Disaggregating RocksDB: A Production Experience

SIYING DONG, SHIVA SHANKAR P, SATADRU PAN, ANAND ANANTHABHOTLA, DHANABAL EKAMBARAM, ABHINAV SHARMA, SHOBHIT DAYAL, NISHANT VINAYBHAI PARIKH, YANQIN JIN, ALBERT KIM, SUSHIL PATIL, JAY ZHUANG, SAM DUNSTER, AKANKSHA MAHAJAN, ANIRUDH CHELLURI, CHAITANYA DATYE, LUCAS VASCONCELOS SANTANA, NITIN GARG, and OMKAR GAWDE, Meta, USA

As in the general industry, there is a trend in Meta's data centers to migrate data from locally attached SSDs to cloud storage. We extended RocksDB [26], a widely used open-source storage engine designed and built for local SSDs, to leverage disaggregated storage. RocksDB's design, such as its data and log files' access patterns, makes an append-only distributed file system a desirable underlying storage. At Meta, we built disaggregated RocksDB using Tectonic File System [35], which so far had mainly been used for our data warehouse and blob storage stacks. We identified that metadata overhead and tail latencies were Tectonic's major performance gaps and addressed them accordingly. We improved the reliability, performance and other requirements with both general and customized optimizations to the core engine in RocksDB. We also took the time to deeply understand the common challenges presented by applications running on RocksDB and implemented enhancements to address them. This architecture enabled RocksDB to adapt to a more distributed architecture for performance enhancements.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Distributed storage**.

Additional Key Words and Phrases: disaggregated storage, log-structured merge-tree, rocksdb, distributed file system

ACM Reference Format:
Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2, Article 192 (June 2023), 24 pages. https://doi.org/10.1145/3589772

1 INTRODUCTION

RocksDB [26] is an open source storage engine widely used inside and outside of Meta. Historically, it was mainly used for storing data on local SSDs. However, disaggregated storage, where RocksDB accesses underlying storage systems over a network, can help achieve higher efficiency by allowing CPU and storage to be independently provisioned on demand.

Several years ago, few RocksDB applications could afford to place their storage remotely on the network, but network bandwidth increased fast over the past years. For example, ten years

Authors' address: Siying Dong, siying.d@meta.com; Shiva Shankar P, shiva@meta.com; Satadru Pan, satadru@meta.com; Anand Ananthabhotla, anand76@meta.com; Dhanabal Ekambaram, dhanabal@meta.com; Abhinav Sharma, sharma@meta.com; Shobhit Dayal, shobhitdayal@meta.com; Nishant Vinaybhai Parikh, npparikh@meta.com; Yanqin Jin, yanqin@meta.com; Albert Kim, albertkim@meta.com; Sushil Patil, sushilpa@meta.com; Jay Zhuang, zjay@meta.com; Sam Dunster, sdunster@meta.com; Akanksha Mahajan, akankshamahajan@meta.com; Anirudh Chelluri, acvs@meta.com; Chaitanya Datye, cdatye@meta.com; Lucas Vasconcelos Santana, vslucas@meta.com; Nitin Garg, gargn@meta.com; Omkar Gawde, omgawde@meta.com, Meta, 1 Hacker Way, Menlo Park, California, USA, 94025.

2836-6573/2023/6-ART192
https://doi.org/10.1145/3589772

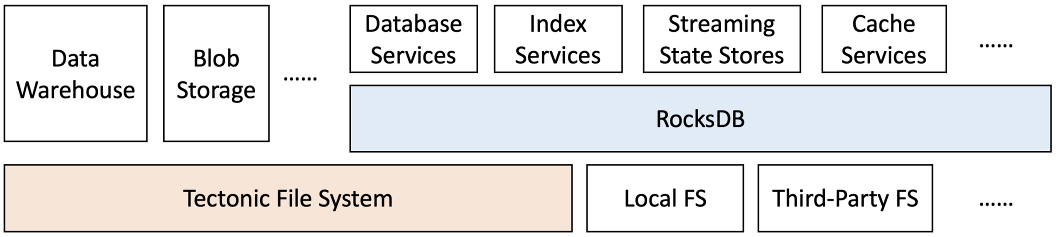

Fig. 1. Different applications are able to run on disaggregated storage with RocksDB.

ago, we were still upgrading hosts from 1Gbps network interface card to 10Gbps, while now they are at least 25Gbps, and often 50Gbps or 100Gbps. On the other hand, most users require similar disk IO bandwidth as ten years ago. This trend makes disaggregated storage attractive for an increasing number of applications. Although the network remains a bottleneck, many use cases are space limited rather than IOPS limited in which case reduced IOPS may be an acceptable trade-off. Furthermore, if the disaggregated storage is implemented as a reliable service, users can take advantage of the offered reliability in various ways, such as improving in-region availability and reducing cross-region data transfers. Because of that, we started to devise a solution for a storage engine running on top of disaggregated storage.

At Meta, we chose the approach of continuing to use RocksDB, but storing data on a distributed file system instead of local SSDs. We kept using RocksDB because its main data structure, Log-Structured Merge-tree (LSM-tree)[34], minimizes space usage, a common bottleneck, and also proved to be efficient for distributed file systems[21]. With LSM-trees, data files are immutable once created. A managed storage service can also provide fault tolerance and RocksDB applications can take advantage of this in various ways, such as reducing in-datacenter failover time. Since RocksDB already serves a variety of services, extending the types of storage it supports naturally allows these services to run on disaggregated storage, as shown in Figure 1; our experience shows that doing so is possible with satisfactory performance. Tectonic File System[35], an append-only distributed file system (DFS) initially built for data warehouse and blob storage, fits the required storage characteristics. We decided to run RocksDB on top of Tectonic.

To run RocksDB applications on a DFS, we faced four major challenges (§3): a) Disaggregate storage setup introduces additional network hops that often results in latency regression compared to a local SSD setup. b) Benefits of a managed storage system (such as quick failover) can only be realized when the data is stored in a fault-tolerant manner. However adding redundancy adds extra overhead in terms of storage footprint and SSD endurance. c) With disaggregated storage, multiple nodes can access files for the same RocksDB directory. When a failover happens, there must be a guarantee that the previous node is no longer able to modify the data anymore. d) The RocksDB library needs to react differently to remote IO behaviors (failures, timeouts, etc) compared to local IO behaviors. Section 4 outlines our solution to these problems.

Running an LSM-tree based database engine on top of an append-only distributed file system is not new, but we believe our experiences and take-aways are unique in several ways. Firstly, we demonstrate that it is possible to use RocksDB in both local and disaggregated mode: Even though disaggregated performance may not match local performance, it is sufficient for a large proportion of RocksDB applications at Meta (§5).

Secondly, we share our lessons from transforming a distributed file system that was built for hard drives and mainly used for data warehouse and blob storage, to serve RocksDB (§8). We believe this transformation is a common pattern and the lessons here can be widely applicable.

Finally, we share some common challenges faced by RocksDB applications and their solutions. We introduce a case study of ZippyDB [15], an established database service inside Meta: how it resolves the challenges of non-RocksDB files, building new Replicas, validating quality, and garbage collection of files (§6). To the best of our knowledge, similar experiences have not been previously shared.

By virtue of RocksDB's wide usage, the lessons we learned are useful for those who might consider transitioning from local to disaggregated storage for their production RocksDB systems.

2 BACKGROUND AND MOTIVATION

This section provides an overview of RocksDB and Tectonic as well as the motivation behind running RocksDB on Tectonic.

2.1 RocksDB

RocksDB is an open source database storage engine widely used by a variety of data services. It is mainly used to store data on SSDs, but some users also have RocksDB store data on hard drives or memory based file systems. A common engine offers reliability, performance and manageability benefits for different use cases.

RocksDB is implemented using Log-Structured Merge-tree (LSM-tree)[34]. Whenever data is written to RocksDB, the written data is first buffered and also written to an on-disk Write Ahead Log (WAL). The data is later flushed to a Sorted String Table (SST) data file on disk. Each SST file stores data in sorted order, divided into blocks. Hot blocks are cached in a memory-based block cache to reduce I/O. Once written, each SST file is immutable.

SST Files are often merged together to create a new set of SST files in a process called compaction. In doing so, deleted and overwritten data is removed, and the new SST file is optimized for read performance and space efficiency. Since keys are in sorted order in each SST file, both compaction reads and writes are sequential. Writes can be buffered using any buffer size, because the SST files output by the compaction process are not used for reads until the compaction completes. Usually, most writes to the storage system are done by the compaction process.

2.2 Why Disaggregated Storage?

Flash based SSDs are widely used by various services in Meta. Initially, SSDs were always used locally where database servers access data on SSD via direct PCI-e. This setup allows database servers to take advantage of SSDs' high throughput and low latency characteristics, and it still fits many applications today. However, this architecture can waste resources and makes service management harder.

With local SSDs, CPU and storage are often unbalanced within a host. [26] Also, some users cannot use all the space due to flash erase budget or read bandwidth limits, while others do use up all the space but also waste some I/O or erase cycles. Furthermore, every service needs to reserve enough buffer and headroom, but they are not used most of the time.

