Leveraging NVMe SSDs for Building a Fast, Cost-effective, LSM-tree-based KV Store

CHENG LI, HAO CHEN, and CHAOYI RUAN, University of Science and Technology of China, China
XIAOSONG MA, Qatar Computing Research Institute, HBKU, Qatar
YINLONG XU, University of Science and Technology of China, China and Anhui Province Key Laboratory of High Performance Computing, China

Key-value (KV) stores support many crucial applications and services. They perform fast in-memory processing but are still often limited by I/O performance. The recent emergence of high-speed commodity non-volatile memory express solid-state drives (NVMe SSDs) has propelled new KV system designs that take advantage of their ultra-low latency and high bandwidth. Meanwhile, to switch to entirely new data layouts and scale up entire databases to high-end SSDs requires considerable investment.

As a compromise, we propose SpanDB, an LSM-tree-based KV store that adapts the popular RocksDB system to utilize *selective deployment of high-speed SSDs*. SpanDB allows users to host the bulk of their data on cheaper and larger SSDs (and even hard disc drives with certain workloads), while relocating write-ahead logs (WAL) and the top levels of the LSM-tree to a much smaller and faster NVMe SSD. To better utilize this fast disk, SpanDB provides high-speed, parallel WAL writes via SPDK, and enables asynchronous request processing to mitigate inter-thread synchronization overhead and work efficiently with polling-based I/O. To ease the live data migration between fast and slow disks, we introduce TopFS, a stripped-down file system providing familiar file interface wrappers on top of SPDK I/O. Our evaluation shows that SpanDB simultaneously improves RocksDB's throughput by up to 8.8× and reduces its latency by 9.5–58.3%. Compared with KVell, a system designed for high-end SSDs, SpanDB achieves 96–140% of its throughput, with a 2.3–21.6× lower latency, at a cheaper storage configuration.

CCS Concepts: • **Information systems → Key-value stores;**

Additional Key Words and Phrases: Key-value store, SSD, write-ahead log

ACM Reference format:
Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. 2021. Leveraging NVMe SSDs for Building a Fast, Cost-effective, LSM-tree-based KV Store. *ACM Trans. Storage* 17, 4, Article 27 (October 2021), 29 pages.
https://doi.org/10.1145/3480963

Cheng Li and Hao Chen contributed equally to this research.

This work was supported in part by National Nature Science Foundation of China (Grants No. 61832011, No. 61802358, and No. 61772486), and USTC Research Funds of the Double First-Class Initiative (Grant No. YD2150002006).

Authors' addresses: C. Li, H. Chen, and C. Ruan, University of Science and Technology of China, NHPCC Room 503, East Campus, JinZhai Road 96, Hefei, Anhui, P.R China; emails: chengli7@ustc.edu.cn, {cighao, rcy}@mail.ustc.edu.cn; X. Ma, Qatar Computing Research Institute, P.O. Box: 34110, QCRI, HBKU Research Complex, Education City, Doha, Qatar; email: xma@hbku.edu.qa; Y. Xu (corresponding author), University of Science and Technology of China, Hefei, China, Anhui Province Key Laboratory of High Performance Computing, Hefei, China; email: ylxu@ustc.edu.cn.

1553-3077/2021/10-ART27 $15.00
https://doi.org/10.1145/3480963

1 INTRODUCTION

Persistent **key-value (KV)** stores are widely used today to store data in various formats/sizes for a wide range of applications, such as online shopping [37], social networks [17], metadata management [11], and so on. The write-friendly **log-structured merge tree (LSM-tree)** is widely adopted as the underlying storage engine by mainstream KV stores, such as RocksDB [1], LevelDB [33], Cassandra [28], and X-Engine [37]. It remains appealing as production KV environments are often found write-intensive [14, 19, 30, 37, 51], especially due to aggressive memory caching [16, 55, 58].

The recent availability of fast, commodity **non-volatile memory express solid-state drives (NVMe SSDs)** can bring dramatic KV performance boosts, as demonstrated by recent systems, such as KVell [51] and KVSSD [45]. By either discarding the LSM-tree data structures designed for hard disks or offloading KV data management to specialized hardware, these systems provide high throughput and scalability, with the entire dataset hosted on high-end devices.

This work, instead, aims at *adapting mainstream LSM-tree-based KV design* to fast NVMe SSDs and I/O interfaces, with a special focus on *cost-effective deployment in production environments*. This is motivated by our study (Section 2) showing that current LSM-tree-based KV stores fail to exploit the full potential of NVMe SSDs. For example, deploying RocksDB atop Optane P4800X only improves throughput by 23.58% compared with a SATA SSD for a 50%-write workload. In particular, the I/O path of common KV store designs severely under-utilizes ultra-low latency NVMe SSDs, especially for small writes. For instance, going through ext4 brings a latency 6.8–12.4× higher than via the Intel SPDK interfaces [42].

This hurts particularly **write-ahead-logging (WAL)** [57], crucial for data durability and transaction atomicity, which sits on the critical path of writes and is bottleneck-prone [36]. Second, existing KV request processing assumes slow devices, with workflow designs embedding high software overhead or wasting CPU cycles if switched to fast, polling-based I/O.

In addition, new NVMe interfaces come with access constraints (such as requiring binding the entire device for SPDK access, or recommending pinning threads to cores). This complicates KV design to utilize high-end SSDs for different types of KV I/O, and renders current common practices, such as synchronous request processing less efficient.

Finally, top-of-the-line SSDs like the Optane are costly for large-scale deployment. As large, write-intensive KV stores inevitably possess large fractions of cold data, to host all data on these relatively small and expensive devices is likely beyond the budget of users or cloud database service providers.

Targeting these challenges, we propose SpanDB, an LSM-tree-based KV system that adopts *partial deployment of high-end NVMe SSDs*. It is based on a comprehensive analysis of bottlenecks/challenges in porting a popular KV store to use SPDK I/O (Section 2) and contains the following innovations:

- It scales up the processing of all writes and reads of more recent data by incorporating a relatively small yet fast ***speed disk* (*SD*)**, while scaling out data storage on one or more larger and cheaper ***capacity disks* (*CD*)**.
- It enables fast, parallel accesses via SPDK to better utilize the SD, bypassing the Linux I/O stack and allowing high-speed WAL writing in particular. (To the best of our knowledge, this is the first work studying SPDK support for KV stores.)
- It implements a lightweight file system (TopFS) tailored for SPDK interface, which manages KV data on raw NVMe SSDs and facilitates easy data migration between CD and SD.
- It devises an asynchronous request processing pipeline suitable for polling-based I/O, which removes unnecessary synchronization, aggressively overlaps I/O wait with in-memory processing, and adaptively coordinates foreground/background I/O.

- It strategically and adaptively partitions data according to the actual KV workload, actively involving the CD for its I/O resources, especially bandwidth, to help share the write amplification common in contemporary KV systems.

We implement SpanDB as an extension to Facebook's RocksDB [1], a leading KV store deployed in many production systems [2, 7]. SpanDB re-designs RocksDB's KV request processing, storage management, and group WAL writing to utilize fast SPDK interfaces, and retains RocksDB's data structures and algorithms, such as LSM-tree organization, background I/O mechanism, and transaction support features. Therefore, its design stays complementary to many other RocksDB optimizations [14, 15, 22, 53, 63]. Existing RocksDB databases can be migrated to SpanDB when an SD is added.

Our evaluation using YCSB and LinkBench shows that SpanDB significantly outperforms RocksDB in all categories (throughput, average latency, and tail latency) in all test cases, especially write-intensive ones. Against KVell, a recent system designed to leverage high-end SSDs, SpanDB delivers higher throughput in most cases (at a fraction of KVell's latency), without sacrificing transaction support.

2 BACKGROUND AND MOTIVATION

2.1 LSM-tree-based KV Stores

Overall architecture. LSM-tree-based KV stores organize on-disk data in *levels*, denoted as L_0, $L_1, \ldots, L_k$, with capacity generally growing by 10× between adjacent levels except L_0. KV pairs are stored in ***Static Sorted Tables (SSTs)***, each an immutable file. To avoid data loss/inconsistency, a sequential WAL file, often sized around tens of GBs, is maintained on persistent storage. Updates are logged there before being made visible, upon the completion of a write operation/transaction. To optimize overall performance, these KV stores employ additional in-memory data structures. In-memory updates are made in *MemTables*, one active while the rest are immutable. The active MemTable accommodates updates and becomes immutable when full, whereupon one or more immutable MemTables need to be flushed to make space for a new active one. As loading data from persistent devices is often much slower than from memory, two kinds of caches are employed to keep most recent visited KV data in memory: the page cache (the default setting when using the conventional Linux filesystem and cannot be disabled) and the KV store's internal cache (configurable at user level).

Foreground write/read. Upon the arrival of a write operation/transaction, to avoid data loss and/or inconsistency, its updates (along with associated metadata) must be first appended to a WAL file on persistent storage. Then, the corresponding changes made to the database can be applied to the active MemTable for subsequent visits. Given that failures are common on typical KV platforms today [23, 25, 31, 68], WAL [57] remains an integral part of customer facing databases and sits on the critical path of processing write requests.

User reads may generate random accesses at multiple tree levels, until the target key hits at a certain level or misses all the way. Though production KV systems today greatly improve average read performance through aggressive in-memory caching [16, 30, 55, 58, 66], disk I/O cannot be avoided, especially with larger databases or lower access locality. The inevitable accesses to slow storage contribute heavily to tail latency and may affect the overall performance.

Background flush and compaction. These include (1) *flush*, where an immutable MemTable is written to an L_0 SST file (often making L_0 temporarily larger than L_1), and (2) *compaction*, where SST files selected from a level L_i are read and merged with SSTs of overlapping key ranges at level L_{i+1}, with invalid KV pairs removed. The former is triggered by the number of immutable MemTables reaching a limit, and the latter by a level becoming full. Both operations create large,

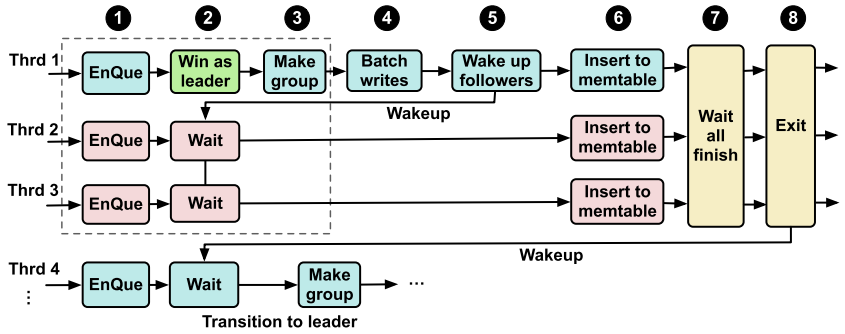

Fig. 1. RocksDB group WAL write workflow.

sequential I/O, whose impact on foreground request processing manifests in I/O contention and write stalls (when user writes need to wait for flushes to empty MemTable space).

Foreground-background coordination. RocksDB and LevelDB control the rate of background I/O through a user-configurable number of flush/compaction threads. They are activated when there are background I/O tasks, sleeping otherwise. Researchers have noted the performance impact of background thread settings and proposed related optimizations [14]. However, existing solutions still retain the background thread design, assuming slow I/O and interrupt-based synchronization, which does not work well with new, polling-based I/O interfaces (to be discussed below).

2.2 Group WAL Writes

The current common practice in writing WAL is *group logging*, which batches multiple write requests for one log data write [32, 35, 59, 84]. This technique is widely adopted by mainstream databases today, including MySQL [6], MariaDB [3], RocksDB [1], LevelDB [33], and Cassandra [49]. Beside fault tolerance, group logging also offers better write performance on slow storage devices (where random accesses tend to be even slower), by promoting sequential writes.

