PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database

Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu,
Song Zheng, Yuhui Wang, Guoqing Ma
{mingsong.cw, zhenjun.lzj, wangpeng.wangp, chensen.cs, caifeng.zhucaifeng,
catttree.zs, yuhui.wyh, guoqing.mgq}@alibaba-inc.com

ABSTRACT

PolarFS is a distributed file system with ultra-low latency and high availability, designed for the POLARDB database service, which is now available on the Alibaba Cloud. PolarFS utilizes a lightweight network stack and I/O stack in user-space, taking full advantage of the emerging techniques like RDMA, NVMe, and SPDK. In this way, the end-to-end latency of PolarFS has been reduced drastically and our experiments show that the write latency of PolarFS is quite close to that of local file system on SSD. To keep replica consistency while maximizing I/O throughput for PolarFS, we develop *ParallelRaft*, a consensus protocol derived from Raft, which breaks Raft's strict serialization by exploiting the out-of-order I/O completion tolerance capability of databases. ParallelRaft inherits the understandability and easy implementation of Raft while providing much better I/O scalability for PolarFS. We also describe the shared storage architecture of PolarFS, which gives a strong support for POLARDB.

PVLDB Reference Format:
Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, Guoqing Ma. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *PVLDB*, 11 (12): 1849 - 1862, 2018.
DOI: https://doi.org/10.14778/3229863.3229872

1. INTRODUCTION

Recently, decoupling storage from compute has become a trend for cloud computing industry. Such design makes the architecture more flexible and helps to exploit shared storage capabilities: (1) Compute and storage nodes can use different types of server hardware and can be customized separately. For example, the compute nodes need no longer to consider the ratio of memory size to disk capacity, which is highly dependent on the application scenario and hard to predict. (2) Disks on storage nodes in a cluster can form a single storage pool, which reduces the risk of fragmentation, imbalance of disk usage among nodes, and space wastage. The capacity and throughput of a storage cluster can be easily scaled out transparently. (3) Since data are all stored on the storage cluster, there is no local persistent state on compute nodes, making it easier and faster to perform database migration. Data reliability can also be improved because of the data replication and other high availability features of the underlying distributed storage system.

Cloud database services also benefit from this architecture. First, databases can build on a more secure and easily scalable environment based on virtualization techniques, such as Xen [4], KVM [19] or Docker [26]. Second, some key features of databases, such as multiple read-only instances and checkpoints could be enhanced with the support of backend storage clusters which provide fast I/O, data sharing, and snapshot.

However, data storage technology continues to change at a rapid pace, so current cloud platforms have trouble taking full advantage of the emerging hardware standards such as RDMA and NVMe SSD. For instance, some widely used open-source distributed file systems, such as HDFS [5] and Ceph [35], are found to have much higher latency than local disks. When the latest PCIe SSDs are used, the performance gap could even reach orders of magnitude. The performance of relational databases like MySQL running directly on these storage systems is significantly worse than that on local PCIe SSDs with the same CPU and memory configurations.

To tackle this issue, cloud computing vendors, like AWS, Google Cloud Platform and Alibaba Cloud, provide instance store. An instance store uses a local SSD and high I/O VM instance to fulfill customers' needs of high performance databases. Unfortunately, running cloud databases on an instance store has several drawbacks. First, instance store has limited capacity, not suitable for a large database service. Second, it cannot survive underlying disk drive failure. Databases have to manage data replication by themselves for data reliability. Third, an instance store uses a general-purpose file system, such as ext4 or XFS. When using low I/O latency hardwares like RDMA or PCIe SSD, the message-passing cost between kernel space and user space compromises the I/O throughput. Even worse, an instance store cannot support a shared-everything database cluster architecture, which is a key feature for an advanced cloud database service.

In this paper, we describe our design and implementation of ***PolarFS***, a distributed file system, which provides ultra-low latency, high throughput and high availability by adopting the following mechanisms. First, PolarFS takes full

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229872

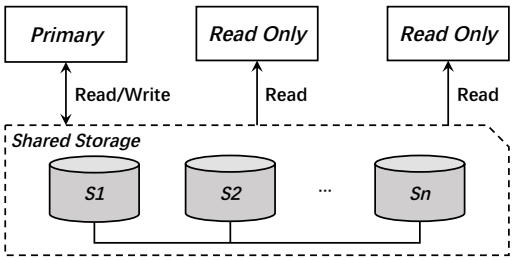


Figure 1: The POLARDB architecture.

advantage of emerging hardware such as RDMA and NVMe SSD, and implements a lightweight network stack and I/O stack in user space to avoid trapping into kernel and dealing with kernel locks. Second, PolarFS provides a POSIX-like file system API, which is intended to be compiled into the database process and replace the file system interfaces provided by operating system, so that the whole I/O path can be kept in user space. Third, the I/O model of the PolarFS's data plane is also designed to eliminate locks and avoid context switches on the critical data path: all unnecessary memory copies are also eliminated, meanwhile DMA is heavily utilized to transfer data between main memory and RDMA NIC/NVMe disks. With all these features, the end-to-end latency of PolarFS has been reduced drastically, being quite close to that of local file system on SSD.

Distributed file systems deployed in the cloud production environment typically have thousands of machines. Within such a scale, failures caused by hardware or software bugs are common. Therefore, a consensus protocol is needed to ensure that all committed modifications will not get lost in corner cases, and replicas can always reach agreement and become bitwise identical.

The Paxos family[23, 22, 7, 18, 38] of protocols is widely recognized for solving consensus. Raft[28], a variant of Paxos, is much easier to understand and implement. Many distributed systems are developed based on Raft. However, once Raft was applied to PolarFS, we found that Raft seriously impedes the I/O scalability of PolarFS when using extra low latency hardware (e.g. NVMe SSD and RDMA whose latency are on the order of tens of microseconds). So we developed *ParallelRaft*, an enhanced consensus protocol based on Raft, which allows out-of-order log acknowledging, committing and applying, while letting PolarFS comply with traditional I/O semantics. With this protocol, parallel I/O concurrency of PolarFS has been improved significantly.

Finally, on top of PolarFS, we implemented POLARDB, a relational database system modified from AliSQL (a fork of MySQL/InnoDB) [2], which is recently available as a database service on Alibaba cloud computing platform. POLARDB follows the shared storage architecture, and supports multiple read-only instances. As shown in figure 1, the database nodes of POLARDB are divided into two types: a primary node and read only (RO) nodes. The primary node can handle both read and write queries, while the RO node only provides read queries. Both primary and RO nodes share redo log files and data files under the same database directory in PolarFS.

PolarFS supports the POLARDB with following features: (1) PolarFS can synchronize the modification of file metadata (e.g. file truncation or expansion, file creation or deletion) from primary nodes to RO nodes, so that all changes are visible for RO nodes. (2) PolarFS ensures concurrent modifications to file metadata are serialized so that the file system itself is consistent across all database nodes. (3) In case of a network partition, two or more nodes might act as primary nodes writing shared files concurrently in PolarFS, PolarFS can ensure that only the real primary node is served successfully, preventing data corruption..

The contributions of this paper include:

- We describe how to build PolarFS, a state-of-art distributed file system with ultra-low latency, leveraging emerging hardware and software optimizations. (Section 3, 4, and 7)
- We propose ParallelRaft, a new protocol to achieve consensus. ParallelRaft is designed for large scale, fault-tolerant and distributed file system. It is modified based on Raft to suit storage semantics. Compared with Raft, ParallelRaft provides better support for high concurrent I/Os in PolarFS. (Section 5)
- We present the key features of PolarFS which give a strong support POLARDB's share storage architecture. (Section 6)

Other parts of the paper are structured as follows. Section 2 gives background information about emerging hardware that PolarFS uses. Section 8 presents and discusses our experimental evaluation. Section 9 reviews the related work and Section 10 concludes the paper.

2. BACKGROUND

This section briefly describes NVMe SSD, RDMA and their corresponding programming models.