If compute and storage are separated, storage can be allocated from a shared pool on demand. Unused space can be shared so a smaller fraction of total space needs to be reserved for buffer and headroom. CPU can also be provisioned and moved between databases easier, because no data copying is necessary. This fast CPU provisioning allows users to set up their hosts with higher CPU usage. As a result, disaggregated storage enables users to maximize general CPU and storage utilization. Additional benefits of using a distributed file system for disaggregation are described in §2.5.

2.3 Why Still RocksDB?

Rather than building or using another storage engine, we decided to continue using RocksDB, for the following reasons. Firstly, we expect that even with disaggregated storage, many use cases are not bounded by I/O but bounded by space usage. RocksDB's good characteristics of space efficiency[25] would carry over to the disaggregated setup.

Secondly, running the same storage engine for local and disaggregated storage makes migration easier and allows the two modes to co-exist for a long time where it is possible to switch between the two modes.

Thirdly, a simple engine that supports both local and disaggregated storage means future improvements to the engine only need to be implemented in the same place yet benefit both modes. This is particularly beneficial to RocksDB, a popular open-source project which external researchers can also leverage for their research.

Finally, we believe RocksDB's main data structure, LSM-tree, is a good fit for disaggregated storage.

2.4 Why Tectonic File System?

Our first attempt to build a disaggregated solution focused on remote block devices. We used ATA over Ethernet (AoE) or Network Block Device (NBD) to connect remote drives to compute nodes, providing the illusion of standard local access paradigms. This simple architecture makes disaggregated storage possible, but we learned that it cannot fully realize the potential benefits. Efficiency gains were limited due to the lack of support of thin provisioning. Furthermore, handling failures in a storage node or network is difficult and service owners find such a system hard to operate.

To solve the limitations of this solution, we explored several solutions including NFS and distributed reliable block-device. However, NFS turned out to be too complex to exploit with, and with block-device interfaces, it was hard to share data across different hosts. We came to the realization that a generic distributed block-device or file system would be too general and may forgo targeted optimization opportunities, thereby limiting the potential of the combined system. RocksDB's data and log files are generated using sequential writes, and turn immutable after the writes. Any solution that also allows random writes would inevitably introduce extra overhead and complexity. File systems assuming immutable data chunks can achieve better efficiency, which is the exact assumption of the Tectonic File System. Hence we decided that we should start with RocksDB on Tectonic and evolve our approach from there. Tectonic is Meta's exabyte scale distributed file system, designed initially to serve data warehouse and blob storage use cases [35]. It provides a hierarchical File System API similar to Hadoop File System (HDFS), and most cloud providers have compatible storage solutions. In many ways, our design choice is validated by previous systems, such as BigTable [21], HBase [6] and Spanner [22], which are all databases on top of append-only distributed file systems.

However, the way we designed RocksDB to support disaggregated storage allows RocksDB to be layered on top of most distributed file systems and object storage systems, so the value of our work can be widely applied. In principle, we need the following API primitives: a) Data are grouped in files (or called objects or blobs) with user specified names b) The files can be constructed in an append-only way, and data can be read using file offsets. c) Files can be grouped into directories (or buckets). We discussed our learning on how the storage system can offer satisfactory efficiency and usability in §8.

Details about the Tectonic file system can be found in [35]. Here we only emphasize some of its aspects to allow readers to understand the optimizations we present in the following sections. Data stored in Tectonic is durable, using data redundancy, either through erasure coding or replication. Tectonic files are composed of blocks. Chunks of a block, as well as parity chunks if it is erasure coded, are stored on separate storage nodes in different failure domains, so that data can be reconstructed from other chunks if a small number of single storage nodes fail. The metadata layer of Tectonic stores various mappings like directory to list of files, file to its block list and block to list of data chunks. The Tectonic client library which runs inside an application (in this case inside the RocksDB library) is responsible for orchestrating RPC calls to metadata and storage nodes, where data chunks are stored.

2.5 Additional benefits of using Tectonic

As described above, using disaggregated storage offers better resource utilization and independent scaling of CPU and storage. Using Tectonic for disaggregated storage provides the following additional benefits.

Reduces the number of replicas. Applications using RocksDB need to provide high availability guarantees and need to tolerate all kinds of failures. Tectonic ensures data availability in common fault scenarios (host, rack or power domain failures). This added availability allows some applications to reduce the number of the replicas needed. In many cases, applications are able to reduce the number of copies from three to two, or from five to three to maintain at least the same level of availability.

Quick failover. Applications may also rely on Tectonic to quickly recover from single host failures. When a service stores data on local SSDs, a host failure may cause terabytes of data to be under-replicated and the replica need to be rebuilt quickly to avoid permanent data loss. Tectonic's fault-tolerance characteristics allow applications to recover from the failure without data copying(§6.2).

Simplicity of not managing storage. Managing SSD drives is not a trivial job, especially for relatively small teams. Many RocksDB users feel it is desirable to avoid this complexity and thus use a dedicated storage service that hides all the problems. As an example, Tectonic automatically places replicas in different power failure domains, so applications automatically get protection against large scale power fault scenarios.

Storage Sharing. Tectonic allows multiple hosts to access the same set of files. This allows background operations, like backups, compactions and verifications to be offloaded to separate tiers (§7). It also enables fast data cloning.

2.6 When Not To Disaggregate

Even though disaggregation provides benefits, RocksDB applications need to weigh different factors. a) RocksDB on Tectonic still has performance gaps in terms of query latency and throughput (§5). Thus applications have to validate that they can tolerate the lower performance. b) To provide durability, Tectonic requires some redundancy to the data, which typically increases storage usage by more than 20% and is often configured to be as high as 50%. c) Tectonic is a complex system and has many dependencies. It is not a problem for most applications, but some primitive services may require fewer dependencies. For example, the metadata service of Tectonic itself is hard to run on RocksDB on Tectonic.

3 ARCHITECTURE OVERVIEW AND MAIN CHALLENGES

This section presents the high level overview of the RocksDB disaggregated architecture and describes some of the challenges it needs to address.

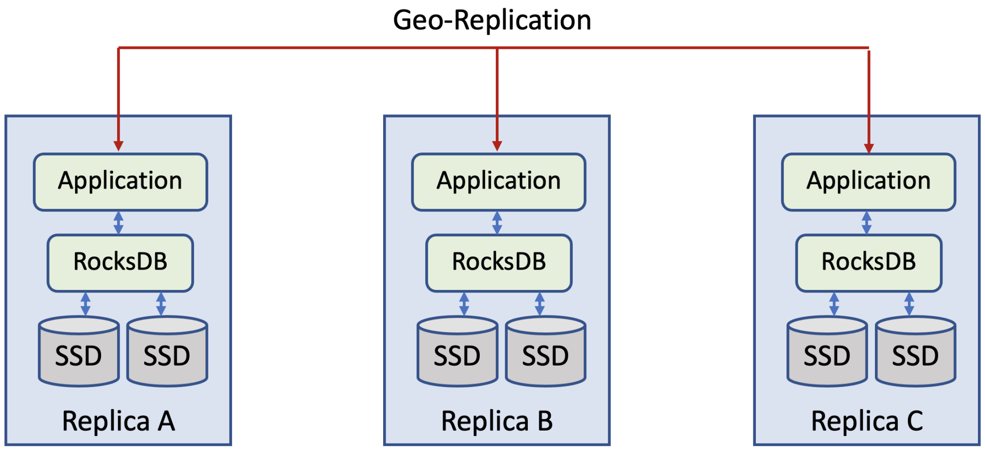

Fig. 2. Typical service using RocksDB on local SSD.

3.1 Architecture

Figure 2 shows a typical data service in Meta before disaggregation, while Figure 3 shows the corresponding architecture after disaggregation. Tectonic clusters are data center local, so applications usually need to keep using their geo-replication logic, with each replica storing data on a Tectonic cluster within the data center. If a compute node fails or load balancing is needed, another compute host will take over and operate on the same data on Tectonic.

The core changes needed to run RocksDB on Tectonic are simple. First, to support Tectonic file reads and writes, a new plugin needs to be developed, which implements RocksDB's storage interface [1] (§6.1) so that RocksDB can work with Tectonic. Second, RocksDB customers need to manage files using a shared Tectonic namespace, as opposed to using an exclusive local file system. Different applications can have their logically isolated namespaces on the cluster. To move from exclusive local file system to Tectonic namespace, some applications keep the exclusiveness by laying out directory structure like 'namespace/.../<host_name>/<db_name>' (§6.2 is an example), while other users use a flat 'namespace/.../<db_name>' structure and rely on other application-specific logic to synchronize accesses to the directory. Either way, Tectonic needs to add support to help users guarantee exclusive access to one directory when needed (§4.3). In addition, a garbage collection functionality may be needed to remove RocksDB directories in Tectonic (§6.4).