Figure 1 illustrates the RocksDB/LevelDB group logging workflow. The WAL write process is sequential: at any time, at most one group is writing to the log. When there is an ongoing write, worker threads handling write requests form a new group by joining a shared queue, with the first en-queued thread designated the group's *leader* (❶–❸). The leader (Thread 1 in this case) collects log entries from peers, until notified to proceed by the leader of the previous group, who just finished writing. This closes the door for the current group and subsequently arriving threads will start a new one.

The leader writes log entries to persistent storage in a single synchronous I/O step (using `fsync`/`fdatasync`, ❹). The leader then wakes up group members to actuate updates in MemTables, making such writes visible to subsequent requests (❺–❻). It finalizes the group commit by advancing the *last visible sequence* to the latest sequence number among its entries (❼), disbanding the group (❽), and passing the green light to the next leader (Thread 4).

2.3 High-Performance SSDs Interfaces and Their Implications

Recent high-end commodity SSDs, such as Intel Optane [41], Toshiba XL-Flash [69], and Samsung Z-SSD [65], offer low latency and high throughput [72]. Recognizing that the Linux kernel I/O stack overhead is no longer negligible in total I/O latency [50, 76], Intel developed **Storage Performance Development Kit (SPDK)** [42, 76], a set of user-space libraries/tools for accessing high-speed NVMe devices. SPDK moves drivers into user space, avoiding system calls and

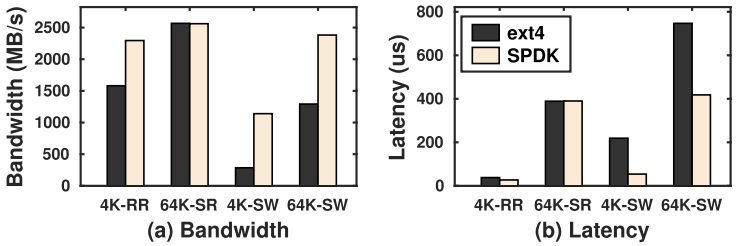

Fig. 2. Optane P4800X performance via ext4 and SPDK at different request sizes by 16 threads. "RR," "SR," and "SW" stand for random read, sequential read, and sequential write, respectively.

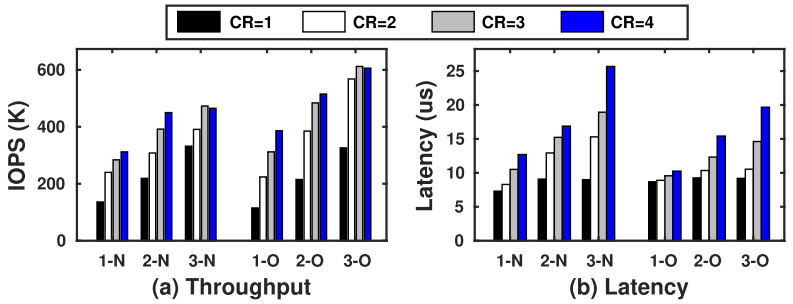

Fig. 3. Concurrency evaluation w. 4 KB sequential writes. N and O stand for the Intel P4610 (N) and Optane (O) SSD, respectively. We configure the number of concurrent write threads from 1 to 3, e.g., "3-N" having three threads writing to SSD N, and make each thread issue 1 to 4 concurrent requests, e.g., "CR=2" having each thread issuing up to two requests.

enabling zero-copy access. It polls hardware for completion instead of using interrupts and avoids locks in the I/O path. Here, we summarize SPDK performance behavior and policy restrictions found relevant to KV stores in this work.

SPDK overall performance. We benchmarked two modern NVMe SSDs, Intel Optane P4800X and P4610. Figure 2 gives Optane results for request type/size combinations, simulating typical LSM-tree-based KV I/O as described earlier (P4610 results show similar trends). We use `write` calls for ext4 (each followed with `fdatasync`), and the SPDK build-in perf tool (`spdk_nvme_perf`) for SPDK.

For large sequential reads, going through a file system (as done by current KV stores) actually matches SPDK results. The 4 KB sequential writes (WAL-style) via ext4, meanwhile, achieve a small fraction of the hardware potential, with latency 4.05× higher than SPDK (IOPS accordingly lower). The 4 KB random read and 64 KB sequential write tests see ext4-SPDK gaps between these extremes. Such results highlight that SPDK may bring significant improvement to KV I/O, especially for logging and write-intensive workloads.

SPDK concurrency. To assess SPDK's capability of serving concurrent sequential writes, we profile individual SPDK requests, and find the bulk of the 7–8 μs single-thread latency indeed occupied by busy-wait, which grows with more threads concurrently writing, due to slower I/O under contention.

We then devise a pipeline scheme, where each thread manages multiple concurrent SPDK requests. It allows to "steal" I/O wait time to issue new requests and check the completion status of outstanding ones (each taking under 1μs). Figure 3 gives latency and throughput results on the Intel P4610 (N) and Intel Optane (O) SSDs. We vary the number of threads ("3-N" having 3 threads

writing to SSD N) and the upper limit for concurrent requests per thread ("CR=2" having each thread issuing up to two requests). NVMe SSDs do offer parallelism beyond utilized by the current RocksDB/LevelDB single-WAL-writer design. In particular, **Optane (O)** shows higher concurrency than P4610 (N), with slower latency and faster throughput growth with more writers. However, even with O, going beyond three concurrent writers does not provide higher SPDK IOPS: Using three loggers each with CR = 3 appears to offer peak WAL speed, which we denote as 3L3R. N, however, saturates at 2L4R.

SPDK access restrictions. The performance benefit of fast SPDK-enabled access to high-end NVMe SSDs comes with strings attached: once an SSD is bound to SPDK, it cannot be accessed via the Linux I/O stack. This simplifies inter-workload isolation associated with user-level accesses but also disables partial deployment of file systems to an SSD accessed via SPDK. In addition, users are recommended to bind SPDK-accessing threads to specific cores [27]. We verified that not doing so brings significant I/O performance loss. This, plus the polling-based I/O mode, renders the common practice of using background flush/compaction threads unsuitable for SPDK accesses: unbound threads suffer slow I/O, while bound threads cannot easily yield core resources when idle.

WAL inefficiencies via SPDK. With high-end NVMe SSDs and faster I/O interfaces, the batch write time (step ❹ in Figure 1) in the above group WAL workflow is dramatically reduced. Meanwhile, batching writes still helps by consolidating small requests. Consequently, the software overhead caused by the synchronous group logging rises to render most of the threads idling on different types of wait (steps ❶–❸ and ❼). For example, we measured that RocksDB spends, on average, 68.1% of write request processing time on these four steps on a SATA SSD accessed via ext4, which grows to 81.0% on Optane via SPDK.

3 SPANDB OVERVIEW

Design rationale. We propose SpanDB, a high performance, cost-effective LSM-tree-based KV store using heterogeneous storage devices. SpanDB advocates the use of a small, fast, and often more expensive NVMe SSD as a ***speed disk (SD)***, while deploying larger, slower, and cheaper SSD (or arrays of such devices) as the ***capacity disk (CD)***. SpanDB uses the SD for two purposes: (1) WAL writes and (2) storing the top levels of the RocksDB LSM-tree.

As WAL processing cost is user-visible and directly impacts latency, we reserve enough resources (cores and concurrent SPDK requests, plus sufficient SPDK queues), to maximize its performance. Meanwhile, WAL data only needs to be maintained till the corresponding flush operation and typically require GBs of space, while today's "small" high-end SSDs, such as Optane, offer over 300 GB. This motivates SpanDB to move the top levels of the RocksDB LSM-tree to the SD. This also offloads a significant amount of flush/compaction traffic from the CD, where the bulk of colder data resides.

SpanDB architecture. Figure 4 gives a high-level view of SpanDB storage structure. Within DRAM, it retains the RocksDB MemTable design, with one mutable and multiple immutable MemTables. Note that SpanDB introduces no modifications to RocksDB's KV data structures, algorithms, or operation semantics. The major difference here lies in its asynchronous processing model (Section 4.1), to reduce synchronization overhead and adaptively schedule tasks.

On-disk data are distributed across the CD and SD, two physical storage partitions. The SD is further partitioned, with a small *WAL area* and the rest of its space used as a *data area*. SpanDB manages the SD as a raw device via SPDK and redesigns the RocksDB group WAL writes (Section 4.2), for fast, parallel logging, improving logging bandwidth by 10×. The data area manages raw SSD pages to host the top levels of the LSM-tree (Section 4.3). To minimize changes to RocksDB, here SpanDB implements *TopFS*, a lightweight file system (including its own cache), which allows easy and dynamic-level relocation. The CD partition, meanwhile, stores the "tree

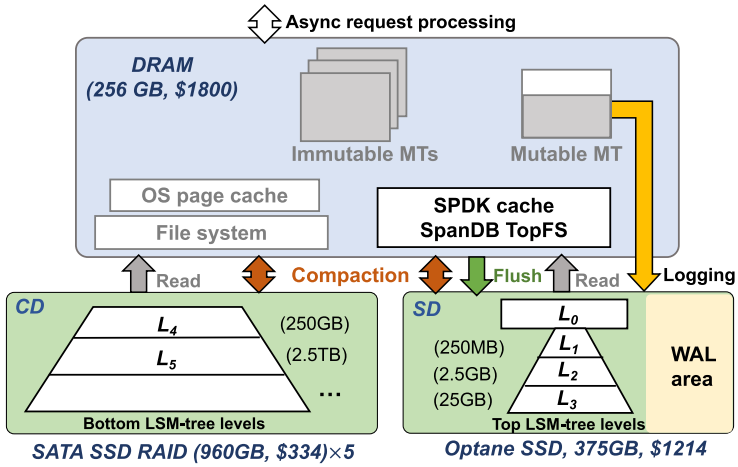

Fig. 4. SpanDB storage overview. The dimmed (grey) components reuse RocksDB implementation. SD, CD, and MT stand for *speed disk*, *capacity disk*, and *MemTable*, respectively.

stump," often containing the colder majority of data. Its management remains unchanged from RocksDB, accessed via a file system and assisted by the OS page cache.

Figure 4 also depicts the different types of SpanDB I/O traffic. While the SD WAL area is dedicated to logging, its data area receives all flush operations, which write entire MemTables to L_0 SST files. In addition, both SD data area and CD accommodate user reads and compaction reads/writes, where SpanDB performs additional optimization to enable simultaneous compaction on both partitions and automatically coordinate foreground/background tasks. Finally, SpanDB is capable of dynamic tree level placement based on real-time bandwidth monitoring of both partitions.

Sources of performance benefits. SpanDB improves LSM-tree-based KV store design in multiple ways:

- By adopting a small yet fast SD accessed via SPDK, it speeds up WAL by fast, parallel WAL writes.
- By using the SD also for data storage, it optimizes the bandwidth utilization of such fast SSDs.
- By enabling workload-adaptive SD-CD data distribution, it actively aggregates I/O resources available across devices (rather than using CD only as an "overflow layer").
- Though mainly optimizing writes, by offloading I/O to the SD, it reduces tail latency with read-intensive workloads.
- By trimming synchronization and actively balancing foreground/background I/O demands, it exploits fast polling I/O while saving CPU resources.

