RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications

SIYING DONG, ANDREW KRYCZKA, and YANQIN JIN, Facebook Inc., 1 Hacker Way, USA
MICHAEL STUMM, Department of Electrical and Computer Engineering, University of Toronto, Canada

This article is an eight-year retrospective on development priorities for RocksDB, a key-value store developed at Facebook that targets large-scale distributed systems and that is optimized for Solid State Drives (SSDs). We describe how the priorities evolved over time as a result of hardware trends and extensive experiences running RocksDB at scale in production at a number of organizations: from optimizing write amplification, to space amplification, to CPU utilization. We describe lessons from running large-scale applications, including that resource allocation needs to be managed across different RocksDB instances, that data formats need to remain backward- and forward-compatible to allow incremental software rollouts, and that appropriate support for database replication and backups are needed. Lessons from failure handling taught us that data corruption errors needed to be detected earlier and that data integrity protection mechanisms are needed at every layer of the system. We describe improvements to the key-value interface. We describe a number of efforts that in retrospect proved to be misguided. Finally, we describe a number of open problems that could benefit from future research.

CCS Concepts: • **Information systems** → **Information storage systems**; **Flash memory**; *Open source software*; **Hierarchical storage management**; **Key-value stores**; **Directory structures;**

Additional Key Words and Phrases: Key-value stores, large-scale applications, RocksDB, SSD, compaction, databases

ACM Reference format:
Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (October 2021), 32 pages.
https://doi.org/10.1145/3483840

1 INTRODUCTION

RocksDB [19, 94] is a high-performance, persistent key-value storage engine created in 2012 by Facebook, based on Google's LevelDB code base [39]. It is optimized for the specific characteristics of **Solid State Drives (SSDs)**, targets large-scale (distributed) applications, and is designed as a library component that is embedded in higher-level applications. As such, each RocksDB instance manages data on storage devices of just a single server node; it does not handle any inter-host operations, such as replication and load balancing, and it does not perform high-level operations, such

Authors' addresses: S. Dong, A. Kryczka, and Y. Jin, Facebook Inc., 1 Hacker Way, Menlo Park, California, USA, 94025; emails: {siying.d, andrewkr, yanqin}@fb.com; M. Stumm, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4; email: stumm@eecg.toronto.edu.

1553-3077/2021/10-ART26 $15.00
https://doi.org/10.1145/3483840

Table 1. Some RocksDB Use Cases and Their Workload Characteristics

	Read/Write	Read Types	Special Characteristics
Databases	Mixed	Get + Iterator	Transactions; backups
Stream Processing	Write-heavy	Get or Iterator	Time windows; checkpoints
Logging/Queues	Write-heavy	Iterator	Uses both SDD and HDD media
Index Services	Read-heavy	Iterator	Bulk loading
Cache	Write-heavy	Get	Dropping data acceptable

as checkpoints—it leaves the implementation of these operations to the application, but provides appropriate support so they can do it effectively.

RocksDB and its various components are highly customizable, allowing the storage engine to be tailored to a wide spectrum of requirements and workloads; customizations can include the **write-ahead log (WAL)** treatment, the compression strategy, and the compaction strategy (a process that removes dead data and optimizes LSM-trees, as described in Section 2). RocksDB may be tuned for high write throughput, high read throughput, space efficiency, or something in between. Due to its configurability, RocksDB is used by many applications, representing a wide range of use cases. At Facebook alone, RocksDB is used by over 30 different applications, in aggregate storing many hundreds of petabytes of production data. Besides being used as a storage engine for ***databases*** (e.g., MySQL [61], Rocksandra [44], CockroachDB [91], MongoDB [76], and TiDB [46]), RocksDB is also used for the following types of services with highly disparate characteristics, summarized in Table 1:

- ***Stream processing***: RocksDB is used to store staging data in Apache Flink [10], Kafka Stream [51], Samza [69], and Facebook's Stylus [13]. These systems use RocksDB in unique ways. For example, some require the ability to checkpoint the state of RocksDB when checkpointing the stream processing system. Others require time window functions so they can lay out the data in a way that allows them to query and reclaim time windows in an efficient way.
- ***Logging/queuing services***: RocksDB is used by Uber's Cherami [68], Iron.io [49], and Facebook's LogDevice (which uses both SSDs and HDDs) [59]. These services demand high write throughput and low write amplification. By taking advantage of RocksDB's customizable compaction scheme, the services are able to write almost as efficiently as appending to a single file while still benefiting from features such as indexing.
- ***Index services***: RocksDB is used by Rockset [87], Airbnb's personalized search [98], and Facebook's Dragon [86]. These services demand good read performance and the ability to load offline-generated data into index services at large scale, for which RocksDB's bulk loading feature was initially created.
- ***Caching on SSD***: Several in-memory caching services use RocksDB to store data evicted from DRAM onto SSDs; these include Netflix's EVCache [57], Qihoo's Pika [80], and Redis [74]. These services tend to have high write rates and mostly issue point lookups. Some implement admission control, so not all data evicted from DRAM cache is written to RocksDB.

There are also other types of use cases, such as in Ceph's BlueStore, which uses RocksDB to store metadata, and the WAL [2] and DNANexus' DNA sequencing [55]. A prior paper presented an analysis of several applications using RocksDB [9]. Table 2 summarizes some of the system metrics obtained from production workloads.

Table 2. System Metrics for a Typical Use Case from Each of the Application Categories

	CPU Utilization	Space Utilization	Flash Endurance	Read Bandwidth
Stream Processing	11%	48%	16%	1.6%
Logging/Queues	46%	45%	7%	1.0%
Index Services	47%	61%	5%	10.0%
Cache	3%	78%	74%	3.5%

Having a storage engine that can support many different use cases offers the advantage that the same storage engine can be used across different applications. Indeed, having each application build its own storage subsystem is problematic, as doing so is challenging. Even simple applications need to protect against media corruption using checksums, guarantee data consistency after crashes, issue the right system calls in the correct order to guarantee durability of writes, and handle errors returned from the file system in a correct manner. A well-established common storage engine can deliver sophistication in all those domains.

Additional benefits from having a common storage engine are achieved when the client applications run within a common infrastructure: the monitoring framework, performance profiling facilities, and debugging tools can all be shared. For example, different application owners within a company can take advantage of the same internal framework that reports statistics to the same dashboard, monitor the system using the same tools, and manage RocksDB using the same embedded admin service. This consolidation not only allows expertise to be easily reused among different teams, but also allows information to be aggregated to common portals and encourages developing tools to manage them.

Given the diverse set of applications that have adopted RocksDB, it is natural for the development priorities of RocksDB to have evolved over time. This article describes how our priorities evolved over the past eight years as we learned practical lessons from real-world applications (both within Facebook and other organizations) and observed changes in hardware trends, causing us to revisit some of our early assumptions. We also describe our RocksDB development priorities for the near future.

Section 2 provides background on SSDs and **Log-Structured Merge (LSM)** trees [72]. From the beginning, RocksDB chose the LSM-tree as its primary data structure to address the asymmetry in read/write performance and the limited endurance of flash-based SSDs. We believe LSM-trees have served RocksDB well and argue they will remain a good fit even with upcoming hardware trends (Section 3). The LSM-tree data structure is one of the reasons RocksDB can accommodate different types applications with disparate requirements. But Section 3 describes how our primary optimization target shifted from minimizing write amplification to minimizing space amplification, and from optimizing performance to optimizing efficiency.

Sections 4–8 describe some of our experiences and lessons learned over the years. For example, lessons learned from serving large-scale distributed systems (Section 4) include (*i*) resource allocation must be managed across multiple RocksDB instances, since a single server may host multiple instances; (*ii*) the data format used must be backward- and forward-compatible, since RocksDB software updates are deployed/rolled-back incrementally; and (*iii*) proper support for database replication and backups is important. Lessons learned from dealing with failures (Section 5) include (*i*) data corruption needs to be detected early to minimize data unavailability and loss; (*ii*) integrity protection must cover the entire system to prevent silent corruptions from propagating to replicas and clients; and (*iii*) errors need to be treated in a differentiated manner. Lessons related to configuration management (Section 6) indicate that having RocksDB be highly configurable has enabled many different types of applications and that suitable configurations can

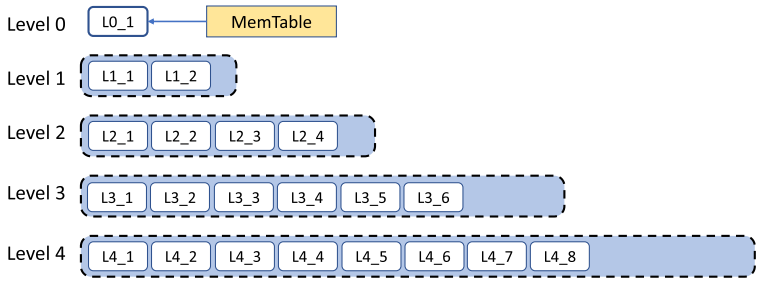

Fig. 1. RocksDB LSM-tree using leveled compaction. Each white box is an SSTable.

have a large positive performance impact, but also that configuration management is perhaps too challenging and needs to be simplified and automated. Lessons on the RocksDB API (Section 7) indicate that the core interface is simple and powerful given its flexibility, but limits the performance for some important use cases; we present our thoughts on improving the interface by supporting application-defined timestamps and columns. Finally, we present a number of development initiatives that in retrospect turned out to be misguided (Section 8).

Section 10 lists several areas where RocksDB would benefit from future research. We close with concluding remarks in Section 11.

2 BACKGROUND

The characteristics of flash-based SSDs have profoundly impacted the design of RocksDB. The asymmetry in read/write performance and limited endurance pose challenges and opportunities in the design of data structures and system architectures. As such, RocksDB employs flash-friendly data structures and optimizes for modern hardware.

2.1 Embedded Storage on Flash-based SSDs

Over the past decade, we have witnessed the proliferation of SSDs for serving online data. The low latency and high throughput device not only challenged software to take advantage of its full capabilities, but also transformed how many stateful services are implemented. An SSD offers hundreds of thousands of **Input/Output Operations per Second (IOPS)** for both of read and write, which is thousands of times faster than a spinning **hard drive (HDD)**. It can also support hundreds of MBs of bandwidth. Yet high write bandwidth cannot be sustained due to a limited number of program/erase cycles. These factors provide an opportunity to rethink the storage engine's data structures to optimize for this hardware.

The high performance of the SSD, in many cases, also shifted the performance bottleneck from device I/O to the network for both latency and throughput. It became more attractive for applications to design their architecture to store data on local SSDs rather than use a remote data storage service. This increased the demand for a key-value store engine that can be embedded in applications.

RocksDB was created to address these requirements. We wanted to create a flexible key-value store to serve a wide range of applications using local SSD drives while optimizing for the characteristics of SSDs. LSM-trees played an important role in achieving these goals.

2.2 RocksDB Architecture and Its Use of LSM-trees

RocksDB uses LSM trees [72] as its primary data structure to store data with the following key operations:

Writes. Whenever data is written to RocksDB, the written data is added to an in-memory write buffer called *MemTable*, as well as an on-disk **Write Ahead Log (*WAL*)**. MemTable is implemented as a skiplist to keep the data ordered with $O(\log n)$ insert and search overheads. The WAL is used for recovery after a failure, but is not mandatory. Once the size of the MemTable reaches a configured size, then (*i*) the MemTable and WAL become immutable, (*ii*) a new MemTable and WAL are allocated for subsequent writes, (*iii*) the contents of the MemTable are *flushed* to a **Sorted String Table (*SSTable*)** data file on disk, and (*iv*) the flushed MemTable and associated WAL are discarded. Each SSTable stores data in sorted order, divided into uniformly sized *blocks*. Once written, each SSTable is immutable. Every SSTable also has an index block with one index entry per SSTable block for binary search.