Applications configure the Tectonic plugin and pass it to the RocksDB library, together with the base directory information which ensures data is read and written to the desired remote directory. The plugin also reads Tectonic specific configurations like replication schemes. Some configurations, like the replication scheme, may be based on the file type passed down by RocksDB. One RocksDB instance is the single writer for the files in the corresponding RocksDB directory, so the instance can cache data and metadata without consistency issues (§4.1.2, §4.1.3).

While the above basic setup will work as a prototype, we faced several problems to make this solution scalable and efficient enough for production. Those challenges and their corresponding solutions are described in Section 3.2 and Section 4 respectively.

[1]The interface that needs to be implemented: https://github.com/facebook/rocksdb/blob/7.6.fb/include/rocksdb/file_system.h

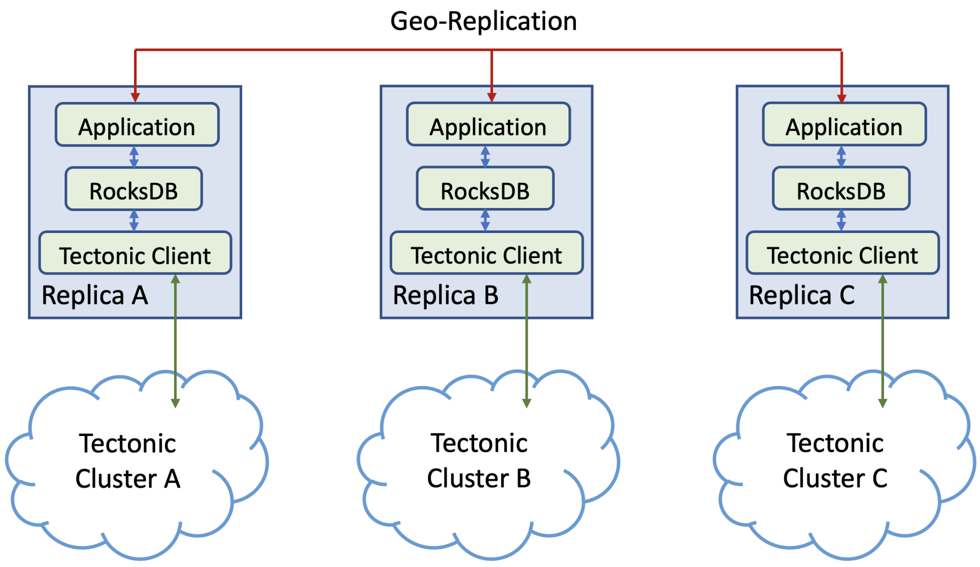

Fig. 3. Typical service using RocksDB on Tectonic.

3.2 Challenges

3.2.1 Performance. Disaggregate storage introduces network hops that result in performance regression compared to local SSD setup. We accept the fact that some applications can never be served by remote storage, but we still hope to expand the range of applications that can be served.

We investigated typical workload patterns and their performance expectations in our environment. We typically expect the 99 percentile latency for small reads and writes (ranging from KBs to hundreds of KBs) which are typically point queries or Get/Put requests from application typically to take less than five milliseconds. For MultiGet queries, as well as iterator or scan operations, we expect the 99 percentile latencies to be on the order of tens of milliseconds. We describe our performance optimizations to meet the above requirements in Section 4.1.

3.2.2 Provide Redundancy With Low Overhead. To realize the benefits of a managed storage system and enable quick failover, DFS data needs to be durable(§2.4). However adding redundancy incurs extra overhead in terms of storage space and SSD endurance. It also adds network overhead in RocksDB applications because of Tectonic's client driven architecture, where the replication or erasure coding of the data happens inside the Tectonic client. We describe how we use different replication schemes for RocksDB SST and WAL files in Section 4.2.

3.2.3 Data Integrity With Multiple Writers. With disaggregated storage, to enable fast failover, multiple compute nodes can access files belonging to the same RocksDB directory. Usually only one node can modify the data. When that node fails, applications would usually pick another node to take over. When that happens, we need to guarantee that the previous node can no longer modify the data. Implementing this guarantee is non-trivial because we cannot usually contact the failed node, yet the node might come back online in the future and attempt further operations. In file system terms, this problem can be reframed as how to ensure that the files and content underneath a single directory can be manipulated by exactly one intended process in a given stretch of time. We describe our solution in Section 4.3.

3.2.4 *Preparing RocksDB for remote IO.*

RocksDB needs to change certain assumptions to support remote IO. Local file systems generally don't have to deal with transient I/O errors, except in special cases such as out-of-space errors. RocksDB has historically treated I/O errors as local file system corruption, in which case it would make the entire DB read-only. RocksDB now needs to handle different errors appropriately, to make sure the DB can be kept operating when possible. A few other examples and their solutions are described in Section 4.4.

4 ADDRESSING THE CHALLENGES

This section describes how we overcame the challenges described above. Most of these changes are local to the RocksDB library. However, to address some of these challenges, we need certain features or performance guarantees from the underlying DFS. We note these requirements throughout this section, and later summarize the learning in §8.

4.1 Performance Optimizations

The RocksDB disaggregated solution needs to bridge the latency gap introduced by remote IOs (§3.2.1). The underlying DFS (Tectonic) needs to provide good tail latency, and the RocksDB layer needs to mask the additional latency from users as much as possible. We describe the DFS tail latency improvements as well as RocksDB specific optimizations. Finally, we evaluate the performance of our disaggregated solution in Section 5.

4.1.1 *Optimizing I/O Tail Latency.*

Overall I/Os could be slowed down by one or two slow storage nodes, potentially causing long tail latencies for both read and write operations. To address the problem, when we suspect one node to be slow, we try to serve the traffic through other nodes. More specifically, we use the following three techniques:

Dynamic Eager Reconstructions. This approach typically issues the first read and then conditionally issues the second read after a delay, using the earliest response. While this is straightforward for replicated data, for erasure coded data, the second read is a reconstruction involving several parallel IO and thus more resource intensive. Furthermore, when latency increases due to cluster health issues, reconstruction reads could lead to further latency deterioration if not done carefully. We address these problems by closely tracking the read latency percentiles for a cluster and continuously adjusting the threshold for issuing the second read based on those percentiles.

Dynamic Append Timeouts. Writes to Tectonic typically involve flushing data to a set of storage nodes. We only wait for acknowledgements from a quorum/subset of storage nodes before acknowledging to the client, while allowing the other writes to be in inflight in the background. This results in better tail latency. However, if there is ongoing maintenance activity in the fleet, more storage nodes may take longer to respond and this technique becomes ineffective. To mitigate this, we use an approach similar to dynamic eager reconstructions described above: if a timeout occurs, we fail the inflight append, take note of the last succeeded size in our metadata, select a new set of storage nodes, record it in our metadata, and retry on a fresh set of storage nodes. This hides the slowness caused by a smaller subset of the fleet. We closely track the append latency percentiles in a cluster, and continuously adjust the timeout for newly issued appends. We plan to make further improvements to eliminate metadata updates in this workflow.

Hedged Quorum Full Block Writes. Unlike appends, where data needs to be closed on a specific set of storage nodes, for large writes, Tectonic creates a full block and hence has the option of selecting any storage node. We split the writing of blocks into two phases: permit acquisition and payload transfer. During the first phase, the client will acquire write permits from a larger pool of storage nodes than required. Storage nodes decide whether to grant permit based on availability of

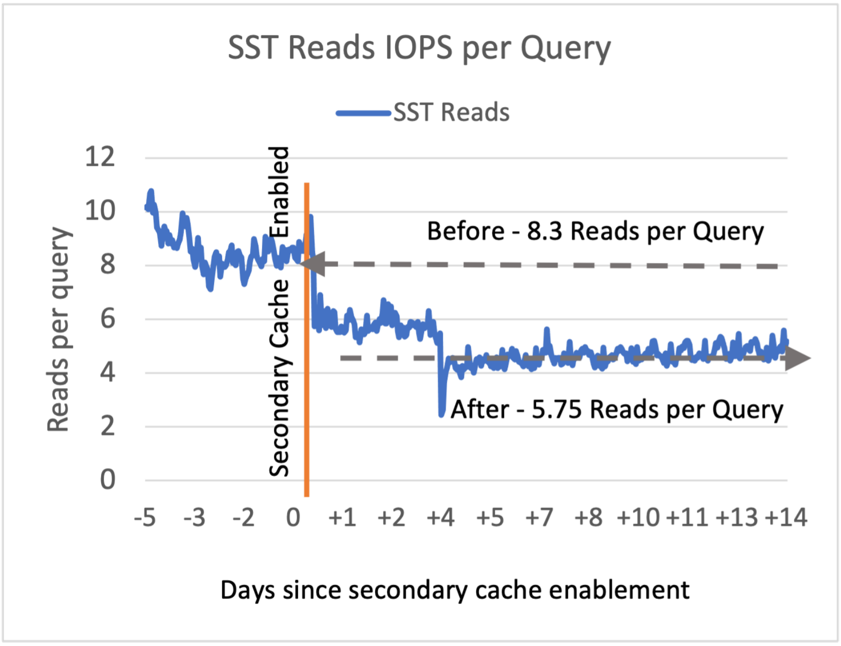

Fig. 4. Graph of SST file reads per ZippyDB read query before/after enabling the secondary cache in ZippyDB.

their own resources (memory, network-bandwidth, CPU). In the second phase, the client selects the subset of nodes that responded earliest for the actual write.

