RubbleDB: CPU-Efficient Replication with NVMe-oF

Haoyu Li, Sheng Jiang, and Chen Chen, *Columbia University;* Ashwini Raina, *Princeton University;* Xingyu Zhu, Changxu Luo, and Asaf Cidon, *Columbia University*

https://www.usenix.org/conference/atc23/presentation/li-haoyu

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

RubbleDB: CPU-Efficient Replication with NVMe-oF

Haoyu Li[1], Sheng Jiang[1], Chen Chen[1], Ashwini Raina[2], Xingyu Zhu[1], Changxu Luo[1], and Asaf Cidon[1]

[1]*Columbia University*, [2]*Princeton University*

Abstract

Due to the need to perform expensive background compaction operations, the CPU is often a performance bottleneck of persistent key-value stores. In the case of replicated storage systems, which contain multiple identical copies of the data, we make the observation that CPU can be traded off for spare network bandwidth. Compactions can be executed only once, on one of the nodes, and the already-compacted data can be shipped to the other nodes' disks, saving them significant CPU time. In order to further drive down total CPU consumption, the file replication protocol can leverage NVMe-oF, a networked storage protocol that can offload the network and storage datapaths entirely to the NIC, requiring zero involvement from the target node's CPU. However, since NVMe-oF is a one-sided protocol, if used naively, it can easily cause data corruption or data loss at the target nodes.

We design RubbleDB, the first key-value store that takes advantage of NVMe-oF for efficient replication. RubbleDB introduces several novel design mechanisms that address the challenges of using NVMe-oF for replicated data, including pre-allocation of static files, a novel file metadata mapping mechanism, and a new method that enforces the order of applying version edits across replicas. These ideas can be applied to other settings beyond key-value stores, such as distributed file and backup systems. We implement RubbleDB on top of RocksDB and show it provides consistent CPU savings and increases throughput by up to 1.9× and reduces tail latency by up to 93.4% for write-heavy workloads, compared to replicated key-value stores, such as ZippyDB, which conduct compactions on all replica nodes.

1 Introduction

To provide high availability, disk-based key value stores are often replicated on multiple machines [2, 21, 22, 25, 35, 43]. A standard architecture for replicating key-value stores is that each machine runs a local key-value instance, and a replication layer controls which replica gets shipped to each instance.

However, key-value stores spend a significant portion of their computing resources on background compaction operations, which rebalance and garbage-collect the data on disk. For example, in the case of log-structured merge trees (LSM trees), the standard disk-based key-value store design [2, 4, 8, 22, 41], previous work has shown that compaction can consume up to 45% of CPU in production workloads, and by avoiding compaction, key-value stores can increase their throughput by up to 2× [12]. We have reproduced these experiments and find that with RocksDB, compaction consumes up to 72% of the total CPU cycles.

This leads to the simple observation that, in the case of replicated key-value stores, where each node sees identical commands, the compaction operations conducted on each machine that stores the replica of the data represent *redundant effort*. Therefore, we can design an architecture, where a *primary* node conducts compaction operations locally, and then ships the already-compacted files to the *secondary* nodes that store the data copies, thereby significantly reducing their CPU consumption.

However, such an approach has two important drawbacks. First, it increases the amount of network traffic because not only do the regular operations need to be replicated, but also the compacted files. Fortunately, network traffic in modern datacenters is often underutilized; for example, cluster traces from Alibaba [1] and Snowflake [47] show that 50-–75% of network capacity consistently remain idle. Therefore, reducing CPU consumption at the expense of additional network traffic is often a desirable trade-off. Second, shipping the files from the primary to the secondary nodes still requires some processing from both: at the extreme, if both ends use TCP, then shipping the files will incur the cost of processing the TCP packets on both ends, as well as the cost of traversing the storage stack on the secondary nodes.

To address the second problem, we turn to NVMe-oF, a networked storage protocol that minimizes CPU costs at secondary nodes. NVMe-oF extends the NVMe protocol to allow one server to access a disk of a remote server directly, with minimal involvement of the remote server's CPU. Even better, most commodity datacenter NICs support offloading the entire NVMe processing at the remote server, by allow-

ing the remote NIC to talk directly to the NVMe storage device. Therefore, if we use NIC-offloaded NVMe-oF, the secondary's host CPU will not be involved at all in processing the incoming replicated files, thereby completely eliminating all of its CPU costs due to compaction.

However, using NVMe-oF to replicate files across storage nodes creates two challenges. First, since the remote node's local file system (e.g., ext4) is not involved in writing the files, it is not aware of the updated file and its location, has no way to read it, and may even accidentally overwrite it. Second, the key-value application running on the remote node must also be synchronized with the incoming files. Its application-level in-memory data structures must be updated to find and read data from new files that were updated on its local storage device, and it must not read data from stale files that were deleted in the compaction process.

In this work, we introduce RubbleDB, the first distributed storage system that leverages offloaded NVMe-oF for efficient replication. The key contributions underlying RubbleDB's design are mechanisms that provide both *file system synchronization* and *application synchronization* at the remote node, so it can safely and correctly read data that was written to it via NVMe-oF.

In order to simplify file system synchronization, we make the observation that modern SSD-based datacenter storage systems [4, 8, 9, 16, 35] write data in large immutable (and often fixed-sized) chunks, and do not allow in-place updates. Therefore, RubbleDB pre-allocates all on-disk data on all nodes as fixed-sized fixed-location files. RubbleDB maintains a *file map* that stores the mappings between the file names and the pre-allocated file locations, and indicates whether a file contains live or stale data. When a new file is replicated, it is sent to a pre-allocated location that does not contain a live file. When a file is deleted in the compaction process, it is simply marked as stale in the map, and is not actually deleted.

For application-level synchronization, RubbleDB needs to keep the secondaries' in-memory data structures synchronized, so when they read data from disk, they read the most up-to-date object versions. To do so, RubbleDB ensures that changes made to the in-memory data structures in the secondary nodes will be consistent with the compactions executed by the primary node. It also carefully synchronizes the deletion of objects flushed from disk or memory, in order to avoid accidentally deleting objects that were processed out-of-order in the secondary nodes.

Our evaluation demonstrates that RubbleDB consistently leads to significant CPU and I/O bandwidth savings compared to a baseline, which represents the architecture of systems such as Meta's ZippyDB [17, 43] or CockroachDB [42], which run compaction on all nodes in a replication group. These savings enable RubbleDB to consistently achieve the same or higher throughput than the baseline across the entire YCSB suite [20], as well as on five traces from Twitter's key-value cache clusters [49]. In particular, RubbleDB provides a speedup up to 1.9× and a tail latency improvement of up to 93.4%. We also show that RubbleDB consistently provides higher performance in different scenarios, including different replication factors, different numbers of RocksDB instances per physical server, and different types of storage devices.

While in this paper we focus on the particular use case of a replicated key-value store, we believe our design ideas are applicable to other common storage applications with primary-backup replication, such as replicated file systems [16, 26, 34, 48] and disaster recovery and backup services [38].

2 Background and Motivation

This section lays out the background and motivation for the paper. §2.1 provides background information on the most common data structure for disk-based key value stores, the log structure merge tree (LSM tree), and demonstrates that background compaction operations in LSM trees consume significant CPU. §2.2 provides a primer on the NVMe-oF protocol and then shows the performance benefit of using NVMe-oF for storage replication with a microbenchmark.

2.1 The High Cost of Compactions

LSM trees. LSM trees [37] are a popular data structure for disk-based key-value stores, which powers many modern key-value stores, such as RocksDB [8], LevelDB [4] and WiredTiger [9]. Since small random writes significantly hurt SSD (and HDD) performance, the main design goal behind LSM trees is that data written to disk is always written in large contiguous chunks and is never updated in-place.