Compaction. The LSM-tree has multiple levels, as shown in Figure 1. The newest SSTables are created by MemTable flushes, as described above, and are placed in Level-0. The other levels are created by a process called *compaction*. The maximum size of each level is limited by configuration parameters. When level-L's size target is exceeded, some SSTables in level-L are selected and merged with the overlapping SSTables in level-(L+1) to create a new SSTable in level-(L+1). In doing so, deleted and overwritten data is removed, and the new SSTable is optimized for read performance and space efficiency. This process gradually migrates written data from Level-0 to the last level. Compaction I/O is efficient, as it can be parallelized and only involves bulk reads and writes of entire files. (To avoid confusion with the terms "higher" and "lower," given that levels with a higher number are generally located lower in images depicting multiple levels, we will refer to levels with a higher number as *older* levels.)

MemTables and level-0 SSTables have overlapping key ranges, since they contain keys anywhere in the keyspace. Each older level, i.e., a level-1 or older level, consists of SSTables covering non-overlapping partitions of the keyspace. To save disk space, the blocks of SSTables in older levels may optionally be compressed.

Reads. In the read path, a key lookup occurs by first searching all MemTables, followed by searching all Level-0 SSTables, followed by the SSTables in successively older levels whose partition covers the lookup key. Binary search is used in each case. The search continues until the key is found, or it is determined that the key is not present in the oldest level.[1] Hot SSTable blocks are cached in a memory-based block cache to reduce I/O as well as decompression overheads. Bloom filters are used to eliminate most unnecessary searches within SSTables.

RocksDB supports multiple different types of compaction [23]. *Leveled Compaction* was adapted from LevelDB and then improved [19]. In this compaction style, levels are assigned exponentially increasing size targets as exemplified by the dashed boxes in Figure 1. Compactions are initiated proactively to ensure the target sizes are not exceeded. *Tiered Compaction* (called *Universal Compaction* in RocksDB [26]) is similar to what is used by Apache Cassandra or HBase [36, 37, 58]. Multiple SSTables are lazily compacted together, either when the sum of the number of level-0 files and the number of non-zero levels exceeds a configurable threshold or when the ratio between total DB size over the size of the largest level exceeds a threshold. In effect, compactions are delayed until either read performance or space efficiency degenerates, so more data can be compacted altogether. Finally, *FIFO Compaction* simply discards old SSTables once the DB hits a size limit and only performs lightweight compactions. It targets in-memory caching applications.

Being able to configure the type of compaction allows RocksDB to serve a wide range of use cases. By using different compaction styles, RocksDB can be configured as read-friendly, write-friendly, or very write-friendly (for special cache workloads). However, application owners

[1]Scans require that all levels be searched.

Table 3. Write Amplification, Overhead, and Read I/O for Three Major Compaction Types under RocksDB 5.9

Compaction	Write Amplification	Max Space Overhead	Avg Space Overhead	#I/O per Get() with bloom filter	# I/O per Get() without filter	# I/O per iterator seek
Leveled	16.07	9.8%	9.5%	0.99	1.7	1.84
Tiered	4.8	94.4%	45.5%	1.03	3.39	4.80
FIFO	2.14	N/A	N/A	1.16	528	967

Results of a random write micro-benchmark with 16-byte keys and 100 byte values (on average 50 bytes after compression) with a total of 500 million keys. After the data is fully populated, random keys are overwritten with the ingestion rate limited to 2 MB/s and results are collected in this phase. The number of sorted runs is set to 12 for Tiered Compaction, and 20 bloom filter bits per key are used for FIFO Compaction. Direct I/O is used and the block cache size is set to be 10% of the fully compacted DB size. Write amplification is calculated as total SSTable file writes vs. the number of MemTable bytes flushed. WAL writes are not included. The number of I/Os per Get() without a filter is particularly high for FIFO compaction, because without the bloom filter, each SSTable needs to be searched until the result is found.

will need to consider tradeoffs among the different metrics for their specific use case[4]. A lazier compaction algorithm improves write amplification and write throughput, but read performance suffers. In contrast, a more aggressive compaction sacrifices write amplification but allows for faster reads. Services such as logging or stream processing can use a write-heavy setup, while database services need a balanced approach. Table 3 depicts this flexibility by way of micro-benchmark results.

In 2014, we added a feature called *column family*[2] [22], which allows different independent key spaces to co-exist in one DB. Each KV pair is associated with exactly one column family (by default the *default* column family), while different column families can contain KV pairs with the same key. Each column family has its own set of MemTables and SSTables, but they share the WAL. Benefits of column families include the following:

(1) each column family can be configured independently; that is, they each can have different compaction, compression, merge operators (Section 6.2), and compaction filters (Section 6.2);
(2) the shared WAL enables atomic writes to different column families[3]; and
(3) existing column families can be removed, and new column families can be created, dynamically and efficiently.

Column families are widely used. One way they are used is to allow different compaction strategies for different classes of data in the same database; e.g., in a database, some data ranges might be write-heavy and other ranges might be read-heavy, in which case compaction can be made more effective overall by placing the two different classes of data into two different column families configured to use different compaction strategies. Another way column families are used is to exploit the fact that a column family can be removed efficiently: If the data known to become obsolete[4] within a time period is placed in the same column family, then a column family can be removed at the appropriate time without having to explicitly delete the KV pairs contained therein.

[2] The name of this feature, "column family," was poorly chosen and not to be confused with "columns" in database tables (discussed in Section 7.2). While the initial motivation was to provide support for implementing columnar databases, it was never used for that purpose. A key lesson we learned was that the naming of features is important and should be done carefully, as it is hard to change a name once it is in use. Unfortunately, the name "column family" started to cause confusion even before the feature was completed.
[3] Atomic writes can either be part of a transaction or part of a write batch.
[4] For example, data with time-to-live (TTL) parameters.

Table 4. Micro-benchmark Measurement of RocksDB Space Efficiency

	Dynamic Leveled Compaction			LevelDB-style Compaction		
# keys	Compacted size	Steady DB size	Extra space overhead	Compacted size	Steady DB size	Extra space overhead
200,000,000	12.01 GB	13.50 GB	12.4%	12.01 GB	15.09 GB	25.6%
400,000,000	24.03 GB	26.86 GB	11.8%	24.03 GB	26.95 GB	12.2%
600,000,000	36.04 GB	40.45 GB	12.2%	36.36 GB	42.50 GB	16.9%
800,000,000	48.05 GB	54.16 GB	12.7%	48.33 GB	57.86 GB	19.7%
1,000,000,000	60.06 GB	67.52 GB	12.4%	60.28 GB	73.77 GB	22.4%

Data is pre-populated and each write is to a key chosen randomly from the pre-populated key space. RocksDB 5.9 with all default options. Constant 2 MB/s write rate.

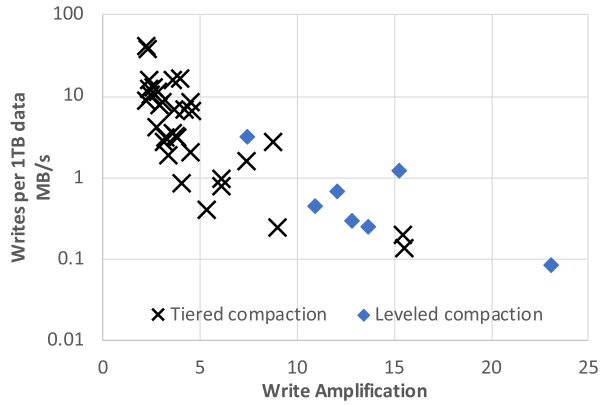

Fig. 2. Survey of write amplification and write rates across 42 randomly sampled ZippyDB and MyRocks applications. Note the logarithmic scale of the y-axis.

3 EVOLUTION OF RESOURCE OPTIMIZATION TARGETS

In this section, we describe how our resource optimization target evolved over time: from write amplification to space amplification to CPU utilization.

3.1 Write Amplification

When we started developing RocksDB, we initially focused on saving flash erase cycles and thus write amplification, following the general view of the community at the time (e.g., Reference [54]). This was rightly an important target for many applications, in particular for those with write-heavy workloads where it continues to be an issue; see Table 1.

Write amplification emerges at two levels. SSDs themselves introduce write amplification: by our observations between 1.1 and 3. Storage and database software also generate write amplification; this can sometimes be as high as 100 (e.g., when an entire 4KB/8KB/16KB page is written out for changes of less than 100 bytes).

Leveled Compaction in RocksDB usually exhibits write amplification between 10 and 30, which is several times better than when using B-trees in many cases. For example, when running LinkBench on MySQL, RocksDB issues only 5% as many writes per transaction as InnoDB, a B-tree based storage engine [61]. Still, write amplification in the 10–30 range is too high for write-heavy applications. For this reason, we added Tiered Compaction, which brings write amplification down

to the 4–10 range although with lower read performance; see Table 3. Figure 2 depicts RocksDB's write amplification under different data ingestion rates as measured on applications running in production. RocksDB application owners often pick a compaction method to reduce write amplification when the write rate is high and compact more aggressively when the write rate is low to achieve space efficiency and better read performance.

3.2 Space Amplification

After several years of development, we observed that for most applications, space utilization was far more important than write amplification, given that neither flash write cycles nor write overhead were usually constraining. In fact, the number of IOPS utilized in practice was low compared to what the SSD could provide (yet still high enough to make HDDs unattractive, even when ignoring maintenance overhead). As a result, we shifted our resource optimization target to disk space.

Fortunately, LSM-trees also work well when optimizing for disk space due to their non-fragmented data layout. However, we saw an opportunity to improve Leveled Compaction by reducing the amount of dead data (i.e., deleted and overwritten data) in the LSM-tree. We developed *Dynamic Leveled Compaction*, where the size of each level in the tree is automatically adjusted based on the size of the oldest (last) level, instead of setting the size of each level statically [19]. The size of the newer levels relative to the size of the oldest level can be a good indication of the magnitude of dead data in the LSM-tree. Hence, capping the ratio between the sizes of the newer levels and the oldest level tends to limit space overhead, something LevelDB-style leveled compaction fails to accomplish. For this reason, Dynamic Leveled Compaction achieves better overall and more stable space efficiency than Leveled Compaction. With the improved approach, we managed to reduce the space footprint of UDB, one of Facebook's databases, to 50% when we replaced InnoDB, a B+Tree based storage engine [60], with RocksDB. Table 4 shows space efficiency measured in a random write benchmark: Dynamic Leveled Compaction limits space overhead to 13%, while Leveled Compaction can add more than 25%. In the worst case, space overhead under Leveled Compaction can be as high as 90%, while it is stable for dynamic leveling.

3.3 CPU Utilization

An issue of concern sometimes raised is that SSDs have become so fast that software is no longer able to take advantage of their full potential. That is, with SSDs, the bottleneck has shifted from the storage device to the CPU, so fundamental improvements to the software are necessary. We do not share this concern based on our experience, and we do not expect it to become an issue with future NAND flash-based SSDs. This is for two reasons. First, only a few applications are limited by the IOPS provided by the SSDs; as we discuss below, most applications are limited by space.

Second, we find that any server with a high-end CPU has more than enough compute power to saturate one high-end SSD. RocksDB has never had an issue making full use of SSD performance in our environment. Of course, it is possible to configure a system that results in the CPU becoming a bottleneck; e.g., a system with one CPU and multiple SSDs. However, effective systems are typically those configured to be well-balanced, which today's technology allows. We also note that intensive write-dominated workloads may indeed cause the CPU to become a bottleneck. However, for some, this can be mitigated by configuring RocksDB to use a more lightweight compression option. For the other cases, the workload may simply not be suitable for SSDs, since it would exceed the typical flash endurance budget that allows the SSD to last 2–5 years.

