of this decision is that our B-tree is update-in-place, which makes it easy to maintain a linked list of all the leaves.

Our key-value scheme achieves locality in the B-tree. When a file is created it is lexicographically near its parent directory, and the file's block maps and fixed-sized metadata are near each other. (If the file is later moved, then it still appears near the original parent directory.) This means that if you create a file in a directory that has only a few files in it, then it is likely that the whole transaction to update the directory inode, add directory entries, and create the file inode will all be on the same page, or at least in the same extent, since the B-tree maintains maintains block as well as page locality (see Section 5).

We use multiple block sizes (which are shown in Figure 4) to address the tension between fragmentation and metadata overhead. Small blocks keep fragmentation low for small files. Big blocks reduce the number of block map entries and other bookkeeping overhead for big files. In our scheme the first few blocks of a file are small, and as the file grows the blocks get bigger. For files larger than 16 KiB, the largest block is no bigger than 1/3 the file size, so that even if the block is nearly empty, we have wasted no more than 1/3 of our storage. We sometimes skip small-block allocation entirely. For example, if the NFS client writes 1 MiB into a newly created file, then we can use 256 KiB blocks right away. This block allocation scheme is 1.5-competitive for any file larger than 10.7 KiB.

Using a B-tree provides us a great deal of flexibility. The best-known alternative to B-tree storage is to store metadata in specialized data structures (an array of inodes; trees of direct blocks, indirect blocks, and doubly indirect blocks; and hash tables of directory entries). But such specialized data structures are difficult to extend. For example, implementing snapshots was fairly straightforward. We can easily extend the schema to store extended file attributes, complex access control lists (e.g, for NFSv4 [76]), or resource forks [79].

RIGHTSLINK

5 THE B-TREE

To hold metadata, we built a B-tree [9] on top of MPSC. MPSC provides transactions that can update up to 15 pages, but we want to think about key-value pairs, not pages, and we want B-tree transactions to be able include non-B-tree pages and blocks, e.g., to allocate a data block and store its tagged pointer in a B-tree key-value pair. The B-tree can perform transactions on a total of 15 values, where a value can be a key-value pair or a non-B-tree page write or block allocation.

Consider the simple problem of executing a B-tree transaction to update a single key-value pair. How many pages must be included in that transaction? The standard B-tree algorithm starts at the root of the tree and follows pointers down to a leaf page where it can access the pair. To update the leaf we need to know that it is the proper page, and the way we know that is by having followed a pointer from the leaf's parent. Between reading the parent and reading the leaf, however, the tree might have been rebalanced, and so we might be reading the wrong leaf. So we we need to include the parent in the transaction. Similarly, we need to include the grandparent and all the ancestors up to the root. A typical B-tree might be five or six levels deep, and so a single key-value pair update transaction involves five or six pages, which would limit us to two or three key-value pairs per transaction. Furthermore, every transaction ends up including the root of the B-tree, creating a scalability bottleneck.

Our solution to this problem is to use ***self-validating pages***, which contain enough information that we can determine if we read the right page by looking at that page in isolation. We arrange every page to "own" a range of keys, for the page to contain only keys in that range, and that every possible key is owned by exactly one page. To implement this self-validation, we store in every page a lower bound and upper bound for the set of keys that can appear in the page (sometimes called "fence keys" [31, 53]), and we store the height of the page (leaf pages are height 0). When we read a page to look up a key, we verify that the page we read owns the key and is the right height, in which case we know that if that page is updated in a successful transaction, that we were updating the right page. Thus, we do not need to include the intermediate pages in the MPSC operation and we can perform B-tree transactions on up to 15 keys.

We usually skip accessing the intermediate B-tree nodes altogether by maintaining a cache that maps keys to pages. If the cache steers us to a wrong page, then either the page will not self-validate or the transaction will fail, in which case we simply invalidate the cache entry and try again. If a key is missing from the cache, then we can perform a separate transaction that walks the tree to populate the cache. It turns out that this cache is very effective, and for virtually all updates we can simply go directly to the proper page to access a key-value pair.

Another problem that could conceivably increase the transaction size is tree rebalancing. In a B-tree, tree nodes must generally be split when they get too full or merged when they get too empty. The usual rule is that whenever one inserts a pair into a node and it does not fit, one first splits the node and updates the parent (possibly triggering a parent split that updates the grandparent, and so on). Whenever one deletes a pair, if the node becomes too empty (say less than 1/4 full), then one merges nodes, updating the parent (which can possibly trigger a parent merge that updates the grandparent, and so on). This means that any insertion or deletion can add as many pages to a transaction as the height of the tree. Those rebalancings are infrequent so they do not introduce a scalability bottleneck, but they do make our MPSC operations too big.

Our solution to the rebalancing problem is to rebalance in a separate transaction. When inserting, if we encounter a page overflow, then we abort the transaction, split the page in a separate transaction, and restart the original transaction. We split the page even if it is apparently nearly empty: As long as there are two keys we can split the page. For merges, we delete keys from the page, and then do a separate transaction afterward to rebalance the tree. It is possible that a page

RIGHTSLINK

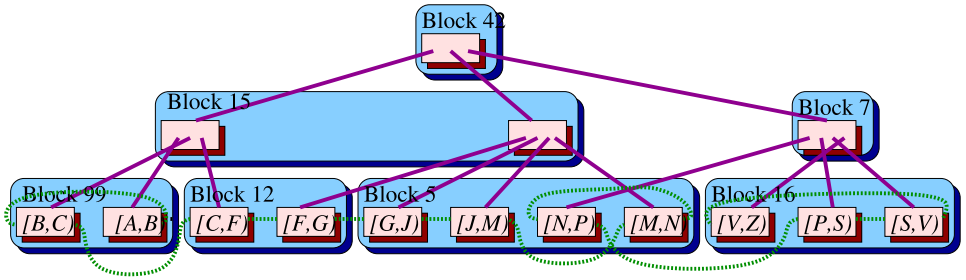

Fig. 13. The B-tree comprises blocks (in blue) and pages (in pink). The pages form a tree. The leaf pages, where the key-value pairs are stored, form a doubly linked list (shown with dashed green lines). Each leaf page is responsible for a range of keys, e.g., $[C, F)$ means the keys from C inclusive to F exclusive. Each block holds a key range of pages for one level. For example, Block 5 has all the leaf pages in the range $[G, P)$.

could end up empty, or nearly empty, and that due to some crash or packet loss, we forget to rebalance the tree. That is OK, because we fix it up the next time we access the page.

To improve locality, we exploit both the page and block structure of MPSC. Figure 13 shows how the B-tree is organized to exploit block locality as well as page locality. Each page is responsible for a key range, and the union of the key ranges in a block is a single key range. When splitting a page, we place the new page into the same block as the old page, and if the block is full, then we insert a new block. If the B-tree schema strives to keep keys that will appear in the same transaction lexicographically near each other, then locality causes those keys to likely be in the same page, or at least the same block. Our MPSC implementation optimizes for the case where some pages of a transaction are in the same extent. With the schema described in Section 4, this optimization is worth about a 20% performance improvement for an operation such as untarring a large tarball.