NVMe SSD. SSD is evolving from legacy protocols like SAS, SATA to NVMe, which has a high-bandwidth and low-latency I/O interconnect. A NVMe SSD can deliver up to 500K I/O operations per second (IOPS) at sub 100μs latency, and the latest 3D XPoint SSD even cuts I/O latency down to around 10μs while providing much better QoS than NAND SSDs. As SSDs are getting faster, the overhead of the legacy kernel I/O stack becomes the bottleneck [37, 36, 31]. As revealed by previous research [6], there are approximately 20,000 instructions needed to be executed only to complete a single 4KB I/O request. Recently, Intel released Storage Performance Development Kit (SPDK) [12], a suite of tools and libraries for building high performance NVMe device based, scalable, user-mode storage applications. It achieves high performance by moving all necessary drivers into user space and operating in a polling mode instead of relying on interrupts, which avoids kernel context switches and eliminates interrupt handling overhead.

RDMA. RDMA technique provides a low-latency network communication mechanism between servers inside data center. For instance, transferring a 4KB data packet between two nodes connecting to the same switch takes about 7μs, which is much faster than traditional TCP/IP network stack. A number of previous works [9, 14, 15, 17, 24, 25, 30] show that RDMA can improve the system performance. Applications interact with the RDMA NIC by accessing a *queue pair* (QP) via the Verbs API. A QP is composed of a *send queue* and a *receive queue*, besides, each QP is associated another *completion queue* (CQ). The CQ is typically used as

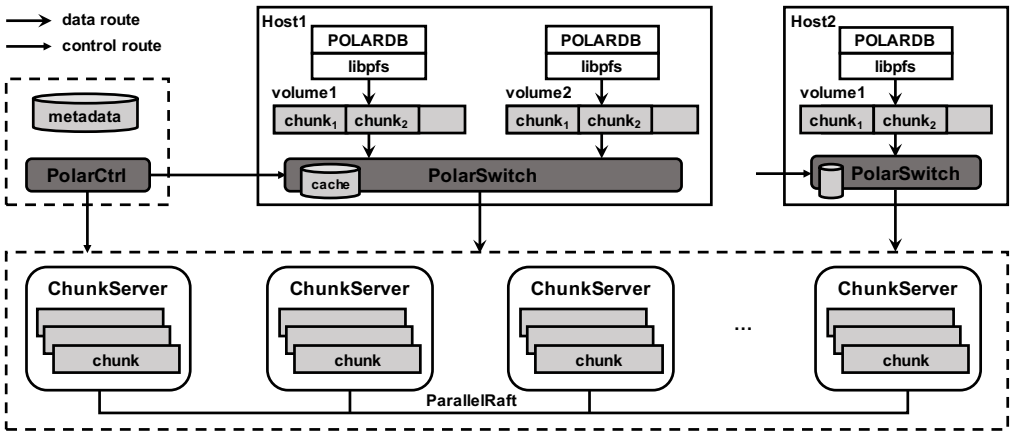


Figure 2: Storage Layer Abstraction.

the polling target for completion event/signal. Send/Recv verbs are commonly referred to as two-sided operations since each Send operation needs a matching Recv operation invoked by the remote process, while Read/Write verbs are known as one-sided operations because remote memory is manipulated by the NIC without involving any remote CPU.

PolarFS uses a hybrid of Send/Recv and Read/Write verbs. Small payloads are transferred by Send/Recv verbs directly. For a large chunk of data or a batch of data, nodes negotiate about the destination memory address on the remote node using Send/Recv verbs, and then complete the actual data transmission over Read/Write verbs. PolarFS eliminates context switches by polling the CQ in user space instead of relying on interrupts.

3. ARCHITECTURE

PolarFS consists of two main layers. The lower one is storage management, and the upper one manages file system metadata and provides file system API. **Storage layer** takes charge of all the disk resource of the storage nodes, and provides a database volume for every database instance. **File system layer** supports file management in the volume, and is responsible for mutual exclusion and synchronization of concurrent accesses to file system metadata. For a database instance, PolarFS stores the file system metadata in its volume.

Here we present major components in a PolarFS cluster, as illustrated in Figure 2. **libpfs** is a user space file system implementation library with a set of POSIX-like file system API, which is linked into the POLARDB process; **PolarSwitch** resides on compute nodes to redirect I/O requests from applications to ChunkServers; **ChunkServers** are deployed on storage nodes to serve I/O requests; **PolarCtrl** is the control plane, which includes a set of masters implemented in micro-service, and agents deployed on all compute and storage nodes. PolarCtrl uses a MySQL instance as metadata repository.

3.1 File System Layer

The file system layer provides a shared and parallel file system, designed to be accessed concurrently by multiple database nodes. For example, in the scenario of POLARDB, when the primary database node executes a create table DDL statement a new file is created in PolarFS, enabling

```
int     pfs_mount(const char *volname, int host_id)
int     pfs_umount(const char *volname)
int     pfs_mount_growfs(const char *volname)

int     pfs_creat(const char *volpath, mode_t mode)
int     pfs_open(const char *volpath, int flags, mode_t mode)
int     pfs_close(int fd)
ssize_t pfs_read(int fd, void *buf, size_t len)
ssize_t pfs_write(int fd, const void *buf, size_t len)
off_t   pfs_lseek(int fd, off_t offset, int whence)
ssize_t pfs_pread(int fd, void *buf, size_t len, off_t offset)
ssize_t pfs_pwrite(int fd, const void *buf, size_t len, off_t offset)
int     pfs_stat(const char *volpath, struct stat *buf)
int     pfs_fstat(int fd, struct stat *buf)
int     pfs_posix_fallocate(int fd, off_t offset, off_t len)
int     pfs_unlink(const char *volpath)
int     pfs_rename(const char *oldvolpath, const char *newvolpath)
int     pfs_truncate(const char *volpath, off_t len)
int     pfs_ftruncate(int fd, off_t len)
int     pfs_access(const char *volpath, int amode)

int     pfs_mkdir(const char *volpath, mode_t mode)
DIR*    pfs_opendir(const char *volpath)
struct dirent *pfs_readdir(DIR *dir)
int     pfs_readdir_r(DIR *dir, struct dirent *entry,
              struct dirent **result)
int     pfs_closedir(DIR *dir)
int     pfs_rmdir(const char *volpath)
int     pfs_chdir(const char *volpath)
int     pfs_getwd(char *buf)
```

Figure 3: libpfs interfaces

select statements executed on a RO node to access the file. Therefore, it is necessary to synchronize modifications of file system metadata across nodes to keep consistent, while serializing concurrent modifications to avoid the metadata being corrupted.

3.1.1 libpfs

The libpfs is a lightweight file system implementation running completely in user space. As shown in Figure 3, The libpfs provides a set of POSIX-like file system API. It is quite easy to port a database to run on top of PolarFS.

When a database node starts up, the pfs_mount attaches to its volume and initializes the file system state. The volname is the global identifier of the volume allocated to the POLARDB instance, and the hostid is the index of the database node which is used as a identifier in the disk paxos voting algorithm (see Section 6.2). During the mounting process, the libpfs loads file system metadata from the volume, and constructs data structures e.g. the directory tree, the file mapping table and the block mapping table in the

main memory (see Section 6.1). The pfs_umount detaches the volume and releases resources during database destruction. After the volume space grows, the pfs_mount_growfs should be called to recognize newly allocated blocks and mark them available. Rest functions are file and directory operations which are equivalent to counterparts in POSIX API. Details of file system metadata management is described in Section 6.

3.2 Storage Layer

The storage layer provides interfaces to manage and access volumes for the file system layer. A volume is allocated for each database instance, and consists of a list of chunks. The capacity of a volume ranges from 10 GB to 100 TB, which meets the requirements of vast majority of databases, and the capacity can be expanded on demand by appending chunks to the volume. A volume can be randomly accessed (read, write) in 512B alignment, like traditional storage devices. Modifications to the same chunk carried in a single I/O request are atomic.

Chunk. A volume is divided into chunks which are distributed among ChunkServers. A chunk is the minimum unit of data distribution. A single Chunk does not span across more than one disk on ChunkServer and its replicas are replicated to three distinct ChunkServers by default (always located in different racks). Chunks can be migrated between ChunkServers when hot spots exist.