As a representative system for LSM trees, we provide a primer on how RocksDB, a popular key-value store works. In RocksDB, to avoid small random writes to disk, all incoming data writes are batched in memory, in a data structured called the MemTable. Each entry in the MemTable has a sequence number that enables key versioning. MemTables can be *active*, which means that they are mutable and can be updated with new incoming updates, and *immutable*, which means they are waiting to be flushed and cannot be updated further. Eventually, the immutable MemTables get flushed to disk and written using a format called sorted string table (SST) files, which are composed of sorted key-value pairs. SST files are composed of blocks, each of which can be a data block or a metadata block. The metadata blocks include index blocks whose entries point to the keys at the start of each data block.

SST files are organized hierarchically into levels (L0, L1, ..., LN), where the "upper levels" (e.g., L0 is "higher" than L1 in the hierarchy) store the more recently updated versions of each key-value pair. Data from the MemTable is flushed into L0, which stores files with overlapping key ranges, while the files in lower levels (L1,...,LN) have non-overlapping key ranges.

A key feature of LSM tree-backed stores is *background compaction*, which periodically scans multiple SST files from two adjacent levels, combines them into a single file, and

flushes the new file into the lower level. In this process, deleted and overwritten keys are discarded, freeing up space for new data. Compactions are necessary not only for freeing up space on disk, but also for reducing the number of I/Os required on average to read data from the LSM tree [37].

To reconstruct the LSM tree after a failure, RocksDB persists a log containing changes to the tree, e.g., deletion or generation of SST files. RocksDB records such changes using *version edits*, where a version represents the current set of SST files in the tree. For example, a version edit may record the removal of stale SST files and the generation of new merged files. Although compaction jobs run in parallel, they produce version edits in a serializable order because RocksDB protects the tree status with a mutex.

CPU consumption of compactions. Compactions are expensive and can affect the performance of the key-value store. A compaction job requires reading the data of all the files involved in the compaction (often involving tens of MB of data or more), sorting them, and writing them back to disk.

As an example, we measure the CPU time consumed by compactions by running a microbenchmark (described in §5.2) on a replicated 3-node key-value store, where each node conducts compaction locally, under a data ingestion microbenchmark (YCSB load [20]). In this workload, 72% of CPU time was dedicated solely for compaction jobs! Due to their high cost of compactions, there is a large body of work on reducing their resource consumption in single-node LSM trees [12, 13, 29, 32, 39], e.g., by delaying them, synchronizing them with incoming requests, or optimizing the LSM tree data structures and parameters to reduce their cost.

Saving compaction CPU and I/O bandwidth in replicated key-value stores. Our focus is orthogonal to these single-store optimizations: we make the observation that in settings where the same data is replicated on a set of R key-value stores, we do not have to run R identical compaction jobs across all nodes, which are essentially performing the same exact computation. Therefore, compaction can occur only once (on the *primary* node), and the already-compacted SST files can be shipped to the *secondary* nodes, which hold the backup copy of the data.

Such an approach has the potential to significantly reduce CPU consumption on the secondary nodes, since they no longer need to issue read and write I/O and sort the compacted data, the latter of which typically consumes the most CPU during compaction jobs [12, 13, 29]. In addition, this would eliminate the compaction read I/O of secondary nodes, since they would not need to read the files that need to be merged by the compaction job, but it would not eliminate the secondary's write I/O, since the new file would still have to be written back to the disk. Finally, it would also reduce the memory pressure on the secondary nodes due to compaction.

However, executing compactions only on primary nodes has a price. The primary cost of this approach is increased network bandwidth and NIC resource consumption, since now not only the "regular" incoming read/write requests need to be replicated, but also the post-compaction SST files. Fortunately, in many datacenters the network is often underutilized: for example, in traces from Alibaba [1] and Snowflake [47], 50–75% of the network capacity is idle. In addition, the primary node would consume some additional CPU in shipping the files to the secondary nodes' disks.

Therefore, since this approach involves a trade-off primarily between minimizing CPU consumption on the secondary nodes and increasing total network bandwidth, we seek to ship the SST files with a protocol that will minimize CPU usage on the secondary nodes. To this end, we turn to NVMe-oF, a state-of-the-art networked storage protocol supported by Linux and modern NICs, which can be run without the involvement of the secondary nodes' CPU.

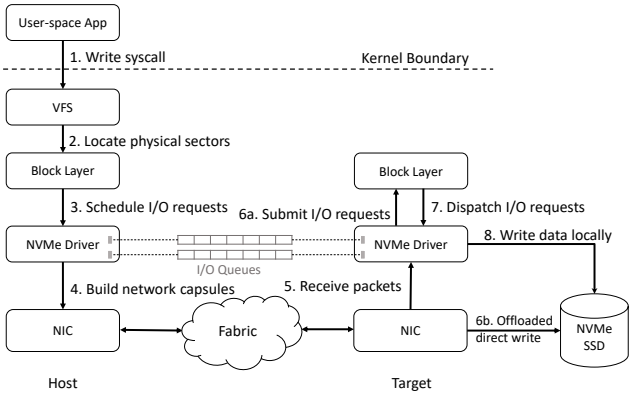


Figure 1: NVMe-oF overview.

2.2 Motivation for Using NVMe-oF

NVMe-oF primer. NVMe-oF is an extension of the NVMe protocol for networked storage. NVMe-oF allows an application to directly access a storage device that is connected to a remote server, using the NVMe protocol. Figure 1 depicts the flow of an NVMe-oF request. The *host* (left side of the diagram) is the server that initiates the request, and the *target* is the remote server and the SSD connected to it. The NVMe-oF request is initiated by an application on the host, which issues a system call, and subsequently traverses the entire OS storage stack, treating it as a regular local NVMe request, until it reaches the NVMe driver.

Take a write request as an example (Figure 1): the userspace application issues a WRITE() system call on the file located on an NVMe-oF mounted disk (step 1), then just like a normal local I/O, it goes through the Linux Virtual File System (VFS) to find the inode, which maps the physical sectors on the disk and is then submitted to the block layer (step 2) where it gets batched by the I/O scheduler, and is dispatched to the host-side NVMe driver (step 3).

The *host* and *target* drivers maintain multiple I/O queues for exchanging the NVMe-oF capsule, which is a data structure that contains essential information needed for an NVMe

	gRPC + WRITE()	NVMe/TCP	NVMe/RDMA
Throughput	1028 MB/s	2986 MB/s	3748 MB/s
CPU	155%	135%	50%

Table 1: Comparison of throughput and CPU consumption of 1 MB writes with different protocols. NVMe-oF (via TCP or RDMA) is much more efficient than replicating through userspace.

communication between the host and the target. The NVMe driver handles this request by constructing a corresponding NVMe-oF command within a capsule, mapping data and metadata from the memory, and submitting it to one I/O queue. The capsule is then forwarded to the relevant network stack (step 4) depending on the fabric type (TCP, RDMA, etc.) and is then forwarded to the target. For NVMe/TCP, the capsule is embedded in TCP packets and contains both data and metadata, while for NVMe/RDMA, the target and the host exchange the capsule using two-sided RDMA operations. With NVMe/RDMA, the capsule records the memory address of the data buffer in the host and the target consequently reads that portion of memory using a one-sided RDMA read.

On the target (step 5), after the driver extracts the NVMe-oF command and user data from the network packet, it generates the block layer request and submits it to the block layer for I/O scheduling (step 6a). The target's NVMe driver, at last, receives the I/O request from the block layer (step 7) and writes the user's data to the local NVMe SSD through the PCIe bus (step 8).