To confirm our view, we surveyed 42 different deployments of ZippyDB [92] and MyRocks [61] in production, each serving a different application. Figure 3 shows the result. Most of the workloads are space-constrained. Some are indeed CPU-heavy, but hosts are generally configured to not be

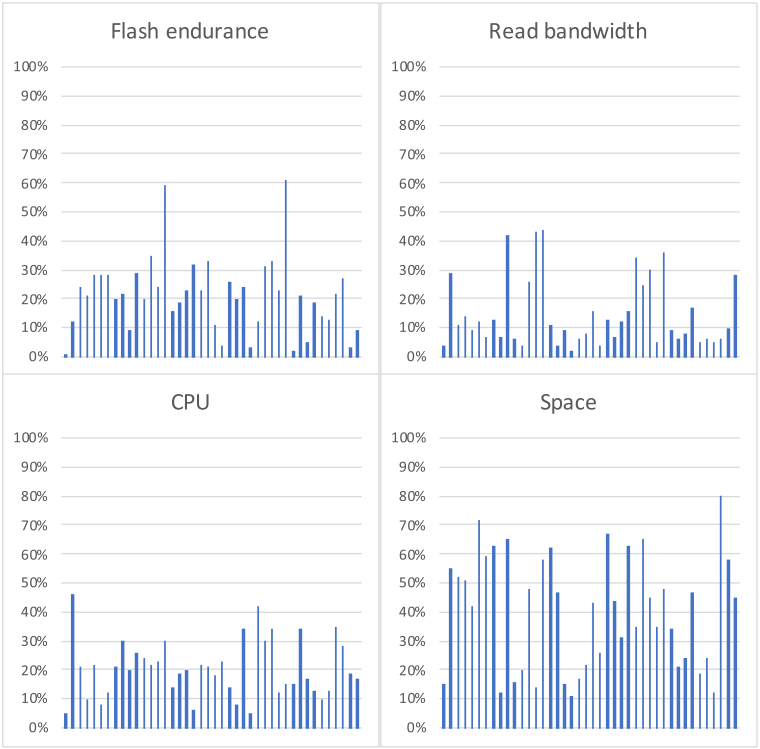

Fig. 3. Resource utilization across four metrics. Each line represents a different deployment with a different workload. Measurements were taken over the course of one month. All numbers are the average across all hosts in the deployment. CPU and read bandwidth are for the highest hour during the month. Flash endurance and space utilization are average across the entire month.

fully utilized to leave headroom for growth and for handling data center or region-level failures. Most of these deployments include hundreds of hosts, so averages give an idea of the resource needs for these use cases, considering that workloads can be freely (re-)balanced among those hosts (Section 4).

Nevertheless, reducing CPU overheads has become an important optimization target, given that the low-hanging fruit of reducing space amplification has been harvested. Reducing CPU overheads improves the performance of the few applications where the CPU is indeed constraining. More importantly, optimizations that reduce CPU overheads allow for hardware configurations that are more cost-effective—until several years ago, the price of CPUs and memory was reasonably low relative to SSDs, but CPU and memory prices have increased substantially, so decreasing CPU overhead and memory usage has increased in importance. Early efforts to lower CPU overhead included the introduction of prefix bloom filters, applying the bloom filter before index lookups, and other bloom filter improvements. There remains room for further improvement.

3.4 Adapting to Newer Technologies

New architectural improvements related to SSDs could easily disrupt RocksDB's relevancy. For example, open-channel SSDs [79, 95], multi-stream SSDs [99], and ZNS [6] promise to improve query latency and save flash erase cycles. However, these new technologies would benefit only a small minority of the applications using RocksDB, given that most applications are space-constrained, not

erase cycle, or latency constrained. Further, having RocksDB accommodate these technologies directly would challenge the unified RocksDB experience. One possible path worth exploring would be to delegate the accommodation of these technologies to the underlying file system, perhaps with RocksDB providing additional hints.

In-storage computing might potentially offer significant gains, but it is unclear how many RocksDB applications would actually benefit from this technology. We suspect it would be challenging for RocksDB to adapt to in-storage computing, as it would likely require API changes to the entire software stack to fully exploit. We look forward to future research on how best to do this.

Disaggregated (remote) storage appears to be a much more interesting optimization target and is a current priority. So far, our optimizations have assumed the SSDs are locally attached, as our system infrastructure is primarily configured this way. However, faster networks currently allow many more I/Os to be served remotely, so the performance of running RocksDB with remote storage has become viable for an increasing number of applications. With remote storage, it is easier to make full use of both CPU and SSD resources at the same time, because they can be separately provisioned on demand (something much more difficult to achieve with locally attached SSDs). As a result, optimizing RocksDB for remote flash storage has become a priority. We are currently addressing the challenge of long I/O latency by trying to consolidate and parallelize I/Os. We have adapted RocksDB to handle transient failures, pass QoS requirements to underlying systems, and report profiling information. However, more work is needed.

Byte-addressable, **non-volatile memory (NVM)** [12], such as Intel's Optane DC [47], is a promising technology being commercialized. We are investigating how best to take advantage of of this technology. Several possibilities worth considering are:

(1) use NVM as an extension of DRAM—this raises the questions of (*i*) how best to implement key data structures (e.g., block cache or MemTable) with a combination of DRAM and NVM, and (*ii*) what overheads are introduced when trying to exploit the offered persistency;
(2) use NVM as the main storage of the database—however, we note that RocksDB tends to be bottlenecked by space or CPU, rather than I/O; and
(3) use NVM for the WALs—it is questionable whether this use case alone justifies the costs of NVM, considering that we only need a small staging area before it is moved to SSD.

3.5 Appropriateness of LSM-trees Revisited

We continuously revisit the question of whether LSM-trees remain appropriate, but repeatedly come to the conclusion that they do. The price of SSDs has not dropped enough to change the space and flash endurance bottlenecks for most use cases. While our main conclusion remains the same, we continue to hear requests to further reduce write amplification to below that which RocksDB can provide. We noted that some of the high write rate use cases have many large objects. If we separate large objects and store them separately, then they can be compacted less frequently so we can write less to SSDs. It is not uncommon for databases to store large objects separately for different reasons, including recent systems WiscKey [56] and ForrestDB [3]. We are also adding similar support with a new feature (called BlobDB [93]).

4 LESSONS ON SERVING LARGE-SCALE SYSTEMS

RocksDB is a building block for a wide variety of large-scale distributed systems with disparate requirements. Over time, we learned that improvements were needed with respect to resource management, support for replication and backups, WAL treatment, and data format compatibility.

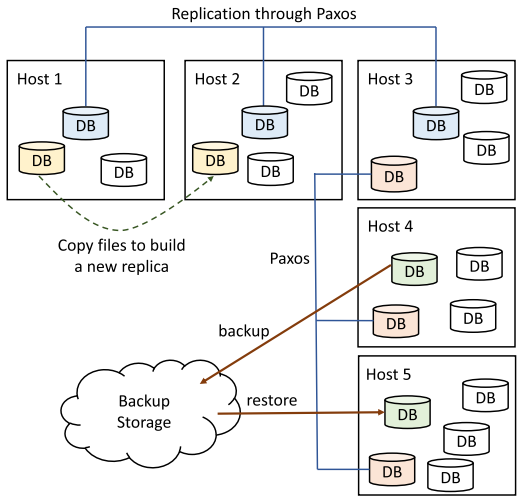

Fig. 4. Typical database service using RocksDB.

4.1 Resource Management

Large-scale distributed data services typically partition the data into *shards* that are distributed across multiple server nodes for storage. For example, Figure 4 shows how a database service might be using RocksDB. The size of shards is limited, because a shard is the unit for both load balancing and replication, so shards in a consistent state need to be copied between nodes. As a result, each server node will typically host tens or hundreds of shards. In our context, a separate RocksDB instance is used to service each shard, which means that a storage host will have many RocksDB instances running on it. These instances can either all run in one single address space or each in its own address space.

The fact that a host may run many RocksDB instances has resource management implications. Resources that need to be managed include: (1) the memory for write buffer, MemTables, and block cache, (2) compaction I/O bandwidth, (3) compaction threads, (4) total disk usage, and (5) file deletion rate (described below). Some of these resources need to be managed on a per-I/O device basis. Given that multiple instances share the host's resources, the resources need to be managed both globally (per host) and locally (per instance) to ensure they are used fairly and efficiently. RocksDB allows applications to create one or more resource controllers (implemented as C++ objects passed to different DB objects) for each type of resource mentioned above and also do so on a per instance basis [28, 29, 31]. For example, a C++ object implementing a compaction rate limiter that, say, limits the total compaction rate to 100 MB/s can be passed to multiple instances, so in aggregate the compaction rate will not exceed this rate. Finally, we learned it is important to support prioritization among RocksDB instances to make sure a resource is prioritized for the instances that need it most.

Another lesson we learned when running multiple instances in one process: Threads doing similar type of work (e.g., background flushes) should be in a pool that is shared across all similar instances (e.g., shards of a database) on a host. A shared host-wide pool limits the number of CPU cores that can be simultaneously saturated to the size of the thread pool. Since RocksDB background threads involve I/O intensive work, capping the number of those threads also limits I/O. Unpooled or private, per instance thread pools would not be able to limit parallelism across instances, and CPU and I/O spikes can occur when demand for background work is simultaneously

high across many instances. However, with a shared pool, instances not lucky enough to be allocated a thread from the pool will need to enter their work into a queue. In extreme cases, delaying background work for a long period of time can cause the LSM-tree to become bloated with an attendant increase in read- and space-amplification, or can cause the configuration limits set to protect those amplification factors to be breached, possibly causing foreground write stalls.

Global (per host) resource management is more challenging when the RocksDB instances run in separate processes, given that the processes execute independently and do not naturally share information. Two strategies can be applied. First, each instance could be configured to use resources conservatively, as opposed to greedily. For example, with compaction, each instance would initiate fewer compactions than "normal" and ramp up only when compactions are behind. The downside of this strategy is that the global resources may not be fully exploited, leading to sub-optimal resource usage. The second, more challenging, strategy is for the instances to share resource usage information among themselves and to adapt accordingly in an attempt to optimize resource usage more globally. More work will be needed to improve host-wide resource management in RocksDB.

4.2 Support for Replication and Backups

RocksDB is a single node library. The applications that use RocksDB are responsible for replication and backups if needed. Each application implements these functions in its own way (for legitimate reasons), so it is important that RocksDB offers appropriate support for these functions.

4.2.1 Replication. Bootstrapping a new replica by copying all the data from an existing one can be done in two ways. First, all the keys can be read from a source replica and then written to the destination replica. We call this *logical copying*. On the source side, snapshots ensure a consistent view of the source data. Further, RocksDB supports data scanning operations by offering the ability to minimize the impact on concurrent online queries; e.g., by providing the option to not cache the result of these operations and thus prevent cache trashing. On the destination side, bulk loading is supported and also optimized for this scenario.

Second, bootstrapping a new replica can be done by copying SSTables and other files directly. We call this *physical copying*. RocksDB assists physical copying by identifying existing database files at a current point in time and preventing them from being mutated or deleted while the copy is ongoing. Supporting physical copying is an important reason RocksDB stores data on an underlying file system, as it allows each application to use its own tools. We believe the potential performance gains of RocksDB directly using a block device interface or heavily integrating with the SSD **Flash Translation Layer (FTL)** does not outweigh the aforementioned benefit.