The size of a chunk is set to 10 GB in PolarFS, which is significantly larger than the unit size in other systems, e.g. the 64 MB chunk size used by GFS [11]. This choice decreases the amount of metadata maintained in metadata database by orders of magnitude, and also simplifies the metadata management. For example, a 100 TB volume only contains 10,000 chunks. It is a relatively small cost to store 10,000 records in metadata database. Moreover, all of these metadata can be cached in the main memory of PolarSwitch, thus avoiding the extra metadata accessing costs on the critical I/O path.

The downside of this design is that the hot spot on one chunk cannot be further separated. But thanks to the high ratio of chunks to servers (now is around 1000:1), typically a large number of database instances (thousands or more), and the inter-server chunk migration capability, PolarFS can achieve load balance at the whole system level.

Block. A chunk is further divided into blocks inside the ChunkServer, and each block is set to 64 KB. Blocks are allocated and mapped to a chunk on demand to achieve thin provisioning. A 10 GB chunk contains 163,840 data blocks. The mapping table of chunk's LBA (Logical Block Address, the linear address range from 0 to 10 GB) to blocks are stored locally in ChunkServer, together with the bitmap of free blocks on each disk. The mapping table of a single chunk occupies 640 KB memory, which is quite small and can be cached in memory of the ChunkServer.

3.2.1 PolarSwitch

PolarSwitch is a daemon process deployed on the database server, together with one or more database instances. In each database process, libpfs forwards I/O requests to the PolarSwitch daemon. Each request contains addressing information like the volume identifier, offset and length, from which the related chunks can be identified. An I/O request may span across chunks, in that case, the request is further divided into multiple sub-requests. Finally an elemental request is sent to the ChunkServer where the leader of the chunk resides.

PolarSwitch finds out the locations of all replicas belonging to a chunk by looking up the local metadata cache, which is synchronized with PolarCtrl. Replicas of a chunk form a consensus group, one is the leader and others are followers. Only the leader can answer I/O requests. leadership changes in the consensus group are also synchronized and cached in PolarSwitch's local cache. If response timeout happens, PolarSwitch would keep retrying with exponential backoff while detecting whether leader election happens, switch to new leader and retransmit immediately if it does.

3.2.2 ChunkServer

ChunkServers are deployed on storage servers. There are multiple ChunkServer processes running on a storage server, each ChunkServer owns a standalone NVMe SSD disk and is bound to a dedicated CPU core, so that there is no resource contention between two ChunkServers. ChunkServer is responsible to store chunks and provides random accesses to chunks. Each chunk contains a write ahead log (WAL), modifications to the chunk are appended to the log before updating chunk blocks to ensure atomicity and durability. ChunkServer uses a piece of fixed size 3D XPoint SSD buffer as a write cache for WAL, logs are preferred to be placed in 3D XPoint buffer. If the buffer is full, ChunkServer will try to recycle dated logs. If there is still not enough space in 3D XPoint buffer, logs are written into NVMe SSD instead. Chunk blocks are always written into NVMe SSD.

ChunkServers replicate I/O requests with each other using the ParallelRaft protocol and form a consensus group. A ChunkServer leaves its consensus group for various reasons, and may be handled differently. Sometimes it's caused by occasional and temporary faults, e.g. network unavailable temporarily, or the server is upgraded and rebooted. In this situation it's better to wait for the disconnected ChunkServer to come back online, join the group again and catch up with others. In other cases, faults are permanent or tend to last for a long time, e.g. the server is damaged or taken offline. Then all chunks on the missing ChunkServer should be migrated to others, in order to regain a sufficient number of replicas.

A disconnected ChunkServer will always make effort to rejoin the consensus group autonomously, in order to shorten unavailable time. However complementary decisions can be made by PolarCtrl. PolarCtrl periodically collects the list of ChunkServers that has disconnected before, and picks off ChunkServers seems like having permanent fault. Sometimes it is hard to make the decision. A ChunkServer with a slow disk, for example, may have a much longer latency than others, but it can always respond to aliveness probes. Machine learning algorithms based on performance metrics statistics of key components is helpful.

3.2.3 PolarCtrl

PolarCtrl is the control plane of a PolarFS cluster. It is deployed on a group of dedicated machines (at least three) to provide high-available services. PolarCtrl provides cluster controlling services, e.g. node management, volume management, resource allocation, metadata synchronization, monitoring and so on. PolarCtrl is responsible for (1) keeping track of the membership and liveness of all ChunkServers

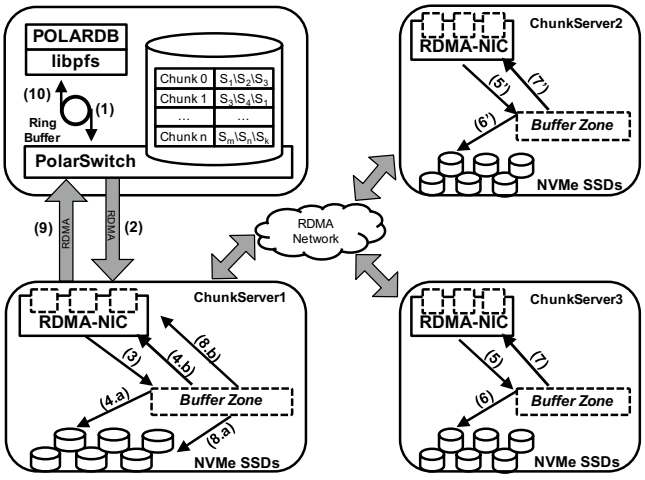


Figure 4: The Write I/O Execution Flow.

in the cluster, starting migration of chunk replicas from one server to the others if the ChunkServer is overloaded or unavailable for duration longer than a threshold value. (2) maintaining the state of all volumes and chunk locations in metadata database. (3) creating volumes and assigning chunks to ChunkServers. (4) synchronizing metadata to PolarSwitch using both push and pull methods. (5) monitoring the latency/IOPS metrics of each volume and chunk, collecting trace data along the I/O path. (6) scheduling inner-replicas and inter-replicas CRC checks periodically.

PolarCtrl periodically synchronizes cluster metadata (e.g. chunk locations of a volume) with PolarSwitch through control plane commands. PolarSwitch saves the metadata in its local cache. When receiving an I/O request from libpfs, PolarSwitch routes the request to the corresponding ChunkServer according to the local cache. Occasionally, PolarSwitch would fetch metadata from PolarCtrl if the local cache falls behind the centre metadata repository.

As a control plane, PolarCtrl is not on the critical I/O path, its service continuity can be provided using traditional high availability techniques. Even during the short interval between PolarCtrl's crash and recovery, I/O flows in PolarFS would not likely be affected due to the cached metadata on PolarSwith and the self-management of ChunkServer.

4. I/O EXECUTION MODEL

When POLARDB accesses its data, it will delegate a file I/O request to libpfs by the PFS interface, usually by pfs_pread or pfs_pwrite. For write requests, there is almost no need to modify file system meta data since device blocks are preallocated to files by pfs_fallocate, thus avoiding expensive meta data synchronization among write and read nodes. This is a normal optimization for database systems.

In most common cases, libpfs simply maps the file offset into a block offset based on the index tables already built when mounting, and chops the file I/O request into one or more smaller fixed size block I/O requests. After the transformation, the block I/O requests are sent by libpfs to PolarSwitch through a shared memory between them.

The shared memory is constructed as multiple ring buffers. At one end of the shared memory, libpfs enqueues a block I/O request into a ring buffer selected in a round robin way and then waits for its completion. At the other end , PolarSwitch constantly polls all ring buffers, with one thread dedicated to a ring buffer. Once it finds new requests, PolarSwitch dequeues the requests from ring buffers and forwards them to ChunkServers with the routing information propagated from PolarCtrl.

Chunkserver uses the write ahead logging (WAL) technique to ensure atomicity and durability, in which I/O requests are written to the log before they are committed and applied. Logs are replicated to a collection of replicas, and a consensus protocol named ParallelRaft (Detailed in the next section) is used to guarantee the data consistency among replicas. An I/O request is not recognized to be committed until it is persistently recorded to the logs on a majority of replicas. Only after that this request can be responded to the client and applied on the data blocks.