The choice of 15 pages per transaction is driven by the B-tree implementation. There is one infrequent operation requiring 15 pages. It involves splitting a page in a full block: A new block is allocated, block headers are updated, and the pages are moved between blocks. Most transactions touch only one or two pages.

6 DASD: NOT YOUR PARENT'S DISK DRIVE

This section explains how we implemented MPSC using Paxos state machines (which we discuss further in Section 7). MPSC is implemented by a distributed block store, called ***DASD***.[2] A single extent is implemented by a Paxos state machine, so multipage transactions within an extent are straightforward. To implement transactions that cross extents, we use 2-phase commit.

Given that Paxos has provided us with a collection of replicated state machines, each with an attached disk, each implementing one extent, we implement two-phase commit [34, 52, 56] on top of Paxos. The standard problem with two-phase commit is that the transaction coordinator can fail and the system gets stuck. Our extents are replicated, so we view the participants in a transaction as being unstoppable.

It would be easy to implement two-phase commit with $3n$ messages. One could send n "prepare" messages that set up the pages, then n "decide" messages that switch the state to commited, and then n "release" messages that release the resources of the transaction. (Each message drives a state transition, which is replicated by Paxos.) The challenge is to implement two-phase commit on n extents with only $2n$ messages and state changes. Every filesystem operation would benefit from the latency being reduced by 1/3.

[2]Direct-Access Storage Device (DASD) was once IBM's terminology for disk drives [43].

RIGHTSLINK

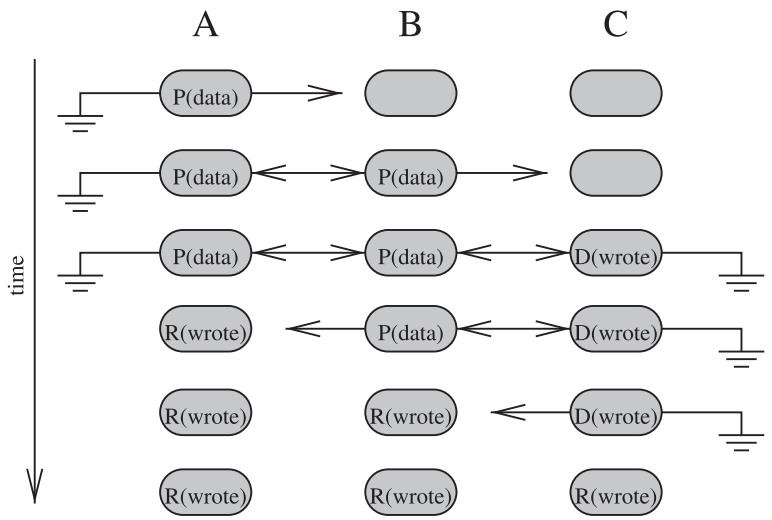

Fig. 14. Three extents performing a DASD transaction. Each column is an extent, and each row is a point in time, with time moving downward. A double-arrowed line shows two extents pointing at each other. A single-arrowed line shows one extent pointing at the other, with no pointer back. Ground represents a null pointer.

To perform an atomic operation with only $2n$ messages, for example on three pages, the system progresses through the states shown in Figure 14. The system will end up constructing and tearing down, in the Paxos state machine, a doubly linked (not circular) list of all the extents in the transaction. Each of these steps is initiated by a message from a client, which triggers a state change in one Paxos state machine (which in turn requires several messages to form a consensus among the Paxos replicas). The client waits for an acknowledgment before sending the next message.

(1) Extent A receives a prepare message. A enters the prepared state, indicated by "P(data)", and records its part of the transaction data and its part of the linked list (a null back pointer, and a pointer to B).
(2) Extent B receives a prepare message, enters the prepared state, and records its data and pointers to A and C.
(3) Extent C receives a prepare-and-decide message, enters the decided state (committing the transaction), and records its data (indicated by "D(wrote)") and the back pointer to B.
(4) Extent A receives a decide-and-release message, notes that the transaction is committed, and releases its resources (such as memory) associated with the transaction, indicated by "R(wrote)." The head of the linked list is now gone.
(5) Extent B receives a decide-and-release message, notes the commit, and releases its resources.
(6) Extent C receives a release message (it had already decided) and releases its resources.

Thus we implement two-phase commit in exactly $2n$ messages with $2n$ state transitions. Note that the final transition of state C does not actually need to be done before replying to the client, and could be piggybacked into the prepare step of the next transaction, potentially reducing the latency to $2n - 1$ messages.

The system maintains the invariant that either a prefix or a suffix of the linked list exists, which is useful if the transaction is interrupted. There are two ways that the system can be interrupted,

RIGHTSLINK

before the commit (in which case the transaction will abort), or after (in which case the cleanup is incomplete). The prefix-suffix property helps in both of these cases. If the transaction gets aborted (at or before step 3), then a prefix exists. If we encounter a prepared state, then we can follow the linked list forward until we either find a dangling pointer or a decided state. If we find a dangling pointer, then we can delete the prepare record that contained the dangling pointer, preserving a prefix. (At the final point, C, of the linked list, we must extract a promise that C will never decide that the transaction commits. This can be accomplished by introducing a conflict on the read slot for the page.) If we find a decided state, then the cleanup was interrupted, so it can proceed back along the linked list until we find the beginning or a dangling pointer and move the state forward to released.

Our scheme relies on the fact that each state transition occurs one after the other, and hence the critical path of the transition is also $2n$ messages. There are schemes in which one can move the states forward in parallel. For example, one could broadcast "prepare" messages to all the extents, then have one extent decide, and then broadcast decide messages to them all, then release messages, so that the critical path would be only 4 long. This results in $3n$ state transitions (minus one or two, depending on how clever you are). If you think that big transactions are common, then that is valuable, but we have found that most transactions are short so it is better to do the transaction serially.

We optimize the case when there are several pages in a single extent to use fewer messages.

7 PIPELINED PAXOS

In this section, we explain our Paxos implementation, and in particular how we pipeline Paxos operations.

Lamport-Paxos [50, 51] is an algorithm to achieve consensus on a single value. Lamport-Paxos requires two phases, called phase 1 and phase 2 by Lamport.

To achieve consensus on a log (as opposed to one value), one common algorithm is Multi-Paxos [20], which treats the log as an array indexed by slot, running Lamport-Paxos independently on each array element. It turns out that you can run a "vector" phase 1 for infinitely many elements of the array with a single pair of messages and that you can reuse the outcome of phase 1 for as many phase 2 rounds as you want. In this setup, people tend to call phase-1 "master election" and infer all sorts of wrong conclusions, e.g., that there is only one master at any time and that phase 1 is some kind of "failover."