4 DESIGN AND IMPLEMENTATION

4.1 Asynchronous Request Processing

KV stores like RocksDB and LevelDB (plus many new systems based on them [13–15, 18, 22, 53, 56, 63, 81]) use embedded DB processing, where all foreground threads assume the "client" role, each synchronously processing one KV request at a time. With such processing often being I/O-bound (especially with WAL writes), users typically obtain higher overall throughput (requests per second) by *thread-overprovisioning*, having more client threads than cores. With fast NVMe SSDs and interfaces such as SPDK, as discussed in Section 2.3, thread synchronization (such as sleep and

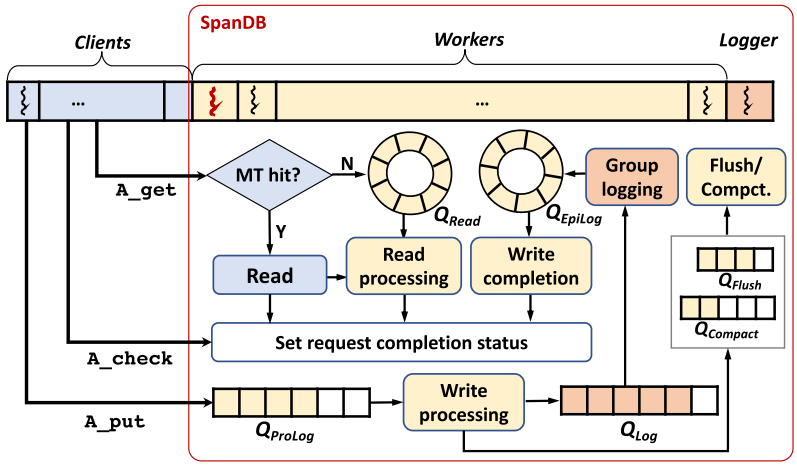

Fig. 5. Asynchronous request processing workflow of SpanDB.

wakeup) could easily take longer than an I/O request. In this case, thread overprovisioning not only trades off latency but also reduces overall resource utilization and consequently throughput.

In addition, with polling-based SPDK I/O, having threads co-exist on the same cores loses the appeal of improving CPU utilization during I/O waits. This also applies to the common practice of managing LSM-tree flush/compaction tasks using background threads. In particular, as "fsync" with SPDK I/O involves busy-wait, the existing RocksDB design of unleashing potentially many background threads would create huge disruption to other threads and waste CPU cycles.

Recognizing these, SpanDB adopts asynchronous request processing, as illustrated in Figure 5. On an n-core machine, users configure the number of client threads as N_{client}, each occupying one core. The remaining $(n-N_{client})$ cores host SpanDB internal *server* threads, internally partitioned into two roles: *loggers* and *workers*. All these threads spin on their assigned cores. Loggers are dedicated to WAL writes, while workers handle both background processing (flush/compaction) and non-I/O tasks such as MemTable reads and updates, WAL entry preparation, and transaction related locking/synchronization. Based on the write intensity observed, a *head-server* thread automatically and adaptively decides the number of loggers, who are bound to cores with SPDK queue allocation that protect WAL write bandwidth.

Asynchronous APIs. SpanDB provides simple, intuitive asynchronous APIs. For existing RocksDB synchronous `get` and `put` operations, it adds their asynchronous counterparts `A_get` and `A_put`, plus `A_check` to examine request status. Similar API expansion applies to `scan` and `delete`. Accordingly, SpanDB expands RocksDB's `status` enumeration.

Figure 6 gives a sample client code segment. Here the client adopts the inherent spirit of asynchronous processing: to overlap wait with active work. It issues `A_put` requests in a loop, moving on to check the status of outstanding requests (and perform proper processing upon their completion), followed by issuing another request. A new request may be temporarily rejected by SpanDB, via the *IsBusy* status set within the `A_put` call, in which case the client will resubmit later.

SpanDB request processing. SpanDB manages the stages of foreground request processing, as well as background flush/compaction tasks in a number of queues. These queues pass sub-tasks among threads and also provide feedback on a certain system component's stress level. Based on such feedback, SpanDB could regulate the client request issuing rate (via the aforementioned `IsBusy` interface) or dynamically adjust its internal task allocation among workers.

```
Request *req = null;
while(true){
    if(req == null)
        req = GenerateRequest();
    LogsDB->A_put(req->key, req->value, req->status);// issue async req

    if(!(req->status->IsBusy())){
        pending_queue->enqueue (req);
        req = null; // ready to generate next req
    }
    for(Request* r in pending_queue){
        if (A_check(r->status)==completed) { // check outstanding reqs
            pending_queue.remove(r);
            custom_process(r);
        }
    } // end for
} // end while
```

Fig. 6. SpanDB API example.

Figure 5 illustrates the relevant SpanDB task queues. The flush and compaction queues (Q_{Flush} and $Q_{compact}$) are from RocksDB's existing design, though SpanDB modifies the actual operations to use SPDK I/O. In addition, SpanDB adds four queues: one for reads (Q_{Read}), and three to break up writes (Q_{ProLog}, Q_{Log}, and Q_{EpiLog}).

For asynchronous reads, SpanDB retains the RocksDB synchronous model when a request requires no I/O. With typical locality in KV applications, many reads are served from the MemTable, especially with larger MemTables enabled by spacious DRAM today. Given a key, the client quickly checks whether it is a MemTable hit and if so, completes the read operation itself. Such a "lucky read" takes only 4–6 μs end to end, as opposed to 30 μs on average even when reading from Optane under contention. Otherwise, the client inserts the request into Q_{Read} and returns. A worker will later pick it up, completing the rest of the RocksDB read routine and setting its completion status.

For asynchronous writes, SpanDB breaks its processing into three parts. The client simply dumps a request into Q_{ProLog}, to be processed by a worker. The latter generates a WAL log entry, which in turn is passed into Q_{Log}. Both queues are designed to promote batched logging (described in Section 2.2): a worker/logger would grab all the items in these queues. Beyond batching, the loggers pipeline log writes, maximizing SPDK write concurrency (see Section 4.2). After writing a batch to the SD, a logger adds the appropriate requests to Q_{EpiLog}, for workers to complete their final processing, including the actual MemTable updates. Like reads, tasks here require individual attention and no speedup can be achieved from their batching. As seen in Figure 5, Q_{ProLog} and Q_{Log} are flat lock-free queues, which allow easy "grab all" dequeuing. The other two, Q_{Read} and Q_{EpiLog}, are circular queues and only require locks in dequeue operations.

Task scheduling. The above SpanDB queues provide natural feedback for adjusting internal resource allocation. Our SPDK benchmarking results (Figure 3) shows that high-end NVMe SSDs offer parallelism but can be saturated by a few cores each issuing several concurrent requests. Hence SpanDB starts with one logger, growing and shrinking this allocation between 1 and 3 according to the current write intensity. The workers, however, are flexible to work on all the other queues, both foreground and background. Among the three foreground queues, SpanDB performs load balancing based on their queue length weighted by their average per-task processing time. Between the foreground and background queues, SpanDB prioritizes foreground, with an adaptive threshold to monitor background queue length, to proactively perform cleaning up, especially with write-intensive workloads.

Transaction support. SpanDB fully supports transactions and provides an asynchronous commit interface `A_commit` by making a few minor changes to RocksDB. Note that in RocksDB's

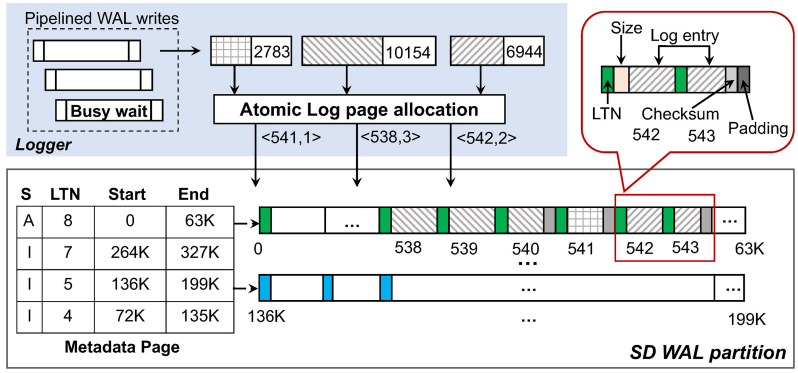

Fig. 7. SpanDB's parallel WAL logging mechanism.

transaction mode, writes will generate WAL entries in an internal buffer, which is only written by the commit call. The difference here is that `A_commit` inserts corresponding write tasks into Q_{ProLog}.

4.2 High-speed Logging via SPDK

Enabling parallelism and pipelining. SpanDB uses SPDK to flush log entries to raw NVMe SSD devices, bypassing the file system and Linux I/O stack. It retains the group logging mechanism described in Section 2.3, but enables multiple concurrent WAL write streams. Rather than having one client as leader (and forcing followers to wait), it employs dedicated loggers, who issue simultaneous batch writes. Each logger grabs all requests it sees in Q_{Log} and aggregates these WAL entries into as few 4KB blocks as possible. It performs pipelining by stealing the SPDK busy-wait time for one request to prepare/check others, as introduced in Section 2.3. For instance, with 2L4R, there are up to 8 outstanding WAL write groups.

Log data management. Parallel WAL writes complicate log data management, especially on a raw device without a file system. Luckily, with atomic 4 KB SPDK writes, coordinating concurrent WAL streams adds little overhead.

SpanDB allocates a configurable number of logical pages on the SD to its WAL area (10 GB in our evaluation), each with a unique ***logical page number (LPN)***. One of them is set aside as a *metadata page*. At any time, there is only one mutable MemTable, whose log "file" grows. We allocate a fixed number of *log page groups*, each containing consecutive pages and large enough to hold logs for one MemTable. Occupied log pages are organized by their corresponding MemTables: SpanDB conveniently reuses the RocksDB MemTable's "log file number" field as a ***log tag number (LTN)***, embedded at the beginning of all log pages for recovery.

Figure 7 gives an example of having four MemTables, one mutable (active) and three immutable, with different status ("A" for "active" and "I" for "inactive") in the metadata page. The long stripes in the bottom show two of the log page groups allocated. After a MemTable is flushed, its entire stripe of log pages is recycled, guaranteeing a MemTable's contiguous log storage. For each immutable MemTable, the metadata page records the start and end LPN of its log pages. Given that typical KV stores use a small number of MemTables, one page is more than enough to hold such metadata.

With loggers issuing concurrent requests, each supplying a WAL data buffer and size, the only synchronization point is log page allocation. We implement lightweight *atomic page allocation* with **compare-and-swap (CAS)** operations. Figure 7 shows three requests allocated one, three, and two pages, respectively, who can then proceed in parallel. These WAL writes do not modify the

metadata page, where the per-MemTable end LPN is only appended when that MemTable becomes immutable.

Within a log page, the logger first records the current LTN, followed by a set of log entries, each annotated with its size. The zoom-in part in Figure 7 portraits such layout, including the per-entry checksum (already calculated in RocksDB).

Correctness. SpanDB's parallel WAL write design preserves the RocksDB consistency semantics. It does not change the concurrency control mechanism used to coordinate and order client requests. Therefore, transactions with happened-before restrictions never appear out of order in the log pages, as briefly explained below. RocksDB's default isolation guarantee is `READ COMMITTED`. It also checks *write-write conflicts* and serializes two concurrent transactions that simultaneously update common KV items. With these two isolation guarantees, for any two update transactions T_1 and T_2, `READ COMMITTED` implies that if T_1 happens before T_2 (i.e., T_2 sees the effects of T_1), then T_1 must commit before T_2 started. By the design of the RocksDB group WAL write protocol, the above implies that the log entries of T_1 and T_2 should appear in two batches, where the batch commit of T_1 arrive earlier than and complete before the one of T_2. While log batches are written in parallel with SpanDB, they pass a serialization point for atomic page allocation. Therefore, T_1's batch is still guaranteed to obtain a lower sequence number than the one of T_2, for the latter to see the updates of the former. Similarly, When recovering from WAL data, SpanDB always performs redo in ascending order of sequence numbers.

Log recovery. Recovery is rather straightforward. When rebooting from a crash, the recovery process first reads the metadata page, to retrieve the number of log page groups and their corresponding page address ranges. The actual recovery from a log page group is highly similar to RocksDB's from a log file. Again, the LTN number in each page helps identify the "end" of the active log page group.