Figure 4 shows how a write I/O request is executed inside PolarFS. (1) POLARDB sends a write I/O request to PolarSwitch through the ring buffer between PolarSwitch and libpfs. (2) PolarSwitch transfers the request to the corresponding chunk's leader node according to the local cached cluster metadata. (3) Upon the arrival of a new write request, the RDMA NIC in the leader node will put the write request into the pre-registered buffer and add a request entry into the request queue. An I/O loop thread keeps polling the request queue. Once it sees a new request arriving, it starts processing the request right away. (4) The request is written to the log block on the disk through SPDK and is propagated to the follower nodes through RDMA. Both operations are asynchronous calls and the actual data transmission will be triggered in parallel. (5) When the replication request arrives at a follower node, the RDMA NIC in the follower node will also put the replication request into the pre-registered buffer and add it into the replication queue. (6) Then the I/O loop thread on the follower is triggered, and writes the request to disk through SPDK asynchronously. (7) When the write callback returns successfully, an acknowledge response is sent back to the leader through RDMA. (8) When responses are successfully received from majority of followers, the leader applies the write request to the data blocks through SPDK. (9) After that, the leader replies to PolarSwitch through RDMA. (10) PolarSwitch then marks the request done and notifies the client.

Read I/O requests are (more simply) processed by leader alone. In ChunkServer there is a submodule named IoScheduler which is responsible to arbitrate the order of disk I/O operations issued by concurrent I/O requests to execute on the ChunkServer. IoScheduler guarantees that a read operation can always retrieve the latest committed data.

ChunkServer uses polling mode along with event-driven finite state machine as the concurrency model. The I/O thread keeps polling events from RDMA and NVMe queues, processing incoming requests in the same thread. When one or more asynchronous I/O operations are issued and other requests are needed to be handled, the I/O thread will pause processing current request and save the context into a state machine, then switch to process the next incoming event. Each I/O thread uses a dedicated core and uses separated RDMA and NVMe queue pairs. As a result, an I/O thread is implemented without locking overheads since there is no shared data structure between I/O threads, even though there are multiple I/O threads on a single ChunkServer.

5. CONSISTENCY MODEL

5.1 A Revision of Raft

A production distributed storage system needs a consensus protocol to guarantee all committed modifications are not lost in any corner cases. At the beginning of the design process. we chose Raft after consideration of the implementation complexity. However, some pitfalls soon emerged.

Raft is designed to be highly serialized for simplicity and understandability. Its logs are not allowed to have holes on both leader and followers, which means log entries are acknowledged by follower, committed by leader and applied to all replicas in sequence. So when write requests execute concurrently, they would be committed in sequence. Those requests at the tail of queue cannot be committed and responded until all preceding requests are persistently stored to disks and answered, which increases average latency and cuts down throughput. We observed the throughput dropping by half as I/O depth rising from 8 to 32.

Raft is not quite suitable for the environment using multiple connections to transfer logs between a leader and a follower. When one connection blocks or becomes slow, log entries would arrive at follower out of order. In other words, some log entries in the front of queue would actually arrive later than those at the back. But a Raft follower must accept log entries in sequence, which means it cannot send an acknowledge to inform the leader that subsequent log entries have already been recorded to disk, until those preceding missing log entries arrived. Moreover, The leader would get stuck when most followers are blocked on some missing log entries. In practice, however, using multiple connections is common for a highly concurrent environment. We need to revise the Raft protocol to tolerate this situation.

In a transactional processing system such as database, concurrency control algorithms allow transactions to be executed in an interleaved and out-of-order way while generating serializable results. These systems naturally can tolerate out-of-order I/O completions arising from traditional storage semantics and address it by themselves to ensure data consistency. Actually, databases such as MySQL and AliSQL do not care about the I/O sequences of underlying storage. The lock system of a database will guarantee that at any point of time, only one thread can work on one particular page. When different threads work on different pages concurrently, database only need the I/Os be executed successfully, their completion order would not matter. Therefore we capitalize on that to relax some constraints of Raft in PolarFS to develop a consensus protocol which is more suitable for high I/O concurrency.

This paper proposes an improved consensus protocol over Raft named ***ParallelRaft***, and the following sections will present how ParallelRaft solves the above problems. The structure of ParallelRaft is quite similar to Raft. It implements the replicated state machine through a replicated log. There are leader and followers, and leader replicates log entries to followers. We follow the same approach of problem decomposition like Raft, and divide ParallelRaft into smaller pieces: log replication, leader election and catch up.

5.2 Out-of-Order Log Replication

Raft is serialized in two aspects: (1) After the leader sends a log entry to a follower, the follower needs to acknowledge it to inform that this log entry has been received and recorded, which also implicitly indicates that all preceding log entries have been seen and saved. (2) When the leader commits a log entry and broadcasts this event to all followers, it also admits all preceding log entries have been committed. ParallelRaft breaks these restrictions and executes all these steps out-of-order. Therefore, there is a basic difference between ParallelRaft and Raft. In ParallelRaft, when an entry has been recognized to be committed, it does not mean that all the previous entries have been committed successfully. To ensure the correctness of this protocol, we have to guarantee: (1) Without these serial restrictions, all the committed states will also be complaint with the storage semantics used by classical rational databases. (2) All committed modifications will not get lost in any corner cases.

The out-of-order log executing of ParallelRaft follows the rules: If the writing ranges of the log entries are not overlapped with each other, then these log entries are considered without conflict, and can be executed in any order. Otherwise, the conflicted entries will be executed in a strict sequence as they arrived. In this way, the newer data will never be overwritten by older versions. ParallelRaft can easily know the conflict because it stores the LBA range summary of any log entry not applied . The following part describes how the Ack-Commit-Apply steps of ParallelRaft are optimized and how the necessary consistent semantics can be maintained.

Out-of-Order Acknowledge. After receiving a log entry replicated from the leader, a Raft follower would not acknowledge it until all preceding log entries are stored persistently, which introduces an extra waiting time, and the average latency increases significantly when there are lots of concurrent I/O writes executing. However, in ParallelRaft, once the log entry has been written successfully, the follower can acknowledge it immediately. In this way, the extra waiting time is avoided, so that the average latency is optimized.

Out-of-Order Commit. A raft leader commits log entries in serial order, and a log entry cannot be committed until all preceding log entries are committed. Whereas in ParallelRaft a log entry can be committed immediately after a majority of replicas have been acknowledged. This commit semantics is acceptable for a storage system which usually do not promise strong consistency semantics like a TP system. For example, NVMe does not check the LBA of read or write commands to ensure any type of ordering between concurrent commands and there is no guarantee of the order of completion for those commands [13].

Apply with Holes in the Log. As Raft, all log entries are applied in the strict sequence as they are logged in ParallelRaft, so that data file is coherent in all replicas. However, with out-of-order log replication and commit, ParallelRaft allows holes to exist in the log. How can a log entry be applied safely with some preceding log entries still missing? It brings a challenge to ParallelRaft.

ParallelRaft introduces a novel data structure named *look behind buffer* in each log entry to address this issue. The look behind buffer contains the LBA modified by the previous N log entries, so that a look behind buffer acts as a bridge built over a possible hole in the log. N is the span of this bridge, which is also the maximum size of a log hole permitted. Please note that although several holes might exist in the log, the LBA summary of all log entries can always be complete unless any hole size is larger than N. By

means of this data structure, the follower can tell whether a log entry is conflicting, which means that the LBAs modified by this log entry are overlapping with some preceding but missing log entries. Log entries not conflicting with any other entry can be applied safely, otherwise they should be added to a pending list and replied after the missing entries are retrieved. According to our experience in PolarFS with RDMA network, N set to 2 is good enough for its I/O concurrency.

Based on the above Out-of-Order execution methods and rules, the wanted storage semantics for databases can be successfully achieved. Moreover, the latency of multi-replica concurrent write can also be shortened by eliminate unnecessary serial restrictions in ParallelRaft for PolarFS.

5.3 Leader Election

When doing new leader election, ParallelRaft could choose the node which has the newest term and the longest log entry, just the same as Raft. In Raft, the newly elected leader contains all the committed entries from previous terms. However, the leader elected in ParallelRaft might not initially meet this requirement because of possible holes in the log. Hence an additional merge stage is needed to make the leader have all the committed entries before starting processing requests. Before the merge stage finishes, the new elected node is only leader candidate, after the merge stage finished, it has all the committed entries and becomes the real leader. In the merge stage, the leader candidate needs to merge unseen entries from rest members of a quorum. After that, the leader would start to commit entries from previous terms to the quorum, which is same as Raft.