In the past few years, major NIC model lines (e.g., NVIDIA ConnectX, Broadcom Stingray, Intel IPU) have supported completely offloading the NVMe-oF target datapath to the NIC, and allowing the NIC to directly write the data to the NVMe device. This offers an alternate datapath that bypasses the target's CPU completely (step 6b). When the NIC attached to the target receives an NVMe capsule from the *host*, it executes the NVMe request and directly writes data on the NVMe SSD via DMA.

Potential benefit of NVMe-oF for replication. Popular distributed storage systems (e.g., CockroachDB [42] and Ceph [48]) often use an RPC (e.g., gRPC [3]) to send data from the primary to the secondary node, which in turn is written locally to the SSD (e.g., with a WRITE() system call).

We compare the throughput and CPU usage of this userspace-based baseline with two NVMe-oF protocols (NVMe/RDMA, which stands for NVMe-oF over RDMA and NVMe/TCP, which stands for NVMe-oF over TCP), in a microbenchmark that writes 1 MB data chunks over the network in a closed loop, with two servers using the same experimental setup on CloudLab [40] described in §5.1. In the experiment, each server contains one primary node that is writing to a secondary node on the second server, with a total of 256 available cores. The aggregate results are shown in Table 1. The result shows that the throughput of gRPC with WRITE() is only 34% of the throughput NVMe/TCP while the CPU usage is 20% higher. In addition to the more complex logic in the RPC framework, the userspace stack requires extra user-kernel boundary crossings and context switches when the data buffer is delivered to the userspace application from the TCP/IP stack in the kernel and then written to the local file which incurs a kernel trap. NVMe/TCP, on the other hand, processes the data write completely in the NVMe driver in the kernel, therefore saving a substantial amount of CPU cycles in each write request, thereby increasing the throughput. In addition, NVMe/RDMA outperforms NVMe/TCP due to the elimination of unnecessary copying and CPU bypassing.

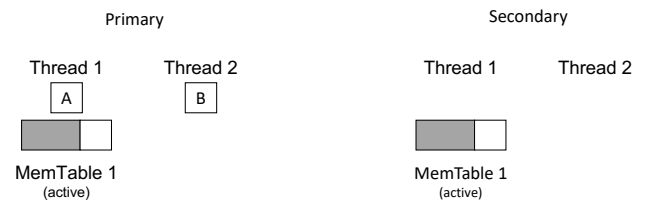
3 Challenges

Substituting a userspace replication protocol with NVMe-oF introduces challenges at two different layers: at the file system level and the application level.

File system inconsistency. NVMe-oF introduces inconsistency at the file system level. A naive way to ship files through NVMeoF is to simply allocate a new file on the remote disk and write to it. However, in such a scheme, the secondary node will not even see the new SST files in its file system. This is because the SST files are created in the primary's file system, and NVMe-oF only forwards NVMe commands, which get executed below the file system layer in the secondary node's storage stack (see Figure 1). So, the primary and secondary nodes may see different files systems on the same NVMe disk. Even worse, the data sent by the host could accidentally overwrite data in physical blocks at the secondary that it is not supposed to access, since the local file system of the target may have changed its file-to-block mapping.

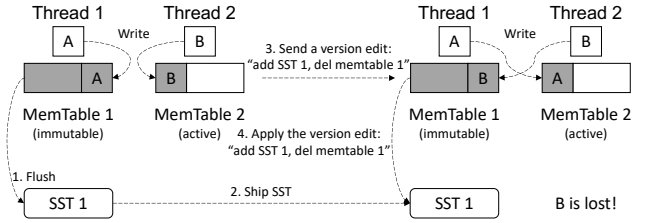
Application inconsistency. Even if the target's file system is synchronized with the host's view, NVMe-oF introduces inconsistency at the application level. Since the persistent key-value store maintains in-memory data structures (e.g., to buffer writes), these data structures may not be synchronized between the primary and the secondary, leading to data loss. In particular, in RocksDB, there will be discrepancies between the primary and secondary node within their MemTables, which store the values of recently-written data in memory.

Figure 2 shows an example where discrepancies in the primary and secondary's MemTables cause data loss in the secondary. Consider the case where there is one active MemTable (MemTable 1), which is nearly full and only has capacity for one more object (Figure 2a). Now consider that two objects (A and B) arrive concurrently. Both primary and secondary use two threads to process incoming requests, and in this case RocksDB does not provide any guarantee on the order that the writes will be processed. In the primary, object A is written before B, and is therefore written to MemTable 1, which is sealed and marked inactive, while object B is written to the newly active MemTable 2. Next, the primary forwards objects A and B to the secondary, but the secondary applies them in the opposite order due to non-deterministic thread scheduling: B is written to MemTable 1, and A is written to MemTable 2. Consequently, the secondary's MemTable 1 stores different



(a) Initial state, single active MemTable.

(b) The secondary executes concurrent requests at a different order.



(c) The secondary removes MemTable 1, losing object B.

Figure 2: An example of inconsistency across node MemTables.

data than the primary's MemTable 1 (Figure 2b).

Now, the primary flushes MemTable 1 to disk, causing it to delete the objects stored in MemTable 1 from memory. If it then ships the new SST file to the secondary, and instructs it to also delete it to delete MemTable 1 as well, this will result in the loss of B at the secondary, because B will not be stored neither in its MemTables, nor on its disk (Figure 2c). In this case, the reason for the data loss is due to the fact that thread scheduling across the nodes in a non-deterministic fashion, so operations are applied in a different order, causing discrepancies.

Making matters worse, even if we had a way to force secondary nodes to process requests in the same order as the primary, the content of the MemTables would still diverge. This is because RocksDB's MemTables store their data using randomized skip lists, which will cause MemTables in different nodes to contain a different number of entries and become full at different times.

4 Design and Implementation

We present the design and implementation of RubbleDB, and explain the key mechanisms that allow RubbleDB to address the inconsistencies introduced by replication via NVMe-oF.

RubbleDB is a replicated key-value store, composed of a set of RocksDB instances, with a replication layer on top. RubbleDB uses chain replication [45] to provide strong consistency and fast recovery. The client only communicates with the replicator layer, which is in charge of dispatching requests to the proper primary node (in case of write) or tail node (in case of a read) and of handling failure recovery. Figure 3

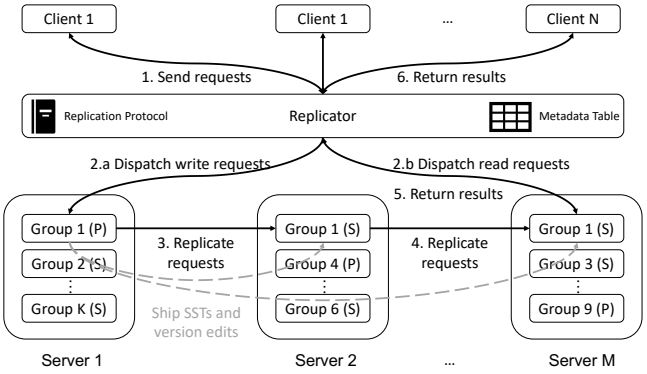


Figure 3: RubbleDB overview.

depicts the overall architecture of RubbleDB. There are N clients and K replication groups, and in between sits the replicator layer. Replication groups contain R RocksDB instances or *nodes*, one of which is the primary, and the others are secondaries. Only the primary performs flush or compaction jobs. Therefore, in addition to replicating client write requests, the primary node also ships compacted SST files via NVMe-oF, assuming sufficient network bandwidth is available. If the network becomes congested, RubbleDB can fall back to local compaction on all replicas. Specifically, RubbleDB compares the latencies of shipping SST files and local compaction. If the former is consistently greater over a time period, RubbleDB falls back to regular compaction. Different replication groups store disjoint key spaces. By default, the R replicas are stored on R different random servers. In the future, we plan to support other more sophisticated data placement policies [18, 19]. We intentionally keep each replication group small (by default 10 GB), so the recovery load can be spread across multiple nodes in the cluster when a server or disk fails. It is worth noting that we assume no dishonest or malicious node (e.g., we assume all nodes operate under a single organization in a single data center). Next, we discuss the design details of the two main key components of RubbleDB: the replicator layer and replication groups.