4.2.2 Backups. Backup is an important feature for most databases and other applications. For backups, applications have the same logical vs. physical choice as with replication. One difference between backups and replication is that applications often need to manage multiple backups. While most applications implement their own backups (to accommodate their own requirements), RocksDB provides a backup engine for applications to use if their backup requirements are simple. This feature helps users bootstrap functionality that is important, but easily over-looked.

4.2.3 Challenges on Updating Replicas. With replicas, updates need to be applied to each replica in a consistent order. A straightforward solution is to issue the writes to each replica sequentially, but this has negative performance implications, as the application will not be able to issue the writes concurrently from multiple threads. Further, when a replica falls behind in its updates, one needs a mechanism for it to catch up faster.

While various solutions have been used by different applications to address these issues, they all have limitations [33]. The challenge is that applications could increase write throughput by

issuing writes slightly out of order, but then they will not be able to serve reads from a consistent state. One possible solution is for users to do snapshot reads with their own sequence numbers if RocksDB supports multi-versioning with user defined timestamps, as described in Section 7.

4.3 WAL Treatment

Traditional databases tend to force a **write-ahead-log (WAL)** write upon every write operation to ensure durability. In contrast, large-scale distributed storage systems typically replicate data for performance and availability, and they do so with various consistency guarantees. For example, if copies of the same data exist in multiple replicas, and one replica becomes corrupted or inaccessible, then the storage system uses valid replica(s) from other unaffected hosts to rebuild the replica of the failed host. For such systems, RocksDB WAL writes are less critical. Further, distributed systems often have their own replication logs (e.g., Paxos logs), in which case RocksDB WALs are not needed at all.

We learned it is helpful to provide options for tuning WAL sync behavior to meet the needs of different applications [32]. Specifically, we introduced differentiated WAL operating modes: (*i*) synchronous WAL writes, (*ii*) buffered WAL writes, and (*iii*) no WAL writes at all. For buffered WAL treatment, the WAL is periodically written out to disk in the background at low priority so as not to impact RocksDB's traffic latencies.

4.4 Data Format Compatibility

Large-scale distributed applications run their services on many hosts, and they expect zero downtime. As a result, software upgrades are incrementally rolled out across the hosts, and when issues arise, the updates are rolled back. In light of continuous deployment [84], these software upgrades occur frequently; for example, RocksDB issues a new release once a month. For this reason, it is important that the data on disk remain both backward- and forward-compatible across the different software versions. A newly upgraded (or rolled back) RocksDB instance must be able to make sense of the data stored on disk by the previous instance. Further, RocksDB data files may need to be copied between distributed instances for replica building or load balancing, and these instances may be running different versions. A lack of a forward compatibility guarantee caused operational difficulties in some RocksDB deployments, which led us to add this guarantee.

RocksDB goes to great lengths to ensure data remains both forward- and backward-compatible (except for new features). This is challenging both technically and process-wise, but we have found that the effort pays off. For backwards compatibility, RocksDB must be able to understand all formats previously written to disk, which adds considerable software and maintenance complexities. For forward compatibility, future data formats need to be understood, and we aim to maintain forward compatibility for at least one year. This can be achieved, in part, by using generic techniques, such as those used by Protocol Buffer [43] or Thrift [90]. For configuration file entries, RocksDB needs to be able to identify new, unknown configuration entries and use best-effort guesses on how to apply the entries or when to disregard the entries. We continuously test different versions of RocksDB with different versions of its data.

5 LESSONS ON FAILURE HANDLING

Through production experience, we have learned three major lessons with respect to failure handling. First, data corruption needs to be detected early to minimize the risk of data unavailability or data loss, and in doing so to pinpoint where the error originated. The longer corrupted data is left in place, the higher the risk the remaining replicas will have a simultaneous outage, which would result in data unavailability; and the higher the risk the same data in the remaining replicas is also corrupted, which could result in permanent data loss. RocksDB achieves early detection by

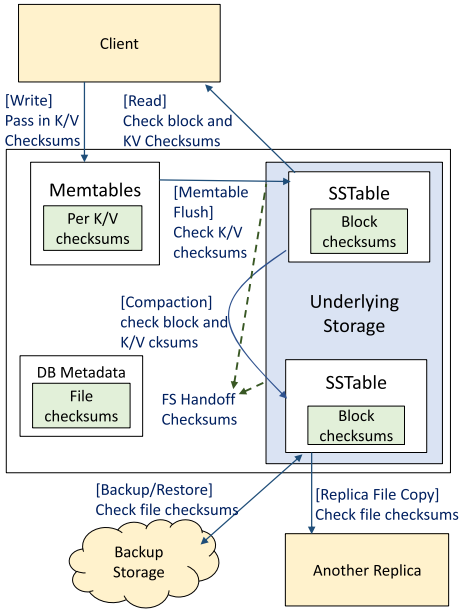

Fig. 5. Four types of checksums.

checksumming data at multiple layers and verifying those checksums as the data traverses through the system.

Second, integrity protection must cover the entire system to prevent silent hardware data corruptions [18, 45] from being exposed to RocksDB clients or spreading to other replicas (see Figure 5). Detecting corruptions only when the data is at rest, or when data is sent over the wire, is insufficient, because corruptions can be introduced by faulty software, a faulty CPU, or other faulty hardware components. While infrastructure services continuously scan all hosts in a fleet to remove those with faulty hardware, some hardware flaws may not be detected by the scan.

Third, errors need to be treated in a differentiated manner. Initially, we treated all non-EINTR filesystem errors the same. If an error was encountered in the read path, then we simply propagated the error to the client that issued the request. If an error was encountered in the write path, then we treated it as unrecoverable and permanently froze all writes; to resume writing, the RocksDB instance had to be restarted, with attendant operational overheads. To minimize such occurrences, we began classifying errors by severity and only interrupt operations for errors deemed unrecoverable.

5.1 Frequency of Silent Corruptions

An interesting question is how frequent silent data corruptions occur in practice within RocksDB. This is not straightforward to assess. Storage devices that do not offer end-to-end data protection (e.g., DIF/DIX[5]) support are typically used, primarily for cost reasons. Instead, applications rely on RocksDB block checksums to detect storage media corruptions. While applications that use RocksDB often run data consistency checks that compare replicas for integrity, the errors they catch could have been introduced either by RocksDB or by the client application (e.g., when replicating, backing up, or restoring data).

[5] DIF: Data Integrity Field (a.k.a. T10 PI) [14]. DIX: Data Integrity Extension.

However, one way we can estimate the frequency of corruptions introduced at the RocksDB level is to compare primary and secondary indexes in the MyRocks database tables that have both; any inconsistencies would have been introduced at the RocksDB level and would include those caused by CPU or memory corruption. Based on our measurements, corruptions are introduced at the RocksDB level roughly once every three months for each 100 PB of data. Worse, in 40% of those cases, the corruption had already propagated to other replicas.

Data corruptions also occur when transferring data, often because of software bugs. For example, a bug in the underlying storage system when handling network failures caused us to see, over a period of time, roughly 17 checksum mismatches for every petabyte of physical data transferred.

5.2 Multi-layer Protection

Data corruptions need to be detected as early as possible, before they are further propagated, to minimize downtime and data loss. Most RocksDB applications have their data replicated on multiple hosts and, often, they run consistency checks by comparing the checksums of the replicas. When a checksum mismatch is detected, the corrupt replica is discarded and replaced with a correct one. However, this is a viable option only as long as a correct replica still exists and is accessible.

Today, RocksDB checksums file data at multiple levels to identify corruptions that may have occurred in the layers beneath it. The different checksums (as well as the planned application layer checksum) are shown in Figure 5. Having multiple levels of checksums is important, as it helps detect corruptions early and protects against different types of threats.

5.2.1 Block Integrity. Block checksums, inherited from LevelDB, prevent data corrupted at or below the file system from being exposed to the client. Specifically, each SSTable block or WAL fragment has a checksum attached to it, generated when the data is created. Unlike the file checksum that is verified only when the file is moved, this checksum is verified every time the block is read, either to serve a request from the application or for compactions, due to its smaller scope.

5.2.2 SSTable Integrity. SSTable file checksums, added in 2020, protect against corruption caused by the underlying storage system from being propagated to other replicas and against corruption caused during the process of transferring SSTable files over the wire. Thus, each SSTable is protected by its own checksum, generated when the table is created. An SSTable's checksum is recorded in the metadata's SSTable file entry and is validated with the SSTable file wherever it is transferred.[6] However, we note that other files, such as WAL files, are still not protected this way.

5.2.3 Handoff Integrity. An established technique for detecting write corruptions early is to generate a handoff checksum on the data to be written to the underlying file system and pass it down along with the data, where it is verified by the lower layers [77, 101]. We wish to protect WAL writes using such a write API, since, unlike SSTables, WALs benefit from incremental validation on each append. Unfortunately, local file systems rarely support this.[7] However, when using remote storage, the write API can be changed to accept a separate checksum, hooking into the storage service's internal ECC. RocksDB can then use checksum combining techniques on the existing WAL fragment checksums to efficiently compute a write handoff checksum. Since our storage service performs write-time verification, we expect it to be extremely rare that corruptions will not be detected until they are read.

[6] These corruptions would eventually be detected when the corrupted blocks are read to serve client queries or for compaction. However, this might be too late, as a copy of the uncorrupted data may then no longer be available.
[7] Some specialized stacks, such as Oracle ASM [78], do support this.

5.3 End-to-end Protection with Key-value Integrity Protection

While the layers of protection described above prevent clients from being exposed to corrupt data in many cases, they are not comprehensive. One deficiency of the protections mentioned so far is that data is unprotected above the file I/O layer; e.g., data in the MemTable and the block cache. Data corrupted at this level will be undetectable and thus will eventually be exposed to the application. Further, flush or compaction operations can persist corrupted data, making the corruption permanent.

To address this problem, we are currently implementing per-KV checksums to detect corruptions that occur above the file I/O layer. This checksum will be transferred along with the KV pair wherever it is copied, although we will elide it from file data where alternative integrity protection already exists.

5.4 Severity-based Error Handling

Most of the faults RocksDB encounters are errors returned by the underlying storage system. These errors can stem from a multitude of issues, from severe problems like a read-only file system, to transient problems such as a full disk or a network error accessing remote storage. Early on, RocksDB reacted to such issues either by simply returning error messages to the client in the case of reads or by permanently halting all write operations.

Today, we aim to interrupt RocksDB operations only if the error is not locally recoverable; e.g., transient network errors should not require user intervention to restart the RocksDB instance. To implement this, we improved RocksDB to periodically retry operations after encountering an error classified as transient. By doing so, we obtain operational benefits, as clients do not need to manually mitigate RocksDB for a significant portion of faults that occur.

6 LESSONS FROM CONFIGURATION MANAGEMENT AND CUSTOMIZABILITY

RocksDB is highly configurable so applications can optimize for their workload. However, we have found configuration management to be a challenge (Section 6.1). At the same time, we have found customizability through call-back functions to be surprisingly powerful (Section 6.2). In Section 6.3 and Section 6.4, we highlight some of the configuration complexities associated with KV-pair deletions and memory management.

6.1 Managing Configurations

Initially, RocksDB inherited LevelDB's method of configuring parameters where the parameter options were directly embedded in the code. This caused two problems. First, parameter options were often tied to the data stored on disk, causing potential compatibility issues when data files created using one option could not be opened by a RocksDB instance newly configured with another option. Second, configuration options not explicitly specified in the code were automatically set to RocksDB's default values. When a RocksDB software update included changes to the default configuration parameters (e.g., to increase memory usage or compaction parallelism), applications would sometimes experience unexpected consequences.