When merging entries, ParallelRaft also uses a similar mechanism as Raft. An entry with the same term and index will be promised to be the same entry. There are several abnormal conditions: (1) For a committed but missing entry, the leader candidate can always find the same committed entry from at least one of the follower candidates, because this committed entry has been accepted by the majority of quorum. (2) For an entry that is not committed on any of the candidate, if this entry is not saved on any of them either, the leader can skip this log entry safely since it cannot have been committed previously according to the ParallelRaft or Raft mechanism. (3) If some candidates saved this uncommitted entry (with same index but different term), then leader candidate chooses the entry version with the highest term, and recognizes this entry as a valid entry. We must do this because of the other two facts: (3.a) ParallelRaft's merge stage must be done before a new leader can serve for user requests, which determines that if a higher term is set to an entry, the lower one with the same entry index must not have been committed before, and the lower term entry must have never attended to previous successfully completed merge stages, otherwise the higher term entry cannot be with the same index. (3.b) When a system crashed with a uncommitted entry saved in a candidate, this entry acknowledgment may have been sent to the previous leader and replied to the user, so we cannot simply abandon it otherwise user data might get lost. (More precisely, if the total number of failed nodes plus nodes with this uncommitted entry (the highest term of these entries with same index) exceeds the number of other nodes in the same quorum, then this entry might have committed by the failed leader. Therefore, we should commit it for user data safety.)

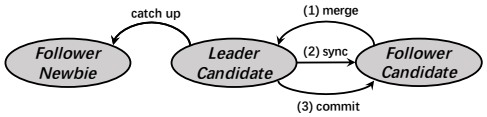


Figure 5: ParallelRaft Leader Election.

Take the situation with 3 replicas for example, Figure 5 shows the process of leader election.

First, the follower candidate sends its local log entries to the leader candidate. The leader candidate receives these entries and merges with its own log entries. Second, the leader candidate synchronizes states to the follower candidate. After that, the leader candidate can commit all the entries and notify the follower candidate to commit. Finally the leader candidate upgrades to a leader, and the follower candidate upgrades to a follower as well.

With above mechanisms, all committed entries can be recovered by the new leader, which means that ParallelRaft will not lost any committed state.

In ParallelRaft, a *checkpoint* is made from time to time, all log entries before the checkpoint are applied to data blocks and the checkpoint is allowed to contain some log entries committed after the checkpoint. For simplicity, all log entries before the checkpoint can be regarded as pruned, although in practice they are reserved for a while until short of resource or some threshold reached.

In our real implementation, ParallelRaft choose the node with newest checkpoint as the leader candidate instead of the one with the longest log, for the sake of the Catch Up implementation. With the merge stage, it is easy to see that the new leader will reach the same state when starting serving. Therefore, this choice does not compromise the correctness of ParallelRaft.

5.4 Catch Up

When a lagging follower wants to keep up with the leader's current data state, it will use *fast-catch-up* or *streaming-catch-up* to re-synchronize with the leader. Which one is used depends on how stale the state of the follower is. The *fast catch up* mechanism is designed for incremental synchronization between leader and follower, when the differences between them are relatively small. But the follower could lag far behind the leader. For instance, the follower has been offline for days, so a full data re-synchronization is inevitable. PolarFS thus proposes a method named *streaming catch up* for this task.

Figure 6 presents the different conditions of leader and follower when re-synchronization starts. In Case 1, leader's checkpoint is newer than the follower's latest log index, and log entries between them would be pruned by leader. Therefore, *fast catch up* cannot handle this situation, and *streaming catch up* should be used instead. Case 2 and Case 3 can be fixed by *fast catch up*.

In the following cases, we can always assume that the leader's checkpoint is newer than the follower's, since if this condition is not met, the leader can make a checkpoint immediately which is newer than any other.

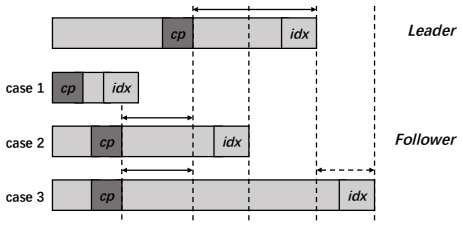
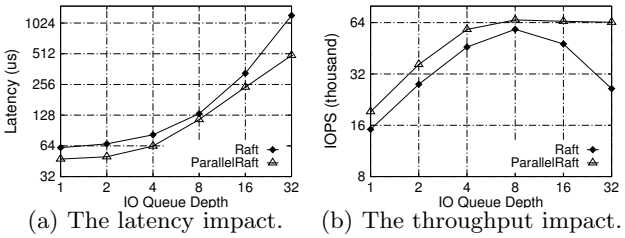


Figure 6: ParallelRaft Fast Catch Up Process.



(a) The latency impact. (b) The throughput impact.

Figure 7: Performance of Raft and ParallelRaft for 4KB random write request.

Log entries after the checkpoint can be divided into four classes: committed-and-applied, committed-but-unapplied, uncommitted, and holes.

Fast Catch Up. Follower may have holes after its checkpoint, which will be filled during *fast catch up*. First, The holes between follower's checkpoint and leader's checkpoint are filled with the help of *look behind buffer* again, through which we shall discover LBAs of all missing modifications. These holes are filled directly by copying from leader's data blocks, which contains data newer than follower's checkpoint. Second, the holes after leader's checkpoint are filled by reading them from leader's log blocks.

In Case 3, the follower may contain uncommitted log entries from some older terms. Like Raft, these entries are pruned, and will then be handled following the above steps.

Streaming Catch Up. In streaming catch up, log histories older than the checkpoint are deemed useless for full data re-synchronization, and the content of data blocks and log entries after the checkpoint are all must be used to rebuild the state. When copying the chunks, a whole chunk is split into small 128KB pieces and using lots of small tasks, each of which only syncs a single 128 KB piece. We do this for establishing a more controllable re-synchronization. The further design motivation is discussed in Section 7.3 .

5.5 Correctness of ParallelRaft

Raft ensures the following properties for protocol correctness: *Election Safety*, *Leader Append-Only*, *Log Matching*, *Leader Completeness*, and *State Machine Safety*. It is easy to prove that ParallefRaft has the property of **Election Safety**, **Leader Append-Only**, and **Log Matching**, since ParallelRaft does not change them at all.

ParallelRaft's out-of-order commitment introduces the key difference from standard Raft. A ParallelRaft log might lack some necessary entries. Therefore, the key scenario we need to consider is how a new elected leader should deal with these missing entries. ParallelRaft add a merge stage for leader electing. In the merge stage, the leader will copy the missing log entries, re-confirm whether these entries should be committed, and then commit them in the quorum if necessary. After the merge stage in Section 5.3, the **Leader Completeness** is guaranteed.

In addition, as mentioned in Section 5.2, the out-of-order commitment is acceptable in PolarFS for databases.

Then we will prove that the out-of-order apply will not violate the **State Machine Safety** property. Although ParallelRaft allows nodes do out-of-order apply independently, thanks to the look behind buffer (Section 5.2), the conflicting logs can only be applied in a strict sequence, which means that the state machine (data plus committed log entries) on all nodes in the same quorum will be consistent with each other.

With all these properties, the correctness of ParallelRaft is guaranteed.

5.6 ParallelRaft Versus Raft

We did a simple test to show how ParallelRaft can improve the I/O concurrency. Figure 7 shows the average latency and throughput of Raft and ParallelRaft when I/O depth varies from 1 to 32 and single FIO job. Both algorithms are implemented using RDMA. The slight performance difference at the beginning can be neglected because they are different software packages implemented independently. As the I/O depth increases, the performance gap between two protocols becomes wider. When I/O depth grows to 32, the latency of Raft is approximately 2.5X times of the latency of ParallelRaft, with less than half of the IOPS ParallelRaft achieves. It is noticeable that the IOPS of Raft drops considerably when I/O depth is larger than 8, whereas ParallelRaft maintains the overall throughout at a steady high level. The results demonstrate the effectiveness of our proposed optimization mechanisms in ParallelRaft. The out-of-order acknowledgement and commitment improve the I/O parallelism, which enables PolarFS to maintain superior and stable performance even under heavy workload.