In Multi-Paxos, if the operation on slot $S + 1$ depends on the state after slot S, then you must wait for slot S (and all previous slots) to finish phase 2. (We do not say "commit" to avoid confusion with two-phase commit, which is a different protocol.) This Multi-Paxos is not pipelined.

You can pipeline Multi-Paxos with a small modification. You tag each log entry with a unique log sequence number (LSN) and you modify Paxos so that an acceptor accepts slot $S + 1$ only if it agrees on the LSN of slot S. Thus, the Paxos phase 2 message is the Lamport phase 2 plus the LSN of the previous slot. By induction, two acceptors that agree on a LSN agree on the entire past history.

Now you can issue phase 2 for $S + 1$ depending on S without waiting for S to complete, because the acceptance of $S + 1$ retroactively confirms all speculations that you made.

The pipelined Multi-Paxos state, per slot, is the Lamport-Paxos state (a ***ballot*** B and the slot's contents) plus the LSN. You can use whatever you want as LSNs, as long as they are unique, but a convenient way to generate LSNs is to use the pair $\langle E, S \rangle$ where the ***epoch*** E must be unique. As it happens, Lamport phase 1 designates a single winner of ballot B, so you can identify E with the winner of ballot B in phase 1 and be guaranteed that nobody else wins that ballot. In the $E = B$ case,

RIGHTSLINK

you can reduce the per-slot state to the single-value E, with the dual-role of LSN for pipelining and of ballot for Lamport-Paxos.

Our Paxos algorithm is almost isomorphic to Raft [67]. Essentially Raft is Multi-Paxos plus conditional LSNs plus $E = B$. However, Raft always requires an extra log entry to make progress and cannot be done in bounded space. If you recognize that you are just doing good-old Paxos, then you can make progress by storing a separate ballot B in constant space.

The idea of the acceptance conditional on the previous LSN appeared in viewstamped replication [65] (which did not discuss pipelining). It is used specifically for pipelining in Zookeeper, except that Zookeeper tries to reinvent Paxos but incorrectly assumes TCP is an ideal pipe [8]. Conditional acceptance is also used in Raft in the same way as in viewstamped replication, except that Raft lost the distinction between proposer and acceptor, which prevents it from having a speculative proposer state that runs ahead of acceptors.

7.1 Recovery

Here we explain how our Paxos system recovers from failures.

The on-disk state of a Paxos acceptor has two main components: the log (of bounded size, a few MB) and a large set of page shards (tens of GB). A shard comprises an erasure-coded fragment of a page and some header information such as a checksum. To write a shard, the Paxos proposer appends the write command to the log of multiple acceptors. When a quorum of acceptors has accepted the command, the write is considered done (or "learned" in Paxos terminology). The proposer then informs acceptors that a command has been learned, and acceptors write the erasure-coded shard to disk.

As long as all acceptors receive all log entries, this process guarantees that all acceptors have an up-to-date and consistent set of shards. However, acceptors may temporarily disappear for long enough that the only way for the acceptor to make progress is to incur a log discontinuity. We now must somehow rewrite all shards modified by the log entries that the acceptor has missed, a process called recovery.

The worst-case for recovery is when we must rewrite the entire set of shards, for example, because we are adding a new acceptor that is completely empty. In this ***long-term*** recovery, as part of their on-disk state, acceptors maintain a range of pages that need to be recovered, and they send this ***recovery state*** back to the proposer. The proposer iterates over such pages and overwrites them by issuing a Paxos read followed by a conditional Paxos write, where the condition is on the page still being the same since the read. When receiving a write, the acceptor subtracts the written page range from the to-be-recovered page range and sends the updated range back to the proposer.

Long-term recovery overwrites the entire extent. For discontinuities of short duration, we use a less-expensive mechanism called ***short-term recovery***. In addition to the long-term page range, acceptors maintain a range of log slots that they have lost, they update this range when incurring a discontinuity and communicate back this slot range to the proposer. The proposer, in the Paxos state machine, maintains a small pseudo-LRU cache of identifiers of pages that were written recently, indexed by slot. If the to-be-recovered slot range is a subset of the slot range covered by the cache, then the proposer issues all the writes in the slot range, in slot order, along with a range R whose meaning is that the present write is the only write that occurred in slot range R. When receiving the write, the acceptor subtracts R from its to-be-recovered slot range and the process continues until the range is empty. If the to-be-recovered slot range overflows the range of the cache, then the acceptor falls into long-term recovery.

In practice, almost all normal operations (e.g., software deployments) and unscheduled events (e.g., power loss, network disruption) are resolved by short-term recovery. We need long-term recovery when loading a fresh replica and (infrequently) when a host goes down for a long time.

RIGHTSLINK

The time it takes to run short-term recovery is roughly bounded by the amount of time that the replica was off line: A replica simply needs to catch up on any updates that it missed while it was down. Long-term recovery is proportional to the extent size and runs at a few hundred megabytes per second, which is throttled to avoid slowing down the continuing workload.

7.2 Checkpointing and Logging

Multi-Paxos is all about attaining consensus on a log, and then we apply that log to a state machine. All memory and disk space in FSS is statically allocated, and so the logs are of a fixed size. The challenge is to checkpoint the state machine so that we can trim old log entries. The simplest strategy is to treat the log as a circular buffer and to periodically write the entire state machine into the log. Although for DASD extents, the state machine is only a few MB, some of our other replicated state machines are much larger. For example we use a five-way replicated hash table, called Minsk, to store configuration information for bootstrapping the system: Given the identity of the five Minsk instances, a Dendron instance can determine the identity of all the other Dendron instances. If the Paxos state machine is large, then checkpointing the state machine all at once causes a performance glitch.

Here is a simple scheme to deamortize checkpointing. Think of the state machine as an array of bytes, and every operation modifies a byte. Now, every time we log an update to a byte, we also pick another byte from the hash table and log its current value. We cycle through all the bytes of the table. Thus, if the table is K bytes in size, then after K update operations we will have logged every byte in the hash table, and so the most recent $2K$ log entries have enough information to reconstruct the current state of the hash table. We do not need to store the state machine anywhere, since the log contains everything we need.

This game can be played at a higher level of abstraction. For example, suppose we think of the hash table as an abstract data structure with a `hash_put` operation that is logged as a logical operation rather than as operations on bytes. In that case every time we log a `hash_put` we also log the current value of one of the hash table entries and take care to cycle through all the entries. If the hash table contains K key-value pairs, then the entire hash table will be reconstructable using only the most recent $2K$ log entries. This trick works for a binary tree, too.

8 AVOIDING CONFLICTS

This section outlines three issues related to transaction conflicts: avoiding too much retry work, avoiding congestion collapse, and reducing conflicts by serializing transactions that are likely to conflict.