4.1.2 RocksDB Metadata Cache. RocksDB has higher performance requirements for some file metadata operations such as listing directories, checking whether a file exists, and looking up file sizes. These operations are also used by other applications, such as data warehouse applications, but RocksDB uses them much more frequently and has more stringent latency requirements. To address this challenge, we leveraged the insight that the underlying RocksDB directory will be accessed and modified at any point in time by only one process and thus the metadata can be cached proactively. With caching, we effectively bypassed almost all metadata lookup operations. This cache is always consistent, as the RocksDB directory is accessed and modified by only one process at a time, using the IO Fencing approach described in §4.3.

4.1.3 RocksDB Local Flash Cache. To allow some read-heavy applications to use RocksDB on Tectonic, we implemented support for a block cache on non-volatile media, such as a local flash device or NVM/SCM. It can be viewed as an extension of RocksDB's current block cache, which caches data in DRAM. The non-volatile block cache acts as a second tier cache that contains blocks evicted from the volatile cache. Those blocks are then promoted to the volatile cache as they become hotter due to access. The cache is officially referred to in RocksDB as the SecondaryCache. Internally the cache is implemented using Cachelib[14][17]. [2]

Using the secondary cache in our production environment with ZippyDB results in a 50-60% improvement in read IOPS (Figure 4) and a 30-40% improvement in ZippyDB level read latency (Figure 5). The cache configuration used was 20GB of primary cache and 100GB of secondary cache.

[2]This improvement is fully open sourced, including the SecondaryCache plugin that uses cachelib[2]

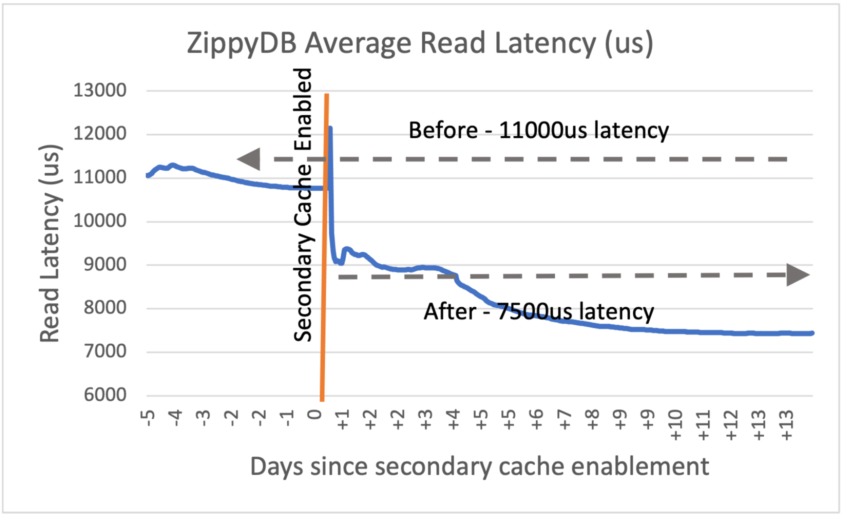

Fig. 5. ZippyDB average read latency before/after enabling secondary cache.

4.1.4 RocksDB IO Handling. Since Tectonic has different IO characteristics than local SSD, we wondered whether RocksDB should change its way of issuing IO. We realized that Tectonic shares some IO characteristics with HDDs, like having higher read/write latency and preferring large writes to smaller ones. By tuning the RocksD knobs built for HDDs, performance on Tectonic was also greatly improved.

RocksDB provides flexibility in adjusting IO in the compaction path because RocksDB's compaction read and write I/Os are sequential and the buffering size is tunable. When running RocksDB on Tectonic, applications usually see satisfactory performance when configuring the compaction read size to 4MB or 8MB and the compaction write buffer size to 64MB or larger. Compaction latency is still likely to increase - fortunately RocksDB's feature of parallel memtable flushes and compactions can help absorb it.

Although tuning existing knobs can solve many performance problems, we still observed long query latencies for operations that require many IOs; Two examples: (1) Users read too many keys with MultiGet(); this problem can be mitigated by using parallel I/O, described in §4.1.5. (2) an iterator reads too many consecutive data blocks. RocksDB uses readahead to reduce IO in the iterator path, which has two modes, a fixed configured readahead or an adaptive one. HDD users usually use a fixed readahead, but this approach generates excessive network bandwidth when used with Tectonic. The adaptive mode starts by reading one block and then continuously doubles the read size until a maximum of 256KB. However this approach warms up too slowly for Tectonic. To improve adaptive readahead for Tectonic, we make RocksDB use an initial readahead size based on historic stats and make the maximum size configurable.

4.1.5 RocksDB Parallel IO. Even if the average latency for a single read from Tectonic is only several hundred microseconds higher than that from a local SSD, this can add up to a high latency gap if one query requires multiple IOs. This often happens when the application reads multiple keys using one MultiGet() operation. Hence, a subsequent improvement we made was to reduce MultiGet latency for IO bound workloads by issuing multiple data block reads in parallel for keys in the same

SST file. [3] While this helps reduce latency in general for MultiGet, it is especially important for the Tectonic use case because of the higher latency of reading from remote storage. The parallelization of IO, however, causes an increase in CPU utilization in the Tectonic client IO path. While the increase when measured using micro-benchmarks was significant (50%), the absolute increase is quite small because the IO component is a small part of overall MultiGet processing.

4.1.6 RocksDB Compaction Tuning. Since Tectonic performance provides lower read and write IOPs, it would limit RocksDB performance for both read QPS and compaction throughput. Furthermore, it takes more space for data redundancy.

We didn't observe a common pattern in which way bottlenecks are shifted, mostly workload dependent. If users feel a need to adjust based on changed bottlenecks after moving to Tectonic, they should be able to adjust RocksDB compaction accordingly. Interestingly, few users feel a need to change compaction strategy when moving to Tectonic, which perhaps indicates that the bottlenecks don't usually change with the move.

One compaction setting that might impact performance when using Tectonic is the target SST file size. On local files, users typically set the SST file size between 32MB to 256MB. Intuitively, if a SST file is too small, there will be too many files which might slow down some operations like opening a DB. Based on our experience, unless the target SST file size is set to be smaller than 64MB, it has minimal impact on performance.

4.2 Redundancy with low overhead

Disaggregated solution need to provide redundancy with low overhead (§3.2.2). Tectonic provides many encoding schemes with different durability and availability guarantees. It allows RocksDB users to decide which encoding settings to use for different file types.

For SST files, we want highly durable encoding with low overhead as they consume the majority of the space and write bandwidth. We chose [12,8] encoding (4 parity blocks for 8 data blocks) for SST files as it only uses 1.5x space and write bandwidth overhead while wide enough to align with our deployment using 6 or 12 failure domains providing high durability SLA guarantees.

For WAL and other log files, we need to support low tail latency for small (sub-block) appends with persistence. We use 5-Way Replica (R5) encoding for WAL and other log files for the following reasons. First, Replica encoding provides better tail latencies with no RS-Encoding overhead for small size writes. Second, unlike RS-Encoding, we don't need write size alignment or padding for replica encoding. Finally, we use R5 instead of R3 or other settings, as R5 is sufficient to meet our availability needs based on host failure probabilities.

For some scenarios with heavy log updates, however, 5x network overhead of R5 is too high. Hence, on the Tectonic side, we added support for striped RS-encoded [12] small (sub-block) appends (with persistence). With striped encoding, we need to gather the entire data of a stripe, or of multiples of stripes before flushing the data. For example, with [12, 8] encoding with 8KB stripes, we split 8 KB data into eight 1KB data buffers, generate four 1KB parity buffers and flush those twelve 1KB buffers to 12 storage nodes. Inside the storage node, each 1KB buffer is appended to the XFS file corresponding to the chunk, which is usually 8MB in size. The stripe size is pre-determined on a per file-type basis. For the occasional non-aligned data that needs to be flushed, it is padded with zeros to achieve alignment, and then encoded and flushed to the storage nodes. Smaller stripes might be less efficient for random reads, as data needs to be assembled from multiple nodes and then decoded. However, log files are almost always read sequentially, so this approach works well.

[3]Improvements mentioned in this section are all in open source RocksDB, except the storage specific implementation of the file async IO.

For a few use cases, we use erasure encoding with higher overhead to reduce tail latencies using hedging techniques (§4.1.1).

4.3 Data Integrity With Multiple Writers

To solve the problem of data getting corrupted by multiple writers, as described in §3.2.3, we built functionality fencing off write operations from previous nodes using a cooperative **IO Fencing** protocol, which is conceptually similar to monotonically increasing token based distributed locks [4].