6. FS LAYER IMPLEMENTATION

In PolarFS's file system layer, the metadata management could be divided into two parts. The first one is to organize metadata to access and update files and directories within a database node. The second one is to coordinate and synchronize metadata modification among database nodes.

6.1 Metadata Organization

There are three kinds of file system metadata: directory entry, inode and block tag. A single directory entry keeps a name component of a file path and has a reference to an inode. A collection of directory entries are organized into a directory tree. An inode describes either a regular file or a directory. For a regular file, the inode keeps a reference to a set of block tags, each describing the mapping of a file block number to a volume block number; for a directory, the inode keeps a reference to a set of subdirectory entries in this parent directory.(The references mentioned above are actually metaobject identifiers as explained below).

The three kinds of metadata are abstracted into one data type called metaobject which has a field holding the specific data for a directory entry, an inode or a block tag. This common data type is used to access metadata both on disk

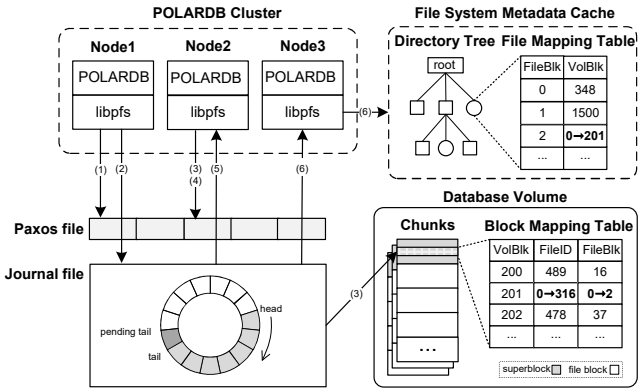


Figure 8: Overview of File System Layer.

and in memory. Metaobjects are initialized on disk in continuous 4k size sectors when making a new file system, with each metaobject assigned a unique identifier. In pfs_mount, all metaobjects are loaded into memory and partitioned into chunks and type groups.

A metaobject is accessed in memory with a tuple (metaobject identifier, metaobject type). The upper bits of the identifier is used to find the chunk to which the metaobject belongs, the type is to find the group in the chunk, and at last the lower bits of the identifier is used as the index to access the metaobject in the group.

To update one or more metaobjects, a transaction is prepared and each update is recorded as a transaction operation. The transaction operation holds the old value and new value of its modified object. Once all updates are done the transaction is ready to be committed. The commit process needs coordination and synchronization among database nodes, as described in the following subsection. If the commit failed, the update is rollbacked with old value saved in the transaction.

6.2 Coordination and Synchronization

To support synchronization of file system metadata, PolarFS records metadata modification as transactions into a journal file.The journal file is polled by database nodes for new transactions. Once found, new transactions are retrieved and replayed by the nodes.

Normally, there is only one instance to write the journal file and multiple instances to read it. In the case of network partition or administration, there may be multiple instances writing to the journal file. In these cases, we need a mechanism to coordinate writes to the journal file. PolarFS uses the disk paxos algorithm [10] for this purpose. Note that the purpose of the disk paxos algorithm used here is quite different from the ParallelRaft. The latter is to ensure data consistency among the chunk replicas.

The disk paxos algorithm runs on a file composed of 4K size pages which can be atomically read or written. These pages are interpreted as one leader record plus data blocks for each database node. The data block pages are used by each database node to write the proposal of itself and to read the proposals of others. The leader record page contains information of current paxos winner and the log anchor for the journal. The disk paxos algorithm is only run by a write node. The read-only nodes do not run disk paxos. Instead they poll by checking the log anchor in the leader record page. If the log anchor changes, the read-only nodes know there are new transactions in the journal file and they will retrive these new transactions to update its local metadata.

We explain a transaction commit process with an example, as illustrated in Figure 8. It has the following steps:

(1) Node 1 acquires paxos lock, which is initially free, after assigning block 201 to file 316. (2) Node 1 starts recording transactions to journal. The position of the latest written entry is marked with *pending tail*. After all entries are stored, the pending tail becomes the valid tail of journal. (3) Node 1 updates the superblocks with the modified metadata. At the same time, Node 2 tries to obtain the mutex which is already held by Node 1. Node 2 must fail and will retry later. (4) Node 2 becomes the legal owner of mutex after Node 1 releases the lock, but the new entries in journal appended by Node 1 determine that the local metadata cache in Node 2 is stale. (5) Node 2 scans new entries and releases the lock. Then Node 2 rollbacks unrecorded transactions and updates local metadata. Finally Node 2 retries the transactions. (6) Node 3 starts to synchronize metadata automatically and it only needs to load incremental entries and replay them in local memory.

7. DESIGN CHOICES AND LESSONS

Besides the system architecture, the I/O model and the consensus protocol, there are still some interesting design topics worth discussing. Some of them belong to PolarFS itself, and others are aimed for database features.

7.1 Centralized or Decentralized

There are two design paradigms for distributed systems: *centralized* and *decentralized*. Centralized systems, such as GFS [11] and HDFS [5], contain a single master, which is responsible for keeping metadata and the membership management, This kind of systems are relatively simple to implement, but the single master may become a bottleneck of the whole system, in the sense of both availability and scalability. Decentralized systems like Dynamo [8] are in the opposite way. In this system, nodes are of peer-to-peer relationship, metadata are sharded and redundant among all nodes. A decentralized system is supposed to be more reliable but also more complex to implement and reasoning.

PolarFS makes a trade-off between centralized and decentralized designs. On one hand, PolarCtrl is a centralized master which is responsible for administration tasks such as resource management, and accepting control plane requests like creating volume. On the other hand, ChunkServers are talking with each other, running a consensus protocol to handle failure and recovery autonomously without PolarCtrl involved.

7.2 Snapshot from Bottom to Top

Snapshot is a common need for databases. PolarFS supports snapshot feature, which simplifies upper POLARDB snapshot design.

In PolarFS, the storage layer provides reliable data access service to upper layers, POLARDB and libpfs have their own log mechanisms to ensure their transaction atomicity and durability. PolarFS storage layer provides a snapshot of *disk outage consistency*, and POLARDB and libpfs rebuild their own consistent data image from the underlying PolarFS snapshot.

The *disk outage consistency* here means that if the snapshot command is triggered at a time point T, then some time point T_0 exists so that all the I/O operations before T_0 should be included in the current snapshot, and the I/O operations after T must be excluded. However, the states of I/O operations within the interval $[T_0, T]$, which is usually a very short time span, are undetermined. This behavior resembles what happens upon power off while a disk is still being written.

When an unexpected disaster happened or an active auditing is need, PolarCtrl assigns the newest data snapshot to the POLARDB instance, POLARDB and libpfs will use their logs stored to rebuild a consistent state.

PolarFS implements the *disk outage consistency* snapshot in a novel way, which does not block user I/O operations during snapshot is being made. When user issues a snapshot command, PolarCtrl notifies PolarSwitch to take a snapshot. From then on, PolarSwitch adds a snapshot tag, which indicates its host request happened after the snapshot request, to the following requests. On receiving the request with a snapshot tag, ChunkServer will make a snapshot before handling this request. ChunkServer makes a snapshot by copying block mapping meta information, which is fast, and handles the future requests which will modify those blocks in a Copy-on-Write way. After the request with snapshot tag is finished, PolarSwitch stops adding snapshot tag to incoming requests.

7.3 Outside Service vs. Inside Reliability

Reliability of an industrial system is extremely important, especially for the system like PolarFS, which is undertaking 7X24 public cloud services. In such a system, there should be all sorts of reliability maintaining tasks to prevent loss of service and related revenue. It is a big challenge for PolarFS to run these tasks efficiently enough while providing smooth service for heavy user workloads.

For a practical example, the *streaming catch up* in ParallelRaft, when undergoing a heavy workload, the leader will keep generating an overwhelming amount of logs that the follower cannot catch up even after a long period of log fetching. The time to copy the whole chunk can also be quite long and unpredictable due to I/O throttling. That can lead us to a trade-off: the shorter the recovery time is, the more resources it takes, and the more sacrifice to system performance; whereas if it takes a long time to recover, the system availability would be at risk.