In a distributed system, one must handle errors in a disciplined fashion. The most common error is when a transaction is aborted, because it conflicts with another transaction. Retrying transactions at several different places in the call stack can cause an exponential amount of retrying. Our strategy is that the storage host does not retry transactions that fail. Instead, it attempts to complete one transaction, and if it fails, then the error is returned all the way back to Orca, which can decide whether to retry. Orca typically sets a 59 s deadline for each NFS operation and sets a 1 s deadline for each MPSC. Since the NFS client will retry its operation after 60 s, it is OK for Orca to give up after 59 s. One special case is handing READ and WRITE requests, in which case a partial result can be returned if the full request times out.

Some errors, such as transaction conflicts, impute congestion. In some situations the request transaction did not complete, because some "housekeeping" operation needed to be run first (such as to rebalance two nodes of the B-tree). Doing the housekeeping uses up the budget for a single transaction, so an error must returned to Orca, but in this case the error does not impute congestion.

RIGHTSLINK

When two transactions conflict, one aborts, which is inefficient. We use in-memory locking to serialize transactions that are likely to conflict. For example, when Orca makes an FSS call to access an inode, it sends the request to the storage host that is likely to be the Paxos leader for the extent where that inode is stored. That storage host then acquires an in-memory lock so that two concurrent calls accessing the same inode will run one after another. Orca maintains caches that map key ranges to extent numbers and extent numbers to the leader's IP address. Sometimes one of the caches is wrong, in which case, as a side effect of running the transaction, the storage host will learn the correct cache entries, and inform Orca, which will update its cache. The in-memory lock is used for performance and is not needed for correctness. The technique of serializing transactions that are likely to conflict is well known in the transactional-memory literature [54, 83].

To avoid performance collapse, Orca employs a congestion-control system similar to TCP's window-size management algorithm [80]. Every Orca process maintains two congestion windows per back-end server. One window applies to Dendron and the other to DASD. A window's state comprises one real number, w, the window size.

The window size is expressed in terms of the number of requests that can be sent to the server in one round trip time (RTT) period. We initialize the window size 1. If a reply from a server indicates congestion, then we decrease the window size to w' as

$$w' = \begin{cases} w - 1/(2C) & \text{if } w \geq 1, \text{ and} \\ w - w/(2C) & \text{otherwise.} \end{cases}$$

If a reply indicates "no congestion," then we increase window size as

$$w' = \begin{cases} w + 1/(wC) & \text{if } w \geq 1, \text{ and} \\ w + w/C & \text{otherwise.} \end{cases}$$

The constant C is a tuning parameter, which we set to 200 based on empirial measurements.

The meaning of w is as follows. If $w \geq 1$, then we allow up to w requests to be outstanding. If $w < 1$, then we delay requests so that only one request is sent every $1/w$ as often so that on average only w requests are outstanding. For example, if the window size is $1/2$, then a thread will wait 2 RTT periods before sending the request.

The congestion indicator from the server depends on a combination of latency and explicit congestion notification (ECN) [69]. If an acceptor or a disk I/O takes too long to respond to a request, then we set the congestion indicator. If any queue length becomes too full, then we set the congestion indicator (specifically if a queue length approaches the number of threads in the Dendron or DASD server the queue is "too full"). This approach reduces the chance of congestion collapse and keeps the system operating in a domain where latency is low.

9 PERFORMANCE

In the introduction we promised customers a convex combination of 100 MB/s of bandwidth and 3,000 IOPS for every terabyte stored. Those numbers are throughput numbers and to achieve those numbers the NFS clients may need to perform operations in parallel. This section first explains where those throughput numbers come from and then discusses FSS latency.

To make concrete throughput statements, we posit a simplified model in which the network bandwidth determines performance. The network bottleneck turns out to be on the storage hosts. If the virtual network interfaces (VNICs) on the NFS clients are the bottleneck, then the customer can add NFS clients. If the presentation host is the bottleneck, then additional mount targets can be provisioned. The performance of the NVMe is fast compared to the several round trips required by Paxos (in contrast to, e.g., Gaios, which needed to optimize for disk latency instead of network latency, because disks were so slow [13]).

RIGHTSLINK

We define performance in terms of the ratio of bandwidth to storage capacity. There are four components to the performance calculation: raw performance, replication, scheduling, and oversubscription. The ***raw performance*** is the network bandwidth divided by the disk capacity, without accounting for replication, erasure coding, scheduling, or oversubscription.

Replication consumes both bandwidth and storage capacity. Using 5:2 erasure coding, for each page of data, half a page is stored in each of five hosts. This means we can sell only 40% of the raw storage capacity. The network bandwidth calculation is slightly different for writes and reads. For writes, each page must be received by five different storage hosts running Paxos. That data are erasure coded by each host then written to disk. Thus, for writes, replication reduces the raw network bandwidth by a factor of 5.

For reads we do a little better. To read a page we collect all five erasure-coded copies, each of which is half a page and reconstruct the data using two of the copies. We could probably improve this further by collecting only two of the copies, but for now our algorithm collects all five copies. So for reads, replication reduces the bandwidth by a factor of 5/2.

Scheduling further reduces the bandwidth but has a negligible effect on storage capacity. Queueing theory tells us that trying to run a system over about 70% utilization will result in unbounded latencies. We do not do quite that well. We find that we can run our system at about 1/3 of peak theoretical performance without affecting latency. This factor of 3 is our ***scheduling overhead***.

Since not all the customers are presenting their peak load at the same time, we can sell the same performance several times, a practice known as oversubscription. In our experience, we can oversubscribe performance by about a factor of 5.

The units of performance simplify from MB/s/TB to s^{-1}, so 100 MB/s/TB is one overwrite per 10,000 s.

For input-outputs per second (IO/s) we convert bandwidth to IOPS by assuming that most IOs are operations on 32 KiB pages, so we provide 3,000 IO/s/TB. The cost of other IOs can be expressed in terms of reads: A write costs 2.5 reads, a file creation costs 6 reads, an empty-file deletion costs 8 reads, and a file renaming costs about 10 reads.

This performance model appears to work well on every parallel workload we have seen. To test this model, we measured how much bandwidth a relatively small test fleet can provide. (We are not allowed to do these sorts of experiments on the big production fleets.) We measured on multiple clients, where each client has its own mount target on its own Orca. This fleet has 41 storage instances each with a 10 Gbit/s VNIC for a total raw performance of 51.25 GB/s. After replication that is 10.25 GB/s of salable bandwidth. Dividing by 3 to account for scheduling overhead is 3.4 GB/s. Those machines provide a total of 200 TB of salable storage, for a ratio of 17 MB/s/TB. According to our model, with fivefold oversubscription, this fleet should promise customers 85 MB/s/TB.

Figure 15 shows measured bandwidth. The variance was small so we measured only six runs at each size. The measured performance is as follows. When writing into an empty file, block allocation consumes some time, and a single client can get about 434 MB/s, whereas 12 clients can get about 2.0 GB/s. When writing into an existing file, avoiding block allocation overhead, the performance is about 536 MB/s and 2.4 GB/s for 1 and 12 clients, respectively.