We mandate that a process trying to own a RocksDB directory must first 'IO Fence' the directory using a token (a variable length byte string), and then use the same token for all subsequent operations on that directory and files under it. The expectation is that the fencing will be successful if the token is lexicographically greater than the previous fencing token used by any other process on that directory. Internally, Tectonic executes a sequence of steps to guarantee fencing. First, it updates the token on the directory in the metadata system, provided it compares favorably. This has the effect of preventing any new file mutation operations (create, rename, delete) by processes with stale tokens from taking effect. Tectonic then iterates over the list of mutable (unsealed) files under the directory, and performs two actions, (a) it updates the token on the file metadata, and (b) seals the tail writable block of the file by connecting to the storage nodes and sealing their corresponding chunks. Action (b) has the effect of forcing any stale writer to coordinate with the metadata system and thereby learn about the staleness and give up retrying. If at any point in this sequence, the fencing fails due to the token being superseded by a higher token, then that instance of fencing is considered to have failed, and the process will not attempt opening RocksDB for mutation.

4.4 Preparing RocksDB for remote calls

The RocksDB library itself needs to adapt to remote IO calls, whose behavior can be different from the local ones (§3.2.4). All changes within RocksDB are open sourced, although storage specific plugins need to be implemented for it to work, as our internal implementation is for Tectonic.

4.4.1 Differential IO Timeout. Remote IO operations can be slower than local ones for multiple reasons. It could be because of transient failures that are recovered from after retrying for seconds. These IOs can succeed after seconds. However, waiting for seconds may be too long for IOs serving user requests. RocksDB applications often require that their queries finish well within a second. If an IO takes seconds to finish, the result may no longer be useful. On the contrary, IOs issued for database internal operations like compaction or flush can tolerate individual slow IO that may takes seconds or even minutes. Handling failures for those IOs requires more complex operations, so it is beneficial for Tectonic to try longer to avoid failures. We concluded that the timeout should be set differently for different type of IOs.

We set timeout to be many seconds for flushes and compactions and we set it to a sub-second time for Get() or iterators. RocksDB added a configurable parameter, *request deadline*, which has RocksDB return a failure if the request takes longer than the specific time. If a user sets this deadline value, the value is passed down to Tectonic accordingly.

4.4.2 Failure Handling in RocksDB. Historically, RocksDB has handled IO errors during important database operations such as WAL write/sync, background flush, and compaction by switching to read-only mode. This was done to ensure consistency of the database. This mirrored the approach of local file systems like Ext4 and XFS.

Writing and reading data to and from the Tectonic file system may incur high latencies, transient write/read failure, short period data availability issues, and even system level failures. Unlike local file systems, however, these can happen more frequently, and also are very likely to be

Workload	Tectonic	Local SSD
Sequential Write	262.4	264.1
Random Write	19.2	26

Table 1. RocksDB Thoughputs for Write Workloads (MB/s).

transient and recoverable. We realized that we could greatly improve availability of the database by classifying errors appropriately and recovering from errors that are transient. We accomplished this by enhancing the return status from the file system API calls to contain not only the error code, but also metadata about the error such as retry-ability, scope (file scope or entire file system), and whether there is permanent data loss. This allows Tectonic to use these insights to determine whether the errors are recoverable. In RocksDB, we focused on restoring the database to a consistent state and resuming writes after a file system write error.

For some write failures, such as those during background flushes or compaction, the operation is retried without any user downtime. In other cases, such as WAL write errors, we need to temporarily stop writes in order to flush memtables to disk and thus guarantee consistency.

Avoiding downtime for user writes for transient Tectonic failure is important, because write failure can be expensive to some services. For instance, a write failure can force ZippyDB, an application that uses RocksDB, to remove the affected replica from the Paxos quorum and then start to build a new replica.

4.4.3 IO Instrumentation. With a more complex storage stack, it becomes important to have greater visibility into the IO path for troubleshooting purposes. Users often rely on IO tracing on the local file system (like strace). The same tools are not available for Tectonic. Hence, we developed more IO instrumentation in RocksDB, so that the same tools can be used for all types of file systems.

4.4.4 Utilities. RocksDB has several command line tools, but previously they only worked with local file systems. Those tools examine the DB state, SST files, and sometimes help users perform offline operations on the DB. Users also need to be able to run those tools if Tectonic is being used. Similarly, it is desirable to run benchmark and stress tests as well as other test tools against Tectonic. We created a general “object registry” in RocksDB which maintains a mapping from object name patterns to factory functions used for creating objects of specific types. If a user gives a URL denoting a Tectonic cluster, like "tfs://cluster1/db1", RocksDB will check the map and use the Tectonic plugin object to communicate with the underlying Tectonic file system. We were able to migrate existing tools and tests to run on Tectonic with only minimal changes to the open-source code, and this can potentially work for other file system plugins built by other users too.

5 PERFORMANCE BENCHMARK

Benchmark Setup. We evaluated the performance of RocksDB with Tectonic using RocksDB’s microbenchmark tool db_bench. The benchmarks from this tool are widely used by RocksDB developers, users, and hardware vendors. We used the same benchmark settings as listed on RocksDB’s official wiki[10], which we use to validate each RocksDB official release. The benchmark uses 20-byte keys and 400-byte values. The values are generated such that the blocks can be compressed to half their size. We use a total of 1 billion keys, making the physical footprint of the database to be about 200 GB. We set the block cache size to be 16 GB, so the majority of queries will require at least one I/O. The block size is set to 8KB, ZSTD compression is used for most data on disk, but newer data is compressed using LZ4. RocksDB options are not tuned for Tectonic

	Threads	Tectonic (Parallel I/O on) QPS	Tectonic (Parallel I/O on) P50 (ms)	Tectonic (Parallel I/O on) P99 (ms)	Tectonic (Parallel I/O off) QPS	Tectonic (Parallel I/O off) P50 (ms)	Tectonic (Parallel I/O off) P99 (ms)	Local SSD QPS	Local SSD P50 (ms)	Local SSD P99 (ms)
Get	64	N/A	N/A	N/A	54.8K	1.1	2.9	334K	0.19	0.42
	32	N/A	N/A	N/A	41K	0.74	1.6	208K	0.15	0.34
	1	N/A	N/A	N/A	1.2K	0.76	2.0	7.2K	0.14	0.26
MultiGet	32	6.4K	4.51	13.23	5.8K	5.35	12.48	39.3K	0.79	1.51
	16	5.5K	2.74	6.50	4.2K	3.77	6.54	23.1K	0.70	1.26
	8	4.1K	1.88	3.67	2.4K	3.39	6.42	12.4K	0.66	1.20
	1	0.6K	1.69	5.49	0.3K	3.42	9.06	1.7K	0.61	1.07
Iterator	32	14.2K	2.12	5.38	11.3K	2.77	6.38	70.2K	0.30	0.61
	16	10.7K	1.33	2.88	6.4K	2.34	5.94	34.4K	0.26	0.57
	8	6.1K	1.27	2.72	3.2K	2.38	6.01	15.7K	0.25	0.55
	1	0.7K	1.29	3.24	0.3K	3.03	9.36	1.8K	0.25	0.54

Table 2. RocksDB Benchmark Results for Read Workloads.

specifically, except for IO related settings. Direct I/O is used for the local SSD case so that it doesn't use the free DRAM as a page cache. A local flash cache is not used.

We use the same CPU and DRAM settings for all the experiments with 3 1.600GHz physical cores and 64 GB DRAM. We run the benchmarks on Tectonic, and compare with local SSD for performance as a reference. As expected, results from using local SSDs show far lower query latency and higher QPS with the same number of reader threads. We only include these results as a reference; the goal was never to match the performance of using local SSD. RocksDB release 7.4 is used. The only non-open-source component used is the storage interface implementation that allows RocksDB to use Tectonic.

Write Throughput. Table 1 shows the results of sequential and random write benchmarks. In RocksDB, when keys are written in sequential order, they are written to different SST files and those files don't need to be compacted. The benchmark shows that RocksDB on Tectonic can achieve the same sequential write throughput as local mode. When keys are written in random order, RocksDB on Tectonic's throughput is lower than local mode by about 25%. Although Tectonic can provide enough throughput for sequential read and write when operating multiple files, single files' processing speed is limited and it causes some bottlenecks. Tuning RocksDB compaction knobs would be able to address these bottlenecks, but we leave the default setting for a better comparison.

Read Performance. We ran three read tests. The first test issues Get() against random keys. In this test, each read typically maps to one underlying I/O. The latency for single query is very close to random read latency from Tectonic and local SSD, respectively, and as expected, Tectonic I/O latency is about 5x the local SSD latency. The MultiGet test queries 10 nearby keys in one MultiGet() query. The distance of keys are set to 32, so that they are from nearby but not adjacent data blocks, as reading adjacent blocks will be combined into one single I/O by RocksDB. The iterator test reads 10 keys from a random location. Both MultiGet and iterator would benefit from parallel I/O (4.1.5). Without parallel IO query latency from Tectonic is about 5x local SSD latency for both tests. The parallel I/O feature can cut the difference to 3x.

For all read tests, we are not able to obtain a higher throughput by using more threads with Tectonic, due to limited concurrent outstanding I/Os allowed by Tectonic's client. This limit is not fundamental and could be removed, but we haven't prioritized the change, because users are currently satisfied with the current throughput; besides removing the limits would introduce a new challenge of flow control to protect Tectonic storage nodes from being overloaded.