4.1 Replicator Layer

To provide a clean key-value interface from users and hide the complexity of dealing with the replication protocol, RubbleDB uses a replicator layer as a proxy layer between users and replication groups. Users simply send regular RocksDB requests to and receive results from the replicator layer, which transparently handles the replication protocol. The replicator thus has two roles: 1) routing requests to a replica of the group that contains the requested key-value pairs and 2) detecting and recovering from any failed replicas.

Different replication groups contain separate key spaces. To route requests, the replicator maintains a metadata table that records the key space and network addresses for each replica group. Once it receives a request, the replicator first looks up the group number in the metadata table. Next, according to the replication protocol, it forwards the request to a specific

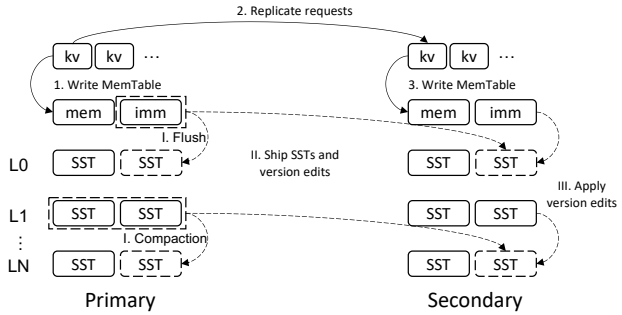


Figure 4: Replication process.

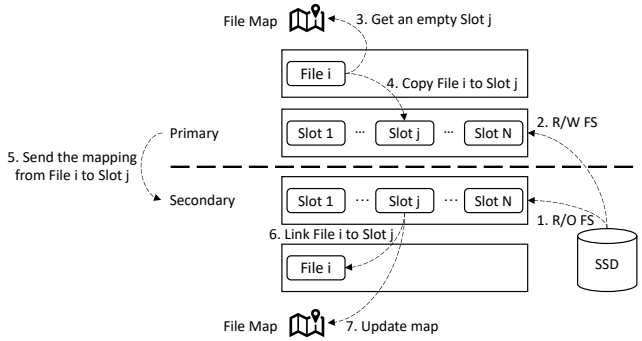


Figure 5: Primary ships SST files to pre-allocated slots.

replica within that group. The replicator also sends heart beat messages to every replica periodically to confirm its health. If it does not receive any replies from a replica after a time threshold, the replica is assumed to have failed, and the replicator starts the recovery process.

In Figure 3 foreground data flows are represented by a solid arrow, while background flows are dashed. The figure only shows the background requests belonging to replication group 1, which is replicated across servers 1, 2 and M. Clients first send requests to the replicator (step 1), who after consulting the metadata table forwards the requests to replication group 1 (step 2). Following the chain replication protocol [45], write requests (e.g., put and update) go to the head (step 2.a), while reads (e.g., get and scan) go to the tail (step 2.b). In the case of writes, the primary (head) replicates the write request to the next secondary in the chain (step 3), which applies the write and then replicates it to the next node in the chain (step 4). When the tail node completes a request (read or write), it will reply to the replicator (step 5), which finally returns the results to the client (step 6).

It is important to note that the replicator is only a logically centralized component that orchestrates traffic and recovery. To prevent the replicator from being a performance bottleneck or a single point of failure, it can be implemented as a distributed fault-tolerant cluster [17, 42]. We leave this direction, as well as other aspects of the replicator's design, such as dynamic load balancing and dynamic key-space partitioning, for future work.

4.2 Replication Groups

Each node within a replication group is a small RocksDB instance, composed of a primary node (head of the chain) and a chain of secondary nodes, which store the backup copies of the data. Figure 4 presents how a primary interacts with one of its secondary nodes. Solid and dashed arrows represent foreground and background operations, respectively. Write requests are executed from the head replica (the primary) to the tail (steps 1-3). Read requests are omitted in Figure 4 because they are only sent to the tail secondary node.

Steps I-III show how RubbleDB avoids background compaction jobs in secondaries. In step I, flush and compaction jobs happen normally in the primary (triggered by filled MemTables or upper-layer SST files). These jobs change the primary's LSM tree in three ways: 1) deletion of the data being compacted (both in-memory immutable MemTables and on-disk SSTs, depicted with dashed rectangles in Figure 4), 2) creation of compacted SSTs (dashed rounded rectangles), and 3) modification of the LSM tree version (information recording current SST files in the tree). RubbleDB ensures that the same changes also occur in the secondaries by shipping both compacted SSTs and version edits over the network (step II). Shipping the compacted SST file addresses 2), so the secondary only needs to delete the original obsolete SST files according to the version edits and update its own LSM tree version (step III).

However, it is not trivial to guarantee the correctness of steps II and III due to the challenges described in §3. In §4.2.1 we discuss how RubbleDB solves the challenge of file system inconsistency, while in §4.2.2 we describe how RubbleDB addresses application inconsistency.

4.2.1 File Pre-allocation

As the primary and secondary nodes mount their own local file systems (e.g., ext4) on top of the same storage device, each local file system will not be aware of changes made by the other file system, e.g., file creation. To ensure that shipped SST files are visible to secondary nodes, RubbleDB uses *file pre-allocation*. Before running, secondary nodes allocate many pre-allocated file slots, which we call a *file pool* on their local storage devices, after which, the primary mounts these devices. So both sides will be aware of the file pool in their local file systems. During runtime, the primary ships an SST file to a secondary by writing the content to a fixed-sized slot in the pool with direct I/O (to make sure the file gets written to disk and bypasses the primary's local buffer cache). Thus, only the data blocks of the slot file are updated and the inode remains unchanged. The secondary can also read the content with direct I/O after the file is written.

Note that this means that secondary and primary nodes cannot rely on the buffer cache to cache hot data blocks from disk. Fortunately, RocksDB (and most other key-value stores) implements its own userspace-based cache, the block cache, which can replace the operating system's buffer cache.

There are four practical issues with this pre-allocation

scheme: 1) determiming the size of slot files, 2) managing slot files in the pool, 3) avoiding dynamic file remapping by the local file systems, and 4) ensuring that RocksDB will correctly point to the pre-allocated files even when it changes file names. We discuss each issue below.

File size. To guarantee that the primary can find a slot to ship SST, secondaries need to allocate a sufficient number of file slots for every possible file size. Fortunately, key-value stores like RocksDB typically store data in more or less fixed-sized (or size-capped) files. Moreover, the number of SST files in each layer of an LSM tree is also limited by compaction. For example, by default in RocksDB, the size of an SST file is 64 MB and the maximum number of SST files is 4448. In this case, a secondary would need to create 4448 64 MB file slots. In our implementation we use a fixed-size file that is slightly larger (17 MB) than the target file size of RubbleDB's RocksDB instances (16 MB), since files may occasionally exceed the target size. When files are smaller than the fixed size of the slot, the remainder of the slot is zero-padded.

Slot management. The primary acquires slots in the pool before shipping SST files to secondaries. Similarly, when deleting an SST file post compaction, the corresponding slot is released. We design a *file map* to track the mapping between slots and SST files and to indicate whether a slot contains a live SST file. Both the primary and secondary nodes have a copy of the map. It is necessary for secondaries to own a map copy because once the primary fails, one of them will be chosen as the new primary.