However, the one complication we find is that as SpanDB recycles log page groups, which contain old log pages, during recovery SpanDB needs to know which pages of the current log group have been overwritten. RocksDB relies on the file system during recovery: it reads whatever data is contained in the active log file. Without the file system, SpanDB could persist a separate metadata update or wipe out old log pages (e.g., by writing 0s) before recycling them. Both approaches double the WAL I/O volume and cut the SD's effective WAL write bandwidth in half. Instead, we reuse the per-MemTable LTN as a log page "color." Since the SSDs can guarantee 4K atomic writes to the device and the LTN is always written at the beginning of a page, the pages themselves reveal the location of the last successful writes. Recall the metadata page maintains the current/active LTN (the one with status "A")—a page within this group but with an obsolete LTN has not yet been overwritten from the current MemTable.

4.3 Offloading LSM-tree Levels to SD

For sustained, balanced execution of KV servers, SpanDB migrates the top levels of the RocksDB LSM-tree to the SD, offering users more return from their hardware investment. Below, we discuss the major challenges and solutions involved.

Ensuring WAL write priority. While flush/compaction could eventually block foreground writes if neglected long enough, in most cases, their latency remains hidden from users. Therefore, the SD should ideally utilize the *residual bandwidth* available, but yield to WAL writes, whose latency is fully visible to users. SPDK provides enough NVMe *queue pairs* (each composed of one submission and one completion queue): 31 on Intel Optane P4800X and 128 on Intel P4610. This enables separate management of different request types. Unfortunately, none of the existing commodity SSDs implement priority management over these queues [34]. Also, these queues offer very limited operations: users could only issue requests and check completion status.

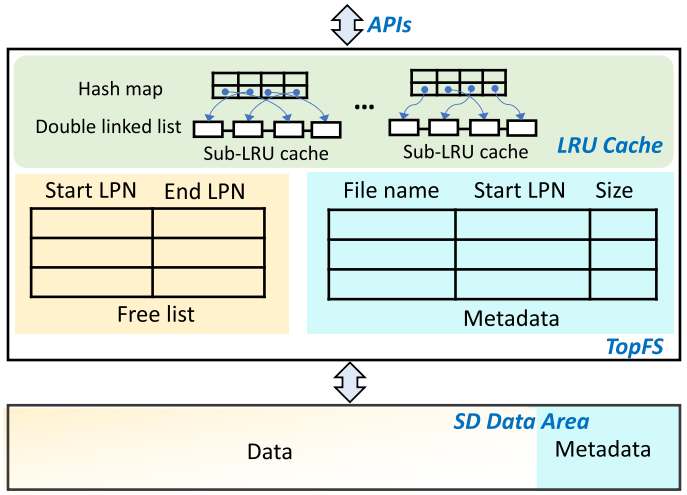

Fig. 8. The architecture of TopFS.

Therefore, besides the foreground-background coordination done at its queues (Section 4.1), SpanDB needs to prioritize WAL requests. We found their priority could be effectively protected by (1) allocating dedicated queues to each logger request slot (i.e., 8 queues for L2R4), (2) reducing the flush/compaction I/O request size from the RocksDB default of 1 MB to 64 KB to minimize their I/O contention with WAL, and (3) limiting the number of *worker* threads assigned to perform flush/compaction.

Dynamic-level placement. With all the above mechanisms, we can dynamically adjust the number of tree levels residing on SD. Initially, we pursued an analytical model to directly compute an optimal SD-CD level partitioning that maximizes overall system throughput. However, we could not find accurate LSM-tree write amplification models that agree with our measurement. In particular, state-of-the-art work on this front [54] seems to not take into account write speed and its variation. Our tests show that these factors could heavily impact the transient "tree shape" (with the top levels bulging out at different degrees beyond their size limit) and consequently the write/read amplification level.

Therefore, SpanDB settles for ad hoc, dynamic partitioning, by observing the sustained resource utilization level imbalance between the SD and CD. Its head-server thread monitors the SD bandwidth usage, and triggers the SST file relocation when it is below BW_L, till either it reaches BW_H or the SD is full, where BW_L and BW_H are two configurable thresholds.

Rather than migrating data between SD and CD, as the SST files constantly go through merging, SpanDB gradually "promotes" or "demotes" a whole level by redirecting their file creation to a different destination. It has a pointer that indicates currently which levels should go to the fast NVMe device. For example, a pointer of three covers all top three levels. However, this pointer only determines the destination of *new* SST files. Therefore, it is possible to have a new L3 file on SD and an older L2 file on CD, though such "inversions" are rare as the top levels are smaller and their files are updated more often.

4.4 TopFS Design

One constraint in using SPDK on an NVMe SSD is that the whole device has to be unbound from the native kernel drivers, and cannot be accessed through the conventional I/O stack. Therefore,

Table 1. The Required APIs of TopFS

	API	Description
File operations	`newWritableFile(fname, size)`	Create a file for writing
	`newRandomAccessFile(fname)`	Open an file for reading, return a file object
	`fileExists(fname)`	Check if the file exists
	`deleteFile(fname)`	Delete a file
	`getFileSize(fname)`	Get the file's size
	`renameFile(fname, des_name)`	Rename a file
	`close()`	Close the file
Data operations	`read(offset, size, data)`	Read data from the offset with given size
	`append(data, size)`	Append the data to the file's local buffer
	`fsync()`	Write the buffered data to the storage device

one cannot partition the SD to use SPDK only for writing WAL to one area and install a file system on the other. To minimize modifications to RocksDB I/O, SpanDB implements TopFS, a stripped-down file system, providing familiar file interface wrappers on top of SPDK I/O.

The SST files' append-only and thereafter immutable nature, plus their single-writer access pattern, simplifies the TopFS design. For example, file sizes are known at creation (for flush, with an immutable MemTable's size fixed) or have a known limit (for compaction). Also, each SST file is written in entirety once, by a single thread, from either flush or compaction. In both cases, the input data (memtable and corresponding WAL for flush, and source SST files for compaction) are not deleted till the SST file write successfully completes. In addition, TopFS does guarantee data persistence upon file close. These patterns allow simplified consistency management and recovery, and enable the allocation of per-file contiguous LPN ranges, similar to the aforementioned log page groups.

Figure 8 shows the overall architecture of TopFS. TopFS manages space allocation using an LPN free list, where contiguous LPN ranges are merged. Metadata management is simple: a hash table, indexed by file name, stores the files' start LPNs and sizes. SPDK bypasses the OS page cache, but implements a similar one.

APIs. We use TopFS to manage the top-level SST files on raw NVMe devices. Due to the unique characteristics of the SST files mentioned above, TopFS only implements a minimal set of APIs. As shown in Table 1, TopFS' APIs generally fall into two categories: those used for file operation (e.g., create/delete/rename files), and those for data operations (e.g., file data read/write). The SST files are only created during compaction or flush, and in these cases, the file size is known upon creation, so we can use *newWritableFile(fname, size)* to create new files by allocating the LPNs on SD for them and returning the corresponding file objects. Since SST files are append-only, we use `append(data, size)` to write new data to the end of the target file, which will be first stored in the corresponding file local buffer. When *fsync()* is called, the buffered data will be flushed to the underlying SD. Users can adjust the invocation frequency of *fsync()* to allow batching writes and tune the size of the batched write request to the storage device. Finally, the *read(offset, size, data)* function is used to read the data from the offset in the file. All these APIs make TopFS compatible with the original RocksDB design, leading to easy data migration between SD and CD.

Space allocation. TopFS manages space allocation using an LPN free list, which contains a group of <start LPN, end LPN> pairs, each indicating a free range of LPNs. When SpanDB calls *newWritableFile()* to create a file on SD, TopFS will inspect the <start LPN, end LPN> pairs in the free list to find one suitable range whose size is larger than the target file demands. If it is found, then it will allocate a contiguous group of LPNs from that range according to the file's size, and

update the <start LPN, end LPN> pair's end LPN. Upon file deletion triggered by calling *deleteFile()*, TopFS will recycle its pages by returning the corresponding LPNs to the free list.

Metadata management. TopFS uses the first 1 MB space of the data area on SD as its metadata area. Since the SD's capacity is rather limited and each file's metadata is very small (about 30 Bytes), this 1 MB space is sufficient. Note that we can scale up/down the metadata space when SD becomes larger or smaller. To enable fast metadata lookups, we cache all the metadata in DRAM. In addition, the metadata will be modified upon file creation or deletion. TopFS persists such metadata changes to SD prior to acknowledging the *new* or *delete* requests.

Data cache. Another challenge SpanDB faces is that SPDK bypasses the OS page cache. If it is unattended, then this brings excellent raw I/O but disastrous application I/O performance. To overcome this, we implement SpanDB's own cache on TopFS. Here, we allow setting the granularity of cached objects to be either files or pages. Our evaluation (Figure 13) will show that the page-level caching brings significant performance gains when memory is constrained, compared with file-level caching. However, they both deliver identical performance when the cache space is large. We use the LRU algorithm implementation in Facebook's HHVM project [4] to manage the cached objects with the following optimization. To alleviate the contention between different threads, we split the global LRU cache into multiple sub-caches. Each sub-cache consists of a doubly linked list and a hash map, where the former keeps track of the access order of pages, and the latter accelerates the list lookup. The hash map is indexed by LPN, storing pointers to the proper doubly linked list. Each sub-LRU cache is protected using locking to avoid write-write conflicts.

4.5 Limitations and Discussions

Though we believe SpanDB can benefit a wide range of applications, there are still several open challenges to address before making it more general and applicable to an even wider context. More specifically, we recognize three limitations with the current SpanDB approach, to be overcome in the future.

SD sharing. Due to its simple design principles presented above, TopFS cannot be shared by multiple SpanDB instances. However, as the capacity of high-end NVMe SSDs continues to grow [5, 8], multi-tenancy is considered to be important in cloud environments for better resource provisioning. To address this, we suggest two ways to run multiple SpanDB processes on a single NVMe SSD. First, we can make TopFS to be a general file system with locking, isolation, and consistency features to handle concurrent accesses to the shared space. However, this would complicate the design of TopFS and result in performance degradation. As a result, compared to the shared file system, the storage virtualization technique is more promising. One physical NVMe SSD can be partitioned into several virtual NVMe SSDs, each of which can serve one SpanDB instance and run its build-in TopFS. There have been already a few related works that focus on NVMe SSD virtualization [60, 75, 77], and we can leverage such techniques in SpanDB.

Read-intensive workloads. For all-read workloads, SpanDB produces little speedup and introduces slight overhead in asynchronous processing. To alleviate this overhead, SpanDB can detects the absence of writes and switches to synchronous processing. It is doable though it might be easier for RocksDB applications to simply stay with the synchronous read APIs (or even write ones too, if the workload is expected to be read-intensive). Just like with file systems or message passing libraries, mixed-use of synchronous and asynchronous KV APIs works correctly with SpanDB.

CPU utilization. With this prototype, SpanDB assumes the use of mature resource scaling technology common in data-centers/clouds, and targets maximizing throughput on a given set of cores within a node/VM. So all the internal threads check the different queues in a busy loop. This may waste the CPU cores under light loads. But it is very easy to enable queue wait monitoring in SpanDB, with its head-server thread directing other internal threads to sleep under light loads.

Table 2. The List of Enterprise Disks that Are Tested (Pricing from CDW-G on 09/15/2020 [9])

ID	Model	Interface	Capacity	Price	Seq. write bandwidth	Write latency	Endurance (DWPD)
H	WD HDD Ultrastar DC HA210	SATA	1 TB	$106 $0.11/GB	110 MB/s	4,200 μs	∞
S	Intel SSD DC S4510	SATA	960 GB	$248 $0.26/GB	510 MB/s	37 μs	1.03
N1	Intel SSD DC P4510	NVMe	4.0 TB	$978 $0.25/GB	2,900 MB/s	18 μs	1.03
N2	Intel SSD DC P4610	NVMe	1.6 TB	$634 $0.40/GB	2,080 MB/s	18 μs	1.03
O	Intel Optane SSD P4800X	NVMe	375 GB	$1221 $3.25/GB	2,000 MB/s	10 μs	30

DWPD (Drive Writes Per Day) measures the times/day one could overwrite an entire drive for its lifetime. Note that H and S are used in (4+1) RAID5 arrays, while the listed numbers here are single-disk data.