We hypothesized that we could model this data as a ***simple-speedup*** curve [11] (a variant of Amdahl's law or of Brent and Graham's Theorem [7, 14, 32]). In a simple-speedup scenario, as we increase the number of clients, we see a linear speedup that eventually flattens out to give an asymptotic speedup. The curve is parameterized by two numbers l and a. The first value, l, is the ***linear-speedup*** slope that applies when the number of clients C is small where the performance will be $l \cdot C$. The second value, a, is the ***asymptotic speed***, and indicates the performance for large

RIGHTSLINK

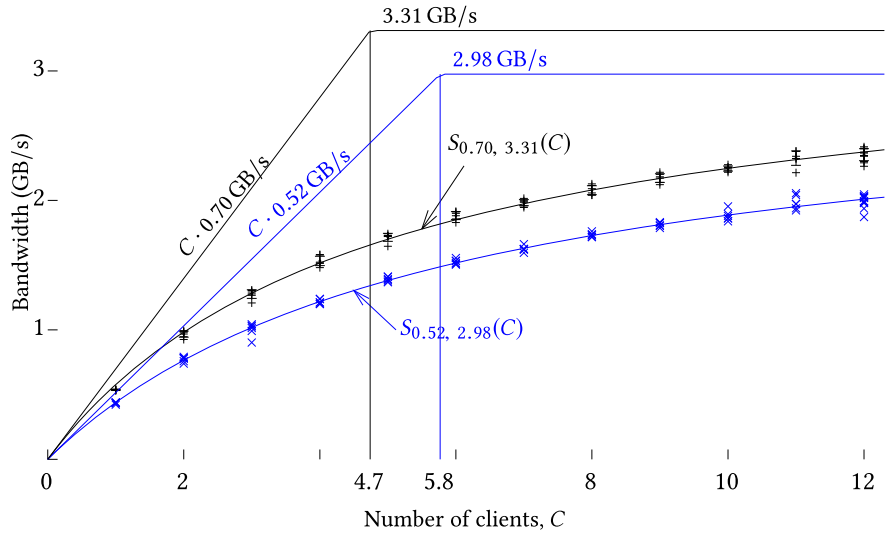

Fig. 15. Measured bandwidth. The X-axis is the number of NFS clients. The Y-axis is the cumulative bandwidth achieved. The crosses (in black) show measured performance writing into a preallocated file. The x's (in blue) show measured performance including block allocation. The functions S are the curves fit to a simple-speedup model, with the corresponding linear-speedup shown as lines passing through the origin and the asymptotic speedups shown as horizontal lines. The number of clients at the intercepts are also shown.

numbers of clients. The simple speedup curve,

$$S_{l,a}(C) = 1/(1/lC + 1/a),$$

is simply half the harmonic mean of the linear-speedup curve lC and the asymptotic speed a.

We fitted of our data to the simple-speedup model and plotted the resulting curves in Figure 15. The asymptotic standard error for the curve fit is less than 1.7%. Visually, the curves fit the data surprisingly well.

We can interpret these curves as follows: When writing to an empty file (which includes block allocation), a few clients can each achieve about 0.52 GB/s, and many clients can achieve a total of 2.98 GB/s. The cutover between "few" and "many" is about 6 clients for this fleet. When writing to a preallocated file, a few clients can each achieve 0.70 GB/s, and many clients can achieve a total of 3.31 GB/s, which is close to our estimate of 3.4 GB/s. The intercept of the speedup curve lines tells us the ***half-power*** points, where half the peak capacity is consumed: The amount of 4.7 clients consume half of the fleet's allocate-and-write capacity, and 5.8 clients consume half of the write-without-allocation capacity.

Low latency is hard to achieve in a replicated distributed system. Latency is the elapsed time from an NFS client's request to response in an otherwise unloaded system. For serial workloads, latency can dominate performance. For example, when running over NFS, the `tar` program creates files one at a time, waiting for an acknowledgment that each file exists on stable storage before creating the next file. After looking at various benchmarks, we concluded that we should simply measure `tar`'s runtime on a well-known tarball such as the Linux 4.19.2 source code distribution, which is 839 MB and contains 65,825 objects. Untarring Linux onto local NVMe device takes about 10 s. The same NVMe served over NFS finishes in about 2.5 minutes. FSS finishes in about 10 minutes. Amazon's EFS, which replicates across a metropolitan region, finishes in about an hour. According to this limited experiment, NFS costs a factor of 15, replication within a

RIGHTSLINK

datacenter costs another factor of 4, and synchronous replication over metropolitan-scale distances costs another factor of 6. Achieving local-filesystem performance in a replicated distributed fault-tolerant filesystem appears ... difficult.

10 THE IVY CONTROL-PLANE STORE

Most cloud services are organized into two parts: a ***data plane***, and a ***control plane***. Often the way these two parts evolve is that the cloud service provider takes an open-source program, puts it into their cloud, and calls it a "service." The open-source program might not be cloud native, and typically does not include facilities for setting up a customer's instance of the service. So whatever needs to be added to provision the service, do billing, do health telemetry, and so forth, is lumped together into the control plane. This situation is much cleaner than that for a file storage service: Anything that the NFS protocol can express is data plane, and anything else is control plane. So, for example, NFS provides no way to create or delete a file system, create, or maintain mount targets, attach a mount target to the customer's VCN, calculate usage-based bills, add presentation hosts or storage hosts, failover Orca processes when a presentation host fails, or delete snapshots. These functions are handled by the FSS control plane.

To perform its function, the control plane needs a data store, which we implement with a second B-tree running an RPC protocol that we call Ivy. We use a second B-tree, because control-plane keys turn out to be much larger than the keys in the data-plane B-tree. The largest keys in the data plane are filenames, which including all their overhead max out at about 300 bytes per key. But the largest control-plane keys are several thousand bytes. Since our B-tree balancing algorithms work well only if the keys are less than about 1/8 of a page, we end up needing larger pages for the control-plane store than for the data-plane store. For the data plane, we want the pages to be as small as possible, because large pages induce a performance penalty. So we chose 8 KiB pages for the data plane and 64 KiB pages for the control plane. Thus the Ivy B-tree employs a different extent type than the data-plane B-tree.

We will briefly discuss Orca failover, which is one of the more interesting problems faced by the control-plane. One of our requirements is that mount targets should exhibit high availability.

It turns out that creating a new IP endpoint is relatively slow using our VCN infrastructure, and so if an Orca server fails it would take too long to remove the existing endpoint and create a new one connected to a different Orca server. But our VCN provides the ability to create an IP alias for an existing endpoint, and we can quickly move the alias. So our strategy is to create two IP endpoints attaching particular Orca servers to the VNC and then use a third IP that the customer uses and that IP can quickly change when an Orca fails.

We use a separate set of servers, called Orca Managers (ORMA), whose job is to monitor and, when necessary, failover mount targets to a different server. We want to tolerate the failure of a entire fault domain, so we run a set of ORMA servers in different fault domains, usually three in each data center. To achieve concensus on the state of an Orca server and to serve various other control plane functions we use the Ivy B-tree.