6 APPLICATION EXPERIENCES: ZIPPYDB

We have switched several different RockDB applications using local storage to use Tectonic. We learned that although RocksDB on Tectonic makes it much easier for them to adopt disaggregated storage, the applications need to address some common production challenges.

In this section we will explain those common challenges by focusing on our experience with ZippyDB. ZippyDB [15] [40] is a reliable, consistent, highly-available and scalable distributed key-value storage service build natively at Meta. It serves a variety of use cases, ranging from storing metadata for storage systems, counting events for both internal and external purposes, to storing product data. It uses Multi Paxos with geographic replication and RocksDB as the storage engine. The motivations for having ZippyDB use disaggregated storage were efficiency from shared storage fleet, faster failover or drain time, and operational simplicity of managed storage.

6.1 Handling non-RocksDB files

Similar to many other services that use RocksDB, ZippyDB also stores some data on file systems directly, e.g. replication logs for its MultiPaxos protocol. Although Tectonic can support logs directly, we felt it was important to have common resource control tooling for Tectonic across RocksDB and its application, such as ZippyDB. Examples include IO rate limiting for all the background tasks, controlling maximum IO sizes, and maintaining the maximum amount of non-flushed dirty data, etc. To solve this problem, ZippyDB moved all their file operations to use RocksDB's storage interface, for that the file system operations and those operations are sufficient for RocksDB to operate on. The interface is implemented as a common unified storage interface for local file systems and Tectonic, so ZippyDB can run on both local file systems and Tectonic, the same way as RocksDB.

Since replication log (rlog) writes are performed in the context of user IO, when running on Tectonic, the writes have more overhead due to the network communications. With ZippyDB on the local file system, we perform rlog writes to OS page cache before acknowledgment. This tradeoff provides sufficient performance and durability to the client under the assumption that the probability of concurrent kernel crashes (or host failures) in multiple replica hosts is very low. To optimize rlog writes on Tectonic, we kept the same model. We write into a shared memory buffer before acknowledgement, while in the background, we asynchronously write the data from kernel memory to Tectonic. We use small appends with striped erasure encoding (§4.2) for this data to ensure high availability with relatively low overhead. With this approach, user write latency isn't impacted by latency from Tectonic, there is a reduction in the the number of Tectonic IOs, and network amplification is minimized

6.2 Building New Replicas

Similar to all data services, ZippyDB's replicas need to be rebuilt under various conditions. One common condition is a host failure. In a local file system, when a host fails, we need to build a new replica by copying the DB persistent state (all DB files) from a remaining healthy replica, usually from another region. Until this copying is done, the system will be running with reduced replication. As a result, the time needed to rebuild a new replica directly impacts overall system availability because during this time, a failure of another replica will leave insufficient nodes for Paxos (for a 3-replica setup), hurting service availability. With Tectonic, when a replica rebuild is needed to handle host failure, ZippyDB will copy all files used by the failed replica to a new data directory and a new compute node will open the DB from that directory. This file copying is very fast, thanks to Tectonic's fast file copy operation, which updates the new file's metadata to point to the same physical data as the original file. Similar steps are followed to rebuild an in-region replica for load-balancing purposes where the old replica is torn down at the end of successful fast copy. Tectonic allowed ZippyDB to reduce the duration of in-region replica rebuild from about 50 minutes down to within one minute.

Data copying is still sometimes needed for cross-region load balancing and cluster decommission. For these cases, we cannot leverage Tectonic's fast copy operation. This copy could be done in the same way as with the local file system, by using RocksDB's common storage interface. Alternatively, Tectonic also provides a tool for users to copy files across clusters, which guarantees copying is done using the right network and I/O priority.

6.3 Verifying Correctness And Performance

Applications often would like to validate data correctness and performance before fully deploying to Tectonic. To ensure smooth migration without compromising reliability, we decided to plan the Tectonic migration such that we could quickly fallback to the pre-migration state. To achieve this, we built a mirroring file system using RocksDB's storage interface. This storage implementation uses two underlying storage systems. One serves as the source of truth which serves all incoming IO requests, while the other asynchronously mirrors the updates of the first.

Forward mirroring. In this mode, ZippyDB continues to use local flash as source of truth, but maintains an asynchronous copy on Tectonic HDD. This way, ZippyDB and Tectonic get exposed to each other at reduced risk/cost.

Mirror based recovery. When a DB instance using mirroring fails over to a different server, we seed the DB instance with the asynchronous copy from Tectonic. Since the DB state can be inconsistent with the asynchronous copy, we worked on a best efforts recovery mode in RocksDB which can bring the DB up to the most recent consistent state from the asynchronous copy. We start to use the DB while still copying data from Tectonic into the local file system.

Reverse mirroring. In the second step in the migration (after forward mirroring), ZippyDB maintains its source of truth on Tectonic flash, but keeps a secondary asynchronous copy on local flash. We run new workloads in this mode for a few weeks and after verifying reliability and performance, we switch over to the final mode of running directly on Tectonic flash without using local flash.

6.4 Files Garbage Collection

In scenarios like use case deletion or hardware failures, we may leave behind storage for DB instances on flash when the compute node that owned the DB instances is dead/unreachable. With locally attached flash, this state gets wiped when the failed compute node is recycled. With Tectonic, such DB instance state remains without being accessed/deleted. We added a background service to identify and cleanup dead DB instances by comparing live DB instances from ZippyDB against all DB instances on Tectonic.

6.5 Wins With RocksDB on Tectonic

Many ZippyDB servers can't efficiently utilize all space or CPU capacity, for reasons mentioned in §2.2, and we faced various challenges to improve efficiency. RocksDB on Tectonic allowed us to achieve higher utilization. In one ZippyDB cluster using local SSDs, the space utilization is 35%,

Production Metrics	Tectonic	Local SSDs
Space Utilization	75%	35%
Per DB Availability	99.99999%	99.99993%
Per DB Failover Time	49 seconds	51 minutes

Table 3. Comparing ZippyDB on Local SSDs and Tectonic

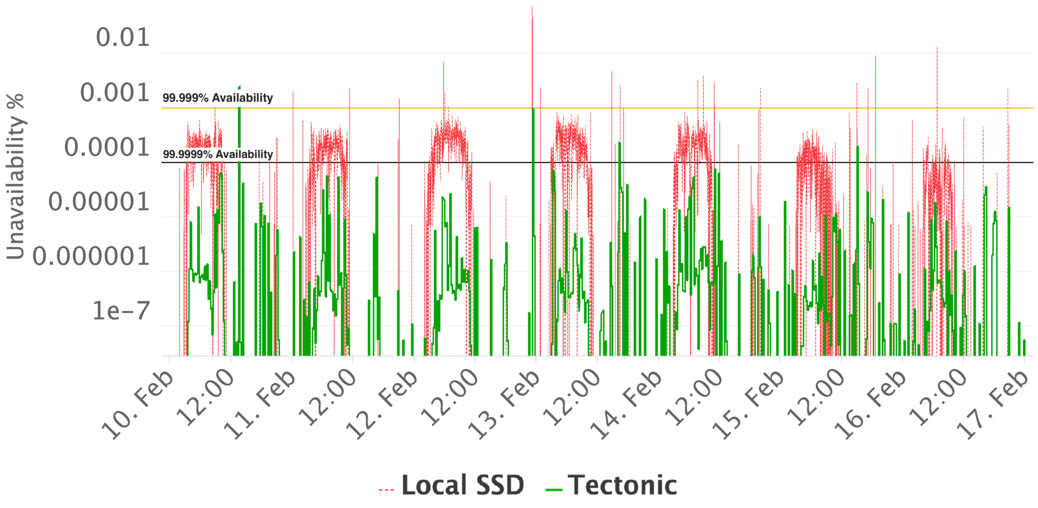

Fig. 6. Comparisons of ZippyDB Availability For a Week.

while a ZippyDB cluster on Tectonic uses 75% of the space. Even considering extra overhead for storing every byte, we still realized significant savings. Storage reliability helped us shorten service unavailability caused by hardware failures, as mentioned in §6.2. In our real time tracking, the availability of ZippyDB on Tectonic is usually higher than 99.99999%, while with ZippyDB on local SSDs, we often see it around 99.99993%, as shown in Figure 6. Table 3 compares some metrics between ZippyDB on Tectonic and local SSDs.

6.6 Performance Analysis

For this performance analysis, we selected four different applications, layered on top of ZippyDB that each store at least hundreds of TBs of data. As shown in Table 4, they serve relatively light read and write workloads, which are friendly to Tectonic. Each uses a replication factor up to three, with one replica stored on Tectonic and the other two on local SSD. There is no preference for which replica is the primary or secondary and they also serve read traffic equally. This setup allows us to compare the performance of the two.

While replicas on Tectonic and SSD use RocksDB instances running on the same type of server, they have slightly different configurations. Local SSD hosts don't use direct I/O and OS page cache has quite a good hit rate. While OS page cache is not used when using Tectonic, a small local flash cache is used, sized about 4x the block cache size, which is a small fraction of the DB size. It typically reduces the number of I/Os needed from Tectonic, compared to the local SSD case. In spite of different configurations, we believe the latency number demonstrates a real world experience.