5 EVALUATION

5.1 Experimental Setup

We implemented SpanDB[1] on top of RocksDB, with around 6,000 lines of C++ code for its core functionality, plus 300 lines for integration with RocksDB.

Platform. We use a server with two 20-core Xeon Gold 6248 processors and 256 GB DRAM, running CentOS 7.7. The storage setting, denoted in "CD-SD" pairs, involves four data center device types. Among them, SATA SSDs (Intel S4510, "S") are used to form an 4+1 RAID5 group. As SPDK does not apply to SATA devices, S is used as CD only. Beside Optane P4800X ("O"), we test two more Intel DC NVMe SSDs as CD and SD, respectively: P4510 ("N1") and P4610 ("N2"), the former being larger, cheaper, and with higher bandwidth. The device details are in Table 2. Finally, we access CD via ext4, widely adopted in KV stores studies [18, 22, 56, 63].

Baseline and system configurations. Our natural baseline is vanilla RocksDB (v6.5.1), the base of SpanDB's development. Unless otherwise stated, all tests with RocksDB and SpanDB share the following configurations. Considering the current trend of larger DRAM space in servers, we use four 1 GB MemTables, and set the maximum WAL size to 1 GB. RocksDB is set to use up to 6 threads for compaction and 2 for flush. We follow common practice in performance evaluation that turns off compression when using synthetically generated requests [14, 53, 56, 63]. The remaining parameters are set to RocksDB default. Additionally, we compare with two recent key value stores designed for high-performance SSDs, namely, KVell [51] and RocksDB-BlobFS [67].

Workloads and Datasets. We run microbenchmarks and two popular KV workloads, YCSB [21] and Facebook's LinkBench [10]. For most tests with YCSB, we follow common practice [51, 53, 56] and use 1 KB KV item size, loading a 512 GB database with randomly generated keys as the initial state. The query phase issues 20M requests (preceded by 30% extra requests for warm-up).

5.2 Microbenchmark Results

Adaptive KV data placement. To assess SpanDB's automatic LSM-tree level placement, we use 3 YCSB-like workloads with different write intensity (Figure 9). We tested a 512 GB database on the S-O device combination. To compare with SpanDB's adaptive setting ("Auto"), we configured SpanDB with different fixed placement options: "PL" ("pure logging," where the SD is used solely

[1]Publicly available at https://github.com/SpanDB/SpanDB.

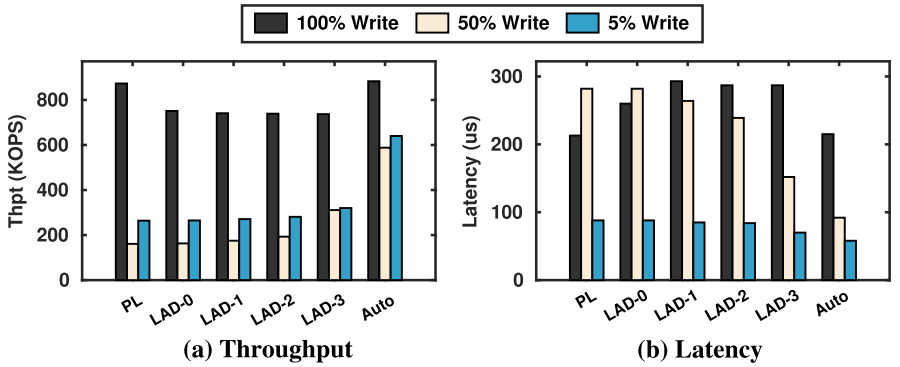

Fig. 9. Impact of data placement in SpanDB (S-O steup, 512 GB database). "PL" represents a "pure logging" configuration, where the SD is used solely for WAL writes, "LAD-n" indicates that the top n levels of the LSM-tree is placed on the SD, and "Auto" corresponds to the SpanDB's adaptive setting.

for WAL writes), and "LAD-n" (the top n levels of the LSM-tree is placed on the SD). The left and right charts show request processing throughput and average latency, respectively.

The results indicate that different workloads see different configuration sweet points. With a write-only workload, PL enables the fastest absorption of write bursts, dedicating the SD to WAL writes. The fixed placement plans (LAD-0 to LAD-3) deliver lower yet almost uniform performance, due to that their total data size (27.3 GB) is rather small relative to the total 512 GB database. They are slower in writes by adding flush/compaction traffic to the SD, while moving one more or fewer (small) layer here has little impact. Please note that we cannot evaluate LAD-4 here, as the total database size (over 300 GB), plus the WAL area and the temporary top tree level growth to accommodate fast writes, would run out of the usable space of the Optane disk (around 330 GB in our experience). SpanDB's auto policy here matches the PL performance by adopting the same placement.

With more read-intensive workloads, using the SD for data helps by speeding up reads. Again, only LAD-3 brings visible improvement as the previous levels are quite small. SpanDB's auto placement, however, roughly doubles throughput and halves latency from LAD-3. Its dynamic strategy does not have to migrate an entire tree level: here it ends up moving about 72% of the L4 data to SD, cutting average read latency significantly. For the rest of the article, we evaluate SpanDB with its auto data placement.

Adaptive background I/O coordination. Here we use a multi-phase workload to simulate time-varying user behavior common in production environments [37]. It begins with bursty requests, issuing 1.5M requests at the beginning of multiple 25 s episodes with 50% writes and 50% reads (Zipfian key distribution), followed by around 35 s of 100% writes, and finally 25 s of 95% reads. Figure 10 portraits the request throughput, latency, and background activity level (flush/compaction task counts as reported in RocksDB).

We compare SpanDB's auto adaptation with two fixed configurations: "BG-L" (RocksDB default, one thread each for compaction and flush), and "BG-H" (6 compaction and 2 flush threads). During phase 1 (bursty), BG-H performs the worst, with 2× higher average latency and 39% lower average throughput than BG-L during each burst. After an initial period of write accumulation, the foreground tasks become severely interfered by its aggressive compaction. "Auto" behaves quite similarly to BG-L during the write request bursts, prioritizing foreground tasks. Unlike with the fixed thread allocation in RocksDB, its background I/O is not constrained to a few threads. So after the burst passes, SpanDB Auto loses no time in catching up with background tasks, resulting in

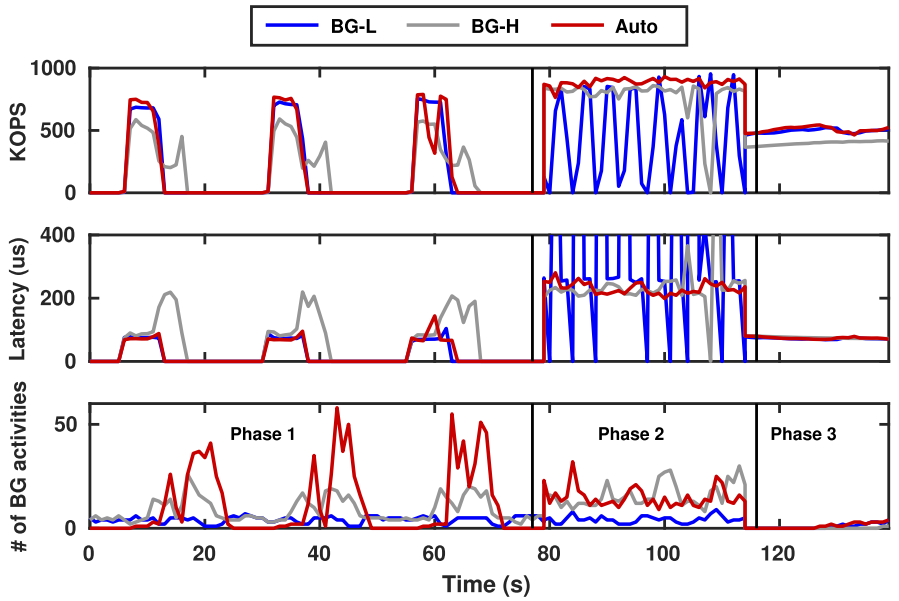

Fig. 10. Impact of SpanDB background I/O configurations (S-O setup, 512 GB database).

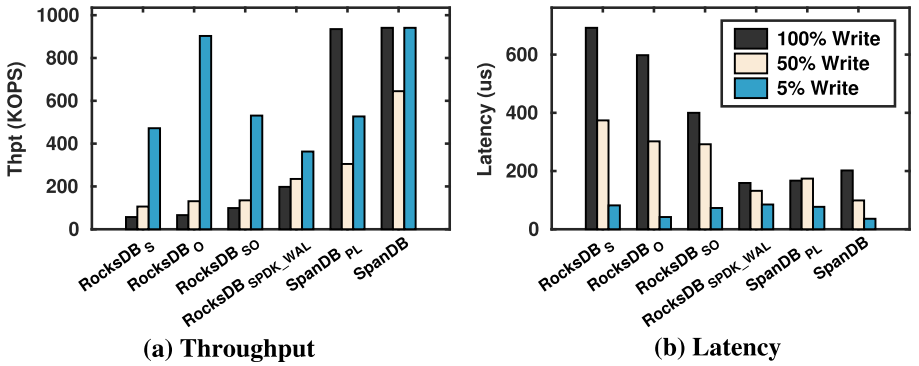

Fig. 11. Performance of different RocksDB and SpanDB configurations (S-O setup, 100 GB database).

"background compaction bursts" (red peaks in the bottom figure) much more intense than both BG-L and BG-H. Overall, this leads to faster completion of backlogged compaction tasks, and better preparation for future write bursts.

In the second phase (all-write), BG-L regularly stalls foreground processing, producing dramatic latency/throughput fluctuations, which does not happen with the more compaction-conscious BG-H or Auto. With higher background resource allocation, BG-H still performs worse than Auto (due to its less pro-active compaction), obtaining slightly lower throughput and having one write stall. For the last phase (read-intensive), with light flush/compaction load, BG-L and Auto achieve nearly identical performance, while BG-H lags behind in throughput, due to wasting thread allocation (as required by SPDK to be bound to a core) to background tasks.

This confirms that SpanDB's asynchronous workflow, designed mainly to reduce software overhead with polling I/O, also enables adaptive background task scheduling.