To address these issues, RocksDB first introduced the ability for a RocksDB instance to open a database with a string parameter containing the specified configuration options. Later, RocksDB introduced support for optionally storing an options file along with the database. We also introduced two tools: (*i*) a validation tool that validates whether the options for opening a database was compatible with the target database; and (*ii*) a migration tool that rewrites a database to be compatible with the desired options (although this tool is limited).

Table 5. The Number of Distinct Configurations Used across 39 ZippyDB Deployments

Config Area:	Compaction	I/O	Compression	SSTable file	Plug-in functions
Configurations:	14	4	2	7	6

A more serious problem with RocksDB configuration management is the large number of configuration options. In the early years of RocksDB, we consciously made the design choice of supporting many configuration options, and over time, we introduced many new "knobs" and introduced support for pluggable components, all with the goal of allowing applications to realize their performance potential. This proved to be a successful strategy for gaining traction early on. However, a common complaint now is that there are far too many options and that it is too difficult to understand their effects; i.e., it has become very difficult to specify an "optimal" configuration.

More daunting beyond having many configuration parameters to tune is the fact that the optimal configuration depends not just on the application that has RocksDB embedded, but also on the workload generated by the clients of that application. Consider, for example, ZippyDB, an in-house developed, large-scale distributed key-value store with RocksDB embedded [92]. ZippyDB serves numerous different applications, sometimes individually, sometimes in a multi-tenant setup. Although significant efforts go into using uniform configurations across all ZippyDB use cases wherever possible, the ZippyDB workloads are so different for the different use cases, a uniform configuration is not practically feasible when performance is important. Table 5 shows that across the 39 ZippyDB deployments we sampled, over 25 distinct RocksDB configurations were being used.

Tuning configuration parameters is particularly challenging for applications with RocksDB embedded that are shipped to third parties. Consider a third party using a database such as MySQL or ZippyDB for one of their applications. The third party will typically know very little about RocksDB and how it is best tuned. And the database owners have little appetite for tuning the systems of their clients.

These real-world lessons triggered changes in our configuration support strategy. We have spent considerable effort on improving out-of-box performance and simplifying configurations. Our current focus is on providing *automatic adaptivity*, while continuing to support extensive explicit configuration given that RocksDB continues to serve specialized applications. We note that pursuing adaptivity while retaining explicit configurability creates significant code maintenance overhead, but we believe the benefits of having a consolidated storage engine outweighs the code complexity.

6.2 The Power of Call-back Functions

LSM-trees require data to be compacted periodically, where RocksDB re-writes the data to older levels. We believed it would be beneficial to have the application participate in the compaction process to allow them to add extra functionality that otherwise would require extra read and write operations. We added support for two call-back functions, referred to as *compaction filter* [24] and *merge operator* [25], that are defined by the application. Both are widely used today.

6.2.1 Compaction Filter. A compaction filter is a call-back function that is called during compaction by RocksDB for each KV-pair being processed. The application can then decide whether to (*i*) discard (remove) the KV-pair, (*ii*) modify the value, or (*iii*) leave the KV-pair as is.

This feature became popular almost immediately after it was released, and over time, it was being used in more and more ways. For example, it was used to implement **time-to-live (TTL)** functionality, where the expiration time was encoded within each KV pair, allowing the filter to remove the expired data during compaction. Or it was used to implement garbage collection as part of a **multi-version concurrency control (MVCC)** solution. Compaction filters were also

used to modify data, say, to migrate old data to a new format or to alter data based on time. Finally, compaction filters were sometimes used simply to collect statistics.

Compaction filters also turned out to be beneficial for certain administrative tasks that required all data to be scanned. While users can scan the entire dataset, and possibly delete or modify some of the data using `delete()` or `put()` operations, using compaction filters leads to a more efficient solution with significantly fewer I/O operations. Using compaction filters is also more convenient for many users in that they no longer have to manage periodic tasks and be concerned about write spikes that can be an operational challenge.

Using compaction filters has some serious limitations, however. For example, improperly using compaction filters can break some basic data consistency guarantees, and snapshot reads may no longer be repeatable (if the data is modified between reads). Hence, compaction filters are easier to use when consistency is not required. We consider the question of how to implement compaction filters more safely an open problem. Another limitation of compaction filters is that they do not allow multiple KV pairs to be atomically dropped or modified; e.g., it is not possible to atomically drop a KV-pair and the corresponding data in a secondary index. Despite these limitations, compaction filters are widely used, which in our view demonstrates that even imperfect solutions can go a long way if the application can find appropriate workarounds.

6.2.2 Merge Operator. RocksDB natively supports three types of operations to update the stored data: `put()`, `delete()`, and `merge()`. Each such operation results in the writing of a corresponding record to the MemTable and then SSTables. The merge operation allows the application to update the value of an existing key without first having to read the KV pair and without having to write out the entire KV-pair. On subsequent read or compaction operations, the contents of a merge record is "merged" with the contents of previous put or merge records of the same key. When a read operation or a compaction process encounters a merge record and a previous put record, or multiple merge records, RocksDB invokes an application-specified call-back merge operator, which can be used to combine them into a single one, which can be a put record or a merge record.

A powerful use case of the merge operator is the implementation of read-modify-write operations, say, to implement a counter or to update an individual field within a complex document. Using the merge operator leads to far lower overheads compared to implementing it with separate reads and writes of entire KV pairs. However, it negatively affects read performance in that the search for a KV pair does not necessarily end when the first entry is found, and in the worst case, the search must traverse all levels or until a Put record is found for the key. (More frequent compactions mitigates this downside.)

As with the compaction filter, we learned that many applications are willing to adopt a feature such as the merge operator, even with disadvantages; in particular, they accept the disadvantages to be able to benefit from reducing the amount of write I/O.

6.3 Optimizing Deletions

Deletion is often an overlooked operation on LSM-trees. Since there is no way to directly remove key-value pairs, deletion is achieved by adding a special marker to the LSM-tree, called a *Tombstone*, to indicate that a key has been deleted. This makes deletes fast, but can make subsequent queries slower. We note that a tombstone for a given key cannot be removed from an SSTable during compaction unless it is certain that the key is not present in any SSTable at one of the older levels.[8]

[8]In our current implementation, a tombstone is removed only when it reaches the oldest level or when the key ranges maintained in the metadata of each SSTable indicate that the key is not present in any older level.

6.3.1 Range Scans over a Large Range of Tombstones. Applications often delete a large range of consecutive or nearby keys. If that is the case, then scans that require iterating over a key range may encounter many tombstones that have to be skipped over, negatively impacting query latencies and wasting CPU and I/O resources. For example, if keys are full path names of files in a file system, then deleting a directory will result in a (potentially) large sequence of keys that have been deleted. As another example, consider the implementation of a queue, where entries are deleted after they have been dequeued. The head of the queue will then naturally follow a large number of deleted keys. Iterating over these deleted entries can cause significant overhead but does not contribute to the query result. In the extreme, we have observed queries having to iterate over millions of tombstones, just to return a few KV-pairs.

One way to address this issue is to initiate compactions when there are many consecutive tombstones. We first added a feature whereby the likelihood of compaction is increased whenever the ratio of the number of tombstones to the number of total entries crosses the 50% threshold, and the likelihood is further increased as the ratio increases. This helps in some cases, but not when the ratio does not cross the 50% threshold, yet many tombstones happen to be adjacent to each other. Hence, we added another feature that allowed the application to tag an SSTable for compaction after it has been generated. Specifically, when generating an SSTable during compaction, each entry is fed to a plug-in, and when the SSTable file is complete, the plug-in is queried to determine whether the file should be further compacted, in this case to further push tombstones to older levels and ultimately be removed when they reach the oldest level. RocksDB can also report the number of tombstones scanned on a query, allowing the application to issue an explicit compaction request. Finally, we added a feature whereby scan operations cease iterating over the keys once a set number of tombstones have been encountered. The result of the scan will then be incomplete, which informs the application of a tombstone-heavy range it may wish to compact, and yields control to the application to decide whether to continue the scan or give up.

These features can mitigate the problem for many applications, at least to some degree. But they also have serious limitations. For one, compactions take time, and scans will continue to perform poorly while compaction is ongoing. Further, with a higher frequency of compactions, write amplification may increase substantially, which some applications may not be able to tolerate. One idea we have been exploring is to support range deletes. Then, when iterating over a deleted range, all keys in the range can be skipped over by seeking to the end of the range. However, finding an efficient implementation has been more challenging than we expected. To work well, it needs to scale to frequent range deletes. It also needs to be compatible with multi-versioning, so when two ranges overlap, they cannot simply be merged. Further work is still needed to achieve the performance we desire. For us, it is perhaps one of the more challenging issues RocksDB faces today, and we are seeking ideas for a better solution.

6.3.2 Reclaiming Disk Space. Typically, when data gets deleted, applications expect that the data will ultimately no longer consume disk space. But this can take time, namely, until the tombstone reaches the oldest level. This is a problem given that the only way for applications to reclaim disk space is by deleting data. Applications may wish to have deleted KV pairs removed from disk within a limited period of time. Compaction based on the number of deletes can help to speed up data removal, but it is often not fast enough. For this reason, we added a feature whereby the application can specify a time threshold and RocksDB will ensure that any tombstone representing deleted data will reach the oldest level within that threshold. This feature is implemented by having the SSTable maintain (in its metadata) the earliest (i.e., oldest) time an entry in the SSTable was first added to the system, with compactions scheduled accordingly.

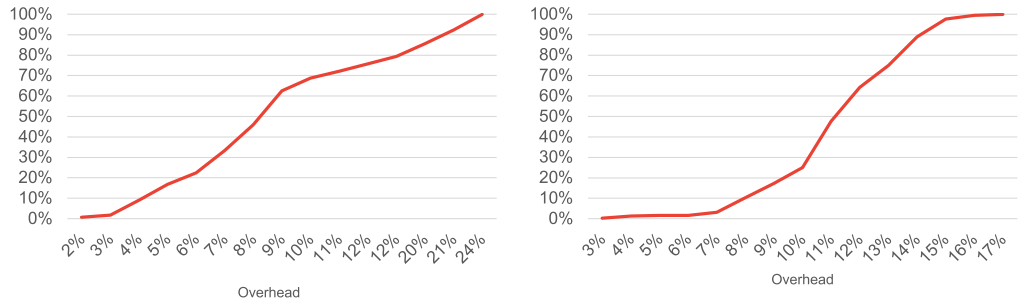

Fig. 6. CDF of memory overheads (external fragmentation + metadata) using jemalloc. *Left:* Samples taken from 40 ZippyDB clusters, with each data point representing the average of the cluster. *Right:* Samples taken from 953 hosts from a single representative ZippyDB cluster.

6.3.3 Rate-limited File Deletions. RocksDB typically interacts with the underlying storage device via a file system. These file systems are flash-SSD-aware; e.g., XFS, with realtime discard, may issue a TRIM command [48] to the SSD whenever a file is deleted. TRIM commands are commonly believed to improve performance and flash endurance [38], as validated by our production experience. However, they may also cause performance issues. TRIM is more disruptive than we originally thought: In addition to updating the address mapping (most often in the SSD's internal memory), the SSD firmware also needs to write these changes to FTL's[9] journal in flash, which in turn may trigger SSD's internal garbage collection, causing considerable data movement with an attendant negative impact on foreground I/O latencies. To avoid TRIM activity spikes and associated increases in I/O latency, we introduced rate limiting for file deletion to prevent multiple files from being deleted simultaneously (which happens after compactions).

6.4 Managing Memory

RocksDB has to manage a large amount of DRAM. Most of it is used for the SSTable block cache and for storing the MemTables. While databases tend to manage their own buffer pools, RocksDB instead relies on a third-party allocator; in particular, jemalloc [21]. This simplifies RocksDB's management of the block cache, as it only needs to track the allocations and the associated bookkeeping.