In a flush or compaction job, the primary first acquires empty slots in its file map and then executes the compaction. After shipping the compacted files to the secondary nodes, it sends the map updates to all secondaries with the version edits, so the same updates are applied in all the secondary nodes. After receiving the updates, the secondary marks the slots of the old files, whose space can be overwritten, as released, and it updates the primary's file map to notify it about the slot release. The reason slots are released by secondary nodes is to avoid the case where the primary node releases a slot, and then acquires it again before the secondary node was notified of the slot release, which would be viewed by the secondary as an illegal operation, where a new file overwrites an already-acquired slot.

File remapping. The pre-allocated file slots' mappings from file offset to physical block address may change over time. Various reasons can cause remapping, including dynamic volume management, file system extent adjustment, etc.. To minimize interference from the file system and volume management, RubbleDB uses a dedicated and static disk partition for the file pool in each secondary node. The partition is mounted as read-only in the secondary, since the secondary never writes to its SSD drive, and read-write in the primary node. In case of a crash, where a secondary needs to become a primary, it remounts with read-write mode.

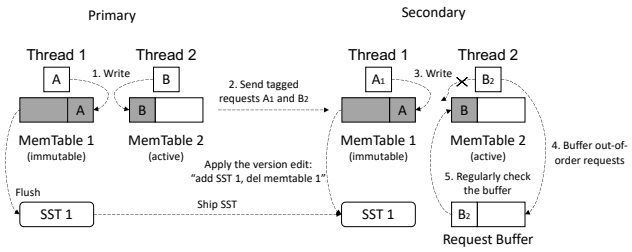


Figure 6: Partially-ordered writes using MemTable ID.

Renaming RocksDB names each SST file with a unique integer, e.g., `002023.sst`. This leads to another issue of pre-allocating slot files: their fixed file names. Because the secondary mounts the partition as read-only, it cannot rename the slot files to the RocksDB format. To address this issue, RubbleDB creates a symbolic link from a file with the RocksDB-defined name to the slot file, so the RocksDB instance on secondary can correctly access its read-only file pool.

Multiple groups. If there are multiple replication groups, primary nodes from different groups will acquire slots concurrently. To avoid contention, RubbleDB creates a dedicated SST pool (and map) for each group. Recall that since each pool sits on a different disk partition, there are no concurrent writers to a file in RubbleDB.

Figure 5 summarizes the file replication workflow. Before the replication group is formed, a disk partition is created for the file pool on each secondary. The primary and secondary mount the partition as read-only (step 1) and read-write (step 2), respectively. Suppose that the primary node generates SST file i in a flush job, it first queries the file map for an empty slot j to ship the SST file (step 3). Next, the content of SST file i is written in slot j with direct I/O (Step 4). The data of slot j will be transferred to the secondary node's SSD via NVMe-oF. At the end of the flush job, the primary sends the mapping between file i and slot j to the secondary (step 5), so the secondary knows how to create the correct symbolic link (step 6) and update its file map copy (step 7).

4.2.2 LSM Tree Synchronization

Flush and compaction jobs are essentially performing merge sort and do not change the actual state of RocksDB from the client's perspective[1]. These merge sorts contain inputs: MemTables and SST files to be merged in the case of flush and compaction, respectively, while the output is always SST files that will be written to disk. This property implies that the inputs and output of a flush or compaction job must contain the same set of live key-value pairs. Primary nodes naturally satisfy this requirement since they execute compaction locally. However, secondary nodes sometimes have mismatched sets of inputs and output live key-value pairs when applying version edits. Recall from the example in Figure 2, in the secondary node, the input to the flush job (MemTable 1) has

[1]Although stale data will be discarded during compaction jobs, it is already ignored by RocksDB since read requests fetch the most recent data.

different live objects from the output (SST 1). Consequently, the secondary node loses B while redundantly storing two copies of A.

To guarantee the data consistency of secondary nodes, they need to ensure that the inputs and outputs of every version edit contain the same set of live objects before applying it. However, comparing all objects across multiple MemTables or SST files is very costly. Instead, RubbleDB forces a partial order of requests and total order of version edits. These two ordering techniques synchronize the secondary nodes' LSM trees with the primary's. We describe them below.

Partially-ordered writes. Figure 6 describes how RubbleDB addresses the MemTable discrepancy issue discussed in the example in Figure 2 by ordering write requests with MemTable ID. In the primary, after an object is inserted to the active MemTable, each write request is returned with the ID of that MemTable (step 1). The primary tags each write request with this MemTable ID and forwards it to the secondary (step 2, the subscripts are the IDs). With the IDs, the secondary now knows to which MemTable the primary wrote each request. The secondary follows the same order as the primary, by maintaining a request buffer to cache out-of-order requests. For example, even if the secondary scheduled thread 2 before thread 1, it will fail to write B_2 to MemTable 1 as its tag (2) does not match with the MemTable ID (1) (step 3). So, thread 2 will store the request B_2 in the request buffer (step 4). When thread 1 executes, A_1 it will be written to MemTable 1. Therefore, MemTables 1 on both the primary and secondary nodes will have the same set of objects, which will not cause data loss like in Figure 2c. Last, every time RocksDB switches to a new MemTable, each thread in a secondary checks the buffer to execute any request that can be applied correctly to the MemTables, i.e. its tag is equal to the ID of the active MemTable (step 5).

This scheme represents a partial order because secondary nodes only sort write requests belonging to different MemTables. Write requests that have the same MemTable ID as the primary's MemTable have identical tags and can execute in any order. This does not affect the correctness when all updates in a MemTable have unique keys because MemTables (skip lists in RocksDB by default) and flush or compaction jobs (merge sorts) will sort them anyway. However, in the case where there are updates for the same key, as both MemTables and flush or compaction only select the most recent update, the secondary has to maintain the same order among those different updates. RubbleDB achieves such an order by further splitting the key space among threads. For example, all updates of key A will be handled by primary's thread 1 in Figure 6. Then, RubbleDB relies on in-order request delivery (e.g., streaming RPC) to ensure those updates arrives at a single thread of the secondary in the same order.

Totally-ordered version edits. Partially ordering writes only guarantees that the secondary nodes eventually have the same live objects in their MemTables as the primary node. However, due to request buffering, updates applied on the secondary nodes may lag the primary, so the same MemTable ID in a secondary node may have fewer entries than the one in the primary. Such lag introduces challenges when applying version edits in secondaries. Back to Figure 6, suppose that at time t, the version edit `(add SST 1, del MemTable 1)` arrives at the secondary but the request A_1 has not been executed. Applying the version edit at time t may allow the client to read A_1, even if it has not been written. This breaks the consistency guarantee of chain replication, which requires that a client can only read a value after it has received an acknowledgment that the value has been written successfully.

To avoid the scenario above, we have to ensure that the sets of live objects in the inputs and outputs of each version edit in the secondary are the same. We exploit the fact that in RocksDB flush or compaction jobs generate version edits in a serializable order (the current version is protected by a mutex) even though they run in parallel. So, the primary node tags version edits with sequence numbers to indicate their order, and the secondary nodes maintain a counter and a buffer for version edits. The counter is incremented every time the secondary applies a version edit. The secondary checks two conditions before applying an edit: 1) whether the sequence number is equal to the counter and 2) whether its inputs are ready. The latter is checked for flush jobs only, since the inputs of a compaction are always ready if it passes step 1) (i.e. the previous flush or compaction job has finished). A MemTable is ready only when it becomes immutable (full). If either of the two conditions fails, the version edit is cached in the buffer, which is regularly checked by all threads.

With these two ordering techniques, RubbleDB synchronizes the LSM tree state in a replication group and addresses the challenge of application inconsistency.