Ivy's keys and values are array of bytes that can be of length up to about 8 KiB. Ivy supports four RPC methods: `GET`, `COMMIT`, `ITERATE`, and `CONFIRM`.

`GET` Takes up to seven keys and returns their corresponding values along with a sequence number. The Ivy sequence numbers are a little different from the Paxos LSNs or the MPSC slot numbers, but they serve much the same function: If a value changes, then the sequence number returned for that value will change. If a key-value pair is not present, then Ivy will return a NULL value along with a sequence number.

RIGHTSLINK

In contrast to the data-plane B-tree, the control-plane B-tree does not allow inconsistent reads. In the data plane, performance is so important that the client programmer must program around inconsistent reads. Control-plane performance is less critical so we made it easier to program by providing only consistent reads. To provide consistent reads on subsequent `GET` operations in the same transaction, the client also provides the set of keys and sequence numbers that it has read so far. If any of those keys have had their values modified (which can be determined by examining the sequence numbers), then the `GET` operation will return an error code.

`COMMIT` allows the client to modify up to seven key value pairs. The client provides a set of key-value-sequence-number triples. That is, for each key the client provides a value and the sequence number that was used to read that value. Only if none of those values have changed can the commit succeed.

`ITERATE` allows client to iterate over keys in lexicographic order. The `ITERATE` function returns keys with no values or sequence numbers. The `ITERATE` function is not transactionally consistent, and so the client must perform a `GET` on any keys it wants to use in a transaction.

`CONFIRM` allows client to confirm that value of up to seven key value pairs did not change after their respective sequence numbers were obtained. Compared to `GET` and `COMMIT`, `CONFIRM` is light weight.

We do not store a sequence number on every key-value pair. Instead we use ***ownership records*** [35]. We maintain a collection of $N = 1{,}000{,}000$ ownership records in the Ivy B-tree (they are spread around in the key space so that they are likely to be on different pages to reduce contention). Each ownership record contains a sequence number that is just a number that monotonically increases over time. We employ a hash function, h, mapping each key k to a number in the range $[0, N)$, and the sequence number stored in record $h(k)$. This allows us to compute a sequence number even for keys that are not present in the tree.

An alternative design would have been to put a sequence number in each key-value pair and use that sequence number for any lookup between that key and the next present key.

We partition Ivy key space into tables. Every key belongs to one of the tables and single `COMMIT` call can modify key-value pairs from different tables.

Since Ivy keys and values can have relatively long length (up to 8 KiB), we can encode various complex data structures into key names.

Orca servers can break in various different ways, but we assume that as long as an Orca server can write to Ivy it is still live. A process runs on every Orca server that periodically writes heartbeat values to its key in Ivy.

When one of the ORMA servers detects that an Orca server did not post a heartbeat we start a failover process. The failover process has multiple steps, some of which are outside of our control (for example, moving the IP alias for an endpoint is a call to the VCN system.) We organize the steps so that they are idempotent and so it is safe repeat them multiple times. The decision to fail over an Orca host is recorded in Ivy along with which of the idempotent steps have completed. This workflow of small transactions can be used to effect a large operation that we sometimes call a ***second order transaction (SOT)***. All three ORMA servers are allowed to make progress on one of the SOTs, but because of Ivy's `COMMIT` semantics, only one of them will succeed. If one of the ORMA servers is not available to make progress with one of the SOTs, then any other ORMA server will continue to make progress. We use a similar mechanism to implement various other control plane workflows.

RIGHTSLINK

11 RELATED WORK

MPSC is a variation of load-link/store-conditional [47] and seems less susceptible to the ABA problem (in which the same location is read twice and has the same value for both reads, tricking the user into thinking that no transaction has modified the location in the meanwhile) than compare-and-swap [24, 25, 38]. Version tagging and the ABA problem appeared in Reference [44, p. A-44].

Sinfonia has many similarities to our system. Sinfonia minitransactions [1] are similar to MPSC. Sinfonia uses uses primary-copy replication [16] and can suffer from the split-brain problem, where both primary and replica become active and lose consistency [23]. To avoid split-brain, Sinfonia remotely turns off power to failed machines, but that is just another protocol that can, e.g., suffer from delayed packets and does not solve the problem. We employ Paxos [50], which is a correct distributed consensus algorithm.

Many filesystems have stored at least some their metadata in B-trees [10, 26, 45, 59, 71, 74, 75] and some have gone further, storing both metadata and data in a B-tree or other key-value store [21, 46, 72, 84, 85]. Our B-tree usage is fairly conventional in this regard, except that we store many filesystems in a single B-tree.

ZFS and HDFS [36, 77, 78, 81] support multiple block sizes in one file. Block suballocation and tail merging filesystems [3, 71, 74] are special cases of this approach.

Some filesystems avoid using Paxos on every operation. For example, Ceph [48] uses Paxos to run its monitors but replicates data asynchronously. Ceph's crash consistency can result in replicas that are not consistent with each other. Some systems (e.g., References [27, 28, 58]) use other failover schemes that have not been proved correct. Some filesystems store all metadata in memory [28, 37, 42, 48, 64], resulting in fast metadata access until the metadata gets too big to fit in RAM. We go to disk on every operation, resulting in no scaling limits.

Two-phase commit on top of Paxos-based transaction coordinators seems to have first been implemented by Google Spanner [22]. Apparently Amazon's elastic file system (EFS) [5] (which provides a similar product to our service, and precedes our service by a few years) also uses two-phase commit on top of Paxos. For a more complete history of layering transactions on top of replicated storage see the related work section in the Spanner paper.

Some cloud providers offer managed storage appliance services [6, 30]. These services provision a virtual machine per filesystem and run specific filesystem stacks on top of pre-provisioned storage. This approach can provide excellent latency, since it operates directly on local storage. Some offerings also provide specialized filesystem stacks (such as Lustre) for specialized use cases. Some customers need the low latency and are willing to tolerate the resulting filesystems that have some combination of being nonscalable, nonelastic, less durable, less consistent, and harder to manage, because they require users to to pre-provision and partition appliances of the right size and throughput.

12 CONCLUSION

We conclude with a brief history of FSS. The team started with Frigo and Kuszmaul, and the first code commit (comprising mostly CRC subroutines) was on July 4, 2016. Paxos and DASD were implemented by the end of July. At that time Paxos was 500 lines of code (LOC), and DASD was 3,100 LOC. The B-tree was working by November 2016. At that time the B-tree was 3,000 LOC Paxos was 10,000 LOC, and DASD was 9,000 LOC. Sandler joined and started Orca implementation on August 3, 2016. Mazzola Paluska joined on September 15, 2016 and implemented the filesystem schema in the B-tree. Control-plane work started in spring 2017. The team grew to about a dozen people in January 2017 and is about two dozen people in fall 2019. As of fall 2019, the B-tree is

RIGHTSLINK

9,000 LOC, Paxos is 16,000 LOC, DASD is 21,000 LOC, and Orca is 15,000 LOC. Limited availability was launched on July 7, 2017, less than one year after first commit (but without a control plane—all configuration was done manually). General availability started January 29, 2018. As of Spring 2019, FSS hosted over 10,000 filesystems containing several petabytes of paid customer data and is growing at an annualized rate of 8- to 60-fold per year (there is some seasonal variation).