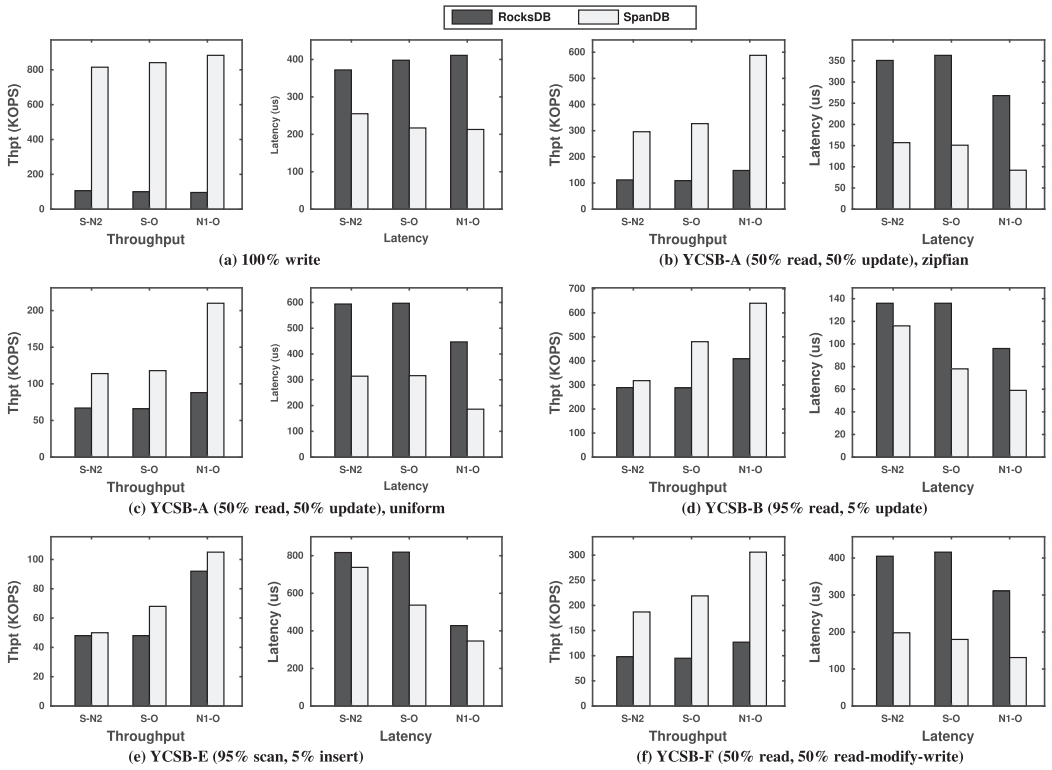

Fig. 12. Throughput and latency of various YCSB workloads, 20M requests on 512 GB database. (YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 95% scan and 5% insert, YCSB-F: 50% read and 50% read-modify-write).

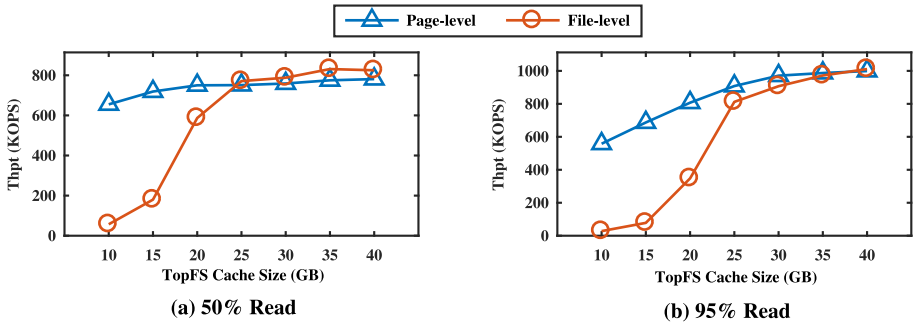

Fig. 13. Throughput of two different workloads, two different cache policies, varied cache sizes, 20M requests on 100 GB database with single O.

Cache efficiency. We compare the performance of the file-level and page-level caching policies in Figure 13. Since read requests are more sensitive to the cache size and policy, we only show the results of YCSA (50%read) and YCSB (95%read) workloads. We additionally change the cache size from 10 to 40 GB. We observe that page-level caching policy significantly outperforms the file-level one when the cache size is small (e.g., smaller than 30 and 25 GB for the two workloads,

respectively). This is because the file-level caching policy results in more eviction operations and suffers high eviction overhead when the memory is limited. However, when increasing the cache size up to 25 and 30 GB for the two workloads, respectively, both policies achieve almost the same performance. This is because the cache efficiency is improved and the number of eviction operations decreases when a large number of objects are cached.

Breakdown analysis. Figure 11 breaks down SpanDB's improvement by incrementally enabling its individual techniques, again with workloads at different write intensity. The first four bar groups show variants of RocksDB, while the last two show variants of SpanDB. To enable $RocksDB_O$, RocksDB's execution on a single fast disk (Optane), we use a smaller database (100 GB). With $RocksDB_{SO}$, RocksDB uses the SD (O) for WAL and CD (S) for all data. $RocksDB_{SPDK\_WAL}$ uses the same setting, only with WAL writes via SPDK instead of ext4. $SpanDB_{PL}$ adds asynchronous processing and parallel WAL writes, while *SpanDB* enables auto-placement of data.

From the all-write results, one sees clearly how little the hardware upgrade matters with RocksDB ($RocksDB_S$ to $RocksDB_O$). Separating logging with data I/O helps ($RocksDB_{SO}$), and adopting SPDK further doubles write throughput. Still, $RocksDB_{SPDK\_WAL}$ only unlocks a small fraction of the Optane disk's concurrent small write performance, as demonstrated by SpanDB (both PL and Auto), who achieves a 4.5× throughput.

When the workload becomes balanced (50% read), the performance growth becomes less dramatic, though still very significant. Here $RocksDB_S$ and $RocksDB_O$ have almost identical performance, as adding a CD helps offloading write traffic, but lowers read speed. SpanDB's auto version beats the best RocksDB variant by 2.74×, and nearly doubles the throughput of its PL version, as the fast SD accelerates reads. With 95%-read (blue bar), $RocksDB_O$ stands out among RocksDB variants, showing that with reads, the vanilla RocksDB on ext4 actually quite efficiently utilizes the Optane disk (consistent with benchmarking results in Figure 2). SpanDB's auto version, in this case, also chooses to place its data on the SD and matches the $RocksDB_O$ performance.

5.3 Overall Performance

We use YCSB and LinkBench to evaluate SpanDB's overall performance against RocksDB, on three CD-SD hardware pairs: S-N2, S-O, and N1-O. Note that RocksDB allows easy assignment of logging destination; therefore, we set it to also writes WAL to the SD (its LSM-tree levels, however, cannot be relocated without substantial code change). Hence the RocksDB baseline evaluated does use both disks in the CD-SD pair, though via the file system.

YCSB write-intensive tests. As SpanDB primarily targets write optimization, we start with write-intensive workloads.

With all-write (Figure 12(a)), issuing 20M write requests (Zipfian key distribution), RocksDB delivers uniformly low performance across all three device pairs. This reveals how existing systems, logging sequentially via a file system, fail to utilize high-end SSDs well. From this baseline, SpanDB dramatically improves *both* throughput and latency, bringing a throughput increase of 7.6-8.8× across different CD-SD combinations, while reducing average latency by 1.5-2×. This higher improvement on throughput than on latency is attributed to our parallel batch logging (both in Q_{ProLog} and Q_{Log}). For example, on S-O, the RocksDB log batch size averages around 20. SpanDB has an average batch size of around 7, but may have multiple threads process batches in parallel.

Figures 12(b) and 12(c) give results for YCSB-A (50% reads and 50% updates), with Zipfian and uniform key distribution, respectively. Having 50% reads, on such a large database, actually slows down overall request processing, as reads cannot be batched. Here SpanDB's improvement over RocksDB remains significant: improving throughput by 2.6–4.0× while reducing latency by 2.2–3.0× (Zipfian distribution). With more reads, both systems are more sensitive to the underlying storage hardware, and the N1-O combination excels due to N1's lower read latency than

Table 3. Latency Comparison in YCSB Tests on S-O

		Median (μs)	P90 (μs)	P99 (μs)	P999 (μs)
All-write (Zipf)	RocksDB	372.0	488.7	856.2	1,506.4
	SpanDB	187.2	425.6	726.8	1,170.1
YCSB-A (Zipf)	RocksDB	284.7	469.4	2,695.1	25,720.8
	SpanDB	68.3	302.9	1,487.5	8,679.9
YCSB-A (uniform)	RocksDB	314.9	810.1	6,296.0	31,971.6
	SpanDB	114.2	450.3	4,147.9	15,697.6
YCSB-B (Zipf)	RocksDB	14.7	471.5	803.4	1,201.1
	SpanDB	25.5	277.1	507.6	715.7
YCSB-E (Zipf)	RocksDB	199.9	2,844.0	6,016.6	8,387.5
	SpanDB	240.6	1,404.1	4,241.6	8,483.9
YCSB-F (Zipf)	RocksDB	297.3	685.4	2,801.7	23,519.0
	SpanDB	65.6	361.2	1,848.2	10,388.9

S. Meanwhile, compared with RocksDB, SpanDB harvests much more performance gain from this device pair.

With uniform distribution, SpanDB's edge over RocksDB is weakened by having more memory cache read misses. Most data reside on the CD, where SpanDB's reads work similarly as the baseline. Still, SpanDB outperforms RocksDB by 1.7–2.4× in throughput and by 1.9–2.4× in latency.

Other standard YCSB tests. Next, we run the other 3 YCSB workloads: B, E, and F. Due to space limit, we give Zipfian results only, and omit C (no writes) and D (similar to B).

With the 95%-read YCSB-B and YCSB-E (Figures 12(d) and 12(e)), SpanDB still delivers moderate enhancement: throughput growth by 1.03×–1.66×, and latency cut by 9.5%–42%. Between them, it has a smaller gain with YCSB-E, dominated by scan operations and with a higher memory hit ratio (from reading a random number of consecutive keys). YCSB-F (Figure 12(f)) contains 50% reads and 50% read-modify-writes. Though its read ratio (75%) is between YCSB-A and YCSB-B, it behaves more like YCSB-A (with read-modify-write dominated by write cost): SpanDB outperforms RocksDB significantly in both throughput and latency.

Among all tests in Figure 12, except for the most read-intensive ones (B and E), SpanDB on the lowest device setting (S-N2) significantly outperforms RocksDB on the highest one (N1-O), demonstrating its cost-effectiveness. The two 95%-read workloads highlight the benefit of a low-latency CD, while SpanDB further boosts performance across all device pairs.

Finally, we report SpanDB's impact on tail latency, as shown in Table 3. Here, we only list the S-O results, and other configurations have similar results. Though the write-oriented SpanDB produces moderate overall performance improvement for read-intensive workloads as shown earlier, it reduces the tail latency of RocksDB in most cases (by 10%–92%). The only exceptions are with YCSB-B and YCSB-E, two 95%-read workloads. With YCSB-B, SpanDB brings a 73% higher median latency than RocksDB. Here most read requests are served from DRAM, with ultra-low latency. The extra overhead introduced by the asynchronous processing could generate a considerable shift in median latency. With YCSB-E, there are more long-running requests (see the P90 numbers), incurring higher queue-wait time for other requests, which does not apply to RocksDB's synchronous processing. Note that in both cases, SpanDB improves the workloads' overall performance (Figure 12). It reduces the average latency of YCSB-B and YCSB-E by 42% and 32%, and improves their throughput by 65% and 41%, respectively.

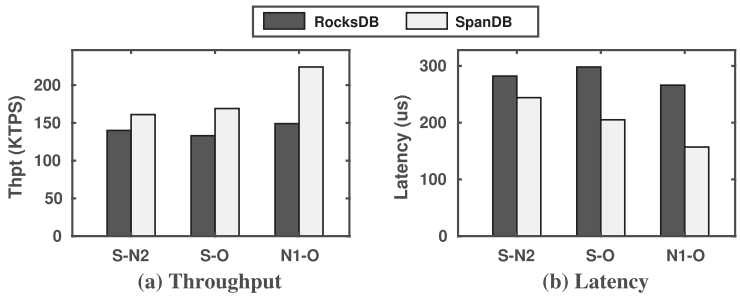

Fig. 14. Performance of LinkBench.

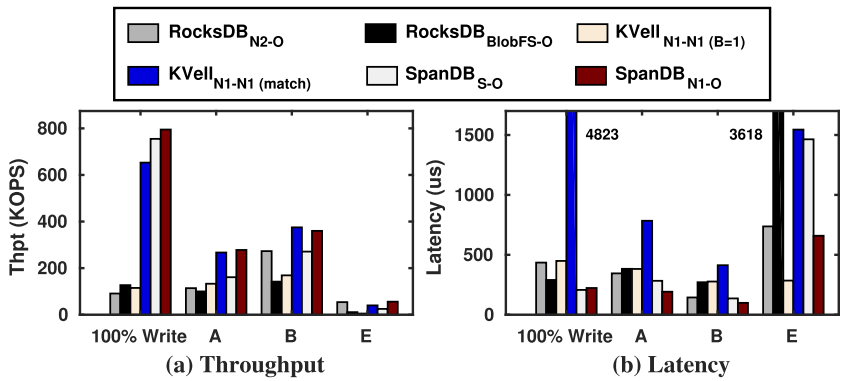

Fig. 15. Additional system comparison, 2 TB database (RocksDB-BlobFS w. 250 GB only).