Our solution is to horizontally split one chunk into small logical pieces, say, pieces of 128 KB. The full data synchronization for a chunk is also split into lots of small subtasks, each of which only resynchronizes a single 128 KB piece. The runtime for these smaller subtasks is much shorter and more predictable. Besides, idle cycles can be inserted between sub-resync-tasks to control the amount of network/disk bandwidth cost for *streaming catch up*.

Other time-consuming tasks, such as full data validation routines checking coherency between replicas, can also be implemented similarly. We have to consider using synchronization flow control to strike a balance between the quality of external service and the system reliability of PolarFS. Due to space limitations we will not further discuss the details of this trade-off here.

8. EVALUATION

We have implemented PolarFS and have also released POLARDB as a public database service on Alibaba cloud. In this section, we evaluated and analyzed PolarFS and POLARDB performance and scalability. For PolarFS, we evaluated it on a cluster by comparing it with Ext4 on local NVMe and CephFS on Ceph [35], a widely used open source distributed storage system. For POLARDB, we compared it with our original MySQL cloud service RDS and POLARDB on local Ext4.

8.1 PolarFS Experimental Setup

For file system performance, we focused on the end-to-end performance of three systems (PolarFS, CephFS and Ext4), including latency and throughput under different workloads and access pattern.

The experiment results of PolarFS and CephFS are collected on a cluster of 6 storage nodes and one client node. Nodes communicates with each other through a RDMA-enabled NIC.

Ceph version is 12.2.3, and we configure its storage engine as bluestore and communication massager type as async + posix. Ceph's storage engine can be configured to run with RDMA verbs, but its file system only supports the TCP/IP network stack, so we configure CephFS to run with TCP/IP. For Ext4, we make a new fresh Ext4 on a SSD after discarding all its old data.

We use FIO [1] to generate various types of workloads with different I/O request sizes and parallelism. FIO will firstly create the files and extend them to the given length (here is 10G) before its performance evaluation.

8.2 I/O Latency

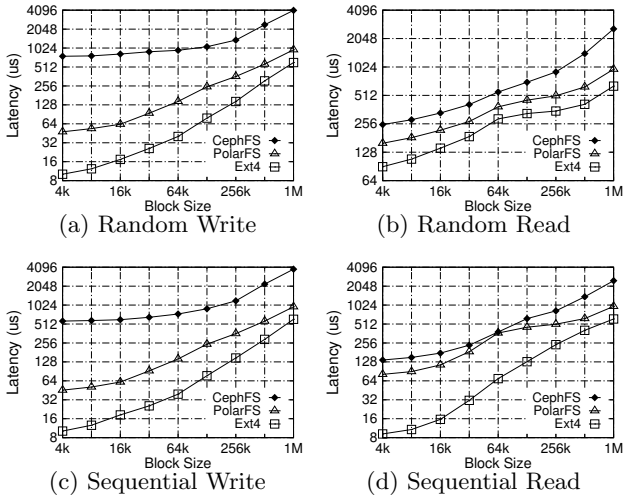


Figure 9: Average latency with different request sizes under different access patterns.

As Figure 9 shows, the latency of PolarFS is about 48μs for 4k random write, which is quite close to the latency of Ext4 on local SSD (about 10μs) compared with the latency of CephFS (about 760μs). The average random write latency of PolarFS is 1.6 to 4.7 times slower than local Ext4, while that of CephFS is 6.5 to 75 times slower than local Ext4, which means distributed PolarFS almost provides the

same performance like local Ext4. The average sequential write latency ratios of PolarFS and CephFS to Ext4 are 1.6-4.8 and 6.3-56. The average random read latency ratios of PolarFS and CephFS to Ext4 are 1.3-1.8 and 2-4. And the average sequential read latency ratios of PolarFS and CephFS to Ext4 are 1.5-8.8 and 3.4-14.9.

Large request (1M) performance reduction of PolarFS / CephFS for Ext4 differs from small request (4k) because the network and disk data transfer take up most of large request execution time.

PolarFS performance is much better than that of CephFS, there are several reasons for this. First, PolarFS processes I/O with a finite state machine by one thread, as described in Section 4. This avoids thread context switch and rescheduling required by an I/O pipeline composed by multiple threads as done by CephFS. Second, PolarFS optimizes memory allocation and paging, memory pool is employed to allocate and free memory during an I/O life cycle, reducing object construction and destruction operations, huge page is also used to reduce TLB miss and paging, there is no counterpart in CephFS. Third, all meta data about data objects in PolarFS are kept in memory, contrasted with that only partial meta data are kept in a small cache for each PG(Placement Group) in CephFS, thereby PolarFS needs no extra I/O for meta data for most I/O operation, but CephFS sometimes has to do extra I/Os for meta data. Last but not least, user space RDMA and SPDK in PolarFS have lower latency than kernel TCP/IP and block driver in CephFS.

8.3 I/O Throughput

Figure 10 shows the overall I/O throughput of three file systems. For the random read/write, Ext4 on local SSD, PolarFS and CephFS IOPS all scale with the client jobs number; Ext4's scalability bottleneck is the single local SSD IOPS, that of PolarFS is network bandwidth, while CephFS's bottleneck is its packet processing capacity. The random 4k write/read I/O throughput of Ext4 and PolarFS are 4.4/5.1, 4/7.7 higher than CephFS.

For the sequential read/write, almost all requests are handled by a single disk both for the local system and for the distributed systems. Ext4 and PolarFS all can scale well with the number of client jobs until they have reached their limit due to the I/O bottleneck, while CephFS's bottleneck is its software, it can not saturate the disk I/O bandwidth.

Without extra network I/O, Ext4's sequential read throughput with 1 client is much higher than PolarFS and CephFS, but its sequential read throughput drops dramatically with client number increasing to 2, the throughput is about the same value as its random read throughput with 2 clients. We repeated the evaluation for several times and the result is reproducible. We guess it is caused by the characteristic of our NVMe SSD. The NVMe SSD has a built-in DRAM buffer. When the firmware guesses the workload looks like sequential read, it would prefetch subsequent data blocks into the DRAM buffer. We guess the prefetch mechanism would work much better with non-interleaved I/O pattern.

8.4 Evaluating POLARDB

To demonstrate the benefits of our ultra-low I/O latency PolarFS for databases, we carried out some preliminary tests on POLARDB, a share-storage cloud database specifically developed with PolarFS. We compared the throughput of three systems: (1) POLARDB on PolarFS, (2) POLARDB on local Ext4, and (3) Alibaba MySQL cloud service RDS for reference. Tests (1) and (2) are under the same hardware environment as the previous PolarFS hardware environment.

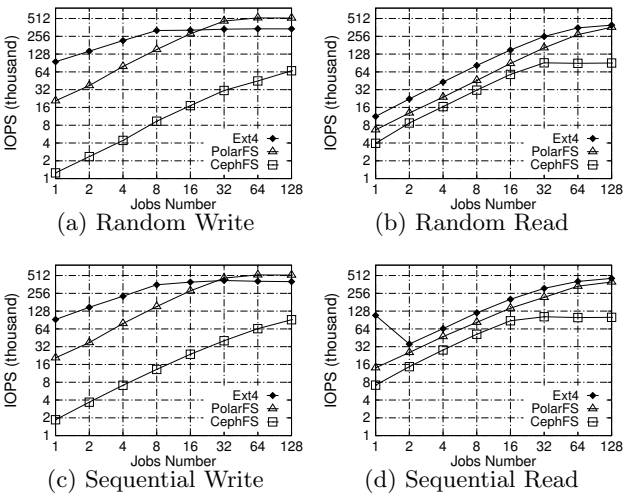


Figure 10: Throughput with 4K request sizes under different access patterns.

We separately evaluated three databases by running Sysbench [21] under simulated read-only, write-only (update : delete : insert = 2:1:1) and read/write-mixed (read : write = 7:2) OLTP workloads. The dataset used in this experiment consists of 250 tables, and each table has 8,500,000 records. The total size of the tested data is 500 GB.

As shown in Figure 11, the throughput of POLARDB on PolarFS is very close to that on the local Ext4, while PolarFS provides additional 3-replica high availability. POLARDB on PolarFS achieved 653K read/sec, 160K write/sec and 173K read-write/sec.