For each use case, we collected end-to-end latency from ZippyDB clients. They are presented in Table 5-8. The latency includes client computing, network communication, server queuing,

Use Case	QPS Per TB		Bandwidth(MB) Per TB		Workload Description
	Read	Write	Read	Write	
1	14	548	0.4	0.37	Stores signals of different audience for a recommendation system. Reads are mostly done using Iterators.
2	33K	28	294	28.42	Stores estimation and statistics for targeted groups of a recommendation system. Reads are mostly done using Iterators.
3	54	12	0.77	5.55	Stores contents' impressions, clicks and other metrics folded in many ways, which will be consumed by a recommendation system. Reads are done by Iterators and MultiGet() with ratio about 6:1.
4	3K	1.7K	0.17	0.27	Extracted signals from media, serving ML models. Reads are mostly done using MultiGet().

Table 4. Summary of ZippyDB Use Cases Sampled for Analysis.

RocksDB latency and more. We also show the latency of single RocksDB operations. Note that a ZippyDB operation doesn't directly map to a RocksDB operation. ZippyDB's proxy layer often batch multiple client requests and combine them into one RocksDB operation. There is also a small cache in the proxy layer and sometimes repeated queries are served from the cache without sending a request to ZippyDB servers. We also show I/O latency for small reads from Tectonic and local file system. Many reads from the local file system are served from the OS page cache, but we filtered them out by ignoring requests that complete within 30 microseconds.

In all the use cases, end-to-end write latency doesn't show significant change. This is because geo-replication operations dominate the latency, which isn't changed with Tectonic. End-to-end read latency is sometimes higher with Tectonic, although the gap for P99, the metric that ZippyDB users care about most, is usually smaller. Fortunately, these ZippyDB users can accept the regression. Read latency from Tectonic are relatively stable across those services. P50 reads take 1-1.3 ms which is several times higher than local SSD. P99 takes several milliseconds. In two use cases, P99 for Tectonic and local SSD is comparable, while in others there are still big gaps.

In both Use Case 1 (Table 5) and Use Case 2 (Table 6), the average RocksDB read query doesn't need I/O and only takes sub-milliseconds. However, in Use Case 1 ZippyDB users issue MultiScan commands, which require multiple iterators, so P50 latency is impacted. Interestingly, the gap of P99 latency is smaller, likely because of Tectonic's read hedging feature. In Use Case 2, most ZippyDB requests map to single RocksDB request, and the latency is comparable. In Use Case 3 (Table 7) average RocksDB query hits I/O. P99 end-to-end latency shows an increase of 50%.

In all, although the average single Tectonic read latency is often several times higher than that of local SSD, the impacts to end users tend to be smaller than that, and even smaller for performance outliers. Many ZippyDB users see this performance regression as acceptable.

Production Metrics	Tectonic	Local
End-To-End MultiScan Latency (P50, ms)	8.9	5.5
End-To-End MultiScan Latency (P99, ms)	49.6	40.0
End-To-End Write Latency (P50, ms)	103	114
End-To-End Write Latency (P99, ms)	876	766
RocksDB IteratorSeek Latency (P50, ms)	0.44	0.33
RocksDB IteratorSeek Latency (P99, ms)	6.6	3.9
Avg Blocks Reads from SST Files	0.077	0.857
Small Read Latency From FS (P50, us)	1325	388
Small Read Latency From FS (P99, us)	5220	2330

Table 5. Analysis of Performance of ZippyDB Use Case 1

Production Metrics	Tectonic	Local
End-To-End Iterator Latency (P50, ms)	1.5	1.4
End-To-End Iterator Latency (P99, ms)	6.5	7.4
End-To-End Write Latency (P50, ms)	44	36
End-To-End Write Latency (P99, ms)	180	145
RocksDB IteratorSeek Latency (P50, ms)	0.12	0.11
RocksDB IteratorSeek Latency (P99, ms)	2.3	1.1
Avg Blocks Reads from SST Files	0.03	0.06
Small Read Latency From FS (P50, us)	1382	346
Small Read Latency From FS (P99, us)	8943	1993

Table 6. Analysis of Performance of ZippyDB Use Case 2

7 ONGOING WORKS AND CHALLENGES

In this section, we describe some ongoing projects for the RocksDB on Tectonic solution. These are open challenges and we hope sharing them can inspire further innovations in the community.

7.1 Secondary Instances

One benefit of computation-storage disaggregation is efficient usage of resources. Users can flexibly scale up or down the amount of resources, like machines, storage space, network bandwidth, etc. to serve the needs of different workloads or the same workload during different times. For example, when the number of read requests increase during peak time, users can add more machines with shared access to the underlying data, and these machines will be dedicated to serving read requests. After peak time and the read traffic goes down, these machines can be taken away from the service to serve other purposes. Similarly, compaction is CPU and IO intensive. Running compaction on the same hosts as user-facing traffic will likely cause SLA violations. This is the motivation for remote compaction.

To support multiple RocksDB instances accessing shared data, we developed support for "secondary" instances. The primary and secondary instances run in a single-writer, multi-readers mode. Secondary instances replay the log files generated by the primary.

There are open challenges too, such as preventing file deletion while secondary instances are using them, as well as finding and applying the recent updates in secondary instances.

Production Metrics	Tectonic	Local
End-To-End Iterator Latency (P50, ms)	10.4	5.8
End-To-End Iterator Latency (P99, ms)	74	50
End-To-End MultiGet Latency (P50, ms)	3.4	2.1
End-To-End MultiGet Latency (P99, ms)	23	16
End-To-End Write Latency (P50, ms)	34	46
End-To-End Write Latency (P99, ms)	63	72
RocksDB IteratorSeek Latency (P50, ms)	0.89	0.99
RocksDB IteratorSeek Latency (P99, ms)	17.5	7.4
RocksDB MultiGet Latency (P50, ms)	1.6	1.2
RocksDB MultiGet Latency (P99, ms)	21	24
ItertorSeek Avg Blocks Reads from SST Files	0.57	0.71
MultiGet Avg Blocks Reads from SST Files	1.4	2.7
Small Read Latency From FS (P50, us)	1019	325
Small Read Latency From FS (P99, us)	4593	4806

Table 7. Analysis of Performance of ZippyDB Use Case 3

Production Metrics	Tectonic	Local
End-To-End MultiGet Latency (P50, ms)	0.754	5.7
End-To-End MultiGet Latency (P99, ms)	60	132
End-To-End Write Latency (P50, ms)	129	137
End-To-End Write Latency (P99, ms)	235	265
RocksDB MultiGet Latency (P50, ms)	0.098	0.11
RocksDB MultiGet Latency (P99, ms)	3.1	1.5
Avg Blocks Reads from SST Files	0.17	0.25
Small Read Latency From FS (P50, us)	1080	346
Small Read Latency From FS (P99, us)	3154	2432

Table 8. Analysis of Performance of ZippyDB Use Case 4

7.2 Remote Compaction

Compaction could compete for CPU/IO resources with primary service. Remote Compaction offloads the compaction to a dedicated host and Tectonic File System enables us to build a uniform compaction service that can serve compaction jobs from any RocksDB database. It not only improves the performance and reliability of primary service by offloading the background jobs but also enables us to prioritize and manage compactions across different databases, which was not possible with local compaction. We also hope it can help increase throughput for spiked traffic and skewed workload by balancing the compaction jobs across different DBs.

We tested the feature on a ZippyDB use case. Although a ZippyDB host only uses Tectonic within the same region, they often live in different data centers. Since a Tectonic cluster is usually deployed within a data center, we can save cross data center network by having remote compaction hosts co-located with Tectonic. In the test, we were able to save more than 50% of cross data center IO usage and saw the average compaction time is reduced by 20.4%.

Challenges remain on scheduling and prioritizing remote compactions across instances and supporting user plug-in functions.

7.3 Tiered Storage

Flash storage costs a lot more dollars and power per byte compared to hard drive storage. It's beneficial to combine flash (SSD) and hard drive (HDD) storage in a database to optimize cost/power. RocksDB on Tectonic File System (which supports both HDD and SSD) provides us an opportunity to design a tiered storage solution that stores cold SSTables on HDD and hot data on SSD. We implemented a solution where the hot data and cold data can be separated to different SST files and placed to different storage media accordingly. We found it effective to predict data coldness by looking at how long data has been inserted, but challenges remain on more sophisticated prediction.

8 LESSONS AND HINDSIGHTS

Generality of RocksDB. RocksDB is used by a diverse range of applications both within Meta and with external industry partners. We found the RockDB on Tectonic solution generic enough to be suitable for a broad spectrum of services. One example is our data warehouse indexing service, which provides low latency lookup for contents of HIVE tables. During the daily refresh of data warehouse tables, the RocksDB instances of the indexing service are loaded within a small time frame. This access pattern causes a spike in Tectonic file open calls. We reduced the number of SST files to reduce the number of metadata calls. We also disaggregated a RocksDB based FIFO cache service, where the oldest SST files are deleted instead of getting compacted.