LinkBench transactional workload. We assess SpanDB's asynchronous transaction processing with Facebook's LinkBench [10] (Figure 14). Our test uses a 206 GB database containing 600M vertices and 2,622M links, performing 20M requests with LinkBench's default configuration: 56% scan, 11% write, 13% read, and 20% read-modify-write operations. Again, for this overall read-intensive workload (around 70%-read), SpanDB fares well against RocksDB, increasing throughput by up to 50.3% and cuts latency by up to 41%. The results demonstrate SpanDB's effectiveness in handling graph OLTP workloads, where WAL writes cannot be forgone.

Comparison w. NVMe SSD-based systems. Finally, Figure 15 compares SpanDB against two recent systems leveraging fast NVMe SSDs: KVell [51] and RocksDB-BlobFS [67]. Here, we test with larger datasets, using a 2 TB database (except for RocksDB-BlobFS, which failed to run with larger sizes, and we included its 250 GB test results for reference.) We assess four YCSB workloads: all-write, A, B, and E.[2]

First, RocksDB-BlobFS, accessing a single Optane via BlobFS, delivers worse performance than RocksDB in most cases, even with a much smaller database. Then, we compare with KVell, which benefits from a shared-nothing design that partitions data across multiple disks, aggressive request batching, and elimination of sorting/compaction. Meanwhile, such a shared-nothing design with no logging creates challenges in handling transactions (which is not supported by the current KVell). As the 2 TB database runs beyond the O-O capacity, here it runs on N1-N1. We test KVell with two batch size settings: "B = 1" (lowest latency and throughput) and "match" (the smallest batch size that surpasses SpanDB's throughput).

[2]KVell's code base does not include YCSB-F, whose implementation was identical to YCSB-A according to the authors.

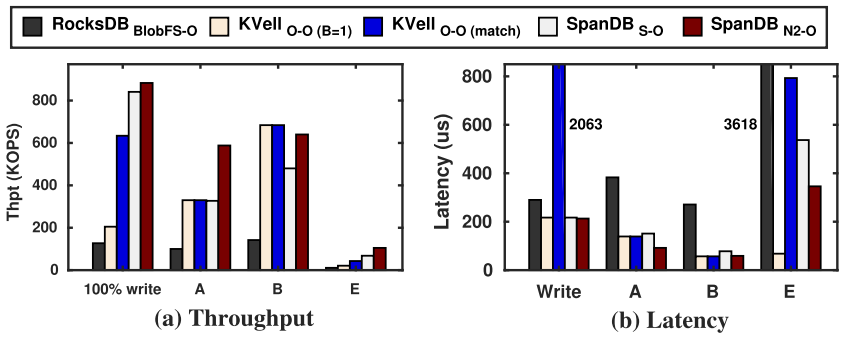

Fig. 16. Additional system comparison, 512 GB database (RocksDB-BlobFS w. 250 GB only).

With all-write, KVell cannot match SpanDB's throughput even at its largest batch size (64), where it suffers huge latency (average at nearly 5,000 μs). Batch size 1 delivers a throughput at 15.2% of SpanDB's S-O level, and an average latency at 2.17×. YCSB-A sees a similar contrast, though to a lesser degree. With the read-intensive YCSB-B, KVell slightly outperforms SpanDB N2-O in throughput with batch size at 3, but reports latency 4.17× higher, while batch size 1 loses in both throughput and latency. SpanDB also wins in scans (YCSB-E), producing a 1.4× throughput and 57% latency reduction compared against KVell at batch size 64.

Finally, we repeat the experiment in Figure 15 with the database size used in most of our tests (512 GB, though RocksDB-BlobFS can only run on a 250 GB one). For KVell, this database size allows us to deploy it on the highest configuration, on two Optane disks (O-O). Figure 16 portraits the results. Here RocksDB-BlobFS (on O) does not appear to bring improvement over our RocksDB results, and consistently falls behind in throughput across all workloads, especially for scans (YCSB-E). KVell, however, does utilize high-speed SSDs well, especially for reads. Meanwhile, it sacrifices latency with its aggressive batching.

SpanDB, even with the much cheaper S-O combination (as opposed to O-O used by KVell), achieves 1.33× and 1.58× higher throughput than KVell's higher throughput ("match" batch size) for all-write and YCSB-E, respectively, with significantly lower latency for the all-write case. With regard to the 50%-read YCSB-A, SpanDB on S-O matches both the throughput and latency of KVell. SpanDB on N2-O, meanwhile, achieves 1.78× higher throughput than KVell with 33.81% lower latency. For the 95%-read YCSB-B, SpanDB on S-O delivers 70.2% of KVell's throughput, while on N2-O it nearly matches KVell's performance (on O-O).

CPU utilization. With the 50%-write YCSB workload, SpanDB's CPU utilization is 94.5%, while RocksDB's CPU utilization is only 63.7%. This is a direct consequence of spinning threads on cores, as required by SpanDB's polling-based I/O and asynchronous request processing: All workers are busy with the request processing and never sleep. However, RocksDB's synchronization during processing write requests also contribute to the lower CPU utilization. Overall the system spends more time doing useful work: SpanDB delivers a 3× throughput improvement RocksDB in this experiment. Meanwhile, under light loads SpanDB can easily enable queue wait monitoring, with its head-server thread directing other internal threads to sleep when necessary.

5.4 Sustained Write Rate Study

Figure 10 features peak write throughput, where each system absorbs around 10 GB of writes. However, accumulated compaction has to be accommodated to keep a sustained write rate. Figure 17 and Figure 18 illustrate this with SpanDB writing using two very different device combinations,

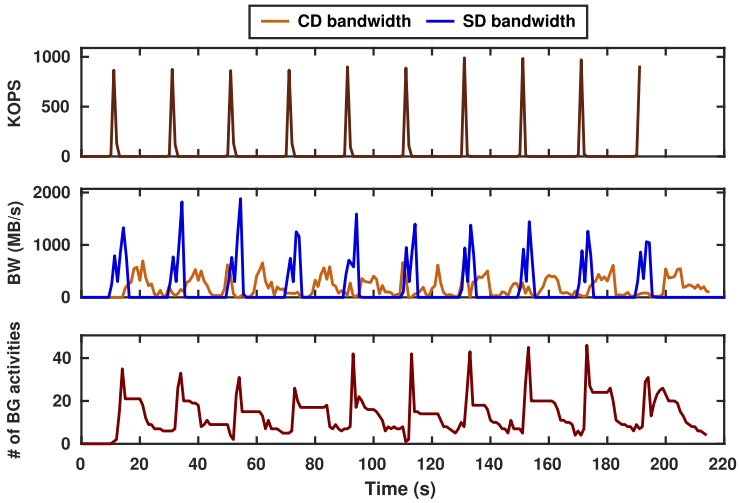

Fig. 17. Performance (throughput, SD/CD bandwidth, and background activities) of SpanDB with bursty writes (Zipfian key distribution, 1,000K requests in every 20 s episode) on H-O, 100 GB database.

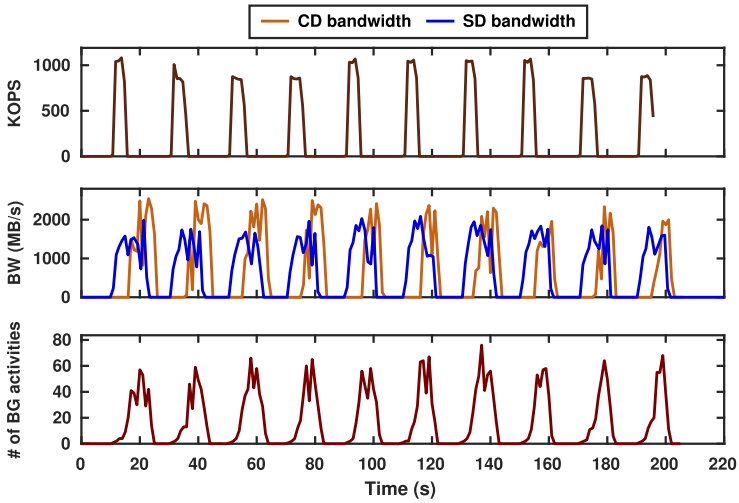

Fig. 18. Performance (throughput, SD/CD bandwidth, and background activities) of SpanDB with bursty writes (Zipfian key distribution, 4,000K requests in every 20 s episode) on N1-O, 100 GB database.

H-O and N1-O, when we issue bursty requests at nearly their highest sustained rate (50 KOPS on H-O and 200 KOPS on N1-O). In both figures, the top chart plots the foreground workload's throughput. The middle one plots the total I/O bandwidth consumed by the CD and the SD, respectively. The bottom one gives the level of background I/O activity (in number of compaction/flush tasks as defined in RocksDB).

Here, we see that while SD absorbs waves of write requests, both it and the CD work actively behind the scene to handle flush and compaction traffic. As a result, even the cheapest H-O configuration could "fake" a peak write speed in customer-facing request processing, as demonstrated by the similar KOPS bursts between H-O and N1-O (top plot). However, with its much stronger

Table 4. Recover Time of RocksDB and SpanDB on Optane

Recovery data size	Recovery time (s)	
	RocksDB	SpanDB
1 GB	2.24	2.23
2 GB	4.82	4.79
4 GB	10.27	10.25

power in background cleaning, N1-O delivers a sustained write rate 4 × higher. This test reveals that, with workloads that have moderate sustained write rate, with little customer-facing random reads (such as certain types of monitoring and sensor data), H-O may indeed provide a low-cost, high-performance solution.

5.5 Recovery

We also tested SpanDB's recovery by inserting system crashes at random time points in our experiments. Specifically, we verified that updates in a MemTable, which were persisted to WAL on SD before a crash, could be correctly recovered upon rebooting. Results show that SpanDB was successfully recovered in all cases.

We further report the time cost of recovering from a crash, comparing RocksDB and SpanDB on the Intel Optane device. Results in Table 4 demonstrate that both SpanDB and RocksDB achieve almost the same performance, reasonable as our earlier results show SPDK and ext4 deliver similar performance for large, sequential reads. As expected, the recovery time grows with total data size recovered.

6 RELATED WORK

Tiered storage. Multiple systems leverage tiering techniques on heterogeneous devices, mainly developing general-purpose file systems, such as NVMFS [61], Strata [48], and Ziggurat [83], transparently operating across NVRAM, SSD, and **hard disc drive (HDD)** layers. SpanDB is similar in exploiting the low latency of fast devices and the high bandwidth/capacity of slower ones. Its major novelty, meanwhile, lies in its KV-specific optimizations, many above the block storage layer. Also, its design addresses performance constraints brought by high-end commodity SSDs (as well as the new SPDK interface), rather than NVRAM units often emulated in evaluation.

HiLSM [52] and MatrixKV [78] use hybrid storage devices for KV. However, they both only intend to use a small portion (8 GB in both papers) of a fast and expensive NVM device. HiLSM uses this space as a layer of fast write buffer, focusing on different in-NVM data organization (hash table and red-black tree) to reduce write amplification. The majority of its tests use emulated NVM. MatrixKV uses this space for storing WAL and L0 data (with L0 and L1 size enlarged to GBs to reduce compaction bottleneck). Neither system discusses SPDK. MatrixKV uses the more expensive Optane PM. HiLSM significantly modifies the LevelDB workflow, while MatrixKV is complementary to SpanDB (their L0-L1 optimizations can be integrated with our system). In addition, they target byte-addressable NVM as the fast device, while SpanDB focuses on the efficient utilization of NVMe SSDs, which currently offer much wider commodity hardware choices and significantly lower cost.