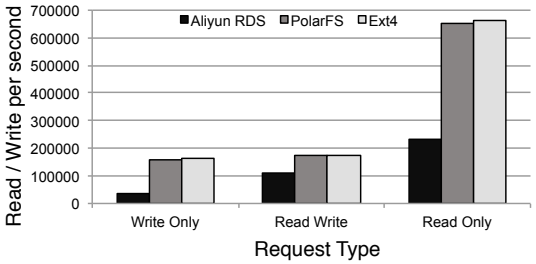


Figure 11: Performance comparison for Aliyun RDS, PolarDB on PolarFS and PolarDB on Ext4 using read/write executed per second as metric.

In addition to integrating with PolarFS, POLARDB also applied a bunch of database optimizations based on Aliyun RDS. As a result, POLARDB achieved almost 2.79/4.72/1.53 times higher Read/Write/RW-Mixed throughput than RDS as shown in the figure. Database-level optimization is beyond the scope of this paper.

9. RELATED WORK

Storage System. GFS [11] and its open source implementation HDFS [5], provides the distributed system service, both GFS and HDFS adopt the master slave archi-

tecture, the master maintains the system meta information, data chunk leader lease, and it is also in charge of the failure recovery. Unlike GFS and HDFS, PolarFS adopts a hybrid architecture. The master node of PolarFS is mainly in charge of resource management, it generally does not add interference in the I/O path, which makes master upgrade and fault tolerance easier.

Ceph [35] is a widely deployed storage systems. It uses CRUSH hash algorithm to locate data object which brings a simple design for both normal I/O execution and fault tolerance. RAMCloud [29] is a low latency key value system which provides fast recovery by replicating data among the cluster in a random way and harnessing the cluster resource to recover failure in parallel. The random data placement will make a machine hosting thousand of users' data in cloud environment, this means a single machine crash will impact thousand of users, which is unacceptable for cloud service providers. PolarFS control master provides a fine-grained trade-off between failure impacting and I/O performance during allocating data chunks to real servers.

ReFlex [20] uses NVMe SSD to implement a remote disk, it proposes a QoS scheduler that can enforce tail latency and throughput service-level objectives, but unlike PolarFS, its remote disk is single replica and has no reliability support. CORFU [3] organizes a cluster of flash devices as a single, shared log that can be accessed concurrently by multiple clients over the network. The shared log design simplifies snapshot and replication in distributed environment, but it raises the engineering difficulty for the existing system to use CORFU, otherwise PolarFS provides a block disk abstraction and POSIX-like file system interfaces to users.

Aurora [32] is Amazon's cloud relational database service on the top of its ECS service. Aurora addresses the network performance constraint between the computation side and storage side by pushing database redo log processing to the storage side. Aurora also modified MySQL's innodb transaction manager to facilitate log replications, whereas POLARDB makes heavy investment in PolarFS but with POSIX-like API which ends up with little modifications to the database engine itself. Aurora's redo logic pushing down design breaks the abstract between database and storage, which make each layer modification harder, while PolarFS and POLARDB keep each layer's abstraction, which makes each layer adopt its newest technologies more easily.

Consensus Protocols. Paxos [22] is one of the most famous consensus algorithm for distributed system. However, it is hard to understand and correctly implement. The author of Paxos only proved that Paxos can solve consensus problem for one instance, and not show how to use "Multi-Paxos" to solve practical problems. Currently, most of the consensus algorithms, such as Zab [16] used by zookeeper, are regarded as the variety of the "Multi-Paxos" algorithm. Zab is used to provide the atomic broadcast primitive property, but it is also hard to understand. Raft [28] is designed for understandability, which is also a variety of "Multi-Paxos" algorithm. It introduces two main constraints, the first one is that the commit log must be sequential, and the second one is that the commit order must be serialized. These two constraints make Raft easily understandable, but also introduce impact on concurrent I/O performance. Some systems use multi groups raft to improve I/O parallelism, which splits the keys into groups, but it cannot solve the problem that a commit log include keys crossing many groups. ParallelRaft achieve this goal through a different way. The commit log is allow to have holes and the log entries can be committed out of order except for the conflicting ones, which will be committed in a strict order.

RDMA Systems. Compared with the traditional TCP / UDP network protocol, RDMA reduces round trip latency by a order of magnitude with less CPU utilization, a lot of new systems use RDMA to improve performance. FaRM [9] is a distributed shared memory based on RDMA, users can use its program primitives to build their application. DrTM [34] is a distributed in-memory database based on RDMA and hardware transaction memory, RDMA can access data from remote side fast and can also cause the local transaction in hardware transaction memory abort, DrTM combines RDMA and hardware transaction memory to implement distributed transaction. Both FaRM and DrTM are focused on in-memory transaction, however PolarFS uses RDMA to build a distributed storage.

Pilaf [27] is a key value store based on RDMA and self-verifying data structure, but unlike PolarFS, the key value interface is relatively simple and there is no distributed consensus protocol in Pilaf.

APUS [33] uses RDMA to build a scalable Paxos, while PolarFS provides a Parallel raft which is friendly to high parallelism I/O. PolarFS use RDMA to build a reliable block device, since RDMA consumes less CPU than traditional network protocol, PolarFS can process I/O in a run-to-completion way which avoids context switch. Combining RDMA and SPDK, storage node can finish I/O without touching its payload during the whole I/O life cycle, all the payload movement is done by the hardware.

10. CONCLUSION

PolarFS is a distributed file system which delivers extreme performance with high reliability. PolarFS adopts emerging hardware and state-of-the-art optimization techniques, such as OS-bypass and zero-copy, allowing it to have the latency comparable to a local file system on SSD. To meet the high IOPS requirements of database applications, we develop a new consensus protocol, ParallelRaft. ParallelRaft relaxes Raft's strictly sequential order of write without sacrificing the consistency of storage semantics, which improves the performance of parallel writes of PolarFS. Under heavy loads, our approach can halve average latency and double system bandwidth. PolarFS implements POSIX-like interfaces in user space, which enables POLARDB to achieve performance improvement with minor modifications.

11. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Theodore Johnson for their valuable suggestions and opinions that helped improve the paper. We also express our gratitude to JingXian Cai, Feng Yu, DongLai Xu, ZunBao Feng, Liang Yin, ChengZhong Yang, QingAn Fan, Xiao Lou, ZhenDong Huang, ZhuShi Cheng, Xi Chen, Ming Zhao, GuangQing Dong, Jian Chen, Jian Zhang, WeiXiang Zhai, ZongZhi Chen, Jun He, GaoSheng Xu, HongYang Shi, FeiFei Fan, Tao Ma, JingXuan Li, Gang Deng, Zheng Liu, ShiKun Tian, HaiPing Zhao, Bing Lang, and Kai Ma who offered significant help for this work.

12. REFERENCES

[1] Fio: Flexible I/O tester. `https://github.com/axboe/fio`.

[2] Alibaba Group. Alisql. `https://github.com/alibaba/AliSQL`.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *NSDI*, pages 1–14, 2012.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[5] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[9] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[10] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[12] Intel. Intel storage performance development kit. `http://www.spdk.io`.

[13] Intel. Nvm express revision 1.1. `http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf`.

[14] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 236–243, 2012.

[15] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, et al. Memcached design on high performance rdma capable interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743–752, 2011.

[16] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, 2014.

[18] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 3. ACM, 2008.

[19] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[20] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 345–359. ACM, 2017.

[21] A. Kopytov. Sysbench: a system performance benchmark. *URL: http://sysbench. sourceforge. net*, 2004.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[23] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[24] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, 2015.

[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. USENIX NSDI*, 2014.

[26] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[27] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[28] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[29] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[30] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX Annual Technical Conference*, pages 347–353, 2012.

[31] S. Swanson and A. M. Caulfield. Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage. *IEEE Trans. Computers*, 46(8):52–59, 2013.

[32] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design

considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[33] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107. ACM, 2017.

[34] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[35] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[36] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[37] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Computer Systems*, 32(2), 2014.

[38] J. Zheng, Q. Lin, J. Xu, C. Wei, C. Zeng, P. Yang, and Y. Zhang. Paxosstore: high-availability storage made practical in wechat. *PVLDB*, 10(12):1730–1741, 2017.