While the block size is configurable in RocksDB, fixed-sized blocks are not actually used, as for example in the case of the Unix buffer cache. Instead, RocksDB uses variable-sized blocks that are sized as close to the specified block size as possible. For example, if a key-value pair size exceeds the specified block size, then a larger block size is chosen to accommodate the pair exactly. Similarly, if a sequence of KV pairs fit within a block of the specified block size, but the next KV pair to be included would cause the block size to be larger than the specified size, then this next KV pair is not included and a smaller block size is used that exactly accommodates the original sequence. Finally, the top-level index and the bloom filter blocks are not of fixed size. Unlike in-place-update data structures, like B-trees, there are few benefits to using fixed-sized blocks strictly, so instead RocksDB delegates the management of variable block sizes to jemalloc.

In our experience, having jemalloc manage memory generally results in reasonable allocation and deallocation overheads. Memory overheads, including external fragmentation and metadata, are also reasonable for most applications. Figure 6 shows the cumulative distribution of memory overheads using jemalloc. The left curve depicts the distribution of average memory overheads for

[9]FTL: Flash Translation Layer.

40 ZippyDB application clusters in our operating environment; the memory overheads vary from between 2% and 25%. Still, external fragmentation can be an issue for some applications. These applications can, in principle, use a different allocator, or they can tune jemalloc appropriately (e.g., by following jemalloc's tuning guide [97]). For instance, one user noted that the source of a significant amount of fragmentation was in thread-specific memory pools (jemalloc's "tcache"), so they tuned related settings, which resulted in a memory saving of 12%. Another user found that jemalloc became inefficient over time. An investigation revealed this was due to interleaving allocations of long-term and short-term objects, causing memory usage to slowly grow, similar to when there is a memory leak. The user was able to address this issue by configuring RocksDB to allocate blocks in the block cache (which tend to be long-lived) from a separate jemalloc arena.

Nevertheless, RocksDB users often struggle to select effective memory-related configuration parameters. For example, many applications wish to allocate most of the host's DRAM to RocksDB, but find it difficult to appropriately set the amount of memory to use for the block cache and the MemTables. RocksDB is able to monitor allocated memory and internal fragmentation through malloc_usable_size() calls, so the memory limit users provide for the block cache and MemTables include both. However, RocksDB has little visibility to jemalloc's external fragmentation and metadata overheads, so users have to guess how much memory to reserve for them. Significant experimentation is often needed to find the best parameter settings. This is complicated by the fact that memory overheads may differ significantly across RocksDB instances even when supporting the same application. This is shown in the right graph of Figure 6 for a typical ZippyDB application. To avoid the fine-tuning of memory parameters for the different instances, users often just use conservative settings, which can result in suboptimal memory utilization with a good amount of DRAM not being used effectively. Ideally, memory configuration tuning would be fully automated.

We believe that the decision to rely on a third-party memory allocator was appropriate at the time, as it allowed us to focus on other, arguably more important areas to improve RocksDB. However, the memory manageability issues will likely have to be addressed in the near future to help users maximize the usage of a specified amount of physical memory without exceeding it.

7 LESSONS ON THE KEY-VALUE INTERFACE

The core **key-value (KV)** interface is surprisingly versatile with just four key operations: `put()`, `delete()`, `get()`, and iterators (scans). Almost all storage workloads can be served by a datastore with a KV API; we have rarely seen an application not able to implement the needed functionality using this interface. This is perhaps a reason why KV-stores are so popular. The KV interface is generic. Both keys and values are variable-length byte arrays. Applications have great flexibility in determining what information to pack into each key and value, and they can freely choose from a rich set of encoding schemes. Consequently, it is the application that is responsible for parsing and interpreting the keys and values. Another benefit of the KV interface is its portability. It is relatively easy to migrate from one key-value system to another.

However, while many use cases achieve excellent performance with this simple interface, we have noticed that it can limit performance for some applications. For example, building concurrency control outside of RocksDB is possible but hard to make efficient, especially if two-phase-commit requires some data persistence before committing the transaction. We added transaction support for this reason, which is used by MyRocks (MySQL+RocksDB) [61]. We continue to add features; e.g., range locking and large transactions support.

In other cases, the limitation is caused by the key-value interface itself. Accordingly, we have started to investigate possible extensions to the basic key-value interface. Here, we describe two possible extensions: support for application-specified timestamps and support for columns.

7.1 Versions and Timestamps

Over the past few years, we have come to understand the importance of data versioning. We have concluded that version information should become a first-class citizen in RocksDB to properly support features, such as **multi-version concurrency control (MVCC)** and point-in-time reads. To achieve this, RocksDB needs to be capable of accessing different versions efficiently.

To date, RocksDB has been using 56-bit sequence numbers internally to identify different versions of KV-pairs. The sequence number is generated by RocksDB and incremented on every client write (hence, all data is logically arranged in sorted order). The client application cannot affect the sequence number. However, RocksDB allows the application to take a *Snapshot* of the DB, after which RocksDB guarantees that all KV pairs that existed at the time of the snapshot will persist until the snapshot is explicitly released by the application. As a result, multiple KV-pairs with the same key may co-exist, differentiated by their sequence numbers.

This approach to versioning is inadequate, as it does not satisfy the requirements of many applications. To read from a past state, a snapshot must have already been taken at the previous point in time. RocksDB does not support taking a snapshot of the past, since there is no API to specify a time-point. Moreover, it is inefficient to support point-in-time reads. Finally, each RocksDB instance assigns its own sequence numbers, and snapshots can be obtained only on a per-instance basis. This complicates versioning for applications with multiple, (possibly replicated) shards, each of which is a RocksDB instance. As a result, it is effectively impossible to create versions of data that offer cross-shard consistent reads.

Applications can work around these limitations by encoding timestamps within the key or within the value. However, they will experience performance degradations in either case. Encoding within the key sacrifices performance for point-lookups, while encoding within the value sacrifices performance for out-of-order writes to the same key and complicates the reading of old versions of keys. We believe adding support for application-specified timestamps would better address these limitations, where the application can tag its data with timestamps that can be understood globally, and do so outside the key or value.

We have added basic support for application-specified timestamps [27] and evaluated this approach with DB-Bench. The results are shown in Table 6. Each workload has two steps: The first step populates the database, and in the second step, we measure performance. For example, in "`fill_seq + read_random`," we populate the initial database by writing a number of keys in ascending order and in step 2 perform random read operations. Relative to the baseline where the application encodes a timestamp as part of the key (transparent to RocksDB), the application-specified timestamp API can lead to a 1.2× or better throughput gain. The improvements arise from treating the timestamp as metadata separate from the key, because then point lookups can be used instead of iterators to get the newest value for a key, and Bloom filters may identify SSTables not containing that key. Additionally, the timestamp range covered by an SSTable can be stored in its properties, which can be leveraged during a search to exclude SSTables that would only contain stale values.

We hope this feature will make it easier for users to implement multi-versioning in their systems for single node MVCC, distributed transactions, or resolving conflicts in multi-master replication. The more complicated API, however, is less straightforward to use and perhaps prone to misuse. Further, the database would consume more disk space than when storing no timestamps and would be less portable to other systems.

7.2 Column Support

Some RocksDB applications expose data organized into columns, including SQL-based database implementations (e.g., MyRocks [61], Rocksandra [44], CockroachDB [91], and TiDB [46]). In

Table 6. DB_bench Micro-benchmark Using the Timestamp API Sees ≥ 1.2× Throughput Improvement

workload	throughput gain
fill_seq + read_random	1.2
fill_seq + read_while_writing	1.9
fill_random + read_random	1.9
fill_random + read_while_writing	2.0

principle, data with columns can be naturally stored in RocksDB using the standard key-value interface; for example, by encoding an entire row within one key-value entry. The resulting performance is often reasonable, perhaps after applying a few straightforward application-level optimizations. However, supporting columns directly allows for significant optimizations for a number of use cases.

For example, if large objects are stored in some columns and other columns are updated frequently, then the updates incur high overheads, because the entire row has to be written each time any column element is modified (assuming a row per key-value pair organization). Supporting columns directly would allow the data of some columns to be written without having to write out the entire row. The performance benefits of column support would be even more pronounced when versioned updates that might arrive out of order are involved. As another example, if a query only involves a subset of the columns, or if only a subset of columns are being written to blindly, then reading entire rows can be avoided if columns are supported.

Some applications have attempted to improve performance for cases similar to those described in these examples. For example, Rocksandra applies partial updates of rows by using the *merge* operator. This improves update performance but at the cost of read performance. Another approach is to encode each column of a row as a separate KV pair. The downsides of this approach are that (*i*) reading an entire row now becomes a range query with attendant extra overheads, and (*ii*) deleting or replacing entire rows becomes much more complex and difficult to do using blind writes.

We believe that adding direct support for columns—which requires an extended API—would enable substantial performance benefits for some important use cases in that:

(1) updates and reads of individual column data can be made much more efficient;
(2) when applications issue a query filtering out values for specific columns, some filtering could be pushed down with specialized filters at the SSTable level;
(3) some columns can be stored in separate column families (Section 2.2); and
(4) when columns are identified explicitly, data can be compressed more efficiently using techniques similar to those used in columnar databases [1].

Further, having direct support for columns would allow RocksDB to validate data integrity between primary and secondary indexes.

8 LESSONS FROM FAILED INITIATIVES

While developing RocksDB, we designed and implemented a number of features that in retrospect turned out to not be as effective as we thought they would. We highlight three of them.

8.1 Support for DRAM-based Storage Media

In 2014, we decided to optimize RocksDB for media with access latencies far lower than those of SSDs; Ramfs was a primary target. To allow users to realize the potential performance benefits of using DRAM directly, we made both the SSTable format and the MemTable format pluggable [30], and we developed Ramfs-friendly plugins. For example, instead of using a standard block-based SSTable, we introduced Ramfs "plain tables" that could be mmapped. This allowed the tables to

be accessed using byte addressing without the extra memory copying, block lookups, and block caching otherwise required when using block-based tables. Further, hash-based indexes could be used for faster lookup within the SSTable and MemTable.

In a sense, this effort was successful in that some applications did indeed adopt and use these features. Strategically, however, this effort was introduced too early in retrospect. Large-scale, pure in-memory *persistent* storage systems simply never gained as much traction as we had originally expected. In-memory data applications, such as in-memory caches (e.g., Memcached), search indexes, and machine learning inference systems, generally did not incorporate RocksDB as part of their solutions, likely because using DRAM directly typically led to far better efficiencies.

8.2 Support for Hybrid Storage Media

SSDs are faster and offer far higher bandwidth than HDDs, but come at a higher cost and have limited write endurance. Hence, we believed the future would include hybrid SSD/HDD storage solutions that lead to better price-performance tradeoffs for many workloads. As a result, we started to add experimental features to support hybrid storage media. For example, in 2014, we added support to allow assigning different data paths to different LSM-tree levels so different levels could reside on different storage media. We also experimented with adding support for alternative caching solutions, such as Flashcache [63], and a new proprietary persistent cache that we integrated into the RocksDB core logic so it could evolve faster.

While these features were just early initial steps, we had hoped users would start using them and we could evolve the features from there. In reality, few users expressed interest. At the time, the hybrid solution required mounting both SSD and HDD media on a host to work well, but we found this type of configuration to be rare in practice. We concluded that hybrid storage solutions were more likely to be of interest with the advent of remote storage, something we had only started to work on in 2018. We have recently revived the hybrid storage media project to support hybrid local/remote storage, but believe more sophisticated approaches are necessary. For example, we noticed that in some applications hot and cold data might be collocated within a data block, so we need to separate them in compaction. Another lesson is that some extra information associated with KV pairs, such as the age of the data, can be useful to predict data coldness.