4.3 Implementation Details

We implement RubbleDB using RocksDB 6.14.0 and gRPC 1.34.0, comprising a total of about 900 and 4000 lines of Java and C++ code, respectively. Each replica in RubbleDB is a RocksDB instance, and different parts of the system communicate with each other using streaming gRPC calls. To simulate concurrent clients, we modify YCSB to issue requests as batches to our replicator in an open loop. We open-source all the code on GitHub[2].

5 Evaluation

We seek to answer four evaluation questions:

Q1: How does RubbleDB's SST file replication affect the CPU, network, and disk I/O usage of RubbleDB? (§5.2)

Q2: How does the replication mechanism of RubbleDB affect its performance under different workloads? (§5.3)

[2]https://github.com/lei-houjyu/RubbleDB

Workload	Composition
YCSB Load	100% inserts
YCSB A	50% Read, 50% Update
YCSB B	95% Read, 5% Update
YCSB C	100% Read
YCSB D	95% Read, 5% Insert
YCSB E	95% Scan, 5% Update
YCSB F	50% Scan, 50% Read Modify Write
YCSB G	100% Update
Twitter Cluster 2	100% Get
Twitter Cluster 15	100% Set
Twitter Cluster 19	75% Get, 25% Set
Twitter Cluster 27	85% Get, 15% Set
Twitter Cluster 31	6% Get, 94% Set

Table 2: Workload Characteristics

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	979	976	987	0
Requests	376	390	397	401
Total	2723		1786	

(a) Replication factor = 2

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	1319	2759	1375	0
Requests	535	990	570	1176
Total	5603		3121	

(b) Replication factor = 3

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	1713	5455	1846	0
Requests	692	1930	745	2254
Total	9790		4845	

(c) Replication factor = 4

Table 3: CPU time (s) breakdown under YCSB load, with a co-location factor of 1 and different replication factors.

Q3: Does the utility of NVMe-oF change as a function of the available storage resources? (§5.3)

Q4: How fast can RubbleDB recover from failures? (§5.4)

5.1 Experimental Setup

Setup. We conduct all experiments on CloudLab [24, 40]. Unless otherwise specified, replication groups run on multiple r6525 servers and clients run on one c6420 machine with the replicator. Each r6525 server has two 32-core AMD 7543 CPUs at 2.8 GHz, 256 GB DDR4 memory, a 1.6 TB Dell Enterprise SSD, and a dual-port Mellanox ConnectX-6 100 Gb NIC. By default, RubbleDB uses the Mellanox NIC's NVMe-oF offload feature. A c6420 server has two 16-core Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384 GB DDR4 Memory. The OS is Ubuntu 20.04 LTS with a Linux version of 5.4.0. We configure NVMe-oF target offloading following NVIDIA's official guide [7].

RocksDB configuration. We intentionally keep each key-value instance small, so that if an instance fails there will be a relatively small amount of data to re-replicate. Therefore, we use 16 MB SST files and MemTables and an L0 of size 64 MB, so the LSM tree will contain 64 GB data at most. Direct I/O is enabled with a 2 GB block cache. We run DB instances on each server within a cgroup with 4 physical cores. The number of background threads is therefore set to 4 (number of cores). All other parameters remain default.

	Baseline		RubbleDB	
	Read	Write	Read	Write
R = 2	163.7	185.6	94.6	206.6
R = 3	241.4	274.4	97.8	309.9
R = 4	343.3	387.5	101.7	410.7

Table 4: The read and write I/O (GB) on one node, with co-location factor of 1 and different replication factors.

	Baseline		RubbleDB	
	gRPC	NVMe-oF	gRPC	NVMe-oF
R = 2	34.5	0	34.5	105.7
R = 3	57.2	0	57.3	211.1
R = 4	80.0	0	80.1	314.7

Table 5: The total network traffic (GB) via gRPC and NVMe-oF on one node, with co-location factor of 1 and different replication factors.

Benchmark. We evaluate RubbleDB on all YCSB [20] workloads and five Twitter production traces [49]. Table 2 summarizes the workloads' read-write ratio. Four clients concurrently access all replication groups.

Baseline. For an apples-to-apples comparison, the baseline is a replicated RocksDB system, which is configured identically to RubbleDB, except that it does not replicate SST files, and does not include the various mechanisms RubbleDB uses to support NVMe-oF replication (e.g., buffering at the secondary nodes, processing version edits in-order). The baseline here would represent the standard approach of replicated key-value stores, such as ZippyDB [17,43] and CockroachDB [42], where each node compacts its data independently.

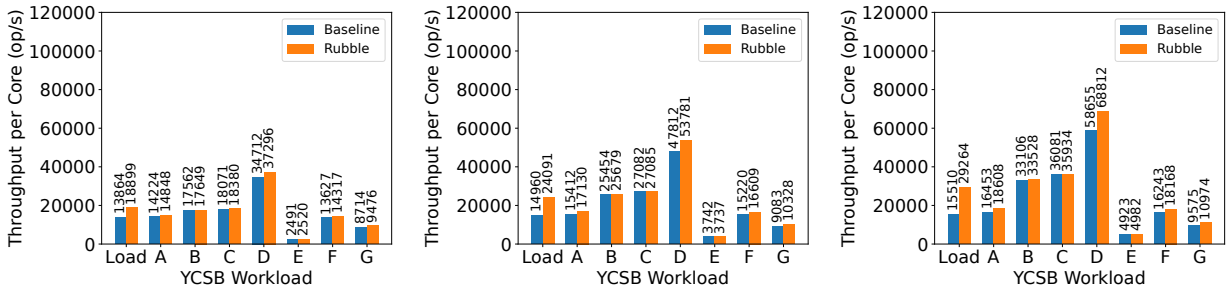
Evaluation metrics and terms. We use two primary evaluation metrics: throughput per core, which represents CPU efficiency, and tail latency. We use two knobs replication factor (R) and co-location factor (C) to indicate the numbers of replication groups (K), servers (M), and replicas in our experiments. We define $C = \frac{K}{M}$ and fix $M = R$, so, $K = C \times R$. For example, a replication factor of 3 and co-location factor of 2 means that on 3 servers ($M = 3$) exist 6 RocksDB instances ($K = 2 \times 3$) (2 primaries and 4 secondaries).

5.2 Performance Breakdown (Q1)

We run the YCSB load workload with a co-location factor of 1 and replication factors of 2, 3, and 4 in this section to collect CPU, disk, and network statistics.

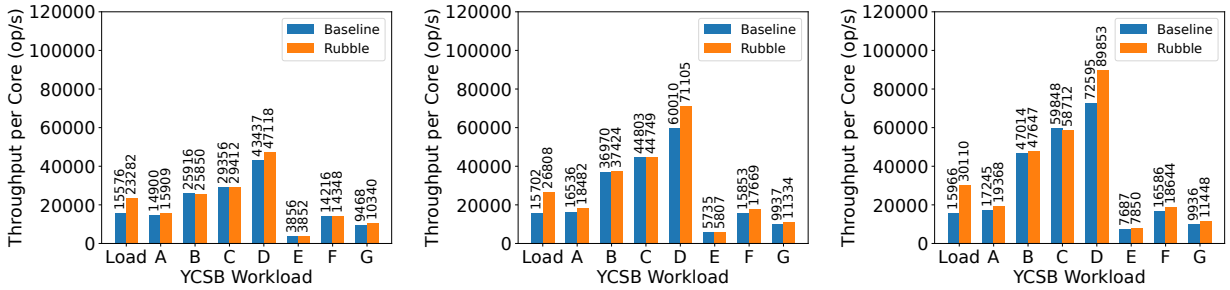
CPU savings. Table 3 presents the amount of CPU time the baseline and RubbleDB spend performing compaction and handling incoming requests. Handling requests includes both reading and writing data from RocksDB, as well as handling the incoming RPCs (i.e. via gRPC), buffering data on the secondary nodes, and applying version edits.