ACKNOWLEDGMENTS

In addition to the authors, the team that built and operates FSS has included the following: the data-plane team of Yonatan (Yoni) Fogel, Michael Frasca, Stephen Fridella, Jan-Willem Maessen, and Chris Provenzano; the control-plane team of Vikram Bisht, Bob Naugle, Michael Chen, Ligia Connolly, Yi Fang, Cheyenne T. Greatorex, Alex Goncharov, David Hwang, Lokesh Jain, Uday Joshi, John McClain, Dan Nussbaum, Ilya Usvyatsky, Mahalakshmi Venkataraman, Viktor Voloboi, Will Walker, Tim Watson, Hualiang Xu, and Zongcheng Yang; the product- and program-management team of Ed Beauvais, Mona Khabazan, and Sandeep Nandkeolyar; solutions architect Vinoth Krishnamurthy; and the engineering-management team of Thomas (Vinod) Johnson, Alan Mullendore, Stephen Lewin-Berlin, and Rajagopal Subramaniyan. Heidi Peabody provided administrative assistance. We further rely on the many hundreds of people who run the rest of Oracle Cloud Infrastructure. Many thanks to the referees for their suggestions on improving this article.

REFERENCES

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamoanolis. 2009. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.* 27, 3 (Nov. 2009). DOI: https://doi.org/10.1145/1629087.1629088

[2] Alibaba 2018. Alibaba Elastic Block Storage. Retrieved September 26, 2018 from https://www.alibabacloud.com/help/doc-detail/25383.htm.

[3] Hervey Allen. 2005. Introduction to FreeBSD additional topics. In *Proceedings of the Pacific Network Operators Group (PacNOG I) Workshop.*

[4] Amazon 2018. Amazon Elastic Block Store. Retrieved September 26, 2018 from https://aws.amazon.com/ebs.

[5] Amazon 2018. Amazon Elastic File System. Retrieved October 12, 2019 from https://aws.amazon.com/efs.

[6] Amazon 2018. Amazon FSx. Retrieved January 22, 2020 from https://aws.amazon.com/fsx.

[7] G. M. Amdahl. 1967. The validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS'67)*, Vol. 30.

[8] Apache Software Foundation. 2009. ZooKeeper Internals. Retrieved from https://zookeeper.apache.org/doc/r3.1.2/zookeeperInternals.html.

[9] Rudolf Bayer and Edward M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Inf.* 1, 3 (Feb. 1972), 173–189. DOI: https://doi.org/10.1145/1734663.1734671

[10] Steve Best and Dave Kleikamp. 2000. JFS layout. *IBM Developerworks.* Retreived from http://jfs.sourceforge.net/project/pub/jfslayout.pdf.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.* 37, 1 (Aug. 25 1996), 55–69. (An early version appeared in the *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95).* 207–216.

[12] Hans-J. Boehm. 2009. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar'09).* 15:1–15:6.

[13] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation.* 141–154.

[14] Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM* 21, 2 (Apr. 1974), 201–206.

[15] Gerth Stølting Brodal, Konstantinos Tsakalidis, Spyros Sioutas, and Kostas Tsichlas. 2012. Fully persistent B-trees. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12).* 602–614. DOI: https://doi.org/10.1137/1.9781611973099.51

RIGHTSLINK

[16] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. The primary-backup approach. In *Distributed Systems* (2 ed.). ACM Press/Addison-Wesley, New York, NY, 199–216.
[17] Brent Callaghan, Brian Pawlowski, and Peter Staubach. 1995. NFS Version 3 Protocol Specification. *IETF RFC 1813*. Retrieved from https://www.ietf.org/rfc/rfc1813.
[18] Rémy Card, Theodore Ts'o, and Stephen Tweedie. 1994. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*.
[19] Călin Casçaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy. *ACM Queue* 6, 5 (Sept. 2008). DOI: https://doi.org/10.1145/1454456.1454466
[20] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*. 398–407. DOI: https://doi.org/10.1145/1281100.1281103
[21] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. 2017. File systems fated for senescence? Nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 45–58.
[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 251–264.
[23] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in partitioned network. *Comput. Surv.* 17, 3 (Sep. 1985), 341–370. DOI: https://doi.org/10.1145/5505.5508
[24] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. 2000. Even better DCAS-based concurrented deques. In *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*. 59–73.
[25] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distrib. Comput.* 15, 4 (Dec. 2002), 255–271. DOI: https://doi.org/10.1017/s00446-002-0079-z
[26] Matthew Dillon. 2008. The Hammer Filesystem. Retrieved from https://www.dragonflybsd.org/hammer/hammer.pdf.
[27] Mark Fasheh. 2006. OCFS2: The oracle clustered file system version 2. In *Proceedings of the 2006 Linux Symposium*. 289–302.
[28] Glustre 2005. GlusterFS. Retrieved from http://www.gluster.org.
[29] Google 2012. Google Persistent Disk. Retrieved September 26, 2018 from https://cloud.google.com/persistent-disk/.
[30] Google 2018. Google Filestore. Retrieved January 22, 2020 https://cloud.google.com/filestore/.
[31] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems* 35, 3 (Jul. 2010). DOI: https://doi.org/10.1145/1806907.1806908
[32] R. L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* 17, 2 (Mar. 1969), 416–429.
[33] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
[34] Jim N. Gray. 1978. Notes on data base operating systems. In *Operating Systems—An Advanced Course*. Lecture Notes in Computer Science, Vol. 60. Springer-Verlag, Chapter 3.
[35] Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. In *Proceedings of the 18th Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. 388–402.
[36] HDFS 2012. Add support for Variable length block. *HDFS Ticket*. Retrieved from https://issues.apache.org/jira/browse/HDFS-3689.
[37] HDFS 2013. HDFS Architecture. Retrieved from http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Large_Data_Sets.
[38] Maurice Herlihy. 1991. Wait-free synchronizatoin. *ACM Trans. Program. Lang. Syst.* 11, 1 (Jan. 1991), 124–149. DOI: https://doi.org/10.1145/114005.102808
[39] M. Herlihy and J. E. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. 289–300. DOI: https://doi.org/10.1145/173682.165164
[40] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (Jul. 1990), 463–492. DOI: https://doi.org/10.1145/78969.78972
[41] Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*. 19–19.
[42] Valentin Höbel. 2016. LizardFS: Software-Defined Storage As It Should Be. Retrieved from https://www.golem.de/news/lizardfs-software-defined-storage-wie-es-sein-soll-1604-119518.html.