Existing work has deployed LSM-tree-based KV stores across multiple devices. For example, Mutant [79] ranks SST files by popularity and places them on different cloud storage devices. PrismDB [62] makes LSM-trees "read-aware" by pinning hot objects to fast devices. SpanDB is similar in placing the top-level SST files to fast devices, but significantly differs from them by

focusing more on write processing (often harder to scale [18, 37]). To this end, it encompasses many new, NVMe-oriented optimizations such as leveraging SPDK, parallel logging, and adaptive flush/compaction.

KV stores optimizations for fast, homogeneous storage. Many recent KV systems target low-latency, non-volatile storage, mostly by designing novel data structures, such as UniKV [81], LSM-trie [73], SlimDB [64], FloDB [15], PebblesDB [63], KVell [51], and SplinterDB [20]. As WAL creates a major performance bottleneck, many of them turned off WAL in evaluation, while KVell completely removed the commit log. This may lead to data inconsistency and a lack of transaction support. SpanDB instead retains the data structure and semantics of the mainstream LSM-tree-based design. Moreover, the above systems assume homogeneous deployment, while SpanDB promotes heterogeneous storage that supplements older, slower devices with small, high-end ones.

Several systems deploy hardware solutions. X-Engine [37, 82] leverages hardware acceleration such as FPGA-accelerated compaction. KVSSD [45] and PinK [40] further offload KV management to specialized hardware, which are not commercially available yet. SpanDB, however, does not require special hardware support.

Another group of work optimizes KV stores on persistent memory, including HiKV [74], NoveLSM [46], NVMRocks [43], Bullet [39], SLM-DB [44], and FlatStore [19]. All use emulators in implementation/evaluation except FlatStore, which uses Intel Optane DCPMM. While KV stores directly running on persistent memory have undeniable performance advantages, hardware cost and capacity limit remain practical issues. The 256 GB Optane DCPMM cost 3.12× higher (per GB) than the O disk used in our tests, and 40.5× higher than N1. Also, they require more expensive processors. These systems, therefore, fit better read-intensive workloads with moderate dataset sizes. Our work targets large databases with substantial write traffic, and aims to deliver high performance while keeping the overall hardware cost low.

Also, FlashStore [24] uses flash as a cache for KV stores. MyNVM [26] reduces DRAM cache demand in MyRocks [29], building a second-layer cache on Optane SSD. SpanDB's SD, instead, is not designed to be a cache.

Logging optimizations. Wang et al. utilized NVM for enhanced scalability via distributed logging [70]. NV-Logging [38] proposes per-transaction logging to enable concurrent logging for multiple transactions. NVWAL [47] exploits NVM to speed up WAL writes in SQLite. Again, the above studies adopt emulation, and though now commodity NVM products are available their cost remains high, as discussed earlier. SpanDB, instead, improves WAL write performance on widely adopted NVMe block devices.

Other related work. The **Staged Event-driven Architecture (SEDA)** decomposes request processing into a sequence of stages and use queues to pipeline, parallelize, and coordinate their execution [71]. Similar ideas have been used in many systems, including DeepFlash [80] and ours.

There are many studies optimizing LSM-tree-based KV stores, such as SILK [14] (I/O scheduling to reduce the interference between client and background tasks), Monkey [22] and ElasticBF [53] (adopting dynamic bloom filter sizes to minimize lookup cost), TRIAD [13] (exploring workload skewness to reduce flush/compaction overhead), WiscKey [56] (separating keys and values to speedup sequential/random accesses), and HashKV [18] (WiscKey optimization targeting update-intensive workloads). Our work is orthogonal and complementary to the above techniques.

SPEICHER [12] focuses on security, leveraging SPDK to reduce system calls in data path when running RocksDB in the SGX enclave. SPEICHER runs on a single storage device and its evaluation aims to demonstrate its low overhead vs. the native RocksDB (no security).

LiveGraph [85] proposes a new data structure (TEL) targeting graph transactions, by promoting fast edge scans and in-place metadata for MVCC. Its LinkBench performance over RocksDB is significant and also workload-dependent. While users may select TEL over LSM for better graph

transaction performance, LiveGraph also uses sequential group WAL writes. It therefore can benefit from SpanDB's parallel logging. Hybrid storage applies there too: e.g., older edges could be stored on a slower disk as many graph database queries scan a given amount of most recent data.

7 CONCLUSION

In this work, we explored a "poor man's design" that deploys a small and expensive high-speed SSD at the most-needed locations of a KV store, while leaving the majority of data on larger, cheaper, and slower devices. Our results reveal that the mainstream LSM-tree-based design can be significantly improved to take advantage of such partial hardware upgrade (while retaining the major data structures and algorithms, as well as many orthogonal optimizations).

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their insightful feedback.

REFERENCES

[1] RocksDB. [n.d.]. A Persistent Key-Value Store for Fast Storage Environments. Retrieved from https://rocksdb.org/.
[2] LinkedIn. [n.d.]. Benchmarking Apache Samza. Retrieved from https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node.
[3] MariaDB. [n.d.]. Group Commit for the Binary Log. Retrieved from https://mariadb.com/kb/en/group-commit-for-the-binary-log/.
[4] GitHub. [n.d.] HHVM. Retrieved from https://github.com/facebook/hhvm.
[5] Enterprise Storage Forum. [n.d.]. High capacity SSDs: How Big Can They Grow? Retrieved from https://www.enterprisestorageforum.com/hardware/high-capacity-ssds-how-big-can-they-grow/.
[6] MySQL. [n.d.]. MySQL Reference Manual. Retrieved from https://dev.mysql.com/doc/refman/5.7/en/replication-options-binary-log.html#sysvar_binlog_order_commits.
[7] I-Programmer. [n.d.]. RocksDB on Steroids. Retrieved from https://www.i-programmer.info/news/84-database/8542-rocksdb-on-steroids.html.
[8] Samsung. [n.d.]. SSD Storage Capacities Increase With Improved Storage Density. Retrieved from https://insights.samsung.com/2016/06/28/ssd-storage-capacities-increase-with-improved-storage-density/.
[9] CDW-G. [n.d.]. Storage & Hard Drives—The CDW-G website. Retrieved from https://www.cdwg.com/content/cdwg/en/products/storage-and-hard-drives.html.
[10] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
[11] Andrew Audibert. Scalable Metadata Service in Alluxio: Storing Billions of Files. Retrieved from https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions-of-files/.
[12] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 173–190.
[13] 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)*.
[14] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*.
[15] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking memory in persistent key-value stores. In *Proceedings of the 12th European Conference on Computer Systems*.
[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. Retrieved from https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.
[17] 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 USENIX Conference on File and Storage Technologies (FAST'20)*.

[18] Helen H. W. Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick P. C. Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*.

[19] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[20] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[22] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM International Conference on Management of Data*.

[23] Jeff Dean. 2009. Designs, lessons and advice from building large distributed systems. In *Proceedings of the International Workshop on Large Scale Distributed Systems and Middleware: Keynote (LADIS'09)*.

[24] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 1414–1425.

[25] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[26] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. Article 42, 13 pages. https://doi.org/10.1145/3190508.3190524

[27] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. I/O is faster than the CPU: Let's partition resources and eliminate (most) OS abstractions. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*. 7 pages. https://doi.org/10.1145/3317550.3321426

[28] Facebook. [n.d.]. Cassandra on RocksDB at Instagram. Retrieved from https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram.

[29] Facebook. [n.d.]. Retrieved from MyRocks. http://myrocks.io/.

[30] Facebook. [n.d.]. Under the Hood: Building and Open-sourcing RocksDB. Retrieved from https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/.

[31] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[32] Dieter Gawlick and David Kinkade. 1985. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.* 8, 2 (1985), 3–10.

[33] Sanjay Ghemawat and Jeff Dean. 2014. LevelDB, A Fast and Lightweight Key/Value Database Library by Google. Retrieved from https://github.com/google/leveldb.

[34] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. Analyzing, modeling, and provisioning QoS for NVMe SSDs. In *Proceedings of the IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC'18)*. IEEE.

[35] Robert Hagmann. 1987. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*.

[36] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. 2019. WAL-SSD: Address remapping-based write-ahead-logging solid-state disks. *IEEE Trans. Comput.* 69, 2 (2019), 260–273.

[37] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the International Conference on Management of Data (SIGMOD'19)*. 15 pages. https://doi.org/10.1145/3299869.3314041

[38] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment* 8, 4 (2014), 389–400.

[39] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*.

[40] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. Retrieved from https://www.usenix.org/conference/atc20/presentation/im.

[41] Intel. [n.d.]. Breakthrough Performance for Demanding Storage Workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[42] Intel. [n.d.]. SPDK: Storage Performance Development Kit. Retrieved from https://spdk.io/.
[43] Andrew Pavlo, Jianhong Li, and Siying Dong. 2017. NVMRocks: RocksDB on Non-Volatile Memory Systems. Retrieved from http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.
[44] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.
[45] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*.
[46] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*. Retrieved from https://www.usenix.org/conference/atc18/presentation/kannan.
[47] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-ahead Logging. *ACM SIGPLAN Notices* 51, 4 (2016), 385–398.
[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 18 pages. https://doi.org/10.1145/3132747.3132770
[49] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44, 2 (2010), 35–40.
[50] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*.
[51] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.
[52] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM: An LSM-based Key-Value Store for Hybrid NVM-SSD Storage Systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 208–216.
[53] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*.
[54] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.
[55] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. Retrieved from https://www.usenix.org/conference/fast19/presentation/liu.
[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 Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
[57] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.
[58] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. Retrieved from https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.
[59] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proceedings of the VLDB Endowment* 7, 2 (2013), 121–132.
[60] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*. 665–676.
[61] S. Qiu and A. L. Narasimha Reddy. 2013. NVMFS: A hybrid file system for improving random write in NAND-flash SSD. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*.
[62] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J. Freedman. 2020. PrismDB: Read-aware Log-structured Merge Trees for Heterogeneous Storage. Retrieved from https://arxiv.org/abs/2008.02352.
[63] 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*.
[64] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[65] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD. Retrieved from https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[66] Dong Siying. 2017. Workload Diversity with RocksDB. Retrieved from http://www.hpts.ws/papers/2017/hpts2017_rocksdb.pdf.

[67] SPDK. BlobFS (Blobstore Filesystem)—BlobFS Getting Started Guide—RocksDB Integration. Retrieved from https://spdk.io/doc/blobfs.html.

[68] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, the Bad, and the Ugly. *ACM SIGARCH Computer Architecture News* 43, 1 (2015) 297–310.

[69] Toshiba. Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution. Retrieved from https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html.

[70] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.

[71] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. 14 pages. https://doi.org/10.1145/502034.502057

[72] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of Intel Optane SSD. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Retrieved from https://www.usenix.org/conference/hotstorage19/presentation/wu-kan.

[73] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

[74] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*.

[75] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou et al. 2020. Spool: Reliable Virtualized NVMe storage pool in public cloud infrastructure. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. 97–110.

[76] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A development kit to build high performance storage applications. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*. IEEE.

[77] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. 2018. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *Proceedings of the IEEE 8th International Symposium on Cloud and Service Computing (SC'18)*. IEEE, 67–76.

[78] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree-based KV stores with matrix container in NVM. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. 17–31.

[79] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in LSM-Tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. 12 pages. https://doi.org/10.1145/3267809.3267846

[80] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable parallel flash firmware for many-core architectures. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. Retrieved from https://www.usenix.org/conference/fast20/presentation/zhang-jie.

[81] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE'20)*.

[82] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated compactions for LSM-based key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*.

[83] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. Retrieved from https://www.usenix.org/conference/fast19/presentation/zheng.

[84] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[85] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.* 13, 7 (Mar. 2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

Received July 2021; accepted August 2021