As expected, the secondary nodes on RubbleDB consume no CPU cycles executing compactions, while in the baseline system, each secondary node consumes roughly the same



(a) Replication Factor = 2 **(b)** Replication Factor = 3 **(c)** Replication Factor = 4

Figure 7: YCSB throughput as a function of replication factor with a co-location factor of 1.



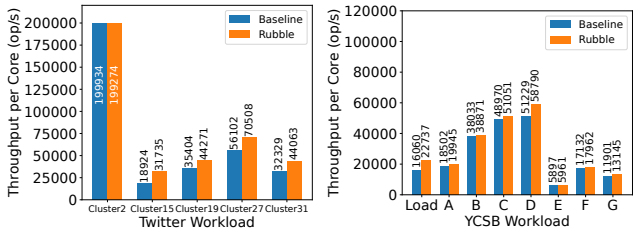
(a) Replication factor = 2. **(b)** Replication factor = 3. **(c)** Replication factor = 4.

Figure 8: YCSB throughput as a function of replication factor with a co-location factor of 2.

amount of CPU cycles as the primary (there are $R-1$ secondary nodes per primary). Under R=2, 3, and 4, the primary node of RubbleDB consumes 0.8%, 4.2%, and 7.8% more compaction CPU than the primary node of the baseline, respectively. This is because the primary has to send compacted SST files and version edits to each secondary node. The overhead increases with the number of secondary nodes.

In terms of handling regular requests, the primary node of RubbleDB consumes slightly more CPU (up to 7.7%) than the baseline's primary node, because it tags every write request with a MemTable ID. The secondary nodes of RubbleDB consume up to 18.8% more CPU than the baseline's, because of the need to buffer incoming requests and version edits. All in all, due to the reduction in the compaction load of the secondary nodes, RubbleDB spends 34.4%, 44.3%, and 50.5% less time processing the same workload than the baseline with R=2, 3, and 4, respectively.

I/O savings. Table 4 reports the amount of data read and written by one node. Since we run the YCSB load workload and disable the write-ahead log, the I/O is caused by compaction. In RubbleDB, only the primary performs compaction, which reads the inputs files and ships compacted SST files to every secondary. Therefore, RubbleDB's read I/O keeps nearly constant, 98.0 GB on average, while its write I/O grows with the replication factor proportionally, averagely $R \times 103.1$ GB. Both the read and write I/O in the baseline, however, increases with the replication factor because all nodes perform compaction. So, RubbleDB saves more read I/O with a higher replication factor, up to 44.2%



(a) **(b)**

Figure 9: Throughput on (a) Twitter cluster traces with a replication factor of 3 and co-location factor of 1, and (b) YCSB using Optane SSD with a replication factor of 2 and co-location factor of 2.

when $R = 4$. There is a modest increase (12.9% at most) in write I/O due to the padding of SST files in RubbleDB, which increases the amount of data that is written for each SST file. We leave reducing the overhead of padding to future work.

Network overhead. Table 5 presents both the gRPC and NVMe-oF traffic. The former consists of forwarding key-value requests and version edits, while the latter includes shipping SST files. The network overhead in RubbleDB includes: (a) sending version edits by gRPC and (b) shipping SST files via NVMe-oF. We approximate (b) by calculating the total volume of shipped SST files. From Table 5, (a) is negligible, and (b) is close to the compaction write I/O.

5.3 End-to-end Performance (Q2, Q3)

Throughput with YCSB. Figures 7 and 8 compare the throughput per core of RubbleDB with the baseline under the load and YCSB workloads, with a co-location factor of 1 and 2, respectively. RubbleDB consistently provides the

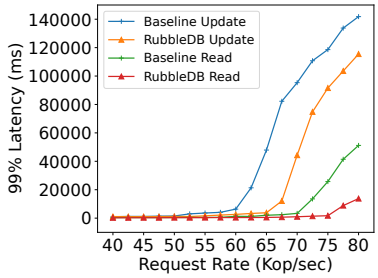


Figure 10: 99% Latency of YCSB A with replication factor of 3 and co-location factor of 1.

same or higher throughput per core compared to the baseline, and has a higher relative speedup for workloads with a high percentage of writes.

As the replication factor increases, RubbleDB provides higher relative gains. For example, under the load workload with a co-location factor of 2, a replication factor of 4 (Figure 7c) yields a speedup of 1.9×, while the speedup of $R = 2$ is 1.5×. The reason is that with higher replication factors, the baseline spends more secondary cores cycles per replication group executing compactions, while RubbleDB experiences a very marginal increase in the primary's CPU consumption (due to the need of shipping SST files to additional secondary nodes). Therefore, with a higher replication factor, RubbleDB has the ability to marshal more available CPU cycles belonging to the freed up secondary node cores, in order to process more incoming requests. In addition, RubbleDB achieves higher absolute throughput and speedup with a co-location factor of 2. The reason is that with more co-located replication groups, RubbleDB is better able to utilize the CPU, since there are more available pending tasks to execute at any given time.

Throughput with Twitter traces. We measure RubbleDB's throughput on five Twitter traces[3] with different read-write ratios, including cluster 2, 15, 19, 27, and 31 [49]. As Figure 9a shows, for write-heavy traces, RubbleDB provides a speedup of 1.7× and 1.4× in clusters 15 and 31, respectively. For cluster 19 and 27, which are ready-dominant, RubbleDB still achieves a 1.3× speedup. These results are largely consistent with the YCSB results.

Tail latency. RubbleDB provides better tail latency than the baseline when there are many compactions. Prior work has shown that compaction jobs interfere with request processing, leading to high tail latencies [12, 13]. Since RubbleDB significantly reduces the overall compaction load, as a result, it decreases the chance that compactions interfere with regular requests.

Figure 10 shows the 99th percentile latency under the YCSB A workload with 3 replicas and a co-location factor of 1, RubbleDB reduces 99th percentile latency of updates and reads by 11.5%-92.1% and 18.4%-93.4%, respectively.

[3]We sample 30GB records from the traces as we have 3 replication groups

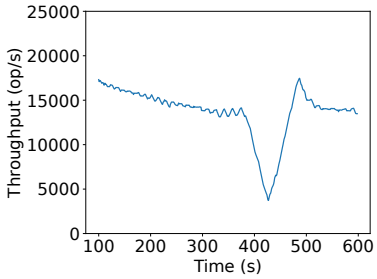


Figure 11: Throughput when a node fails under YCSB load.

The absolute latency is high because our system uses a batch size of 1,000 in evaluation, which means each replica returns a reply after processing all 1,000 requests in a batch. Since write requests go though all replicas sequentially, the update latency will be 3× higher than read latency. We also observe one data point (updates at 40 Kop/sec) showing 14.2% tail latency degradation. This is because out-of-order writes will be cached in the request buffer. Such queuing overhead only appears under light compaction pressure.

Object size. By default, YCSB uses 1 KB objects. When we run YCSB with smaller objects, which are typical in many datacenter settings [11, 17, 49] RubbleDB consistently provides even higher speedups, because a larger fraction of CPU time is spent on compacting data. For example, under YCSB load with a replication factor of 3 and co-location factor of 1, RubbleDB exhibits a 1.6× speedup with 100 B objects compared to a 1.5× speedup with 1 KB objects.

Different storage devices. We try to understand whether a different type of storage device affects RubbleDB's performance. To this end, we run RubbleDB and the baseline on two d750 servers from CloudLab, each of which use Intel Optane SSD P5800X, an SSD with single-digit µs average latencies.We run the experiment with a replication factor of 2 and co-location factor of 2. We are only able to run this experiment with two servers, because of the low availability of Optane SSD on CloudLab.