Lessons for the Underlying DFS. Tectonic was initially built for hard drives to serve data warehouse and blob storage workloads. While transforming Tectonic to serve SSD based RocksDB workloads, we found the following features important: a) Provide ability for an application process to exclusively write to a directory. We achieve it using IO Fencing, discussed in §4.3. b) Support for configurable replication schemes: we discussed it in §4.2. c) Provide satisfactory performance, especially tail latency. We described performance improvements we made in §3.2.1 and §4.1. d) Make small writes efficient. Usually RocksDB appends data a few kilobytes at a time to WAL files which need to be durable. In §4.2, we described how we handle it.

Work Needed For RocksDB Applications. RocksDB applications need some modification to adapt to RocksDB on Tectonic, including handling non-RocksDB files (§6.1), building new replicas (§6.2), and verifying service quality (§6.3).

9 RELATED WORK

Our exploration of RocksDB on Tectonic builds on previous research, drawing inspiration from the observation that compute-storage disaggregation improves elasticity and cost-efficiency. Experiences in the design and implementation of previous distributed file systems and databases (especially Log-structured-merge tree (LSM) based) provided us with valuable insights.

Compute-storage Disaggregation. Recent research[27, 29, 31, 32] has studied the design of storage, memory and network to enable resource disaggregation to meet the needs of different applications. Researchers have applied the idea of resource disaggregation to different areas including operating system [38], file system [18], blob store [33], and analytics [19, 37], data warehousing [42], etc.

Distributed File Systems. There are different ways of virtualizing disaggregated storage, and different interfaces can be exposed to upper layer programs, depending on the size of the deployment, network topology and requirements of applications[1, 3, 5, 7, 11, 28, 30, 33, 35]. Debating which interface is the best is beyond the scope of this paper.

Distributed file system is a frequently chosen interface to manage disaggregated storage in cluster environments[3, 5, 7, 28, 35]. Some of these file systems are POSIX-compliant[3, 5], but it is not uncommon for a distributed file system to support or optimize for only a subset of file operations (e.g., GFS[28] and HDFS[7] both assume that overwrites are rare). Tectonic shares the same insight and supports file appends only. The append-only semantics of these file systems matches the access patterns of LSM-based storage. The distributed file system community has made great efforts to optimize for LSM-based storage. Hailstorm[18] is a lightweight, rack-level, remote file system designed and optimized for LSM-based key-value databases.

Databases and Disaggregated Storage. A number of distributed database systems [24, 39] have been built for disaggregated storage[6, 20–22, 41]. They have complex architectures which are not directly comparable to RocksDB storage engine which is a library embedded in an application's address space.

BigTable [21] and its open-source counterpart, HBase[6], are distributed storage systems for semi-structured data. BigTable divides data into ranges called tablets, each of which can be hosted by different servers. A dedicated server, i.e. the master, is responsible for tracking the mapping from tablets to servers. The lessons and experiences gained from the design and implementation of BigTable can benefit our ongoing and future efforts with RocksDB on Tectonic file system. Spanner[22] is a distributed database managing data stored in a distributed file system called Colossus[23], the next generation Google File System (GFS)[28]. Amazon Aurora[41] is a cloud-native relational database in which database instances offload redo processing to a multi-tenant, scale-out storage service. Aurora started from the base MySQL[8] codebase, and offloaded backup and redo recovery to the storage cluster for cost amortization. In [41], Aurora was compared with MySQL on EBS[1]. PolarDB[20] uses RDMA[9] to connect disaggregated storage with computation nodes.

LSM-tree and Disaggregated Storage. Log-structured merge-tree (LSM-tree) [34] is a widely used data structure for databases on disaggregated storage. BigTable, HBase and PolarDB all store data as SST files. This is mainly because SST files, once written, become immutable so that they can be readily fan-out read by multiple computation nodes, even if they are the input of ongoing compaction. This means they are a good match for the distributed file systems that support or optimize for only file-appends.

The idea of offloading compaction to disaggregated storage has been explored by a few other systems. [16] proposed a solution to integrate compaction offloading with HBase. RocksDB-cloud[36] storage engine by Rockset is a variation of RocksDB, and supports offloading compaction to remote stateless servers[13]. RocksDB-cloud caches SST and WAL files locally and periodically syncs to cloud side.

10 CONCLUSION

People would see better performance when running a database on a directly attached SSD, but it may be more efficient and easier to store data on disaggregated storage instead. It is convenient to have one database storage engine to serve both, and our experience demonstrated that with some targeted improvements, it is possible, and we managed to accomplish this by expanding RocksDB to support the Tectonic File System, and saw the combination brings expected efficiency gains. Lessons were also learned for data services to use RocksDB on Tectonics in production. Running RocksDB on Disaggregated Storage also allows RocksDB to evolve to a more distributed architecture, and we are further exploring this direction.

11 ACKNOWLEDGMENTS

We are grateful to the reviewers of SIGMOD for their valuable comments and suggestions, which have improved the quality of this paper. We also thank Prof. Michael Stumm for providing insightful feedback and edits, as well as Mark Callaghan for his helpful suggestions.

The success of the RocksDB on Tectonic project, as well as its application to ZippyDB, would not have been possible without the invaluable contributions of members of these teams and many other partner teams. We would like to acknowledge the contributions of Dan Meredith, David Felty, Federico Piccinini, Giang Nguyen, Guna Lakshminarayanan, JR Tipton, Jennifer Chan, Joe Hirschfeld, Jorge Guerra, Junjie Wu, Junqing Deng, Kapil Kataria, Karthik Krishnamurthy, Lachlan Mulcahy, Lujin Luo, Madhu Anantha, Michael C Huang, Michael Meng, Mikhail Antonov, Murali Vilayannur, Naveen Ganapathi Subramanian, Nicholas Ormrod, Peter Dillinger, Pratap Singh, Ramkumar Vadivelu, Sachin Lakhanpal, Sai Bathina, Sankalp Kohli, Sarah Wang, Shrikanth Shankar, Shubham Singhal, Sorin Stoiana, Tejasvi Aswathanarayana, Tyler Heucke, Victoria Tsai, Xiaoyu Wang, Yunqiao Zhang, and Zhichao Cao, as well as many others who have played a vital role in this endeavor.

REFERENCES

[1] [n. d.]. Amazon EBS. https://aws.amazon.com/ebs/.

[2] [n. d.]. Cachelib Repo. https://github.com/facebook/CacheLib.

[3] [n. d.]. Ceph File system. https://docs.ceph.com/en/pacific/cephfs/index.html.

[4] [n. d.]. Distributed locks with Redis. https://redis.io/topics/distlock.

[5] [n. d.]. GlusterFS. https://www.gluster.org/.

[6] [n. d.]. Hbase. https://hbase.apache.org/.

[7] [n. d.]. HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[8] [n. d.]. MySQL. https://www.mysql.com/.

[9] [n. d.]. RDMA. http://www.rdmaconsortium.org/.

[10] [n. d.]. RocksDB Benchmark Wiki Page. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[11] 2009. Rados. https://ceph.io/en/news/blog/2009/the-rados-distributed-object-store/.

[12] 2015. Introduction to HDFS Erasure Coding in Apache Hadoop. https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.

[13] 2020. RocksDB-Cloud remote compaction. https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/.

[14] 2021. Cachelib. https://engineering.fb.com/2021/09/02/open-source/cachelib/.

[15] 2021. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb/.

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

[17] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[18] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–316.

[19] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. https://doi.org/10.1145/3190508.3190532

[20] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} Meets Computational Storage: Efficiently Support Analytical Workloads in {Cloud-Native} Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 29–41.

[21] 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 Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[22] 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 Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[23] Jeffrey Dean. 2010. Evolution and future directions of large-scale storage and computation systems at Google. (2010).

[24] David DeWitt and Jim Gray. 1992. Parallel database systems: The future of high performance database systems. *Commun. ACM* 35, 6 (1992), 85–98.

[25] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[26] 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 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

[27] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

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

[29] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. 2018. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Transactions on Storage (TOS)* 14, 4 (2018), 1–18.

[30] Dave Hitz, James Lau, and Michael A Malcolm. 1994. File System Design for an NFS File Server Appliance.. In *USENIX winter*, Vol. 94. 10–5555.

[31] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–15.

[32] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated {Rack-Scale} Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 17–33.

[33] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 1–15.

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

[35] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[36] Rockset. 2018. RocksDBCloud. https://rockset.com/blog/rocksdb-cloud-enabling-the-next-generation-of-cloud-native-databases/.

[37] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[38] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[39] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[40] Amy Tai, Andrew Kryczka, Shobhit Kanaujia, Chris Petersen, Mikhail Antonov, Muhammad Waliji, Kyle Jamieson, Michael J Freedman, and Asaf Cidon. 2018. Live recovery of bit corruptions in datacenter storage systems. *arXiv preprint arXiv:1805.02790* (2018).

[41] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

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

Received November 2022; revised February 2023; accepted March 2023