RIGHTSLINK

[43] IBM. 1966. *Data File Handbook*. Retrieved from http://www.bitsavers.org/pdf/ibm/generalInfo/C20-1638-1_Data_File_Handbook_Mar66.pdf C20-1638-1.
[44] IBM 1983. *IBM System/370 Extended Architecture—Principles of Operation*. IBM. Retrieved from https://archive.org/details/bitsavers_ibm370prinrinciplesofOperationMar83_40542805.
[45] Apple Inc. 2004. *HFS Plus Volume Format*. Retrieved from Technical Note TN1150. Apple Developer Connection. https://developer.apple.com/library/archive/technotes/tn/tn1150.html.
[46] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A write-optimization in a kernel file system. *ACM Trans. Stor.* 11, 4 (Nov. 2015). DOI: https://doi.org/10.1145/2798729
[47] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. 1987. *A New Approach to Exclusive Data Access in Shared Memory Multiprocessors*. Technical Report UCRL-97663. Lawrence Livermore National Laboratory, Livermore, California. Retrieved from https://e-reports-ext.llnl.gov/pdf/212157.pdf.
[48] M. Tim Jones. 2004. Ceph: A Linux petabyte-scale distributed file system. Retrieved from https://www.ibm.com/developerworks/linux/library/l-ceph/index.html.
[49] Sakis Kasampalis. 2010. *Copy on Write Based File Systems Performance Analysis and Implementation*. Master's thesis. Department of Informatics, The Technical University of Denmark. Retrieved from http://sakisk.me/files/copy-on-write-based-file-systems.pdf.
[50] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. DOI: https://doi.org/10.1145/279227.279229
[51] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News* 32, 4 (121) (Dec. 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.
[52] Butler Lampson. 1980. Atomic transactions. In *Distributed Systems—Architecture and Implementation*. Vol. 100. Springer Verlag.
[53] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems* 6, 4 (Dec. 1981), 650–670. DOI: https://doi.org/10.1145/319628.319663
[54] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *Proceedings of the The 2nd ACM SIGPLAN Workshop on Transactional Computing*.
[55] A. J. Lewis. 2002. LVM HOWTO. Retrieved from http://tldp.org/HOWTO/LVM-HOWTO/.
[56] Bruce G. Lindsay. 1980. Single and multi-site recovery facilities. In *Distributed Data Bases*, I. W. Draffan and F. Poole (Eds.). Cambridge University Press, Chapter 10. Also available as Reference [57].
[57] Bruce G. Lindsay, Patricia G. Selinger, Cesare A. Galtieri, James N. Gray, Raymond A. Lorie, Thomas G. Price, Franco Putzolu, Irving L. Traiger, and Bradford W. Wade. 1979. *Notes on Distributed Databases*. Research Report RJ2571. IBM Research Laboratory, San Jose, CA. Retrieved from http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.
[58] Lustre 2003. The Lustre File System. Retrieved from lustre.org.
[59] Avantika Mathur, MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*.
[60] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *Comput. Syst.* 2, 3 (1984), 181–197. DOI: https://doi.org/10.1145/989.990
[61] Microsoft 2017. Microsoft Azure Blob Storage. Retrieved from https://azure.microsoft.com/en-us/services/storage/blobs/. Viewed 2018-09-26.
[62] Microsoft 2018. Microsoft SMB Protocol and CIFS Protocol Overview. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/FileIO/microsoft-smb-protocol-and-cifs-protocol-overview.
[63] Barton P. Miller, Louis Fredersen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. DOI: https://doi.org/10.1145/96267.96279
[64] Moose 2018. MooseFS Fact Sheet. Retrieved from https://moosefs.com/factsheet/.
[65] Brian Oki and Barbara Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*. 8–17. DOI: https://doi.org/10.1145/62546.62549
[66] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*.
[67] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm (Extended Version). Retrieved from https://raft.github.io/raft.pdf. Extended version of Reference [66].
[68] Oracle 2016. Oracle Cloud Infrastructure Block Volumes. Retrieved from https://cloud.oracle.com/en_US/storage/block-volume/features.

RIGHTSLINK

[69] K. K. Ramakrishnan, Sally Floyd, and David L. Black. 2001. The Addition of Explicit Congestion Notification (ECN) to IP. *IETF RFC 3168*. Retrieved from http://www.ietf.org/rfc/rfc3168.txt.

[70] I. S. Reed and G. Solomon. 1960. Polynomial codes over certain finite fields. *J. Soc. Industr. Appl. Math.* 8, 2 (Jun. 1960), 300–304. DOI: https://doi.org/10.1137/0108018

[71] Hans T. Reiser. 2006. Reiser4. Retrieved July 6, 2006 from https://web.archive.org/web/20060706032252 http://www.namesys.com:80/.

[72] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference*. 145–156.

[73] Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Trans. Comput. Logic* 3, 4 (Feb. 2008), 15:1–15:27. DOI: https://doi.org/10.1145/1326542.1326544

[74] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans. Stor.* 9, 3 (Aug. 2013). DOI: https://doi.org/10.1145/2501620.2501623

[75] Mark Russinovich. 2000. Inside Win2K NTFS, Part 1. *ITProToday* (22 Oct. 2000). Retrieved from https://www.itprotoday.com/management-mobility/inside-win2k-ntfs-part-1.

[76] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. 2003. Network File System (NFS) version 4 Protocol. *IETF RFC 3530*. Retrieved from https://www.ietf.org/html/rfc3530.

[77] Chris Siebenmann. 2017. ZFS's `recordsize`, Holes In Files, and Partial Blocks. Retrieved from https://utcc.utoronto.ca/cks/space/blog/solaris/ZFSFilePartialAndHoleStorage.

[78] Chris Siebenmann. 2018. What ZFS Gang Blocks Are and Why They Exist. Retrieved August 30, 2018 from https://utcc.utoronto.ca/ cks/space/blog/solaris/ZFSGangBlocks.

[79] Jon Stacey. 2009. Mac OS X Resource Forks. *Jon's View (blog)*. Retrieved January 23, 2020 https://jonsview.com/mac-os-x-resource-forks.

[80] W. Richard Stevens. 1997. TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms. *IETF RFC 2001*. Retrieved from https://www.ietf.org/html/rfc2001.

[81] Sun Microsystems. 2006. ZFS On-Disk Specification—draft. Retrieved from http://www.giis.co.in/Zfs_ondiskformat.pdf.

[82] Adam Sweeny, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC'96)*. 1–14.

[83] Lingxiang Xiang and Michael L. Scott. 2015. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 234–243. DOI: https://doi.org/10.1145/2755573.2755577

[84] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2017. Writes wrought right, and other adventures in file system optimization. *ACM Trans. Stor.* 13, 1 (Mar. 2017), 3:1–3:21. DOI: https://doi.org/10.1145/3032969

[85] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2018. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 123–138.

Received October 2019; accepted January 2020

RIGHTSLINK