The results are presented in Figure 9b. Interestingly, the usage of low-latency storage does not materially affect RubbleDB's speedup. While the absolute throughput numbers for read-heavy workloads are higher (for an apples-to-apples comparison compare this experiment with Figure 8a), in the load workload the results are nearly identical. The reason is that while Optane SSD has much better latency than the enterprise SSD we use in the other experiments, its bandwidth is relatively similar, and in the case of LSM trees, write throughput will be determined by disk I/O bandwidth rather than I/O latency, since disk writes are sequential and large. We conclude that RubbleDB provides speedups on very different types of storage devices.

5.4 Recovery Performance (Q4)

To test RubbleDB's recovery from failure, we run a 3-node setup with a single replication group, and kill one of the tail

secondary nodes. We follow the recovery algorithm in [45], which designates the "middle" secondary node as the new tail. We plot the throughput over time in Figure 11. As the figure shows, due to the nature of chain replication RubbleDB is still able to service requests throughout the period when the node is down. In total, it takes about a minute and a half for the cluster to get back to its full throughput capacity.

6 Related Work

We split the related work into two categories: (a) replicated key-value stores, (b) systems that share data with different protocols, e.g., NVMe-oF and RDMA.

Replicated key-value stores. The typical design of replicated key-value stores and databases, such as ZippyDB [17,22, 43], CockroachDB [42], MongoDB [6] and Cassandra [31], is to implement a replication layer on top of multiple single-instance key-value stores, such as RocksDB [8], LevelDB [4] and WiredTiger [9]. In all these systems, all nodes that store up backup copies of data perform their own compactions, leading to high CPU and disk read I/O consumption.

There are several prior systems that do some form of compaction offloading. Ahmad et al. [10] propose offloading large compactions in HBase to a remote compaction server in order to reduce load on the primary nodes serving incoming requests. Hailstorm [14], separates the storage and compute layers, and offloads compaction to nodes that have a low load in a peer-to-peer fashion. Both of these systems allow shifting the computational load of compactions from an overloaded node to an underloaded one, but unlike RubbleDB do not reduce the total compaction load on the cluster by running compaction only once for replicated data.

Closer to RubbleDB, Tebis [46] is a replicated key-value store that reduces CPU consumption by avoiding compacting data multiple times for each replicated chunk of data. However, Tebis has several major design differences from RubbleDB and therefore faces different challenges. First, Tebis' design is based on a key-value architecture that separates keys from values [33]. Therefore, secondaries need to rewrite all the pointers in the indices. Due to the choice of key-value separation, Tebis cannot be applied to standard key-value stores that do not separate keys from values, such as RocksDB, LevelDB or WiredTiger. In addition, while key-value separation provides significant gains with large objects, it can degrade performance for small object workloads, which are common in datacenters [11, 17, 49]. Second, in Tebis, only the primary processes requests, whereas secondary nodes merely store replicated SST files. So, Tebis does not encounter the application inconsistency issue in RubbleDB. Third, instead of NVMe-oF, Tebis uses RDMA with local writes to ship SSTs, which cannot leverage the offloading feature of the NIC. Also, Tebis does not need to deal with inconsistencies caused by the file system.

Storage systems that use NVMe-oF. Several systems use NVMe-oF to access data from remote blocks [5, 15, 27], but only allow each application instance to exclusively access their SSDs. Therefore, these systems do not allow a primary node to replicate to a secondary node's disk directly over NVMe-oF. In other words, unlike RubbleDB, in order to replicate data, these systems require the primary to go through the entire application software stack of the secondary nodes.

Storage systems that use RDMA. Similar to NVMe-oF, the RDMA protocol allows one host to access the other hosts' memory without the CPU involvement of the target. There are a large number of in-memory systems that exploit RDMA for faster operations [23,28,35,44]. While both one-sided RDMA and NVMe-oF may introduce synchronization challenges at the target, the challenges are different, since NVMe-oF operates directly on block storage, potentially introducing corruptions to the local file system at the target.

Shared file systems. Shared file systems [26, 36, 48] provide users across different servers with a consistent view of a file system. However, providing a consistent file system abstraction across multiple nodes can come at a significant performance and scalability cost [30]. Since replicated key-value stores do not require a full synchronized file system interface across nodes, running them over a distributed file system would incur unnecessary overhead.

7 Conclusions

This work explores how to utilize NVMe-oF, a CPU-efficient networked storage protocol, for a common storage use case, replication. The main challenge in using NVMe-oF for replication is that data might need to be read by the target node in parallel to the replication process, introducing inconsistency both at the file system and application level. We demonstrate how such inconsistencies can be addressed in the context of a replicated LSM tree-based key-value storage system, RubbleDB, using two primary mechanisms: file pre-allocation and application data structure synchronization. We believe our ideas can be applied in other common storage settings, such as distributed file systems (e.g., HDFS [16], Ceph [48]) and for storage or application backup. In addition, with the trend of NIC accelerators becoming more powerful in contrast with the plateauing of CPU performance, we anticipate using NVMe-oF for common storage operations will become even more attractive in the future.

8 Acknowledgments

We thank our shepherd, Philippe Bonnet, and our anonymous reviewers for their valuable feedback and suggestions. We also thank Muli Ben-Yehuda, Jianan Luo and Michael J. Freedman for their helpful feedback during the project. This work was supported by NSF award CNS #2106530 and ARO award W911NF-21-1-0078 and was conducted on CloudLab.

References

[1] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[2] Apache Cassandra. http://cassandra.apache.org/.

[3] gRPC. https://grpc.io/.

[4] LevelDB. http://leveldb.org/.

[5] LightBits Labs. https://www.lightbitslabs.com/.

[6] MongoDB. https://www.mongodb.com/.

[7] NVIDIA's NVMe-oF Target Offload Configuration. https://enterprise-support.nvidia.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload.

[8] RocksDB. https://rocksdb.org/.

[9] WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[10] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment*, 8(8):850–861, 2015.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[12] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[13] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[14] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed LSM-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, and Yang-suk Kee. Crail-KV: A high-performance distributed key-value store leveraging native KV-SSDs over NVMe-oF. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[16] Dhruba Borthakur. HDFS architecture guide. *Apache Hadoop project*, 53(1-13):2, 2008.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[18] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 31–43, 2015.

[19] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, 2013.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review*, 41(6):205–220, October 2007.

[22] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage*, 17(4), oct 2021.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet,

Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[25] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, page 440–456, Virtual Event, Germany, 2021.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[27] Daegyu Han and Beomseok Nam. Improving access to HDFS using NVMeoF. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–2. IEEE, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[29] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018.

[30] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, New York, NY, USA, 2017. ACM.

[31] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[32] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced I/O performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 673–687, 2021.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016.

[34] Arif Merchant. Keynote address II, INFLOW 2014: Optimal flash partitioning for storage workloads in Google's Colossus file system. 2014.

[35] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41, 2011.

[36] Alex Osadzinski. The network file system (nfs). *Computer Standards & Interfaces*, 8(1):45–48, 1988.

[37] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[38] W Curtis Preston. *Backup and Recovery: Inexpensive Backup Solutions for Open Systems*. O'Reilly Media, Inc., 2007.

[39] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *login Usenix Mag.*, 39(6), 2014.

[41] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[42] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Amy Tai, Andrew Kryczka, Shobhit O Kanaujia, Kyle Jamieson, Michael J Freedman, and Asaf Cidon. Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 977–992, 2019.

[44] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them

remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48, 2020.

[45] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, volume 4, 2004.

[46] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: index shipping for efficient replication in LSM key-value stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 85–98, 2022.

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[48] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[49] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.