8.3 Richer, Higher-level Interfaces

RocksDB primarily supports the traditional key-value interface. Over the years, we made several attempts to extend this interface to make RocksDB more convenient to use for some applications. For example, we added an interface to support functionality similar to Redis lists in 2013; two spatial interfaces, namely, GeoDB and SpatialDB, in 2014; and support for documents in 2015. All of these interfaces were implemented in a layer on top of the core key-value interface.

However, none of these interfaces were adopted in a significant way, and they were ultimately deprecated and removed. We concluded—after the fact—that investments in improving core functionality creates far more value. Most RocksDB users were satisfied with the simple KV interface and were able to easily build their own high-level interfaces on top of that without needing explicit support from RocksDB. Their primary pain points were in fact efficiency and manageability, which led us to conclude that we needed to focus our investments in those areas. In particular, we learned that the core KV interface should be extended only if it helped improve performance dramatically, such as the two initiatives described in the previous section.

9 RELATED WORK

This article is an extended version of a paper presented at the 19th Usenix Conference on File and Storage Technologies [20]. New material in this article includes descriptions of RocksDB column

families; RocksDB callback functions such as compaction filters and merge operators; optimizing deletions in RocksDB; memory management in RocksDB; column support in RocksDB; and descriptions of lessons from failed initiatives.

Overall, our work on RocksDB has benefited from a broad range of research in a number of areas, including the following:

Storage engine libraries. Many storage engines have been built as a library to be embedded in applications. RocksDB's KV interface is more primitive than, for example, BerkeleyDB[71], SQLite[75], and Hekaton[17]. Further, RocksDB differs from these systems by focusing on the performance of modern server workloads, which require high throughput and low latency and typically run on high-end SSDs and multicore CPUs. This differs from systems with more general targets, or built for faster storage media [17, 50].

Key-value stores for SSDs. Over the years, much effort has gone into optimizing key-value stores, especially for SSDs. As early as 2011, SILT [54] proposed a key-value store that balanced between memory efficiency, CPU, and performance. ForestDB [45] uses HB+ trees to index on top of logs. TokuDB [52] and other databases use FractalTree/Bϵ trees. LOCS [96], NoFTL-KV [95], and FlashKV [100] target Open-Channel SSDs for improved performance. While RocksDB benefited from these efforts, our position and strategy for improving performance is different, and we continue to depend on LSM-trees. Several studies have compared the performance of RocksDB with other databases, such as InnoDB [66], TokuDB [19, 61], and WiredTiger [8].

LSM-tree improvements. Several systems also use LSM-trees and improved their performance. Write amplifications is often the primary optimization goal; e.g., WiscKey [56], PebblesDB [81], IAM-tree [42], and TRIAD [5]. These systems go further in optimizing for write amplification than RocksDB, which focuses more on tradeoffs among different metrics. SlimDB [82] optimized LSM-trees for space efficiency; RocksDB also focuses on deleting dead data. Monkey [16] attempts to balance between DRAM and IOPs. bLSM [85], VT-tree [88], and cLSM [41] optimize for the general performance of LSM-trees.

Large-scale storage systems. There are numerous distributed storage systems [11, 15, 34, 35, 62, 91]. They usually have complex architectures spanning multiple processes, hosts, and data centers. They are not directly comparable to RocksDB, a storage engine library on a single node. Other systems (e.g., MongoDB [65], MySQL [67], Microsoft SQL Server [62]) can use modular storage engines; they have addressed similar challenges to what RocksDB faces, including failure handling and using timestamps.

Failure handling. Checksums are frequently used to detect data corruption [7, 40, 67]. Our argument that we need both end-to-end and handoff checksums still mirrors the classic end-to-end argument [83] and is similar to the strategy used by others: Sivathanu et al. [89], ZFS [101, 102], and Linux [77]. Our argument for earlier corruption detection is similar to Reference [53], which argues that domain-specific checking is inadequate.

Timestamp support. Several storage systems provide timestamp support: HBase [35], WiredTiger [64], and BigTable [11]; Cassandra [34] supports a timestamp as an ordinary column. In these systems, timestamps are a count of the number of milliseconds since the UNIX epoch. Hekaton [17] uses a monotonically increasing counter to assign timestamps, which is similar to the RocksDB sequence numbers. RocksDB's ongoing work on application-specified timestamps is complementary to the aforementioned efforts. We hope key-value APIs with a application-specified timestamp extension can make it easier for higher-level clients to support features related to data versioning with low overhead in both performance and efficiency.

Column-aware data model. Column support is standard for relational databases [62, 67, 73]. Many storage engines expose a key-value interface [17, 66, 70, 71]. The WiredTiger storage engine, however, supports named columns [64]. The benefit of tuning per-column storage and caching looks promising for RocksDB as well.

10 FUTURE WORK AND OPEN QUESTIONS

Besides completing the improvements mentioned in the previous sections, including optimizations for disaggregated storage, key-value separation, multi-level checksums application-specified timestamps, column support, and memory management, we plan to unify leveled and tiered compaction and improve adaptivity. However, a number of open questions could benefit from further research.

(1) How can we use SSD/HDD hybrid storage to improve efficiency?
(2) How can we mitigate the performance impact on readers when there are many consecutive tombstone deletion markers?
(3) How should we improve our write throttling algorithms?
(4) How can we most efficiently compare two replicas to ensure they contain the same data?
(5) How can we best exploit NVM? Should we still use the LSM-tree? How should we organize a storage hierarchy that includes NVM?
(6) Can there be a generic integrity API to handle data handoff between RocksDB and the file system layer?

11 CONCLUSIONS

RocksDB has grown from a key-value store serving niche applications to its current position of widespread adoption across numerous industrial large-scale distributed applications. The LSM-tree as the main data structure has served RocksDB well, as it can be tailored for high write throughput, high read throughput, space efficiency, or something in between. Our view on performance has, however, evolved over time. While write and space amplification remain the primary concern, additional focus has shifted to CPU and DRAM efficiency, as well as remote storage.

Lessons learned from serving large-scale distributed systems (Section 4) include (*i*) resource allocation must be managed across multiple RocksDB instances, since a single server may host multiple instances; (*ii*) the data format used must be backward- and forward-compatible, since RocksDB software updates are deployed/rolled-back incrementally; and (*iii*) proper support for database replication and backups is important.

Lessons learned from dealing with failures (Section 5) include (*i*) data corruption needs to be detected early to minimize data unavailability and loss; (*ii*) integrity protection must cover the entire system to prevent silent corruptions from propagating to replicas and clients; and (*iii*) errors need to be treated in a differentiated manner.

Lessons related to configuration management (Section 6) indicate that having RocksDB be highly configurable has enabled many different types of applications and that suitable configurations can have a large positive performance impact, but also that configuration management is perhaps too challenging and needs to be simplified and automated.

Lessons on the RocksDB API (Section 7) indicate that the core interface is simple and powerful given its flexibility, but limits the performance for some important use cases; we presented our thoughts on improving the interface by supporting application-defined timestamps and columns.

Finally, we presented a number of development initiatives that in retrospect turned out to be misguided (Section 8).

APPENDICES

A ROCKSDB FEATURE TIMELINE

	Performance	Configurability	Features
2012	• Multi-threaded compactions		• Compaction filters • Locking SSTables from deletion
2103	• Tiered compaction • Prefix Bloom filter • Bloom Filter for MemTables • Separate thread pool for MemTable flush	• Pluggable MemTable • Pluggable file format	• Merge Operator
2014	• FIFO compaction • Compaction rate limiter • Cache-friendly Bloom filters	• String-based config options • Dynamic config changes	• Backup engine • Support for multiple key spaces ("column family") • Physical checkpoints
2015	• Dynamic leveled compaction • File deletion rate limiting • Parallel Level 0 and 1 compaction	• Separate config file • Config compatibility checker	• Bulk loading for SSTable file integration • Optimistic and pessimistic transactions
2016	• Different compression for last level • Parallel MemTable inserts	• MemTable total size caps across instances • Compaction migration tools	• DeleteRange()
2017	• Separate thread pool for bottom-most compactions • Two-level file indices • Level 0 to level 0 compactions	• Single memory limit for both block cache and MemTable	
2018	• Dictionary compression • Hash index into data blocks		• Automatic recovery from out-of-space errors • Query trace and replay tools
2019	• Batched MultiGet() with parallel I/O	• Configure plug-in function using object registry	• Secondary instance
2020	• Multithreaded single file compression		• Entire file checksum • Automatically recover from retriable errors • Partial support for user-defined timestamps

B RECAP OF LESSONS LEARNED

Some of the lessons we learned include:

(1) It is important that a storage engine can be tuned to fit different performance objectives. (Section 1)

(2) Space efficiency is the bottleneck for most applications using SSDs. (Section 3.2)
(3) CPU overhead is becoming more important to allow systems to run more efficiently. (Section 3.3)
(4) Global, per host, resource management is necessary when many RocksDB instances run on the same host. (Section 4.1)
(5) Having WAL treatment be configurable (synchronous WAL writes, buffered WAL writes, or disabled WAL) offers applications performance advantages. (Section 4.3)
(6) Data replication and backups need to be properly supported. (Section 4.2)
(7) RocksDB needs to provide both backward- and forward- compatibility with respect to stored data and configuration data. (Section 4.4)
(8) It is beneficial to detect data corruptions earlier, rather than eventually. (Section 5, Section 5.2)
(9) Integrity protection must cover the entire system to prevent corrupted data (e.g., caused by bitflips in CPU/memory) from being exposed to clients or other replicas; detecting corruption only when the data is at rest or being sent over the wire is insufficient. (Section 5)
(10) Error handling needs to be treated in a differentiated manner, depending on the causes and consequences of the errors. (Section 5)
(11) Silent CPU and memory corruptions do occur, although very rarely, and data replication does not always protect against them. (Section 5.1)
(12) Automatic configuration adaptivity is helpful in simplifying configuration management. (Section 6.1)
(13) Performance can be improved by providing user-defined callback functions. Existing partial solutions are helpful, but there is room for further improvement. (Section 6.2)
(14) Deleting consecutive keys introduces performance challenges when using LSM-trees. (Section 6.3)
(15) The SSD TRIM operation is good for performance but file deletions need to be rate limited to prevent occasional performance issues. (Section 6.3.3)
(16) Relying on a third-party memory allocator allowed us to focus on other, arguably more important areas, but it causes manageability issues. (Section 6.4)
(17) The key/value interface is versatile, but limits performance for some use cases; adding a timestamp separately from the key and value can offer a good balance between performance and simplicity. (Section 7.1)
(18) Applications can implement columns on top of RocksDB's key/value interface and obtain reasonable performance, but additional performance improvements are possible if the storage engine is aware of columns. (Section 7.2)

C RECAP OF DESIGN CHOICES REVISITED

Some notable design choices revisited include:

(1) Customizability is always good to users. (Section 4, Managing configurations)
(2) RocksDB can be blind to CPU bit flips. (Section 5)
(3) It is OK to panic when seeing any I/O error. (Section 5)

ACKNOWLEDGMENTS

We attribute the success of RocksDB to all current and past RocksDB team members at Facebook, all those who made contributions in the open-source community, as well as RocksDB users. We especially thank Mark Callaghan, the mentor to the project for years, as well as Dhruba Borthakur, the lead founding member of RocksDB. We also appreciate comments on the article by Jason Flinn

and Mahesh Balakrishnan. Finally, we thank our shepherd for the conference version of the article, Ethan Miller, and the anonymous reviewers for both the conference and journal versions of the paper for their valuable feedback—they have helped make this a far better article.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: How different are they really? In *Proceedings of the International Conference on Management of Data (SIGMOD'08)*. 967–980.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP'19)*. 353–369.

[3] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2015. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Trans. Comput.* 65, 3 (2015), 902–915.

[4] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing access methods: The RUM conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT'16)*. 461–466.

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

[6] Matias Bjørling. 2020. Zone Append: A new way of writing to zoned storage. In *Proceedings of the Usenix Linux Storage and Filesystems Conference (VAULT'20)*.

[7] Dhruba Borthakur. 2008. HDFS architecture guide. *Hadoop Apache Project*. Retrieved from http://hadoop.apache.org/docs/stable1/hdfs_design.pdf.

[8] Mark Callaghan. 2016. *MongoRocks and WiredTiger versus LinkBench on a small server*. Retrieved from http://smalldatum.blogspot.com/2016/10/mongorocks-and-wiredtiger-versus.html.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th Conference on File and Storage Technologies (FAST'20)*. 209–223.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bull. IEEE Comput. Societ. Technic. Commit. Data Eng.* 36, 4 (2015).

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. BigTable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 1–26.

[12] An Chen. 2016. A review of emerging non-volatile memory (NVM) technologies and applications. *Solid-state Electron.* 125 (2016), 25–38.

[13] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at Facebook. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. 1087–1098.

[14] Wikipedia contributors. 2020. *Data Integrity Field*. Retrieved from https://en.wikipedia.org/wiki/Data_Integrity_Field.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild et al. 2013. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 1–22.

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the International Conference on Management of Data (SIGMOD'17)*. 79–94.

[17] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 1243–1254.

[18] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. 2017. Optimizing space amplification in RocksDB. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'17)*.

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. 33–49.

[21] Jason Evans. 2019. *Jemalloc Memory Allocator*. Retrieved from http://jemalloc.net.

[22] Facebook. 2021. Column families. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Column-Families.
[23] Facebook. 2021. Compaction. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Compaction.
[24] Facebook. 2021. Compaction filter. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Compaction-Filter.
[25] Facebook. 2021. Merge operator. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Merge-Operator.
[26] Facebook. 2020. Universal compaction. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Universal-Compaction.
[27] Facebook. 2019. User Timestamps (Experimental). *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/User-Timestamp-%28Experimental%29.
[28] Facebook. 2020. SST File Manger. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/SST-File-Manager.
[29] Facebook. 2020. Write buffer manager. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Write-Buffer-Manager.
[30] Facebook. 2021. MemTable. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/MemTable#comparison.
[31] Facebook. 2021. Rate limiter. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Rate-Limiter.
[32] Facebook. 2021. Write Ahead Log. *RocksDB documentation wiki*. Retrieved from https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log.
[33] Jose Faleiro. 2019. The dangers of logical replication and a practical solution. In *Proceedings of the 18th International Workshop on High Performance Transaction Systems (HPTS'19)*.
[34] Apache Software Foundation. 2021. *Apache Cassandra*. Retrieved from https://cassandra.apache.org/.
[35] Apache Software Foundation. 2021. *Apache HBase*. Retrieved from https://hbase.apache.org/.
[36] Apache Software Foundation. 2016. Compaction. *Apache Cassandra Documentation*. Retrieved from https://murukeshm.github.io/cassandra/trunk/operating/compaction.html?highlight=tiered%20compaction#size-tiered-compaction-strategy.
[37] Apache Software Foundation. 2015. *HBase: Add Compaction Policy that Explores More Storefile Groups*. Retrieved from https://issues.apache.org/jira/browse/HBASE-7842.
[38] Tasha Frankie, Gordon Hughes, and Ken Kreutz-Delgado. 2012. A mathematical model of the trim command in NAND-flash SSDs. In *Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE'12)*. 59–64.
[39] S. Ghemawat and J. Dean. 2011. *LevelDB*. Retrieved from https://github.com/google/leveldb.
[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP'13)*. 29–43.
[41] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the European Conference on Computer Systems (EUROSYS'15)*. 1–14.
[42] Caixin Gong, Shuibing He, Yili Gong, and Yingchun Lei. 2019. On integration of appends and merges in log-structured merge trees. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP'19)*. 1–10.
[43] Google. 2021. *Protobuf*. Retrieved from https://opensource.google/projects/protobuf.
[44] Dikang Gu. 2018. Open-sourcing a 10x reduction in Apache Cassandra tail latency. *Instagram Engineering Blog*. Retrieved from https://instagram-engineering.com/open-sourcing-a-10x-reduction-in-apache-cassandra-tail-latency-d64f86b43589.
[45] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS'21)*. 9–16.
[46] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084.
[47] Intel. 2021. *Memory optimized for data-centric workloads*. Retrieved from https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dc-persistent-memory.html.
[48] Intel. 2021. *TRIM Overview*. Retrieved from https://www.intel.com/content/www/us/en/support/articles/000016148/memory-and-storage.html.
[49] Iron. 2021. *Simple, flexible, reliable serverless tools*. Retrieved from https://www.iron.io.

[50] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the International Conference on Management of Data (SIGMOD'15)*. 691–706.
[51] Jay Kreps. 2016. Introducing Kafka Streams: Stream processing made simple. *Confluent*. Retrieved from https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.
[52] B. Kuszmaul. 2010. *How TokuDB fractal tree indexes work*. Technical Report. TokuTek.
[53] Chuck Lever. 2012. End-to-end data integrity requirements For NFS. *Oracle Corp. slide deck*. Retrieved from https://datatracker.ietf.org/meeting/83/materials/slides-83-nfsv4-2.
[54] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd Symposium on Operating Systems Principles (SOSP'11)*. 1–13.
[55] Mike Lin. 2019. Faster BAM Sorting with Samtools and RocksDB. *Dnanexus Developer Blog*. Retrieved from https://devblog.dnanexus.com/faster-bam-sorting-with-samtools-and-rocksdb/.
[56] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in SSD-conscious storage. *ACM Trans. Stor.* 13, 1 (2017), 1–28.
[57] Scott Mansfield, Vu Tuan Nguyen, Sridhar Enugula, and Shashi Madappa. 2016. Application data caching using SSDs: The Moneta project: Next generation EVCache for better cost optimization. *Netflix Technology Blog*. Retrieved from https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.
[58] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J. Tsotras, and Neal E. Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *VLDB J.* 30, 3 (2021), 361–378.
[59] Mark Marchukov. 2017. LogDevice: A distributed data store for logs. *Facebook Engineering Blog*. Retrieved from https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/.
[60] Yoshinori Matsunobu. 2017. Migrating a database from InnoDB to MyRock. *Facebook Engineering Blog*. Retrieved from https://engineering.fb.com/core-data/migrating-a-database-from-innodb-to-myrocks/.
[61] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3217–3230. DOI: https://doi.org/10.14778/3415478.3415546
[62] Microsoft. 2021. *Microsoft SQL Server*. Retrieved from https://www.microsoft.com/en-us/sql-server/.
[63] Domas Mituzas. 2013. *Flashcache at Facebook: From 2010 to 2013 and beyond*. Retrieved from https://engineering.fb.com/2013/10/09/core-data/flashcache-at-facebook-from-2010-to-2013-and-beyond/.
[64] MongoDB. 2021. *WiredTiger Storage Engine*. Retrieved from https://docs.mongodb.com/manual/core/wiredtiger/.
[65] MongoDB. 2021. *The Database for Modern Applications*. Retrieved from https://www.mongodb.com.
[66] MySQL. 2021. *Introduction to InnodeDB*. Retrieved from https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html.
[67] MySQL. 2021. *MySQL*. Retrieved from https://www.mysql.com/.
[68] Xu Ning and Maxim Fateev. 2016. Cherami: Uber Engineering's durable and scalable task queue in Go. *Uber Engineering Blog*. Retrieved from https://eng.uber.com/cherami-message-queue-system/.
[69] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (2017), 1634–1645.
[70] University of Wisconsin Computer Sciences Department. 2012. *SHORE Storage Manager: The multi-threaded version*. Retrieved from https://research.cs.wisc.edu/shore-mt.
[71] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 183–191.
[72] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.
[73] Oracle. 2021. *Oracle Database*. Retrieved from https://www.oracle.com/database/.
[74] Keren Ouaknine, Oran Agra, and Zvika Guz. 2017. Optimization of RocksDB for Redis on flash. In *Proceedings of the International Conference on Compute and Data Analysis*. 155–161.
[75] Mike Owens. 2006. *The Definitive Guide to SQLite*. Apress.
[76] Percona. 2021. MongoRocks. *Percona server for MongoDB documentation*. Retrieved from https://www.percona.com/doc/percona-server-for-mongodb/3.4/mongorocks.html.
[77] Martin K. Petersen. 2008. Linux data integrity extensions. In *Proceedings of the Linux Symposium*, Vol. 4.
[78] Martin K. Petersen and Sergio Leunissen. 2010. Eliminating silent data corruption with Oracle Linux. *Oracle Corp. Slide Deck*. Retrieved from https://oss.oracle.com/~mkp/docs/data-integrity-webcast.pdf.
[79] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. 2020. Open-channel SSD (What is it good for). In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'20)*.
[80] Qihoo. 2021. *Pika*. Retrieved from https://github.com/Qihoo360/pika.

[81] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 497–514.

[82] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048.

[83] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst.* 2, 4 (1984), 277–288.

[84] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-C'16)*. 21–30.

[85] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log-structured merge tree. In *Proceedings of the International Conference on Management of Data (SIGMOD'12)*. 217–228.

[86] Arun Sharma. 2016. Dragon: A distributed graph query engine. *Facebook Engineering Blog*. Retrieved from https://engineering.fb.com/2016/03/18/data-infrastructure/dragon-a-distributed-graph-query-engine/.

[87] Arun Sharma. 2019. *How we use RocksDB at Rockset*. Retrieved from https://rockset.com/blog/how-we-use-rocksdb-at-rockset/.

[88] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST'13)*. 17–30.

[89] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. 2004. *Enhancing File System Integrity through Checksums*. Technical Report FSL-04-04. Computer Science Department, Stony Brook University.

[90] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper*. Retrieved from https://thrift.apache.org/static/files/thrift-20070401.pdf.

[91] 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. 2020. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the International Conference on Management of Data (SIGMOD'20)*. 1493–1509. DOI: https://doi.org/10.1145/3318464.3386134

[92] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. 2019. Who's afraid of uncorrectable bit errors? Online recovery of flash errors with distributed redundancy. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 977–992. Retrieved from https://www.usenix.org/system/files/atc19-tai.pdf.

[93] Levi Tamasi. 2021. Integrated BlobDB. *Facebook Engineering Blog*. Retrieved from https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html.

[94] Facebook RocksDB team. 2021. *A Persistent Key-value Store for Fast Storage Environments*. Retrieved from https://rocksdb.org.

[95] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. NoFTL-KV: Tackling write-amplification on KV-Stores with native storage management. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT'18)*. 457–460.

[96] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EUROSYS'14)*. 1–14.

[97] Qi Wang. 2018. *Jemalloc tuning*. Retrieved from https://github.com/jemalloc/jemalloc/blob/dev/TUNING.md.

[98] Tao Xu. 2015. RocksDB for personalized search at Airbnb. *Video talk*. Retrieved from https://www.codechannels.com/rocksdb-meetup-tao-xu-airbnb-rocksdb-for-personalized-search-at-airbnb/.

[99] Fei Yang, K. Dou, S. Chen, J. U. Kang, and S. Cho. 2015. Multi-streaming RocksDB. In *Proceedings of the Non-volatile Memories Workshop*.

[100] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. 2017. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 1–19.

[101] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Zettabyte reliability with flexible end-to-end data integrity. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST'13)*. 1–14.

[102] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. 29–42.

Received July 2021; accepted August 2021